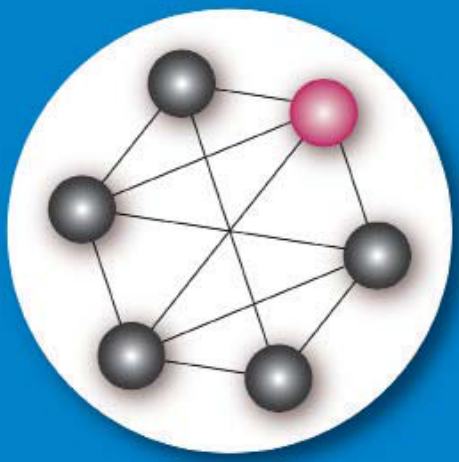


DBTechNet

DBTech VET

SQL Transactions

Teoría y ejercicios en la práctica

En español

Lifelong Learning Programme

Esta publicación se ha desarrollado en el context del proyecto DBTech VET Teachers (DBTech VET). Código: 2012-1-FI1-LEO05-09365.

DBTech VET es un proyecto Leonardo da Vinci Multilateral Transfer of Innovation,
Financiado por la Comisión Europea y los socios del proyecto.

www.DBTechNet.org
DBTech VET

Traducción de SQL Transactions – Theory and Hands-on Exercises
Version 1.3 of the first edition 2013
Versión 1.0 en Español de 2014

Autores: Martti Laiho, Dimitris A. Dervos, Kari Silpiö
Producción: DBTech VET Teachers project
Traducción: José F. Aldana-Montes, Ismael Navas-Delgado y Maria del Mar Roldán-García, Universidad de Málaga

ISBN **TBD** (paperback)
ISBN **TBD** (PDF)

Transacciones SQL
Guía del Alumno

Objetivos

El acceso fiable a los datos debe estar basado en el uso de transacciones correctamente diseñadas con una TOLERANCIA CERO hacia los datos incorrectos en nuestra base de datos. Un programador que no comprende la tecnología de transacciones puede violar fácilmente la integridad de los contenidos de la base de datos y bloquear o ralentizar el Sistema de producción. Al igual que con el tráfico de vehículos, las reglas de acceso a la base de datos deben conocerse y obedecerse. El propósito de este tutorial es presentar los conceptos básicos de la programación de las transacciones usando los principales SGBD. Se presentan además algunos problemas típicos y cómo afinar las transacciones en esos casos.

Usuarios de este Documento

Los usuarios de este tutorial son profesores, formadores y estudiantes en centros de estudio vocacionales e institutos de educación superior orientados a la industria. Igualmente los desarrolladores de aplicaciones en la industria TIC pueden encontrarlo de utilidad para comprender el funcionamiento de las transacciones en otros gestores distintos al usado en su trabajo diario.

Prerrequisitos

Debe tenerse experiencia básica en el manejo de SQL en algún SGBD.

Nivel

Básico

Métodos de Aprendizaje

Los estudiantes deberían experimentar y verificar por si mismos los temas presentados en este tutorial usando SGBD reales. Para este propósito, en este tutorial se presenta un laboratorio virtual gratuito de bases de datos, junto con scripts de prueba proporcionados en la sección de referencias.

Contenidos

1 Transacción SQL: La Unidad Lógica de Trabajo

1.1 Introducción a las transacciones

En la vida diaria la gente realiza distintos tipos de transacciones de negocios, comprando productos, reservando viajes, cambiando o cancelando pedidos, comprando entradas de conciertos, pagando alquileres, recibos de electricidad, recibos de seguros, etc. Las transacciones no hacen referencia únicamente a temas relacionados con los ordenadores. Cualquier tipo de actividad humana que conlleve una **unidad lógica de trabajo** que tenga que ser realizado en su completitud o ser cancelada implica una transacción.

Casi todos los sistemas de información utilizan los servicios de algún Sistema Gestor de Bases de Datos (SGBD) para el almacenamiento y recuperación de los datos. Los SGBD actuales son muy sofisticados tecnológicamente en relación a la seguridad de la integridad de los datos, proporcionando acceso rápido a los datos incluso con múltiples usuarios accediendo de forma concurrente a los mismos. Estos SGBD proporcionan a las aplicaciones servicios fiables para la gestión de la persistencia de los datos sólo en el caso de que estas aplicaciones usen estos servicios fiables de forma correcta. Esto se consigue desarrollando componentes de **acceso a datos** de la aplicación que hagan uso de las **transacciones de bases de datos**.

El uso inapropiado de las transacciones en las aplicaciones puede producir:

- la pérdida de pedidos de clientes y órdenes de envío de pedidos,
- fallos en el registro de una reserva de asientos realizada por pasajeros de trenes o aviones o la incluso la realización de una reserva duplicada.
- la pérdida del registro de llamadas de emergencia en un centro de respuesta de emergencias,
- etc.

Estos incidentes suceden frecuentemente en la vida real, pero las personas a cargo prefieren no revelar estas historias al público. La misión del Proyecto DBTech VET es establecer un marco de trabajo de buenas prácticas y una metodología para evitar este tipo de incidentes desafortunados.

Las transacciones son unidades recuperables de tareas de acceso a datos para la manipulación del contenido de las bases de datos. Estas incluyen también unidades de recuperación de la base de datos (completa) en caso de fallo del sistema, como se muestra en el Apéndice 3. Además proporcionan también las bases para la gestión de la concurrencia en entornos multi-usuario.

1.2 Conceptos Cliente/Servidor en el Entorno SQL

En este tutorial nos centramos en el acceso a datos usando transacciones SQL cuando se ejecuta código SQL de forma interactiva, pero teniendo en mente que el correspondiente acceso a los datos a través de una aplicación usa un paradigma ligeramente distinto (que se presenta en el Apéndice 2).

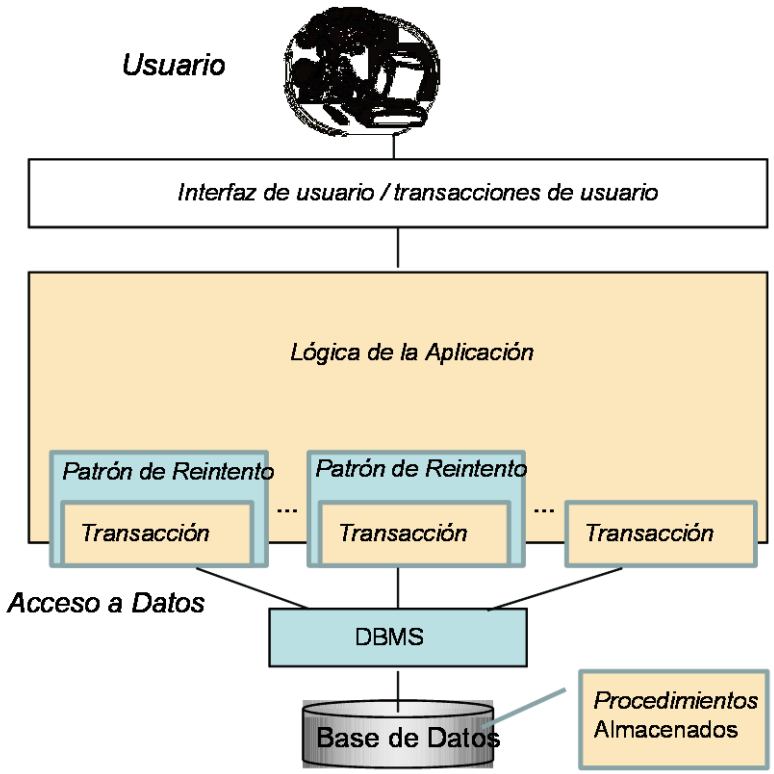


Figura 1.1 Las transacciones SQL respecto a las capas de aplicación

La Figura 1.1 presenta una vista simplificada de la arquitectura típica de una aplicación de bases de datos en la que las transacciones se encuentran localizadas en una capa software diferente a la capa de interfaz de usuario. Desde el punto de vista del usuario final, la aplicación proporciona casos de uso materializándolos como **transacciones de usuario**. Una transacción de usuario puede incluir múltiples transacciones SQL, algunas de las cuales incluyen recuperación de datos, y usualmente la transacción final actualiza el contenido de la base de datos. Los **envoltorios de reintento** implementan acciones de reintento programadas en caso de fallos de concurrencia en las transacciones SQL.

Para comprender el funcionamiento de las transacciones SQL tenemos que introducir algunos conceptos básicos relativos al diálogo cliente-servidor (ver Figuras 2 y 3). Para acceder a una base de datos la aplicación tiene que iniciar una **conexión con la base de datos** que establece el contexto de una **sesión SQL**. Por simplicidad, una sesión SQL se considera el **cliente** SQL, y el servidor de base de datos se corresponde con el **servidor**. Desde el punto de vista del servidor, la aplicación usa los servicios de la base de datos en modo cliente/servidor pasando **comandos SQL** como parámetros a las funciones/métodos a través de una API (Interfaz Programática para Aplicaciones del inglés *application programming interface*)[1] de acceso a datos. Independientemente de la interfaz de acceso a los datos usada, el diálogo de "bajo nivel" con el servidor se basa en el lenguaje SQL, y el acceso fiable a los datos se materializa con el uso apropiado de las transacciones SQL.

[1] Tales como ODBC, JDBC, ADO.NET, LINQ, etc. dependiendo del lenguaje de programación usado, como C++, C#, Java, PHP, etc.

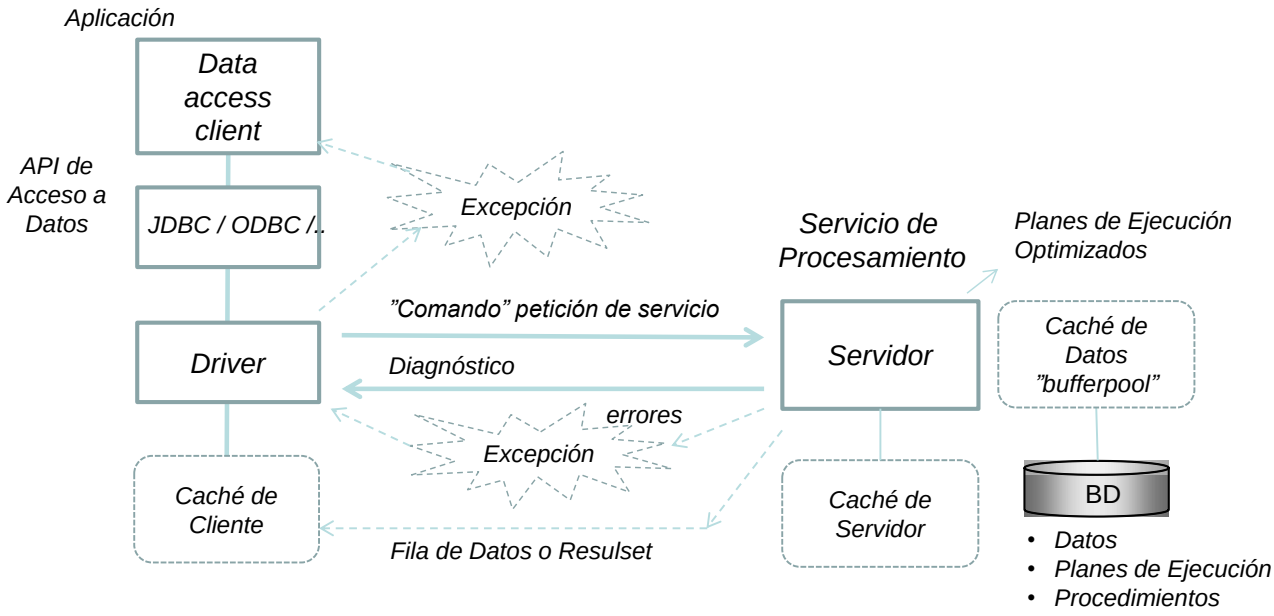


Figura 1.2 Explicación del procesamiento de comando SQL

La Figura 1.2 explica el "viaje de ida y vuelta" del procesamiento cíclico de una sentencia SQL, que se inicia en el cliente mediante una **petición de servicio.** El servidor usa una caché de datos intermedia y los servicios de red. El comando SQL puede implicar una o más **sentencias SQL**. La/s sentencia/s SQL del comando son analizadas en base a los metadatos de la base de datos, optimizadas y finalmente ejecutadas. Para reducir la degradación del rendimiento debido a operaciones de Entrada/Salida de disco, el servidor conserva todas las filas recientemente usadas en un buffer residente en memoria (RAM) y todo el procesamiento de datos tiene lugar ahí.

La ejecución del comando SQL introducido en el servidor es atómica en el sentido de que el comando SQL completo tiene que tener éxito o en caso contrario todo el comando tendrá que deshacerse. Como respuesta al comando SQL, el servidor manda uno o varios mensajes de diagnóstico informando del éxito o fallo del comando. Los errores de ejecución del comando se reflejan en el cliente como un conjunto de excepciones. Sin embargo, es importante comprender que las sentencias SQL como UPDATE o DELETE tienen éxito en la ejecución incluso ante la ausencia de filas afectadas. Desde el punto de vista de la aplicación, estos casos podrían verse como un fallo, pero en lo que concierne a la ejecución del comando supone una ejecución correcta. Por tanto, el código de la aplicación tiene que verificar los mensajes de diagnóstico enviados por el servidor para determinar el número de filas afectadas por la operación en cuestión.

En el caso de que una sentencia SELECT el cliente accede al resultado fila a fila. Estas filas se recuperan directamente del servidor a través de la red o de la caché del cliente.

1.3 Transacciones SQL

Cuando la lógica de aplicación requiere la ejecución de múltiples comandos SQL de forma atómica, las sentencias necesitan incluirse en una unidad lógica de trabajo (LUW del inglés *logical unit of work*) llamada **transacción SQL** que finaliza con un **COMMIT** o un **ROLLBACK**. El primero acepta/registra todos los cambios que la transacción ha hecho en la base de datos, mientras que el segundo deshace todos ellos.

La ventaja del comando ROLLBACK es que cuando la lógica de la aplicación programada en la transacción no puede completarse, no es necesario realizar una serie de operaciones de vuelta atrás comando a comando, sino que el trabajo puede cancelarse completamente mediante el comando ROLLBACK, lo cual siempre tiene éxito. Aquellas transacciones no confirmadas al final del programa o ante fallos del sistema tendrán que deshacerse automáticamente por el sistema. Igualmente, en caso

de conflictos por concurrencia, algunos SGBD desharán automáticamente los efectos de una transacción, como se explica a continuación.

Nota: De acuerdo con el estándar ISO SQL y tal y como está implementado en DB2 y Oracle, cualquier comando SQL al comienzo de una sesión SQL o después del final de una transacción iniciará automáticamente una nueva transacción SQL. Este caso se conoce como **inicio implícito** de una transacción SQL.

Algunos SGBD como por ejemplo SQL Server, MySQL, PostgreSQL y Pyrrho funcionan por defecto en modo **AUTOCOMMIT.** Esto significa que la ejecución de cada comando SQL se confirma automáticamente y que los efectos/cambios del comando **no podrán ser deshechos**. Por tanto, en el caso de que se produzcan errores la aplicación deberá realizar las operaciones inversas en la unidad lógica de trabajo, cosa que puede no ser posible después de realizarse ciertas operaciones en los clientes SQL. En caso de que se rompa la conexión la base de datos deberá dejarse igualmente en un estado inconsistente. Por tanto, si se quiere usar una lógica transaccional real necesitaremos iniciar una transacción mediante un comando de **inicio explícito**, como *BEGIN WORK*, *BEGIN TRANSACTION*, o *START TRANSACTION*, dependiendo del SGBD usado.

Nota: En MySQL/InnoDB, una sesión SQL puede configurarse para usar tanto transacciones implícitas como explícitas mediante la siguiente sentencia:

```
SET AUTOCOMMIT = {0 | 1}
```

Donde 0 implica el uso de transacciones explícitas y 1 el uso de modo AUTOCOMMIT.

Nota: Algunos productos como Oracle hacen uso de confirmaciones implícitas de la transacción ante la ejecución de cualquier sentencia DDL (CREATE, ALTER o DROP de algún objeto como TABLE, INDEX, VIEW, etc.).

Nota: En SQL Server la instancia completa, incluyendo sus bases de datos, puede configurarse para el uso de transacciones implícitas. Una conexión (**sesión SQL**) ya iniciada puede cambiarse para usar transacciones explícitas o volver al modo AUTOCOMMIT ejecutando la siguiente sentencia:

```
SET IMPLICIT_TRANSACTIONS {ON | OFF}
```

Nota: Igualmente algunas utilidades como los procesadores de línea de comandos (CLP del inglés *Command Line Processor*) de DB2 de IBM, y algunas interfaces de acceso a datos como ODBC y JDBC funcionan por defecto en modo AUTOCOMMIT. Por ejemplo, en la API JDBC cada transacción necesita iniciarse usando el método de conexión del objeto:

```
<connection>.setAutoCommit(false);
```

En vez de una secuencia simple de tareas de acceso a los datos, algunas transacciones SQL pueden implicar una lógica de programa compleja. En estos casos, la lógica de la transacción tomará decisiones durante su ejecución dependiendo de la información recuperada de la base de datos. Incluso en este caso, la transacción SQL puede considerarse como una "unidad lógica de trabajo" (LUW) indivisible que puede terminar de forma exitosa o deshacerse. Sin embargo el fallo en una transacción

normalmente no genera de forma automática un ROLLBACK[2], sino que tiene que ser diagnosticada en el código de la aplicación (ver "Diagnosticando errores SQL") y la aplicación se encarga ella misma de realizar el ROLLBACK.

1.4 Lógica de la Transacción

Consideremos la siguiente tabla de cuentas de banco:

```
CREATE TABLE Accounts (
acctId INTEGER NOT NULL PRIMARY KEY,
balance DECIMAL(11,2) CHECK (balance >= 0.00)
);
```

Un ejemplo típico de libro de texto sobre transacciones SQL es la transferencia de dinero entre cuentas bancarias (por ejemplo 100 euros):

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE acctId = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctId = 202;
COMMIT;
```

Si el sistema falla o pierde la conexión entre el cliente y el servidor después de la primera sentencia UPDATE, el protocolo de la transacción garantiza que no se perderá dinero de la cuenta 101, ya que la transacción será finalmente deshecha (rollback). Sin embargo, la transacción está lejos de ser fiable:

a) En el caso de que una de las cuentas no exista la sentencia UPDATE se ejecutará y finalizará de forma correcta en términos SQL. Por tanto, sería necesario inspeccionar los mensajes de diagnóstico SQL y comprobar que el número de filas afectadas por cada uno de los comandos UPDATE es el mismo.

b) En el caso de que la primera sentencia UPDATE falle debido a que el balance de la cuenta 101 vaya a ser negativo (violando por tanto la restricción CHECK establecida en la tabla), entonces se procede a ejecutar de forma satisfactoria la segunda sentencia UPDATE, lo que llevará a un estado erróneo en la base de datos.

A partir de este ejemplo sencillo nos damos cuenta de que los desarrolladores de las aplicaciones deben ser conscientes de la forma en la que se comportan los SGBD y cómo los diagnósticos SQL se inspeccionan en las APIs de acceso a datos. Incluso entonces, hay mucho más por aprender además de un número de operaciones de configuración de la base de datos.

1.5 Diagnosticando errores SQL

En lugar de una secuencia simple de tareas de acceso a datos, algunas transacciones SQL pueden implicar una lógica de programa compleja. En estos casos, la lógica de la transacción tomará decisiones durante su ejecución dependiendo de la información recuperada de la base de datos. Incluso en este caso, la transacción SQL puede considerarse como una "unidad lógica de trabajo" (LUW) indivisible que puede terminar de forma exitosa o ser deshecha. Sin embargo el fallo en una transacción normalmente no genera de forma automática un ROLLBACK, sino que debería ser

[2] A parte de los SGBD incluidos en nuestro laboratorio DebianDB, después de un error en una transacción PostgreSQL rechazará todos los comandos y solo aceptará ROLLBACK, mientras que Pyrrho deshace todos los comandos previos en la transacción dejando a la aplicación la decisión de si continuar o no.

diagnosticada por el código de la aplicación y la aplicación se encargará ella misma de realizar el ROLLBACK.

El estándar ISO SQL-98 definía un indicador de tipo entero **SQLCODE** cuyo valor es 0 en caso de éxito, 100 si se produce un error "NOT FOUND" y otros valores definidos por cada producto y que estaban descritos en los manuales de referencia de cada producto. El número de filas procesadas correctamente se encuentra en el área de diagnosis SQL. En el caso de SQL embebido este área es accedida a través de una estructura llamada *SQL Descriptor Area (SQLDA)*.

En el estándar ISO SQL-92 el SQLCODE se reemplaza por **SQLSTATE** que es una cadena de 5 caracteres de los cuales los 2 primeros indican la clase de código de error o advertencia SQL, y los tres últimos codifican subtipos de los errores. La cadena de ceros ("00000") indica ejecución correcta. Se han estandarizado cientos de valores (por ejemplo para violaciones de restricciones de integridad) y cada SGBD define algunos adicionales. Por ejemplo, los valores de SQLSTATE que comienzan por "40" indican una transacción perdida por un conflicto de concurrencia, quizás un error en un procedimiento almacenado, conexión perdida o un problema en el servidor.

El lenguaje SQL ha sido extendido con funcionalidades procedimentales, incluyendo disparadores (*triggers*), y APIs de llamada a nivel de interfaz cliente/servidor (*CLIs APIs*). El grupo X/Open ha extendido el lenguaje SQL para proporcionar mejor información de diagnóstico para las aplicaciones cliente sobre lo que ha ocurrido en el servidor. Con la sentencia **GET DIAGNOSTICS** pueden obtenerse elementos de información más detallados y puede navegarse a través de los registros de diagnóstico que informan de múltiples errores o advertencias. Esto ha sido extendido igualmente en los estándares ISO SQL desde SQL:1999, pero solo una parte ha sido implementada por los distintos SGBD. Por ejemplo DB2, Mimer y MySQL 5.6 implementan estas extensiones. El siguiente ejemplo presenta los elementos de diagnóstico en la lectura en MySQL 5.6:

