Gestión de Transacciones en Bases de Datos

Universidad de Sevilla

Departamento de Lenguajes y Sistemas Informáticos

Objetivos


Transacciones
Aislamiento
Sistema
Fallos
Leer
Concurrencia
Gránulo
Acid
Read
Mariadb
Transacción
Sgbd
Datos
Estado
Cuenta
Commit
Debe
Recuperación
Problemas
Escribir

Gestión de Transacciones en Bases de Datos

- Concepto de transacción y sus propiedades.
- Problemas provocados por la concurrencia y posibles soluciones.
- Niveles de aislamiento
- Tipos de fallos y políticas de recuperación.
- Transacciones en MariaDB

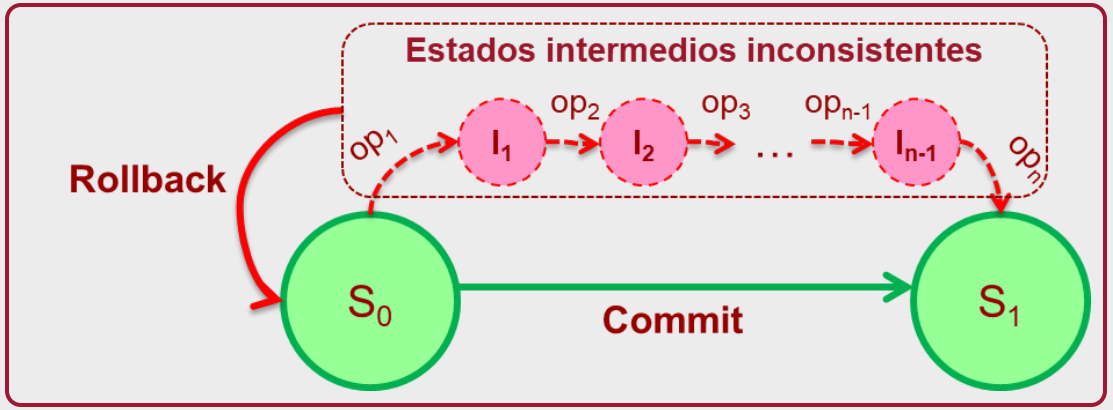
Concepto de Transacción en una BD

Secuencia de operaciones (op1..opN) que se comporta como una **unidad lógica** de trabajo.

Se deben **almacenar**

Commit: Confirmar cambios. Terminación normal, alcanzando nuevo estado consistente

Rollback: Terminación anormal, volviendo al estado inicial consistente



Propiedades ACID de una Transacción en BD

Las transacciones deben ser **ACID** para garantizar estados consistentes de la BD

Atomicity: las operaciones o son completadas y procesadas totalmente o son descartadas y no tienen efecto.

Consistency: las operaciones no pueden violar la integridad de los datos.

Isolation: cada operación debe ser independiente y ser llevada a cabo sin que afecte al resultado de otra transacción.

Durability: el resultado de una operación debe ser persistente en el tiempo.

Ejemplo de transacción ACID: Cajero Automático

Atomicity

> Al sacar dinero se debe descontar de la cuenta y dar el dinero al cliente. Las dos operaciones o ninguna.

Consistency

> No se puede sacar una cantidad negativa, o más del límite asociado a la tarjeta (saldo o crédito).

Isolation

> a) Si dos titulares sacan concurrentemente una suma conjunta que está por debajo del límite global, el saldo final es indiferente al orden.
> b) Si dos titulares sacan por encima del límite autorizado concurrentemente, solo una de las transacciones debe concluir.

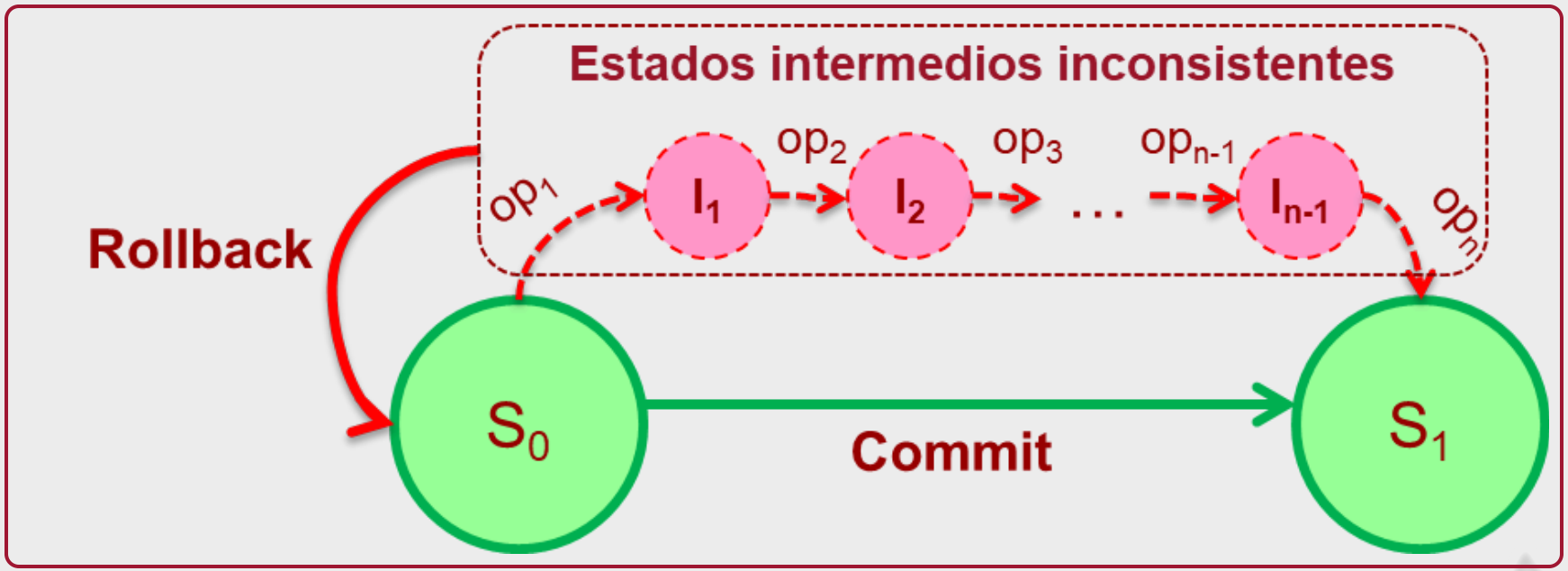
Durability

> Si la transacción termina satisfactoriamente, el movimiento debe quedar registrado en la cuenta del cliente y el cajero registra la operación.

ACID. Atomicidad

Atómica: si se confirma (**Commit**), se ejecutan todas las operaciones; si se cancela (**Rollback**), no se ejecuta ninguna.

No se puede ejecutar parcialmente, dejando la base de datos en un estado intermedio (Ii), que puede ser **inconsistente**.

