UNIDAD 8 MANIPULACIÓN DE LOS DATOS

1 Introducción

2 Inserción de datos

3 Modificación de datos

4 Borrado de datos

5 Actualizaciones con subconsultas

6 Transacciones

6.1 ¿Qués es una transacción?

6.2 Propiedades ACID

6.3 Comandos de transacción en MySQL

6.4 Estados de los datos durante la transacción

6.5 Niveles de aislamiento. Acceso concurrente a los datos

1 Introducción

El lenguaje SQL incluye un conjunto de sentencias que permiten modificar, borrar e insertar nuevas filas en una tabla. Este subconjunto de comandos del lenguaje SQL recibe la denominación de Lenguaje de Manipulación de Datos (DML, Data Manipulation Language)

Al conjunto de instrucciones DML que se ejecutan consecutivamente, se les llama transacciones y se pueden anular todas ellas o aceptar, ya que una instrucción DML no es realmente efectuada hasta que no se acepta **(COMMIT)**

2 Inserción de datos

SENTENCIA INSERT

Para añadir nuevas filas a una tabla utilizaremos la sentencia INSERT. Podemos insertar usando una sentencia INSERT normal, o usando un SELECT.

El formato típico de la sentencia INSERT es:

```
INSERT INTO nombre_tabla (lista_columnas) VALUES (lista_valores)
```

Donde:

- nombre_tabla: es el nombre de la tabla en la que queremos insertar una fila.
- lista_columnas: incluye las columnas cuyos valores queremos insertar separados por comas.
- lista_valores: indica los valores que se insertarán separados por comas.

Ejemplo 1: Insertar una nueva fila en la tabla árbitros de la base de datos liga

```
INSERT INTO arbitros (id_arbitro, nombre, apellidos, antigüedad, salario)
VALUES (15, 'Sara', 'Comin', 3, 1000)
```

Este formato de inserción tiene además las siguientes características:

- ❒ En la lista de columnas se pueden indicar todas o algunas de las columnas de la tabla. En este último caso, aquellas columnas que no se incluyan en la lista quedarán sin ningún valor en la fila insertada, es decir, se asumirá el valor NULL para las columnas que no figuran en la lista, siempre que en la creación de la tabla no hayamos especificado esa columna como NOT NULL.
- ❒ Los valores incluidos en la lista de valores deberán **corresponderse posicionalmente** con las columnas indicadas en la lista de columnas, de forma que el primer valor de la lista de valores se incluirá en la columna indicada en primer lugar, el segundo en la segunda columna, y así sucesivamente.
- ❒ Se puede utilizar **cualquier expresión** para indicar un valor, siempre que el resultado de la expresión **sea compatible con el tipo de dato de la columna** correspondiente.
- ❒ Los valores constantes de tipo carácter o fecha deben ir entre comillas simples.

También se puede insertar filas en una tabla sin especificar la lista de columnas, pero en este caso la lista de valores deberá coincidir en número y posición con las columnas de la tabla. El orden establecidos para las columnas será el indicado al crear la tabla (el mismo que aparece al dar una instrucción SELECT * FROM …)

El formato genérico cuando no se especifica la lista de columnas es:

```
INSERT INTO nombre_tabla VALUES (lista_valores)
```

Ejemplo 2: Instrucción INSERT sin lista de columnas

```
INSERT INTO arbitros VALUES (15, 'Sara', 'Comin', 3, 1000)
```

Cuando no se especifica lista de columnas, se deberán especificar valores para todas las columnas. Si en algún caso no queremos incluir ningún valor para una columna deberemos indicar explícitamente el valor NULL para dicha columna.

INSERT CON SELECT

Para insertar registros de otra tabla, la sintaxis es:

```
INSERT INTO tabla [IN base_externa] (campo1, campo2,…,campoN)
SELECT tabla_origen.campo1, tabla_origen.campo2,…, tabla_origen.campoN
FROM tabla_origen
```

La condición SELECT puede incluir la cláusula WHERE para filtrar los registros a copiar. Si tabla y tabla_origen tiene la misma estructura podemos simplificar la sintaxis a:

```
INSERT INTO tabla SELECT tablaorigen.* FROM tablaorigen;
```

Ejemplo 3: Copiar los datos de la tabla árbitros a una nueva tabla arbitros2

```
INSERT INTO arbitros2 SELECT arbitros.* FROM arbitros;
```

Ejemplo 4: Copiar los árbitros con una antigüedad mayor de 5 años a una nueva tabla arbitros_viejunos

```
INSERT INTO arbitros_viejunos SELECT arbitros.* FROM arbitros
WHERE antiguedad>5;
```

Lo anterior son las sentencias SQL estándar que os van a funcionar en cualquier SGBD que utilice SQL.

