Tema 6 Transacciones.

1 Transacciones.

Las consultas a las bases de datos se ejecutan una detrás de otra. Para un sitio web que sirve páginas, no suele importar mucho el orden en que éstas se ejecutan, siempre y cuando se ejecuten rápido. Pero algunos tipos de consultas necesitan ejecutarse en un orden específico, como aquellas que dependen de los resultados de la anterior, o grupos de actualizaciones que necesitan ser hechos en su totalidad. Todos los tipos de tablas en MySQL pueden emplear bloqueos, pero sólo las tablas InnoDB y BDB tienen un sistema de transacciones.

A lo largo de este capítulo iremos viendo los mecanismos de bloqueo y transacciones. Para realizar los ejemplos y ejercicios del mismo, será necesario que tengamos dos ventanas distintas del cliente MySQL abiertas. Cada una de ellas tendrá una conexión distinta y, por tanto, podremos ver como los bloqueos impuestos por una de ellas afectan a la otra.

Una **transacción es un grupo de sentencias SQL agrupadas como si fueran una**. Un ejemplo típico es una transacción bancaria. Si queremos transferir el dinero de una cuenta a otra, necesitaremos al menos dos consultas:

```
UPDATE cuentas SET dinero = dinero-transferencia
WHERE cuenta=”cuenta1”;

UPDATE cuentas SET dinero = dinero+transferencia
WHERE cuenta=”cuenta2”;
```

En principio no parece haber problemas con esto, pero... ¿que sucedería si se fuera la luz o se cayera el sistema después de que se hubiera completado la primera consulta pero no la segunda? Que tendríamos dos clientes enfadados, y con razón, porque el primero habría dejado de tener el dinero mientras que el segundo no lo habría recibido. **En estos casos es cuando necesitamos que se ejecuten ambas consultas o no se ejecute ninguna**. Para ello las unimos en lo que se denomina una transacción.

1.1 Condiciones ACID. Atomicidad-Consistencia-Isolation-Durabilidad

Cada orden o comando SQL deben cumplir las llamadas condiciones “ACID”, que son las siguientes:

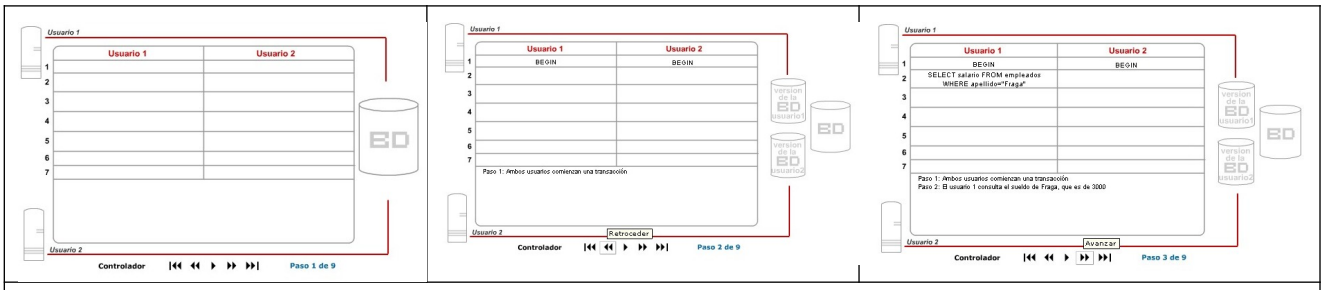
- **A (atomicidad):** las operaciones que se realizan deben poder considerarse como una sola. Eso significa que, o se llevan a cabo todas, o no se realiza ninguna.
- **C (consistencia):** cualquier operación que sea validada o cancelada **no puede dejar datos inconsistente**s (por ejemplo violando reglas de integridad referencial).
- **I (aislamiento o *isolation* en inglés):** las modificaciones realizadas por la operación deben aislarse de las modificaciones llevadas a cabo por otras operaciones que se ejecuten al mismo tiempo. El gestor de la base de datos debe aislar los datos ‘sucios’ para evitar que otros usuarios usen información no confirmada o validada.
- **D (durabilidad):** una vez realizada con éxito la operación, las modificaciones deben persistir en la base de datos.

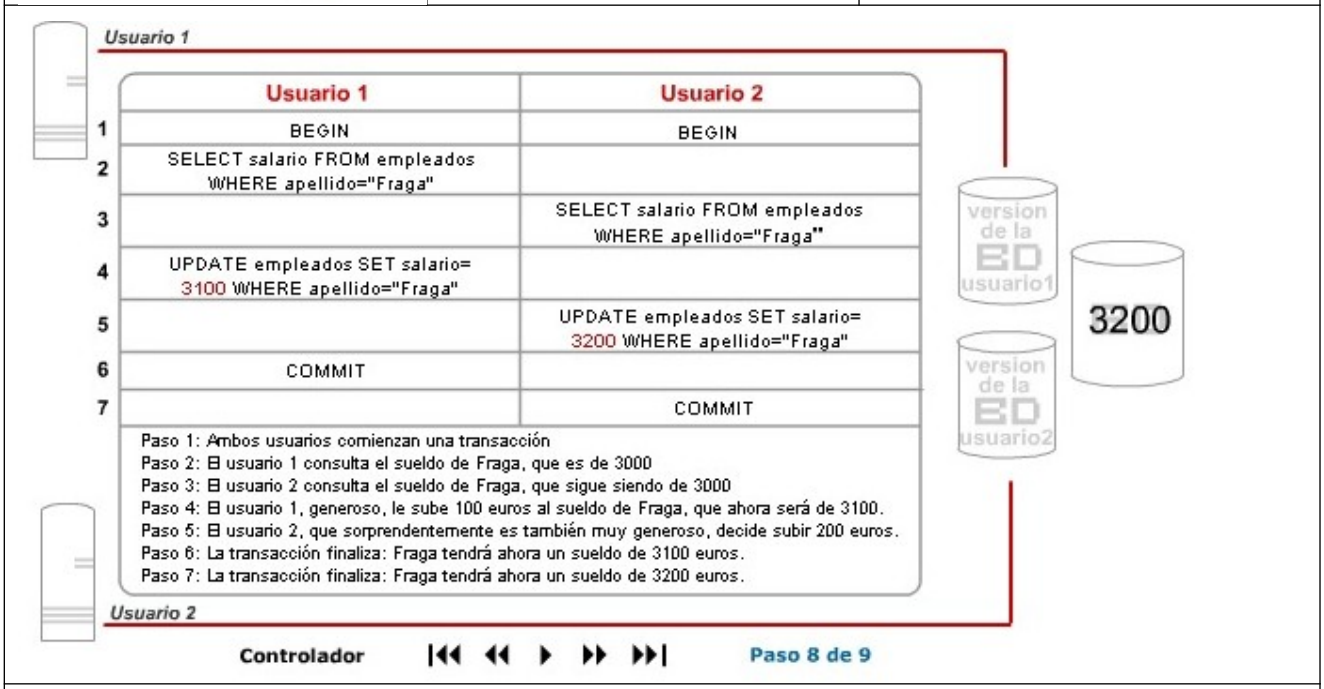
Las transacciones tienen el objetivo de lograr que una serie de comandos (y no sólo uno) cumplan dichas condiciones.