```
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
                            @sqlcode = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;
```

Algunas implementaciones de SQL con características procedimentales disponen de indicadores de diagnóstico en registros especiales o funciones del lenguaje. Por ejemplo, en Transact-SQL (también llamado T-SQL) de MS SQL Server algunos indicadores de diagnóstico están disponibles en @@-variables como @@ERROR para errores nativos, o @@ROWCOUNT para el número de filas procesadas.

En el lenguaje SQL nativo de IBM DB2 los indicadores de diagnóstico (de ISO SQL) SQLCODE y SQLSTATE están disponibles en la extensión procedimental del lenguaje para procedimientos almacenados, como se muestra a continuación:

```
<Sentencia SQL >
IF (SQLSTATE <> '00000') THEN
     <manejo de errores >
END IF;
```

En los bloques BEGIN-END del lenguaje **PL/SQL** de Oracle el manejo de error (o excepción) se codifica en la parte final del código mediante la directiva EXCEPTION como se muestra a continuación:

```
BEGIN
     <procesamiento>
EXCEPTION
WHEN <nombre de la excepción > THEN
     <manejo de la excepción>;
...
WHEN OTHERS THEN
     err_code := sqlcode;
     err_text := sqlerrm;
     <manejo de la excepción >;
END;
```

Las implementaciones más tempranas de los registros de diagnósticos pueden encontrarse en ODBC y SQLExceptions y SQLWarnings de JDBC. En la API de JDBC del lenguaje Java los errores SQL dan lugar a excepciones SQL. Estas excepciones tienen que tratarse posteriormente a través de estructuras de control *try-catch* como se muestra a continuación (ver Apéndice 2):

```
... throws SQLexception {
..
try {
     ...
     <Sentencia(s) JDBC>
}
catch (SQLexception ex) {
     <manejo de excepciones>
}
```

El número de filas afectadas/procesadas se devuelve a la aplicación mediante los métodos de ejecución de JDBC (*rowcount*).

1.6 Práctica de Laboratorio (Hands-on)

> **Nota**: ¡No te creas todo lo que leas! Para el desarrollo de aplicaciones fiables tienes que **experimentar y verificar** por ti mismo los servicios que ofrece tu SGBD. Los SGBD difieren en la forma en la que dan soporte incluso para los servicios de transacciones SQL básicos.

En el Apéndice 1 presentamos las pruebas de transacciones explícitas e implícitas, COMMIT y ROLLBACK, y la lógica de las transacciones usando MS SQL Server, pero el lector debería intentar verificar por sí mismo el comportamiento de SQL Server en los apartados A1.1-A1.2. Para estos experimentos se puede descargar de forma gratuita SQL Server Express del sitio Web de Microsoft.

En el primer laboratorio práctico (**HoL**) se describe el uso de la máquina virtual de la red DBTechNET (DebianBD) que viene con varios gestores pre-instalados: IBM DB2 Express-C, Oracle XE, MySQL GA, PostgreSQL, y Pyrrho. MySQL GA no es el gestor más fiable pero dado que es el más usado en los ciclos formativos, en adelante lo usaremos como primer gestor para mostrar las tareas de manipulación de las transacciones.

Así que, bienvenido al viaje del misterio de las transacciones usando MySQL. El comportamiento de MySQL depende del motor de base de datos usado. El motor por defecto de versiones previas no tiene soporte de transacciones. Pero desde la versión 5.1 el motor por defecto trae soporte para transacciones. El motor por defecto es InnoDB, que en algunos casos producirá resultados inesperados.

Nota: Las series de experimentos siguen la misma temática para todos los SGBD instalados en DebianDB y los scripts correspondientes pueden encontrarse en el Apéndice 1. Las pruebas no tienen como objetivo demostrar los problemas de MySQL pero no podemos evitar presentarlos, ya que los desarrolladores de aplicaciones tienen que conocer los errores de los distintos productos y ser capaces de aplicar soluciones que los resuelvan en cierta medida.

Esperamos que el lector haya revisado el documento de "inicio rápido" del laboratorio de bases de datos DebianDB. Este documento explica la instalación y la configuración inicial necesaria para poner la máquina virtual en funcionamiento.

Una vez tenemos la máquina funcionando es importante destacar que el usuario obtiene acceso directamente (con usuario y clave, "student" y "Student1" respectivamente). Para poder proceder con la creación de la base de datos en MySQL se debe usar la cuenta "root" con clave "P4ssw0rd" como se explica en la guía de inicio rápido. Esto puede realizarse desde la línea de comandos, que se accede seleccionando "*Terminal/Use the command line*" (Figura 1.4).

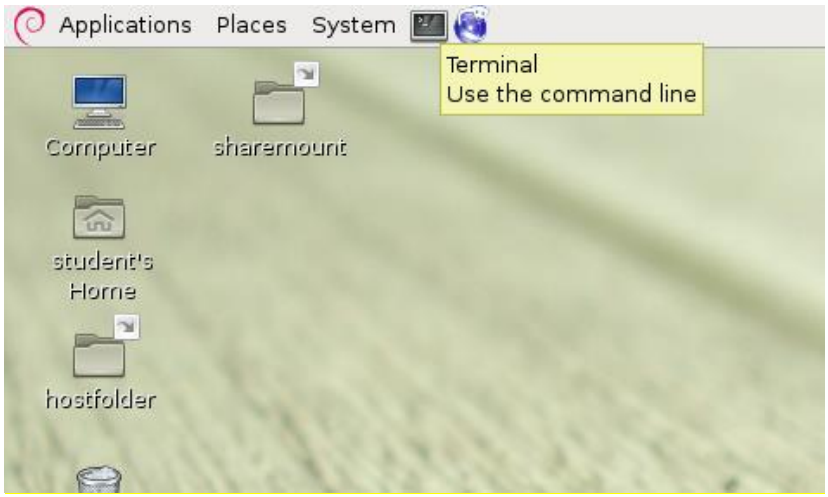


Figura 1.4 "Terminal / Use the command line"

A continuación deben ejecutarse los siguientes comandos Linux en el terminal para iniciar el cliente **mysql** (Figura 1.5):

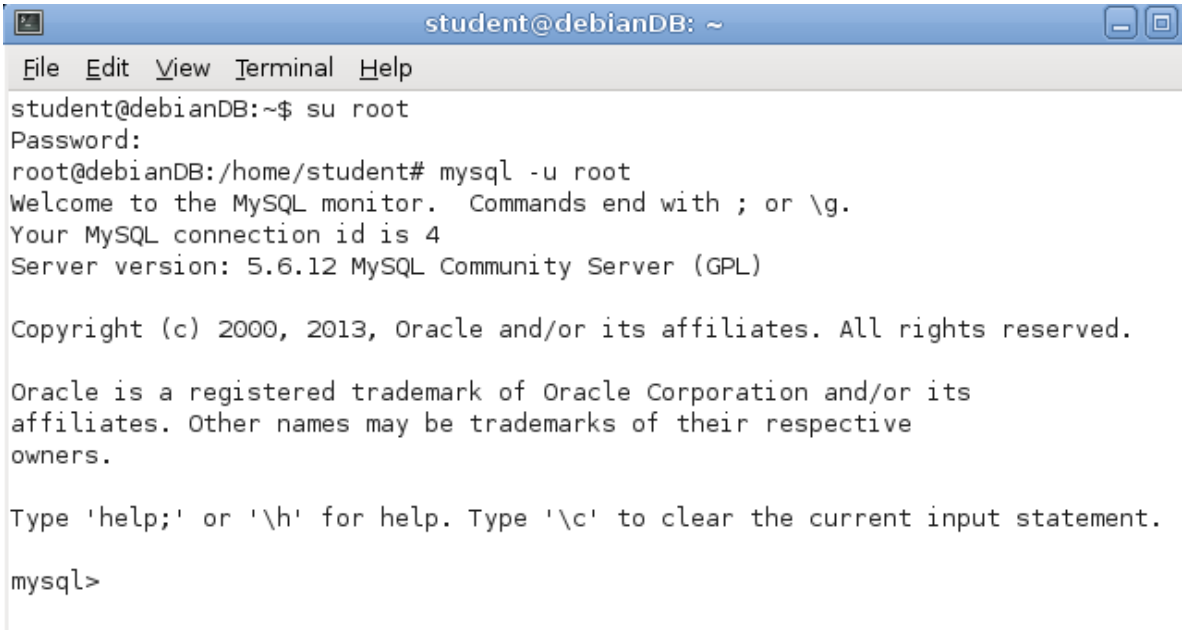


Figura 1.5 Inicialización de una sesión MySQL como usuario 'root'

El comando SQL que crea una nueva base de datos llamada "TestDB" es el siguiente:

```
-----------------------------------
CREATE DATABASE TestDB;
-----------------------------------
```

Para dar privilegios en la base de datos al usuario "student" el usuario "root" debe ejecutar el siguiente comando:

```
-----------------------------------
GRANT ALL ON TestDB.* TO 'student'@'localhost';
-----------------------------------
```

Es el momento de terminar la sesión del "root", para lo que se debe salir de MySQL y de la sesión del "root":

```
-----------------------------------
EXIT;
exit
-----------------------------------
```

> **Note**: Si durante la sesión de práctica sucede que la pantalla de DebianDB se vuelve negra y muestra la ventana de entrada de contraseña para el usuario actual ("student") del sistema, la clave necesaria para desbloquear la pantalla es "password" ☺

Ahora el usuario "student" puede iniciar MySQL para acceder a la base de datos TestDB:

```
-----------------------------------
mysql
use                                                    TestDB;
-----------------------------------
```

Este es el comienzo de la nueva sesión MySQL.

EJERCICIO 1.1

Creamos una primera tabla llamada "T" que tiene tres columnas: id (de tipo *integer*, clave primaria), s (de tipo *string* de longitud máxima 30), y si (de tipo *small integer*):

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si SMALLINT);
```

Después de cada sentencia SQL, MySQL muestra mensajes de diagnóstico de la ejecución del comando.

Para asegurarse de que la tabla existe y tiene la estructura deseada usamos el comando DESCRIBE:

```
DESCRIBE T;
```

Nota: MySQL en Linux es insensible a las diferencias entre mayúsculas y minúsculas con la excepción de los nombres de tablas y bases de datos. Esto significa que los siguientes comandos funcionarían: "describe T", "describe T", "create TaBle T…". Sin embargo "use testDB", y "describe t" hacen referencia a una base de datos y una tabla diferente a la pretendida.

Es ahora necesario añadir datos a la tabla:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (1, 'first');
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
-----------------------------------
```

Una sentencia "SELECT * FROM T" confirma que las tres tuplas se han insertado correctamente (véase como la columna "si" contiene valores NULL).

Nota: Asegúrese siempre de usar punto y coma (";") al final de cada comando antes de pulsar "*intro*".

Teniendo en cuenta todo lo aprendido hasta el momento vamos a tratar de deshacer la transacción actual:

```
-- --------------------------------
ROLLBACK;
SELECT * FROM T ;
-- --------------------------------
```

Parece que ha funcionado, pero después de ejecutar nuevamente la sentencia "SELECT * FROM T" parece que la tabla continua teniendo las tres tuplas añadidas anteriormente. ¡Vaya sorpresa!

El origen de esta primera sorpresa tiene un nombre: “AUTOCOMMIT”. MySQL se inicia en modo AUTOCOMMIT, en el que cada transacción debe iniciarse mediante el comando “START TRANSACTION”. Y después de acabar la transacción MySQL vuelve de Nuevo al modo AUTOCOMMIT. Vamos a probar esto con los siguientes comandos:

```
-- --------------------------------
START TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T ;
ROLLBACK;

SELECT * FROM T;
-- --------------------------------
```

Pregunta:

- Comparar los resultados obtenidos de la ejecución de los dos comandos SELECT * FROM T

EJERCICIO 1.2

A continuación ejecutamos los siguientes comandos:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Preguntas:

- ¿Que se obtiene como resultado de ejecutar la sentencia SELECT * FROM T?
- ¿Conclusiones obtenidas con respecto a la existencia de posibles limitaciones en el uso de START TRANSACTION en MySQL/InnoDB?

EJERCICIO 1.3

Ahora desactivamos el modo AUTOCOMMIT, pero borramos antes el contenido de la tabla:

```
-----------------------------------
SET AUTOCOMMIT = 0;

DELETE FROM T WHERE id > 1;
-----------------------------------
```

Insertamos de nuevo:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
SELECT * FROM T;
-----------------------------------
```

y ROLLBACK:

```
-----------------------------------
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Pregunta:

- ¿Cuál es la ventaja/desventaja de usar “SET TRANSACTION”, comparado con “SET AUTOCOMMIT”, para desactivar el modo AUTOCOMMIT?

Nota: Mientras que en el terminal Linux de MySQL es posible usar las teclas arriba y abajo para acceder a sentencias previamente ejecutadas, esto no es posible en el resto de SGBDs instalados en DebianDB.

Nota: El uso de dos guiones consecutivos en SQL indica que lo que sigue es un comentario.

EJERCICIO 1.4

```
-- Inicializar en caso de querer repetir el ejercicio 1.4
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
DROP TABLE T2;  --
COMMIT;
```

Podemos clasificar los comandos SQL en DDL (*Data Definition Language*) o DML (*Data Manipulation Language*). Ejemplos de DDL son “CREATE TABLE”, “CREATE INDEX”, y “DROP TABLE”. Ejemplos de DML son “SELECT FROM”, “INSERT INTO”, “DELETE FROM”. Teniendo los ejemplos anteriores en mente podemos pensar un poco más en el ámbito de actuación de “ROLLBACK”:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (9, 'will this be committed?');
CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;

SELECT * FROM T;  -- What has happened to T ?
SELECT * FROM T2; -- What has happened to T2 ?
-- Compare this with SELECT from a missing table as follows:
SELECT * FROM T3; -- assuming that we have not created table T3

SHOW TABLES;
DROP TABLE T2;
COMMIT;
```

Pregunta:

- ¿Qué conclusion obtenemos?

EJERCICIO 1.5

Borramos nuevamente el contenido de la tabla T:

```
-----------------------------------
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
COMMIT;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Vamos a analizar ahora qué sucede si existe un error y cómo esto ocasiona un ROLLBACK en MySQL:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'La prueba de errores comienza aqui');
-- la division por cero debería fallar
SELECT (1/0) AS dummy FROM DUAL;
-- Actalizamos una tupla inexistente
UPDATE T SET s = 'foo' WHERE id = 9999 ;
-- y borramos una tupla inexistente
DELETE FROM T WHERE id = 7777 ;
--
INSERT INTO T (id, s) VALUES (2, 'Esto es un duplicado');
INSERT INTO T (id, s)
       VALUES (3, 'Probamos a introducir una cadena de tamanio mayor que el
permitido');
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow para 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, '¿Sigue active la transaccion?');
SELECT * FROM T;
COMMIT;

DELETE FROM T WHERE id > 1;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Preguntas:

- ¿Qué se ha descubierto sobre el rollback automático cuando hay errores SQL en MySQL?
- ¿Es la división por cero un error?
- ¿Reacciona MySQL ante desbordamientos?
- ¿Qué aprendemos de los siguientes resultados?

```
mysql> UPDATE T SET s = 'foo' WHERE id = 9999 ;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> INSERT INTO T (id, s) VALUES (2, 'Esto es un duplicado');
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
```

En este caso el valor “23000” mostrado por el cliente mysql es el valor de SQLSTATE que indica que se ha violado una restricción de clave primaria y 1062 el código de error interno de MySQL.

Los diagnósticos para las sentencias INSERT fallidas en nuestro ejemplo pueden ser localizados con la sentencia GET DIAGNOSTICS, nueva en la versión 5.6 de MySQL:

```
mysql> GET DIAGNOSTICS @rowcount = ROW_COUNT;
Query OK, 0 rows affected (0.00 sec)

mysql> GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
    ->                             @sqlcode = MYSQL_ERRNO ;
Query OK, 0 rows affected (0.00 sec)
```

Las variables que comienzan por “@” son variables locales del lenguaje SQL en MySQL. Hacemos uso de las variables locales en nuestros ejercicios para simular el nivel de aplicación ya que en este libro nos centramos en el nivel del lenguaje SQL. En las APIs de acceso a datos los valores de diagnóstico se leen directamente en variables del lenguaje de programación usado en la aplicación. Veamos el ejemplo usando variables locales:

```
mysql> SELECT @sqlstate, @sqlcode, @rowcount;
+-----------+----------+-----------+
| @sqlstate | @sqlcode | @rowcount |
+-----------+----------+-----------+
| 23000     |     1062 |        -1 |
+-----------+----------+-----------+
1 row in set (0.00 sec)
```

EJERCICIO 1.6

```
DROP TABLE Accounts;
SET AUTOCOMMIT=0;
```

MySQL no da soporte a la sintaxis para **restricciones a nivel de columna de tipo CHECK**, solo la sintaxis a nivel de tupla. Pero incluso aunque admita la sintaxis, no usa estas restricciones como veremos en el siguiente ejemplo:

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL CONSTRAINT unloanable_account CHECK (balance >= 0)
);
```

Admite el uso de CHECK a nivel de tupla:

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL ,
CONSTRAINT unloanable_account CHECK (balance >= 0)
) ENGINE = InnoDB;
```

Pero aunque acepte la sintaxis no usa las restricciones CHEVK como veremos en el siguiente ejemplo que debería fallar:

```
INSERT INTO Accounts (acctID, balance) VALUES (100,-1000);
SELECT * FROM Accounts;
ROLLBACK;
```

Nota: La sentencia CHECK no se elimina para mantener los ejemplos de forma similar a los usados en otros productos. Todos los productos tienen errores y los programadores deben conocerlos y adaptar sus programas a estas eventualidades. Este problema concreto puede solventarse creando TRIGGERS, que están fuera del alcance de este documento pero pueden encontrarse en el script AdvTopics_MySQL.txt.

