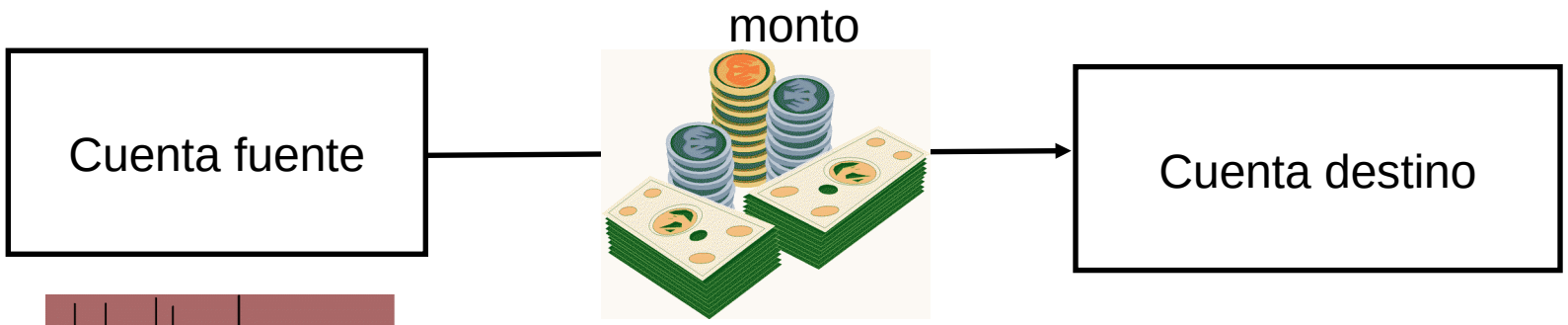
Transacciones

- **Datos inconsistentes**
  - En una BD se puede generar inconsistencia cuando se detiene un sistema, por problemas de suministro, de hardware, cuando varios usuarios acceden al mismo dato, etc. Para solucionar esto existen:
    - Transacciones
    - Niveles de aislamiento y Bloqueos

Concurrencia - Transacciones

Transferencia de dinero desde una cuenta corriente a otra:



Definición de la transacción:

Retirarse de la cuenta fuente disminuyendo (actualizando) el saldo en el monto a transferir.

Depositarse en la cuenta destino, aumentando su saldo con el monto.

Registrarse el evento (en una tabla para tal efecto)

Concurrencia - Transacciones

Transferencia de dinero desde una cuenta corriente a otra:

```
UPDATE ctacte  SET saldo = saldo - 100
    WHERE IdCuenta = 102
```

SE ACTUALIZA LA BASE DE DATOS PERO ANTES DE ACTUALIZAR EL SALDO DE LA CUENTA DESTINO ...

La BD no está íntegra !!! Donde quedó el dinero ???

Concurrencia - Transacciones

Transferencia de dinero desde una cuenta corriente a otra:

```
UPDATE ctacte  SET saldo = saldo - 100
   WHERE IdCuenta = 102
```

Sale de una cuenta

```
UPDATE ctacte SET saldo = saldo + 100
   WHERE IdCuenta = 158
```

Se agrega a la otra

```
INSERT INTO movim (idMov, tipoMov, fecha,
                   ctaOri, ctaDest, monto)
          VALUES ( 10256, 8,CURRENT_DATE,
                   102, 258, 100 )
```

Se asienta el movimiento

COMMIT

recién se "compromete" a la base de datos con las acciones

Concurrencia - Transacciones

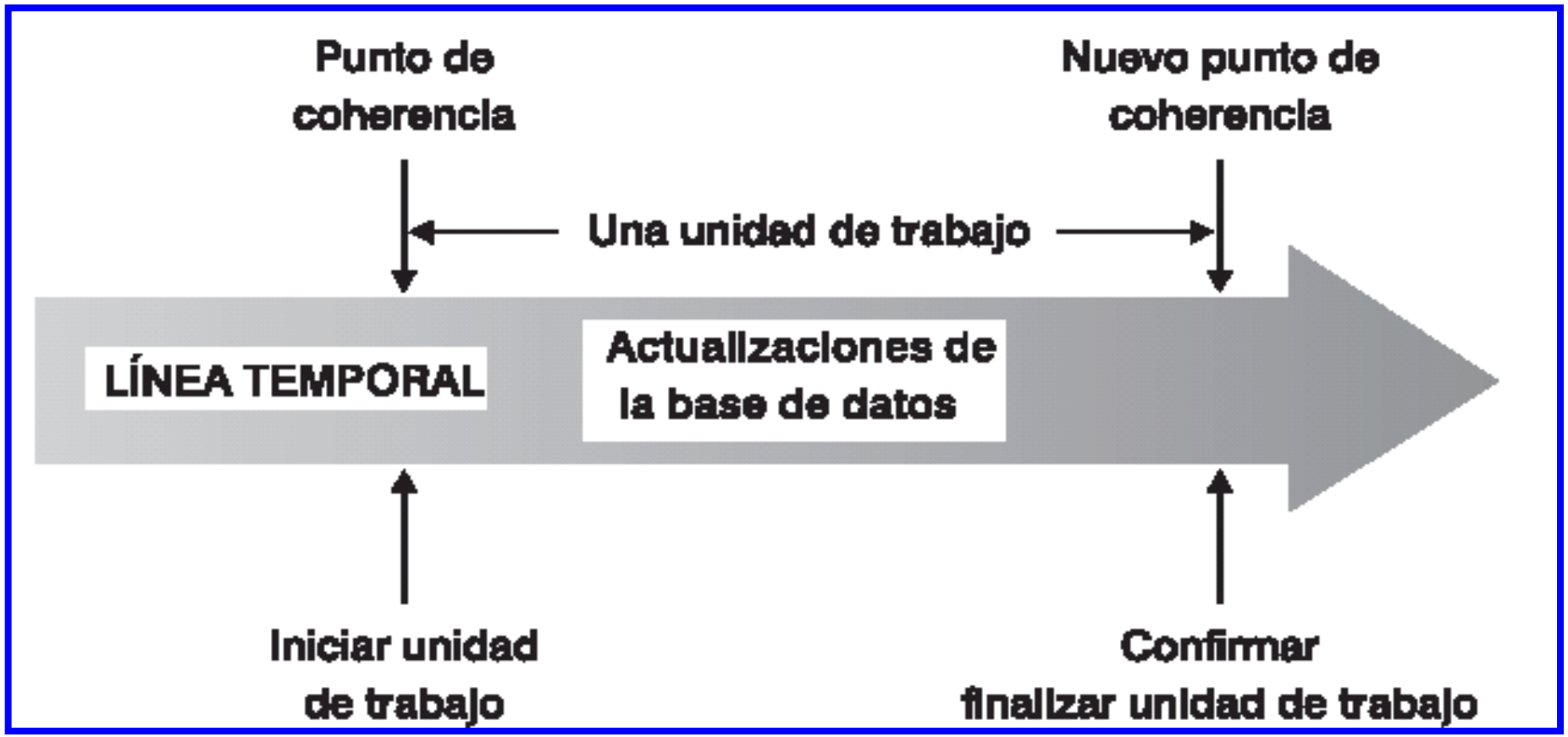
Transferencia de dinero desde una cuenta corriente a otra:

```
UPDATE ctacte  SET saldo = saldo - 100
   WHERE IdCuenta = 102

UPDATE ctacte SET saldo = saldo + 100
   WHERE IdCuenta = 158

INSERT INTO movim (idMov, tipoMov, fecha,
                   ctaOri, ctaDest, monto)
            VALUES ( 10256, 8,CURRENT_DATE,
                     102, 258, 100 )

If (error en alguna de las instrucciones)
      ROLLBACK
Else
      COMMIT
```

