Unidad 5: Tratamiento de datos

Índice de contenidos

1 Edición de la información con herramientas gráficas. 2
1.1 *HeidiSQL*. 2
1.2 *phpMyAdmin*. 3
1.3 *MySQL Workbench*. 4
2 Edición de la información mediante sentencias *SQL*. 5
2.1 Inserción de registros. 6
2.2 Modificación de registros. 7
2.3 Borrado de registros. 8
2.4 Sentencia REPLACE. 8
3 Integridad referencial. 9
3.1 Integridad en actualización y supresión de registros. 9
3.2 Supresión y actualización en cascada. 10
4 Subconsultas y composiciones en órdenes de edición. 10
4.1 Inserción de registros a partir de una consulta. 10
4.2 Modificación de registros a partir de una consulta. 11
4.3 Supresión de registros a partir de una consulta. 11
5 Guiones. 13
6 Transacciones. 13
6.1 Definición. 13
6.2 Propiedades *ACID*. 14
6.3 AUTOCOMMIT. 14
6.4 Instrucciones para el manejo de transacciones. 15
6.5 Problemas asociados al acceso concurrente a los datos. 17
6.6 Niveles de aislamiento. 18
6.6.1 Evaluación de los niveles de aislamiento ante el problema *Dirty Read*. 19
6.6.2 Evaluación de los niveles de aislamiento ante el problema *Non-Repeatable Read*. 20
6.6.3 Evaluación de los niveles de aislamiento ante el problema *Phantom Read*. 21
6.7 Ejemplo de uso de transacciones (REPEATABLE READ). 22
6.8 Transacciones y bloqueos. 24
6.9 Estrategias de bloqueo. 24
6.10 *Deadlock*. 25
6.11 Aspectos a tener en cuenta a la hora de utilizar transacciones. 26
7 Políticas de bloqueo. 27
7.1 Bloqueos compartidos y exclusivos. 27
7.2 Bloqueos automáticos. 28
7.3 Bloqueos manuales. 28
7.3.1 Sentencia SELECT ... FOR UPDATE / LOCK IN SHARE MODE. 28
7.3.2 Sentencias LOCK TABLES y UNLOCK TABLES. 29

Las ***BD* no tienen razón de ser sin la posibilidad de hacer operaciones** de manipulación (añadir, modificar o suprimir) **para el tratamiento de la información almacenada en ellas**.

En esta unidad se va a ver que existen distintos **medios para realizar el tratamiento de los datos**. Desde la **utilización de herramientas gráficas** hasta el **uso de sentencias del lenguaje *SQL*** que permiten realizar ese tipo de operaciones de una forma menos visual, pero con más detalle, flexibilidad y rapidez. El uso de unos mecanismos u otros dependerá de los medios disponibles y de las necesidades como usuarios de la *BD*.

Pero **la información no se puede almacenar en la *BD* sin tener en cuenta que debe seguir una serie de requisitos en las relaciones existentes entre las tablas que la componen**. Todas las operaciones que se realicen respecto al tratamiento de los datos deben asegurar que las relaciones existentes entre ellos se cumplan en todo momento.

Por otro lado, la **ejecución de las aplicaciones puede fallar en un momento dado y eso no debe impedir que la información almacenada sea correcta**. O incluso el mismo **usuario debe tener la posibilidad de cancelar una determinada operación y dicha cancelación no debe suponer un problema para que los datos almacenados se encuentren en un estado fiable**.

Todo esto requiere disponer de una serie de herramientas que aseguren esa **fiabilidad de la información, y que además puede ser consultada y manipulada en sistemas multiusuario sin que las acciones realizadas por un determinado usuario afecten a las operaciones de los demás usuarios**.

1 Edición de la información con herramientas gráficas.

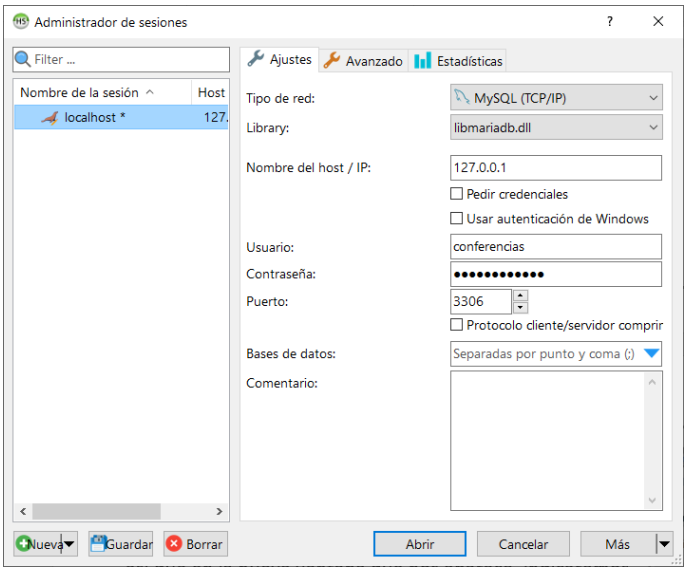
Los *SGBD* como *MariaDB*, ofrecen mecanismos para la **manipulación de la información** contenida en las *BD*. Principalmente se dividen en **herramientas gráficas y herramientas en modo texto** (terminal, consola o línea de comandos).

Para realizar el **tratamiento de los datos por línea de comandos** se **requiere la utilización de un lenguaje de *BD* como *SQL***, lo cual implica el conocimiento de dicho lenguaje.

En cambio, si se dispone de **herramientas gráficas**, no es necesario conocer las sentencias de un lenguaje de ese tipo, y **permiten la inserción, modificación y borrado de datos desde un entorno gráfico con la posibilidad de uso del ratón y una ventana que facilita esas operaciones con un uso similar a las aplicaciones informáticas a las que se está acostumbrado** como usuario.

MariaDB ofrece en su distribución la herramienta gráfica ***HeidiSQL*** (solo en *Windows*), aunque como ya se ha visto en unidades anteriores también se puede utilizar ***phpMyAdmin*** y ***MySQL Workbench*** (existen además muchas **otras aplicaciones comerciales y de uso libre** como ***Sequel Pro*, *DBeaver*, *MyDB Studio*, *SQLyogMySQL GUI*, …**).

1.1 *HeidiSQL.*



Iniciar *HeidiSQL*, se mostrará la ventana del *Administrador de sesiones*. **Introducir el *usuario* y la *clave* de acceso y pulsar sobre el botón *Abrir*** (ver imagen).

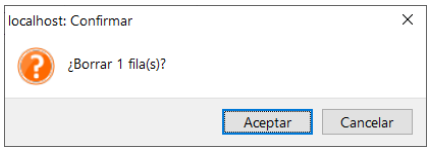
En el **panel lateral** de la izquierda de la nueva ventana, **seleccionar la *BD* y la tabla deseada**. Seguidamente en el panel derecho **seleccionar la pestaña** Datos. Se mostrará la información almacenada en la tabla y **en la barra de herramientas se activarán las opciones de edición** : primero ([*Ctrl+Inicio*]), último ([*Ctrl+Fin*]), insertar fila en la tabla ([*Ins*]), borrar fila/s seleccionada/s (([*Ctrl+Supr*])), enviar ([*Ctrl+Intro*]) y cancelar ([*ESC*]) edición respectivamente.

Para **añadir una nueva fila** hacer **clic sobre el botón** **o pulsar la tecla [*INSERT*]**. Detrás de la fila seleccionada se insertará una nueva en la que se podrá **introducir los datos**. Se puede ir **cambiando entre las distintas columnas utilizando la tecla [*TABULADOR*] o bien haciendo clic sobre la que interese**. **Paga guardar la nueva fila hacer clic**

sobre el botón **o bien utilizar la combinación de teclas [*Ctrl+Intro*]** (si se va pulsando la tecla [*TABULADOR*], al pulsarla desde la última columna la fila se guardará automáticamente). Si se quiere **<u>cancelar la inserción de la nueva fila</u> hacer clic sobre el botón** **o bien pulsar la tecla [*ESC*]**.

Para **<u>editar una fila existente</u>**, hacer **doble clic sobre la columna del registro en concreto a modificar** (**o bien** una vez seleccionada la columna pulsar **[*F2*]**), **realizar los cambios y hacer clic sobre el botón** **para enviar los cambios**. Si finalmente no se desea guardar los cambios hacer clic sobre **para cancelar la edición o bien pulsar la tecla [*ESC*]**.

Para **<u>borrar una fila</u>** hay que **seleccionarla**, haciendo clic sobre cualquiera de sus columnas, y seguidamente **hacer clic sobre el botón** **o bien utilizar la combinación de teclas [*Ctrl+Supr*]**, aparecerá un **cuadro de diálogo** en el que se pregunta si se desea confirmar el borrado de la fila. En caso afirmativo hacer **clic sobre el botón *Aceptar*** y en caso contrario sobre *Cancelar*.

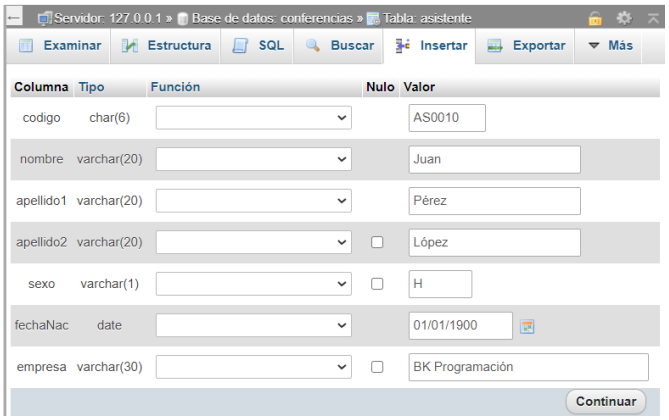


1.2 *phpMyAdmin.*

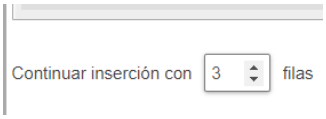
➢ **<u>Añadir registros a una tabla.</u>**

Lo primero que se tiene que hacer es **abrir *phpMyAdmin*** (<u>http://localhost/phpmyadmin/</u>) **en el navegador e identificarse con un usuario y clave válidos**.

Seleccionar la *BD* deseada en el panel lateral izquierdo, **seleccionar la tabla** deseada (Ej.: *asistente* de la *BD conferencias*), pulsar sobre la **pestaña *Insertar***, situada en la zona superior. Se va a añadir una única fila, así que **en la nueva página que se muestra, indicar los valores** y pulsar en ***Continuar***.

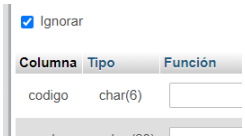


Se puede insertar una única fila rellenando los datos y pulsando sobre el botón *Continuar* de dicha sección, o bien, se puede **indicar un número concreto de filas y rellenar todos los campos de una única vez**. Para indicar el número de registros a insertar situarse en la **zona inferior de *phpMyAdmin* y elegir la cantidad de registros** desde ***Continuar inserción con … filas***.

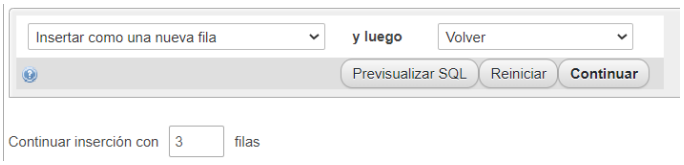


Se actualizará la página con la cantidad de formularios que se acaba de indicar, seguidamente indicar los datos a insertar. Esta opción es muy útil cuando se tengan valores repetitivos y se necesite copiar y pegar los valores.

Tener en cuenta que **existe en cada formulario una casilla** llamada ***Ignorar***. **Cuando el formulario contenga valores, la casilla se desactiva automáticamente, pero en el caso de que el formulario esté vació, *Ignorar* permanecerá activado**. Esto servirá para cuando se hayan creado más formularios de los necesarios, así *phpMyAdmin* solo insertará los campos que contengan algún valor, evitando la inserción de registros vacíos.



Finalmente, se puede **indicar la acción que se realizará después de pulsar el botón de *Continuar*** (para la inserción de los datos). Para ello se tiene el desplegable *"y luego"* en el que se podrá seleccionar ***Volver*** para regresar al listado de registros o bien ***Insertar un nuevo registro***, para permanecer en la página actual y seguir insertando valores.



➢ **<u>Modificar registros de una tabla.</u>**

Una vez insertados los registros si se comprueba que se tiene un error (en algún valor) será necesario realizar la modificación de dicho valor. Para ello *phpMyAdmin* ofrece las herramientas necesarias.

Lo primero será **acceder a la pestaña de visualización de los registros** y para ello, **pulsar sobre la pestaña**

Examinar. Aparece la tabla con los registros que se han ido insertado y se ve que, **para cada fila o registro, se dispone de unos enlaces *Editar***.

Pulsando el **enlace *Editar* de una fila**, se podrá **modificar su contenido y posteriormente guardar los cambios**.

Si se quiere crear una copia de una fila, pulsar sobre el enlace *Copiar* de la fila y se accederá al formulario donde si se quiere, se podrán modificar los valores necesarios. Finalmente hacer clic en ***Continuar*** **para guardar los valores**.

- **Eliminar registros de una tabla.**

  Como es lógico, en el caso de querer eliminar alguna fila, **pulsar sobre el enlace *Borrar* de dicha fila**.

  Se deben **tener en cuenta** dos aspectos fundamentales:
  - ✓ En el **caso de eliminar un registro, se perderá dicha información y no se podrá recuperar**, por lo que se debe tener especial cuidado.
  - ✓ En **campos autonuméricos, al eliminar un registro** y posteriormente crear uno nuevo, su índice continuará con el siguiente número correlativo, **no se podrá recuperar ese valor** (a no ser que se haga manualmente).

- **Operaciones en masa.**

  También se tiene la posibilidad de eliminar o modificar valores en masa, es decir, varios o todos a la vez. Para ello, se deberán **marcar las casillas de las filas que se quieran seleccionar**, **o** bien pulsar sobre ***Seleccionar todo*** para seleccionarlas todas. A continuación, se dispone de los **botones *Editar*, *Copiar*, *Borrar* y *Exportar* para actuar** sobre todos los registros seleccionados.

1.3 *MySQL Workbench.*

Una vez creada la conexión, solo es necesario hacer clic sobre el nombre de la conexión para acceder al servidor de *BD*. **Seleccionar la *BD* deseada en el panel izquierdo**. Para **desplegar la *BD*** pulsar sobre ▶, para **ver las tablas hacer clic nuevamente sobre** ▶ Tables

.

Una vez que se ven las tablas en la parte izquierda de *MySQL Workbench*, **seleccionar la que interese**, y aparecerán tres opciones ( ).

Se tiene la opción de **buscar manualmente en la tabla** haciendo **clic sobre el botón** . De esta forma se abrirá la tabla y se podrá examinar cada una de sus columnas y entradas.

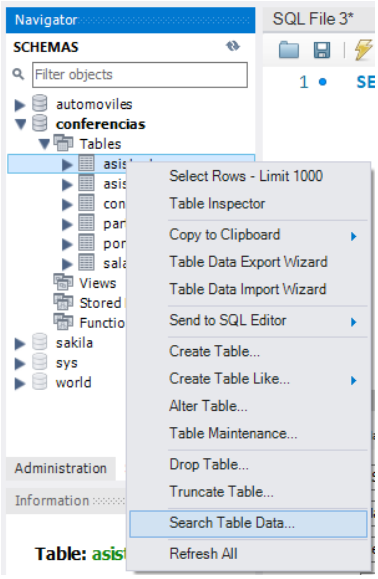
Para visualizar y editar entradas en la parte derecha es necesario **hacer clic en *Form Editor*** (ver imagen).

Utilizando las flechas es posible desplazarse entre las entradas hasta hallar la adecuada y editarla (2 / 14). Los **botones que tienen una tabla con un "+" y un "-"** ( ) sirven para **agregar una entrada a dicha tabla o para eliminar la entrada** que se muestra en ese momento.