```
SET AUTOCOMMIT=0;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;

-- intentemos hacer una transaferencia
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
ROLLBACK;

-- Comprobemos si funciona el uso de CHECK:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
ROLLBACK;
```

Intentemos realizar ahora una transferencia de 500 euros de la cuenta 101 a una cuenta que no exista (acctID=777):

```
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts ;
ROLLBACK;
-----------------------------------
```

Preguntas:

- ¿Se ejecutan las dos actualizaciones pese a que la segunda de ellas trata de actualizar una tupla inexistente?
- ¿Si sustituimos el ROLLBACK por un COMMIT, habría tenido éxito la transacción teniendo efectos permanentes en la base de datos?
- ¿Viola la transacción una regla de integridad de los datos que debe conservarse para todos los datos contenidos en la tabla en cuestión?. En caso afirmativo, ¿realiza alguna acción MySQL para resolverlo o manda mensajes de diagnóstico para que sea la aplicación la encargada de detectar el problema y tomar las acciones oportunas para conservar la integridad de los datos?

Ejercicio 1.7 Las transacciones SQL como unidades de recuperación

Experimentaremos ahora con la propiedad de "**unidad de recuperación**" de las transacciones SQL en el caso de transacciones no confirmadas. Para probarlo iniciaremos una transacción y luego cortaremos la conexión con el cliente mysql. Posteriormente al volver a conectarnos veremos si los cambios hechos en las transacciones sin confirmar se encuentran en la base de datos.

Esta prueba de desconexión es algo similar a lo que sucedería en el caso de aplicaciones Web si existen pérdidas en la conexión.

Añadimos una tupla a T:

```
-----------------------------------
SET AUTOCOMMIT = 0;
INSERT INTO T (id, s) VALUES (9, 'Veamos que sucede');
SELECT                        *                        FROM                        T;
-----------------------------------
```

A continuación veamos que sucede si se interrumpe la conexión usando un "Control C" (Ctrl-C) (Figura 1.6):

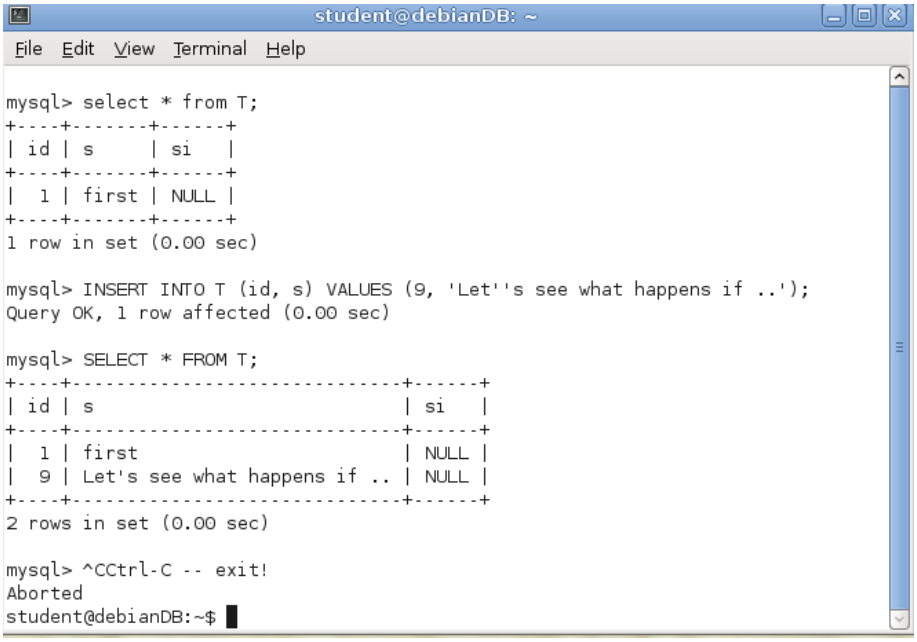


Figura 1.6 Simulación de un fallo en un SGBD

A continuación cerramos el terminal y abrimos uno nuevo con el inicio por tanto de una nueva sesión:

```
-----------------------------------
mysql
USE                                                                    TestDB;
SET                                                                 AUTOCOMMIT=0;
SELECT                          *                       FROM                    T;
COMMIT;
-----------------------------------
```

Pregunta:

- ¿Algún comentario sobre el contenido de la tabla T?

Todos los cambios hechos en la base de datos son trazados en el log de transacciones de la base de datos. El Apéndice 3 explica como los servidores de bases de datos usan este log para volver al último estado de transacciones confirmadas antes del error del sistema. El ejercicio 1.7 podría extenderse para probar errores del sistema si en vez de cancelar la sesión del cliente mysql simplemente abortamos el servidor MySQL.

Parte 2. Transacciones Concurrentes

Aviso:
¡No te creas todo lo que leas! Para el desarrollo aplicaciones fiables tienes que **experimentar y verificar** por ti mismo los servicios de tu SGBD. Los SGBD difieren en la forma en la que dan soporte incluso para los servicios de transacciones SQL básicos.

Una aplicación funcionando correctamente en un entorno mono-usuario puede no funcionar de forma correcta cuando se ejecute de forma concurrente con otros clientes (instancias de la misma u otras aplicaciones) en un entorno multi-usuario como se muestra en la Figura 2.1.

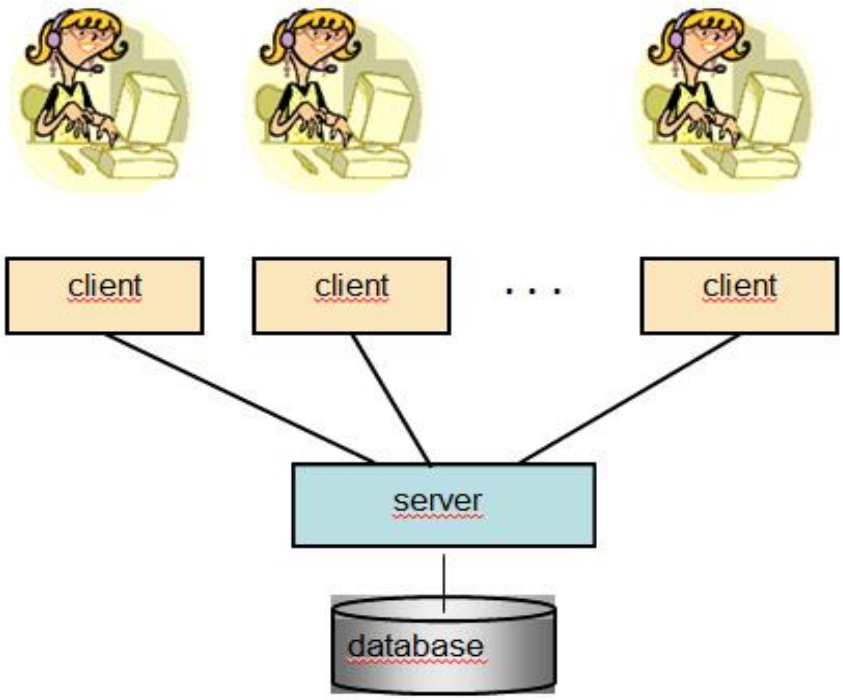


Figura 2.1. Múltiples clientes accediendo a la misma base de datos (en un entorno multi-usuario)

1.7 2.1 Problemas de Concurrencia – Posibles Riesgos de Fiabilidad

Sin los servicios de control de la concurrencia apropiados en un sistema gestor de bases de datos o con la falta de conocimiento sobre cómo usar estos servicios de forma adecuada, el **contenido** en la base de datos o los **resultados** de nuestras consultas podrían corromperse, y dejar de ser fiables.

En esta sección se tratan los problemas (anomalías) típicos debidos a la concurrencia:

- Problema de la pérdida de actualizaciones (*lost update*).
- Problema de la lectura sucia (*dirty read*) debido a la lectura de datos sin confirmar de alguna transacción concurrente.
- Problema de la lectura no repetible (*non-repeatable read*) debido a una lectura que al repetirse no devuelva las mismas tuplas.
- Problema de la lectura fantasma (*phantom read*) debido a que durante la transacción algunas tuplas no son vistas por la transacción.

Posteriormente presentaremos cómo estos problemas pueden resolverse de acuerdo al estándar ISO SQL y mediante los SGBD reales.

2.1.1 Problema de la Actualización Perdida

C. J. Date presenta el siguiente ejemplo (Figura 2.2) en el que dos usuarios en dos cajeros automáticos distintos están sacando dinero de la misma cuenta con un balance inicial de 1.000 Euros.

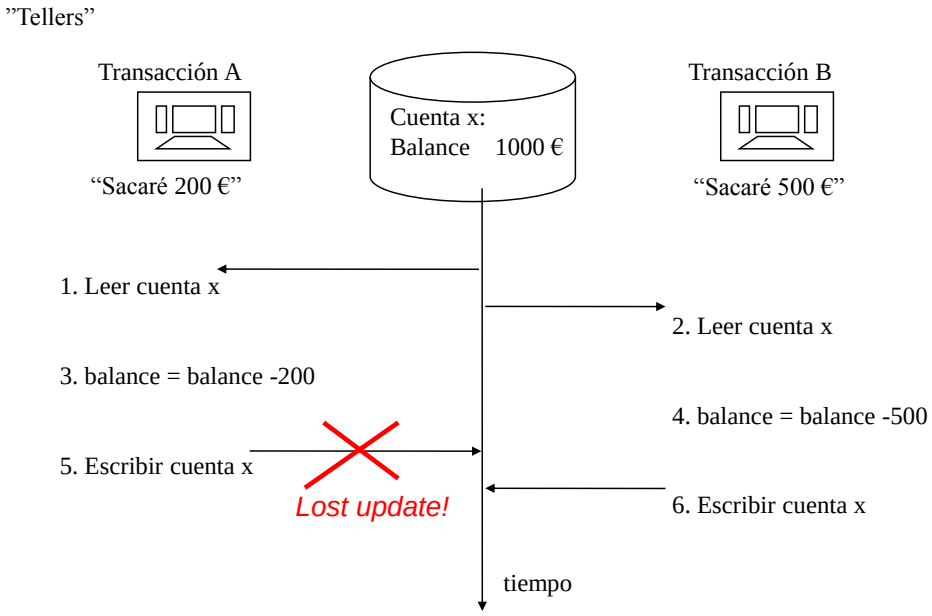


Figura 2.2. Problema de la actualización perdida

Sin un control de la concurrencia, el resultado de 800 Euros de la operación de escritura de la transacción A en el paso 5 se perderá en el paso 6, ya que la transacción B escribirá el nuevo balance de 500 Euros que ha calculado. Si esto sucede antes de la finalización de la transacción A el fenómeno se denomina "**Actualización Perdida**". Sin embargo, 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.

Si este escenario se implementa usando secuencias "SELECT … UPDATE" y se protege bloqueando el esquema, entonces en vez de tener el problema de la actualización perdida, el escenario se convierte en un INTERBLOQUEO (se explica más tarde). En este caso, por ejemplo, la transacción B debería deshacerse (ROLLBACK) por el SGBD y la transacción A podría finalizar correctamente.

El escenario puede implementarse usando actualizaciones sensibles (***sensitive updates***) basándose en los valores actuales como sigue:

```
UPDATE Accounts SET balance = balance – 200
WHERE acctID = 100;
```

Protegiéndolo mediante el bloqueo del esquema (explicado posteriormente), el escenario podría funcionar sin problemas.

2.1.2 Problema de la Lectura Sucia (*Dirty Read*)

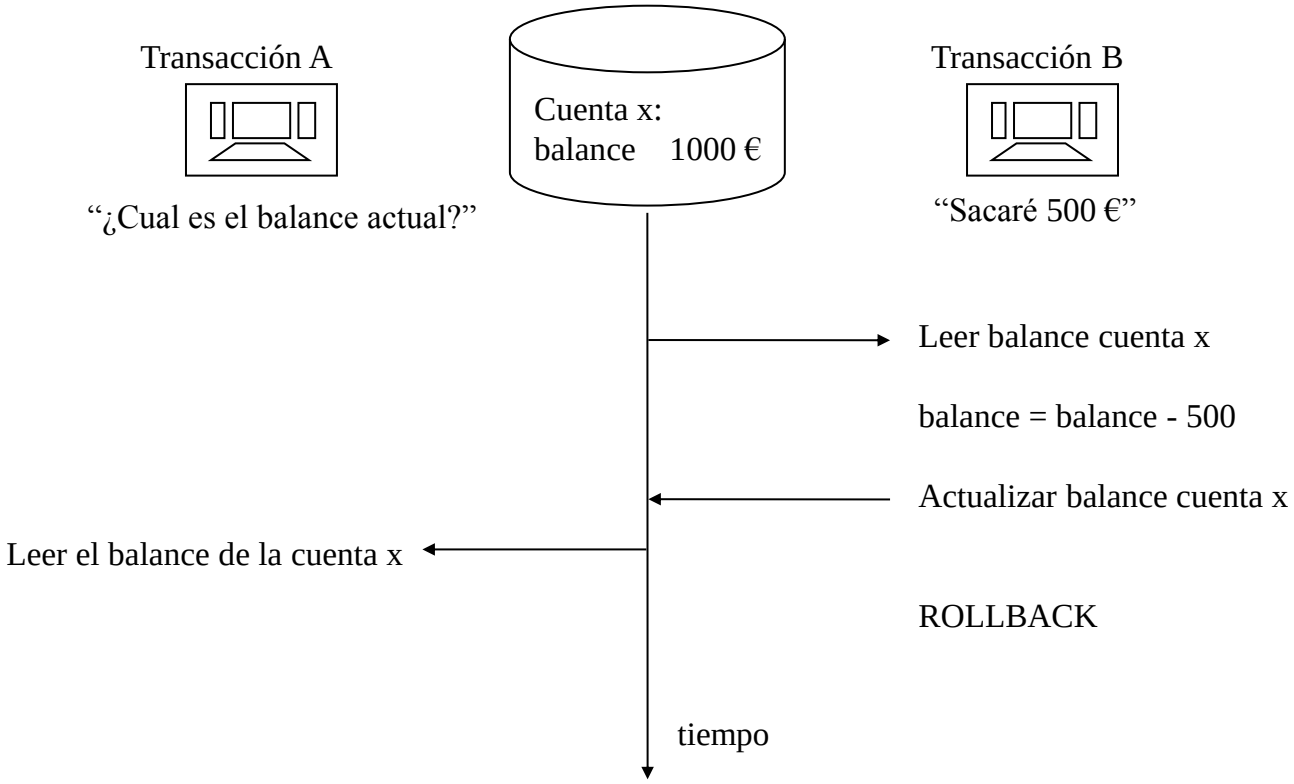


Figura 2.3. Ejemplo de una lectura sucia

La anomalía de lectura sucia presentada en la Figura 2.3 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). Este tipo de transacción no debe hacer ninguna actualización en la base de datos ya que podría llevar a datos corruptos. De hecho, cualquier uso de datos sin confirmar es arriesgado y puede llevar a decisiones o acciones incorrectas.

2.1.3 Problema de las No-Reproducibilidad

La anomalía de no-reproducibilidad presentada en la Figura 2.4 significa que los resultados de las consultas en la transacción no son estables, por lo que si las consultas deben repetirse, algunas de las tuplas previamente obtenidas podrían no estar disponibles. Esto no excluye tampoco el caso de que puedan aparecer nuevas tuplas en las consultas repetidas.

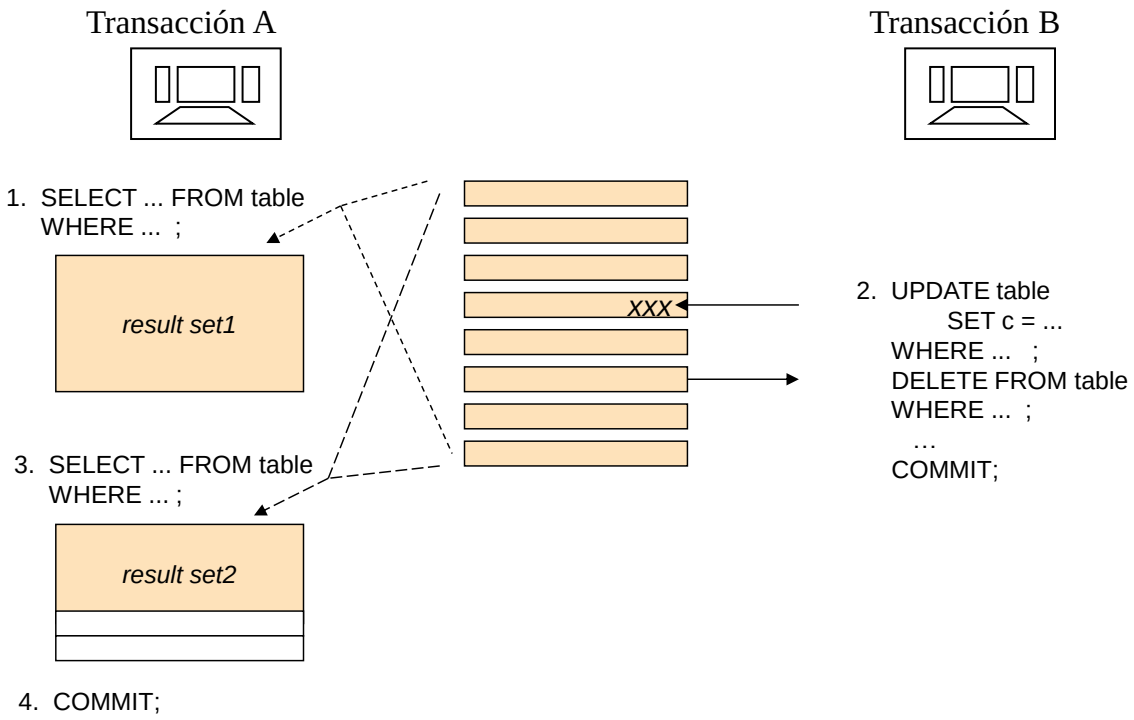


Figura 2.4. Problema de una lectura no-reproducible en la transacción A

2.1.4 Problema de la Aparición de Fantasmas

La anomalía de la lectura fantasma presentada en la Figura 2.5 significa que los resultados de las consultas en la transacción pueden incluir nuevas tuplas en consultas repetidas. Esto puede incluir tanto la inclusión de nuevas tuplas como la modificación de valores en una tupla que afecten a las condiciones de las consultas realizadas.

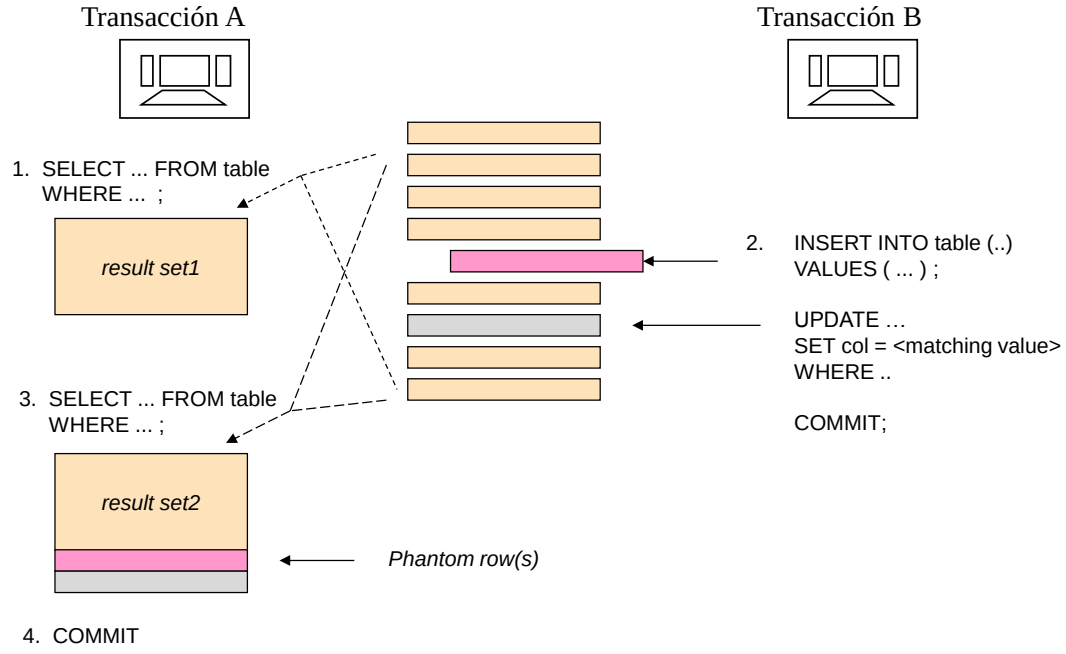


Figura 2.5. Ejemplo del problema de la lectura fantasma

1.8 2.2 El principio ACID de la Transacción Ideal

El principio ACID, presentado por Theo Härder y Andreas Reuter en 1983 en el ACM Computing Surveys, define la idea de fiabilidad de las transacciones SQL en un entorno multi-usuario. El acrónimo viene de las iniciales en inglés de las siguientes propiedades:

Atómico (**A**tomic)	Una transacción tiene que ser una serie atómica ("todo o nada") de operaciones que o bien tienen éxito y serán confirmadas (COMMIT) o todas las operaciones son deshechas (ROLBACK).
Consistente (**C**onsistent)	Las series de operaciones llevarán el estado de la base de datos de un estado consistente a otro. Por tanto, en el momento en que se confirme la transacción (COMMIT), las operaciones de la transacción no deben violar ninguna restricción de la base de datos (claves primarias, claves únicas, claves externas, comprobaciones) (*primary keys, unique keys, foreign keys, checks*). La mayoría de los SGBD aplican las restricciones inmediatamente a cada operación. La interpretación más restrictiva de la consistencia requiere que la lógica de la aplicación en la transacción tiene que ser correcta y adecuadamente probada (transacción bien-formada) incluyendo el manejo de excepciones.
Aislado (**I**solated)	La definición original de Härder y Reuter, "Los eventos dentro de una transacción deben ocultarse de otras transacciones concurrentes" (*"Events within a transaction must be hidden from other transactions running concurrently"*), no se satisfice en la mayoría de los SGBD como se explica en nuestro artículo "*Concurrency Paper*". Sin embargo debe ser considerado por los desarrolladores de aplicaciones. Los SGBD actuales usan varias técnicas de control de la concurrencia para proteger las transacciones concurrentes de los efectos de otras y los desarrolladores de aplicaciones deben conocer cómo usar estos servicios adecuadamente.
Permanente (**D**urable)	Los resultados confirmados en la base de datos sobrevivirán en el disco incluso ante posibles fallos del sistema.

El principio ACID requiere que una transacción que no cumpla con estas propiedades no debe ser confirmada, pero serán la aplicación o el servidor de bases de datos los encargados de deshacer dichas transacciones.

1.9 2.3 Niveles de Aislamiento

La propiedad de aislamiento del principio 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 base de datos.

El estándar ISO SQL no define cómo debe implementarse el control de la concurrencia, pero basándose en las anomalías derivadas de la concurrencia como los fenómenos de mal comportamiento ilustrados previamente, define los niveles de aislamiento que deben resolver estas anomalías como se define en la Tabla 2.1. La tabla 2.2 ilustra explicaciones breves de las restricciones de lectura resultantes de las configuraciones de nivel de aislamiento. Algunos de los niveles de aislamiento son menos restrictivos y algunos más restrictivos dando un mejor aislamiento, pero quizás a coste de bajar el rendimiento de la base de datos.

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.

Tabla 2.1. Niveles de aislamiento ISO SQL para resolver anomalías de concurrencia

Nivel de Aislamiento\Fenómeno	Actualización Perdida	Lectura Sucia	Lectura No-Reproducible	Fantasmas
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

Tabla 2.2. Niveles de aislamiento de ISO SQL (y DB2)

Nivel de aislamiento	**DB2 nivel aisl.**	**Explicación**
LECTURA NO CONFIRMADA	UR	Permite lecturas "sucias" de datos no confirmados escritos por transacciones concurrentes.
LECTURA CONFIRMADA	CS (CC)	No permite lecturas de datos sin confirmer de transacciones concurrentes. Oracle y DB2 (9.7 y posteriores) leerán la última versión confirmada de los datos (en DB2 esto se llama "*Currently Committed*", CC), mientras que algunos productos esperarán hasta que los datos sean confirmados.
LECTURA REPRODUCIBLE	RS	Permite la lectura solo de datos confirmados, y es posible repetir la lectura sin ningún UPDATE o DELETE hecho por una transacción concurrente en el conjunto de tuplas accedidas.
SERIALIZABLE	RR	Permite la lectura solo de datos confirmados, y es posible repetir la lectura sin ningún INSERT, UPDATE, o DELETE hecho por las transacciones concurrentes en el conjunto de tablas accedidas.

Nota1 Nótese la diferencia en el nombre de los niveles de aislamiento entre ISO SQL y DB2. DB2 definía originalmente 2 niveles de aislamiento: CS para Estabilidad de Cursores (*Cursor Stability*) y RR para Lectura Reproducible (*Repeatable Read*). Estos nombres no han cambiado incluso tras los 4 niveles de aislamiento propuestos por ISO SQL y tienen diferente semántica para Lecturas Reproducibles.

Nota2 Adicionalmente a los niveles de aislamiento de ISO SQL, en Oracle y SQL Server existe un nivel de aislamiento llamado **Snapshot**. En este caso la transacción ve solo una captura de los datos confirmados como estaban al comienzo de la transacción. Los cambios hechos por transacciones concurrentes no son visibles. Sin embargo, Oracle lo llama *SERIALIZABLE*.

Más adelante se discutirá qué niveles de aislamiento proporcionan los diferentes SGBD estudiados y cómo se implementan. Dependiendo del SGBD, los niveles de aislamiento pueden definirse como el nivel por defecto de la base de datos; como el nivel por defecto de las sesiones SQL al comienzo de una transacción; o incluso en algunos casos como instrucciones de ejecución a nivel de instrucción/tabla. Como buena práctica y de acuerdo con el estándar ISO SQL recomendamos que se configure el nivel de aislamiento al comienzo de cada transacción de acuerdo con las necesidades

de aislamiento de dicha transacción. De acuerdo con el estándar ISO SQL, Oracle y SQL Server definen el nivel de aislamiento con la siguiente sintaxis:

```
SET TRANSACTION ISOLATION LEVEL <isolation level>
```

pero en el caso de DB2 se usa la siguiente sintaxis:

```
SET CURRENT ISOLATION = <isolation level>
```

A pesar de las diferentes sintaxis y los diferentes nombres en cada SGBD, ODBC y JDBC hacen uso de los nombres definidos en ISO SQL. En el caso de JDBC el nivel de aislamiento se define como un parámetro del método de conexión `setTransactionIsolation`:

```
<connection>.setTransactionIsolation(Connection.<transaction isolation>);
```

en el que `<transaction isolation>` se define usando las palabras reservadas correspondientes al nivel de aislamiento (TRANSACTION_SERIALIZABLE por ejemplo para Serializable isolation). El nivel de aislamiento será relacionado por JDBC con el nivel correspondiente en el SGBD usado en cada caso. Si no existe correspondencia el driver JDBC lanzará una excepción de tipo SQLException.

1.102.4 Mecanismos de Control de la Concurrencia

Los SGBD modernos usan principalmente los siguientes mecanismos de Control de la Concurrencia para el aislamiento:

- *Esquema Multi-Granular basado en la Reserva o Esquema de Control de la Concurrencia basado en la Reserva (*llamado MGL del inglés *Multi-Granular Locking scheme*[3] o LSCC del inglés *Locking Scheme Concurrency Control*)
- *Control de la Concurrencia con Múltiples Versiones (*llamado MVCC *del inglés Multi-Versioning Concurrency Control*)
- *Control de la Concurrencia Optimista* (OCC *del inglés Optimistic Concurrency Control*).

[3] En la literatura algunos autores llaman a este esquema "*pessimistic concurrency control*" (PCC) y al *multi-versioning* lo llaman "*optimistic concurrency control*" aunque el mecanismo OCC tienen una semántica de concurrencia distinta.

2.4.1 Esquema de Control de la Concurrencia basado en la Reserva (LSCC)

La Tabla 2.3 define los esquemas de bloqueo básicos que el **gestor de bloqueo** del servidor de bases de datos usa automáticamente para proteger la integridad de los datos en operaciones de lectura y escritura. Debido a que sólo se permite una operación de escritura para una tupla de forma simultánea, el gestor de bloqueo intenta obtener un bloqueo exclusivo (**X-lock**) en las tuplas afectadas por una operación de escritura como INSERT, UPDATE, o DELETE. Un bloque exclusivo se permitirá únicamente cuando no exista ninguna otra transacción que tenga un bloqueo sobre los mismos recursos. Una vez obtenido el bloqueo exclusivo éste **se mantendrá hasta el final de la transacción**.

El gestor de bloqueo protege la integridad de las operaciones de lectura, como SELECT, mediante S-locks, que pueden concederse a múltiples clientes concurrentes que estén leyendo, ya que no se molestarán mutuamente. Esto puede depender del nivel de aislamiento elegido por la transacción. El nivel de aislamiento LECTURA NO CONFIRMADA (del inglés *READ UNCOMMITTED*) no requiere protección **S-lock** para realizar la lectura, pero en otros casos de nivel de aislamiento, este bloqueo puede ser necesario para realizar la lectura y será concedido siempre que no haya otra transacción con un bloqueo concedido de tipo X-lock en las tuplas afectadas.

Tabla 2.3 Compatibilidad de S-locks y X-locks

Cuando una transacción necesita un bloqueo de este tipo para una tupla	Cuando otra transacción ya tiene un bloqueo sobre la misma tupla de este tipo		Cuando ninguna otra transacción tiene un bloque sobre la tupla
	S-lock	X-lock	
S-lock	Bloque concedido	Esperar a liberación	Bloque concedido
X-lock	Esperar a liberación	Esperar a liberación	Bloque concedido

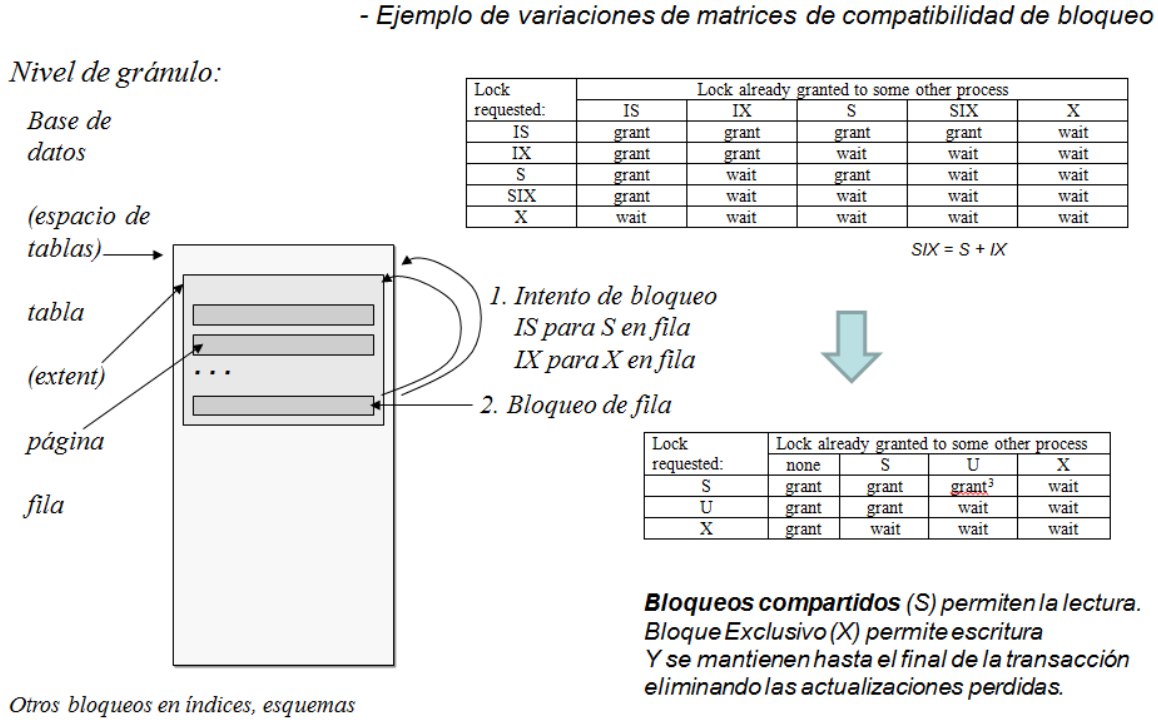
En el caso de aislamiento de tipo LECTURA CONFIRMADA (del inglés READ COMMITTED) el bloqueo compartido de una tupla se liberará justo después de la lectura de la fila, mientras que para LECTURA REPETIBLES (del inglés REPEATABLE READ) y SERIALIZABLE, el bloqueo se mantendrá hasta el final de la transacción. Todos los bloqueos de la transacción se liberan al final de la misma[4] independientemente de cómo haya finalizado la transacción.

Nota En algunos SGBD el dialecto SQL usado incluye comandos de bloqueo explícito de las tablas, y en estos casos el bloqueo es también liberado de forma implícita al finalizar la transacción, e incluso antes en el caso de bloqueo compartido en el caso de aislamiento LECTURA CONFIRMADA.

No existen comandos de desbloqueo de tabla explícitos en los dialectos SQL salvo en MySQL/InnoDB.

[4] Excepto de la tupla actual que mantenga el cursor en algunos productos como DB2.

Los esquemas de bloqueo usados por los SGBD son realmente más complicados, usando bloqueos a diferentes granularidades como tupla, página, índices, esquemas, etc. Además se usan modos de bloqueo a parte de los compartidos y exclusivos comentados. Para las peticiones de bloqueo de tuplas los gestores de bloqueo generan primero el correspondiente intento de bloqueo a nivel de granulo mayor, lo que permite llevar el control del bloqueo a distintos niveles con el esquema **Multi-Granular basado en la Reserva** (**MGL**) como se describe en la Figura 2.6.



Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

Lock requested:	Lock already granted to some other process			
	none	S	U	X
S	grant	grant	grant[3]	wait
U	grant	grant	wait	wait
X	grant	wait	wait	wait

Figura 2.6. Control de compatibilidad de bloqueo a diferentes niveles de granularidad

El protocolo de bloqueo podrá resolver el problema de las actualizaciones perdidas, pero si las transacciones en competencia usan un nivel de aislamiento que mantiene los bloqueos compartidos hasta el final de la transacción, entonces el bloqueo pasará a producir otro problema como se muestra en la Figura 2.7. Ambas transacciones están esperando en un ciclo sin fin de espera llamado **Interbloqueo**. En los primeros SGBD este era un problema crucial, pero los productos modernos incluyen una hebra llamada **Detector de Interbloqueo** que normalmente se activa cada 2 segundos (el tiempo de activación puede configurarse) para buscar interbloqueos y cuando encuentra uno selecciona una víctima de las transacciones en espera y la deshace automáticamente.

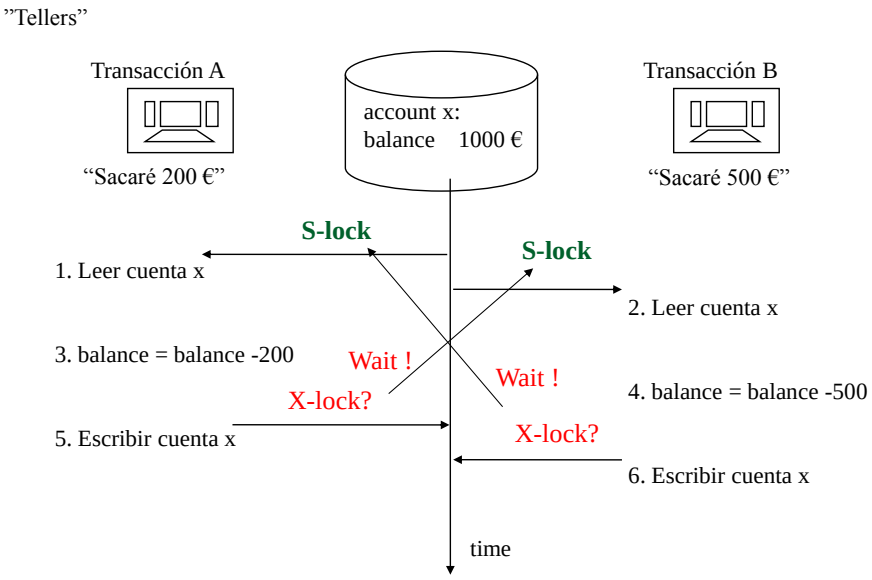


Figura 2.7. Problema de actualización perdida resuelto mediante LSCC pero llevando a un estado de interbloqueo.

La aplicación de la víctima seleccionada obtendrá un mensaje de excepción de interbloqueo y deberá reintentar la transacción tras un tiempo de espera aleatorio y corto. Ver el patrón de reintento en el código Java del ejemplo BankTransfer del Apéndice 2.

Nota Se debe recordar que un SGBD no puede reiniciar automáticamente la víctima en caso de interbloqueo, y es responsabilidad de la aplicación o del servidor de aplicaciones en el que el cliente esté instalado. Es importante también tener en cuenta que un interbloqueo no es un error y que el abortar la transacción víctima es un servicio proporcionado por el servidor, por lo que las aplicaciones clientes pueden seguir funcionando en el caso de que las transacciones concurrentes no puedan simplemente proseguir.

2.4.2 Control de la Concurrencia con Múltiples Versiones (MVCC)

En la técnica MVCC, el servidor mantiene una cadena histórica ordenada por marcas de tiempo de las versiones actualizadas de las tuplas. De esta forma, puede usarse una versión confirmada al comienzo de cada transacción para cada tupla actualizada. Esta técnica de control de la concurrencia elimina los tiempos de espera en las lecturas y proporciona 2 niveles de aislamiento: cualquier transacción con nivel **`LECTURA CONFIRMADA`** obtendrá las versiones de las tuplas actualizadas más recientemente de la cadena histórica, y las transacciones con nivel **`INSTANTÁNEA`** verán las últimas versiones en su estado al comienzo de la transacción o aquellas escritas por la propia transacción. En nivel INSTANTÁNEA la transacción nunca ve las filas fantasma y puede incluso evitar que las transacciones concurrentes escriban en filas fantasma, mientras que SERIALIZABLE basado en LSCC (MGL) evita que transacciones concurrentes escriban en tuplas fantasma en la base de datos. De hecho, la transacción continúa viendo las filas fantasma ("*ghost*") de la instantánea (aquellas filas que han sido eliminadas por transacciones concurrentes).

A pesar del nivel de aislamiento, la escritura es protegida típicamente incluso en sistemas MVCC mediante algún mecanismo de bloqueo de tuplas[5]. La actualización de tuplas cuyos contenidos hayan sido actualizados por otras transacciones generará algunos mensajes de error indicando que la instantánea es demasiado antigua.

Implementaciones de MVCC

Como ejemplo de implementación de MVCC la Figura 2.8 presenta el mecanismo usado por Oracle. Oracle llama al nivel de aislamiento INSTANTÁNEA, SERIALIZABLE.

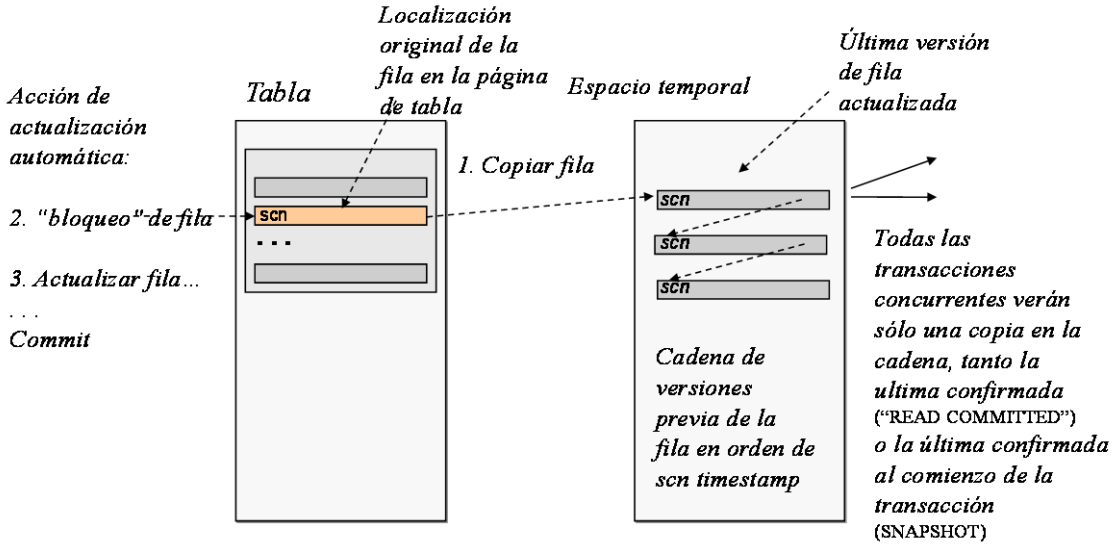


Figura 2.8. MVCC

En MVCC de Oracle, la primera transacción en escribir una tupla (es decir en insertar, actualizar o borrar) obtendrá una especie de bloqueo en la tupla y ganará la competencia por la escritura y los escritores concurrentes se ponen en una cola de espera. Los bloqueos de tupla se implementan mediante el marcado de las tuplas escritas mediante Número de Cambios del Sistema (*System Change Numbers, SCN*) de la transacción (números de secuencia de las transacciones iniciadas). En cuanto al SCN de una fila pertenece a una transacción activa, esa tupla queda bloqueada por esa transacción. El uso de bloqueos de escritura significa que podrían producirse interbloqueos, pero en vez de interrumpir automáticamente a las víctimas del interbloqueo, Oracle detecta inmediatamente el bloqueo de tupla que conduciría a un interbloqueo, lanza una excepción a la aplicación cliente y espera a que el cliente resuelva el interbloqueo mediante un comando ROLLBACK explícito.

La técnica de control de concurrencia de Oracle puede llamarse CC híbrida ya que además de MVCC con bloqueo de tupla implícito, Oracle ofrece comandos "LOCK TABLE" explícitos de bloqueo y también bloqueo explícito de tuplas mediante la sentencia