Las modificaciones nunca se producen en la BD

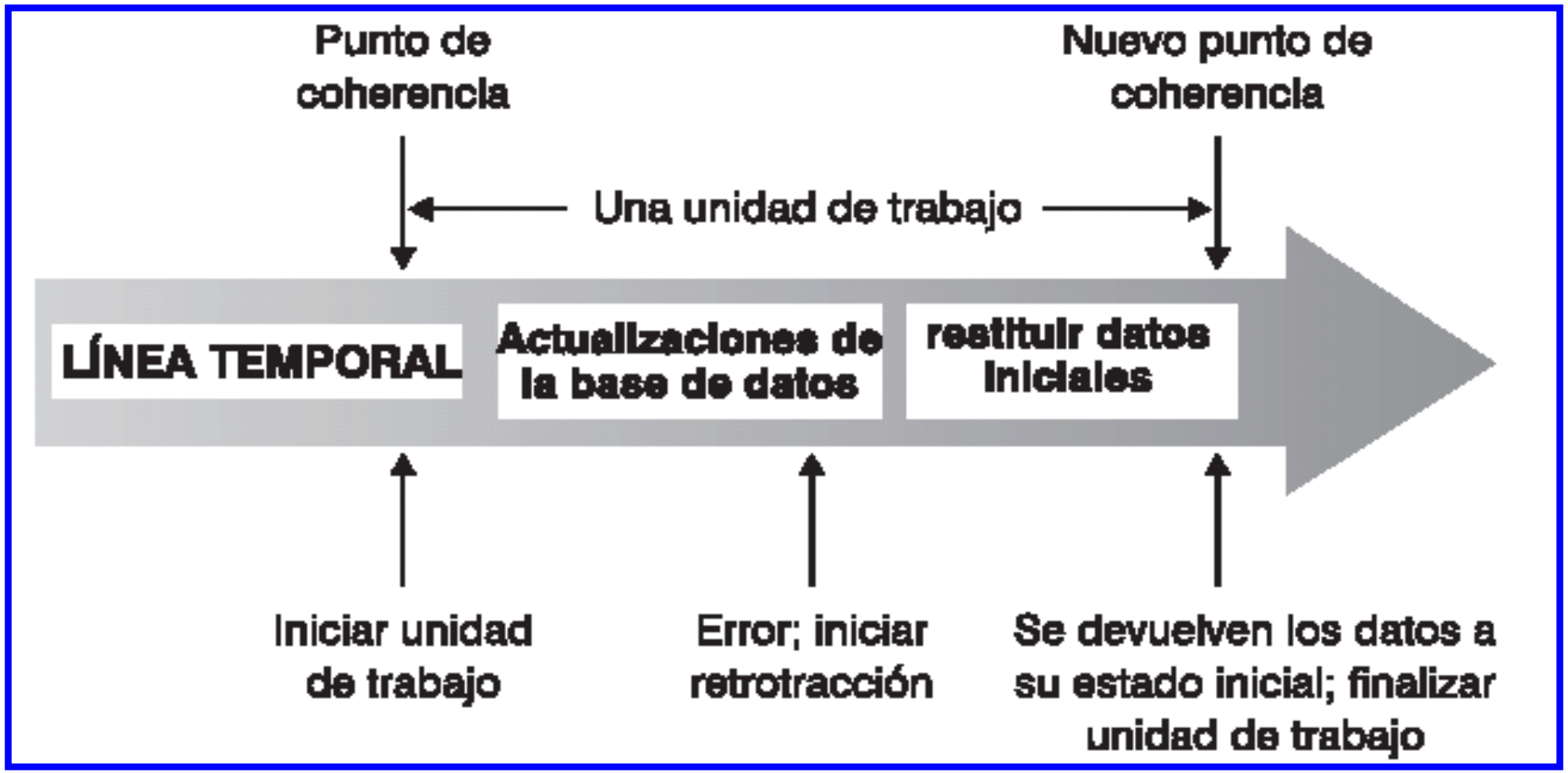
Concurrencia – Transacción o Unidad de Trabajo

Es una secuencia recuperable de operaciones dentro de un proceso de aplicación, la secuencia te agrupa como un todo.



