Control de concurrencia

Administración de Bases de Datos

Departamento de Lenguajes y Sistemas Informáticos

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Contenido

1. Motivación
2. Problemas de concurrencia
3. Planes de transacciones
4. Protocolos de serialización
5. Técnicas de control de concurrencia
6. Niveles de aislamiento

Motivación

En el tema anterior trabajamos con transacciones sin fijarnos en su contexto de ejecución.

Sin embargo…

- Los sistemas informáticos actuales son multi-usuario.
- Algunas aplicaciones requieren acceso simultáneo a los mismos datos.

Motivación

- Compañía: *Uber*[1]
  - Proveedor de "Movilidad como servicio".
  - Opera en más de 900 áreas metropolitanas.
  - En junio 2021: ~101M de usuarios activos.

- Arquitectura software "Big Data"[2]:
  - Clúster con 100.000 cores virtuales.
  - Más de 100 PBytes de datos.
  - Más de 120.000 consultas SQL al día.
    - *Cifras de 2017*

[1] Uber @ Wikipedia: https://en.wikipedia.org/wiki/Uber
[2] Uber's Big Data Platform: https://eng.uber.com/uber-big-data-platform/

Motivación

- Compañía: Monzo[1]
  - Banco británico digital
    - Similar a Revolut/N26/...
  - En marzo 2020: más de 4M de clientes

- Datos de su BBDD:
  - Rendimiento: más de 300.000 lecturas / segundo[2].
  - Tamaño: 2 tablas clave[3]:
    - *ledger_entries* (›20 billones de filas)
      - Una fila por cada transacción económica realizada.
    - *user_stats* (›4 billones de filas y ›1.000 columnas)
      - Una fila por cada usuario y día activo.

[1]Monzo @ Wikipedia: https://en.wikipedia.org/wiki/Monzo
[2]Monzo Tech blog Dec. 2019: https://monzo.com/blog/we-secured-thousands-of-cassandra-clients-to-keep-monzos-data-safe
[3]Monzo community fórum: https://community.monzo.com/t/monzonaut-ama-theo-borrowing-data-scientist/121560/59?page=4

Motivación

- Asegurar que cada transacción se ejecuta de forma independiente del resto es fundamental.

- Se considera la transacción como unidad lógica de concurrencia.
  - Propiedad *Isolation* de ACID

Concurrencia y paralelismo

- Los sistemas actuales:
  - Hardware: CPUs multi-nucleo (o multi-procesador)
  - Software: SSOO multi-usuario y multi-proceso

- *Concurrencia*: un mismo procesador ejecuta varios procesos de forma simultánea.

- *Paralelismo*: varios procesadores ejecutan varios procesos de forma simultánea.

Problemas de concurrencia

- Al ejecutar 2 (o más) transacciones de forma concurrente pueden surgir diferentes problemas.

- Ejemplo:
  - Sistema de reservas de asientos para vuelos
  - 2 transacciones:

```
READ(X)
X = X - N
WRITE(X)
READ(Y)
Y = Y + N
WRITE(Y)
```