```
"SELECT ... FOR UPDATE"
```

que proporciona formas de evitar incluso filas fantasma invisibles. En Oracle una transacción puede declararse también de modo Sólo Lectura.

Microsoft también se ha dado cuenta de los beneficios de MVCC, y desde la versión SQL Server 2005, es posible configurar SQL Server para utilizar versiones de tuplas mediante la configuración de

[5] SolidDB es un SGBD que usa MVCC sin bloquear las operaciones de escritura. En SolidDB el primero en tratar de escribir es el que lo consigue.

propiedades de la base de datos usando comandos de Transact-SQL. Desde la versión 2012 se puede realizar mediante propiedades de la base de datos tal como muestra la Figura 2.9.

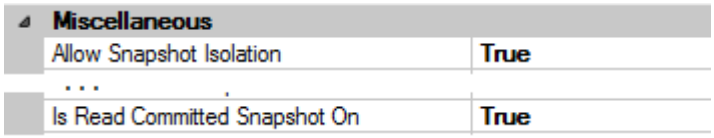


Figura 2.9 Configuración de SQL Server 2012 para dar soporte a Instantáneas

El mecanismo de control de la concurrencia de MySQL / InnoDB es un híbrido CC que proporciona los siguientes cuatro niveles de aislamiento para lectura:

- Lectura no confirmada sin bloqueo o versionado de tuplas,
- Lectura confirmada (realmente "*Read Latest Committed*") usando MVCC,
- Lectura reproducible (realmente Instantánea) usando MVCC, y
- Serializable usando MGL CC con X- y S-locks, previniendo las tuplas fantasma.

2.4.3 Control de la Concurrencia Optimista (OCC)

En la OCC original, todos los cambios realizados por la transacción se mantienen aparte de la base de datos y se sincronizan con la base de datos sólo en la fase de confirmación. Esto ha sido implementado, por ejemplo, en Pyrrho de la Universidad de West of Scotland. El único e implícito nivel de aislamiento disponible en el SGBD Pyrrho es SERIALIZABLE (ver http://www.pyrrhodb.com).

* * *

Resumen:

El estándar ISO SQL desarrollado por ANSI está basado originalmente en la versión del lenguaje SQL de DB2, que IBM donó a ANSI. El mecanismo de control de la concurrencia en DB2 utiliza un esquema de bloqueo Multi-Granular y en aquel momento DB2 sólo tenía 2 niveles de aislamiento: Estabilidad del cursor (CS) y la Lectura repetible (RR). Obviamente esto tuvo influencia en la semántica de los nuevos niveles de aislamiento (véase la Tabla 2.2) definidos por ANSI / ISO SQL, que puede entenderse en términos de bloqueo compartido.

El estándar SQL no dice nada acerca de las implementaciones, así que también se han aplicado otros mecanismos de concurrencia y los mismos nombres de aislamiento se han utilizado con diferentes interpretaciones, como hemos visto anteriormente. Bajo el título "Niveles de aislamiento" en la tabla 2.4, los niveles de aislamiento (en fondo azul) Lectura no confirmada, Lectura Confirmada, Lectura Reproducible y Serializable representan los niveles de aislamiento como los entendemos según la semántica del estándar SQL. Los mismos nombres entre comillas en las columnas de los SGBD, por ejemplo "serializable", indican diferentes semánticas que los niveles de aislamiento del estándar. El nombre de nivel de aislamiento "Read latest committed" ha sido inventado por nosotros, ya que se utilizan varios nombres en los SGBD para el nivel de aislamiento en el que la operación de lectura se realiza sin las esperas de bloqueo. Esta semántica basada en instantáneas ha aumentado la confusión, y los conceptos de aislamiento utilizados en el estándar podrían necesitar aclaraciones y ampliaciones. Un grupo de desarrolladores del estándar ANSI / SQL y escritores de libros de texto de bases de datos confiesan el problema en su artículo "Una crítica de los niveles de aislamiento ANSI SQL" (Berenson et al, 1995).

Aplicando el Ejercicio 2.7 de los próximos ejercicios prácticos utilizando diferentes SGBD se muestran algunos problemas de instantáneas en el caso de la escritura en la base de datos, ya que la escritura va a violar la coherencia de la instantánea. El uso seguro de la instantánea, por tanto, estaría limitado al uso en la generación de informes. La Tabla 2.4 resume también algunas otras diferencias entre los SGBD.

Tabla 2.4 Funcionalidades de transacción proporcionadas en el estándar ISO/SQL y en los SGBD

	ANSI/ISO SQL	DB2	Oracle	SQL SERVER	MySQL/InnoDB	PostgreSQL	Pyrrho
	SQL:2006	LUW 9.7	12g1	2012	5.6	9.2	4.8
autocommit (server-side)	n/a	n/a	n/a	yes	yes	yes	yes
Transaction Limits							
implicit start	yes	yes	yes	(configurable)	(configurable)		
explicit start				yes	yes	yes	yes
implicit commit on DDL	n/a	n/a	yes	n/a	yes	n/a	n/a
COMMIT	yes	yes	yes	yes	yes	yes	yes
COMMIT WORK	yes	yes		yes	yes	yes	n/a
ROLLBACK	yes	yes	yes	yes	yes	yes	yes
SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
ROLLBACK TO SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
RELEASE SAVEPOINT	yes	yes	yes	n/a	yes	yes	n/a
Isolation levels							
Read Uncommitted	yes	UR	n/a	yes	yes	n/a	n/a
"read latest committed"	n/a	CS (currently committed)	"read committed"	(configurable)	"read committed"	"read committed"	n/a
Read Committed	yes	CS	n/a	yes	n/a	n/a	n/a
Repeatable Read	yes	RS	n/a	yes	n/a	n/a	n/a
snapshot		n/a	"serializable"	(configurable)	"repeatable read"	"serializable"	"serializable"
Serializable	yes	RR	explicit locking	yes	yes	explicit locking	n/a
CC mechanism	n/a						
MGL (locking)		yes		yes	yes		
MVCC (versioning)			yes	(configurable)	yes	yes	
OCC							yes
Cursor processing							
- WITH HOLD	yes	yes		default			
- optimistic locking				yes			

Instantánea significa una "vista" consistente de la base de datos al inicio de la transacción. Como tal, se adapta perfectamente a transacciones de sólo lectura, ya que en ese caso la transacción no bloquea las transacciones concurrentes. El uso de la instantánea no elimina la existencia de fantasmas, ya que simplemente no están incluidos en la instantánea. En SGBD con MVCC, como por ejemplo, en Oracle y PostgreSQL, la semántica de ISO SQL Serializable que impide la existencia de fantasmas se puede implementar usando adecuadamente bloqueos a nivel de tabla. En el caso de aislamiento de instantánea, todos los SGBD permiten comandos INSERT, pero los servicios en otras operaciones de escritura parecen variar.

En la tabla 2.4 se puede observar que DB2 es el único SGBD en nuestro DebianDB que no admite el aislamiento de instantánea.

2.5 Laboratorio Práctico

Iniciamos una sesión MySQL como de costumbre (Figura 2.10):

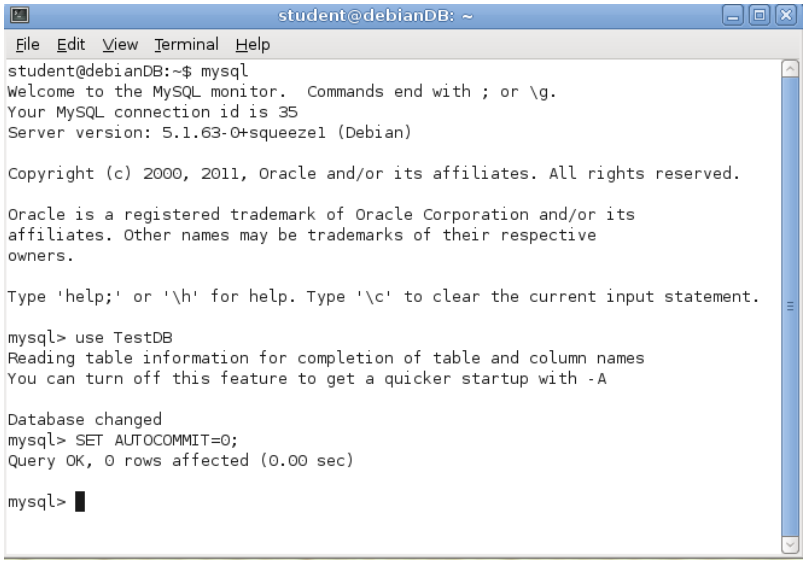


Figura 2.10 Inicio de una sesión MySQL

Ejecicio 2.1 Para las pruebas de concurrencia necesitamos un Segundo terminal con otra sesión de MySQL como se muestra en la Figura 2.10. A la izquierda tenemos la Sesión A y la derecha la sesión B. Comenzamos ambas desconectando el modo AUTOCOMMIT:

```
-----------------------------------
use TestDB
SET                                                    AUTOCOMMIT=0;
-----------------------------------
```

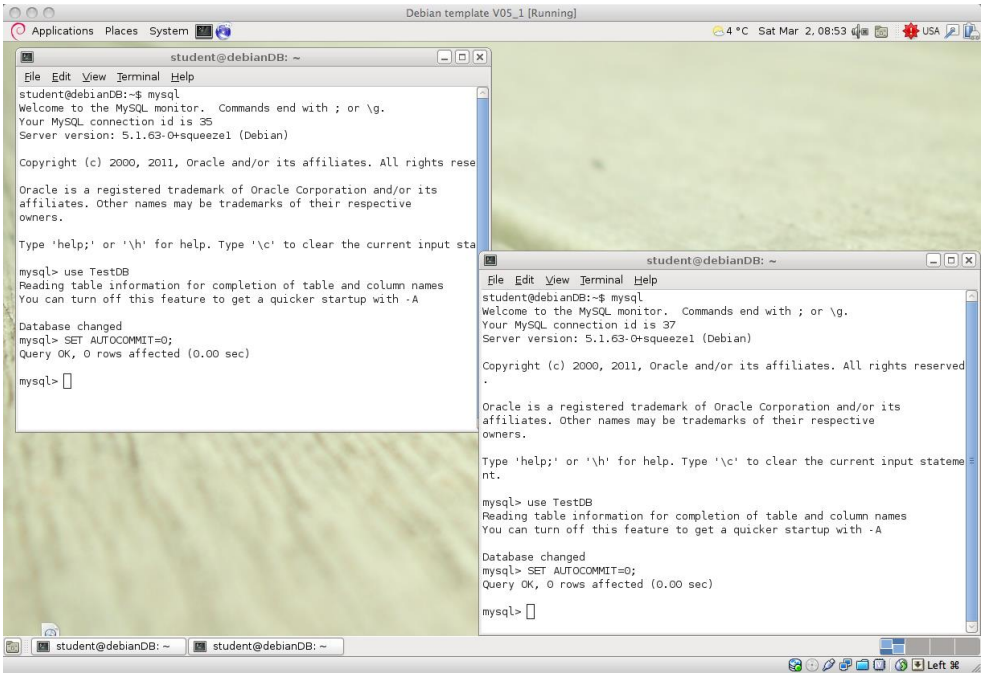


Figura 2.11 Dos sesiones concurrentes

Las transacciones concurrentes pueden bloquearse, como veremos, debido al acceso a los mismos datos. Por tanto, debemos diseñar las transacciones lo más cortas posibles para hacer el trabajo necesario. La inclusión en las transacciones de un dialogo con el usuario final puede llevar a tiempos

de espera catastróficos para el entorno de producción. Por lo tanto, **las transacciones no deben dar el control al interfaz de usuario hasta que la transacción haya finalizado.**

Nota:
Ningún SGBD puede reiniciar de forma automática la víctima abortada en caso de interbloqueo. El reinicio de la transacción es responsabilidad del código de la aplicación o del servidor de aplicaciones que aloja el servicio de acceso a datos. Es importante destacar también que el interbloqueo no es un error y que el parar la transacción es un servicio del servidor por lo que las aplicaciones podrán continuar funcionando pese al interbloqueo.

Consejo:
Se suele pensar que los interbloqueos y la forma en que se gestionan en los SGBDs son las principales fuentes de tiempos de respuesta lentos en el servidor de bases de datos.
Se ha mostrado ya que las transacciones que se ejecutan concurrentemente y acceden a los mismos datos pueden bloquearse mutuamente. Por lo tanto, las transacciones deben diseñarse para ser lo más cortas posibles, para realizar trabajos simples. La inclusión de diálogos con el usuario final en la lógica de la transacción puede llevar a tiempos de espera catastróficos para el entorno de producción. Por ello, **nunca se debe pasar el control de la transacción a diálogos en las interfaces de usuario**.

El nivel de aislamiento actual (por defecto) se puede conocer usando una sentencia SELECT:

```
-----------------------------------
SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
-----------------------------------
```

Parece que por defecto MySQL tiene un nivel de aislamiento REPEATABLE READ tanto a nivel global como local.

Por seguridad, vamos a borrar la tabla *Accounts* y la volvemos a crear y rellenar con datos:

```
-----------------------------------
DROP TABLE Accounts;
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT       remains_nonnegative       CHECK       (balance       >=       0)
);
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Ejercicio 2.1 Como se ha explicado anteriormente, el problema de la actualización perdida implica sobrescribir el valor actualizado de una fila por otra transacción (que se ejecuta simultáneamente), antes de la terminación de dicha transacción. Todos los SGBD modernos con servicios de control de la concurrencia impiden esto, por lo que el problema es imposible de reproducir en las pruebas. Sin embargo, después de la operación de confirmación cualquier transacción simultánea descuidada puede sobrescribir los resultados sin antes leer el valor actual confirmado. Llamamos a este caso "Sobreescritura Ciega" o "Escritura Sucia", y es incluso demasiado fácil de producir.
La situación de sobreescritura ciega se produce, por ejemplo, cuando el programa de aplicación lee los valores de la base de datos, actualiza los valores en memoria, y luego escribe el valor actualizado de nuevo en la base de datos. En la tabla siguiente se simula la parte del programa de aplicación

utilizando variables locales en MySQL. Una variable local es sólo un marcador de posición nombrado en la memoria de trabajo de la sesión actual. Se identifica por el símbolo de arroba (@) añadido al principio de su nombre, y puede estar incrustado en la sintaxis de instrucciones SQL, siempre que se esté seguro de que siempre recibirá un solo valor escalar.

La tabla 2.4 muestra el orden de las sentencias SQL que se ejecutarán en este ejercicio. El objetivo es simular la situación de actualización perdida ilustrada en la Figura 2.2: la transacción en la Sesión A consiste en retirar € 200 de la cuenta 101, y la transacción en la Sesión B es la retirada de € 500 desde la misma cuenta y se sobrescribirá el balance de la cuenta (perdiendo la información sobre los 200 € retirados por la transacción A). Así, el resultado de la sobrescritura ciega es el mismo que el resultado de un problema de actualización perdida.

Nota:
En el paso número 4, se debe tener en cuenta que en MySQL el tiempo de espera de bloqueo predeterminado es de 90 segundos. Por lo tanto, la transacción (cliente) A debe proceder al paso 5, sin demora, tras el paso número 4 de la transacción B.

Restauramos la tabla *Accounts*:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- amount to be transfered by A SET @amountA = 200; SET @balanceA = 0; -- init value SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101; SET @balanceA = @balanceA - @amountA; SELECT @balanceA;	
2		SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- amount to be transfered by B SET @amountB = 500; SET @balanceB = 0; -- init value SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101; SET @balanceB = @balanceB - @amountB;
3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;

Tabla 2.4 Problema de la Sobreescritura Ciega, simulado usando variables locales

Teniendo en cuenta la ejecución concurrente (entrelazada, para ser más exactos) de las transacciones A y B en la Tabla 2.4, A se prepara para retirar € 200 desde la cuenta bancaria en el paso 1 (no habiendo actualizado la base de datos aún), B se prepara para retirar € 500 en el paso 2, A actualiza la base de datos en el paso 3, B actualiza la base de datos en el paso 4, A comprueba el balance de la cuenta bancaria para asegurarse de que es como se espera antes de que se confirme en el paso número 5, y B hace lo mismo en el paso número 6.

Preguntas:

a) ¿Se comporta el sistema de la forma esperada?
b) ¿Hay evidencia de los datos perdidos en este caso?

Nota:
Todos los SGBD implementan mecanismos de control de la concurrencia para que en todos los niveles de aislamiento nunca aparezca la anomalía de actualización perdida . Sin embargo, siempre existe la posibilidad de que por descuido se den códigos de aplicación donde operaciones de 'sobreescritura ciega' efectivamente produzcan los mismos efectos catastróficos que con la anomalía de actualización. En la práctica, esto es como tener la anomalía de pérdida de actualización en escenarios de transacciones concurrentes entrando por la puerta trasera.

Ejercicio 2.2 Repetimos el ejercicio 1 pero ahora usando el nivel de aislamiento SERIALIZABLE.

Primero el contenido original de la tabla es restaurado:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; -- amount to be transfered by A SET @amountA = 200; SET @balanceA = 0; -- init value SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101; SET @balanceA = @balanceA - @amountA; SELECT @balanceA;	
2		SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; -- amount to be transfered by B SET @amountB = 500; SET @balanceB = 0; -- init value SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101; SET @balanceB = @balanceB - @amountB;
3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;

Tabla 2.5 Escenario 2.1 manteniendo los S-locks

Preguntas:

a) ¿Concusiones obtenidas?
b) ¿Qué sucede si se reemplaza ‘SERIALIZABLE’ por ’REPEATABLE READ’ en ambas transacciones?

Nota:
mySQL implementa REPEATABLE READ mediante MVCC, y SERIALIZABLE mediante MGL/LSCC

Ejercicio 2.2b Competencia SELECT-UPDATE sobre el mismo recurso.

Repetimos el ejercicio 2.2 pero usando "*sensitive updates*" en los escenarios SELECT-UPDATE sin variables locales.

Restauramos la tabla:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; SELECT balance FROM Accounts WHERE acctID = 101;	
2		SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; SELECT balance FROM Accounts WHERE acctID = 101;
3	UPDATE Accounts SET balance = balance – 200 WHERE acctID = 101;	
4		UPDATE Accounts SET balance = balance – 500 WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;

Tabla 2.5b Competencia SELECT-UPDATE sobre lo smismos recursos

Pregunta:

- ¿Qué conclusiones se obtienen?

Ejercicio 2.3 Competencia de UPDATE-UPDATE en dos recursos en diferente orden

Reiniciamos la tabla como de costumbre:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;	
2		SET AUTOCOMMIT=0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; UPDATE Accounts SET balance = balance -200 WHERE acctID = 202;
3	UPDATE Accounts SET balance = balance+100 WHERE acctID = 202;	
4		UPDATE Accounts SET balance = balance + 200 WHERE acctID = 101;
5	COMMIT;	
6		COMMIT;

Tabla 2.6 Escenario 2.1 manteniendo los S-locks

Pregunta:

- ¿Que conclusiones se obtienen?

Nota: El nivel de aislamiento no tiene ningún papel en este escenario, ¡pero es una buena práctica definir siempre el nivel de aislamiento al comienzo de cada transacción! Puede haber algún procesamiento oculto, por ejemplo por las comprobaciones de clave externa y disparadores usando el acceso para leer. El diseño de los disparadores es en realidad un problema de los administradores de bases de datos, y no está en el ámbito de este tutorial.

Ejercicio 2.4 Para continuar con las anomalías de transacción, hacemos ahora un intento para producir la aparición de una situación de lectura sucia. Una transacción A se ejecuta en modo (por defecto en MySQL) REPEATABLE READ, mientras que la transacción B se configura para ejecutarse a nivel READ UNCOMMITTED:

Reiniciamos la tabla de cuentas:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;	
2		SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT * FROM Accounts; COMMIT WORK;
3	ROLLBACK; SELECT * FROM Accounts; COMMIT;	

Tabla 2.7 Problema de lectura sucia

Preguntas

a) ¿Conclusiones?
b) ¿Y si reemplazamos 'READ UNCOMMITTED' por 'READ COMMITTED' en la transacción B?
c) ¿Y si reemplazamos 'READ UNCOMMITTED' por 'REPEATABLE READ' en la transacción B?
d) ¿Y si reemplazamos 'READ UNCOMMITTED' por 'SERIALIZABLE' en la transacción B?

Ejercicio 2.5 A continuación veamos la anomalía de lectura no repetible:

Restauramos la tabla:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; SELECT * FROM Accounts WHERE balance > 500;	
2		SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202; COMMIT WORK;
3	-- repeating the same query SELECT * FROM Accounts WHERE balance > 500; COMMIT;	

Tabla 2.8 Problema de lectura no reproducible

Preguntas:

a) ¿Lee la transacción A los mismos resultados en el paso 3 y el paso 1?
b) ¿Qué sucede si se establece el nivel de aislamiento de la transacción A como LECTURA REPRODUCIBLE?

Ejercicio 2.6 Se intenta ahora producir el ejemplo típico de los libros de texto de '*insert phantom*':

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT WORK;
-----------------------------------
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;	
2	-- Accounts having balance > 1000 euros; SELECT * FROM Accounts WHERE balance > 1000;	
3		SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; INSERT INTO Accounts (acctID, balance) VALUES (303,3000); COMMIT;
4	-- Can we see the new account 303? SELECT * FROM Accounts WHERE balance > 1000; COMMIT;	

Tabla 2.9 Problema "Insert phantom"

Preguntas:

a) ¿Tiene que esperar la transacción B a la A?
b) ¿Es el acctID=303 que inserta la transacción B visible en el entorno de la transacción A?
c) ¿Afecta esto al resultado del paso 4 si cambiamos el orden de los pasos 2 y 3?
d) MySQL/InnoDB usa Multi-Versioning para el nivel de aislamiento LECTURA REPRODUCIBLE pero, ¿cuál es el nivel de tiempo de la instantánea leída?
e) ¿Qué hemos aprendido de esta prueba?