Concurrencia – Transacción o Unidad de Trabajo

Transacción con anomalía.



Concurrencia – Transacción o Unidad de Trabajo

Se inicia cuando:

- **Se inicia un PA.**
- **Termina la UT anterior por otro motivo que no sea el fin del PA.**

Finaliza cuando:

- **Se pide la confirmación de los cambios.**
- **Se pide la retrotracción de los cambios.**
- **Finaliza el PA.**

El inicio y finalización de una UT definen los puntos de coherencia.

Concurrencia – Transacción o Unidad de Trabajo

Las Transacciones seguras deben cumplir con:

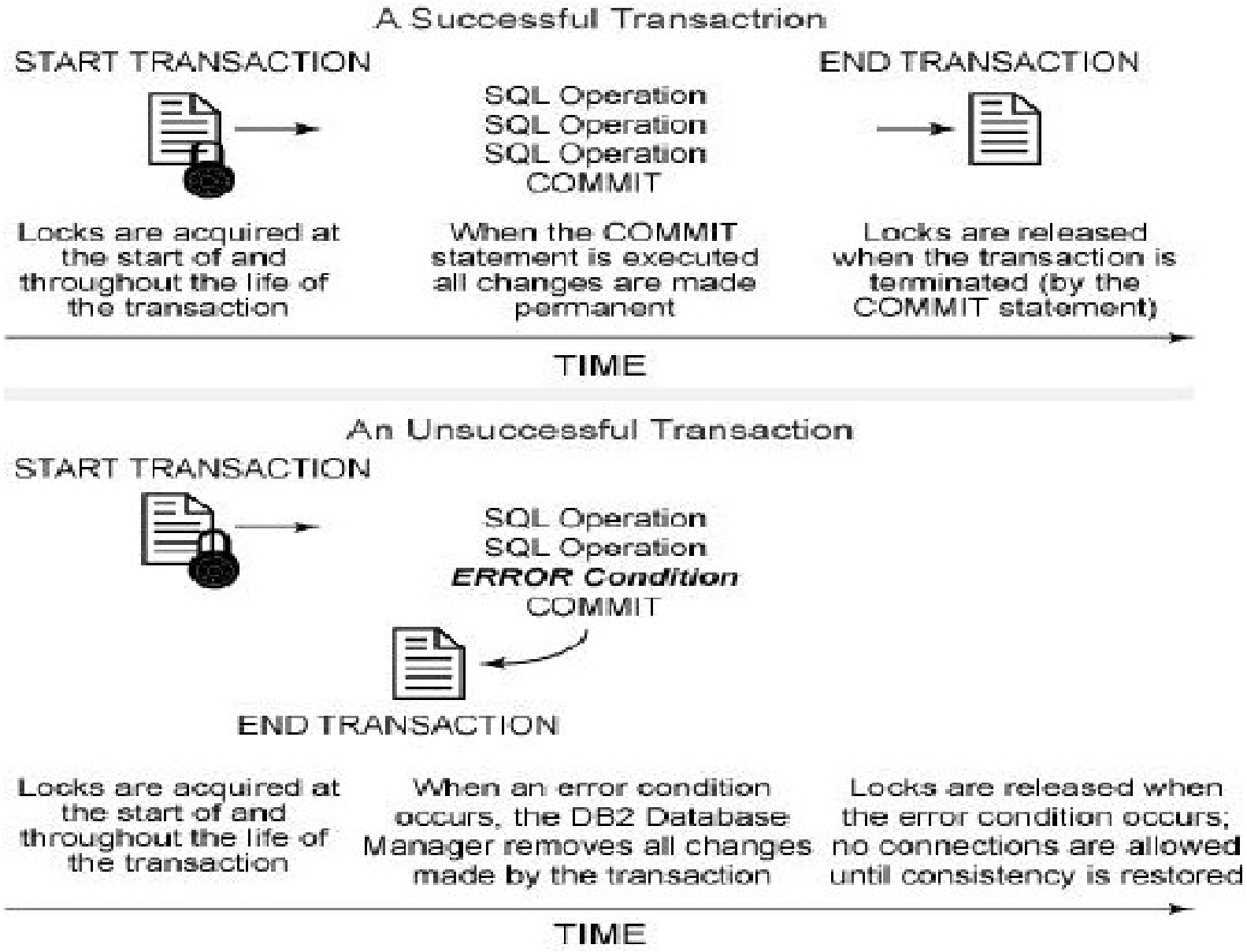
Propiedades ACID: acrónimo de **A**tomicity, **C**onsistency, **I**solation and **D**urability: **D**urabilidad, **A**islamiento, **C**onsistencia e **I**ndivisibilidad

- Atomicidad**:** asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.
- Consistencia**:** asegura que sólo se empieza aquello que se puede acabar. Solo se ejecutan aquellas operaciones que no van a romper la reglas y directrices de integridad de la base de datos.
- Aislamiento**:** asegura que una operación no puede afectar a otras. Esto es, dos transacciones sobre la misma información nunca generará ningún tipo de error.
- Durabilidad**:** asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

Concurrencia – Transacción o Unidad de Trabajo

- Cuando una UT se inicia los datos que se manejan en ella son bloqueados para que otros procesos no ocasionen inconsistencia.
- En el caso de retrotraer los cambios no efectuados (uncommitted data), para restaurar la consistencia de la BD, se usan de los archivos “Log” de transacciones, que continen:
  - Información acerca de cada sentencia SQL ejecutada por una UT
  - Además, si en esa UT fue o no ejecutada exitosamente (commit o rollback).