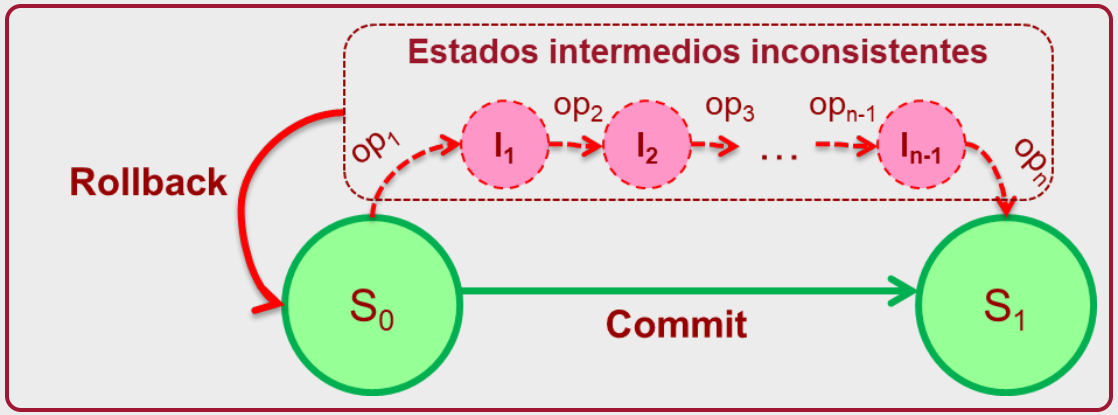


ACID. Consistencia

Consistente: a partir de un estado inicial consistente (S_0), deja la base de datos en otro estado final consistente ($_S1$), que satisface todas las **reglas de integridad**.

Los estados intermedios ($I_{1..n}$) NO son necesariamente consistentes.

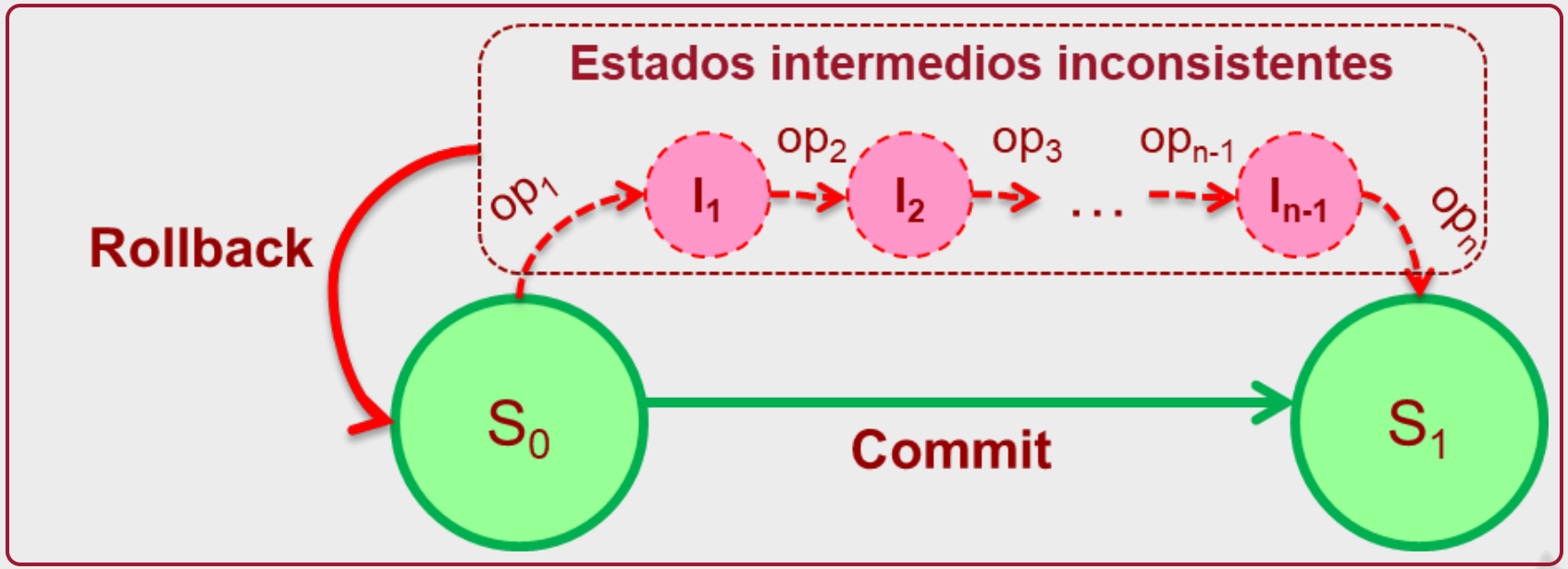
Pueden existir datos en memoria que no deberían estaren la BD (página sucia o **dirty-page**).



ACID. Aislada

Isolated (Aislada): se realiza como si fuera la única transacción ejecutándose en la base de datos.

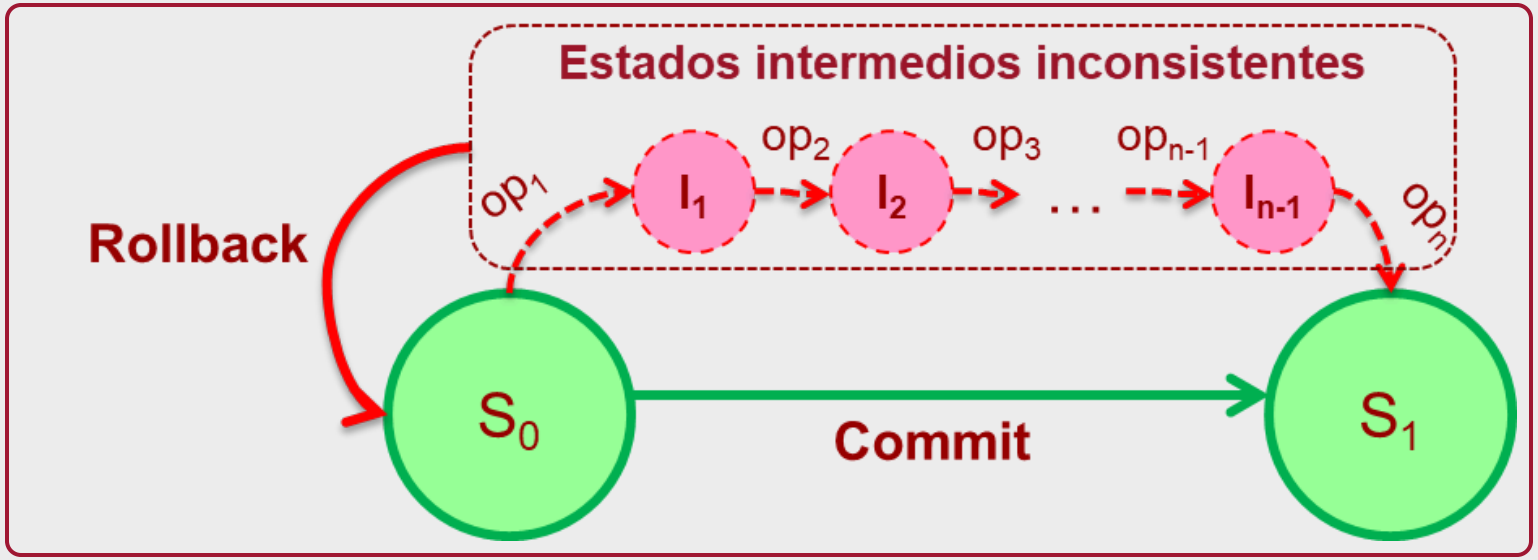
Los estados intermedios ($I_{1..n}$) no deberían ser visibles para otras transacciones que consultan el estado de la BD.



ACID. Durable o Persistente

Durable: si se confirma (**Commit**), los cambios en la BD se hacen permanentes (o Persistentes). Si se cancela (**Rollback**) el estado que persiste en el almacenamiento debe ser el inicial.

El **SGBD es responsable** de que los cambios sean permanentes, incluso aunque se produzca un fallo.