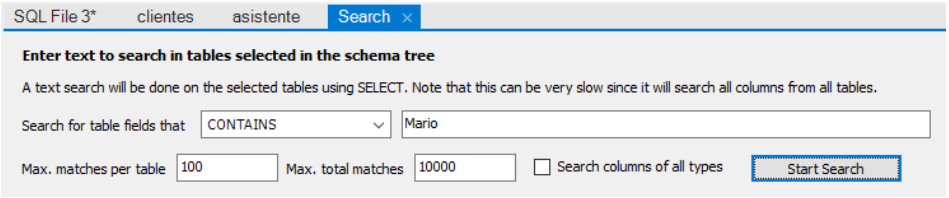
Para poder **guardar los cambios se debe hacer clic en *Apply*** (abajo a la derecha) y guardar los cambios haciendo **clic nuevamente en *Apply*** en la ventana que aparece.

Para **buscar información específica dentro de**

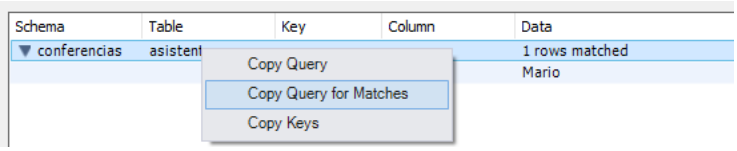
una tabla y posteriormente editarla solo se tiene que hacer **clic con el botón derecho sobre la tabla** que interese y elegir ***Search Table Data…***



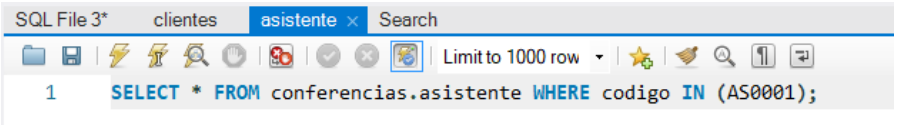
En la caja junto a la palabra ***Contains*** **poner** la frase o **palabra que se busca** y hacer **clic en *Start Search***.



En la lista de resultados hacer **clic derecho sobre la entrada que interese y elegir** *Copy Query for Matches*.



En una **pestaña de *Query* nueva pegar la información que se copió anteriormente y hacer clic en** .



Una vez realizado el proceso **ir a la vista *Form Editor* y editar el contenido de la entrada**. Una vez terminado guardar de la misma forma que se hizo anteriormente.

2 Edición de la información mediante sentencias *SQL*.

El lenguaje ***SQL* dispone de una serie de sentencias para la edición** (*inserción*, *actualización* y *borrado*) **de los datos almacenados en una *BD***. Ese conjunto de sentencias recibe el nombre de ***Data Manipulation Language*** (***DML***).

Antes de comenzar con el estudio de las **sentencias `INSERT`, `UPDATE` y `DELETE`**, hay que **conocer como el *SGBD* gestiona estas instrucciones**, ya que hay **dos posibilidades de funcionamiento**:

- ✓ **Que queden automáticamente validadas y no haya posibilidad de dar marcha atrás**. En este caso, los efectos de toda sentencia de actualización de datos que tenga éxito **son automáticamente accesibles desde el resto de conexiones** de la *BD*.
- ✓ **Que queden en una cola de sentencias y se pueda dar marcha atrás**. En este caso, se dice que las **sentencias de la cola están pendientes de validación**, y el **usuario deberá ejecutar**, cuando crea conveniente, una instrucción para **validarlas (`COMMIT`)** o bien para **deshacerlas (`ROLLBACK`)**.

  Este funcionamiento implica que **los efectos de las sentencias pendientes de validación no se ven por el resto de conexiones de la *BD*, pero sí son accesibles desde la conexión donde se han efectuado**. Al ejecutar `COMMIT`, todas las conexiones acceden a los efectos de las sentencias validadas. En caso de ejecutar `ROLLBACK`, las sentencias desaparecen de la cola y ninguna conexión (ni la propia ni el resto) accede a los efectos correspondientes, es decir, es como si nunca hubieran existido.

Estos **posibles funcionamientos forman parte de la gestión de transacciones que proporciona el *SGBD*** y hay que estudiarlos con más detenimiento. **A la hora de ejecutar sentencias `INSERT`, `UPDATE` y `DELETE` se debe conocer el funcionamiento del *SGBD* para poder actuar en consecuencia**.

Así, por ejemplo, ***MariaDB* funciona con validación automática si no se indica lo contrario** y, en cambio, *Oracle* funciona con cola de sentencias pendientes de confirmación o rechazo.

En ***MariaDB***, si se quiere **desactivar la opción de `AUTOCOMMIT`** (validación automática) que se tiene **por defecto**, habrá que **ejecutar** la siguiente sentencia:

```
SET AUTOCOMMIT = 0;
```

Las sentencias *SQL* que se verán a continuación pueden ser ejecutadas desde cualquiera de los entornos gráficos vistos en el punto anterior o bien desde *MySQL Client* de *MariaDB* (línea de comandos).

***MariaDB* puede operar en diferentes modos** *SQL* dependiendo del valor de la **variable del sistema `sql_mode`**. El **modo estricto controla los valores no válidos o faltantes** en las sentencias `INSERT` o `UPDATE`.

Si el **modo estricto no está activo**, ***MariaDB*** (predeterminado en versiones <= *MariaDB 10.2.3*) **inserta valores ajustados para valores no válidos o faltantes** (truncando cadenas que son demasiado largas o ajustando valores numéricos que están fuera de rango) **y produce advertencias** (el modo estricto se puede saltar utilizando `INSERT IGNORE` o `UPDATE IGNORE`).

Con el **modo estricto** establecido (predeterminado en las versiones >= *MariaDB 10.2.4*), las **sentencias que modifican información de las tablas** (ya sea transaccional `STRICT_TRANS_TABLES` o para todas `STRICT_ALL_TABLES`) **fallarán y devolverán un error**. Se puede consultar la variable *sql_mode* utilizando: `SELECT @@sql_mode;`

Para **establecer el modo estricto** ejecutar (si no se indica `GLOBAL` se establece para la sesión del cliente solo):

```
SET [GLOBAL] @@sql_mode=CONCAT(@@sql_mode,',STRICT_TRANS_TABLES');
```

Si se utiliza *XAMPP* se puede editar el archivo `my.ini` ubicado en `C:\xampp\mysql\bin` y en la línea `sql_mode` bajo `[mysqld]` añadir "`STRICT_TRANS_TABLES`" para que el cambio realizado sea permanente.

Más información en: https://mariadb.com/kb/en/sql-mode/

2.1 Inserción de registros.

La sentencia **`INSERT` permite la inserción de nuevas filas o registros en una tabla existente**. El formato más sencillo de utilización de la sentencia `INSERT` tiene la siguiente **sintaxis**:

```
INSERT [INTO] nombre_tabla[(lista_campos)]
VALUES (lista_valores) [, (lista_valores), ...];
```

Donde **`nombre_tabla`** es el **nombre de la tabla en la que se quiere añadir los nuevos registros**. En **`lista_campos`** se indicarán los **campos de dicha tabla en los que se desea escribir los nuevos valores indicados en `lista_valores`**. **Es posible omitir la lista de campos (`lista_campos`), si se indican todos los valores de cada campo y en el orden en el que se encuentran en la tabla**.

Cada uno de los elementos de la **lista de campos** (`lista_campos`) **y** de **la de valores** (`lista_valores`) estarán **separados por comas**. Hay que tener en cuenta también que **cada campo de `lista_campos` debe tener un valor válido en la posición correspondiente de la `lista_valores`** (si no se recuerdan los tipos de datos de cada campo se puede utilizar la sentencia `DESCRIBE` o `DESC` seguida del nombre de la tabla a consultar).

En el siguiente ejemplo se inserta un nuevo registro en la tabla *asistente* en el que se tienen todos los datos disponibles:

```
INSERT INTO asistente(codigo, nombre, apellido1, apellido2, sexo, fechaNac, empresa)
VALUES('AS0015', 'Julia', 'Hernández', 'Sáenz', 'M', '1992/11/10', 'BK Programación');
```

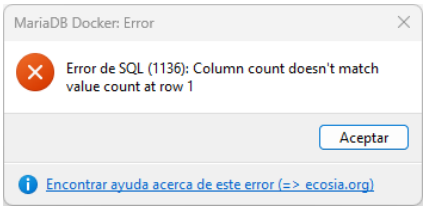
En este otro ejemplo, se inserta un registro de igual forma, pero sin indicar todos los datos:

```
INSERT INTO asistente(codigo, nombre, apellido1, sexo, fechaNac)
VALUES('AS0016', 'María', 'Durán', 'M', '1979/03/30');
```

Al hacer un `INSERT` en el que no se especifiquen los valores de todos los campos, se obtendrá el valor `NULL` en aquellos campos que no se han indicado (si no se han declarado como `NOT NULL`).

Si la lista de campos indicados no se corresponde con la lista de valores, se obtendrá un error en la ejecución. Por ejemplo, si no se indica el campo *empresa*, pero sí se especifica un valor para dicho campo:

```
INSERT INTO asistente(codigo, nombre, apellido1, sexo, fechaNac)
VALUES('AS0017', 'Lucas', 'López', 'H', '1989/11/22', 'Big Soft');
```



En *HeidiSQL* se obtiene el error que se muestra en la imagen.

Tener en cuenta que en *HeidiSQL* y otros entornos gráficos es necesario recargar la tabla para que las filas insertadas se muestren.

Más información sobre `INSERT` *en* https://mariadb.com/kb/en/insert/

Consideraciones a tener en cuenta:

- ✓ Al insertar datos en una tabla donde hay **campos que permiten valores `NULL` o tienen un valor por defecto** definido, **no se necesitan proporcionar valores** para esos campos en la sentencia `INSERT`, y el sistema manejará automáticamente los valores `NULL` o utilizará los valores por defecto especificados. Si se desea insertar valores nulos, se deberá proporcionar explícitamente la palabra clave `NULL` para esos campos.
- ✓ Al trabajar con **campos `ENUM` o `SET`**, se deben **proporcionar valores válidos que estén definidos en la declaración de la tabla**. Si se intenta insertar un valor que no está permitido, se generará un error. Recordar que en un campo `ENUM` solo se puede seleccionar un valor de una lista de opciones predefinidas, mientras que en un campo **`SET` se pueden seleccionar múltiples valores** de la lista predefinida.
- ✓ Cuando se tiene una **columna que es una clave foránea**, es necesario **asegurarse de que el valor que se está insertando en esa columna corresponda a una fila existente en la tabla referenciada**.
- ✓ Si se tienen **restricciones de unicidad** en la tabla (clave primaria o un índice único), es necesario **asegurarse de que los valores que se insertan no violen esas restricciones**. De lo contrario, la inserción fallará.
- ✓ Asegurarse de que los **valores que se insertan coincidan con los tipos de datos esperados por las columnas correspondientes** en la tabla. Si se intenta insertar un valor de un tipo de dato incorrecto, la inserción fallará.
- ✓ Si se están realizando **múltiples inserciones**, **considerar envolverlas en una transacción**. Esto garantiza que todas las inserciones se completen correctamente o ninguna se complete si ocurre un error.
- ✓ Para mejorar el rendimiento al insertar grandes cantidades de datos, **utilizar inserciones múltiples en lugar de inserciones individuales** (se pueden insertar varias filas en una sola sentencia `INSERT`).
- ✓ Asegurarse de que el **usuario que ejecuta la sentencia `INSERT` tenga los privilegios necesarios** para realizar la inserción en la tabla específica.
- ✓ Cuando se tiene un **campo `AUTO_INCREMENT`, no es necesario proporcionar un valor para ese campo al insertar datos**, ya que **se generará automáticamente** por el *SGBD*.

  Si se necesita **obtener el valor del id que se ha generado después de la inserción**, se puede usar la función **`LAST_INSERT_ID()`** (Esto es útil si se necesita utilizar ese valor después de la inserción).

  En una tabla con un campo `AUTO_INCREMENT`, se puede **insertar valores** en ese campo de varias **formas**:
  - **Dejar el campo sin especificar**: si se omite el campo `AUTO_INCREMENT` en la lista de columnas al insertar un nuevo registro, el ***SGBD* generará automáticamente un valor** para ese campo.
  - **Insertar `NULL`**: también se puede insertar `NULL` en el campo `AUTO_INCREMENT` **y el *SGBD* generará automáticamente un valor** para ese campo.
  - **Insertar `0`**: si se intenta insertar explícitamente un valor de `0` en un campo `AUTO_INCREMENT`, **el *SGBD* ignorará ese valor y generará automáticamente el siguiente valor**.
  - **Insertar un valor concreto**: se puede insertar un valor específico, aunque se debe **tener cuidado**, ya que, **si el valor especificado ya existe en la tabla o si viola alguna restricción única, se generará un error**.

  Si se inserta un valor específico en un campo `AUTO_INCREMENT` mayor al que corresponde, el sistema continuará generando valores incrementales a partir del número que se haya insertado (valor `AUTO_INCREMENT` se ajustará para que el próximo valor generado sea mayor que el valor que se proporcionó).

2.2 Modificación de registros.

La sentencia UPDATE permite modificar una serie de valores de determinados registros de las tablas de la *BD*. La forma más sencilla de utilizar la sentencia UPDATE tiene la siguiente **sintaxis**:

```
UPDATE nombre_tabla
SET nombre_campo1=valor1 [, nombre_campo2=valor2, ...]
[WHERE condición(es)];
```

Donde **`nombre_tabla`** será el **nombre de la tabla en la que se quieran modificar los datos**. Se pueden especificar los nombres de campos que se deseen de la tabla indicada, **a cada campo especificado se le debe asociar el nuevo valor utilizando el signo =**. **Cada emparejamiento `campo=valor` debe separarse del siguiente utilizando comas (,)**.

La **cláusula `WHERE`** seguida de la condición **es opcional**. **Si se indica, la actualización de los datos sólo afectará a los registros que cumplen la condición**. Por tanto, hay que tener en cuenta que, **si no se indica la cláusula `WHERE`, los cambios afectarán a todos los registros**.

Por ejemplo, si se desea poner a 20 el *precio* de cada *conferencia* (todas las conferencias):

```
UPDATE conferencia SET precio=20;
```

En este otro ejemplo se puede ver la actualización de dos campos, poniendo a 20 el *precio* y borrando la información del campo *tema* de todas las conferencias:

```
UPDATE conferencia SET precio=20, tema=NULL;
```

Para que los **cambios afecten a determinados registros y no a todos los de la tabla hay que especificar una condición**. Por ejemplo, si se quiere cambiar el *precio* de todas las conferencias del *turno* de tarde a 30:

```
UPDATE conferencia SET precio=30 WHERE turno='T';
```

Cuando termina la ejecución de una sentencia UPDATE, se muestra la cantidad de registros (filas) que han sido actualizados, o el error correspondiente si se ha producido algún problema.

Más información sobre UPDATE *en* https://mariadb.com/kb/en/update/

Consideraciones a tener en cuenta:

- ✓ **Restricciones de unicidad**: si se realiza una **actualización que podría violar las restricciones de unicidad** en la tabla (clave primaria o índice único), asegurarse de que las nuevas valores no generen conflictos.
- ✓ **Validación de tipos de datos**: asegurarse de que los **nuevos valores** que se están utilizando en la actualización **coincidan con los tipos de datos esperados por las columnas** correspondientes en la tabla.
- ✓ **Condiciones de actualización**: especificar las condiciones adecuadas en la sentencia UPDATE para que **solo se actualicen las filas deseadas**.
- ✓ **Privilegios de usuario**: asegurarse de que el **usuario que ejecuta la sentencia UPDATE tenga los privilegios necesarios** para actualizar las columnas y filas específicas en la tabla.
- ✓ **Cascada de claves foráneas**: si la **clave primaria que se está actualizando es referenciada por claves foráneas en otras tablas**, se debe tener en cuenta el efecto de cascada de las actualizaciones.