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
ERROR 1568 (25001): Transaction characteristics can't be changed while a transaction is in progress
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
Query OK, 0 rows affected (0.00 sec)
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
.
```

Ejercicio 2.7 Un estudio de SNAPSHOT con diferentes tipos de fantasmas

Creamos una nueva tabla:

```
DROP TABLE T;
-- Setup the test
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first',  1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;
```

Paso	Sesión A	Sesión B
1	SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ; SELECT * FROM T WHERE i = 1;	
2		SET AUTOCOMMIT = 0; SET TRANSACTION ISOLATION LEVEL READ COMMITTED; INSERT INTO T (id, s, i) VALUES (6, 'Insert Phantom', 1); UPDATE T SET s = 'Update Phantom', i = 1 WHERE id = 2; DELETE FROM T WHERE id = 5; SELECT * FROM T;
3	-- Let's repeat the query and try some updates SELECT * FROM T WHERE i = 1; INSERT INTO T (id, s, i) VALUES (7, 'inserted by A', 1); UPDATE T SET s = 'update by A inside snapshot' WHERE id = 3; UPDATE T SET s = 'update by A outside snapshot' WHERE id = 4; UPDATE T SET s = 'update by A after B' WHERE id = 1;	
4		COMMIT;
5	SELECT * FROM T WHERE i = 1; UPDATE T SET s = 'updated after delete?' WHERE id = 5; SELECT * FROM T WHERE i = 1;	
6	COMMIT; SELECT * FROM T;	

Tabla 2.10 Problemas de fantasma al insertar o actualizar, o bien borrar tuplas

Preguntas:

a) ¿Son los *insert* y *update* que realiza la transacción B visibles a la transacción A?
b) ¿Qué sucede si A trata de actualizar la tupla 1 actualizada por la transacción B?
c) ¿Qué sucede al tratar A de actualizar la tupla 3 que ha sido eliminada por la transacción B?
d) Comprar estos resultados con escenarios similares en SQL Server
e) ¿Cómo se pueden prevenir los fantasmas mientras la transacción A está activa?

Nota: El SELECT en una transacción MySQL/InnoDB que se ejecuta usando el nivel de aislamiento REPEATABLE READ crea una instantánea consistente. Si la transacción manipula tuplas en las tablas base de la instantánea, entonces ésta ya no es consistente.

Listado 2.1 Resultados en una prueba de ejecución

```
mysql> -- step 5
mysql> SELECT * FROM T WHERE i = 1;
+----+-----------------------------+------+
| id | s                           | i    |
+----+-----------------------------+------+
|  1 | update by A after B         |    1 |
|  3 | update by A inside snapshot |    1 |
|  5 | to be or not to be          |    1 |
|  7 | inserted by A               |    1 |
+----+-----------------------------+------+
4 rows in set (0.01 sec)

mysql> UPDATE T SET s = 'updated after delete?'
    ->  WHERE id = 5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> SELECT * FROM T WHERE i = 1;
+----+-----------------------------+------+
| id | s                           | i    |
+----+-----------------------------+------+
|  1 | update by A after B         |    1 |
|  3 | update by A inside snapshot |    1 |
|  5 | to be or not to be          |    1 |
|  7 | inserted by A               |    1 |
+----+-----------------------------+------+
4 rows in set (0.00 sec)

mysql> -- step 6
mysql> COMMIT;
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM T;
+----+-------------------------------+------+
| id | s                             | i    |
+----+-------------------------------+------+
|  1 | update by A after B           |    1 |
|  2 | Update Phantom                |    1 |
|  3 | update by A inside snapshot   |    1 |
|  4 | update by A outside snapshot  |    2 |
|  6 | Insert Phantom                |    1 |
|  7 | inserted by A                 |    1 |
+----+-------------------------------+------+
6 rows in set (0.00 sec)
```

Nota: Al final del Apéndice 1 tenemos los resultados del ejercicio 2.7 usando SQL Server 2012 para su comparación.

Parte 3 Algunas Buenas Prácticas

Una transacción de usuario normalmente tiene múltiples diálogos con la base de datos. Algunos de estos diálogos sólo recopilarán datos de la base de datos, dando soporte a la transacción de usuario. Como paso final de la transacción de usuario, algunos botones "Guardar" activarán una transacción SQL que actualizará la base de datos.

Las transacciones SQL, aún en la misma secuencia de la transacción de usuario, pueden tener diferentes requisitos de fiabilidad y de aislamiento. Se debe definir siempre el nivel de aislamiento de la transacción al comienzo de cada transacción.

De acuerdo con el estándar SQL, el nivel de aislamiento LECTURA NO CONFIRMADA sólo puede utilizarse en las transacciones SÓLO LECTURA (Melton y Simon, 2002), pero los SGBD no fuerzan esto.

Los SGBD difieren unos de otros en términos de los servicios de control de concurrencia y el comportamiento de la gestión de transacciones, por lo que es importante para la fiabilidad y el rendimiento que el desarrollador de la aplicación conozca el comportamiento del SGBD a utilizar.

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 el aislamiento SERIALIZABLE con la semántica ISO / SQL debe utilizarse si el desarrollador no puede decidir qué nivel de aislamiento proporciona un aislamiento suficientemente fiable. Es importante entender que el aislamiento INSTANTÁNEA garantiza sólo conjuntos de resultados consistentes, pero no conserva el contenido base de datos. Si usted no puede permitir los fantasmas y su SGBD sólo soporta los niveles de aislamiento de instantáneas, es necesario estudiar las posibilidades de bloqueos explícitos.

Las transacciones SQL no deben contener ningún diálogo con el usuario final, ya que esto ralentizaría el procesamiento. Dado que las transacciones SQL pueden ser deshechas durante la transacción, no deberían afectar a otra cosa que a la base de datos. La transacción SQL debe ser tan corta como sea posible, para minimizar la competencia de la concurrencia y el bloqueo de las transacciones simultáneas.

Evite comandos DDL en las transacciones. Los COMMITS implícitos debidos a DDL pueden dar lugar a transacciones involuntarias.

Cada transacción de SQL debe tener una tarea bien definida, empezando y terminando en el mismo componente de la aplicación. En este tutorial no cubriremos el tema del uso de transacciones SQL en procedimientos almacenados, ya que los lenguajes de procedimientos almacenados y sus implementaciones son diferentes en los distintos SGBD. Algunos de los SGBD en nuestro laboratorio DebianDB no permiten sentencias COMMIT en las rutinas almacenadas. Sin embargo, la transacción puede ser forzada a rollback incluso en mitad de alguna rutina almacenada, y esto tiene que gestionarse también en el código de llamada en la aplicación.

El contexto técnico de una transacción SQL es una conexión de base de datos única. Si una transacción falla debido a un conflicto de concurrencia, se debe en muchos casos usar un patrón de reintento en el código de la aplicación con un límite de unos 10 intentos. Sin embargo, si la transacción es dependiente del contenido recuperado de la base de datos en alguna transacción SQL previa de la misma transacción de usuario, y algunas transacciones simultáneas han cambiado ese contenido en la base de datos y para la transacción actual por lo tanto no se actualiza el contenido, entonces esta operación no debe reintentarse. Pero el control debe ser devuelto al usuario para el posible reinicio de la transacción de usuario en su totalidad. Cubriremos esto en nuestro "RVV Paper".

En el caso de que la conexión se pierda debido a problemas de red, se debe abrir una nueva conexión con la base de datos para reintentar la transacción SQL.

Lecturas Recomendadas, Enlaces y Referencias

Berenson, H. et al, “A Critique of ANSI SQL Isolation Levels”, Technical Report MSR-TR-95-51, Microsoft Research, 1995 (available freely on multiple websites)

Melton, J., Simon, A. R., ”SQL:1999: Understanding Relational Language components”, Morgan Kaufmann, 2002

Structured Query Language (SQL) Version 2 – The Open Group
http://www.opengroup.org/onlinepubs/9695959099/toc.pdf

* * *

The OVA file of our virtual Database Laboratory “DebianDB” can be downloaded from
http://www.dbtechnet.org/download/DebianDBVM05.zip (3.9 GB)

and “Quick Start Guide” for accessing the DBMS products installed in the Lab can be downloaded from
http://www.dbtechnet.org/download/QuickStartGuideToDebianDB.pdf

Scripts for experiments of Appendix1 applied to DB2, Oracle, MySQL, and PostgreSQL at
http://www.dbtechnet.org/VET/EN/SQL_Transactions_Appendix1.zip

For more information on concurrency control technologies of DB2, Oracle and SQL Server see our “Concurrency Paper” at
http://www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf

For more information on SQL transactions as part of a user transaction, and applying of row version verification (RVV aka “optimistic locking”) in various data access technologies see our “RVV Paper” at
http://www.dbtechnet.org/papers/RVV_Paper.pdf

Apéndice 1 Experimentando con las transacciones en SQL Server

Bienvenido al viaje del misterio del mundo de las transacciones SQL usando su SGBD favorito para comprobar y experimentar por sí mismo lo que ha aprendido en este tutorial. Los distintos SGBD se comportan de forma diferente en el uso de las transacciones, lo que puede sorprenderle, y a sus clientes, si no es consciente de estas diferencias al desarrollar sus aplicaciones. Si tiene tiempo de abordar los experimentos en diferentes SGBD podrá constatar estas diferencias.

Los scripts de los distintos SGBD pueden encontrarse en
http://www.dbtechnet.org/VET/EN/SQL_Transactions_Appendix1.zip

En este apéndice presentamos los experimentos usando SQL Server Express 2012. Debido a que sólo existe versión de este SGBD para Windows, no está disponible en DebianDB. Se presentan además la mayor parte de los resultados para que pueda compararlo con sus experimentos. Si quiere comprobar estos resultados puede descargar SQL Server Express 2012 de forma gratuita del portal del Microsoft para plataformas Windows (Windows 7 o posterior).

En estos experimentos se usa SQL Server Management Studio (SSMS). En primer lugar crearemos la base de datos “TestDB” como se indica a continuación usando los parámetros de configuración por defecto, y mediante el comando USE nos conectaremos e iniciaremos la sesión SQL.

```
CREATE DATABASE TestDB;
USE TestDB;
```

Parte 1 Experimentando con las transacciones individuales

SQL Server funciona de forma automática en modo AUTOCOMMIT. Usando transacciones explícitas podemos construir transacciones de varios comandos. Sin embargo, se puede configurar el servidor para usar transacciones implícitas. Para configurar una sesión con transacciones implícitas usamos el siguiente comando:

```
SET IMPLICIT_TRANSACTIONS ON;
```

que se usará hasta el final de la transacción salvo que se cambie con el siguiente comando:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Comencemos a experimentar. Mostraremos al comienzo los resultados más importantes.

```
--  Autocommit mode
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si SMALLINT);
Command(s) completed successfully.

INSERT INTO T (id, s) VALUES (1, 'first');
(1 row(s) affected)

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

ROLLBACK; -- ¿Qué sucede?
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)
```

```
BEGIN TRANSACTION;  -- comenzamos una transacción explícita
INSERT INTO T (id, s) VALUES (2, 'second');
SELECT * FROM T;
id          s                            si
----------- ---------------------------- ------
1           first                        NULL
2           second                       NULL
(2 row(s) affected)

ROLLBACK;
SELECT * FROM T;
id          s                            si
----------- ---------------------------- ------
1           first                        NULL
(1 row(s) affected)
```

El comando de ROLLBACK ha funcionado, pero volvemos al modo AUTOCOMMIT

```
-- Ejercicio 1.2
INSERT INTO T (id, s) VALUES (3, 'third');
(1 row(s) affected)

ROLLBACK;
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                            si
----------- ---------------------------- ------
1           first                        NULL
3           third                        NULL
(2 row(s) affected)

COMMIT;
Msg 3902, Level 16, State 1, Line 2
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.

-- Ejercicio 1.3
BEGIN TRANSACTION;
DELETE FROM T WHERE id > 1;
(1 row(s) affected)

COMMIT;
SELECT * FROM T;
id          s                            si
----------- ---------------------------- ------
1           first                        NULL

(1 row(s) affected)


-- Ejercicio 1.4
-- DDL (Data Definition Language) incluye los comandos CREATE, ALTER y DROP.
-- ¡Usemos commandos DDL!
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (2, '¿se conforma este dato?');
CREATE TABLE T2 (id INT);  -- Probado un commando DDL
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
GO  -- GO marca el final de una serie de commandos que son mandadas al servidor
(1 row(s) affected)

(1 row(s) affected)
id
-----------
1
```

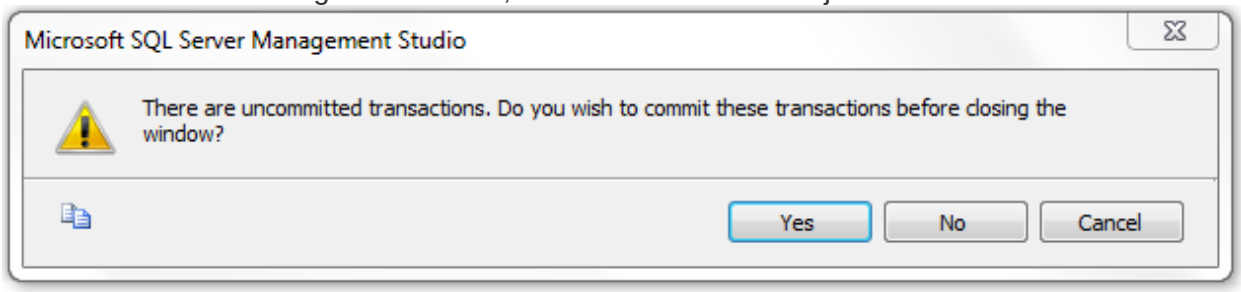
```
(1 row(s) affected)

SELECT * FROM T;  -- ¿Qué ha pasado en T?
id          s                             si
----------- ----------------------------- ------
1           first                         NULL
(1 row(s) affected)

SELECT * FROM T2; -- ¿Qué ha pasado en T2?
Msg 208, Level 16, State 1, Line 2
Invalid object name 'T2'.

-- Ejercicio 1.5
DELETE FROM T WHERE id > 1;
COMMIT;
----------------------------------------------------------------------
-- Probando si un error lleva a un rollback automático
--   @@ERROR es el SQLCode de Transact-SQL, y
--   @@ROWCOUNT es el contador de filas afectadas
----------------------------------------------------------------------
INSERT INTO T (id, s) VALUES (2, 'Comienza la prueba');
(1 row(s) affected)

SELECT 1/0 AS dummy;   -- la division por cero debería fallar
dummy
-----------
Msg 8134, Level 16, State 1, Line 1
Divide by zero error encountered.

SELECT @@ERROR AS 'sqlcode'
sqlcode
-----------
8134
(1 row(s) affected)

UPDATE T SET s = 'foo' WHERE id = 9999;  -- actualizar una tupla inexistente
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Updated'
Updated
-----------
0
(1 row(s) affected)

DELETE FROM T WHERE id = 7777; -- borrar una tupla inexistente
(0 row(s) affected)
SELECT @@ROWCOUNT AS 'Deleted'
Deleted
-----------
0
(1 row(s) affected)

COMMIT;
SELECT * FROM T;
id          s                             si
----------- ----------------------------- ------
1           first                         NULL
2           Comienza la prueba            NULL
(2 row(s) affected)

INSERT INTO T (id, s) VALUES (2, 'Hola, soy un duplicado')
INSERT INTO T (id, s) VALUES (3, '¿Que pasa si insert un valor demasiado grande?')
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow para 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, '¿Sigue active la transaccion??');
SELECT * FROM T;
COMMIT;
GO

Msg 2627, Level 14, State 1, Line 1
```

```
Violation of PRIMARY KEY constraint 'PK__T__3213E83FD0A494FC'. Cannot insert duplicate key in object
'dbo.T'. The duplicate key value is (2).
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 2
String or binary data would be truncated.
The statement has been terminated.
Msg 220, Level 16, State 1, Line 3
Arithmetic overflow error for data type smallint, value = 32769.
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 4
String or binary data would be truncated.
The statement has been terminated.
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           Comienza la prueba             NULL
(2 row(s) affected)


BEGIN TRANSACTION;
SELECT * FROM T;
DELETE FROM T WHERE id > 1;
COMMIT;

-- Ejercicio 1.5b
-- Ejercicio especial para SQL Server
SET XACT_ABORT ON;  -- En este modo un error genera un rollback de forma automática
SET IMPLICIT_TRANSACTIONS ON;
SELECT 1/0 AS dummy;    -- division por cero
INSERT INTO T (id, s) VALUES (6, 'insertar despues de arithm. error');
COMMIT;
SELECT @@TRANCOUNT AS '¿tenemos una transaccion?'
GO
dummy
-----------
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.

SET XACT_ABORT OFF; -- Este modo no genera el rollback automático
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           Comienza la prueba             NULL
(2 row(s) affected)
-- ¿Qué le pasa a la transacción?

--=================================================================
-- A1.2   Experimentando con la lógica de transacciones
-------------------------------------------------------------------
-- Ejercicio 1.6: COMMIT y ROLLBACK
-------------------------------------------------------------------
SET NOCOUNT ON;  -- saltando las el mensaje de las “n filas aceptadas”
DROP TABLE Accounts;
SET IMPLICIT_TRANSACTIONS ON;
--
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL
        CONSTRAINT unloanable_account CHECK (balance >= 0)
);
COMMIT;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
```

```
COMMIT;

-- intentemos una transaferencia
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
ROLLBACK;

-- Probemos que la restricción CHECK realmente funciona:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 2
The UPDATE statement conflicted with the CHECK constraint "unloanable_account". The conflict
occurred in database "TestDB", table "dbo.Accounts", column 'balance'.
The statement has been terminated.
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
acctID      balance
----------- -----------
101         1000
202         4000
ROLLBACK;

-- Lógica de transacción
-- Usando la estructura IF de Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 4
The UPDATE statement conflicted with the CHECK constraint "unloanable_account". The conflict
occurred in database "TestDB", table "dbo.Accounts", column 'balance'.
The statement has been terminated.

IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
COMMIT;

-- ¿Qué sucede con la cuenta inexistente?
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts ;
acctID      balance
----------- -----------
101         500
202         2000
ROLLBACK;


-- Arreglemos el caso usando la estructura IF de Transact-SQL
SELECT * FROM Accounts;
```

```
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 500 WHERE acctID = 707;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

-----------------------------------------------------------------------
-- Ejercicio 1.7  Probando la recuperación de la base de datos
-----------------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (9, '¿Qué sucede si ..');
SELECT * FROM T;
id          s                             si
----------- ----------------------------- ------
1           first                         NULL
9           ¿Qué sucede si  ..            NULL
```

Si nos salimos de Management Studio, obtenemos este mensaje



Para los propósitos de este experimento seleccionamos ”No”.

Reiniciando Management Studio y conectándose a TestDB podemos estudiar lo que ha sucedido con nuestra última transacción sin confirmar listando el contenido de la tabla T:

```
SET NOCOUNT ON;
SELECT * FROM T;
id          s                             si
----------- ----------------------------- ------
1           first                         NULL
```

Parte 2 Experimentando con transacciones concurrentes

Para los experimentos de concurrencia abriremos dos ventanas de consulta en paralelo siendo las mismas las sesiones A y B que accedan a la base de datos TestBDB. Indicamos que se muestren los resultados en modo texto:

Query -> Results To -> Results to Text

y usamos en ambas sesiones transacciones implícitas

```
SET IMPLICIT_TRANSACTIONS ON;
```

Para una mejor visualización podemos configurar Management Studio para que muestre las ventanas verticalmente (presionando con el botón secundario del ratón en el título de la ventana y seleccionando la opción "*New Vertical Tab Group*" (ver figura 1-1)

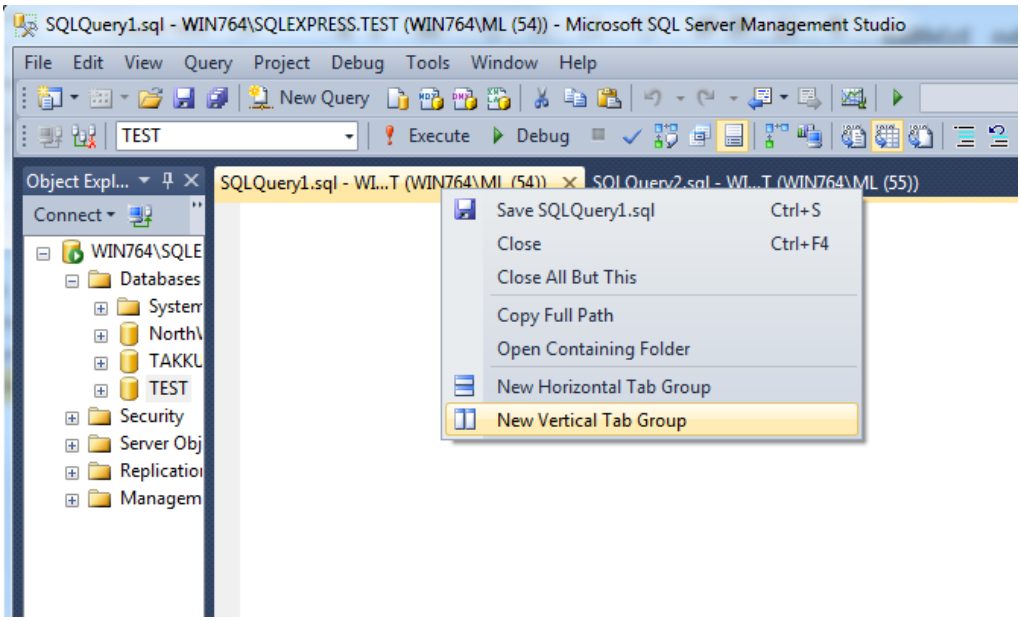


Figura 1-1. Mostrar ventanas verticalmente una junto a la otra

```
-- Ejercicio 2.1

-- 0. Para comenzar con datos frescos
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
-- Experimento 3  "Simulacion de actualizacion perdida"
--
```

El problema de la actualización perdida aparece si algunas filas insertadas o actualizadas por una transacción son a su vez actualizadas o borradas por alguna transacción concurrente antes de que la primera transacción finalice. Esto sería posible en sistemas basados en ficheros como NoSQL, pero los SGBD relacionales modernos previenen este caso. Sin embargo, después de que se confirme la primera transacción, cualquier transacción en competencia puede sobrescribir las tupla de la transacción confirmada. Simularemos este escenario usando el nivel de aislamiento LECTURA CONFIRMADA que no mantiene los S-locks. Primero las aplicaciones cliente leen el balance liberando los S-locks:

```
-- 1. cliente A inicia
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         1000