Concurrencia – Transacción o Unidad de Trabajo



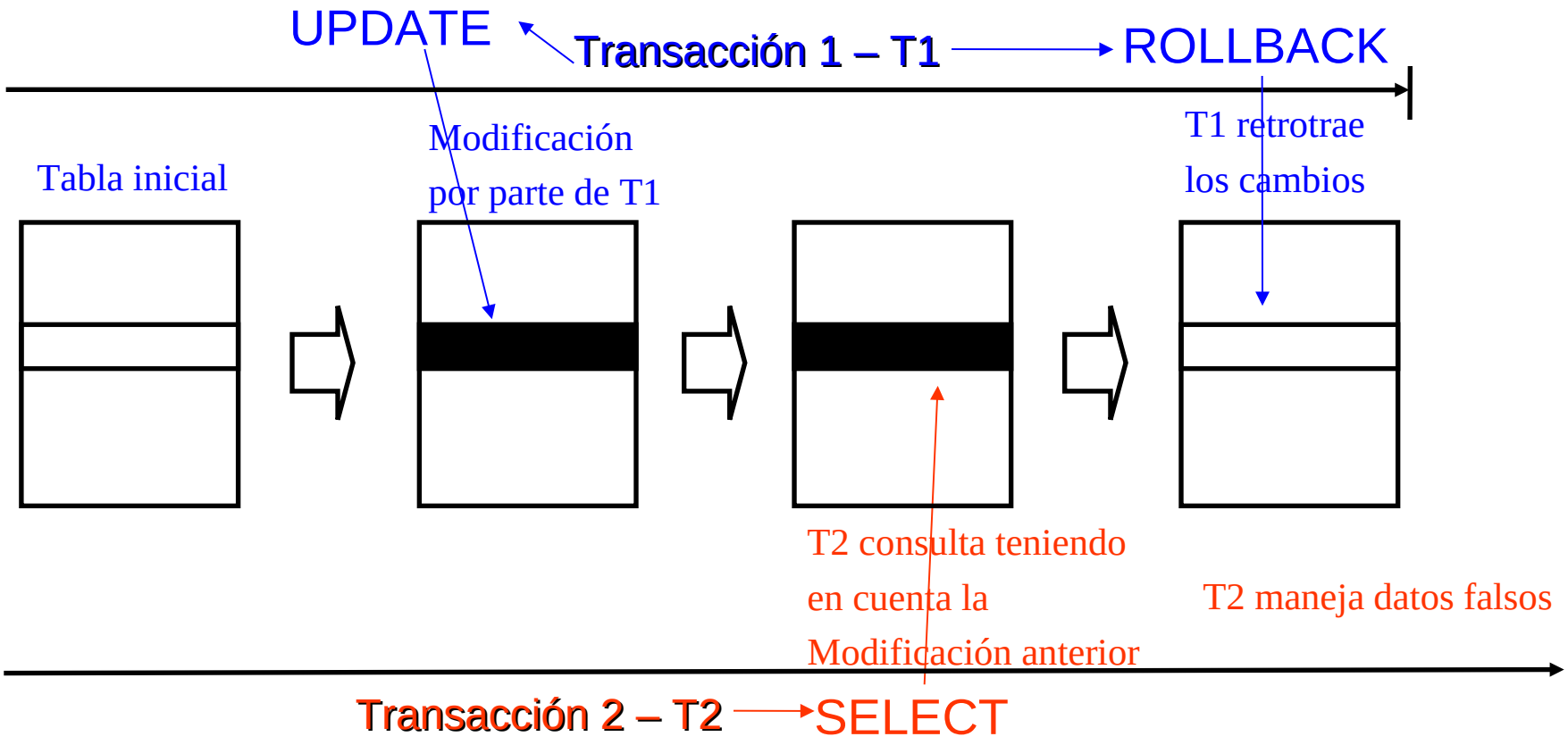
Concurrencia – Transacción o Unidad de Trabajo

- Mientras sea un solo proceso el que accede a los datos, no existen demasiados problemas y el uso de Transacciones los soluciona casi a todos.

- **¿Qué ocurre cuando los datos son accedidos por múltiples procesos simultáneamente?**

Concurrencia – Anomalías en UT por concurrencia

Lectura falsa (o sucia) (uncommitted read)