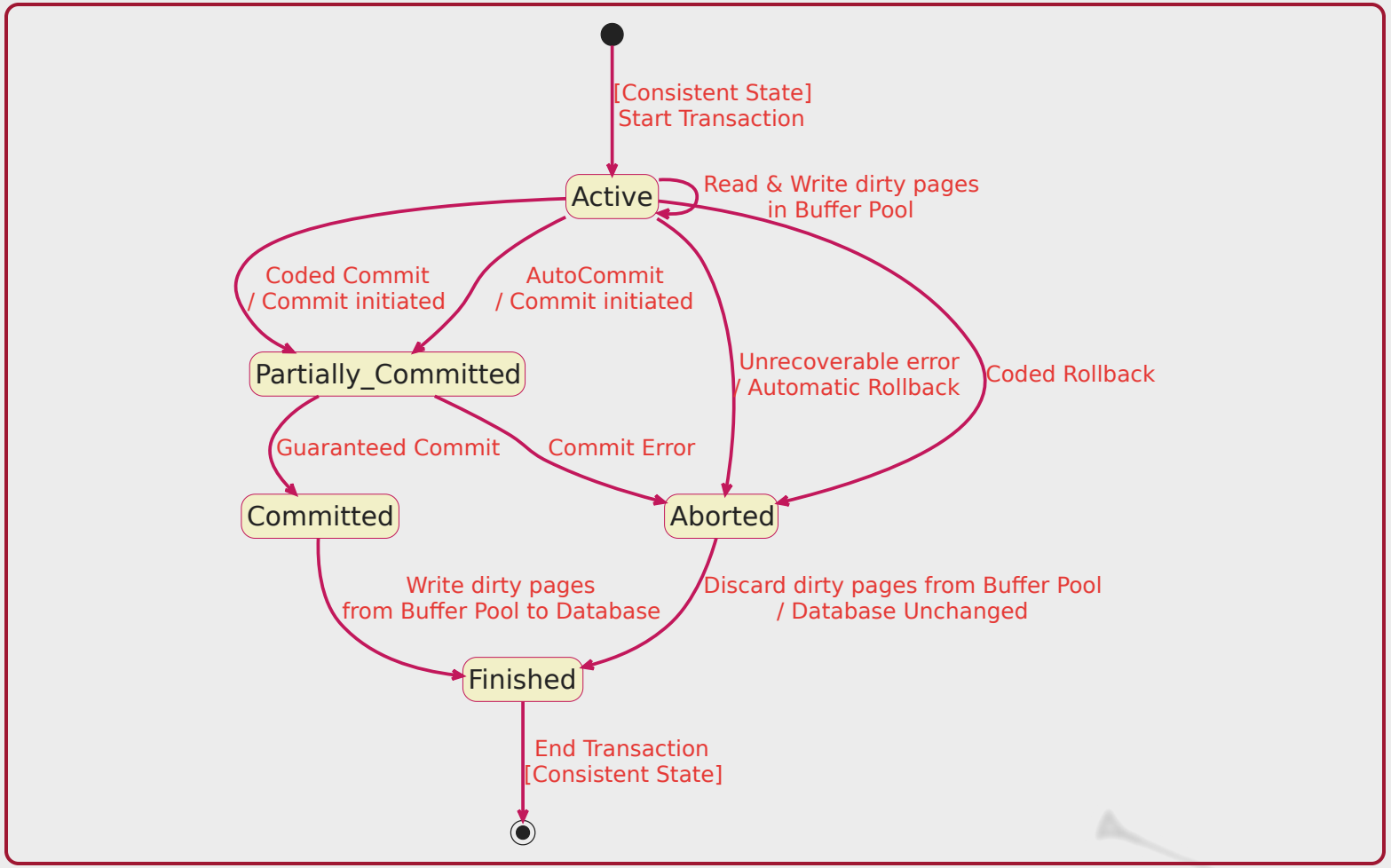


Estados de una transacción

Una transacción accede a elementos de datos (**Gránulos**) para leerlos (**Read**) o modificarlos (**Write**).

Un gránulo puede ser una **página**, una **tabla**, un **rango**, una **fila** o una **columna** de una fila. Cuanto menor es el gránulo, más fina es la granularidad de acceso.

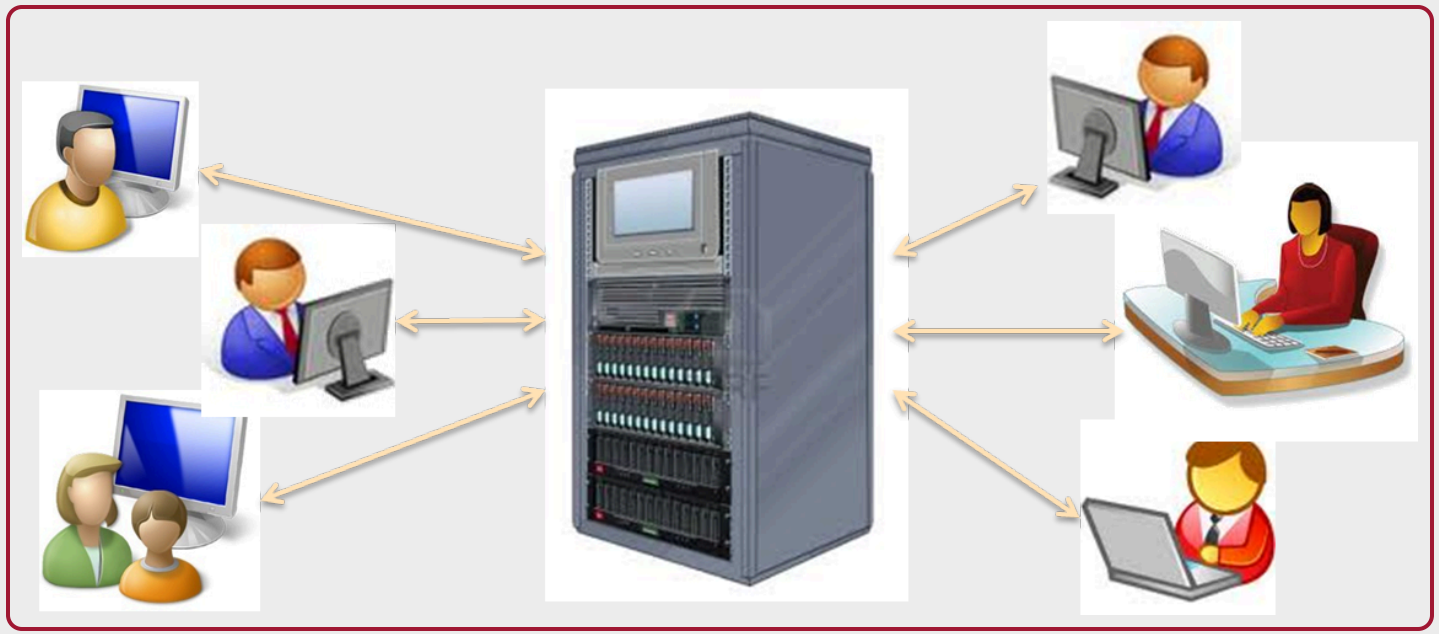
Un gránulo modificado en memoria (**buffer pool**) que no se ha escrito en la BD física es una página sucia (dirty page).



Problemas de Concurrencia

Múltiples usuarios simultáneos del sistema de información provocan la ejecución de transacciones concurrentes.

Si el aislamiento (Isolation Level) no está garantizado, pueden surgir inconsistencias.



Problemas de Concurrencia. Lost Update

Unas transacciones sobrescriben las actualizaciones de otras.