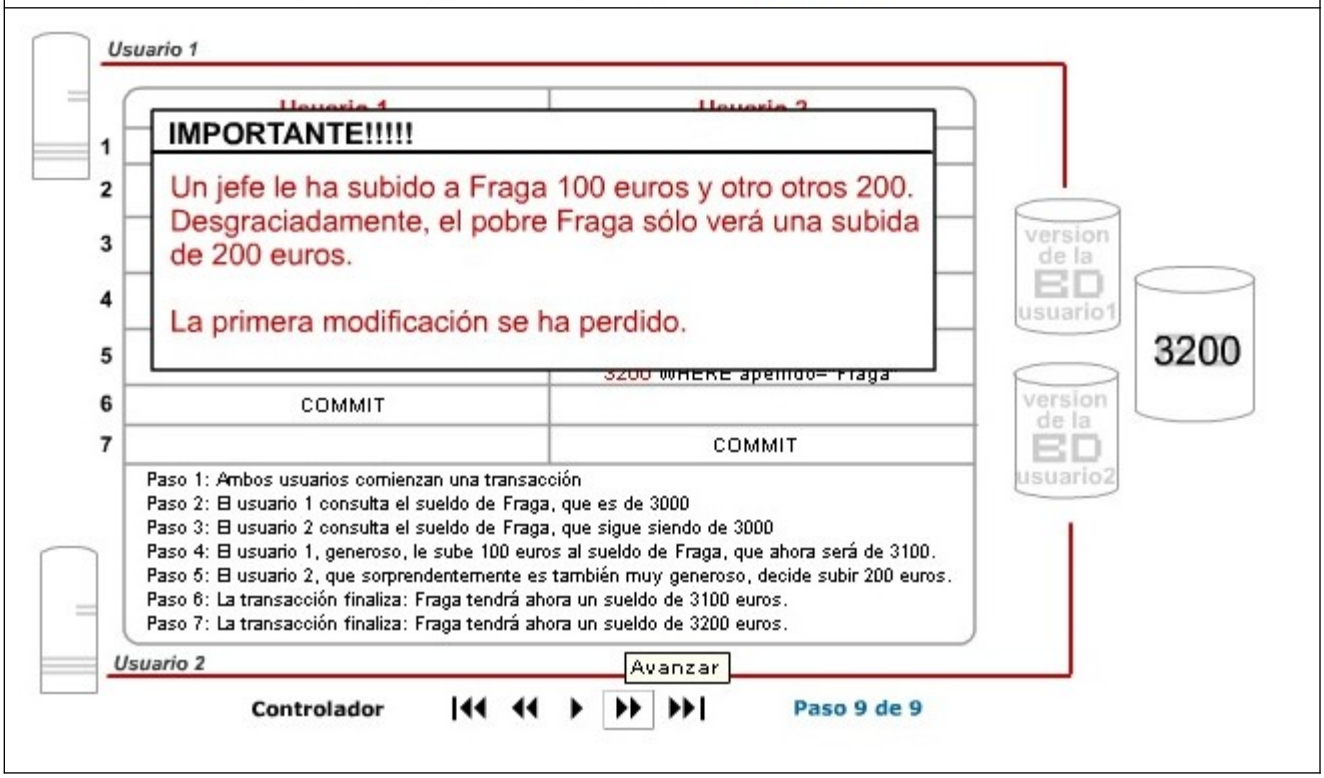
De las cuatro condiciones ACID, es la de aislamiento la que produce varios problemas al mecanismo de transacciones. Son los siguientes:

- **Problema de la modificación perdida**: surge cuando dos o más transacciones acceden a la misma fila y modifican su valor basándose en el valor original de la misma. Como cada transacción ignora la existencia del resto de transacciones, la última modificación sobreescribe las modificaciones realizadas por las otras transacciones.
- **Lectura sucia:** ocurre cuando una transacción modifica una fila y una segunda transacción lee esa fila antes de que la primera transacción comprometa el cambio. Si la primera transacción retrocede y deshace el cambio, la información leída por la segunda transacción se vuelve incorrecta.
- **Lectura no repetible:** sucede cuando una transacción lee una fila y una segunda transacción modifica esa fila. Si la segunda transacción confirma finalmente ese cambio, las siguientes lecturas de la primera transacción producen resultados diferentes al de la primera lectura.
- **Lectura fantasma:** se produce cuando una transacción lee un conjunto de filas que satisfacen una condición de búsqueda y, después, una segunda transacción modifica los datos. Si la primera transacción repite la lectura con las mismas condiciones de búsqueda, el resultado será distinto.