2.3 Borrado de registros.

La **sentencia DELETE** es la que **permite eliminar o borrar registros de una tabla**. Esta es la **sintaxis** que se debe tener en cuenta para utilizarla:

```
DELETE FROM nombre_tabla [WHERE condición(es)];
```

Al igual que se ha visto en las sentencias anteriores, **nombre_tabla** hace referencia a la **tabla sobre la que se hará la operación**, en este caso de borrado. Se puede observar que la **cláusula WHERE** es **opcional**, **si no se indica**, se debe tener muy claro que **se borrará todo el contenido de la tabla**, aunque **la tabla seguirá existiendo**. Por ejemplo, si se usa la siguiente sentencia, se borrarán todos los registros de la tabla *ponente*:

```
DELETE FROM ponente;
```

Para ver un ejemplo de uso de la sentencia DELETE en la que se indique una condición, suponer que se quieren eliminar todos los ponentes cuya *especialidad* es '*Programación*':

```
DELETE FROM ponente WHERE especialidad='Programación';
```

Como resultado de la ejecución de este tipo de sentencia, se obtendrá un mensaje de error si se ha producido algún problema, o bien, el número de filas que se han eliminado.

Consideración: al eliminar una fila, las **referencias de clave foránea en otras tablas pueden requerir una acción de cascada** para mantener la integridad referencial. Se debe **considerar si se desea que las eliminaciones se propaguen automáticamente** a las tablas relacionadas (CASCADE) **o si se prefiere manejar las eliminaciones manualmente**.

Más información sobre DELETE *en* https://mariadb.com/kb/en/delete/

2.4 Sentencia **REPLACE**.

***MariaDB* tiene una extensión del lenguaje *SQL* que permite insertar una nueva fila que, en caso de que la clave**

primaria coincida con otra fila, previamente sea eliminada. Se trata de la sentencia REPLACE.

Hay **tres posibles sintaxis** para la sentencia REPLACE:

```
a) REPLACE [INTO] nombre_tabla [(columna1, ...)]
        VALUES ({expr | DEFAULT}, ...) [, (...), ...];
b) REPLACE [INTO] nombre_tabla SET columna1 = {expr | DEFAULT}, ...;
c) REPLACE [INTO] nombre_tabla [(columna1, ...)] SELECT ...;
```

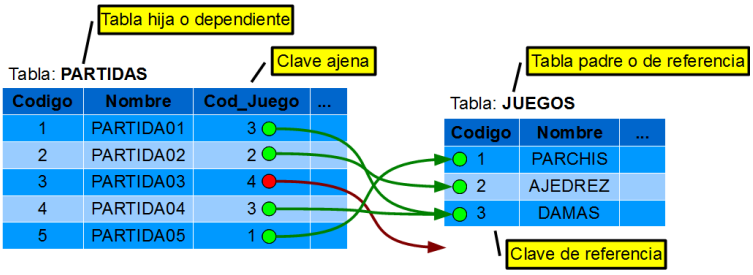
Más información sobre REPLACE *en* https://mariadb.com/kb/en/replace/

3 Integridad referencial.

Dos tablas se pueden relacionar si tienen en común uno o más campos. La restricción de integridad referencial requiere que haya coincidencia en todos los valores que deben tener en común ambas tablas. Cada valor del campo que forma parte de la integridad referencial definida, debe corresponderse, en la otra tabla, con otro registro que contenga el mismo valor en el campo referenciado.

Por ejemplo, suponer que se tienen las tablas *PARTIDAS* y *JUEGOS*. En la tabla *PARTIDAS* existe un campo que referencia al juego al que corresponde, mediante su código de juego. Por tanto, no puede existir ninguna partida cuyo código de juego no se corresponda con ninguno de los juegos de la tabla *JUEGOS*.

En el ejemplo, no se cumple la integridad referencial, ya que la partida *PARTIDA03* corresponde al juego cuyo código es 4, y en la tabla *JUEGOS* no existe ningún registro con ese código. Para que se cumpla la integridad referencial, todos los valores del campo *Cod_Juego* deben corresponderse con valores existentes en el campo *Codigo* de la tabla *JUEGOS*.



Cuando se habla de **integridad referencial** se utilizan los siguientes **términos**:

- ✓ **Clave ajena**: es el **campo o conjunto de campos incluidos en la definición de la restricción que deben hacer referencia a una clave de referencia**. En el ejemplo, la clave ajena sería *Cod_Juego* de la tabla *PARTIDAS*.
- ✓ **Clave de referencia**: **clave única o primaria de la tabla a la que se hace referencia desde una clave ajena**. En el ejemplo, la clave de referencia es el campo *Codigo* de la tabla *JUEGOS*.
- ✓ **Tabla hija o dependiente**: **tabla que incluye la clave ajena, y que, por tanto, depende de los valores existentes en la clave de referencia**. Tabla *PARTIDAS* del ejemplo, que sería la tabla hija de la tabla *JUEGOS*.
- ✓ **Tabla padre o de referencia**: corresponde a la **tabla que es referenciada por la clave ajena en la tabla hija**. Esta tabla **determina las inserciones o actualizaciones que son permitidas en la tabla hija**, en función de dicha clave. En el ejemplo, la tabla *JUEGOS* es padre de la tabla *PARTIDAS*.

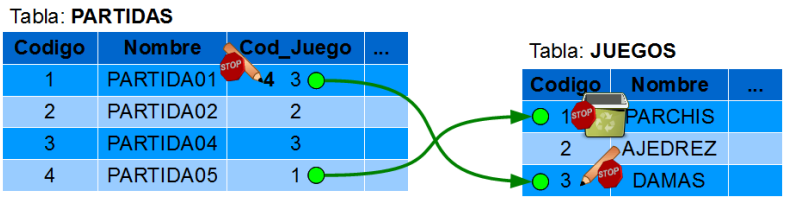
3.1 Integridad en actualización y supresión de registros.

La **relación existente entre la clave ajena y la primaria tiene implicaciones en el borrado y modificación** de sus valores. **Si se modifica el valor de la clave ajena en la tabla hija, debe establecerse un nuevo valor que haga referencia a la clave principal de uno de los registros de la tabla padre**. De la misma manera, **no se puede modificar el valor de la clave principal en un registro de la tabla padre si una clave ajena hace referencia a dicho registro**.

Los borrados de registros en la tabla de referencia también pueden suponer un problema, ya que **no pueden suprimirse registros que son referenciados por una clave ajena desde otra tabla**.

Ejemplos de situaciones problemáticas:

- ✓ En el registro de la partida con nombre *PARTIDA01* no puede ser modificado el campo *Cod_Juego* al valor 4, porque no es una clave ajena válida, puesto que no existe



un registro en la tabla *JUEGOS* con esa clave primaria.

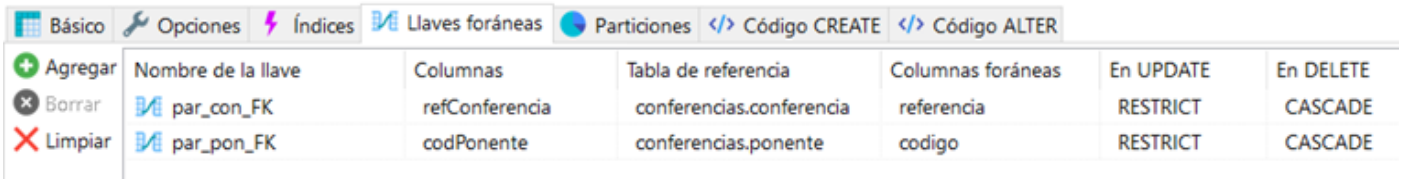
- ✓ El código del juego *DAMAS* no puede ser cambiado, ya que hay registros en la tabla *PARTIDAS* que hacen referencia a dicho juego a través del campo *Cod_Juego*.
- ✓ Si se eliminara en la tabla *JUEGOS* el registro que contiene el juego *PARCHIS*, la partida *PARTIDA05* quedaría con un valor inválido en el campo *Cod_Juego*.

Cuando se hace el borrado/actualización de registros en una tabla de referencia, se puede configurar la clave ajena de diversas maneras para que se conserve la integridad referencial:

- ✓ **No permitir** (`RESTRICT`): es la **opción por defecto**. En **caso de que se intente borrar/actualizar en la tabla de referencia un registro que está siendo referenciado desde otra tabla, se produce un error en la operación de borrado/actualización impidiendo dicha acción**.
- ✓ **En cascada** (`CASCADE`): **al eliminar/actualizar registros de la tabla de referencia, los registros de la tabla hija que hacían referencia a dichos registros, también son eliminados/actualizados**.
- ✓ **Definir nulo** (`SET NULL`): **los valores de la clave ajena que hacían referencia a los registros que hayan sido eliminados/actualizados de la tabla de referencia, son cambiados al valor** `NULL`.

3.2 Supresión y actualización en cascada.

Las opciones de *supresión o actualización en cascada* o *definir nulo al suprimir o actualizar* pueden establecerse desde el momento de creación de las tablas, al establecer las claves ajenas.



Nombre de la llave	Columnas	Tabla de referencia	Columnas foráneas	En UPDATE	En DELETE
par_con_FK	refConferencia	conferencias.conferencia	referencia	RESTRICT	CASCADE
par_pon_FK	codPonente	conferencias.ponente	codigo	RESTRICT	CASCADE

Si la tabla ya estaba creada, y posteriormente se desea establecer una restricción de clave ajena con opción de supresión o actualización en cascada, se puede establecer desde la pestaña *Llaves foráneas* (en *HeidiSQL*), seleccionando la tabla que contiene el campo con la clave ajena.

Si estas operaciones se quieren realizar con código *SQL*, se dispone de las siguientes opciones durante la declaración de la clave ajena de la tabla: utilizar la opción `ON DELETE/UPDATE CASCADE` para hacer la supresión/actualización en cascada, o bien `ON DELETE/UPDATE SET NULL` si se prefiere definir nulo al suprimir/actualizar. Por ejemplo:

```
… FOREIGN KEY (Cod_Juego) REFERENCES juego(Codigo) ON DELETE CASCADE ON UPDATE CASCADE …
```

Hay que recordar que **una declaración de este tipo debe hacerse en el momento de crear la tabla (`CREATE TABLE`) o posteriormente modificando su estructura (`ALTER TABLE`)**.

> *Más información sobre las claves foráneas en https://mariadb.com/kb/en/foreign-keys/*

4 Subconsultas y composiciones en órdenes de edición.

En los puntos anteriores se ha visto una serie de sentencias del lenguaje *SQL* que han servido para realizar operaciones de inserción (`INSERT`), modificación (`UPDATE`) y borrado (`DELETE`) de registros. Pero, los **valores que se añadan o se modifiquen también podrán ser obtenidos también como resultado de una consulta**.

Además, las **condiciones que se han podido añadir** hasta ahora **a las sentencias, pueden ser también consultas**, por lo que pueden establecerse condiciones bastante más complejas.

4.1 Inserción de registros a partir de una consulta.

Se ha visto la posibilidad de insertar registros en una tabla a través de la sentencia `INSERT`, por ejemplo:

```
INSERT INTO ASISTENTE (codigo, nombre, apellido1, apellido2, sexo, fechaNac, empresa)
  VALUES ('AS0015', 'Julia', 'Hernández', 'Sáez', 'M', '1992/11/10', 'BK Programación');
```

Es posible insertar en una tabla valores que se obtienen directamente del resultado de una consulta. Suponer, por ejemplo, que se dispone de una tabla *ASISTENTES_SUPLENTES* con la misma estructura que la tabla *ASISTENTE*. Si se quiere insertar en esa tabla todos los usuarios que tienen el campo empresa a `NULL`:

```
INSERT INTO asistentes_suplentes SELECT * FROM asistente WHERE empresa IS NULL;
```

Observar que **en este caso no se debe especificar la palabra `VALUES`, ya que no se está especificando una lista de valores**.

4.2 Modificación de registros a partir de una consulta.

La acción de actualizar registros mediante la sentencia `UPDATE` también puede ser utilizada con consultas para realizar modificaciones más complejas de los datos. **Las consultas pueden formar parte de cualquiera de los elementos de la sentencia `UPDATE`** (`SET` y `WHERE`).

Por ejemplo, la siguiente sentencia modifica el precio de las conferencias que se celebren en las salas cuya capacidad sea mayor que 200, el precio que se les asignará será el de la conferencia que tenga mayor precio:

```
UPDATE conferencia SET precio = (SELECT MAX(precio) FROM conferencia)
WHERE sala IN (SELECT nombre FROM sala WHERE capacidad>200);
```

4.3 Supresión de registros a partir de una consulta.

Al igual que en las sentencias `INSERT` y `UPDATE`, también **se pueden hacer borrados de registros (`DELETE`) utilizando consultas como parte de la condición que delimita la operación**. Ej.:

```
DELETE FROM conferencia WHERE sala IN (SELECT nombre FROM sala WHERE capacidad>200);
```

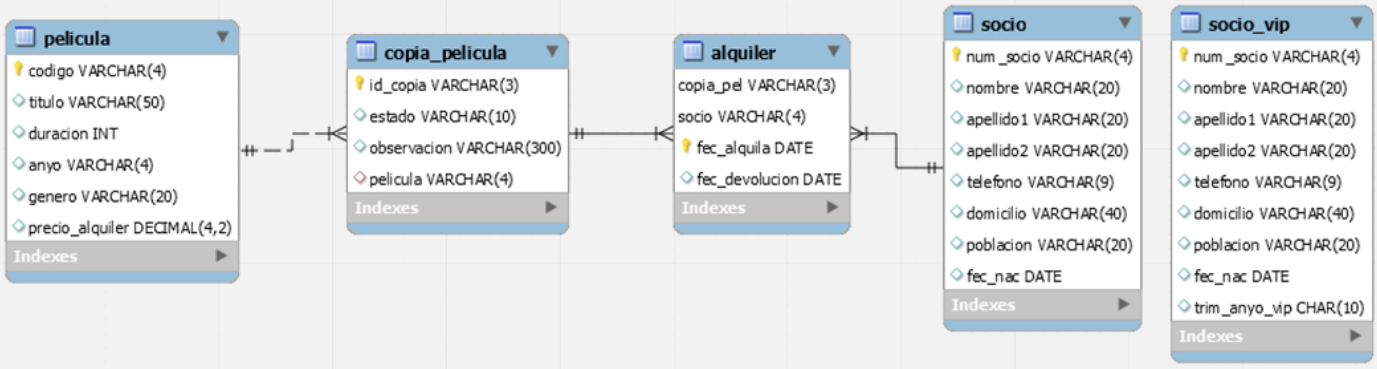
TAREA

Utilizando sentencias *SQL*, realizar las siguientes operaciones sobre la *BD* "tienda" (se dispone del script de creación de la *BD* en la unidad 4 o en la misma unidad 5):

1. **Insertar un nuevo fabricante (por ejemplo, *Apple*) indicando su *id* y su *nombre*.**
2. **Insertar dos nuevos fabricantes (por ejemplo, *MSI* y *TP-Link*) indicando solamente su *nombre* haciendo uso de una única sentencia *SQL*.**
3. **Insertar un nuevo producto (ver algún producto del fabricante en cuestión) asociado a uno de los nuevos fabricantes. La sentencia de inserción debe incluir: *id*, *nombre*, *precio* e *id_fabricante*.**
4. **Insertar dos nuevos productos (ver productos del fabricante en cuestión) asociado a uno de los nuevos fabricantes. La sentencia de inserción debe ser única e incluir: *nombre*, *precio* e *id_fabricante*.**
5. **Crear una nueva tabla con el nombre *fabricante_productos* que tenga las siguientes columnas: *nombre_fabricante*, *nombre_producto* y *precio*. Una vez creada la tabla insertar todos los registros de la *BD* "*tienda*" en esta tabla haciendo uso de una única operación de inserción.**
6. **Crear una vista con el nombre *vista_fabricante_productos* que tenga las siguientes columnas: *nombre_fabricante*, *nombre_producto* y *precio*.**
7. **Eliminar el fabricante *Xiaomi*. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios se deberían realizar para que fuese posible borrarlo?**
8. **Eliminar el fabricante *Asus*. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios se deberían realizar para que fuese posible borrarlo?**
9. **Actualizar el código del fabricante *Huawei* y asignarle el valor 30. ¿Es posible actualizarlo? Si no fuese posible, ¿qué cambios se deberían realizar para que fuese posible actualizarlo?**
10. **Actualizar el código del fabricante *Lenovo* y asignarle el valor 20. ¿Es posible actualizarlo? Si no fuese posible, ¿qué cambios se deberían realizar para que fuese posible actualizarlo?**
11. **Actualizar el precio de todos los productos sumándole 5 € al precio actual.**

12. **Eliminar todas las impresoras que tienen un precio menor de 200 €.**
13. **Incrementar en un 5% el precio de los productos que pertenecen a los fabricantes en los que el nombre en su segundo carácter tiene una "e".**
14. **(*) Añadir un "*" al final del nombre del fabricante para aquellos en los que el precio medio de sus productos sea superior a 200.**
15. **Eliminar todos los fabricantes que no tienen productos asignados.**

Utilizando el script *SQL* proporcionado en la unidad (*BD-U5.- Script tarea.sql*):



NOTA: el último campo *trim_anyo_vip* almacenará valores como *TRIM1_2014*, *TRIM2_2014*, *TRIM3_2014*, *TRIM4_2014*, etc. con el objetivo de saber qué socio/s han conseguido la condición de *VIP* en cada uno de los trimestres de cada año.