Ejemplo: la escritura de la T2 se pierde (**lost update**)

Transacción 1	**Transacción 2**
x_1 = LEER(X);	
	x_2 = LEER(X); x_2 = x_2 + 1; ESCRIBIR(X, x_2); COMMIT;
x_1 = x_1 + 1; ESCRIBIR(X, x_1); COMMIT;	

Problemas de Concurrencia. Dirty Read

Se lee un valor que está siendo modificado por otra transacción que no ha finalizado y que podría cancelarse o fallar.

Ejemplo: la lectura de T2 es "sucia" (**dirty read**).

Transacción 1	**Transacción 2**
x_1 = LEER(X); $x_1 = x_1 + 1$; ESCRIBIR(X, x_1);	
	x_2 = LEER(X); ESCRIBIR(X, x_2); COMMIT;
ROLLBACK;	

Problemas de Concurrencia. Non-repeatable read

Se obtienen lecturas diferentes durante la misma transacción.

Ejemplo: las lecturas de T1 son diferentes (**non-repeatable read**)

Transacción 1	Transacción 2
x_1 = LEER(X); IMPRIMIR(x_1);	
	x_2 = LEER(X); x_2 = x_2 + 1; ESCRIBIR(X, x_2); COMMIT;
x_1 = LEER(X); IMPRIMIR(x_1); COMMIT;	

Problemas de Concurrencia. Phantom read

Se obtienen diferentes tuplas {X} durante la misma transacción.

Ejemplo: una tupla de más $\{x_k\}$ en T1 (**phanton read**)

Transacción 1	Transacción 2
$\{x_i\}$= LEER({X}); IMPRIMIR($\{x_i\}$);	
	$\{x_j\}$= LEER({X}); $\{x_j\}$= $\{x_j\}$ ∪ $\{x_k\}$; ESCRIBIR({X}, $\{x_j\}$); COMMIT;
$\{x_i\}$= LEER({X}); IMPRIMIR($\{x_j\}$); COMMIT;	

Problemas de Concurrencia. Planes de Ejecución

¿Cómo evitar problemas de concurrencia?

Ejecutando las transacciones (Plan de ejecución (Schedule Plan)) secuencialmente, pero reducen mucho el rendimiento del SGBD.

$S_s = \{T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \cdots \rightarrow T_n\}$, es un Plan Serial o **Secuencial**

Ejecutando las transacciones concurrentemente pero con un Plan de Ejecución (Schedule Plan) Secuenciable (Serializable):

$S_c = \{T_1 \| T_2 \| T_3 \| \ldots \| T_n\}$, es un Plan **Concurrente**

Debe tener el mismo efecto final que las transacciones ejecutadas secuencialmente (Secuenciabilidad o Serializabilidad).

Problemas de Concurrencia. Estrategias/Algoritmos

El SGBD puede usar distintas estrategias o algoritmos para para garantizar la Serializabilidad de un Plan Concurrente S_c

Bloqueos (Protocolo de bloqueo en dos fases (Two Phase Lock Protocol))

- Ordenación de operaciones por marcas de tiempo (Timestamps) (Algoritmos de ordenación por Timestamp (Total o Parcial))
- Múltiples versiones del mismo dato temporal (Gránulos) (Algoritmos de ordenación por Timestamp Multiversión).

Two Phase Lock Protocol

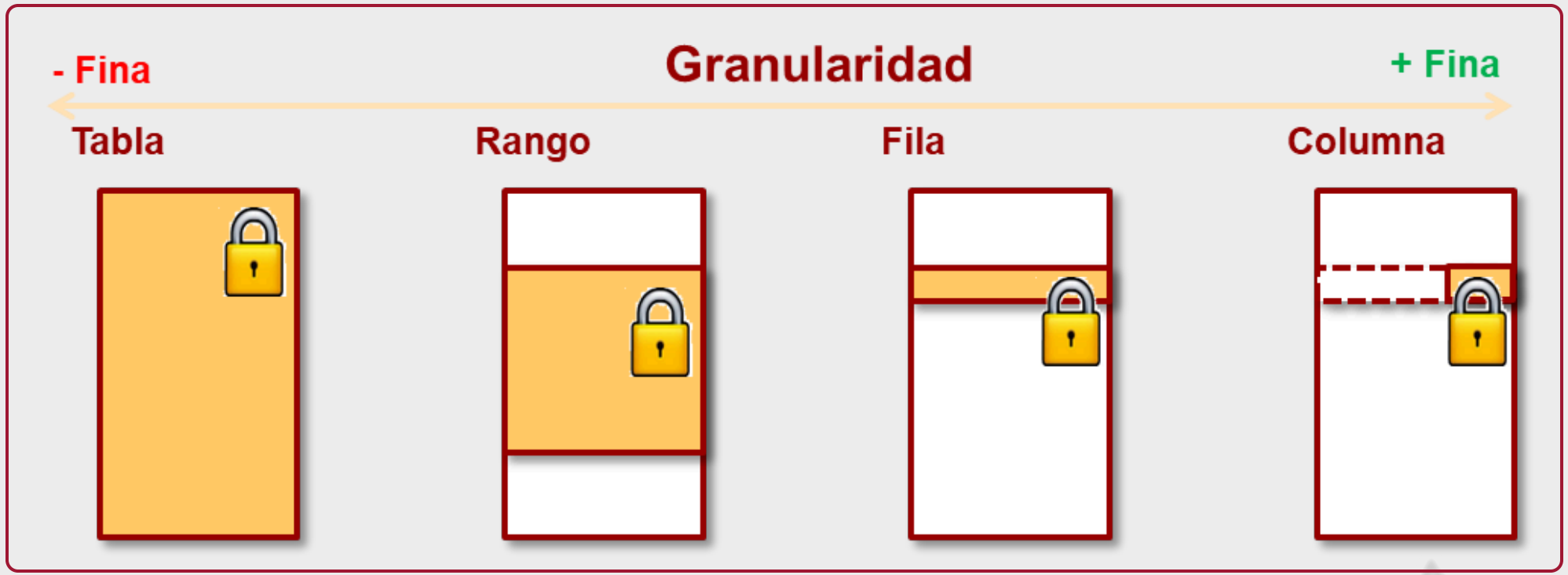
Una transacción, que puede acceder a un gránulo consigue bloquearlo en un modo compatible con otras transacciones. En caso contrario la transacción entra en cola de espera por ese gránulo (Wait Queue).

- Bloqueo en Lectura (Compartido (Read Lock)): un gránulo puede estar bloqueado para lectura por varias transacciones (bloqueos concurrentes compatibles en lectura). Los intentos de bloqueo exclusivo deben esperar (Wait Queue) si el gránulo está bloqueado en lectura.
- Bloqueo en Escritura (Exclusivo (Write Lock)): un gránulo puede estar bloqueado para escritura por una sola transacción; los demás intentos de bloqueo (lectura o escritura) deben esperar (Wait Queue).

Pueden generarse Abrazos Mortales (Deadlocks) cuando existen ciclos de espera entre transacciones. El sistema resuelve los Deadlocks abortando transacciones.

Two Phase Lock Protocol

Con granularidad más fina la eficiencia del sistema es mayor. Es decir, mejor throughput, al reducir la Wait Queue y los Deadlocks



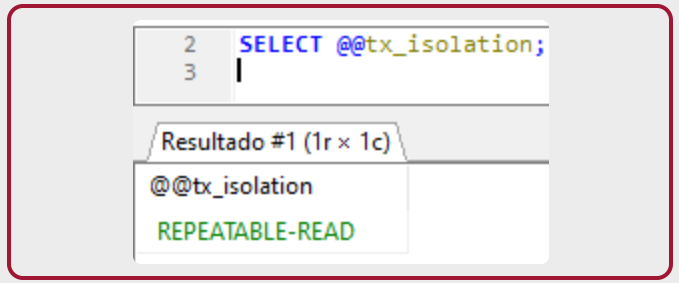
SQL92 Isolation Levels

READ_UNCOMMITTED: permite leer datos no confirmados de otras transacciones.