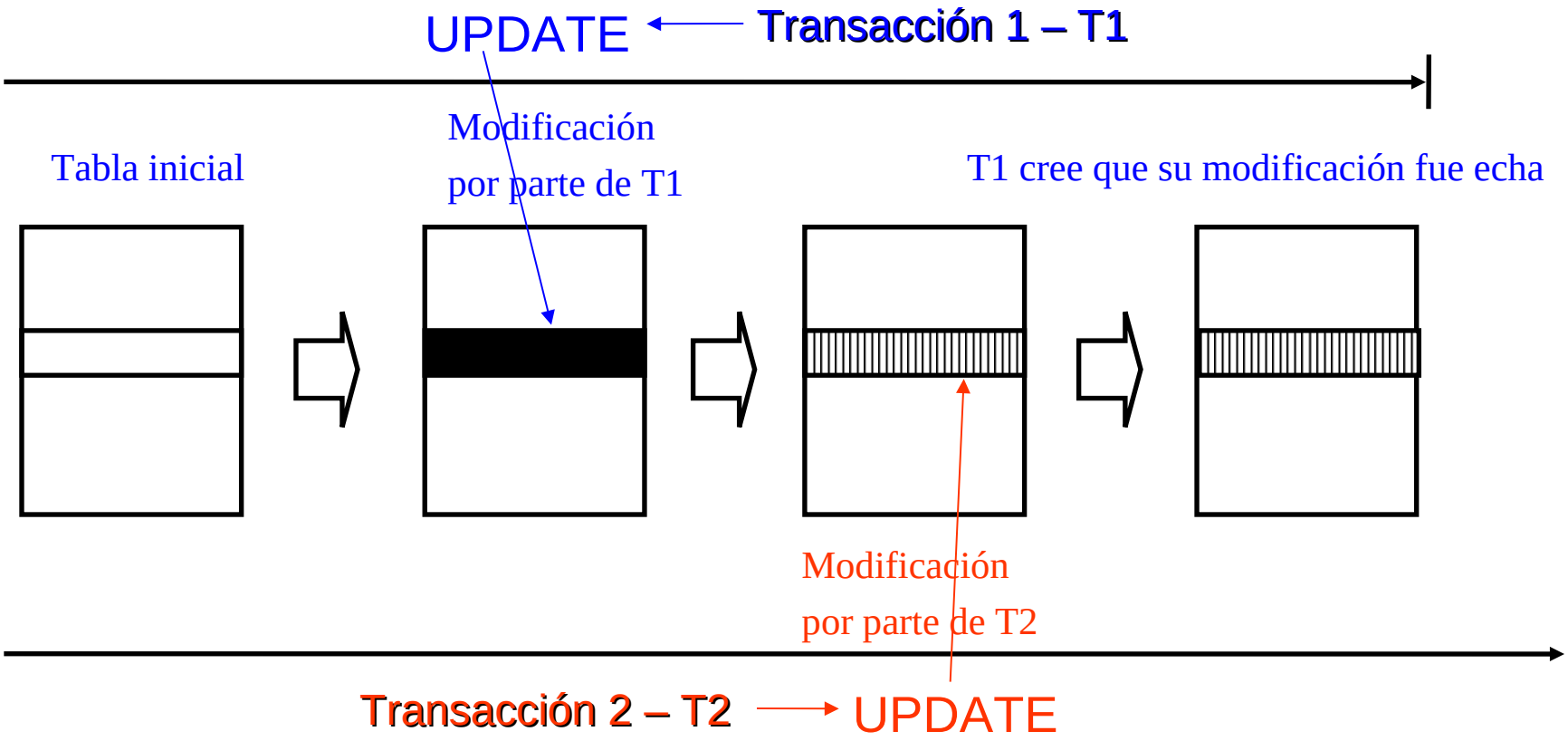
Concurrencia – Anomalías en UT por concurrencia

1. En resumen cuando se da un caso de Lectura falsa o sucia (uncommitted read)

1. Una instrucción dentro de transacción T1 hace algún tipo de modificación
2. Luego en otra transacción T2 se consultan los datos con esa modificación
3. Luego T1 retrotrae los datos al estado inicial (sin la modificación)
4. Por lo tanto T2 posee datos que no son reales (falsos)

Concurrencia – Anomalías en UT por concurrencia

Actualización perdida (lost update)



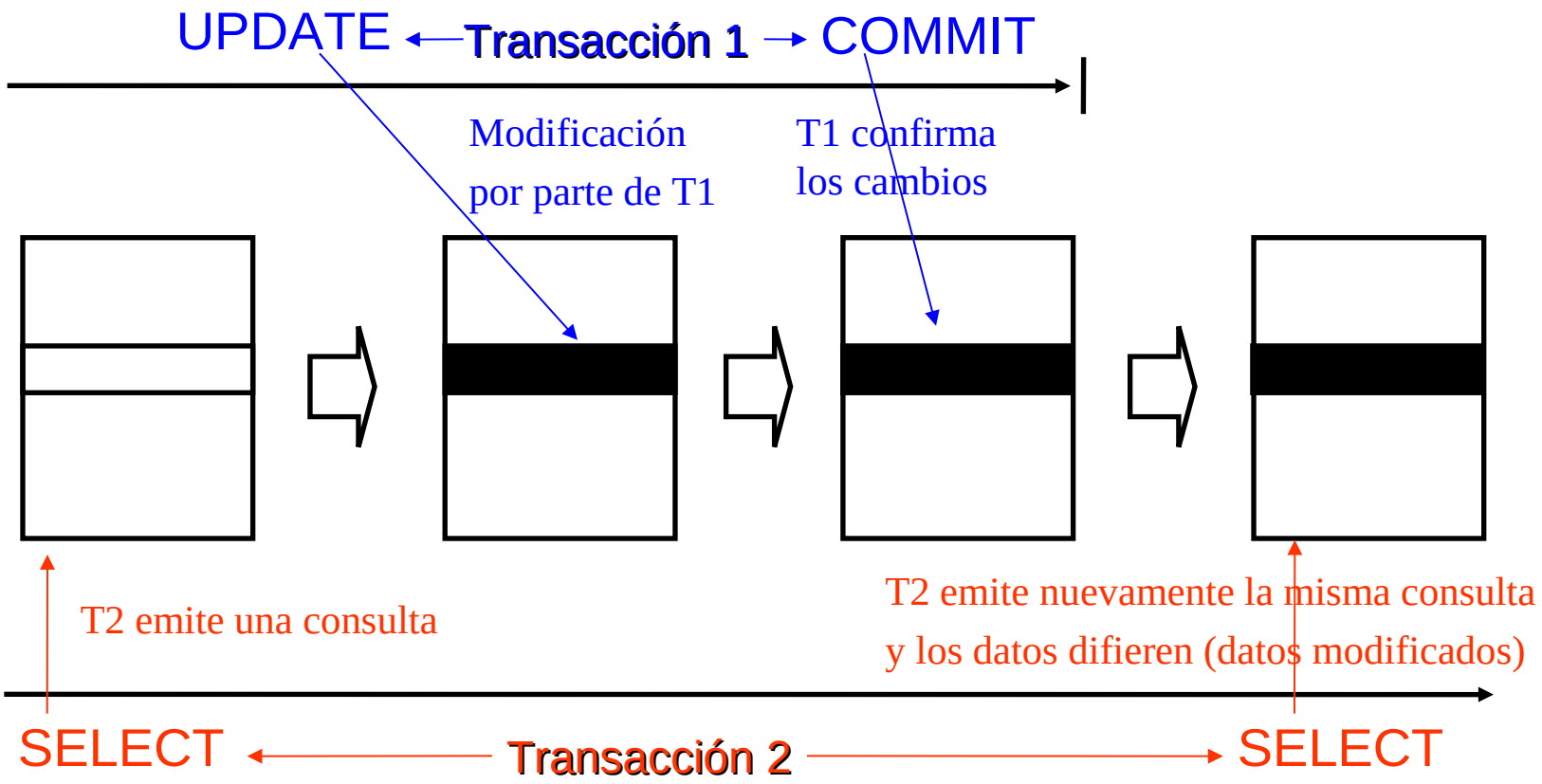
Concurrencia – Anomalías en UT por concurrencia