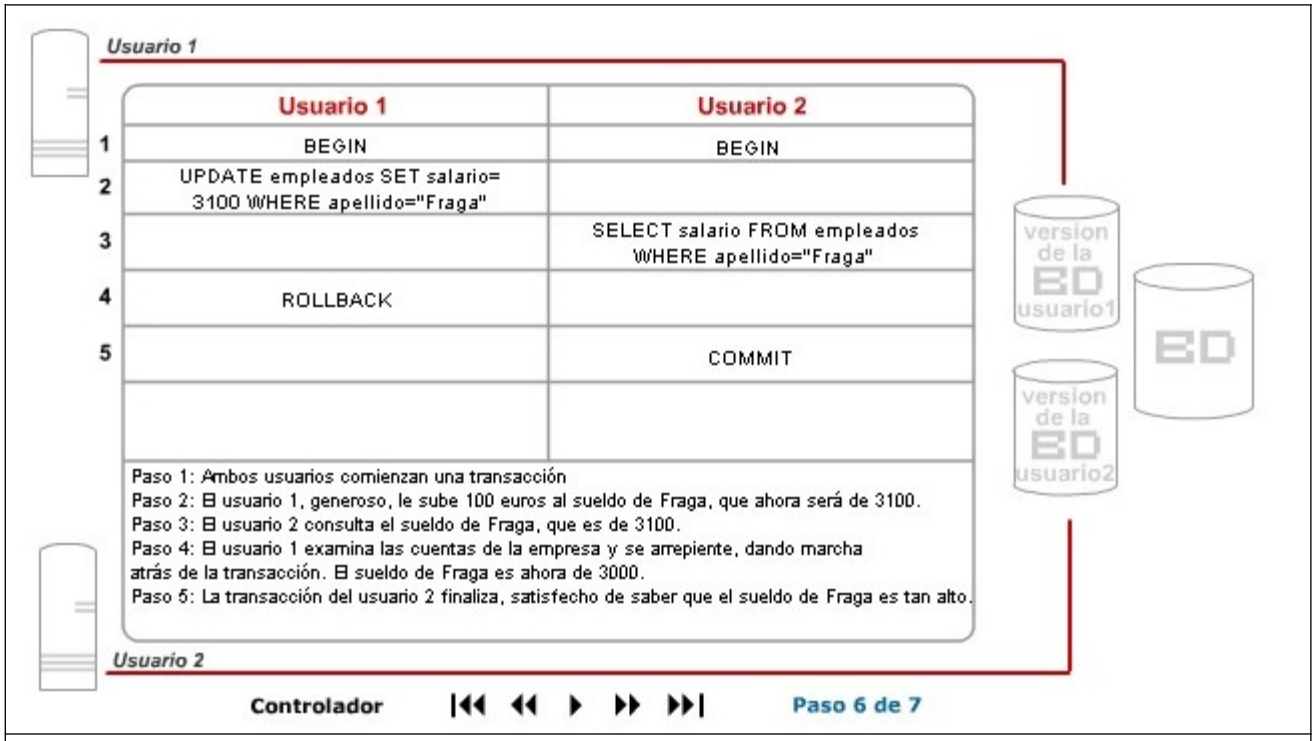
1.1.1 Problema de la modificación perdida.