Para la realización de esta tarea es necesario descargar y ejecutar el script "*BD-U5.- Script tarea.sql*", el cual crea las tablas e inserta los datos necesarios.

Utilizando sentencias *SQL* y alguna de las interfaces gráficas vistas, realizar las siguientes operaciones adjuntando capturas de pantalla:

16. **Insertar un nuevo socio con los siguientes datos:**
    Núm. Socio: *1007*
    Nombre: *Francisco*
    Apellido 1: *Sánchez*
    Domicilio: *Avda. de las Palmeras, 2*
    Población: *Armilla*
    Fecha de nacimiento: *02/02/1970*

    NOTA: el resto de campos no deben ser introducidos.
17. **Registrar el primer alquiler del socio 1007 que se lleva la copia 105 (correspondiente a la película "*Lo imposible*") en la fecha actual (en la que se está haciendo la tarea).**

    NOTA: la fecha de devolución no se debe introducir aún.
18. **Modificar el estado de la copia de la película con *id* 101 a "*ESTROPEADA*" incluyendo como observación "*Rayado*".**
19. **Eliminar la película cuyo título es "*El Orfanato*".**

Utilizando sentencias *SQL*, realizar las siguientes operaciones:

20. **Insertar una nueva película (inventando los datos) y después dar de alta dos copias para dicha película con el estado "*FUNCIONA*".**
21. **Actualizar todas las películas que tengan como género "*Animación*" y reemplazarlo por "*Dibujos*".**
22. **Eliminar aquellos socios cuya última película alquilada sea anterior al 1 de diciembre de 2014.**
23. **Incrementar en 20 céntimos el precio del alquiler a todas las películas que tengan más de dos copias.**
24. **Eliminar todas las copias de las películas que contengan la palabra "*FROZEN*" y que su estado sea "*ESTROPEADA*".**
25. **(** Opcional) Actualizar el precio de alquiler de aquellas películas cuyo número total de alquileres (de todas sus copias) sea inferior a la media de los alquileres de todas las películas (no tener en cuenta las películas sin copias, lo que se alquilan son las copias). El precio debe reducirse en un 50% de su precio original.**
26. **(** Opcional) Insertar en la tabla *socio_vip* todos los datos del socio que más número de alquileres tenga en el último trimestre del año 2014 insertando en el campo *trim_anyo_vip* el valor "*TRIM4_2014*".**

5 Guiones.

Los ***SGDB* desarrollaron un conjunto de técnicas con el motivo de ampliar la capacidad del lenguaje *SQL* y aumentar la eficiencia en el desarrollo de aplicaciones sobre *BD***. Estas técnicas son:

- ✓ **Guiones.**
- ✓ **Disparadores o *triggers*.**
- ✓ **Eventos.**
- ✓ **Excepciones.**
- ✓ **Procedimientos.**
- ✓ **Funciones.**

Los guiones (**o scripts *SQL***) son **conjuntos de instrucciones que se ejecutan en una *BD* para realizar tareas específicas**. Estas tareas pueden incluir la creación de tablas, la inserción de datos, la actualización y eliminación de registros. En esencia, los guiones *SQL* son programas relativamente simples diseñados para interactuar con *BD*.

Los guiones también son conocidos como *scripts*, **se guardan en archivos de texto plano con la extensión `.sql`** donde se pueden especificar las sentencias que serán ejecutas por el *SGBD*. Su principal **ventaja** recae en su **facilidad de uso ya sea desde consola o mediante la interfaz gráfica** de las aplicaciones cliente que se conectan al *SGBD*. Ejemplos: *scripts* de cada una de las unidades.

A continuación, se muestra un ejemplo de guion, donde se selecciona la *BD* a usar, se crea una vista que listará los 10 primeros comentarios que se encuentren en la tabla `comentarios` de la *BD* `wordpress` y para finalizar dará permiso de consulta (`SELECT`) al usuario `consultor` sobre la vista:

```
USE wordpress;
CREATE OR REPLACE VIEW comentarios_p10 AS
      SELECT id, nombre, fecha, comentario FROM comentarios LIMIT 10;
GRANT SELECT ON comentarios_p10 TO consultor;
```

Una vez guardado el fichero con la extensión `.sql` puede ser ejecutado en cualquier momento. Destacar la cláusula `REPLACE VIEW` que permite reemplazar la vista si ya existe una con el mismo nombre.

El resto de técnicas se verán en la próxima unidad dedicada a la programación de *BD*.

6 Transacciones.

6.1 Definición.

Una transacción es un **grupo secuencial de sentencias *SQL*** (seleccionar, insertar, actualizar o eliminar) **que se realizan como una sola unidad de trabajo**, es decir, **de forma indivisible o atómica**.

En otras palabras, **una transacción nunca se completará a menos que cada operación individual que contenga tenga éxito**. Si falla alguna operación dentro de la transacción, la transacción completa fallará.

> ***Transacción en SQL**: conjunto de sentencias SQL, agrupadas lógicamente, que o bien se ejecutan todas sobre la BD o bien no se ejecuta ninguna.*

Un traspaso de dinero entre dos cuentas bancarias es un buen ejemplo para explicar su funcionamiento. Para lograrlo se tienen que escribir varias sentencias *SQL* que hagan lo siguiente:

1. Verificar la disponibilidad de la cantidad solicitada en la primera cuenta.
2. Deducir la cantidad solicitada de la primera cuenta.
3. Depositar la cantidad en la segunda cuenta.

Notar que, si algo falla en este proceso, todo debe revertirse a su estado anterior (inicial de partida).

Una **transacción *SQL* finaliza** con un **`COMMIT`**, para **aceptar todos los cambios** que la transacción ha realizado en la *BD*, o un **`ROLLBACK`** para **deshacerlos**.

Por defecto, *MariaDB* utiliza el motor de almacenamiento *InnoDB*, que **es un sistema transaccional**, es decir, que soporta las características que hacen que una *BD* pueda garantizar que los datos se mantengan consistentes (*MyISAM* no permite el uso de transacciones).

El **uso de transacciones permite realizar operaciones de forma segura y recuperar datos si se produce algún fallo en el servidor durante la transacción**, pero por otro lado las transacciones **pueden aumentar el tiempo de ejecución** de las instrucciones.

Las **transacciones deben cumplir** las cuatro **propiedades *ACID***.

6.2 Propiedades *ACID*.

Las **propiedades que garantizan los sistemas transaccionales** son las características llamadas ***ACID*** (acrónimo inglés de ***Atomicity***, ***Consistency***, ***Isolation*** y ***Durability***). Estas propiedades garantizan que las transacciones se puedan realizar en una *BD* de forma segura.

Se dice que **un *SGBD* es "*ACID compliant*" cuando cumple** con las siguientes propiedades:

- ✓ **Atomicidad (*Atomicity*)**: se dice que un *SGBD* garantiza la atomicidad **si cualquier transacción o bien finaliza correctamente, o bien no deja ningún rastro de su ejecución**. Esta propiedad quiere decir que una transacción es indivisible, o se ejecutan todas las sentencias o no se ejecuta ninguna.
- ✓ **Consistencia (*Consistency*)**: se habla de consistencia cuando **la concurrencia de diferentes transacciones no puede producir resultados anómalos**. Esta propiedad asegura que después de una transacción la *BD* estará en un estado válido y consistente (una transacción lleva la *BD* de un estado consistente a otro consistente).
- ✓ **Aislamiento (*Isolation*)**: **cada transacción dentro del sistema se debe ejecutar como si fuera la única que se ejecuta en ese momento** (garantiza que cada transacción se ejecuta de manera independiente). Cuando muchas transacciones se pueden llevar a cabo simultáneamente por uno o varios usuarios, la realización de una no debe afectar jamás a las otras y por tanto no llevar a una situación de error.
- ✓ **Durabilidad (*Durability*)**: **si se confirma una transacción, el resultado de esta debe ser definitivo y no se puede perder** (permanente). Una vez que la transacción ha sido confirmada (COMMIT) y por tanto los cambios en la *BD* guardados, éstos deben perdurar en el tiempo, aunque el *SGBD* o el propio equipo fallen.

El **sistema de almacenamiento *InnoDB* es el que hay que utilizar para trabajar con transacciones**, pero **puede haber casos en que sea interesante considerar otros tipos de motores de almacenamiento** (ya comentados en la unidad 3). Por ello, ***MariaDB* también ofrece** otros sistemas que no admiten transacciones como, por ejemplo, *MyISAM*, *Memory*, *Merge*, …

Para **obtener una lista de los motores de almacenamiento soportados** por la versión de *MariaDB* que se esté usando ejecutar la orden **`SHOW ENGINES;`**

Engine	Support	Comment	Transactions	XA	Savepoints
CSV	YES	Stores tables as CSV files	NO	NO	NO
MRG_MyISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary...	NO	NO	NO
Aria	YES	Crash-safe tables with MyISAM heritage. Used for i...	NO	NO	NO
MyISAM	YES	Non-transactional engine with good performance and...	NO	NO	NO
SEQUENCE	YES	Generated tables filled with sequential values	YES	NO	YES
InnoDB	DEFAULT	Supports transactions, row-level locking, foreign ...	YES	YES	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO

Más información en https://mariadb.com/kb/en/choosing-the-right-storage-engine/

6.3 AUTOCOMMIT.

Algunos *SGBD* como ***MariaDB***/*MySQL* (**si se trabaja con el motor *InnoDB***), *SQL Server*, *PostgreSQL*, … **funcionan por defecto en modo AUTOCOMMIT**. Esto significa que **la ejecución de cada sentencia *SQL* se confirma automáticamente** y que los efectos/**cambios de la sentencia no podrán ser deshechos** (transacciones implícitas).

Aunque la variable AUTOCOMMIT está activada por defecto al inicio de una sesión *SQL*, se puede configurar para indicar si se quiere trabajar con transacciones implícitas o explícitas.

Se puede **consultar el valor actual de AUTOCOMMIT** haciendo:

```
SELECT @@AUTOCOMMIT;        -- 0 implica el uso de transacciones explícitas
                            -- y 1 el uso de modo AUTOCOMMIT (implícitas)
```

Para **desactivar/activar la confirmación automática** (AUTOCOMMIT) ejecutar:

```
SET AUTOCOMMIT = {0 | 1};  -- 0 desactivar (explícitas) y 1 activar (implícitas)
```

Si se desactiva la confirmación automática siempre se tendría una transacción abierta y los cambios sólo se

aplicarían en la *BD* ejecutando la sentencia COMMIT de forma explícita. Recordar, que para **trabajar con transacciones en *MariaDB*** es necesario **utilizar *InnoDB***.

6.4 Instrucciones para el manejo de transacciones.

Por defecto, *MariaDB* automáticamente confirma las sentencias que no forman parte de una transacción. Los **resultados de cualquier sentencia UPDATE, DELETE o INSERT no precedida por START TRANSACTION serán inmediatamente visibles para todas las conexiones**.

La variable que controla la confirmación automática de las sentencias *SQL* es AUTOCOMMIT, que **se establece como verdadera por defecto**. Se puede cambiar de la siguiente forma `SET AUTOCOMMIT = 0;` y para devolver AUTOCOMMIT a verdadero `SET AUTOCOMMIT = 1;`. Para **consultar el estado de AUTOCOMMIT** ejecutar: **`SELECT @@AUTOCOMMIT;`**

***InnoDB** es el motor de almacenamiento más utilizado con soporte para transacciones. Es compatible con ACID, admite bloqueo de nivel de fila, recuperación de fallos y control de concurrencia de múltiples versiones.*

Para poder utilizar transacciones en *MariaDB* se debe **utilizar el motor de almacenamiento *InnoDB***.

Los **pasos para realizar una transacción** en *MariaDB* son los siguientes:

- ✓ **Indicar que se va a realizar una transacción** con la sentencia **START TRANSACTION**, BEGIN o BEGIN WORK.
- ✓ **Realizar las operaciones de manipulación** de datos sobre la *BD* (insertar, actualizar o borrar filas).
- ✓ Si las **operaciones se han realizado correctamente** y se quiere que los cambios se apliquen de forma permanente sobre la *BD* usar la sentencia **COMMIT**. Sin embargo, si durante las operaciones **ocurre algún error** y no se quieren aplicar los cambios realizados, se puede deshacerlos con la sentencia **ROLLBACK**.

Sintaxis:

```
START TRANSACTION [transaction_property [, transaction_property] ...] | BEGIN [WORK]
COMMIT [WORK]
ROLLBACK [WORK]
SET autocommit = {0 | 1}

transaction_property:
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
```

La *BD* crea una instantánea de los datos en el momento en que se inicia la transacción. Dentro de esa transacción, los datos que se ven serán consistentes entre sí, incluso si otros procesos están realizando cambios en la *BD*.

Sentencias para el manejo de transacciones en *MariaDB*:

- ✓ **START TRANSACTION / BEGIN**: **inicia una nueva transacción**. Si ya **existe una iniciada, esta finaliza con confirmación de datos** (COMMIT). La confirmación automática permanece deshabilitada hasta que finaliza la transacción con COMMIT/ROLLBACK (luego, el modo de confirmación automática vuelve a su estado anterior).
- ✓ **COMMIT**: **termina la transacción guardando en la *BD* todos los cambios realizados**. Cualquier tipo de bloqueo que se mantuviera durante la transacción queda liberado y vuelve al modo AUTOCOMMIT.
- ✓ **ROLLBACK**: **termina la transacción deshaciendo todos los cambios que se hubieran realizado sobre la *BD***. Libera los bloqueos realizados por la transacción y vuelve al modo AUTOCOMMIT.
- ✓ **SAVEPOINT punto_de_salvaguarda**: **crea un punto de salvaguarda al que se puede retroceder** utilizando la sentencia ROLLBACK TO SAVEPOINT. Si en una transacción existen dos SAVEPOINT con el mismo nombre sólo se tendrá en cuenta el último que se haya definido.
- ✓ **ROLLBACK [WORK] TO [SAVEPOINT] punto_de_salvaguarda**: **realiza un ROLLBACK de todas las sentencias ejecutadas desde que se creó el punto de salvaguarda indicado**.
- ✓ **RELEASE SAVEPOINT punto_de_salvaguarda**: **elimina** un **SAVEPOINT**. No deshace (ROLLBACK) ni confirma (COMMIT) ningún cambio.
- ✓ **SET TRANSACTION ISOLATION LEVEL**: **permite cambiar el nivel de aislamiento de las transacciones**.
- ✓ **LOCK TABLES**: **permite bloquear explícitamente una o varias tablas**. A la vez **cierra todas las transacciones abiertas**. Para desbloquear las tablas (todas las que estuvieran bloqueadas) hay que ejecutar la sentencia **UNLOCK TABLES**.