- En resumen cuando se da un caso de Actualización perdida (lost update)

1. Una instrucción dentro de transacción T1 hace algún tipo de modificación a un dato
2. Luego en otra transacción T2 alguna instrucción realiza otra modificación al mismo dato.
3. Luego T1 confirma los cambios y cree que su modificación fue echa correctamente pero en realidad se perdió

Concurrencia – Anomalías en UT por concurrencia

Lectura irrepetible (nonrepeatable read)



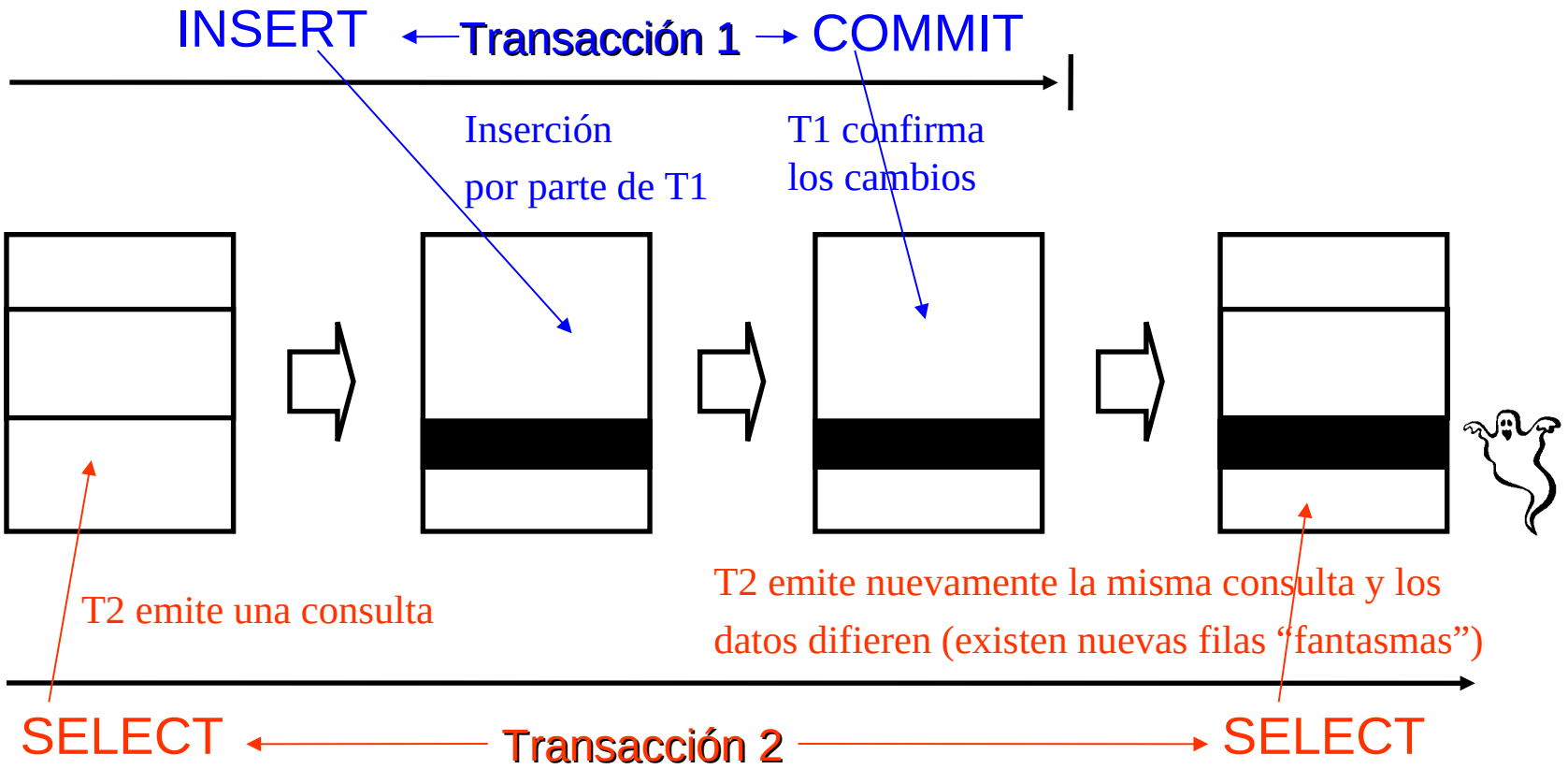
Concurrencia – Anomalías en UT por concurrencia

- En resumen cuando se da un caso de Lectura irrepetible (nonrepeatable read)

1. En una transacción T2 se emite una consulta.
2. Luego en otra transacción T1 alguna instrucción realiza otra modificación a un dato que responde al predicado de la consulta en T2.
3. T1 Confirma los cambios.
4. Luego T2 emite la misma consulta y los resultados difieren de la anterior ya que existen datos modificados.