-- 2. cliente B inicia
SET NOCOUNT ON;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
```

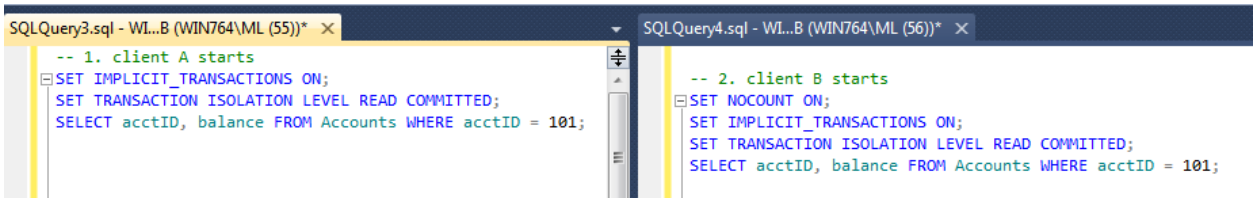
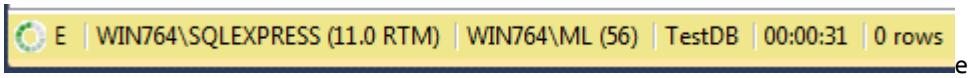


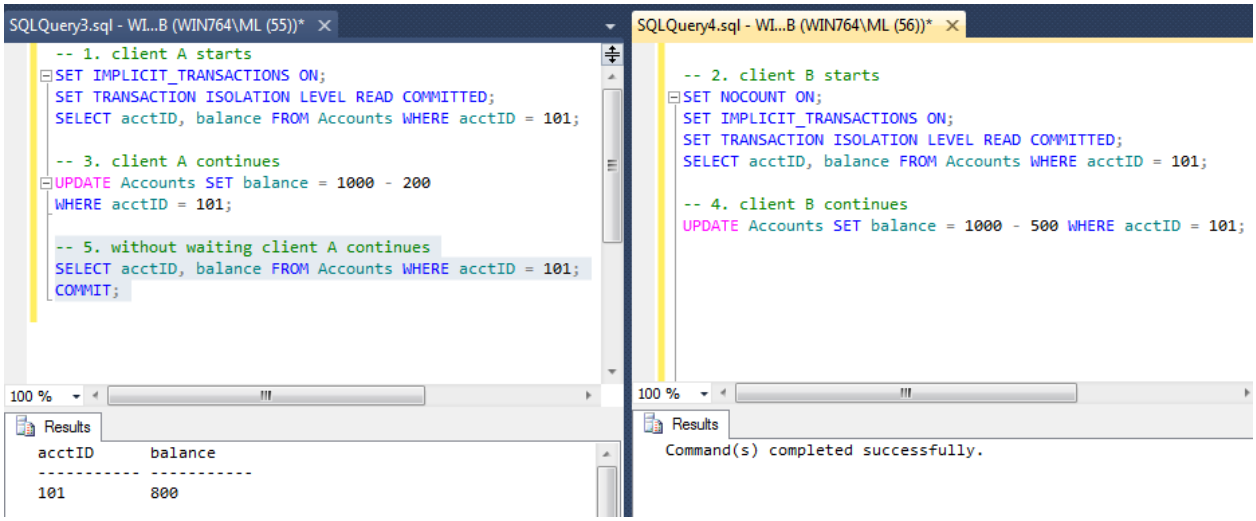
Figura 1-2. Sesiones en competencia en sus ventanas

```
-- 3. cliente A continua
UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;

-- 4. cliente B continua
UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;
```

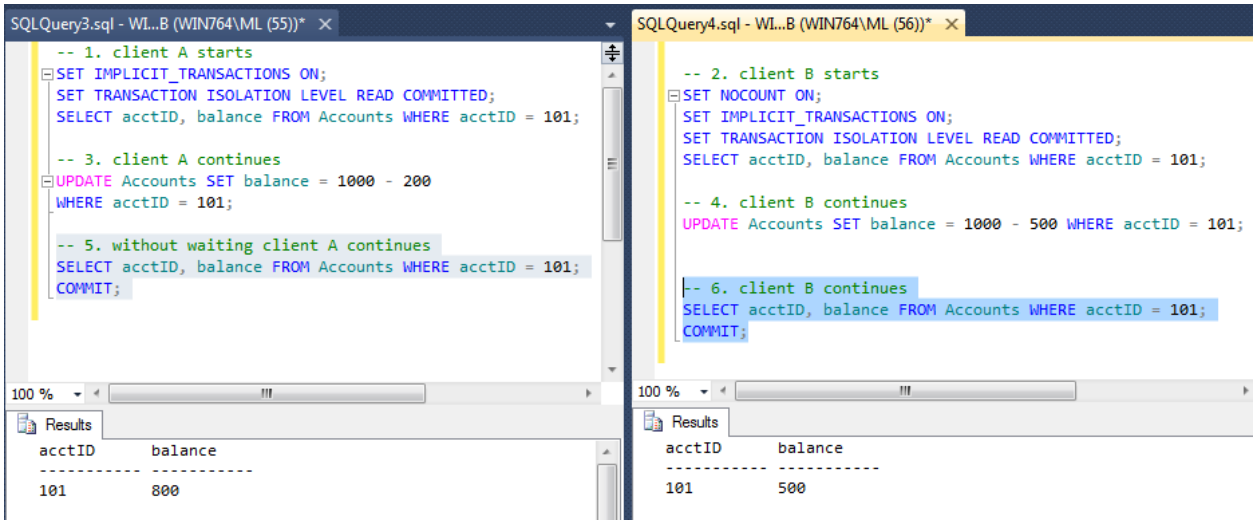


esperando por A

```
-- 5. cliente A continua
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;
```



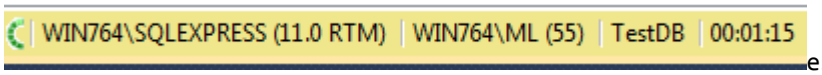

```
-- 6. cliente B continua
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;
```



¡Así que el resultado final es incorrecto!

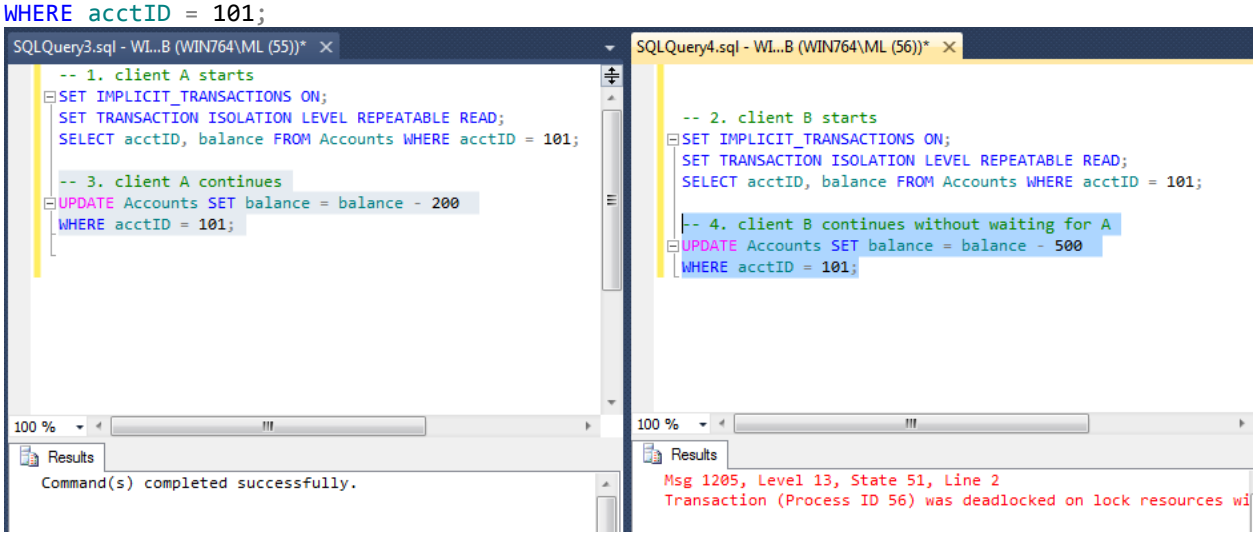
Nota: En este experimento no tenemos un caso real de actualización perdida, pero después de la confirmación de A, B puede proceder con la sobrescritura de las actualizaciones de A. Llamamos a este problema sobrescritura ciega (**Blind Overwriting)**" o escritura sucia (**Dirty Write)**. Esto se puede resolver con sentencias de actualización que usen ***sensitive updates*** como "SET balance = balance – 500 "

```
-----------------------------------------------------------------
-- Ejercicio 2.2  "Lost Update Problem" resuelto mediante bloqueos,
--              (competencia sobre un único recurso)
-----------------------------------------------------------------
--     usando SELECT .. UPDATE en ambos clients A y B
--     se intent sacar dinero de la misma cuenta.
--
-- 0. Restauramos los valores usando el cliente A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT WORK;

-- 1. cliente A inicia
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 2. cliente B inicia
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 3. cliente A continua
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 101;
```



esperando por B

```
-- 4. cliente B continua sin esperar a A
UPDATE Accounts SET balance = balance - 500
WHERE acctID = 101;
```

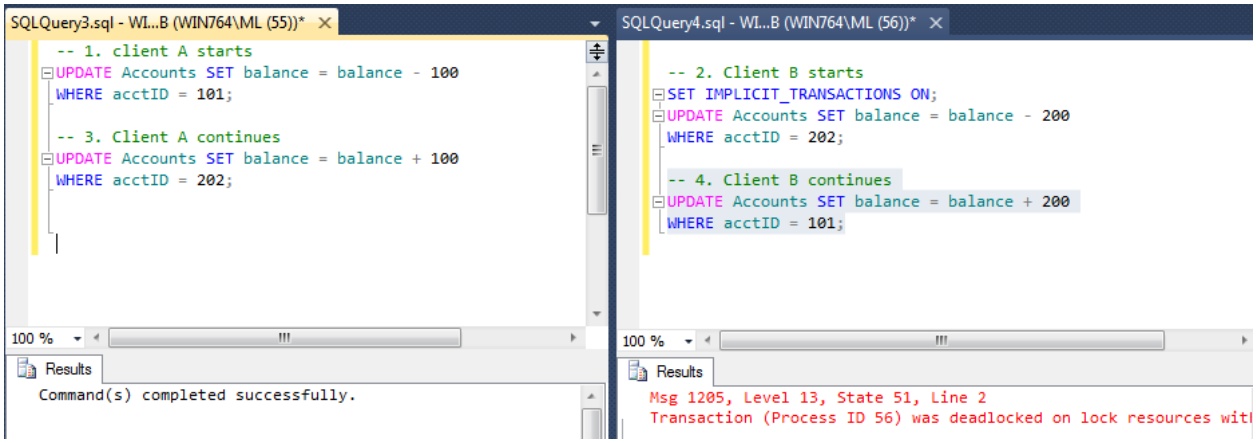



```
-- 5. El client superviviente se confirma
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         800

COMMIT;
```

```
-----------------------------------------------------------------
-- Ejercicio 2.3  Competencia con dos recursos en diferente orden
--                usando UPDATE-UPDATE
-----------------------------------------------------------------
--
-- Cliente A transfiere 100 euros de 101 a 202
-- Cliente B transfiere 200 euros de 202 a 101
--
-- 0. A reinicia los valores
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT WORK;

-- 1. cliente A inicia
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;

-- 2. Cliente B inicia
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 202;

-- 3. Cliente A continua
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;
COMMIT WORK;
```

Executing... | WIN764\SQLEXPRESS (11.0 RTM) | WIN764\ML (55) | TestDB | 00:00:58 esperando por B

```
-- 4. Cliente B continua
UPDATE Accounts SET balance = balance + 200
WHERE acctID = 101;
--
```




```
-- 5. Cliente A continua si puede ..
COMMIT;
```

****

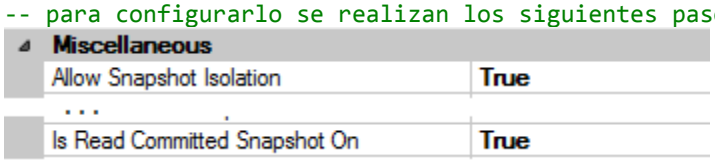
A continuación experimentaremos con anomalías de la concurrencia, conocidas como riesgos de fiabilidad de los datos por el estándar ISO SQL

- ¿Podemos identificarlos?
- ¿Cómo solucionamos estas anomalías?

```
-----------------------------------------------------------------
-- Ejercicio 2.4     ¿Lectura sucia?
-----------------------------------------------------------------

-- 0. Cliente A restaura los datos
```

```
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT WORK;

-- 1. Cliente A inicia
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;

-- 2. Cliente B inicia
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
COMMIT WORK;

-- 3. Cliente A continua
ROLLBACK;
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
COMMIT;

-- Ejercicio 2.5     ¿Lectura no reproducible?
-----------------------------------------------------------------------
-- Algunas tuplas leídas por la transacción actual pueden no aparecer en el resultado si se
-- repite la lectura antes del final de la transacción

-- 0. El cliente A restaura los datos
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT WORK;

-- 1. Cliente A inicia
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
--  Listamos las cuentas con balance  > 500 euros:
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
101         1000
202         2000

-- 2. Cliente B inicia
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
COMMIT WORK;

-- 3. Cliente A continua
-- ¿Vemos las mismas tuplas que en el primer paso?
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
202         2500
```

```
COMMIT;

-----------------------------------------------------------------
-- Ejercicio 2.6    ¿Fantasma de inserción?
-----------------------------------------------------------------
-- Tuplas insertadas por la transacción concurrente que podría ver la transacción concurrente
-- antes de terminar la transacción
--
-- 0. Cliente A limpia los datos
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT WORK;

-- 1. cliente A inicia
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- Cuentas de balance > 1000 euros:
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
101         1000
202         2000

-- 2. Cliente B inicia
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID,balance) VALUES (303,3000);
COMMIT;
-- 3. Cliente A continúa
-- Veamos el resultado:
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
303         3000

COMMIT;

Pregunta: ¿Cómo evitamos los fantasmas?


--=================================================================
-- Estudios de Instantáneas
-----------------------------------------------------------
-- La base de datos tiene que configurarse para soportar el nivel de aislamiento SNAPSHOT.
-- Creamos una nueva base de datos
CREATE DATABASE SnapsDB;
-- para configurarlo se realizan los siguientes pasos
```



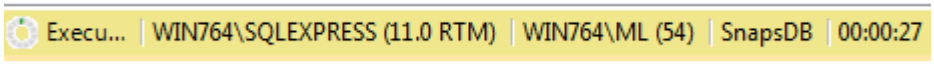

```
-- Tanto el ciente A como B pasan a usar esta nueba BD
USE SnapsDB;

-----------------------------------------------------------------
-- Ejercicio 2.7    Estudio con varios tipos de fantasmas
-----------------------------------------------------------------
USE SnapsDB;
-- 0. Establecemos la base
DROP TABLE T;
GO
SET IMPLICIT_TRANSACTIONS ON;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first',  1);
```

```
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;

-- 1. cliente A inicia
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT ;
SELECT * FROM T WHERE i = 1;
id          s                            i
----------- ---------------------------- ------
1           first                        1
3           third                        1
5           to be or not to be           1


USE SnapsDB;
-- 2. Cliente B inicia,
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO T (id, s, i) VALUES (6, 'Insert Phantom', 1);
UPDATE T SET s = 'Update Phantom', i = 1 WHERE id = 2;
DELETE FROM T WHERE id = 5;
SELECT * FROM T;
id          s                            i
----------- ---------------------------- ------
1           first                        1
2           Update Phantom               1
3           third                        1
4           forth                        2
6           Insert Phantom               1

-- 3. Cliente A continúa
-- Repitamos la consulta y hagamos algunas actualizaciones
SELECT * FROM T WHERE i = 1;
id          s                            i
----------- ---------------------------- ------
1           first                        1
3           third                        1
5           to be or not to be           1
INSERT INTO T (id, s, i) VALUES (7, 'inserted by A', 1);
UPDATE T SET s = 'update by A inside snapshot' WHERE id = 3;
UPDATE T SET s = 'update by A outside snapshot' WHERE id = 4;
UPDATE T SET s = 'update by A after B' WHERE id = 1;
SELECT * FROM T WHERE i = 1;
id          s                            i
----------- ---------------------------- ------
1           update by A after B          1
3           update by A inside snapshot  1
5           to be or not to be           1
7           inserted by A                1
```

```
-- 3.5 Cliente C
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T;
id          s                            i
----------- ---------------------------- ------
1           update by A after B          1
2           Update Phantom               1
3           update by A inside snapshot  1
4           update by A outside snapshot 2
6           Insert Phantom               1
7           inserted by A                1
(6 row(s) affected)
```

```
-- 4. Cliente B continúa
SELECT * FROM T;
```

```
id          s                             i
----------- ----------------------------- ------
1           first                         1
2           Update Phantom                1
3           third                         1
4           forth                         2
6           Insert Phantom                1

-- 5. Cliente A continúa
SELECT * FROM T WHERE i = 1;
id          s                             i
----------- ----------------------------- ------
1           update by A after B           1
3           update by A inside snapshot   1
5           to be or not to be            1
7           inserted by A                 1

UPDATE T SET s = 'update after delete?' WHERE id = 5;
```



esperando por B

```
-- 6. Cliente B continúa sin esperar a A
COMMIT;


-- 7. Cliente A continúa
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot isolation to
access table 'dbo.T' directly or indirectly in database 'SnapsDB' to update, delete, or insert the
row that has been modified or deleted by another transaction. Retry the transaction or change the
isolation level for the update/delete statement.

-- 8. Cliente B continúa
SELECT * FROM T;
id          s                             i
----------- ----------------------------- ------
1           first                         1
2           Update Phantom                1
3           third                         1
4           forth                         2
6           Insert Phantom                1

(5 row(s) affected)
```

Tarea: Explicar el experimento y los diferentes resultados de los pasos 3.5 y 4.

Apéndice 2 Transacciones en programación Java

Podemos experimentar con las transacciones SQL y los servicios del SGBD que se utilizarán sólo con SQL interactivo en las herramientas de cliente de SQL, llamados Editores SQL. Sin embargo, para las aplicaciones el acceso a datos se programa utilizando algunas APIs de acceso a datos. Para mostrar un ejemplo corto de la escritura de transacciones SQL utilizando alguna API, presentamos el ejemplo Bank Transfer implementado en un programa Java utilizando JDBC como API de acceso a datos. Suponemos que el lector de este apéndice ya está familiarizado con la programación Java y la API de JDBC, y la Figura 2-1 sirve sólo como un resumen visual que ayuda a memorizar los principales objetos, métodos, y su interacción.

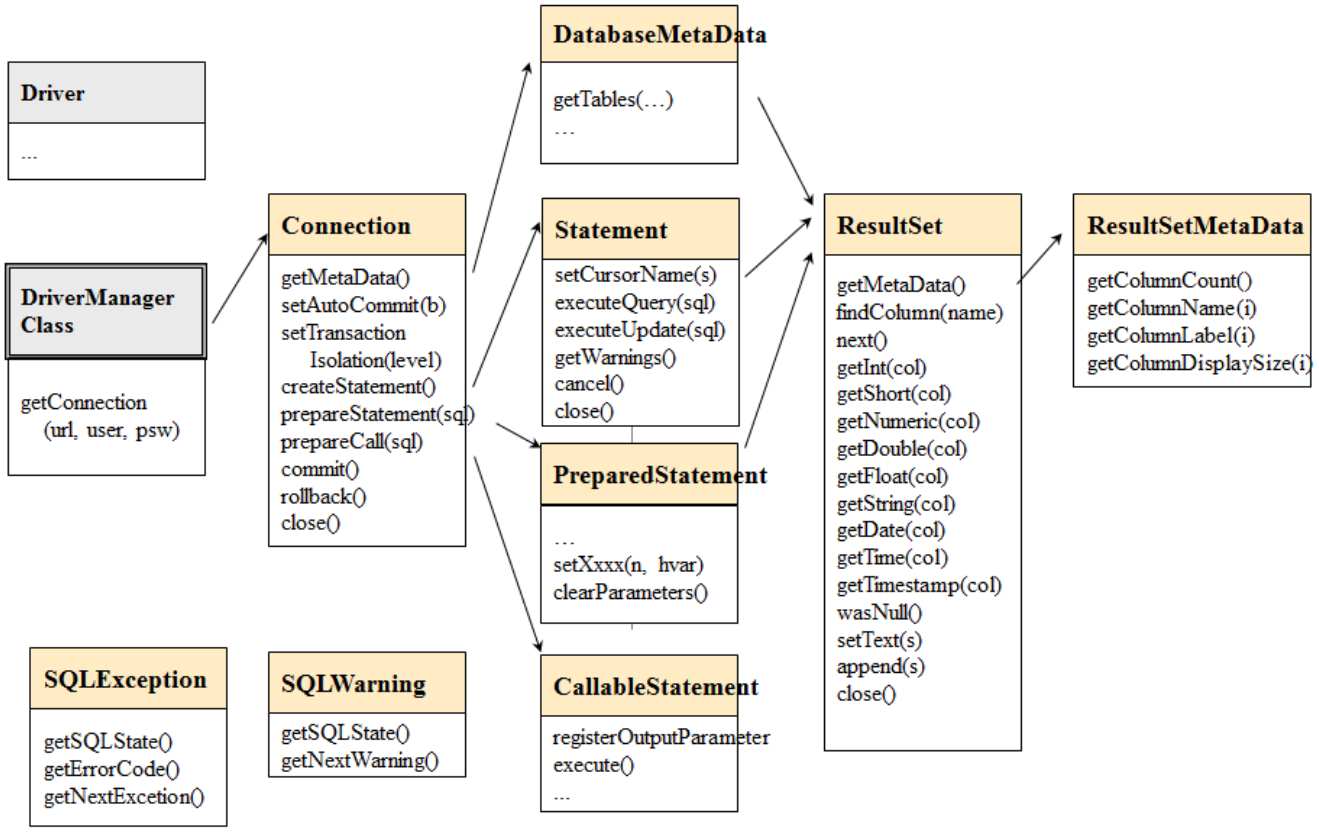


Figura 2-1 Vista simplificada de la API de JDBC

Ejemplo: BankTransfer

El programa BankTransfer transfiere una cantidad de 100 euros de una cuenta bancaria (fromAccount) a otra cuenta bancaria (toAccount). El nombre del controlador, la dirección URL de la base de datos, el nombre de usuario, la contraseña del usuario, fromAccount y toAccount se leen de los parámetros de línea de comandos del programa.

Para la prueba, el usuario debe iniciar dos ventanas de comandos en plataformas Windows (Terminales en plataformas Linux) y ejecutar el programa de forma simultánea en ambas ventanas para que el fromAcct de la primera sea la toAcct de la otra y viceversa, ver las secuencias de comandos a continuación.

Después de actualizar el fromAcct, el programa espera a que se pulse ENTER para continuar y que las ejecuciones de prueba se sincronicen. Así podemos probar el conflicto de concurrencia. El código del programa demuestra también el patrón de acceso a datos ReTryer.

Listado 2-1 BankTransfer usando JDBC

/* DBTechNet Concurrency Lab 15.5.2008 Martti Laiho

   BankTransfer.java

Salvar como BankTransfer.java y compilar como se indica

javac BankTransfer.java

Ver BankTransferScript.txt para los scripts aplicados a SQL Server, Oracle y DB2