Tener en cuenta que, **si AUTOCOMMIT se establece en verdadero** (1)**, COMMIT y ROLLBACK no hacen nada**.

Independientemente del valor de la propiedad AUTOCOMMIT*, todas las sentencias DDL del lenguaje SQL (*ALTER*,* CREATE*,* DROP*, …) tienen el* COMMIT *o confirmación de manera implícita o automática.*

Ejemplo 1: ¿Qué ocurrirá en la *BD* de un taller de reparación de coches al ejecutar las siguientes sentencias *SQL*? Se añade un nuevo cliente, pero no su vehículo en la *BD* del taller.

```
START TRANSACTION;
INSERT INTO cliente VALUES('00001', '123456789Z', 'Martín Sánchez', 'Pepe', 'Andalucía, 33', '18001');
SAVEPOINT cliente_insertado;
INSERT INTO vehiculo VALUES('1234-BCD', 'Seat', 'Ibiza', 'azul', NULL, '00001');
ROLLBACK TO SAVEPOINT cliente_insertado; COMMIT;
```

Seguidamente, un cliente nuevo lleva su vehículo al taller. En recepción se insertan los siguientes datos:

```
INSERT INTO cliente(codCliente, nif, apellidos, nombre, direccion, cp)
 VALUES('00002', '987654321P', 'Moreno López', 'Carlos', 'Madrid, 28', '18002');
INSERT INTO vehiculo(matricula, marca, modelo, color, fechaMatriculacion, idCliente)
 VALUES('4321-DVR', 'Peugeot', '107', 'rojo', '2012-02-11', '00002');
INSERT INTO reparacion VALUES('4321-DVR', '2022-01-10', 100.00, 'Sustituir luces', 0, NULL, NULL);
```

Una vez que el vehículo se ha reparado, se avisa al cliente para que pase a recogerlo. Si se quieren registrar las operaciones en las tablas afectadas haciendo uso de una transacción, se tendría lo siguiente:

```
START TRANSACTION;
UPDATE reparacion SET fechaSalida=CURDATE(), reparado=1, observaciones='Sin observaciones'
 WHERE matricula='4321-DVR';
INSERT INTO incluyen VALUES ('LD_222', 11, 1);
UPDATE recambio SET stock=stock-1 WHERE idRecambio='LD_222';
INSERT INTO intervienen VALUES ('9000', 11, 0.15);
INSERT INTO realizan VALUES (11, '111000888', 0.15);
COMMIT;
```

Ejemplo 2:

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;
CREATE TABLE cliente (
    id INT UNSIGNED PRIMARY KEY,
    nombre VARCHAR(20) NOT NULL
);
START TRANSACTION;
INSERT INTO cliente VALUES (1, 'Pepe');
COMMIT;
-- 1. ¿Qué devolverá esta consulta?
SELECT * FROM cliente;
SET AUTOCOMMIT = 0;
INSERT INTO cliente VALUES (2, 'María'), (20, 'Juan');
DELETE FROM cliente WHERE nombre = 'Pepe';
-- 2. ¿Qué devolverá esta consulta?
SELECT * FROM cliente;
ROLLBACK;
-- 3. ¿Qué devolverá esta consulta?
SELECT * FROM cliente;
```

Ejemplo 3:

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;
CREATE TABLE cuentas (
    id INTEGER UNSIGNED PRIMARY KEY,
    saldo DECIMAL(11,2) NOT NULL CHECK (saldo >= 0.0)
);
INSERT INTO cuentas VALUES (1, 1000), (2, 2000), (3, 0);
-- 1. Consultar el estado actual de las cuentas.
SELECT * FROM cuentas;
-- 2. Suponer que se quiere realizar un traspaso de dinero entre dos cuentas:
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
-- 3. ¿Qué devolverá esta consulta?
SELECT * FROM cuentas;
-- 4. Suponer que se quiere realizar un traspaso de dinero entre dos cuentas con
--    la siguiente transacción y una de las dos cuentas no existe:
```

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 9999; -- Se debe comprobar que la cuenta no existe
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;    -- y hacer ROLLBACK, no daría error UPDATE
ROLLBACK;
-- 5. ¿Qué devolverá esta consulta?
SELECT * FROM cuentas;
-- 6. Suponer que se quiere realizar un traspaso de dinero entre dos cuentas con
--    la siguiente transacción y la cuenta origen no tiene saldo:
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 3; -- Salta error, se debe capturar y hacer ROLLBACK
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
ROLLBACK;
-- 7. ¿Qué devolverá esta consulta?
SELECT * FROM cuentas;
```

Ejemplo 4:

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;
CREATE TABLE producto (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  precio DECIMAL(7,2)
);
INSERT INTO producto (id, nombre) VALUES (1, 'Primero'), (2, 'Segundo'), (3, 'Tercero');
-- 1. Comprobar las filas que existen en la tabla.
SELECT * FROM producto;
-- 2. Se ejecuta una transacción que incluye un SAVEPOINT
START TRANSACTION;
INSERT INTO producto (id, nombre) VALUES (4, 'Cuarto');
SAVEPOINT sp1;
INSERT INTO producto (id, nombre) VALUES (5, 'Cinco'), (6, 'Seis');
ROLLBACK TO sp1;
-- 3. ¿Qué devolverá esta consulta?
SELECT * FROM producto;
```

6.5 Problemas asociados al acceso concurrente a los datos.

En una *BD* a la que accede un solo usuario, un dato puede ser modificado sin tener en cuenta que otros usuarios puedan modificar el mismo dato al mismo tiempo. Sin embargo, **en una *BD* multiusuario, las sentencias contenidas en varias transacciones pueden actualizar los datos simultáneamente**. Las transacciones ejecutadas simultáneamente, deben generar resultados consistentes. Por tanto, **una *BD* multiusuario debe asegurar**:

- ✓ **Concurrencia de datos**: asegura que los **usuarios pueden acceder a los datos al mismo tiempo**.
- ✓ **Consistencia de datos**: asegura que **cada usuario tiene una vista consistente de los datos**, incluyendo los cambios visibles realizados por las transacciones del mismo usuario y las transacciones finalizadas de otros usuarios.

En una *BD* monousuario, no son necesarios los bloqueos ya que sólo modifica la información un único usuario. Sin embargo, cuando diferentes usuarios acceden y modifican datos, la *BD* debe proveer un **mecanismo para prevenir la modificación concurrente del mismo dato**. Los **bloqueos permiten obtener** los siguientes requerimientos fundamentales en la *BD*:

- ✓ **Consistencia**: los **datos que están siendo consultados o modificados por un usuario no pueden ser cambiados por otros** hasta que el usuario haya finalizado la operación completa.
- ✓ **Integridad**: los **datos y sus estructuras deben reflejar todos los cambios efectuados sobre ellos en el orden correcto**.

El ***SGBD MariaDB* proporciona concurrencia de datos, consistencia e integridad en las transacciones mediante sus mecanismos de bloqueo**. Los **bloqueos se realizan de forma automática y no requiere la actuación del usuario**.

Sin los servicios de control de la concurrencia apropiados en un *SGBD* o con la falta de conocimiento sobre cómo usar estos servicios de forma adecuada, el contenido en la *BD* o los resultados de las consultas podrían corromperse, y dejar de ser fiables.

Cuando **dos transacciones distintas intentan acceder a los mismos datos** pueden ocurrir los siguientes **problemas típicos** debidos a la concurrencia:

- ✓ **Pérdida de actualizaciones (*Lost Update*)**: inconsistencia de datos que se produce en la ejecución concurrente

de transacciones, donde **el orden de las lecturas y escrituras en las distintas transacciones producen la inconsistencia de datos**. Todos los ***SGBD*** **modernos implementan algún mecanismo de control de la concurrencia que protege las operaciones de escritura de ser sobrescritas por transacciones concurrentes antes del final de la transacción**.

Transacción 1	Transacción 2
SELECT saldo FROM cuentas WHERE id = 1;	
	SELECT saldo FROM cuentas WHERE id = 1;
~~UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;~~	
	UPDATE cuentas SET saldo = saldo - 500 WHERE id = 1;

- ✓ **Lecturas sucias** (***Dirty Read***): sucede cuando una **segunda transacción lee datos que están siendo modificados por una transacción antes de que haga COMMIT**. Significa que la transacción acepta el riesgo de leer datos no fiables (no confirmados) que podrían cambiar o actualizarse con datos que podrían ser deshechos (ROLLBACK).

Transacción 1	Transacción 2
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;	
	SELECT saldo FROM cuentas WHERE id = 1;
ROLLBACK	

- ✓ **Lecturas no repetibles** (***Non-Repeatable Read***): se produce cuando **una transacción consulta los mismos datos dos veces** durante la ejecución de la misma y la **segunda vez encuentra que los datos han sido modificados** por otra transacción (no-reproducibilidad).

Transacción 1	Transacción 2
SELECT saldo FROM cuentas WHERE id = 1;	
	UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
SELECT saldo FROM cuentas WHERE id = 1;	

- ✓ **Lecturas fantasmas** (***Phantom Read***): este error ocurre cuando una **transacción ejecuta dos veces una consulta** que devuelve un conjunto de filas y en la **segunda ejecución de la consulta aparecen nuevas filas** (o desaparecen) **en el conjunto** que no existían cuando se inició la transacción.

Transacción 1	Transacción 2
SELECT SUM(saldo) FROM cuentas;	
	INSERT INTO cuentas VALUES (4, 3000);
SELECT SUM(saldo) FROM cuentas;	

6.6 Niveles de aislamiento.

La **propiedad de aislamiento de *ACID* es un desafío**. Dependiendo del mecanismo de control de la concurrencia **puede dar lugar a conflictos de concurrencia y tiempos de espera muy elevados**, bajando la velocidad de producción de la *BD*.

Una **sesión** es una **conexión única a la *BD* que comienza cuando se entra en el sistema (*login*) y termina cuando se desconecta de ella**.

Cada vez que se abre el programa *HeidiSQL* o la *shell* se está abriendo una sesión (se pueden tener varias instancias de estos programas abiertas, por lo que se tendrían varias sesiones). Para **averiguar el identificador de conexión que asigna el *SGBD*** ejecutar:

```
SELECT CONNECTION_ID();
```

Cada sesión trabaja en su propia zona de memoria pudiendo llegar incluso a bloquear los datos de la *BD* con los que trabaja.

Para **evitar que sucedan los problemas de acceso concurrente** que se han comentado en el punto anterior se pueden **establecer diferentes niveles de aislamiento** que **controlan el nivel de bloqueo durante el acceso a los datos** dentro de una transacción.

Los **niveles de aislamiento determinan la manera en que las transacciones de una sesión pueden afectar a los datos recuperados o accedidos por otra sesión**. Hay por tanto dos conceptos interrelacionados: por un lado, la concurrencia (varias sesiones realizando transacciones al mismo tiempo) y por otro el grado de consistencia de los datos.

Cuanto **mayor es el grado de aislamiento, menor es el número de transacciones que se pueden realizar concurrentemente pero también es menor la posibilidad de que interfieran las transacciones**. Por otro lado, cuanto

menor es el grado de aislamiento, mayor es el número de transacciones que se pueden realizar concurrentemente pero el riesgo de conflicto entre transacciones es elevado.

El **estándar *ANSI/ISO SQL-92* define 4 niveles de aislamiento** (soportados todos ellos por el sistema de almacenamiento *InnoDB* de *MariaDB* y por la gran mayoría de *SGBD* modernos):

- ✓ **Lectura no confirmada** (**READ UNCOMMITTED**): también conocido como lectura sucia, es el **nivel más bajo de aislamiento**. Permite que una **transacción pueda leer filas que todavía no han sido confirmadas** (COMMIT). Este nivel es el que proporciona **mayor nivel de concurrencia** ya que **no se realiza ningún bloqueo**, por lo tanto, permite que sucedan los **problemas *Dirty Read*, *Non-Repeteable Read* y *Phantom Read***.
- ✓ **Lectura confirmada** (**READ COMMITTED**): **sólo se permite la lectura de datos que han sido confirmados**. En este caso los datos leídos por una transacción pueden ser modificados por otras transacciones, por lo tanto, **se pueden dar los problemas *Non-Repeteable Read* y *Phantom Read***. Soluciona el problema *Dirty Read*. Ofrece un **buen equilibrio entre el rendimiento y la consistencia de los datos**, y es adecuado para la mayoría de las aplicaciones. *PostgreSQL*, *Oracle* y *SQL Server* trabajan por defecto en este nivel.
- ✓ **Lectura repetible** (**REPEATABLE READ**): es el **nivel por defecto en *MariaDB/MySQL***. Las **lecturas repetidas de un conjunto de datos por la misma transacción dan los mismos resultados**. **Ningún cambio hecho en la *BD* por otros usuarios será visto por la transacción lanzada hasta que esta se confirme o deshaga**, es decir, si se repite dentro de una transacción una misma sentencia SELECT, esta devolverá siempre los mismos resultados (excepto cuando la misma transacción pudiera realizar cambios). **Sólo puede suceder el problema *Phantom Read*** (nuevas filas que han sido insertadas por otras transacciones).
- ✓ **Serializable** (**SERIALIZABLE**): mayor nivel de aislamiento. **Las transacciones se aíslan completamente dando la impresión de que se ejecutan secuencialmente, una detrás de otra**. Para conseguir esto, **los *SGBD* bloquean cada fila leída para que otras sesiones no puedan modificar estos datos hasta que la transacción finalice**. El bloqueo dura hasta que la transacción se confirme o deshaga. Este nivel disminuye mucho el rendimiento y **puede provocar situaciones de bloqueo (*deadlock*** o abrazo mortal**)**. Este nivel es **similar a REPEATABLE READ, pero todas las sentencias SELECT son convertidas implícitamente a SELECT ... LOCK IN SHARE MODE**.

La siguiente tabla muestra los **problemas que pueden ocurrir en cada uno de los niveles de aislamiento**:

NIVEL DE AISLAMIENTO / PROBLEMA	*Actualización perdida*	*Lectura sucia*	*Lectura no repetible*	*Lectura fantasma*
Lectura no confirmada	NO posible	POSIBLE	POSIBLE	POSIBLE
Lectura confirmada	NO posible	NO posible	POSIBLE	POSIBLE
Lectura reproducible	NO posible	NO posible	NO posible	POSIBLE
Serializable	NO posible	NO posible	NO posible	NO posible

Es importante **destacar que los niveles de aislamiento no indican nada acerca de restricciones de escritura**. Para las **operaciones de escritura se usa típicamente alguna protección de bloqueo**, y **una escritura se protege siempre contra sobreescritura de otras transacciones hasta el final de la transacción**.