Veamos ahora alguna característica particular de MySQL.

INSERT EN MySQL

Esto que voy a poner a continuación es un resumen del manual de MySQL, en concreto del siguiente enlace https://dev.mysql.com/doc/refman/8.4/en/insert.html

Formato 1

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

Donde:

- **LOW PRIORITY**: hace que la inserción se retrase mientras haya clientes leyendo de la tabla. Sirve sólo para tablas que acepten bloqueos como son MyISAM, MEMORY y MERGE.
- **DELAYED**: permite continuar con otras operaciones mientras la inserción se retrasa hasta que

no haya clientes accediendo a la tabla, es decir, como la anterior, pero permite seguir trabajando. Esta opción está obsoleta, se acepta en MySQL 5.7, pero se ignora.

- **HIGH PRIORITY**: deshabilita el efecto de la variable de sistema – low-priority-updates si está activa (=1). Esta variable hace que las operaciones de modificación tengan menor prioridad que las de consulta.
- **IGNORE**: obvia los errores en la inserción. Por ejemplo, no podemos insertar registros con claves repetidas, pero no nos informará.
- **PARTITION**: especifica las particiones donde se pretende insertar los datos. Una partición sería una parte de la tabla. El particionado nos permite distribuir porciones de una tabla individual en diferentes segmentos conforme a unas reglas establecidas por el usuario.
- **DEFAULT**: sirve para establecer el valor del campo por defecto creado cuando se definió la tabla.
- **ON DUPLICATE KEY UPDATE**: cuando el valor de una clave (primaria PRIMARY KEY o secundaria UNIQUE) se repite, ésta cláusula permite actualizar uno o varios campos del registro correspondiente.

  Veamos un ejemplo de esto Suponer que tenemos una tabla con tres campos, a, b y c (a es la clave) y queremos insertar un nuevo registro (1, 2, 3), entonces si ya existe un registro con el valor del campo a igual a 1, se actualizarán los valores.

  ```
  INSERT INTO tabla (a, b, c) VALUES (1, 2, 3) ON DUPLICATE KEY UPDATE b=2, c=3;
  ```

Nota: para insertar valores autonuméricos ponemos cero, el sistema calculará automáticamente el valor numérico correspondiente.

Formato 2

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

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

Veamos un ejemplo:

Para insertar un nuevo jugador en la tabla jugadores de id 500, nombre Juan y el resto de valores por defecto (si los hubiéramos puesto en la creación de la tabla)

```
INSERT INTO jugadores SET id=500, nombre='Juan';
```

Formato 3

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

Insertamos los valores en la tabla utilizando una consulta SQL. Es una manera rápida de insertar valores en una tabla utilizando datos de otras tablas.

3 Modificación de datos

En ocasiones necesitaremos modificar alguno de los datos de las filas existentes en una tabla. Por ejemplo cambiar el salario de los empleados. Entonces usaremos la sentencia UPDATE cuyo formato genérico es el siguiente:

```
UPDATE nombre_tabla
SET nombre_columna = valor [,nombre_columna = valor, …]
[WHERE condicion]
```

Donde:

- nombre_tabla: es el nombre de la tabla donde se encuentran las filas que queremos modificar
- la cláusula SET va seguida de una o más asignaciones todas con el mismo formato nombre_columna=valor.
- nombre_columna, indica la columna cuyo valor queremos cambiar.
- valor, es una expresión que indica el nuevo valor para la columna. Se pueden incluir literales, variables del sistema, nombres de columnas, etc. No se pueden incluir funciones de agregado ni subconsultas.
- La cláusula WHERE permite especificar una condición de selección de las filas. En el caso de que no se utilice, la actualización afectará a todas la tabla.

Ejemplo 1: Modificar la posición del jugador cuyo número es el 15 cambiando la posición que tuviese por el de Base y además aumentar el salario del mismo a 300 euros.

```
UPDATE jugadores
SET salario = salario+300, posicion= 'base'
WHERE emp_no = 15;
```

La actualización anterior afecta a una única fila, pero podemos escribir comandos de actualización que afecten a varias filas.