READ_COMMITED: sólo permite leer en una transacción los datos confirmados de otras, pero si se repite la lectura en la misma transacción entonces pueden desaparecer datos eliminados por otra transacción entre esas dos lecturas (Non-Repeatable-Read). Suele ser el nivel por defecto.

REPEATABLE_READ: respecto a READ_COMMITED, garantiza que están, al menos, las mismas filas que se acaban de leer en la misma transacción. Es la opción por defecto de MariaDB.

SERIALIZABLE: garantiza la secuenciabilidad de la transacción con todas las demás transacciones concurrentes; si no puede garantizarlo, la transacción se aborta.



Niveles de aislamiento

Isolation Level	Lost update	Dirty read	Non repeatable read	Phantom read
READ_UNCOMMITTED	No	No	No	No
READ_COMMITTED	No	Sí	No	No
REPEATABLE_READ	Sí	Sí	Sí	No
SERIALIZABLE	Sí	Sí	Sí	Sí

A mayor nivel de aislamiento:

- Mayor consistencia
- Peor rendimiento

Fallos en BD. Recuperación

Fallos del medio. Recuperación en frío (Cold Recovery)

Fallos software. Recuperación en caliente (Warm Recovery)

Fallos del medio. Cold Recovery

Fallo hardware del medio de almacenamiento por *avería, desastre, sabotaje*, etc.

El medio de almacenamiento está dañado. Hay pérdida permanente de datos. No es posible arrancar el sistema normalmente, mediante un rearranque en caliente (**Warm Recovery**).

La recuperación es responsabilidad de los administradores de la base de datos, normalmente mediante copias de seguridad (**Backups**).

Fallos Software. Warm Recovery

Fallos locales en transacciones

- Violación de reglas de integridad, tipos incorrectos, nombres de tablas o columnas erróneos, errores de sintaxis, división por cero, etc.
- Lo detecta el SGBD y lo debe resolver la aplicación mediante el tratamiento de excepciones oportuno.
- Error de programación, interbloqueo, fallo de una orden al SGBD, etc.
- Puede detectarlo tanto el SGBD como la aplicación, aunque el tratamiento (ROLLBACK) es responsabilidad del SGBD.
- El medio no está dañado, ni la BD ni el fichero de apoyo a la recuperación automática (Log).
- El Log siempre se escriben en disco antes que las transacciones, para poder recuperarlas después.

Fallos Software. Warm Recovery

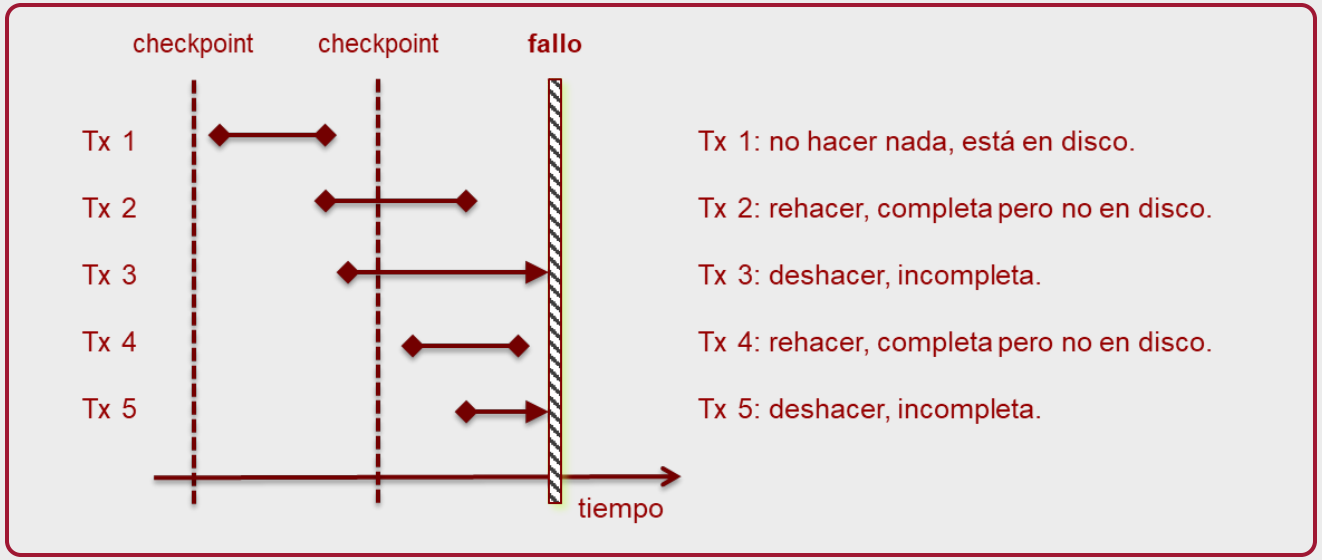
Fallos globales o del sistema

- Problemas que afectan a todo el sistema, como fallos de alimentación eléctrica, caídas del sistema operativo o el SGBD, etc.
- El SGBD es responsable de su recuperación mediante el uso del Log.

Algoritmos de Warm Recovery

Recuperan una transacción o el conjunto de transacciones en vuelo al caer el sistema (Fallos del Sistema)

- Cada cierto tiempo, el SGBD para el procesamiento de transacciones y vuelca todos los datos en memoria en el sistema de almacenamiento: Checkpoint.

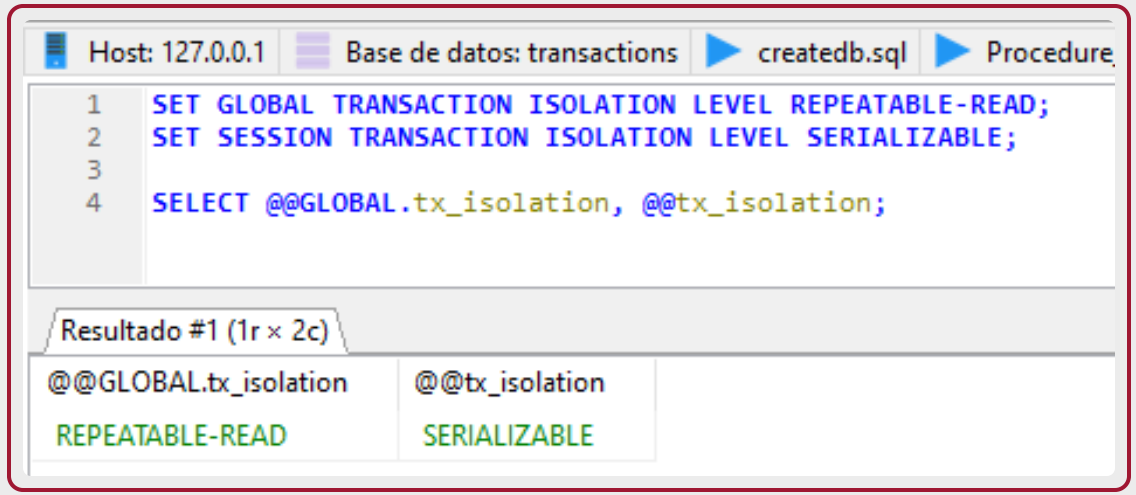


Transacciones en MariaDB. Isolation Levels

Define el nivel de aislamiento para:

- Una transacción
- Las transacciones dentro de una sesión
- De todas las sesiones

a partir del momento de la ejecución del COMANDO



Transacciones en MariaDB. Autocommit

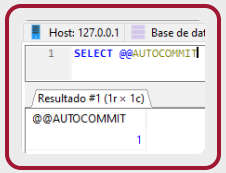
Por defecto `AUTOCOMMIT = 1;` (Activado)

> Cualquier modificación a la BD intenta escribirse inmediatamente en la BD física (Commit implícito), descargando páginas sucias del Buffer Pool.