Se puede **consultar el nivel de aislamiento que se está utilizando**, consultando el contenido de la variable global y de sesión **@@tx_isolation**. Ejemplo:

```
SELECT @@GLOBAL.tx_isolation, @@tx_isolation; -- SHOW VARIABLES LIKE 'tx_isolation';
```

El **nivel de aislamiento por defecto** se puede **cambiar** ejecutando la **sentencia**:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
    SERIALIZABLE}; -- Si no se indica GLOBAL o SESSION se establece únicamente para la siguiente transacción
SET [GLOBAL|SESSION] [@@]tx_isolation='{READ-UNCOMMITTED | READ-COMMITTED | REPEATABLE-READ | SERIALIZABLE}';
```

Más información en https://mariadb.com/kb/en/set-transaction/

6.6.1 Evaluación de los niveles de aislamiento ante el problema *Dirty Read*.

En este ejemplo se va a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla. Para simular los dos usuarios se van a utilizar dos terminales distintas para conectar al *SGBD MariaDB*. Desde la terminal 1 se van a ejecutar las siguientes sentencias *SQL*:

```
DROP DATABASE IF EXISTS niveles;
CREATE DATABASE niveles CHARACTER SET utf8mb4;
```

```
USE niveles;
CREATE TABLE cuentas (
    id INTEGER UNSIGNED PRIMARY KEY,
    saldo DECIMAL(10,2) CHECK (saldo >= 0.0)
);
INSERT INTO cuentas VALUES (1, 1000), (2, 2000), (3, 0);
-- 1. Se configura que en esta sesión se va a utilizar el nivel de aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 2. Se ejecuta una transacción para transferir dinero entre dos cuentas
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1; -- Se actualiza el saldo de cuenta_1 a 900
```

NOTA: observar que la transacción que se está ejecutando en la terminal 1 todavía no ha finalizado, ya que no se ha ejecutado COMMIT ni ROLLBACK aún.

Ahora desde la terminal 2 ejecutar las siguientes sentencias *SQL*:

```
-- 1. Seleccionar la base de datos
USE niveles;
-- 2. Configurar que en esta sesión se va a utilizar el nivel de aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 3. Iniciar una transacción y observar los datos que existen en la tabla cuentas
START TRANSACTION;
SELECT * FROM cuentas WHERE id = 1; -- Se produce una lectura sucia, saldo cuenta_1 = 900
```

Ahora ejecutar ROLLBACK en la terminal 1 para finalizar la transacción que estaba sin finalizar.

```
-- 3. Deshacer las operaciones realizadas en la transacción
ROLLBACK; -- Se deshacen operaciones de la transacción y saldo cuenta_1 vuelve a valer 1000
```

Desde la terminal 2 volver a ejecutar esta sentencia:

```
-- 4. Se observan los datos que existen en la tabla cuentas
SELECT * FROM cuentas WHERE id = 1; -- Se lee nuevamente el valor de saldo de cuenta_1 1000
COMMIT;
```

Repetir el ejemplo utilizando los otros niveles de aislamiento (READ COMMITTED, REPEATABLE READ y SERIALIZABLE) y comprobar qué sucede en cada caso (actividad de clase 2).

6.6.2 Evaluación de los niveles de aislamiento ante el problema *Non-Repeatable Read*.

En este ejemplo se va a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla. Para simular los dos usuarios se van a utilizar dos terminales distintas para conectar al *SGBD MariaDB*. Desde la terminal 1 se van a ejecutar las siguientes sentencias *SQL*:

```
DROP DATABASE IF EXISTS niveles;
CREATE DATABASE niveles CHARACTER SET utf8mb4;
USE niveles;
CREATE TABLE cuentas (
    id INTEGER UNSIGNED PRIMARY KEY,
    saldo DECIMAL(10,2) CHECK (saldo >= 0.0)
);
INSERT INTO cuentas VALUES (1, 1000), (2, 2000), (3, 0);
-- 1. Se configura que en esta sesión se va a utilizar el nivel de aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 2. Se ejecuta una transacción para transferir dinero entre dos cuentas
START TRANSACTION;
SELECT * FROM cuentas WHERE id = 1;
```

NOTA: observar que la transacción que se está ejecutando en el terminal A todavía no ha finalizado, ya que no ha ejecutado COMMIT ni ROLLBACK aún.

Ahora desde la terminal 2 ejecutar las siguientes sentencias *SQL*:

```
-- 1. Seleccionar la base de datos
USE niveles;
-- 2. Configurar que en esta sesión se va a utilizar el nivel de aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 3. Iniciar una transacción y actualizar los datos de la tabla cuentas
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
-- 4. Finalizar la transacción con COMMIT
COMMIT;
```

Volver a ejecutar en la terminal 1 la misma consulta que se ejecutó al inicio de la transacción:

```
-- 4. Volver a ejecutar la misma sentencia para observar los datos que existen en la tabla cuentas
SELECT saldo FROM cuentas WHERE id = 1; -- El saldo obtenido es distinto
COMMIT;
```

Repetir el ejemplo utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`) y comprobar qué sucede en cada caso (actividad de clase 2).

6.6.3 Evaluación de los niveles de aislamiento ante el problema *Phantom Read*.

En este ejemplo se va a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla. Para simular los dos usuarios se van a utilizar dos terminales distintas para conectar al *SGBD MariaDB*. Desde la terminal 1 ejecutar las siguientes sentencias *SQL*:

```
DROP DATABASE IF EXISTS niveles;
CREATE DATABASE niveles CHARACTER SET utf8mb4;
USE niveles;
CREATE TABLE cuentas (
    id INTEGER UNSIGNED PRIMARY KEY,
    saldo DECIMAL(10,2) CHECK (saldo >= 0.0)
);
INSERT INTO cuentas VALUES (1, 1000), (2, 2000), (3, 0);
-- 1. Configurar que en esta sesión se va a utilizar el nivel de aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 2. Ejecutar una transacción para transferir dinero entre dos cuentas
START TRANSACTION;
SELECT SUM(saldo) FROM cuentas;
```

NOTA: observar que la transacción que se está ejecutando en la terminal 1 todavía no ha finalizado, ya que no ha ejecutado `COMMIT` ni `ROLLBACK` aún.

Ahora desde la terminal 2 ejecutar las siguientes sentencias *SQL*:

```
-- 1. Seleccionar la base de datos
USE niveles;
-- 2. Configurar que en esta sesión se va a utilizar el nivel de aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 3. Iniciar una transacción y actualizar los datos de la tabla cuentas
START TRANSACTION;
INSERT INTO cuentas VALUES (4, 3000);
-- 4. Finalizar la transacción con COMMIT
COMMIT;
```

Ahora volver a ejecutar en la terminal 1 la misma consulta que se ejecutó al inicio de la transacción:

```
-- 4. Volver a ejecutar la misma sentencia para observar los datos que existen en la tabla cuentas
SELECT SUM(saldo) FROM cuentas; -- Total obtenido es distinto al insertarse la cuenta 4
COMMIT;
```

Repetir el ejemplo utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`) y comprobar qué sucede en cada caso (actividad de clase 2).

6.7 Ejemplo de uso de transacciones (REPEATABLE READ).

El siguiente ejemplo ayudará a comprender el manejo de las transacciones haciendo uso de la consistencia en lectura (REPEATABLE READ) que es el nivel de aislamiento por defecto del *SGBD MariaDB*.

Antes de comenzar con el ejemplo crear una *BD* `test` y dentro de ella una tabla `alumnos` con dos campos (`id` y `alumno`). Una vez creada la tabla, insertar los 5 registros que se muestran en la imagen. Para seguir el ejemplo, abrir dos sesiones de cliente de línea de comandos o bien de *HeidiSQL* (**con *phpMyAdmin* no funciona ya que genera *id's* de sesión distintos para cada ejecución de una sentencia**). Se puede consultar el identificador de cada sesión ejecutando **`SELECT CONNECTION_ID();`**

Alumnos (5r × 2c)

id	alumno
1	Alumno 1
2	Alumno 2
3	Alumno 3
4	Alumno 4
5	Alumno 5

```
MariaDB [(none)]> SELECT CONNECTION_ID();
+-----------------+
| CONNECTION_ID() |
+-----------------+
|              10 |
+-----------------+
1 row in set (0.000 sec)

MariaDB [(none)]> USE test;
Database changed
MariaDB [test]>
```

Sesión 2

Sesión 1

```
MariaDB [(none)]> SELECT CONNECTION_ID();
+-----------------+
| CONNECTION_ID() |
+-----------------+
|              11 |
+-----------------+
1 row in set (0.000 sec)

MariaDB [(none)]> USE test;
Database changed
MariaDB [test]>
```

```
MariaDB [test]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [test]> INSERT INTO alumnos VALUES (6, 'Alumno 6');
Query OK, 1 row affected (0.005 sec)

MariaDB [test]> SELECT * FROM alumnos WHERE id=6;
+----+----------+
| id | alumno   |
+----+----------+
|  6 | Alumno 6 |
+----+----------+
1 row in set (0.000 sec)
```

Sesión 2

```
MariaDB [test]> SELECT * FROM alumnos WHERE id=6;
Empty set (0.000 sec)
```

```
MariaDB [test]> COMMIT;
Query OK, 0 rows affected (0.006 sec)
```

Sesión 1

```
MariaDB [test]> SELECT * FROM alumnos WHERE id=6;
+----+----------+
| id | alumno   |
+----+----------+
|  6 | Alumno 6 |
+----+----------+
1 row in set (0.000 sec)
```

```
MariaDB [test]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [test]> UPDATE alumnos SET alumno='Pepe Sánchez' WHERE id=1;
Query OK, 1 row affected (0.006 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> SELECT * FROM alumnos WHERE id=1;
+----+--------------+
| id | alumno       |
+----+--------------+
|  1 | Pepe Sánchez |
+----+--------------+
1 row in set (0.000 sec)
```

Sesión 1

```
MariaDB [test]> SELECT * FROM alumnos WHERE id=1;
+----+----------+
| id | alumno   |
+----+----------+
|  1 | Alumno 1 |
+----+----------+
1 row in set (0.000 sec)

MariaDB [test]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [test]> UPDATE alumnos SET alumno='María Ruiz' WHERE id=2;
Query OK, 1 row affected (0.005 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> SELECT * FROM alumnos WHERE id=2;
+----+------------+
| id | alumno     |
+----+------------+
|  2 | María Ruiz |
+----+------------+
1 row in set (0.000 sec)
```

Sesión 2

```
MariaDB [test]> SELECT * FROM alumnos WHERE id=2;
+----+----------+
| id | alumno   |
+----+----------+
|  2 | Alumno 2 |
+----+----------+
1 row in set (0.000 sec)

MariaDB [test]> COMMIT;
Query OK, 0 rows affected (0.006 sec)

MariaDB [test]> SELECT * FROM alumnos;
+----+--------------+
| id | alumno       |
+----+--------------+
|  1 | Pepe Sánchez |
|  2 | Alumno 2     |
|  3 | Alumno 3     |
|  4 | Alumno 4     |
|  5 | Alumno 5     |
|  6 | Alumno 6     |
+----+--------------+
6 rows in set (0.000 sec)
```

Sesión 1

```
MariaDB [test]> COMMIT;
Query OK, 0 rows affected (0.001 sec)

MariaDB [test]> SELECT * FROM alumnos;
+----+--------------+
| id | alumno       |
+----+--------------+
|  1 | Pepe Sánchez |
|  2 | María Ruiz   |
|  3 | Alumno 3     |
|  4 | Alumno 4     |
|  5 | Alumno 5     |
|  6 | Alumno 6     |
+----+--------------+
6 rows in set (0.001 sec)
```

Sesión 2

```
MariaDB [test]> SELECT * FROM alumnos;
+----+--------------+
| id | alumno       |
+----+--------------+
|  1 | Pepe Sánchez |
|  2 | María Ruiz   |
|  3 | Alumno 3     |
|  4 | Alumno 4     |
|  5 | Alumno 5     |
|  6 | Alumno 6     |
+----+--------------+
6 rows in set (0.000 sec)
```

6.8 Transacciones y bloqueos.

Las **situaciones en las que se producen bloqueos** dentro de una transacción son:

- ✓ Sentencia **UPDATE**: las **filas afectadas se bloquean hasta que se confirme o deshaga la transacción**.
- ✓ Sentencia **INSERT**: si **existe clave primaria**, las **filas insertadas quedan bloqueadas para prevenir que otra transacción pueda introducir otra fila con la misma clave**.
- ✓ Sentencia **LOCK TABLE[S]**: **bloquea la tabla entera**. No es muy eficiente pues reduce la concurrencia.
- ✓ Si dentro de una sentencia **SELECT** se utilizan las **cláusulas FOR UPDATE o LOCK IN SHARE MODE**, todas las **filas devueltas por la sentencia SELECT serán bloqueadas para actualización**:

```
SELECT <opciones_de_la_sentencia> [FOR UPDATE | LOCK IN SHARE MODE];
```

La **cláusula LOCK IN SHARE MODE** suele utilizarse **para garantizar la integridad referencial**. Imaginar que se está intentando introducir los datos de un nuevo departamento al que se le asignará un determinado centro. Existe la clave ajena: `departamento.numce` → `centro.numce`. Puede ocurrir que, durante el proceso, otra transacción elimine la fila correspondiente del centro en la tabla `centro`. Al intentar introducir el departamento se daría una situación de error. Por eso, en estos casos, en primer lugar, hay que asegurarse de que nadie modificará el centro:

```
SELECT * FROM centro WHERE numce=5 LOCK IN SHARE MODE;
```

Por otro lado, la solución anterior no es buena para garantizar la clave única en una tabla. Imaginar que cada departamento se obtiene sumando 10 al último número asignado. Si se utiliza la cláusula `LOCK IN SHARE MODE` puede ocurrir que dos transacciones lean al mismo tiempo el último `id` asignado, le sumen 10 y a la hora de modificarlo provoquen una situación de clave duplicada o como se verá más adelante una situación de abrazo mortal (*deadlock*). La solución para este caso:

```
SELECT id FROM tablaX… FOR UPDATE;
UPDATE tablaX SET id…;
```

en la que primero **se intenta leer con intención de modificar**, utilizando además un **bloqueo FOR UPDATE** y luego, si ha sido posible realizar la operación anterior, entonces incrementarlo (UPDATE `tablaX` …). El nivel de bloqueo es el mismo que para un UPDATE.

Los bloqueos se eliminan cuando la transacción se confirma o deshace.

LOCK IN SHARE MODE y FOR UPDATE <u>aseguran que ninguna otra transacción pueda actualizar las filas seleccionadas</u>. La **<u>diferencia</u> entre ambas cláusulas está en <u>cómo tratan los bloqueos mientras leen los datos</u>**:

- ✓ **LOCK IN SHARE MODE no impide que otra transacción lea la misma fila que estaba bloqueada**. *MariaDB* esperará hasta que todas las transacciones que han modificado las filas seleccionadas se confirmen.
- ✓ **FOR UPDATE evita otras lecturas de bloqueo de la misma fila** (las lecturas que no son de bloqueo aún pueden leer esa fila, LOCK IN SHARE MODE y FOR UPDATE son lecturas de bloqueo).

6.9 Estrategias de bloqueo.

Si una transacción tiene necesidad de leer datos que posteriormente se pueden ver afectados por operaciones de manipulación (INSERT, UPDATE, DELETE), hay que **tomar medidas para que otra transacción no pueda modificar esos datos después de que hayan sido leídos por la primera transacción y antes de ser modificados por esta**.

Posibles **<u>soluciones</u>**:

- ✓ **<u>Estrategia de bloqueo pesimista</u>**: asume que en las transacciones concurrentes es muy fácil de que una fila que se acaba de leer cambie. Por tanto, habrá que **bloquear las filas después de leerlas y otras transacciones que quieran modificar los datos deberán esperar**.
- ✓ **<u>Estrategia de bloqueo optimista</u>**: asume que es muy poco probable que el valor de una fila que se acaba de leer cambie. Si se asume este planteamiento como mínimo **hay que asegurar de que la fila no ha sido modificada después de leerla y si así ha sido entonces la transacción no debe llevarse a cabo aun pudiéndose realizar**.

En la imagen se puede ver un ejemplo de transacción que utiliza la sentencia SELECT … FOR UPDATE para la **<u>implementación de la estrategia pesimista</u>**:

```
START TRANSACTION;
SELECT alumno
  FROM alumnos
  WHERE id=1 FOR UPDATE;
UPDATE alumnos
  SET alumno='Juan Ruiz'
  WHERE id=1;
COMMIT;
```

Es una **estrategia muy segura pues asegura la consistencia entre la lectura (SELECT) y la actualización (UPDATE)**, pero **limita mucho el rendimiento del sistema** al obligar a las transacciones a largas esperas para poder completarse.

En la **estrategia optimista**, como la **transacción no bloquea la fila que lee**, **antes de que realice la modificación sobre la misma debe asegurarse que el valor de la fila no ha cambiado desde que la leyó**; en caso de que no se hubieran producido cambios se realizará y confirmará la transacción, en caso contrario no se llevará a cabo.

Para la elección de una estrategia u otra hay que tener en cuenta la concurrencia y robustez, la estrategia pesimista supone menos errores y reintentos mientras que con la optimista se reduce el tiempo de duración de los bloqueos aumentando por tanto la concurrencia y el rendimiento del sistema.

Generalmente suele utilizarse la estrategia pesimista y sólo se recurre a la optimista si la duración de los bloqueos o el número de filas que se bloquean es elevado en la estrategia pesimista. De todas formas, el uso de una u otra depende mucho de las características de la aplicación.

6.10 *Deadlock*.

También conocido como **abrazo mortal**. Esta situación **ocurre cuando una transacción *A* intenta modificar los datos que están siendo modificados por una transacción *B* y a su vez esta última intenta modificar los datos que están siendo modificados por la primera transacción *A***.

Tº	SUCURSAL A	SUCURSAL B	SITUACION
1	Modifica cuenta X saldo		La cuenta X queda bloqueada por A
2		Modifica cuenta Y saldo	La cuenta Y queda bloqueada por B
3	Modifica cuenta Y saldo		A queda a la espera pues Y está bloqueada
4		Modifica cuenta X saldo	B queda a la espera pues X está bloqueada

Como consecuencia del abrazo mortal, una de las dos transacciones realizará un ROLLBACK provocando una situación de error.

Volver a la situación inicial de la tabla `alumnos`, 5 filas con 5 alumnos con `id` 1 al 5 y de nombres ‘Alumno 1’ a ‘Alumno 5’. Abrir dos sesiones:

```
MariaDB [(none)]> SELECT CONNECTION_ID();
+-----------------+
| CONNECTION_ID() |
+-----------------+
|              13 |
+-----------------+
1 row in set (0.000 sec)

MariaDB [(none)]> USE test;
Database changed
MariaDB [test]>
```

Sesión 1

```
MariaDB [(none)]> SELECT CONNECTION_ID();
+-----------------+
| CONNECTION_ID() |
+-----------------+
|              14 |
+-----------------+
1 row in set (0.000 sec)

MariaDB [(none)]> USE test;
Database changed
MariaDB [test]>
```

```
MariaDB [test]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [test]> UPDATE alumnos SET alumno='Maria Ruiz' WHERE id=1;
Query OK, 1 row affected (0.005 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> SELECT * FROM alumnos WHERE id=1;
+----+------------+
| id | alumno     |
+----+------------+
|  1 | Maria Ruiz |
+----+------------+
1 row in set (0.000 sec)
```

Sesión 2

```
MariaDB [test]> UPDATE alumnos SET alumno='Pepe Sánchez' WHERE id=1;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
MariaDB [test]>
```

Después de un tiempo de espera, aparece un error de tiempo de espera excedido al intentar modificar una fila que está siendo modificada y aún no ha sido confirmada o desechada por otra sesión.

Sesión 1

```
MariaDB [test]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [test]> UPDATE alumnos SET alumno='Pepe Sánchez' WHERE id=2;
Query OK, 1 row affected (0.005 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> SELECT * FROM alumnos WHERE id=2;
+----+--------------+
| id | alumno       |
+----+--------------+
|  2 | Pepe Sánchez |
+----+--------------+
1 row in set (0.000 sec)
```

```
MariaDB [test]> UPDATE alumnos SET alumno='Carlos Moreno' WHERE id=2;
```

La sesión 1 queda a la espera de poder modificar el alumno con id 2. Si desde la sesión 2 se intenta modificar el alumno con id 1 que tiene la sesión 1 bloqueado, y a su vez la sesión 2 tiene bloqueado el id 2 que quiere modificar la sesión 1 se produce un *deadlock*. *MariaDB* lo detecta y termina la transacción en la sesión 2.

```
MariaDB [test]> UPDATE alumnos SET alumno='Marta López' WHERE id=1;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
MariaDB [test]>
```

La sesión 1 termina su espera y ejecuta la actualización por la que se encontraba esperando.

```
MariaDB [test]> UPDATE alumnos SET alumno='Carlos Moreno' WHERE id=2;
Query OK, 1 row affected (43.707 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
MariaDB [test]> SELECT * FROM alumnos;
+----+----------+
| id | alumno   |
+----+----------+
|  1 | Alumno 1 |
|  2 | Alumno 2 |
|  3 | Alumno 3 |
|  4 | Alumno 4 |
|  5 | Alumno 5 |
+----+----------+
5 rows in set (0.000 sec)
```

```
MariaDB [test]> SELECT * FROM alumnos;
+----+---------------+
| id | alumno        |
+----+---------------+
|  1 | Maria Ruiz    |
|  2 | Carlos Moreno |
|  3 | Alumno 3      |
|  4 | Alumno 4      |
|  5 | Alumno 5      |
+----+---------------+
5 rows in set (0.000 sec)

MariaDB [test]> COMMIT;
Query OK, 0 rows affected (0.006 sec)
```

Sesión 2

Sesión 2

```
MariaDB [test]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [test]> SELECT * FROM alumnos;
+----+---------------+
| id | alumno        |
+----+---------------+
|  1 | Maria Ruiz    |
|  2 | Carlos Moreno |
|  3 | Alumno 3      |
|  4 | Alumno 4      |
|  5 | Alumno 5      |
+----+---------------+
5 rows in set (0.001 sec)
```

Al intentar escribir en una fila bloqueada (sin ser *deadlock*), se produce un tiempo de espera de 50 segundos tras el que se produce un error de tiempo de espera excedido y la transacción se deshace.

La **variable de sistema `innodb_lock_wait_timeout`** (`SELECT @@innodb_lock_wait_timeout;`) es la que **almacena el tiempo máximo de espera**. Dicha variable puede modificarse en caso de aplicaciones con largas transacciones.

6.11 Aspectos a tener en cuenta a la hora de utilizar transacciones.

Siempre **debe mantenerse la integridad de la *BD*** para garantizar que los datos que almacena son precisos, fieles a la realidad y lo más correctos posibles. La **fiabilidad es la prioridad número 1**, antes del rendimiento, etc., pero el nivel de aislamiento predeterminado que utilizan algunos *SGBD* favorece el rendimiento antes que la fiabilidad. El

nivel de aislamiento adecuado **debe planificarse con mucho cuidado**, y en **caso de no tenerse claro** cual utilizar se debe **elegir el aislamiento SERIALIZABLE**.

La transacción **debe ser tan corta como sea posible**, para minimizar la competencia de la concurrencia y el bloqueo de las transacciones simultáneas. La **duración de los bloqueos debe ser lo mínimo posible**. De igual manera **debe ser también mínimo el número de filas a bloquear por las transacciones**.

Las transacciones **no deben contener ningún diálogo con el usuario**, ya que esto ralentizaría el procesamiento.

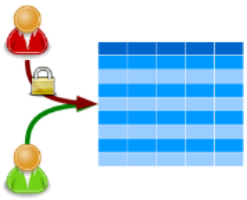
Cada transacción debe tener una **tarea bien definida**.

La **operación de ROLLBACK** (deshacer) **debe evitarse en los casos que se pueda, pues es una operación costosa en tiempo y utilización de recursos**. Por eso, habrá que **adelantarse a las situaciones que pueden provocarlo** como los intentos de inserción de filas de clave duplicada, realizando en este caso búsquedas antes de las inserciones. Relacionado con este punto, **se evitarán en la medida en que se pueda la creación y utilización de puntos de salvaguarda**; sólo si el número de situaciones posibles a controlar es elevado, quizás en ese caso sea aceptable su utilización. Indicar también que la **operación de confirmación o COMMIT al igual que la de ROLLBACK es una operación costosa** pues trae consigo escritura física en disco de los datos de la caché de memoria del *SGBD*. Por eso hay que **utilizar esta instrucción de confirmación en los sistemas en tiempo real de un negocio para mantener los datos íntegros, pero hay que minimizar su uso en situaciones de carga masiva de datos** a través de *scripts*, por ejemplo.

En cada programa **debe especificarse explícitamente el comienzo y fin de transacción y no asumir que los programas externos son los que validarán o desharán la transacción**.

Evitar sentencias *DDL* en las transacciones.

7 Políticas de bloqueo.



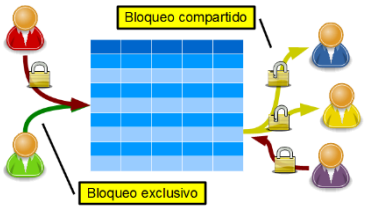
La *BD* permite el uso de diferentes tipos de bloqueos, dependiendo de la operación que realiza el bloqueo.

Los bloqueos afectan a la interacción de lectores y escritores. Un lector es una consulta sobre un recurso, mientras que un escritor es una sentencia que realiza una modificación sobre un recurso. Las siguientes reglas resumen el **comportamiento del *SGBD MariaDB*** sobre lectores y escritores:

- ✓ **Un registro es bloqueado sólo cuando es modificado por un escritor**: cuando una sentencia actualiza un registro, la transacción obtiene un bloqueo sólo para ese registro.
- ✓ **Un escritor de un registro bloquea a otro escritor concurrente del mismo registro**: si una transacción está modificando una fila, un bloqueo del registro impide que otra transacción modifique el mismo registro.
- ✓ **Un lector nunca bloquea a un escritor**: puesto que un lector de un registro no lo bloquea, un escritor puede modificar dicho registro. La **única excepción es la sentencia SELECT … FOR UPDATE**, que es un tipo especial de sentencia SELECT que bloquea el registro que está siendo consultado.
- ✓ **Un escritor nunca bloquea a un lector**: cuando un registro está siendo modificado, la *BD* proporciona al lector una vista del registro sin los cambios que se están realizando.

Hay **dos mecanismos para el bloqueo de los datos en una *BD***: el bloqueo pesimista y bloqueo optimista. En el **bloqueo pesimista de un registro o una tabla se realiza el bloqueo inmediatamente, en cuanto el bloqueo se solicita,** mientras que en un **bloqueo optimista el acceso al registro o la tabla sólo está cerrado en el momento en que los cambios realizados a ese registro se actualizan en el disco**. Esta última situación sólo es apropiada cuando hay menos posibilidad de que alguien necesite acceder al registro mientras está bloqueado, de lo contrario no se podrá estar seguro de que la actualización tenga éxito, porque el intento de actualizar el registro producirá un error si otro usuario actualiza antes el registro. Con el bloqueo pesimista se garantiza que el registro será actualizado.

7.1 Bloqueos compartidos y exclusivos.



En general, la *BD* usa **dos tipos de bloqueos**: bloqueos exclusivos y bloqueos compartidos. Un recurso, por ejemplo, un registro de una tabla, sólo puede obtener un bloqueo exclusivo, pero puede conseguir varios bloqueos compartidos.

- ✓ **Bloqueo exclusivo**: este modo **previene que sea compartido el recurso asociado**. Una **transacción obtiene un**

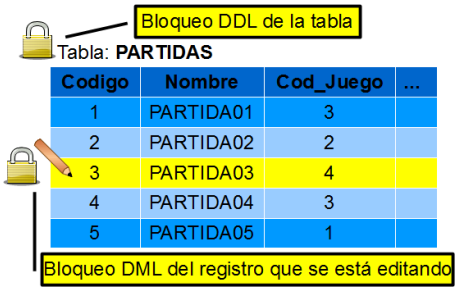
bloqueo exclusivo cuando modifica los datos. La primera transacción que bloquea un recurso exclusivamente, es la única transacción que puede modificar el recurso hasta que el bloqueo exclusivo es liberado.

- ✓ **Bloqueo compartido**: este modo permite que sea compartido el recurso asociado, dependiendo de la operación en la que se encuentra involucrado. Varios **usuarios que estén leyendo datos pueden compartir los datos, realizando bloqueos compartidos para prevenir el acceso concurrente de un escritor que necesita un bloqueo exclusivo**. Varias transacciones pueden obtener bloqueos compartidos del mismo recurso.

Por ejemplo, suponer que una transacción usa la sentencia **SELECT … FOR UPDATE** para consultar un registro de una tabla. La transacción **obtiene un bloqueo exclusivo del registro y un bloqueo compartido de la tabla**. El bloqueo del registro permite a otras sesiones que modifiquen cualquier otro registro que no sea el registro bloqueado, mientras que el bloqueo de la tabla previene que otras sesiones modifiquen la estructura de la tabla. De esta manera, la *BD* permite la ejecución de todas las sentencias que sean posibles.

Más información en https://mariadb.com/kb/en/innodb-lock-modes/

7.2 Bloqueos automáticos.



El *SGBD* ***MariaDB*** **bloquea automáticamente un recurso usado por una transacción para prevenir que otras transacciones realicen alguna acción que requiera acceso exclusivo sobre el mismo recurso**. El *SGBD* adquiere automáticamente diferentes tipos de bloqueos con diferentes niveles de restricción dependiendo del recurso y de la operación que se realice.

Los **bloqueos que realiza el *SGBD MariaDB*** están divididos en las siguientes **categorías**:

- ✓ **Bloqueos *DML***: protegen los datos, **garantizando la integridad de los datos accedidos de forma concurrente por varios usuarios**. Por ejemplo, evitan que dos clientes compren el último artículo disponible en una tienda online. Estos bloqueos pueden ser sobre un sólo registro o sobre la tabla completa.
- ✓ **Bloqueos *DDL***: protegen la definición del esquema de un objeto mientras una operación *DDL* actúa sobre él. Los **bloqueos se realizan de manera automática** por cualquier sentencia *DDL* que lo requiera. Los **usuarios no pueden solicitar explícitamente un bloqueo *DDL***. La ejecución de una **sentencia del *DDL* provoca la confirmación de la transacción de forma implícita** y comienza una nueva.
- ✓ **Bloqueos del sistema**: para **proteger la *BD* interna y las estructuras de memoria**.

7.3 Bloqueos manuales.

Se ha visto que *MariaDB* realiza bloqueos de forma automática para asegurar la concurrencia de datos, su integridad y consistencia en la consulta de datos. Sin embargo, **también se pueden omitir los mecanismos de bloqueo que realiza por defecto *MariaDB***. Esto puede ser **útil en situaciones como las siguientes**:

- ✓ **En aplicaciones que requieren consistencia en la consulta de datos a nivel de transacciones o en lecturas repetitivas**: en este caso, las consultas obtienen datos consistentes en la duración de la transacción, sin reflejar los cambios realizados por otras transacciones.
- ✓ **En aplicaciones que requieren que una transacción tenga acceso exclusivo a un recurso con el fin de que no tenga que esperar que otras transacciones finalicen**.

7.3.1 Sentencia `SELECT ... FOR UPDATE / LOCK IN SHARE MODE`.

La sentencia `SELECT … FOR UPDATE | LOCK IN SHARE MODE` **se utiliza para bloquear temporalmente las filas seleccionadas de una tabla y evitar que otras conexiones las modifiquen al mismo tiempo**. Esto se utiliza en aplicaciones donde es necesario evitar que múltiples usuarios actualicen la misma fila al mismo tiempo.

La **sintaxis básica** de la sentencia `SELECT … FOR UPDATE | LOCK IN SHARE MODE` es la siguiente:

```
SELECT * FROM table_name WHERE condition {FOR UPDATE | LOCK IN SHARE MODE};
```

donde **`table_name` es el nombre de la tabla a bloquear** y **`condition` es la condición que se utiliza para seleccionar las filas que se van a bloquear**. La cláusula **`FOR UPDATE | LOCK IN SHARE MODE` se agrega al final** de

la sentencia SELECT **para indicar que se deben bloquear las filas seleccionadas**.

La **cláusula FOR UPDATE | LOCK IN SHARE MODE de SELECT se aplica solo cuando la confirmación automática (AUTOCOMMIT) se desactiva (0) o se incluye en una transacción**.

Cuando se ejecuta la sentencia SELECT … FOR UPDATE | LOCK IN SHARE MODE, **se bloquean las filas seleccionadas de la tabla hasta que se complete la transacción**. Esto significa que **otras conexiones no pueden modificar y/o leer las filas bloqueadas** hasta que se libere el bloqueo (**a menos que el nivel de aislamiento sea READ UNCOMMITTED**). Para liberar el bloqueo, se debe realizar un COMMIT o ROLLBACK de la transacción.

Si **AUTOCOMMIT se establece en 1, las cláusulas LOCK IN SHARE MODE y FOR UPDATE no tienen efecto**.

Si el **nivel de aislamiento se establece en SERIALIZABLE, todas las sentencias SELECT se convierten en SELECT ... LOCK IN SHARE MODE**.

Es importante tener en cuenta que el uso de la sentencia SELECT … FOR UPDATE | LOCK IN SHARE MODE **puede afectar el rendimiento de la *BD* si se utiliza de manera inadecuada**. Por lo tanto, se recomienda utilizar esta sentencia solo cuando sea necesario y en la menor cantidad posible de filas.

Más información en https://mariadb.com/kb/en/for-update/

7.3.2 Sentencias LOCK TABLES y UNLOCK TABLES.

Las sentencias LOCK TABLES y UNLOCK TABLES se utilizan para **controlar el acceso concurrente a las tablas**.

LOCK TABLES *se suele utilizar* ***cuando el motor de almacenamiento no soporta transacciones****, por ejemplo,* ***MyISAM*** *y es necesario asegurarse de que no se ejecute ninguna transacción entre un* SELECT *y un* UPDATE *o para que la actualización de tablas sea más rápida.*

El comando **LOCK TABLES se utiliza para bloquear una o varias tablas en modo de escritura o lectura**, evitando que otros procesos accedan a ellas de manera concurrente.

El comando **UNLOCK TABLES se utiliza para desbloquear las tablas que fueron bloqueadas previamente** con LOCK TABLES.

Si se bloquea una tabla explícitamente con LOCK TABLES, todas las tablas relacionadas por una restricción de clave foránea se bloquean implícitamente.

```
LOCK TABLE[S]
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...
    [WAIT n|NOWAIT]

lock_type:
    READ [LOCAL]
  | [LOW_PRIORITY] WRITE
  | WRITE CONCURRENT

UNLOCK TABLES
```

Si **se ejecuta la sentencia LOCK TABLES para adquirir un nuevo bloqueo mientras ya se tienen otros bloqueos**, los **bloqueos existentes se liberan** implícitamente antes de que se otorguen los nuevos bloqueos. **Si se inicia una transacción también se liberan los bloqueos que se mantengan** de forma implícita.

Si la **conexión de una sesión de cliente finaliza**, ya sea de forma normal o anormal, el **servidor libera implícitamente todos los bloqueos** de tabla mantenidos por la sesión.

Option	Description
READ	Read lock, no writes allowed
READ LOCAL	Read lock, but allow concurrent inserts
WRITE	Exclusive write lock. No other connections can read or write to this table
LOW_PRIORITY WRITE	Exclusive write lock, but allow new read locks on the table until we get the write lock.
WRITE CONCURRENT	Exclusive write lock, but allow READ LOCAL locks to the table.

UNLOCK TABLES libera explícitamente cualquier bloqueo de tabla retenido por la sesión actual.

La **forma correcta de usar LOCK TABLES y UNLOCK TABLES con motores de almacenamiento transaccionales** (*InnoDB*), es comenzar la transacción con SET AUTOCOMMIT = 0 (no START TRANSACTION) seguido de LOCK TABLES, y no llamar a UNLOCK TABLES hasta que se confirme o deshaga la transacción explícitamente. Por ejemplo, si se necesita escribir en la tabla *t1* y leer de la tabla *t2*, se puede hacer lo siguiente:

```
SET AUTOCOMMIT = 0;
LOCK TABLES t1 WRITE, t2 READ, ...;
... Hacer lo necesario con las tablas t1 y t2 aquí ...
```

```
COMMIT; -- o ROLLBACK;
UNLOCK TABLES;
```

Es importante destacar que el **uso de LOCK TABLES puede afectar el rendimiento de la *BD* si se utiliza de manera inadecuada**. Por lo tanto, se recomienda **usar esta sentencia solo cuando sea necesario y en la menor cantidad posible de tablas**.

Más información en https://mariadb.com/kb/en/lock-tables/

Ejemplo: para comprobar el efecto del bloqueo de una tabla se va a realizar la siguiente práctica utilizando dos sesiones distintas.

Desde la sesión 1 desactivar el modo AUTOCOMMIT y ejecutar el código para bloquear la tabla *clientes* para escritura:

```
SET AUTOCOMMIT = 0;
LOCK TABLE clientes WRITE;
```

Desde la sesión 2 ejecutar: `UPDATE clientes SET nombre = 'José' WHERE codCliente = '00009';`

¿Qué ocurre? Esta operación se mantiene a la espera porque la tabla `clientes` está bloqueada.

De nuevo, desde la sesión 1 intentar ejecutar la operación anterior:

```
UPDATE clientes SET nombre = 'José' WHERE codCliente = '00009';
```

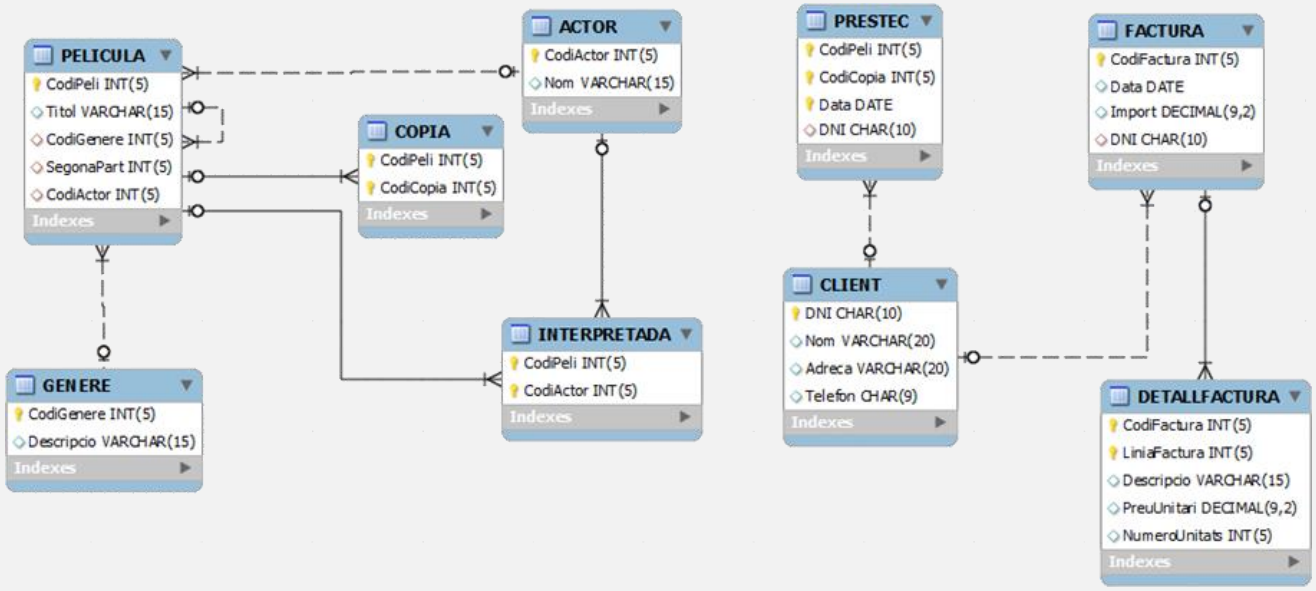
Como se puede comprobar la operación se ejecuta correctamente. Finalmente se deshace la transacción y se desbloquean las tablas.

```
ROLLBACK;
UNLOCK TABLES;
```

¿Se podría modificar el nombre del cliente ahora en la sesión 2? Ahora sí, ya que la tabla ha sido desbloqueada.

TAREA

27. **¿Qué son las propiedades *ACID*?**
28. **¿Cuáles son los cuatro problemas de concurrencia en el acceso a datos que pueden suceder cuando se realizan transacciones? Poner un ejemplo para cada uno de ellos.**
29. **Cuando se trabaja con transacciones, el *SGBD* puede bloquear conjuntos de datos para evitar o permitir que sucedan los problemas de concurrencia comentados en el ejercicio anterior. ¿Cuáles son los cuatro niveles de aislamiento que se pueden solicitar al *SGBD*?**
30. **¿Cuál es el nivel de aislamiento que se usa por defecto en las tablas *InnoDB* de *MariaDB*? ¿Es posible realizar transacciones sobre tablas *MyISAM* de *MariaDB*?**
31. **Basándose en el siguiente esquema:**



Reflexionar sobre las siguientes cuestiones:

a) **Cuando un cliente alquila una o varias películas, hay que registrar este hecho en la tabla préstamo (*PRESTEC*), así como crear una nueva factura (*FACTURA*) con los correspondientes detalles de factura en la tabla *DETALLFACTURA*. ¿Por qué es interesante definir una transacción que haga este conjunto de operaciones en una unidad?**

b) **¿Cuáles deben ser las operaciones previas a la ejecución de una transacción como la diseñada?**

c) **¿Cuáles deben ser las operaciones a ejecutar al finalizar la transacción?**

d) **¿Qué otras operaciones se podrían agrupar a través de transacciones para garantizar la consistencia?**

Crear el esquema de la transacción, e indicar las sentencias *SQL* que contendría y el orden de ejecución.

32. **Considerar que se tiene una tabla donde se almacena información sobre cuentas bancarias definida de la siguiente forma:**

```
CREATE TABLE cuentas (
    id INT UNSIGNED PRIMARY KEY,
    saldo DECIMAL(11,2) CHECK (saldo >= 0)
);
```

Suponer que se quiere realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente transacción:

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 20;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 30;
COMMIT;
```

a) **¿Qué ocurriría si el sistema falla o si se pierde la conexión entre el cliente y el servidor después de realizar la primera sentencia UPDATE?**

b) **¿Qué ocurriría si no existiese alguna de las dos cuentas (id = 20 o id = 30)?**

c) **¿Qué ocurriría en el caso de que la primera sentencia UPDATE falle porque hay menos de 100 € en la cuenta y no se cumpla la restricción del CHECK establecida en la tabla?**

Para conocer el número de visitas a ciertas páginas de un sitio web, se utiliza una tabla en una *BD* de *MariaDB* en la que se almacena el identificador de cada página, un nombre simbólico de la página y el número de visitas que recibe. Cada vez que un usuario accede a una página, se debe incrementar el número de visitas correspondiente. El esquema relacional y una posible extensión inicial son las que pueden ver en la imagen:

paginas

id	nombre	contador
1	Portada	0
2	Noticias	0
3	Resumen	0
4	Comentarios	0
5	Eventos	0

33. **Crear una tabla en *MariaDB* acorde al enunciado y al esquema relacional de la imagen. Utilizar una secuencia autonumérica para la columna *id* y usar valores por defecto (0) para la columna *contador*.**

34. **Insertar los datos que se muestran en la imagen usando el código más simple posible, aprovechando que el campo *id* es autonumérico y que el campo *contador* puede tomar valores por defecto (insertar filas asignando valores sólo a la columna *nombre*).**

35. **Abrir una terminal y *HeidiSQL* e iniciar sesión desde ambas aplicaciones. Seleccionar la *BD* donde se ha creado la tabla y seguidamente cambiar el nivel de aislamiento a READ COMMITTED mediante la siguiente sentencia:**

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
```

Escoger cualquiera de las páginas insertadas en el ejercicio anterior y ejecutar la siguiente sentencia:

```
SELECT contador FROM paginas WHERE id = <id_página_seleccionada>;
```

El objetivo es que desde cada una de las sesiones se sume al valor leído en la consulta anterior un número distinto (por ejemplo, desde la terminal se suma 100 y desde *HeidiSQL* se le suma 200). Actualizar desde cada sesión los datos con la siguiente sentencia:

```
UPDATE paginas set contador = <nuevo_valor> WHERE id = <id_página_seleccionada>;
COMMIT;
```

¿Hay pérdida de actualizaciones? ¿es esto un problema?

36. **Repetir el ejercicio anterior pero esta vez, cambiar la sentencia SELECT por:**

```
SELECT contador FROM paginas WHERE id = <id_página_seleccionada> FOR UPDATE;
```

¿Sigue habiendo pérdida de actualizaciones? ¿qué ocurre? ¿por qué?

37. **Volver a hacer el ejercicio 35 pero esta vez sin utilizar la cláusula FOR UPDATE en el SELECT y cambiar la primera sentencia por la siguiente:**

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

¿Sigue habiendo pérdida de actualizaciones? ¿qué ocurre? ¿por qué?

38. **Cambiar el nivel de aislamiento a READ COMMITTED. Ejecutar las siguientes transacciones (lecturas sucias):**

TRANSACCIÓN 1	TRANSACCIÓN 2
START TRANSACTION; **SELECT contador FROM paginas WHERE id=<id_seleccionado>;** **UPDATE paginas SET contador = <nuevo_valor>** WHERE id=<id_seleccionado>;	
	START TRANSACTION; SELECT contador FROM paginas WHERE id=<id_seleccionado>; UPDATE paginas SET contador = <nuevo_valor> WHERE id=<id_seleccionado>; COMMIT;
ROLLBACK;	

¿Hay lecturas sucias en la transacción 2? ¿y si se cambia el ROLLBACK final de la transacción 1 por un COMMIT?

39. **Volver a repetir el ejercicio anterior usando en ambas transacciones SELECT … FOR UPDATE. ¿Qué ocurre? ¿por qué?**
40. **Volver a repetir el ejercicio 38 sin utilizar la cláusula SELECT … FOR UPDATE y cambiar el nivel de aislamiento a SERIALIZABLE. ¿Qué ocurre? ¿por qué?**
41. **Cambiar el nivel de aislamiento a READ COMMITTED. Ejecutar las siguientes transacciones (lectura no repetible):**

TRANSACCIÓN 1	TRANSACCIÓN 2
START TRANSACTION; **SELECT * FROM paginas** WHERE id=<id_seleccionado>;	
	START TRANSACTION; SELECT contador FROM paginas WHERE id=<id_seleccionado>; UPDATE paginas SET contador = <nuevo_valor> WHERE id=<id_seleccionado>; COMMIT;
SELECT * FROM paginas WHERE id=<id_seleccionado>; COMMIT;	

¿Son las dos lecturas de la transacción 1 iguales? ¿Por qué?

42. **Volver a repetir el ejercicio anterior usando nivel de aislamiento SERIALIZABLE. ¿Qué ocurre? ¿por qué?**
43. **Cambiar el nivel de aislamiento a READ COMMITTED. Ejecutar cada una de las transacciones siguientes (lectura fantasma):**

TRANSACCIÓN 1	TRANSACCIÓN 2
START TRANSACTION; SELECT * FROM paginas;	
	START TRANSACTION; SELECT contador FROM paginas; INSERT INTO paginas(nombre) VALUES ("Fantasma"); COMMIT;
SELECT * FROM paginas; COMMIT;	

¿Son las dos lecturas de la transacción 1 iguales? ¿aparece la fila fantasma? ¿por qué?

44. **Volver a repetir el ejercicio anterior usando el nivel de aislamiento SERIALIZABLE. ¿Qué ocurre? ¿por qué?**