Ejemplo 2: Aumentar el salario de todos los jugadores que sean pivot a 100

```
UPDATE jugadores
SET salario= salario + 100
WHERE posicion='pivot';
```

Si no se incluye la cláusula WHERE la actualización afectará a todas las filas.

Ejemplo 3: Modificar el salario de todos los jugadores incrementándolo en un 3 %.

```
UPDATE jugadores
SET salario = salario + salario*0.3;
```

UPDATE EN MYSQL

Esto que voy a poner a continuación es un resumen del manual de MySQL, en concreto del siguiente enlace https://dev.mysql.com/doc/refman/8.4/en/update.html

Sintaxis para una única tabla:

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

Con la clausula ORDER BY las filas son modificadas en el orden en el que se especifica. La clausula LIMIT establece un limite en el número de filas que pueden ser actualizadas.

Sintaxis para múltiples tablas:

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

4 Borrado de datos

Para eliminar o suprimir filas de una tabla utilizaremos la sentencia DELETE cuyo formato genérico es el siguiente:

```
DELETE FROM nombreTabla [WHERE condicion]
```

Donde:

- nombreTabla: indica la tabla destino donde se encuentran las filas que queremos borrar.
- Condición: permite especificar la condición que deben cumplir las filas que serán eliminadas. En el caso de que **no se especifique ninguna condición** el SGBD **asumirá que se desean eliminar todas las filas** de la tabla; en este caso no se eliminará la tabla pero quedará completamente vacía.

El borrado de un registro no puede provocar fallos de integridad. Hay que tener cuidado. La opción ON DELETE CASCADE hace que se borren todos los relacionados.

Podemos borrar todas las filas de una tabla con la sentencia TRUNCATE, internamente se implementa como un borrado de tabla y una creación de tabla.

```
TRUNCATE TABLE nombre_tabla;
```

DELETE EN MYSQL

Esto que voy a poner a continuación es un resumen del manual de MySQL, en concreto del siguiente enlace https://dev.mysql.com/doc/refman/8.4/en/delete.html

Sintaxis para una única tabla:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
    [PARTITION (partition_name,...)]
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Sintaxis para múltiples tablas:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    tbl_name[.*] [, tbl_name[.*]] ...
```

```
    FROM table_references
    [WHERE where_condition]

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

5 Actualizaciones con SELECT

Inserciones con consultas.

Podemos insertar en una tabla el resultado de una consulta sobre otra tabla. En este caso normalmente se insertarán varias filas con una sola sentencia. Utilizaremos el siguiente formato:

```
INSERT INTO nombretabla [( listacolumnas)] consulta;
```

En el formato anterior podemos destacar:

- La tabla que sigue a la cláusula INTO es, igual que en las ocasiones anteriores, **la tabla destino** en la que queremos insertar filas.
- La **lista de columnas es opcional** pero deberá especificarse cuando las columnas que devuelve la consulta no coinciden en número o en orden con las columnas de la tabla destino.
- La **consulta puede ser cualquier comando de selección** válido (salvo las restricciones indicadas abajo) siempre que exista una correspondencia entre las columnas devueltas y las columnas de la tabla destino, o la lista de columnas.

Supongamos que disponemos de la tabla *nequipos* cuyas columnas son *equipo*, *nombre* y *localidad*; y queremos insertar en ella las filas correspondientes a los equipos de la tabla *equipos* cuyo nombre tiene más de 8 letras.

En este caso existe correspondencia en número y en orden entre las columnas de la tabla destino y las columnas de la selección; por tanto, no hace falta especificar la lista de columnas y el comando requerido será:

```
INSERT INTO nequipo
SELECT * FROM equipos
WHERE LENGTH(nombre)> 8;
```

Si la tabla destino tuviese una estructura diferente deberemos forzar la correspondencia, bien al especificar la lista de selección, bien especificando la lista de columnas, o bien utilizando ambos recursos.

Por ejemplo, supongamos que la tabla destino es *n2equipo* cuyas columnas son *nombre* y *localidad*. En este caso procederemos:

```
INSERT INTO n2equipo
SELECT nombre, localidad FROM equipos
WHERE LENGTH(nombre)> 8;
```

O bien:

```
INSERT INTO n2equipo (nombre, localidad) SELECT nombre, localidad
FROM equipos WHERE LENGTH(nombre)> 8;
```

Restricciones: Además de la necesidad de coincidencia en número y compatibilidad en tipo de las columnas, el estándar establece algunas restricciones para la cláusula INSERT multifila:

- La tabla destino no puede aparecer en la consulta.
- La consulta no puede ser una UNION.
- No se puede incluir una cláusula ORDER BY en la consulta.