```
import java.io.*;
import java.sql.*;
public class BankTransfer {
   public static String moreRetries = "N";

   public static void main (String args[]) throws Exception
   {
      System.out.println("BankTransfer version 2.2");

      if (args.length != 6) {
         System.out.println("Usage:" +
         "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
         System.exit(-1);
      }
      java.sql.Connection conn = null;
     // String moreRetries = "N";
      boolean sqlServer = false;
      int counter = 0;
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      String errMsg = "";
      int amount = 100;
      int fromAcct = Integer.parseInt(args[4]);
      int toAcct = Integer.parseInt(args[5]);

      // SQL Server's explicit transactions will require special treatment
      if (URL.substring(5,14).equals("sqlserver")) {
        sqlServer = true;
      }
            // register the JDBC driver and open connection
      try {
         Class.forName(args[0]);
         conn = java.sql.DriverManager.getConnection(URL,user,password);
      }
      catch (SQLException ex) {
           System.out.println("URL: " + URL);
           System.out.println("** Connection failure: " + ex.getMessage() +
                           "\n SQLSTATE: " + ex.getSQLState() +
                           " SQLcode: " + ex.getErrorCode());
         System.exit(-1);
      }
```

```
    do                                                                    {
    // Retry wrapper block of TransaferTransaction
        if (counter++ > 0) {
          System.out.println("retry #" + counter);
          if (sqlServer) {
            conn.close();
            System.out.println("Connection closed");
            conn = java.sql.DriverManager.getConnection(URL,user,password);
            conn.setAutoCommit(true);
          }
        }
        TransferTransaction (conn,
                        fromAcct, toAcct, amount,
                        sqlServer, errMsg //,moreRetries
                        );
        System.out.println("moreRetries="+moreRetries);
        if (moreRetries.equals("Y")) {
           long pause = (long) (Math.random () * 1000); // max 1 sec.
           System.out.println("Waiting for "+pause+ " mseconds"); // just for testing
           Thread.sleep(pause);
        }
    } while (moreRetries.equals("Y") && counter < 10);  // max 10 retries
    // end of the Retry wrapper block
    conn.close();
    System.out.println("\n End of Program. ");
}

static void TransferTransaction (Connection conn,
      int fromAcct, int toAcct, int amount,
      boolean sqlServer,
      String errMsg //, String moreRetries
      )
      throws Exception {
      String SQLState = "*****";
      try {
          conn.setAutoCommit(false);   // transaction begins
          conn.setTransactionIsolation(
              Connection.TRANSACTION_SERIALIZABLE);
          errMsg = "";
          moreRetries = "N";

          // a parameterized SQL command
          PreparedStatement pstmt1 = conn.prepareStatement(
            "UPDATE Accounts SET balance = balance + ? WHERE acctID = ?");
          // setting the parameter values
          pstmt1.setInt(1, -amount);  // how much money to withdraw
          pstmt1.setInt(2, fromAcct); // from which account
          int count1 = pstmt1.executeUpdate();
          if (count1 != 1) throw new Exception ("Account "+fromAcct + " is missing!");
```

```
        // --- interactive pause for concurrency testing -----
        // In the following we arrange the transaction to wait
        // until the user presses ENTER key so that another client
        // can proceed with a conflicting transaction.
        // This is just for concurrency testing, so don't apply this
        // user interaction in real applications!!!
          System.out.print("\nPress ENTER to continue ...");
          BufferedReader reader = new BufferedReader(
                  new InputStreamReader(System.in));
          String s = reader.readLine();
        // --- end of waiting ------------------------------------

    pstmt1.setInt(1, amount);  // how much money to add
    pstmt1.setInt(2, toAcct);  // to which account
    int count2 = pstmt1.executeUpdate();
    if (count2 != 1) throw new Exception  ("Account "+toAcct + " is missing!");
    System.out.print("committing ..");
    conn.commit();  // end of transaction
    pstmt1.close();
}
catch (SQLException ex) {
    try {
        errMsg = "\nSQLException:";
        while (ex != null) {
            SQLState = ex.getSQLState();
            // is it a concurrency conflict?
            if ((SQLState.equals("40001")   // Solid, DB2, SQL Server,...
              || SQLState.equals("61000")   // Oracle ORA-00060: deadlock detected
              || SQLState.equals("72000"))) // Oracle ORA-08177: can't serialize access
                moreRetries = "Y";
            errMsg = errMsg + "SQLState: "+ SQLState;
            errMsg = errMsg + ", Message:  "+ ex.getMessage();
            errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
            ex = ex.getNextException();
        }
        // SQL Server does not allow rollback after deadlock !
        if (sqlServer == false) {
           conn.rollback();  // explicit rollback needed for Oracle
                       // and the extra rollback does not harm DB2
        }
        // println for testing purposes
        System.out.println("** Database error: " + errMsg);
    }
    catch (Exception e) { // In case of possible problems in SQLException handling
        System.out.println("SQLException handling error: " + e);
        conn.rollback(); // Current transaction is rolled back
        ;  // This is reserved for potential exception handling
    }
} // SQLException
catch (Exception e) {
    System.out.println("Some java error: " + e);
    conn.rollback(); // Current transaction is rolled back also in this case
    ;  // This is reserved for potential other exception handling
} // other exceptions
```

```
  }
}
```

Los scripts del Listado 2-2 pueden usarse para probar el programa en un ordenador Windows en dos ventanas de comandos en paralelo. Estos scripts asumen que el SGBD usado es SQL Server Express, la base de datos se llama “TestDB”, y el driver JDBC está almacenado en el subdirectorio “jdbc-drivers” del directorio del programa.

Listado 2-2 Scripts para experimentar con BankTransfer en Windows

```
rem script for the first window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=101
set toAcct=202
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%

rem script for the second window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=202
set toAcct=101
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%
```

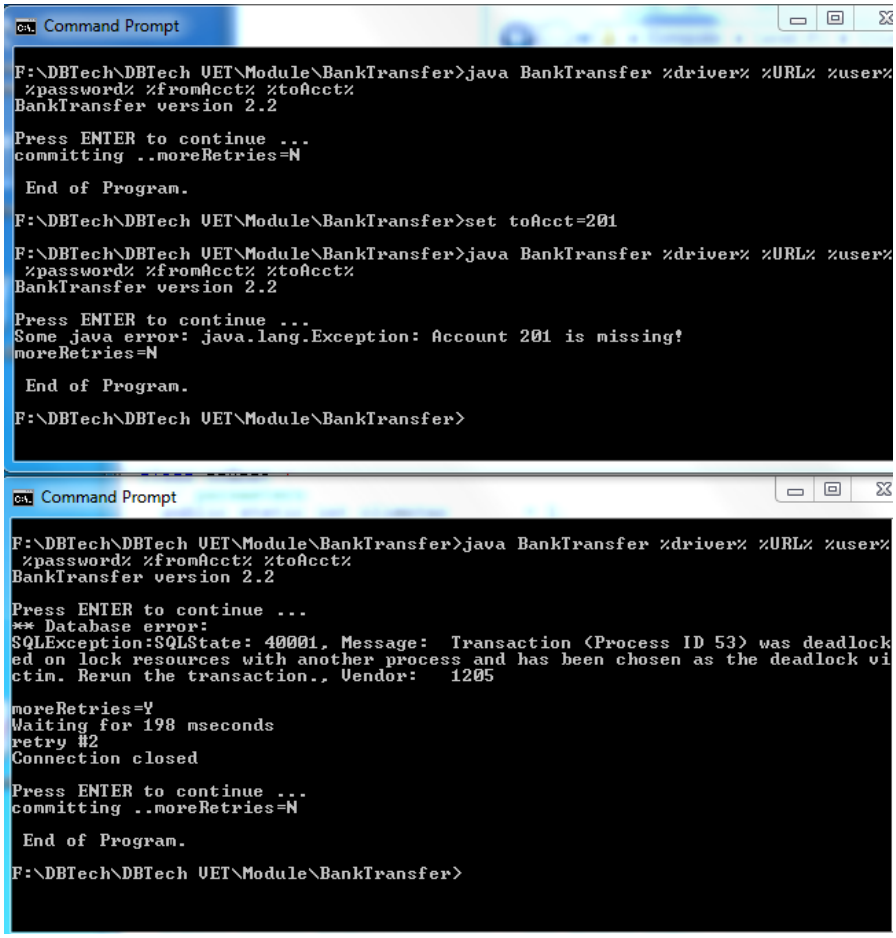


Figura 2-1 Ejemplos de pruebas de BankTransfer en Windows

Los scripts para otros SGBD y plataformas Linux pueden producirse modificando los mostrados.

Listado 2-3 Scripts para probar BankTransfer usando MySQL en Linux

```
Scripts for MySQL on Linux
# creating directory /opt/jdbc-drivers for JDBC drivers
cd /opt
mkdir jdbc-drivers
chmod +r+r+r jdbc-drivers
# copying the MySQL jdbc driver to /opt/jdb-drivers
cd /opt/jdbc-drivers
cp $HOME/mysql-connector-java-5.1.23-bin.jar
# allow read access to the driver to everyone
chmod +r+r+r mysql-connector-java-5.1.23-bin.jar

#******* MySQL/InnoDB ************************************************
# First window:
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password $fromAcct $toAcct

# Second window:
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password $fromAcct $toAcct
#*******************************************************************************
```

Apéndice 3 Transacciones en la Recuperación de Bases de Datos

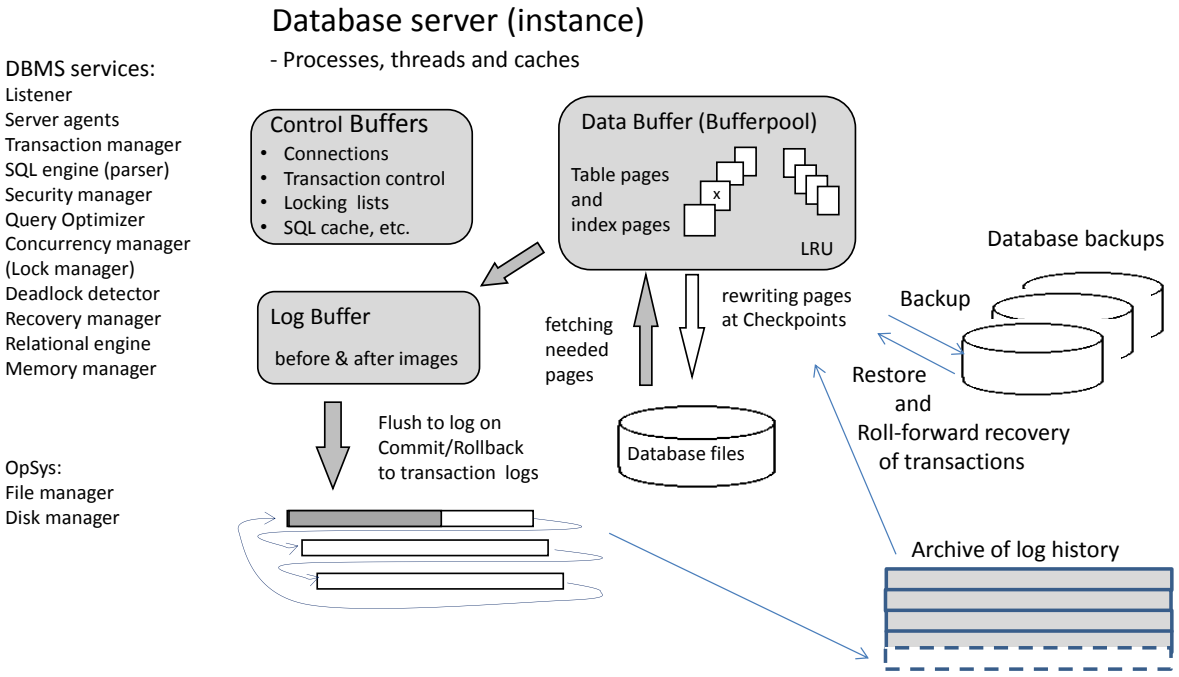


Figura 3-1. Vista general de un Servidor de Bases de Datos

La Figura 3-1 presenta la imagen general del procesamiento interno de un servidor típico de bases de datos. Las transparencias de "Basics of SQL Transactions", que están disponibles en su versión original en inglés en http://www.dbtechnet.org/VET/EN/BasicsOfSqlTransactions.pdf, presentan en más detalle las arquitecturas de los principales SGBDs, incluyendo la gestión de los ficheros de la base de datos y los logs de transacciones, la gestión del bufferpool (cache de datos) en memoria para mantener al mínimo las operaciones de E/S de disco para incrementar el rendimiento, la forma en la que se gestionan las transacciones, la fiabilidad de las operaciones de COMMIT, y la implementación de las operaciones de ROLLBACK.

A continuación se describe como un SGBD puede recuperar la base de datos a partir de las últimas transacciones confirmadas en caso de fallos eléctricos o averías en el servidor. En el caso de fallos de hardware la base de datos se recupera usando el backup de la base de datos y los logs de transacciones desde el último backup (ver http://www.dbtechnet.org/VET/FI/RdbmsRecovery.ppt).

Cuando una transacción SQL se inicia, el servidor de base de datos le dará un número de transacción único, y por cada acción en la transacción se escribirán registros en el archivo de logs de transacciones. Por cada fila procesada la entrada del registro contiene la identificación de la transacción y los contenidos originales de la fila como "imagen previa" y el contenido de la fila después de la operación como "imagen posterior". En el caso de los comandos INSERT la imagen previa está vacía. En el caso de los comandos DELETE la imagen posterior está vacía. También para las operaciones COMMIT y ROLLBACK, se escriben en el log los registros correspondientes, y como parte de esto, todos los registros de la transacción se escriben en el archivo de logs de transacciones en disco. El control de la operación de COMMIT se devolverá al cliente sólo después de que el registro de confirmación se haya escrito en el disco.

Cada cierto tiempo el servidor de base de datos hará una operación de CHECKPOINT en el que los clientes se pararán. Todos los registros de log de la caché se escribirán al fichero de log de transacciones y todas las páginas actualizadas (marcadas con un "bit sucio") en la caché de datos (bufferpool), se escribirán en su sitio en los ficheros de datos de la base de datos, y los "bits sucios" de dichas páginas se limpiarán en la caché de datos. En el log de transacciones, un grupo de

registros de punto de control se escribirán incluyendo una lista de ids de transacciones de las transacciones actualmente en ejecución. Los clientes continuarán su ejecución.

Nota: En un apagado gestionado del servidor no debería haber sesiones SQL activas ejecutándose en el servidor, por lo que la última operación sobre el log de transacciones será escribir un registro de punto de control indicando un **apagado limpio**.

La figura 3-2 presenta un historial de un log de transacciones antes de un fallo en la instancia de la base de datos o una parada completa del servidor, por ejemplo, debido a un fallo eléctrico. Todos los ficheros en el disco pueden leerse, pero los contenidos del bufferpool se han perdido. Esta situación se conoce como Soft Crash.

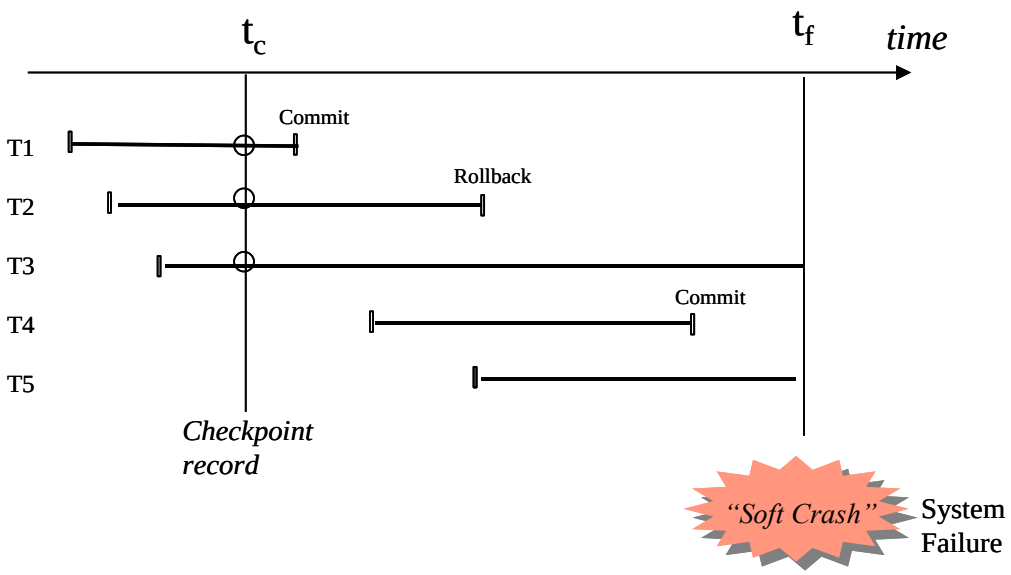


Figura 3-2. Un historial del log de transacciones antes de un *soft crash*

Cuando el servidor se reinicia primero encontrará el último punto de control en el log de transacciones. Si el registro de control está vacío indica que ha sido un apagado limpio y el servidor está listo para servir a los clientes. En otro caso, el servidor inicia una recuperación mediante operaciones de rollback: primero se copian a una lista de transacciones llamada UNDO todos los números de transacciones listados en el registro de punto de control para ser deshechas, y se vacía la lista REDO para los números de los identificadores de transacciones para ser ejecutados de nuevo en la base de datos. A continuación se comprueban los ids de transacciones desde el punto de control y todas las nuevas transacciones se añaden a la lista UNDO y se mueven los ids de las transacciones confirmadas de la lista UNDO a la lista REDO. Después de esto, el servidor procede avanzando hacia atrás en el log escribiendo en la base de datos las imágenes de antes de aquellas tuplas de las transacciones listadas en la lista UNDO, y luego avanza hacia adelante desde el punto de control escribiendo las imágenes de después de aquellas tuplas de las transacciones listadas en la lista REDO. Después de esto, la base de datos se habrá recuperado hasta el nivel de la última transacción confirmada antes del fallo del sistema y el servidor puede comenzar a servir a los clientes. Ver Figura 3-3.

Nota: Aparte del proceso simplificado de recuperación mostrado anteriormente, la mayor parte de los SGBD actuales usan el protocolo ARIES, en el que entre puntos de control donde no hay carga en la base de datos, una hebra perezosa sincroniza las páginas sucias de la caché de datos en los ficheros de datos. Las páginas sincronizadas marcadas con LSN de ARIES pueden omitirse del proceso de recuperación y eso hace que ésta sea más rápida.

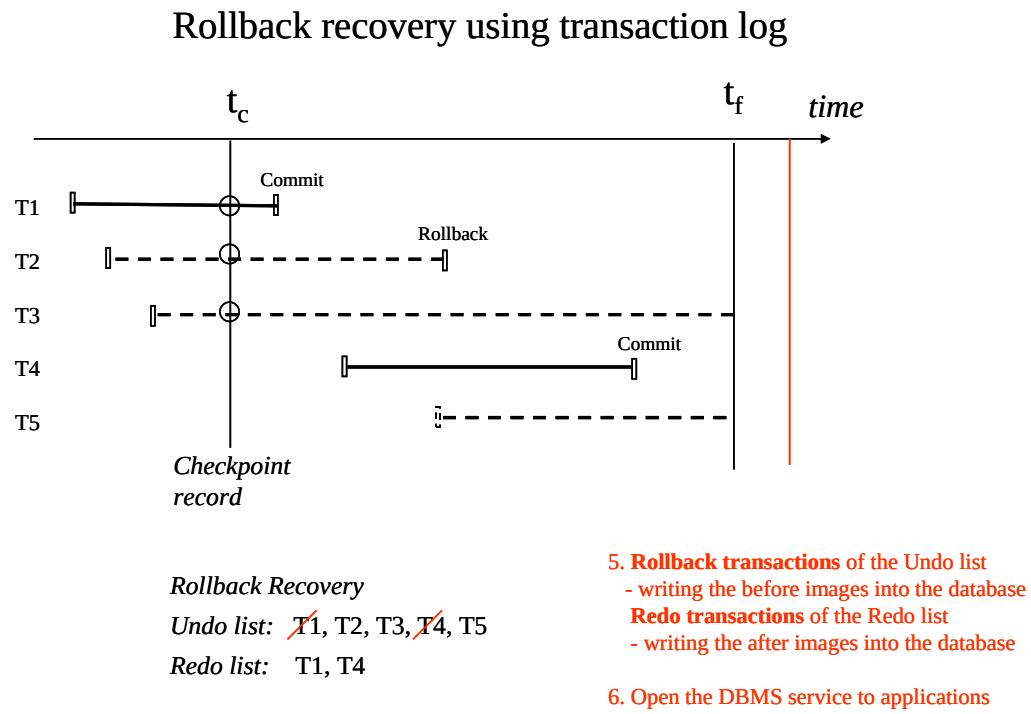


Figura 3-3. Recuperación de la base de datos