Concurrencia – Anomalías en UT por concurrencia

Lectura fantasma (phantom read)



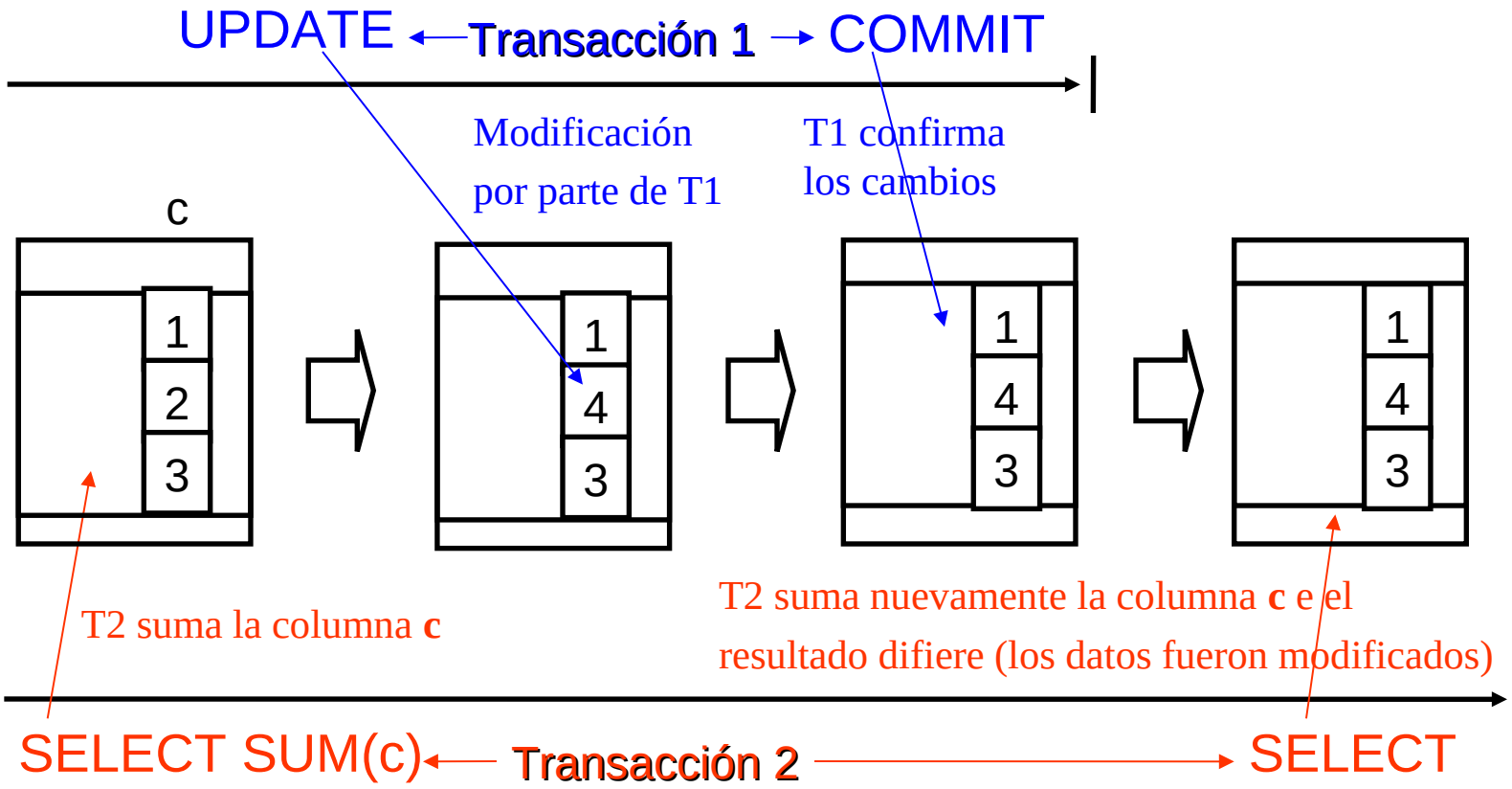
Concurrencia – Anomalías en UT por concurrencia

- En resumen cuando se da un caso de Lectura fantasma (phantom read)

1. En una transacción T2 se emite una consulta.
2. Luego en otra transacción T1 alguna instrucción inserta una nueva fila que responde al predicado de la consulta en T2.
3. T1 Confirma los cambios.
4. Luego T2 emite la misma consulta y los resultados difieren de la anterior ya que existen nuevas filas “fantasmas”.

Concurrencia – Anomalías en UT por concurrencia

Operación inconsistente



Concurrencia – Anomalías en UT por concurrencia

- En resumen cuando se da un caso de Operación inconsistente

1. En una transacción T2 se intenta calcular un dato basado en una función de columna.
2. Luego en otra transacción T1 alguna instrucción modifica un dato en la columna utilizada por la consulta en T2.
3. T1 Confirma los cambios.
4. Luego T2 emite la misma consulta y el resultado difiere de la anterior ya que existen datos modificados en la columna.