En la práctica la mayoría de los productos comerciales permiten saltar estas restricciones.

Modificación y eliminación de filas con subconsultas en la cláusula WHERE.

En ocasiones la condición que deben cumplir las filas que deseamos eliminar o modificar implica realizar una subconsulta a otras tablas. En estos casos se incluirá la subconsulta en la condición y los operadores necesarios tal como estudiamos en el apartado correspondiente del tema SUBCONSULTAS.

Por ejemplo, supongamos que se desea elevar en 500 euros el salario de todos los jugadores cuyo equipo no esté en Madrid.

```
UPDATE jugadores SET salario = salario + 500
WHERE equipo NOT IN
      (SELECT equipo FROM equipos
            WHERE localidad = 'MADRID');
```

6 Transacciones

6.1 ¿Qué es una transacción?

Una transacción en un Sistema de Gestión de Bases de Datos es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica.

Se dice que un SGBD es transaccional si es capaz de mantener la integridad de los datos, haciendo que estas transacciones no puedan finalizar en un estado intermedio. Cuando por alguna causa el sistema debe cancelar la transacción, empieza a deshacer las órdenes ejecutadas hasta dejar la base de datos en su estado inicial (llamado punto de integridad), como si la orden de la transacción nunca se hubiese realizado.

Una transacción está formada por una serie de instrucciones DML. Una transacción comienza con la primera instrucción DML que se ejecute y finaliza con alguna de estas circunstancias:

- Una operación COMMIT o ROLLBACK
- Una instrucción DDL (como ALTER TABLE por ejemplo).
- Una instrucción DCL (como GRANT).
- El usuario abandona la sesión.
- Caída del sistema.

Hay que tener en cuenta que cualquier instrucción DDL o DCL da lugar a un COMMIT implícito, es decir todas las instrucciones DML ejecutadas hasta ese instante pasan a ser definitivas.

6.2 Propiedades ACID

Las transacciones confiables deben ser compatibles con estas cuatro propiedades.

- **Atomicidad:** Una transacción es indivisible, o se ejecutan todas las sentencias o no se ejecuta ninguna.
- **Consistencia:** Después de una transacción la base de datos estará en un estado válido y consistente.
- **Aislamiento:** garantiza que cada transacción está aislada del resto de transacciones y que el acceso a los datos se hará de forma exclusiva.
- **Durabilidad:** los cambios que realiza una transacción sobre la base de datos son permanentes.

6.3 Comandos de transacción en MySQL

START TRANSACTION (o BEGIN)

Esta instrucción comienza una nueva transacción

COMMIT

La instrucción COMMIT hace que los cambios realizados por la transacción sean definitivos e irrevocables. Sólo se debe utilizar si estamos de acuerdo con los cambios, conviene asegurarse mucho antes de realizar el COMMIT ya que las instrucciones ejecutadas pueden afectar a miles de registros.

Cualquier instrucción DDL o DCL da lugar a un COMMIT implícito, es decir, todas las instrucciones DML ejecutadas hasta es instante pasan a ser definitivas.

Además el cierre correcto de la sesión da lugar a un COMMIT, aunque siempre conviene ejecutar explícitamente esta instrucción a fin de asegurarnos de los que hacemos.

ROLLBACK

Esta instrucción regresa a la instrucción al inicio de la transacción, normalmente el último COMMIT, la última instrucción DDL o DCL o al inicio de sesión. Anula definitivamente los cambios, por lo que conviene también asegurarse de esta operación.

Un abandono de sesión incorrecto o un problema de comunicación o de caída del sistema da lugar a un ROLLBACK ímplicito.

SAVEPOINT

Establece un punto de guardado dentro de una transacción, permitiendo revertir a un punto específico en lugar de toda la transacción.

Sintaxis: SAVEPOINT nombre

Para regresar a un punto de ruptura concrero:

- ROLLBACK TO SAVEPINT nombre
- ROLBACK TO nombre

Cuando se vuelve a un punto marcado, las instrucciones que siguen a esa marca se anulan definitivamente.

AUTOCOMMIT

Automáticamente se aceptan todos los cambios realizados después de la ejecución de una senetncia

SQL y no es posible deshacerlos.