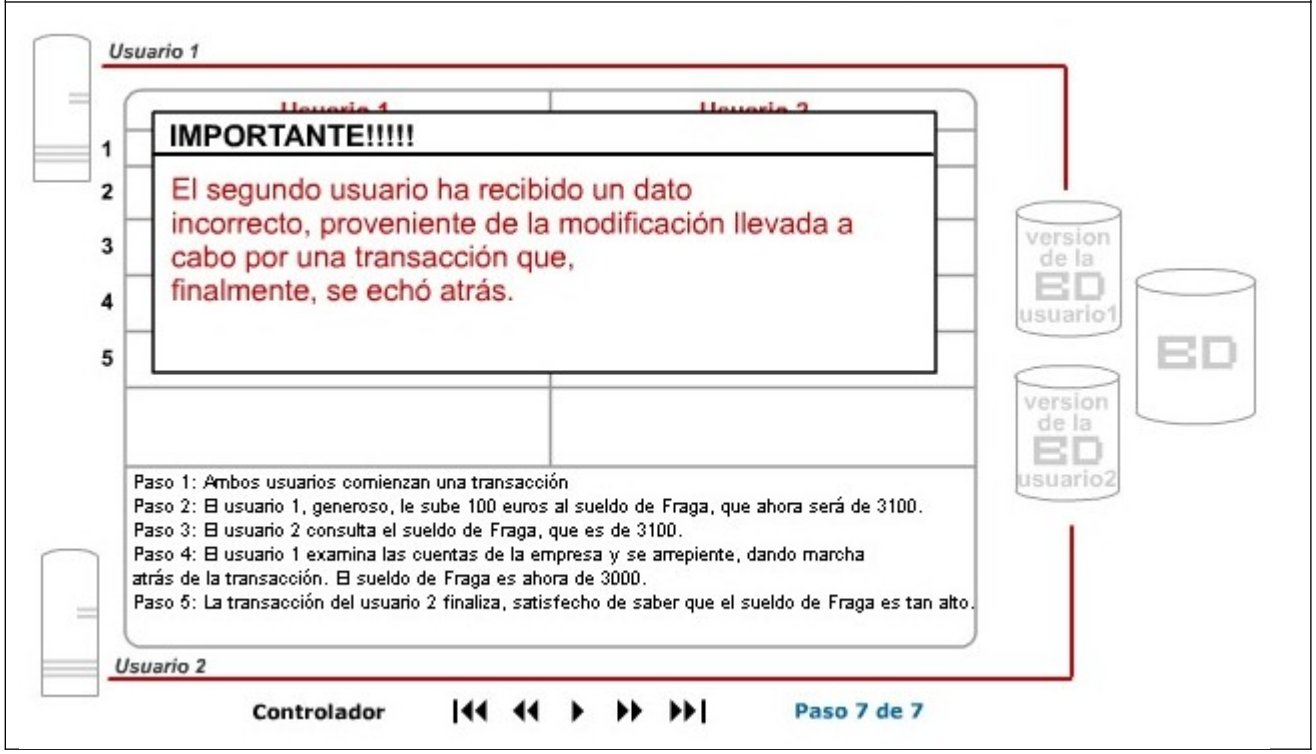




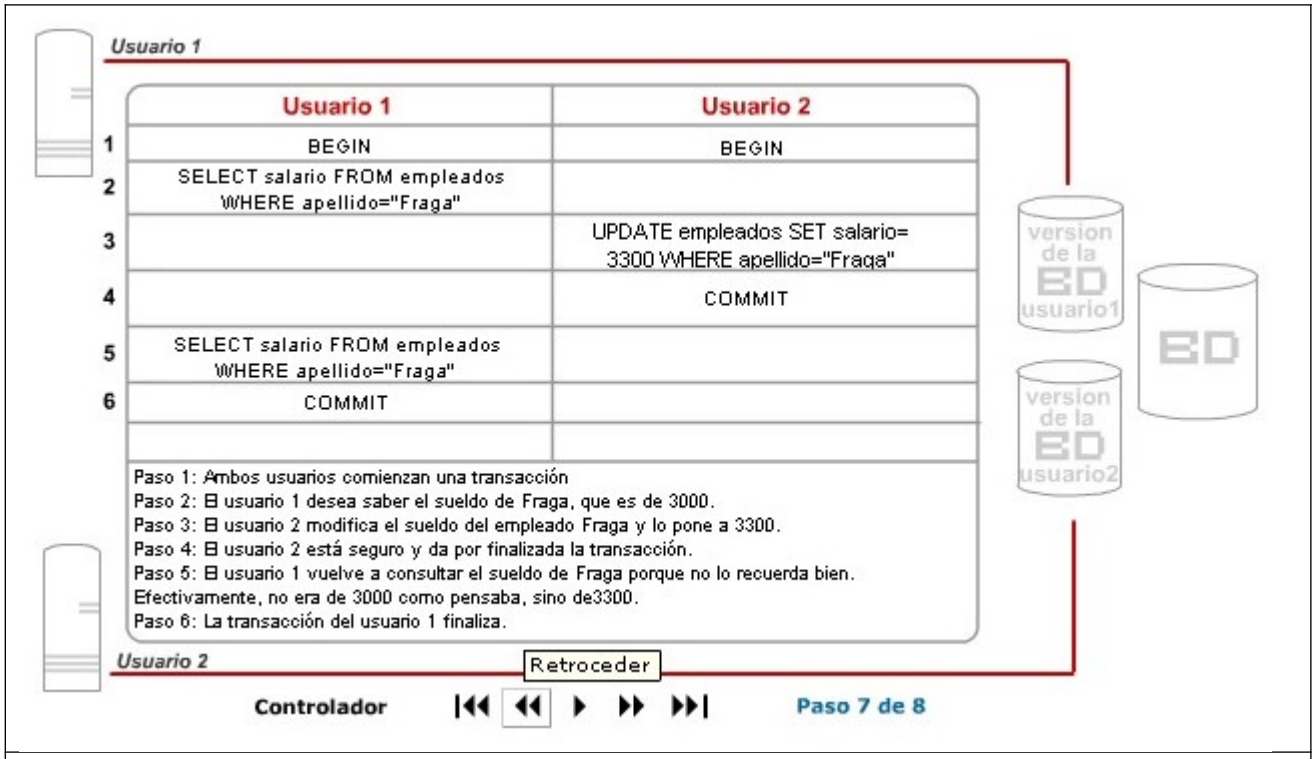


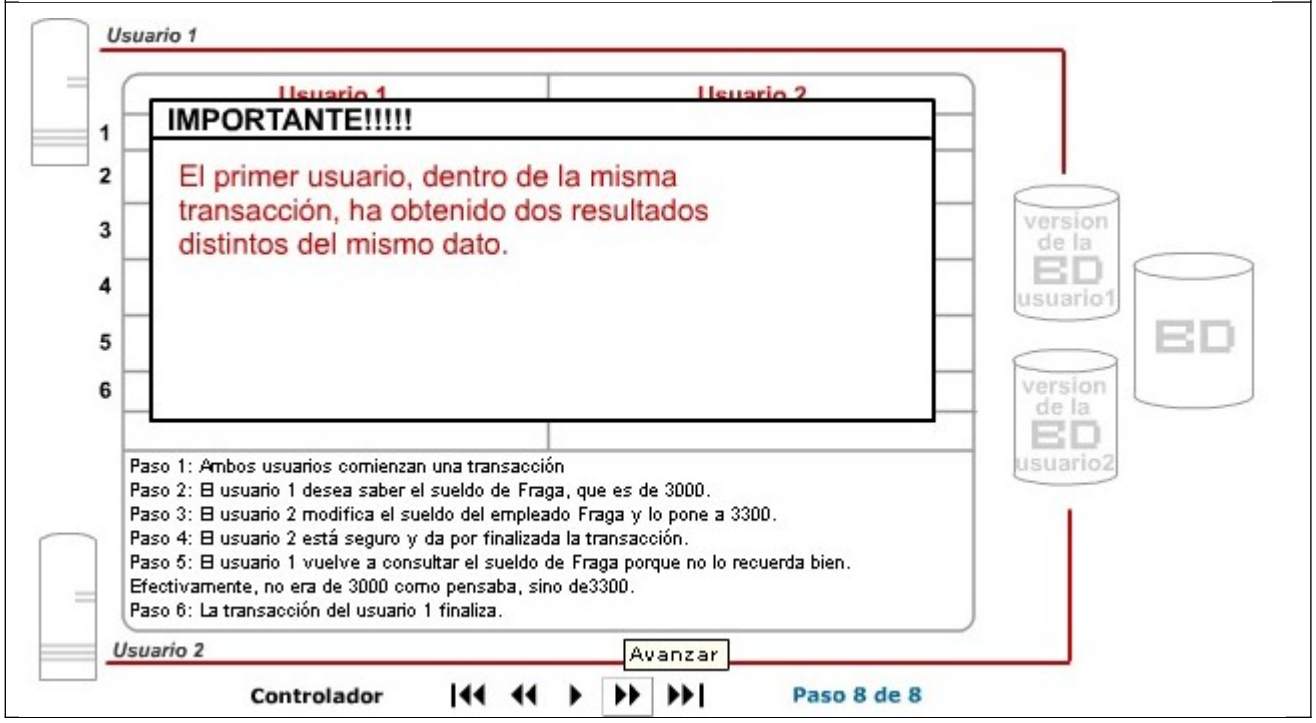
1.1.2 Lectura sucia.