Ejemplo MySQL

START TRANSACTION; ← Inicia una Transacción

COMMIT; ← Confirma los cambios

ROLLBACK; ← Retrotrae los cambios

```
START TRANSACTION;
  SELECT * FROM zonas z;
  INSERT INTO zonas VALUES (100,'Zona 100');
  SELECT * FROM zonas z;
ROLLBACK;
  SELECT * FROM zonas z;
```

Ejemplo MySQL

SAVEPOINT *nombre*; ← Define un punto intermedio de retrotracción.
ROLLBACK TO SAVEPOINT *nombre*; ← Retrotrae a un punto intermedio.

```
START TRANSACTION;
  SELECT * FROM zonas z;
  INSERT INTO zonas VALUES (100,'Zona 100');
SAVEPOINT zona_101;
  INSERT INTO zonas VALUES (101,'Zona 101');
  SELECT * FROM zonas z;
ROLLBACK TO SAVEPOINT zona_101;
  SELECT * FROM zonas z;
COMMIT;
```

Niveles de aislamiento

Definición

Los **Niveles de Aislamiento** determinan como se gestionan las transacciones concurrentes y especifican que datos son visibles entre estas transacciones.

 Es fundamental la buena selección de los niveles de aislamiento a fin de aprovechar los recursos, y maximizar la concurrencia sin que esto vaya en detrimento de la congruencia de los datos.

Tipos

- **Serializable**: organiza las transacciones con el objetivo de que no entren en conflicto. Es el más estricto y el que más sacrifica rendimiento y concurrencia.

- **Lectura Confirmada**: es el predeterminado en la mayoría de los SGBDR. Dentro de una transacción, solo se pueden ver los cambios confirmados. Soluciona el problema de la lectura falsa (o sucia)

Tipos

- **Lectura Repetible**: todas las tuplas leídas en la transacción, son bloqueadas. Se soluciona el problema de lectura sucia y lectura no repetible.

- **Lectura no Confirmada**: con este nivel se pueden ver los resultados de las transacciones no confirmadas. Es el nivel más laxo, permite una alta concurrencia, pero aumenta el grado de potencial incongruencia.

Ejemplos MySQL

Niveles en MySQL:

1) READ-UNCOMMITTED: lee datos no confirmados
2) READ-COMMITTED: lee datos confirmados (por defecto)
3) REPEATABLE-READ: no puede leer datos no confirmados por otras transacciones, los datos leídos no se pueden modificar.
4) SERIALIZABLE: mantiene inalterables los datos leídos, por más que estos sean modificados por otras transacciones.

Ejemplos MySQL

`SHOW VARIABLES LIKE 'transaction_isolation';` ← muestra nivel actual

`SET @@session.transaction_isolation = 'SERIALIZABLE';` ← establece nivel actual para la sesión.

Auto-Commit Transactions: cuando está habilitado, cada instrucción es una transacción

`SHOW VARIABLES LIKE 'autocommit';` ← ver valor actual

`SET autocommit=0;` ← 0 inhabilita, 1 habilita

Ejemplos MySQL (READ-UNCOMMITTED)

```
SET autocommit=0;
SET @@session.transaction_isolation =  'READ-UNCOMMITTED';
START TRANSACTION;
INSERT INTO practico.productos (idProducto, producto, precio, idRubro, idProveedor)
            VALUES (999, 'No confirmado', 33.36, 6, 7);
----- Ejecutar primero, sin COMMIT o ROLLBACK

COMMIT; o ROLLBACK;
```

```
SET @@session.transaction_isolation =  'READ-UNCOMMITTED';
START TRANSACTION;
SELECT * FROM practico.productos WHERE idProducto=999;
COMMIT;
```

Ejemplos MySQL (READ-COMMITTED)

```
SET autocommit=0;
SET @@session.transaction_isolation =  'READ-COMMITTED';
START TRANSACTION;
UPDATE practico.productos SET producto='Modificado' WHERE idProducto=1;
----- Ejecutar primero, sin COMMIT

COMMIT;
```

```
SET @@session.transaction_isolation =  'READ-COMMITTED';
START TRANSACTION;
SELECT * FROM practico.productos;
COMMIT;
```

Ejemplos MySQL (REPEATABLE-READ)

```
SET autocommit=0;
SET @@session.transaction_isolation =  'REPEATABLE-READ';
START TRANSACTION;
INSERT INTO practico.productos (idProducto, producto, precio, idRubro, idProveedor)
            VALUES (9999, 'Repetible', 16.45, 3, 5);
----- Ejecutar primero, sin COMMIT

COMMIT;
```

```
SET @@session.transaction_isolation =  'REPEATABLE-READ';
START TRANSACTION;
SELECT * FROM practico.productos WHERE idProducto=9999;
------ No se lee hasta que no se confirme, se obtendrán siempre los mismos datos

COMMIT;
```

Ejemplos MySQL (SERIALIZABLE)

```
SET @@session.transaction_isolation =  'SERIALIZABLE';
START TRANSACTION;
SELECT * FROM practico.productos WHERE idProducto=10;
----- Ejecutar primero (1), sin COMMIT

----- Ejecutar tercero (3): COMMIT
COMMIT;
```

```
SET autocommit=0;
SET @@session.transaction_isolation =  'SERIALIZABLE';
START TRANSACTION;
UPDATE practico.productos SET producto='Serializable' WHERE idProducto=10;
----- Ejecutar segundo (2), sin COMMIT

----- Cuarto (4), observar que luego de (3), se ejecutó el update si es que no hubo timeout

COMMIT;
```