En ORACLE

SQL*Plus e iSQL*Plus permiten validar automáticamente las transacciones sin tener que indicarlo de forma explícita. Para eso sirve el parámetro AUTOCOMMIT. El valor de ese parámetro se puede mostrar con la orden SHOW de la siguiente manera:

```
SHOW AUTOCOMMIT;
```

Por defecto, el valor es OFF, de manera que las transacciones (INSERT, UPDATE y DELETE) no son definitivas hasta que no hagamos COMMIT. Si queremos que INSERT, UPDATE y DELETE tengan un carácter definitivo sin necesidad de realizar la validación COMMIT, hemos de activar el parámetro AUTOCOMMIT con la **orden SET**:

```
SET AUTOCOMMIT ON;
```

En MYSQL

Para conocer el valor de la variable hay que usar:

```
mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
```

La variable autocommit se puede desactivar con la siguiente instrucción.

```
mysql> SET autocommit=0;
```

Órdenes SQL que fuerzan a que se ejecute un COMMIT sin necesidad de indicarlo:

QUIT	DROP VIEW	DISCONNECT
CREATE TABLE	CONNECT	REVOQUE
DROP TABLE	GRANT	AUDIT
EXIT	ALTER	NOAUDIT
CREATE VIEW		

LOCK TABLES y UNLOCK TABLES

Con el fin de prevenir la modificación de ciertas tablas y vistas en algunos momentos, cuando se requiere acceso exclusivo a las misma, en sesiones paralelas (o concurrentes), es posible bloquear el accesos a las tablas. Protege contra accesos inapropiados de lecturas y escrituras de otras sesionas.

La sintaxis para bloquear algunas tablas o vistas es:

```
LOCK {TABLE | TABLES}

tbl_name [[AS] alias] lock_type [, tbl_name [[AS] alias] lock_type] ...
```

Donde:

```
lock_type: { READ [LOCAL] | WRITE }
```

- READ, permite leer sobre la tabla, pero no escribir en ella.
- WRITE permite que la sesión que ejecuta el bloqueo pueda escribir sobre la tabla, pero el resto de sesiones sólo la puedan leer, hasta que se termine el bloqueo.

Sintaxis para desbloquear

```
UNLOCK {TABLE | TABLES}
```

Cuando se ejecuta LOCK TABLES se hace un COMMIT implícito, por tanto, si había alguna transacción abierta, ésta termina. Si termina la conexión (normal o anormalmente) antes de desbloquear las tablas, automáticamente de desbloquean las tablas.

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

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

No se puede bloquear más de una vez la tabla. Si se necesita bloquear dos veces a la misma tabla, es necesario definir un alias para el segundo bloqueo.

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

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

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

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

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

6.4 Estado de los datos durante la transacción

Si se inicia una transacción usando los comandos DML hay que tener en cuenta que:

- Se puede volver a la instrucción anterior a la transacción cuando se desee.
- Las instrucciones de consulta SELECT realizadas por el usuario que inició la transacción

muestran los datos ya modificados por las instrucciones DML.

- El resto de usuarios ven los datos tal cual estaban antes de la transacción, de hecho los registros afectados por la transacción aparecen bloqueados hasta que la transacción finalice. Estos usuarios no podrán modificar los valores de dichos registros.
- Tras la transacción todos los usuarios ven los datos tal cual quedan tras el fin de la transacción. Los bloqueos son liberados y los puntos de ruptura borrados.

6.5 Niveles de aislamiento. Acceso concurrente a los datos

Cuando se utilizan transacciones, pueden suceder problemas de concurrencia en ella en los accesos a los datos, es decir, problemas ocasionados por el acceso al mismo dato de dos transacciones distintas. Estos problemas están descritos por SQL estándar y son los siguientes:

- **Dirty Read (Lectura Sucia):** Es el problema más importante de todos. Supone que las transacciones en curso puedan leer el resultado de otras transacciones aún no confirmadas. Por ejemplo, vamos a suponer que tenemos dos transacciones activas (1 y 2)

usuarios

id	nombre	edad
1	José	20
2	Juana	25

Transacción 1	Transacción 2
`SELECT edad FROM usuarios WHERE id = 1;` `/* leerá 20 */`	
	`UPDATE usuarios SET edad = 21 WHERE id = 1;` `/* No se hace commit */`
`SELECT edad FROM usuarios WHERE id = 1; /* leerá 21 */`	
	`ROLLBACK; /* LECTURA SUCIA basada en bloqueo */`

→ No existe ningún usuario que tenga la edad de 21, pues la Transacción 2 nunca cometió los cambios!!!