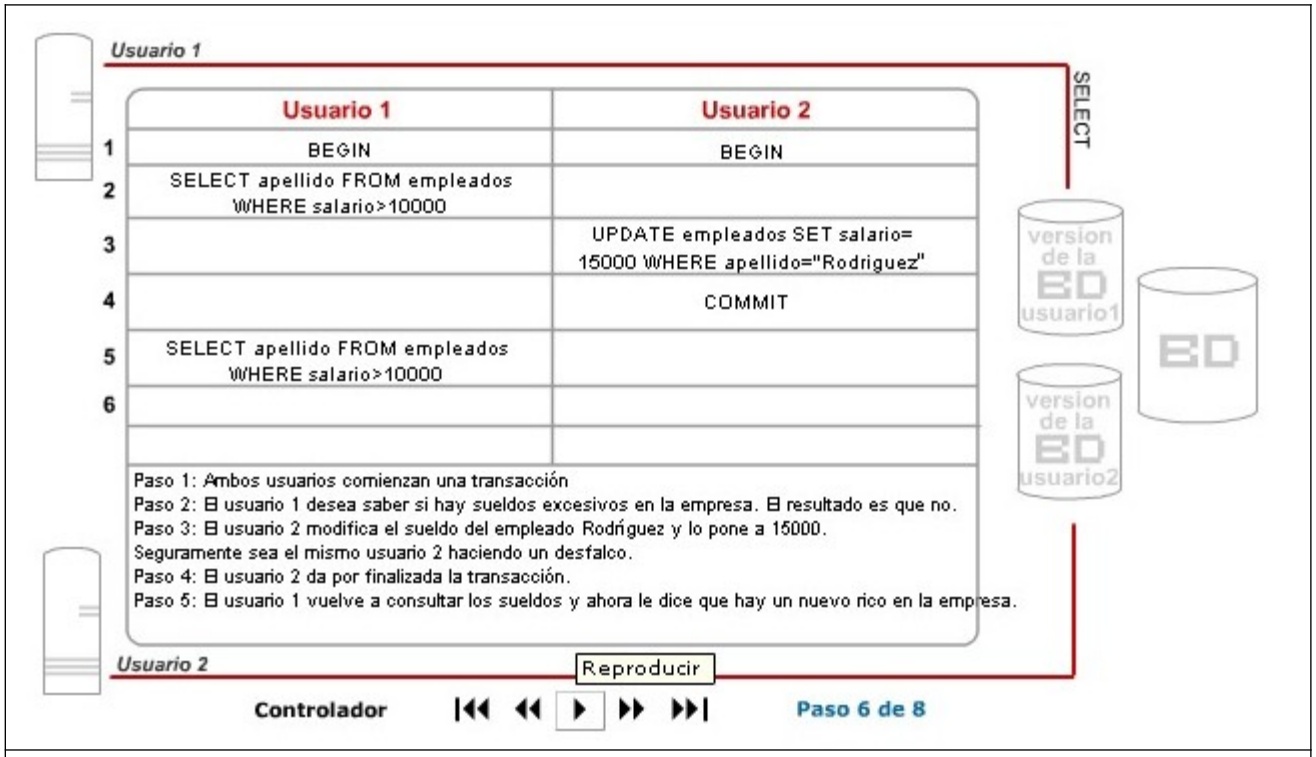


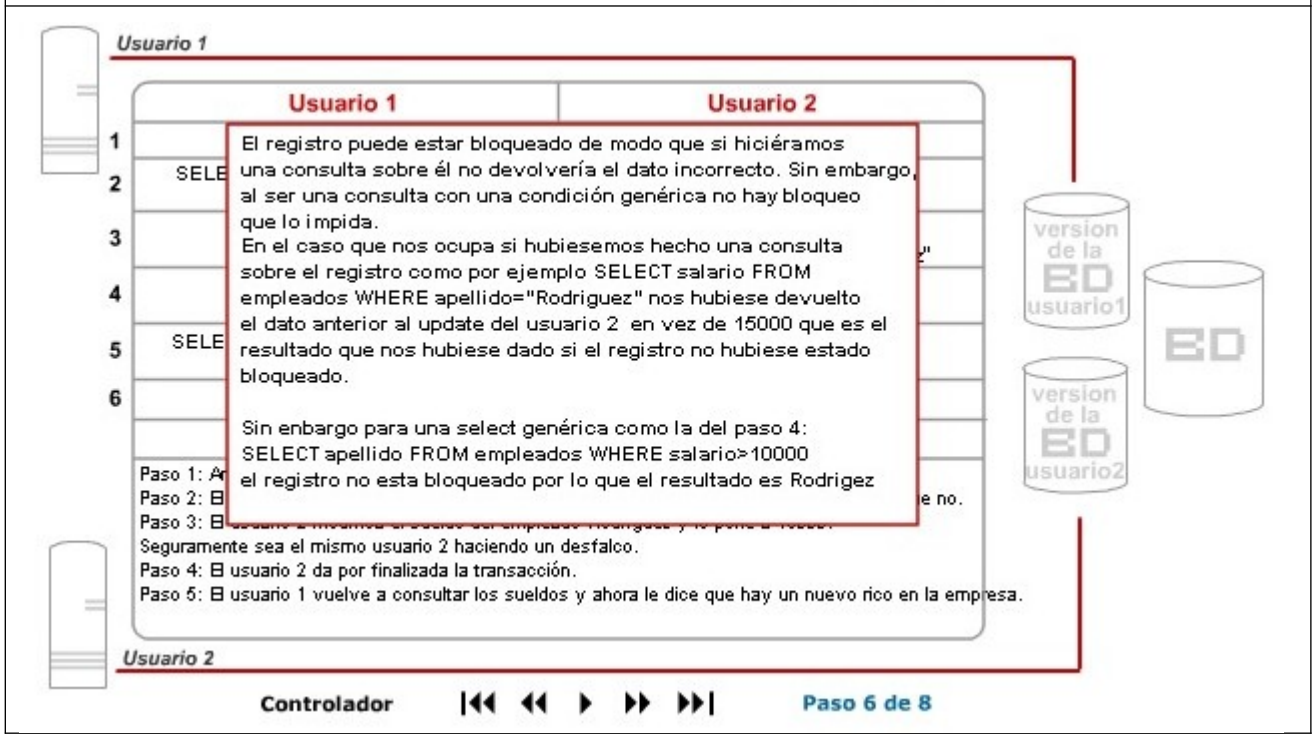
1.1.3 Lectura no repetible.