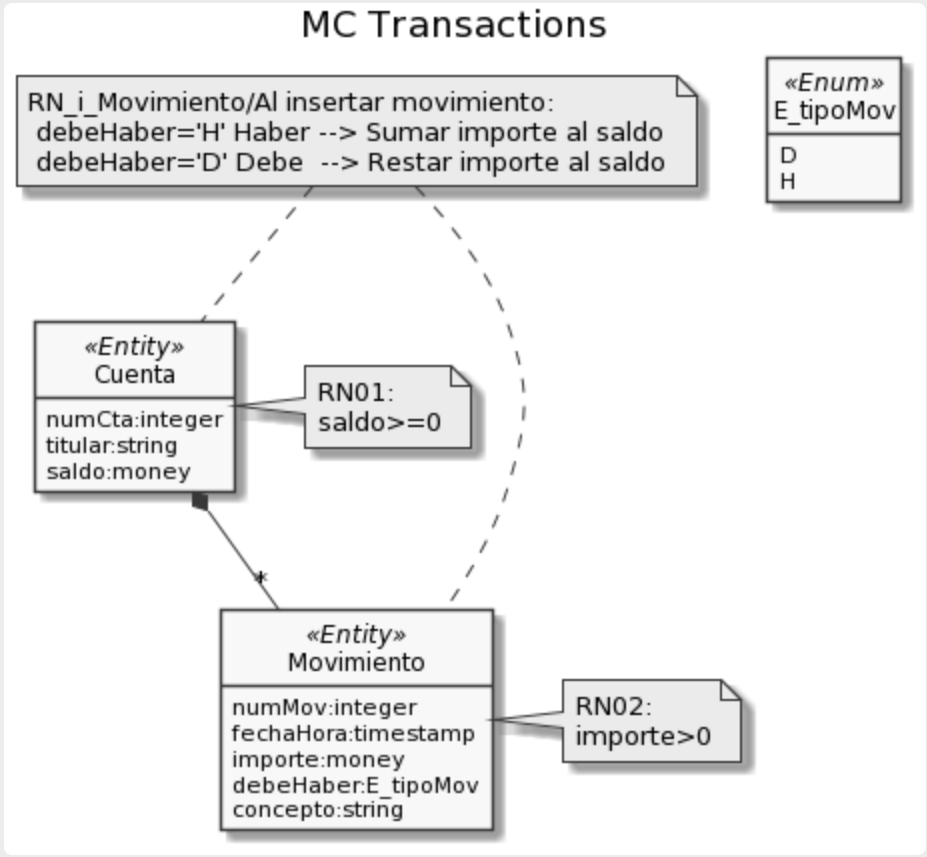
Se desactiva con `SET AUTOCOMMIT=0;`

> Hay que escribir transacciones (START TRANSACTION) y confirmarlas (COMMIT) o abortarlas explícitamente (ROLLBACK).

La verificación de reglas de negocio y escritura queda diferida hasta el final de la transacción.