- **Non-Repeatable reads (Lecturas no repetibles):** Ocurre cuando una transacción activa vuelve a leer un dato cuyo valor difiere con respecto al de la anterior lectura. Lo vemos más claro con un ejemplo.

usuarios

id	nombre	edad
1	José	20
2	Juana	25

Transacción 1	Transacción 2
`SELECT * FROM usuarios WHERE id = 1;`	
	`UPDATE usuarios SET edad = 21 WHERE id = 1; COMMIT;`
`SELECT * FROM usuarios WHERE id = 1; COMMIT;`	

→ La transacción 2 comete correctamente → sus cambios de la fila con id 1 deberían hacerse visibles. Sin embargo, la transacción 1 ya ha leído un valor distinto para edad en esa fila.

Aunque a primera vista este problema no parezca muy importante en realidad sí que lo es, sobre todo cuando X es una clave primaria o ajena. En este caso se puede originar una gran pérdida de consistencia en nuestra base de datos.

En los niveles de aislamiento SERIALIZABLE y REPEATABLE READ, el SGBDR debería devolver el valor antiguo. En los niveles READ COMMITTED y READ UNCOMMITTED, el SGBDR debería devolver el valor nuevo; esto es una lectura no repetible.

- **Phantom reads (Lecturas fantasma):** Este supone el menor problema que se nos puede plantear con respecto a las transacciones. Sucede cuando una transacción en un momento lanza una consulta de selección con una condición y recibe en ese momento N filas y posteriormente vuelve a lanzar la misma consulta junto con la misma condición y recibe M filas con M > N. Esto es debido a que durante el intervalo que va de la primera a la segunda lectura se insertaron nuevas filas que cumplen la condición impuesta en la consulta.

usuarios

id	nombre	edad
1	José	20
2	Juana	25

Transacción 1	Transacción 2
`SELECT * FROM usuarios WHERE edad BETWEEN 10 AND 30;`	
	`INSERT INTO usuarios VALUES ( 3, 'Mica', 27 ); COMMIT;`
`SELECT * FROM usuarios WHERE edad BETWEEN 10 AND 30;`	

→ **La transacción 1 ejecuta la misma consulta dos veces. Los resultados de ambas consultas pueden no coincidir.**

Debido a estos problemas el ANSI establece diferentes niveles de aislamiento (isolations levels) para solventarlos. Hay que tener en cuenta que a mayor nivel de aislamiento mayores son los sacrificios que se hacen con respecto a la concurrencia y al rendimiento. Vamos a enumerar los niveles de aislamiento desde el menor hasta el mayor:

- **Read uncommitted (Lectura sin confirmación)**: En la práctica casi no se suele utilizar este nivel de aislamiento ya que es propenso a sufrir todos los problemas anteriormente descritos. En este nivel una transacción puede ver los resultados de transacciones aún no cometidas. Podemos apreciar que en este nivel no existe aislamiento alguno entre transacciones.
- **Read committed (Lectura confirmada)**: Es el predeterminado para la mayoría de gestores de bases de datos relacionales. Supone que dentro de una transacción únicamente se pueden ver los cambios de las transacciones ya cometidas. Soluciona el problema de las lecturas sucias, pero no el de las lecturas no repetibles ni tampoco el de las lecturas fantasmas.
- **Repeatable read (Lectura repetible)**: Define que cualquier tupla leída durante el transcurso de una transacción es bloqueada. De esta forma se soluciona, además de las lecturas sucias, el problema de las lecturas no repetibles. Aunque en dicho nivel se siguen dando las lecturas fantasmas.
- **Serializable (Lecturas en serie):** Soluciona todos los problemas descritos. Para ello ordena las transacciones con el objetivo de que no entren en conflicto. Este nivel de aislamiento es bastante problemático ya que es, con diferencia, el que más sacrifica en rendimiento y concurrencia.

Nivel de aislamiento	Lecturas sucias	Lecturas no repetibles	Lecturas fantasma
READ UNCOMMITTED	SÍ	SÍ	SÍ
READ COMMITTED	NO	SÍ	SÍ
REPEATEABLE READ	NO	NO	SÍ
SERIALIZABLE	NO	NO	NO

En Oracle

El nivel por defecto es READ COMMITED, además de éste, solo permite SERIALIZABLE.

Se puede cambiar ejecutando el comando:

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

En MySQL

El nivel de aislamiento predeterminado para MySQL es de RESPETABLE READ (Lectura repetible).

```
mysql> SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
```

Se puede cambiar ejecutando el comando:

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