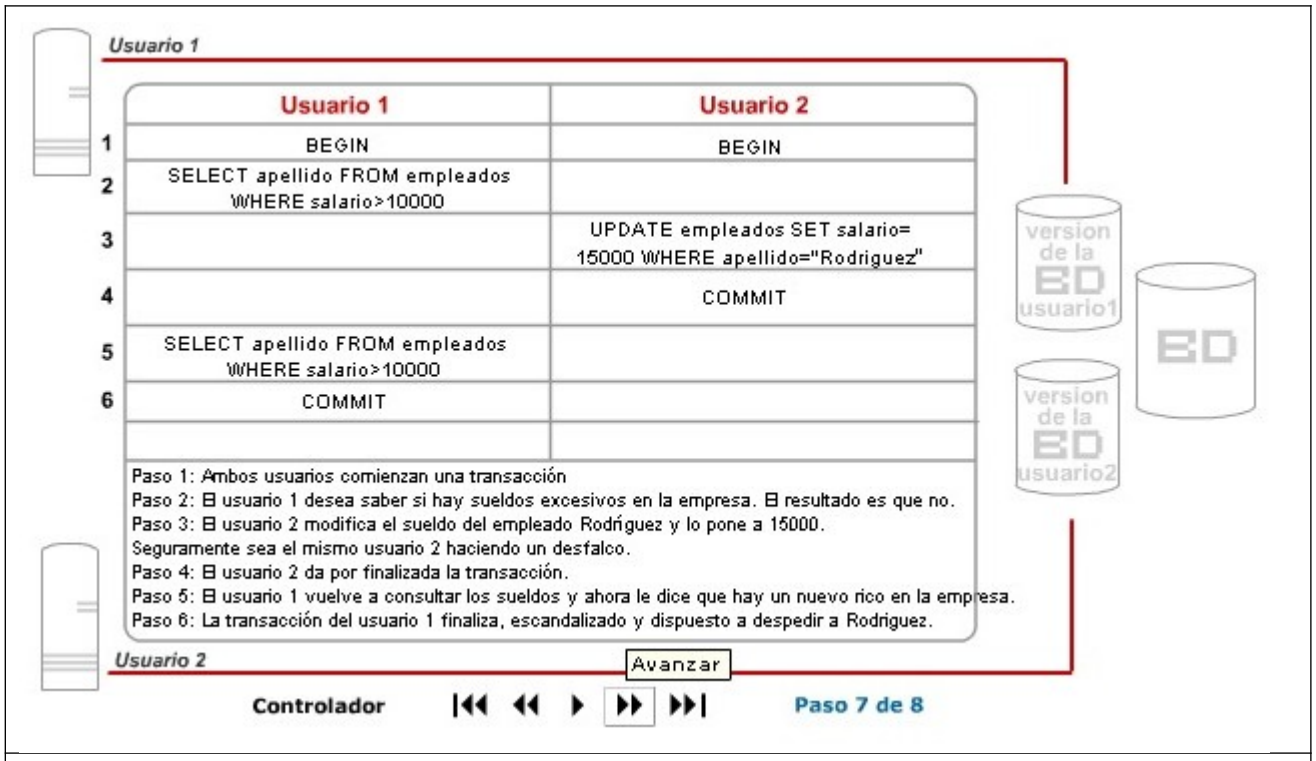


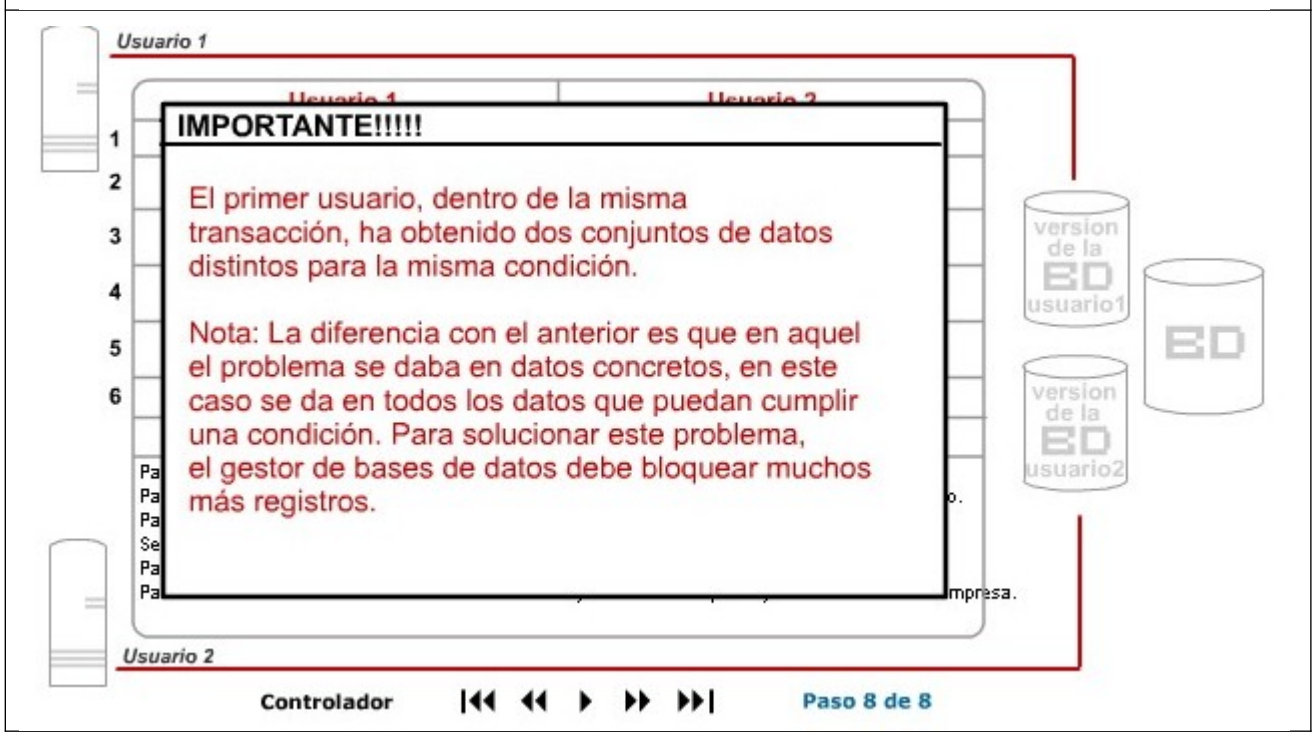


1.1.4 Lectura fantasma.






Usuario 1
Usuario 1
Usuario 2
1 BEGIN BEGIN
2 SELECT apellido FROM empleados WHERE salario>10000
3 UPDATE empleados SET salario= 15000 WHERE apellido="Rodriguez"
4 COMMIT
5 SELECT apellido FROM empleados WHERE salario>10000
6 COMMIT
Paso 1: Ambos usuarios comienzan una transacción
Paso 2: El usuario 1 desea saber si hay sueldos excesivos en la empresa. El resultado es que no.
Paso 3: El usuario 2 modifica el sueldo del empleado Rodríguez y lo pone a 15000.
Seguramente sea el mismo usuario 2 haciendo un desfalco.
Paso 4: El usuario 2 da por finalizada la transacción.
Paso 5: El usuario 1 vuelve a consultar los sueldos y ahora le dice que hay un nuevo rico en la empresa.
Paso 6: La transacción del usuario 1 finaliza, escandalizado y dispuesto a despedir a Rodriguez.
Usuario 2
Avanzar
version de la BD usuario1
version de la BD usuario2
BD
Controlador
Paso 7 de 8


Usuario 1
IMPORTANTE!!!!!
El primer usuario, dentro de la misma transacción, ha obtenido dos conjuntos de datos distintos para la misma condición.
Nota: La diferencia con el anterior es que en aquel el problema se daba en datos concretos, en este caso se da en todos los datos que puedan cumplir una condición. Para solucionar este problema, el gestor de bases de datos debe bloquear muchos más registros.
Usuario 2
version de la BD usuario1
version de la BD usuario2
BD
Controlador
Paso 8 de 8

1.1.5 Niveles de aislamiento.

Cumplir a rajatabla la regla “I” de aislamiento solucionando todos los problemas que hemos visto requeriría, posiblemente, un número de bloqueos demasiado alto. Por eso, se llega a una regla de compromiso que facilite un aislamiento relativo sin grandes penalizaciones de rendimiento. El estándar ANSI SQL-92 detalla cuatro de esas soluciones de compromiso, llamadas niveles de aislamiento. En MySQL se cambian por medio del siguiente comando:

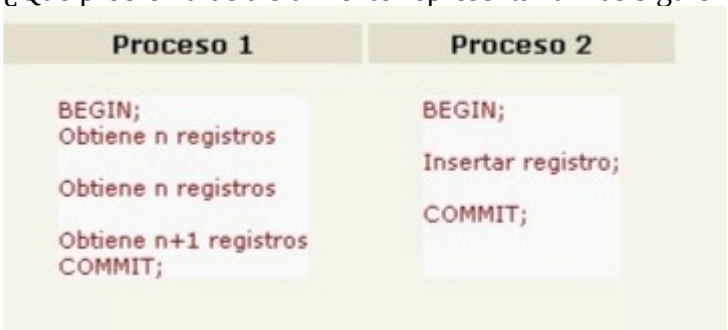
```
SET [ambito]
TRANSACTION ISOLATION LEVEL nivel
```

El ámbito puede ser GLOBAL o SESSION. En el primer caso el nivel se cambia para todas las nuevas transacciones, en el segundo en todas las transacciones de la conexión. Si no se indica ámbito alguno, el nuevo nivel sólo afectará a la siguiente transacción. El nivel puede ser uno de los descritos en la tabla de la derecha, ordenados de menor a mayor aislamiento.

Nivel	Descripción
READ UNCOMMITTED	Este nivel permite a las transacciones leer los datos actualizados por otra transacción aún sin terminar, permitiendo lecturas sucias. Dado que no bloquea nada no aísla, siendo el más rápido.
READ COMMITTED	Este es el nivel empleado por defecto en Oracle o SQL Server. No permite lecturas sucias, pues bloquea todos los registros actualizados por la transacción. No impide el problema de las lecturas no repetibles ni el de las fantasma.
REPEATABLE READ	Este es el nivel empleado por defecto en InnoDB. Soluciona el problema de los datos repetibles pero no el de los fantasmas.
SERIALIZABLE	Evita todos los problemas de aislamiento pero, aparte de reducir el rendimiento, puede provocar la aparición de interbloqueos (un problema de concurrencia que provoca que una transacción no pueda finalizar nunca debido a que otra lo está bloqueando indefinidamente).

Ejercicio: 36.

¿Qué problema de aislamiento representarían las siguientes transacciones?



Es un problema de lectura fantasma, dado que estamos obteniendo n registros y los registros nuevos aparecen de repente en medio de una transacción.

1.2 Transacciones en InnoDB.

Una transacción tendrá una sentencia **START TRANSACTION** o **BEGIN** WORK para marcar el comienzo de la misma y otra sentencia **COMMIT** para indicar el final. Sólo cuando COMMIT es procesado, los cambios indicados en las consultas se harán permanentes. Si algo va mal entre medias podemos usar el comando **ROLLBACK** para volver al punto donde estábamos antes de comenzar la transacción.

Vamos a ver cómo funciona en la práctica en tablas InnoDB. Para ello, vamos a crear una tabla de prueba en la que insertaremos un dato:

```
CREATE TABLE prueba_trans (
  c1 INT, c2 CHAR(10),
  INDEX (c1)
  ) TYPE = InnoDB;

INSERT INTO prueba_trans (c1) VALUES (1);
```

Si hacemos una consulta veremos un sólo dato en la consulta. Ahora es cuando deberemos empezar a hacer pruebas con transacciones. Insertamos un nuevo dato dentro de una transacción y consultamos la tabla:

```
BEGIN;
INSERT INTO prueba_trans (c1) VALUES (2);
SELECT c1 FROM prueba_trans;
```

Veremos que aparecen ambos registros. Pero si ahora escribimos ROLLBACK; y volvemos a repetir la consulta la última inserción habrá desaparecido. Del mismo modo, si volvemos a iniciar una transacción, insertamos la columna, salimos de la sesión y volvemos a entrar, veremos que la tabla no se ha actualizado. **Si se pierde la conexión durante una transacción, será equivalente a un ROLLBACK.**

Por otro lado, parece que estemos viendo sólo los casos en los que las transacciones no se llevan a cabo. Vamos a insertar por tercera vez el famoso registro, pero esta vez completando la transacción:

```
BEGIN;
INSERT INTO prueba_trans (c1) VALUES (2);
C0MMIT;
```

Si salimos de MySql y volvemos a entrar, veremos que en la consulta se nos mostrarán esta vez dos registros.

Nota: Existen otros comandos que, al ejecutarse, dan por finalizada la transacción abierta. Son: **ALTER TABLE**, **CREATE INDEX**, **RENAME TABLE**, **TRUNCATE**, **DROP TABLE** y **DROP DATABASE**.

1.2.1 Lecturas consistentes.

El comportamiento por defecto de las tablas InnoDB es el de las lecturas consistentes. Esto significa que, cuando se realiza una consulta SELECT, MySQL devuelve todos los valores presentes en la base de datos hasta la transacción más reciente completada. Si hay transacciones realizándose en ese momento, cualquier INSERT o UPDATE() no será reflejado en las consultas. Esto tiene una excepción, como vimos en el ejemplo anterior. **La transacción en proceso si que puede ver los cambios reflejados.**

Como vemos, MySQL ofrece una versión distinta de la base de datos a cada transacción. A este comportamiento se le llama control de concurrencia multiversión o MVCC en sus siglas en inglés.

Para hacer la prueba, abre una ventana de MySql Query Browser y ejecuta allí la transacción (sin terminar):

```
BEGIN;
INSERT INTO prueba_trans (c1) VALUES (3);
```

Si ahora abrimos otra sesión distinta (es decir, ejecutamos una segunda instancia), y hacemos una consulta en la misma, veremos que sólo aparecen reflejados los dos primeros registros. En cambio, si realizamos la misma consulta en la primera ventana veremos que incluye el tercer registro.

Si hacemos un COMMIT; en esa ventana, podremos ver como en la otra, al realizar de nuevo la consulta, ya aparece ese tercer registro.

Si estamos empleando el nivel por defecto de InnoDB (REPEATABLE READ), todas las lecturas dentro de la misma transacción devuelven el mismo “fotograma” de la base de datos: el que existía en la primera lectura de la transacción. En cambio, si el nivel es READ COMMITABLE ese fotograma será el que existía al comienzo de la transacción.