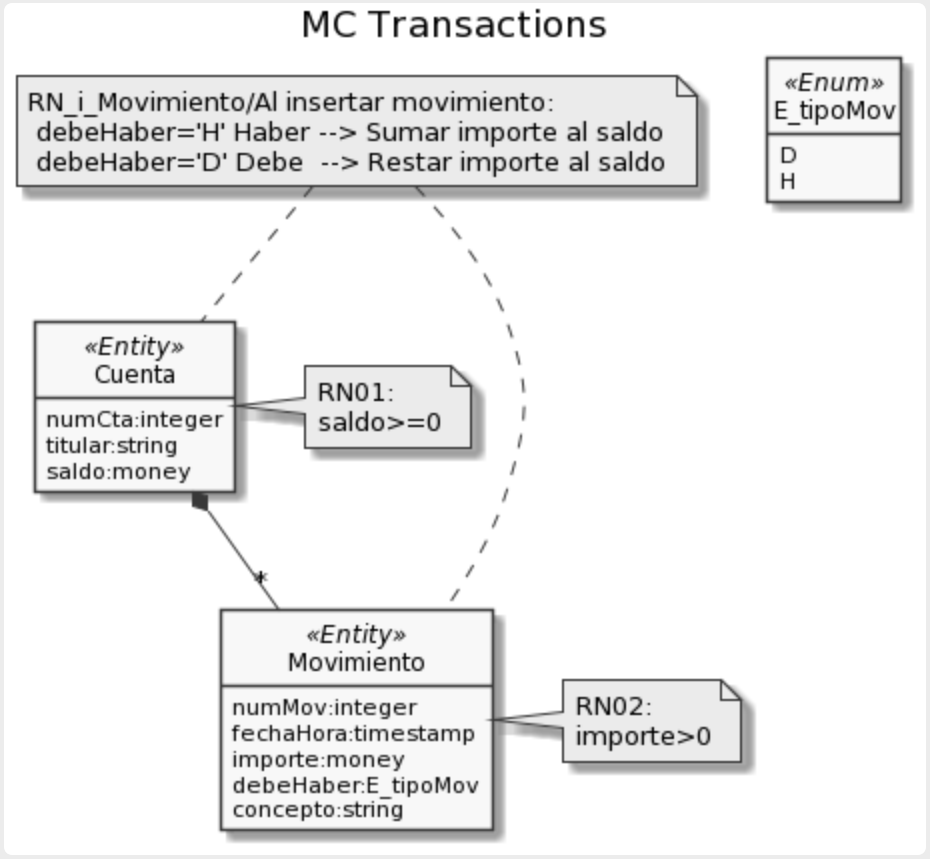


Transacciones en MariaDB. Ejemplo



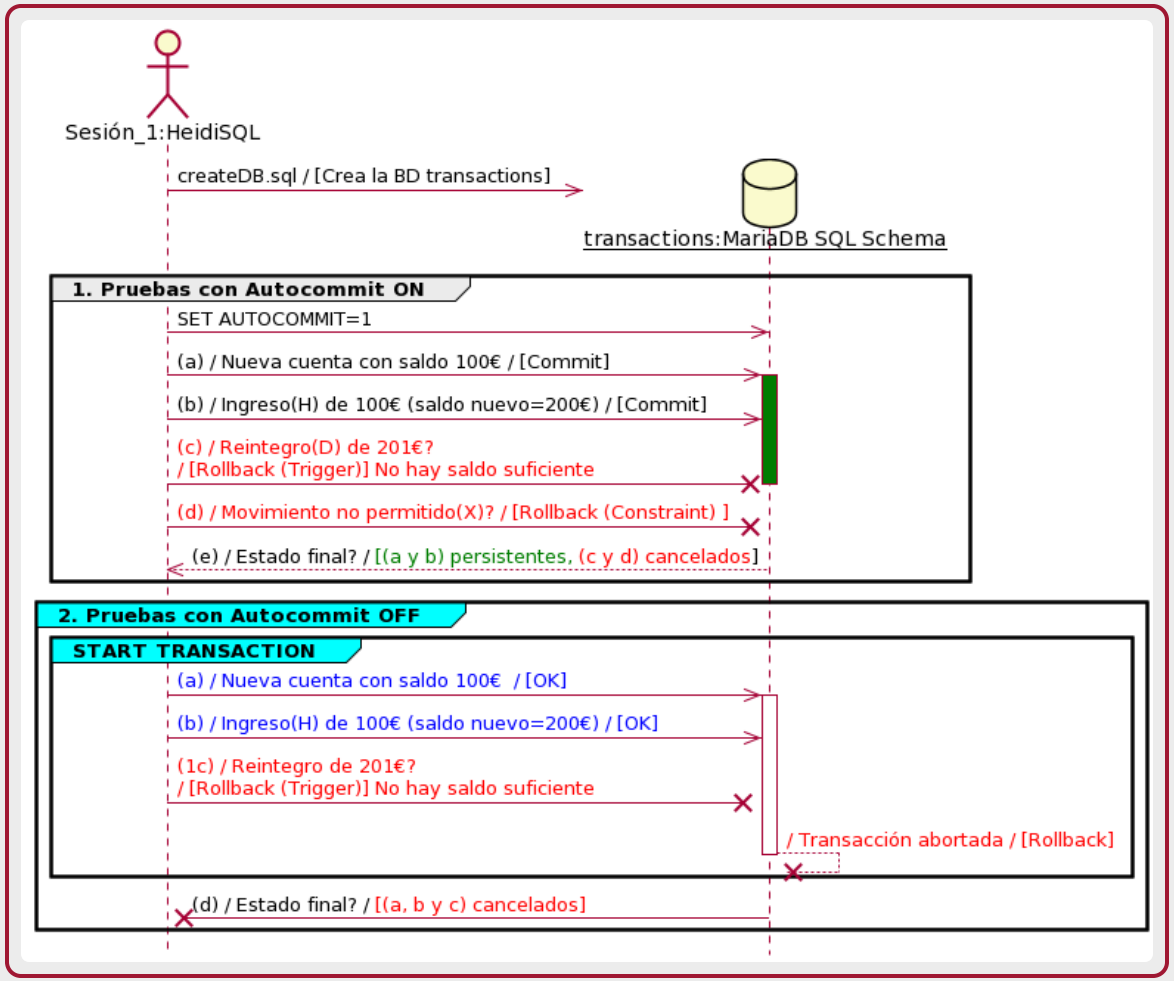

```sql
-- createDb.sql
CREATE DATABASE IF NOT EXISTS transactions;
USE transactions;
DROP TABLE IF EXISTS movimientos;
DROP TABLE IF EXISTS cuentas;
CREATE TABLE cuentas (
  numCta   SMALLINT KEY,
  titular  VARCHAR(20) NOT NULL,
  saldo    DECIMAL(9,2) NOT NULL,
CONSTRAINT CO_RN01_Saldo_Deudor CHECK (saldo>=0)
);
CREATE TABLE movimientos (
  nummov   INTEGER AUTO_INCREMENT KEY,
  fechaHora TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  numCta   SMALLINT NOT NULL,
  importe  DECIMAL(9,2) NOT NULL,
  debeHaberENUM('D','H'),
  concepto     VARCHAR(20) NOT NULL,
  FOREIGN KEY(numCta) REFERENCES cuentas(numCta),
  CONSTRAINT CO_RN02_Importe_Negativo CHECK(importe>0)
);
```

Transacciones en MariaDB. Ejemplo




```
/* Trigger para actualizar saldo con movimientos
   'D': Salidas, 'H': Ingresos
*/
DELIMITER //
CREATE OR REPLACE TRIGGER tMovimientos
AFTER INSERT ON movimientos FOR EACH ROW
BEGIN
  IF (NEW.debeHaber='H') THEN -- Aumenta el saldo
    UPDATE cuentas SET saldo = saldo + NEW.importe
    WHERE numCta=NEW.numCta;
  ELSE  -- Reduce el saldo
    UPDATE cuentas SET saldo = saldo - NEW.importe
    WHERE numCta=NEW.numCta;
END IF;
END; //
DELIMITER ;
```

Transacciones en MariaDB. Ejemplo



Transacciones en MariaDB. Ejemplo AUTOCOMIT=1

Objetivo: Escribir modificaciones hasta que se viole una regla de negocio

```
-- 1. Pruebas con AUTOCOMMIT ON, sin transacciones,
SET AUTOCOMMIT=1; -- Sincronización instrucción a instrucción
-- (a) (Commit): Se añade una cuenta con saldo 100€
INSERT INTO cuentas (numCta, titular, saldo) VALUES (1,'Titular 1',100);
-- (b) (Commit): Se añade un ingreso de 100€ (saldo nuevo=200€)
INSERT INTO movimientos (numCta, importe, debeHaber, concepto)VALUES (1,100,'H','Ingreso');
-- (c) (Rollback):Se intenta insertar un reintegro de 201€, que falla por no haber saldo suficiente (Trigger)
-- Error de SQL (4025): CONSTRAINT `CO_RN01_Saldo_Deudor` failed for `transactions`.`cuentas`
INSERT INTO movimientos (numCta, importe, debeHaber, concepto) VALUES (1,201,'D','Reintegro');
-- (d) (Rollback):Se intenta insertar un movimiento no permitido (Constraint)
INSERT INTO movimientos (numCta, importe, debeHaber, concepto) VALUES (1,10,'X','Reintegro');
-- (e) (Consulta) Cada Commit se ha hecho persistente
```

```
SELECT * FROM cuentas NATURAL JOIN movimientos;
```

Resultado #1 (1r × 8c)

numCta	titular	saldo	nummov	fechaHora	importe	debeHaber	concepto
1	Titular 1	200,00	1	2021-10-04 16:39:46	100,00	H	Ingreso