1.2.2 Bloqueo exclusivo. SELECT ….FOR UPDATE

Las lecturas consistentes no siempre son lo que se necesita. Por ejemplo, ¿que sucedería si hay más de un usuario intentando insertar un registro en la tabla prueba_trans a la vez? Supongamos que deseamos insertar en nuestras dos sesiones de MySQL un nuevo registro en el que el valor del campo c1 ha de ser el siguiente al mayor ya existente. Para ello tendríamos que saber cual es el valor máximo:

```
BEGIN;
SELECT c1 FROM prueba_trans;
```

Si ejecutamos esta consulta en ambas ventanas, nos devolverán ambas los tres registros, siendo el valor del más alto el número 3. El siguiente es 4. Pero, si insertamos en ambas ventanas ese número:

```
INSERT INTO prueba_trans (c1) VALUES (4);
C0MMIT;
```

El resultado será que tendremos dos registros con el valor 4. Para evitarlo podemos emplear un bloqueo exclusivo, que se emplea cuando una transacción desea actualizar datos, impidiendo a las demás el acceso a los mismos. Al indicar a MySQL que lo que estamos leyendo lo leemos para

emplearlo en una actualización, no dejará a ninguna otra transacción leer ese valor hasta que la bloqueadora haya finalizado.

Para comprobarlo, primero borramos los registros insertados y luego ejecutamos en ambas ventanas:

```
BEGIN;
SELECT c1 FROM prueba_trans FOR UPDATE;
```

La segunda ventana quedará a la espera mientras que la primera devolverá los datos, Para desbloquearla finalizamos la transacción:

```
INSERT INTO prueba_trans (ci) VALUES (4);
COMMIT;
```

En ese momento MySQL nos devolverá los datos a la transacción bloqueada, y entre ellos estará el número 4. De modo que ya podemos insertar el 5 en la segunda ventana.

Nota: Algún estudiante avispado se habrá dado cuenta de que quizá sería una consulta más lógica `SELECT MAX(c1) FROM prueba_trans;`. Tiene razón, pero esta consulta no funciona correctamente si la probáis con FOR UPDATE. Esto es debido a que, al hacer una consulta "normal", InnoDB bloquea todos los registros leídos por dicha consulta. En cambio, al hacer una consulta con agregado no se bloquea ninguno y el resultado no es consistente.

1.2.3 Bloqueo compartido. SELECT … LOCK IN SHARE MODE

Acabamos de ver **un bloque exclusivo de lectura: se le llama así porque sólo una transacción puede tener a la vez un bloqueo de este tipo**. Si otra transacción desea imponer ese bloqueo, debe esperar a que la primera deje de bloquear. Sin embargo, existe otro tipo de bloqueo, llamado bloqueo compartido, en el que varias transacciones pueden bloquear conjuntamente.

Este tipo de bloqueo impide modificar datos bloqueados a nuevas transacciones y espera a que finalicen transacciones que ya hayan cometido esa felonía para poder devolver la versión más reciente de los datos. Vamos a ver un ejemplo. Escribamos **en la ventana 1**:

```
BEGIN;
INSERT INTO prueba_trans (c1) VALUES (6);
UPDATE prueba_trans SET c2=”Sebastián” WHERE c1=6;
```

Si hacemos una consulta ordinaria en la ventana número 2 no obtendremos el valor actualizado, porque, si seguimos las reglas de aislamiento, debe ignorar las modificaciones .Compruébalo:

```
SELECT MAX(ci) FROM prueba_trans;
```

Sin embargo si realizamos una consulta con bloqueo compartido (empleando la cláusula LOCK IN SHARE MODE, no obtendremos resultado alguno hasta que la transacción de la ventana 1 finalice:

```
SELECT MAX(c1) FROM prueba_trans LOCK IN SHARE MODE;
```

Si hacemos COMMIT; en la ventana 1 la segunda devolverá el resultado correcto.

1.2.4 COMMIT automático. Por defecto cada sentencia es una transacción.

Por defecto, **a no ser que se especifique el inicio de una transacción** con START TRANSACTION, MySQL ejecuta directamente las sentencias individuales que se le mandan. Este comportamiento puede modificarse (siempre que las tablas soporten transacciones, claro, las tablas Innodb soportan transacciones y bloqueo de registros) poniendo a cero el parámetro AUTOCOMMIT. Pongamos en la ventana 1:

```
SET AUTOCOMMIT=0; /* desactivamos el valor por defecto, cada
orden o sentencia es una transacción.*/

SELECT c1 FROM prueba_trans;
```

Ahora, en la ventana 2, insertamos un nuevo valor:

```
INSERT INTO prueba_trans (c1) VALUES (8);
```

Si volvemos a consultar en la ventana 1 veremos que no se ha actualizado con este nuevo valor. Eso es debido a que la inserción se considera que está ahora en una transacción. Si en la ventana 2 hacemos COMMIT veremos que una nueva consulta devuelve ahora todos los valores.

Podemos volver al modo normal poniendo a 1 el AUTOCOMMIT:

```
SET AUTOCOMMIT=1;
```

1.2.5 Bloqueo de tablas.

Durante todo el capítulo hemos estado viendo el funcionamiento del bloqueo a nivel de registro. Pero éste sólo existe en las tablas InnoDB o BDB. MySQL dispone de otro tipo de bloqueo, el bloqueo de tablas, disponible en todos los tipos de tabla incluyendo MyISAM. Existen dos tipos de bloqueo: de lectura y de escritura. Los primeros permiten que se sigan realizando consultas a la tabla bloqueada pero impide cualquier escritura. Los segundos impiden cualquier tipo de acceso a otros procesos. La sintaxis es:

```
LOCK TABLE nombretabla READ|WRITE
```

Los bloqueos se eliminan todos a la vez de la siguiente manera:

```
UNLOCK TABLES
```

Vamos a ver como funciona en la práctica. En la ventana 1 vamos a poner un bloqueo:

```
LOCK TABLE departamentos2 READ;
```

Ahora en la ventana 2 realizaremos una consulta y una inserción. Veremos que la primera cuela, pero la segunda no:

```
SELECT * FROM departamentos2;
```

```
INSERT INTO departamentos2
VALUES (70, ‘Dirección”, lvloncloa”);
```

La ventana 2 quedará a la espera hasta que en la primera desbloqueemos la tabla:

```
UNLOCK TABLES
```

Consejo 2: Conviene evitar bloquear tablas.

Hay que evitar en la medida de lo posible el bloqueo de tablas en tablas en las que se realicen muchas actualizaciones dado que, en el caso de un bloqueo de escritura, no se podrán leer o escribir registros durante la duración del bloqueo. Y como los bloqueos de escritura tienen prioridad sobre los de lectura, ningún registro se podrá leer hasta que se hayan completado todas las inserciones y actualizaciones se hayan completado, lo que puede provocarle muchos apuros a MySQL.

Una posible manera de evitarlos es realizar la lectura y la escritura a la vez empleando actualizaciones incrementales:

Ejemplo:

Incrementar el salario de Fraga en 10 euros (estamos generosos).

```
UPDATE empleados SET salario=salario+10 WHERE
apellido=’Fraga’;
```