Transacciones en MariaDB. Ejemplo AUTOCOMIT=0

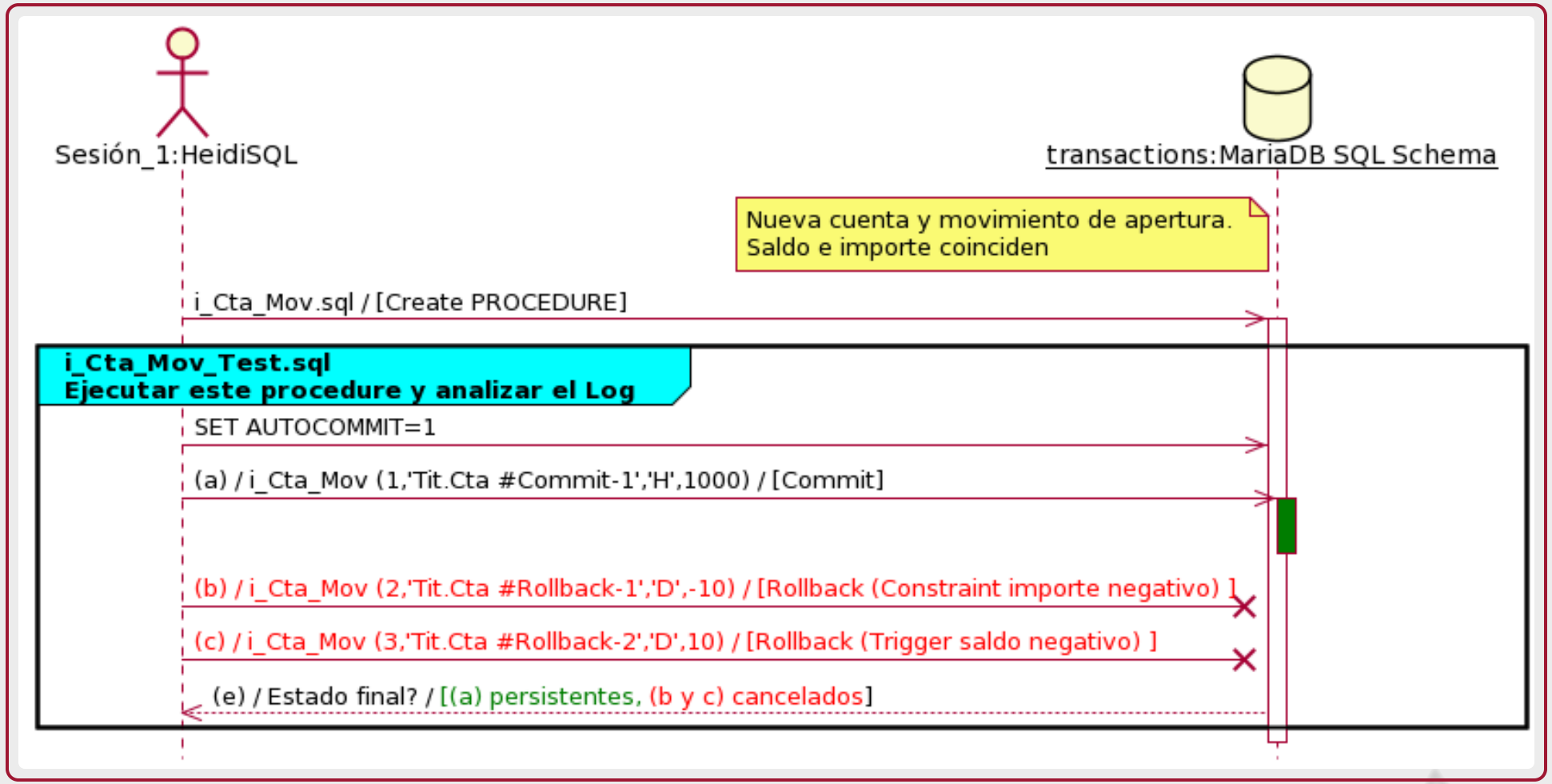
Objetivo: Escribir modificaciones hasta que se viole una regla de negocio

```
-- 2.Pruebas con AUTOCOMMIT OFF y Transacción explícita
SET AUTOCOMMIT=0; -- Activación del control de transacciones
START TRANSACTION; -- Inicio de la transacción
-- (a) (OK): Se añada una cuenta con saldo 100€
INSERT INTO cuentas (numCta, titular, saldo) VALUES (2,'Titular 2',100);
-- (b) (OK): Se añade un ingreso de 100€ (saldo nuevo=200€)
INSERT INTO movimientos (numCta, importe, debeHaber, concepto) VALUES (2,100,'H','Ingreso');
-- (c) (NOK):Se intenta insertar un reintegro de 201€, que falla por no haber saldo suficiente (Trigger)
INSERT INTO movimientos (numCta, importe, debeHaber, concepto) VALUES (2,201,'D','Reintegro');
-- La transacción sigue activa, pendiente de terminación (Commit) o (Rollback)
-- (d) El alumno debe probar a ejecutar manualmente (COMMIT) o (ROLLBACK)
--     y a continuación hacer el SELECT: SELECT * FROM cuentas NATURAL JOIN movimientos;
--     COMMIT WORK;
--         ROLLBACK WORK;
```

Resultado #1 (1r × 8c)

numCta	titular	saldo	nummov	fechaHora	importe	debeHaber	concepto
2	Titular 2	200,00	1	2021-10-31 19:04:43	100,00	H	Ingreso

Transacciones en MariaDB. Ejemplo



Transacciones en MariaDB. Ejemplo

Objetivo: Procedure para abrir cuenta insertando un movimiento, con Commit y Rollback controlados para definir una transacción. O entra la cuenta y su movimiento de apertura, o no entra ninguna de las filas

```
DELIMITER //
CREATE OR REPLACE PROCEDURE `pCta_Mov`(IN p_cta INTEGER, IN p_tit VARCHAR(20),
       IN p_DH CHAR(1), IN p_imp DECIMAL(9,2))
BEGIN # Inserta una Cuenta y un Movimiento como apertura como una transacción
  START TRANSACTION;
  tblock: BEGIN # start: transaction block
  /* catch any exceptions, then rollback */
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN
     GET DIAGNOSTICS @text = MESSAGE_TEXT;
         SET @text = CONCAT('[PR-pCta_Mov] Transacción abortada--> ', @text);
     ROLLBACK;
     SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @text;
    END;
    /* table transactions here */
    INSERT INTO cuentas(numCta,titular,saldo) VALUES(P_cta,P_tit,0);
    INSERT INTO movimientos(numCta,importe,debeHaber,concepto)VALUES (P_cta,P_imp,P_DH,'Apertura cuenta');
    COMMIT; -- Confirmar transacción llegado a este punto
    END tblock; # end: transaction block
END//
DELIMITER ;
```

Transacciones en MariaDB. Ejemplo

Objetivo: Procedure para abrir cuenta insertando un movimiento.

- Commit-1: Si ingreso (‘H’) positivo, se abre la cuenta con dicho importe.
- Rollback-1: Importe negativo. No se abre cuenta ni entra movimiento.
- Rollback-2: Importe positivo cargado (‘D’) sin saldo. No se abre cuenta ni entra movimiento.

```
CALL pCta_Mov (1,'Tit.Cta #Commit-1','H',1000); -- Commit-1
CALL pCta_Mov (2,'Tit.Cta #Rollback-1','D',-10); -- Rollback-1
/* CONSTRAINT `CO_RN02_Importe_Negativo` failed for `transactions`.`movimientos` */
CALL pCta_Mov (3,'Tit.Cta #Rollback-2','D',10); -- Rollback-2
/* CONSTRAINT `CO_RN01_Saldo_Deudor` failed for `transactions`.`cuentas` */
SELECT * FROM cuentas NATURAL JOIN movimientos;
```

Resultado #1 (1r × 8c)

numCta	titular	saldo	nummov	fechaHora	importe	debeHaber	concepto
1	Tit.Cta #Commit-1	1.000,00	1	2021-10-05 15:49:21	1.000,00	H	Apertura cuenta

Conclusiones

Una transacción es una unidad lógica de trabajo que debe completarse íntegramente o no ejecutarse—es el mecanismo que garantiza la coherencia de los datos frente a fallos y acceso concurrente. Las propiedades ACID definen la garantía que proporciona un SGBDR.

La Atomicidad asegura que la transacción es "todo o nada": si falla en cualquier punto, todos los cambios se revierten. Esto previene estados inconsistentes como dinero desaparecido en transferencias bancarias sin aparecer en la otra cuenta.

La Consistencia garantiza que la transacción llevará la base de datos de un estado válido a otro estado válido según las reglas de negocio. Únicamente datos correctos se almacenan; nunca se violarán restricciones de integridad.

El Aislamiento asegura que transacciones concurrentes no interfieren entre sí, evitando lecturas sucias, lecturas fantásma y actualizaciones fantasma. Los niveles de aislamiento (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE) ofrecen diferentes trade-offs entre consistencia y rendimiento.

La Durability garantiza que datos confirmados (committed) persisten incluso ante fallos del sistema. Una transacción completada no se perderá aunque el servidor se caiga. El control de concurrencia mediante locks y log de bits (WAL) implementan estas garantías.

Entender transacciones es crítico para desarrollar aplicaciones confiables que manejan datos importantes. El mal manejo de transacciones causa corrupción de datos, pérdidas financieras y desperfectos en sistemas. Los ORM modernos abstraen este detalle, pero comprender el mecanismo [illegible] esencial.

Gracias

Universidad de Sevilla

Departamento de Lenguajes y Sistemas Informáticos