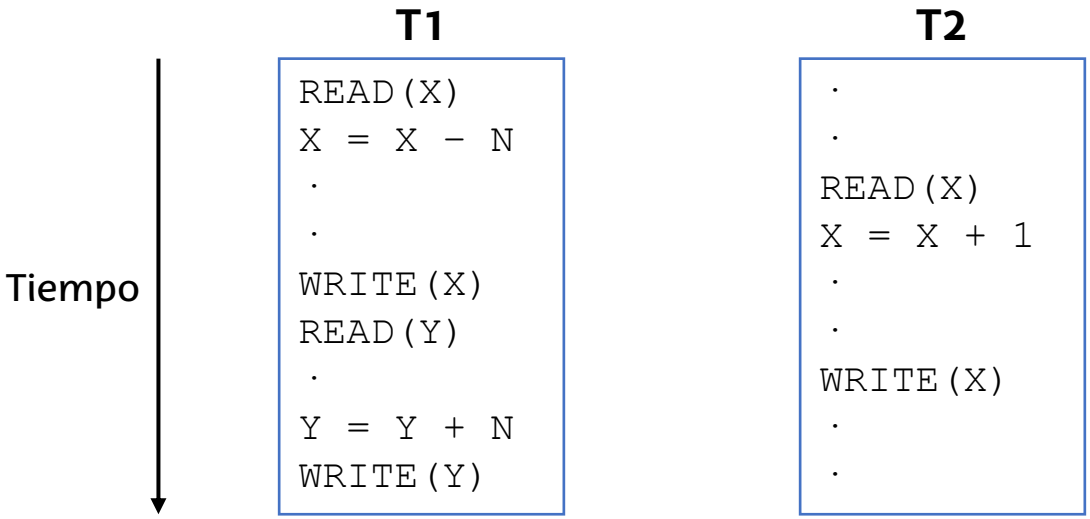
T1: Transfiere N reservas del vuelo X al vuelo Y

```
READ(X)
X = X + 1
WRITE(X)
```

T2: Reserva un asiento en el vuelo X

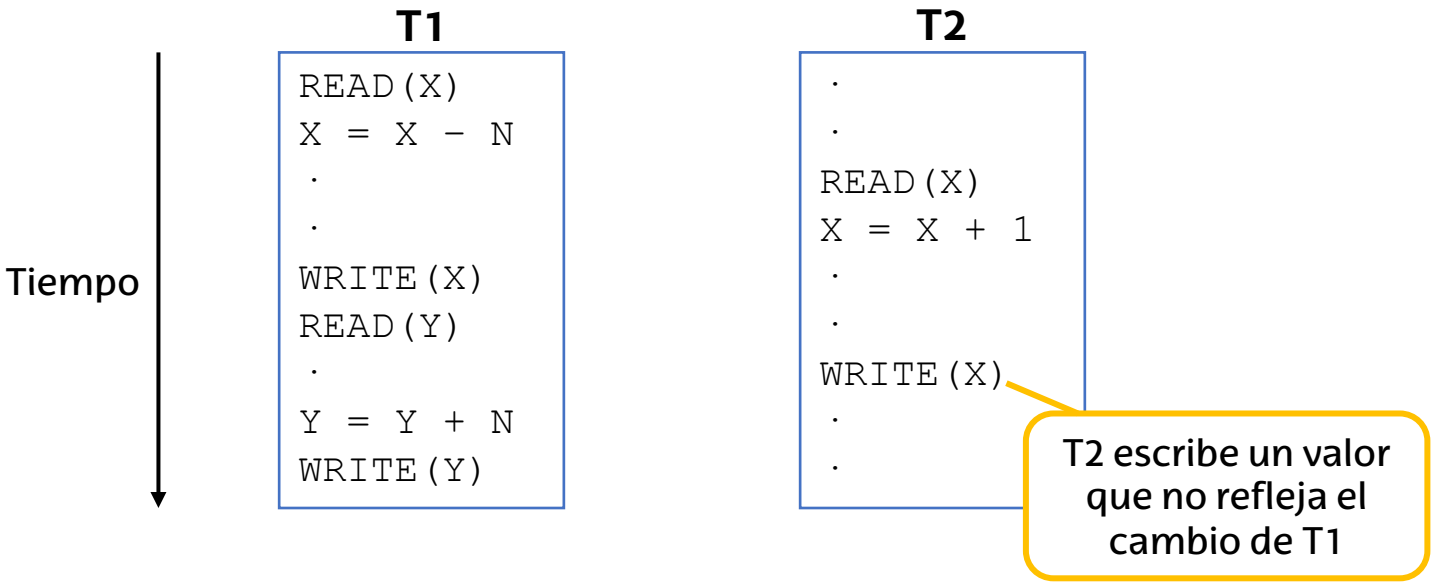
Problemas de concurrencia

- *Suponiendo que ambas transacciones se ejecutan de forma concurrente:*
- Situación 1:



Problemas de concurrencia

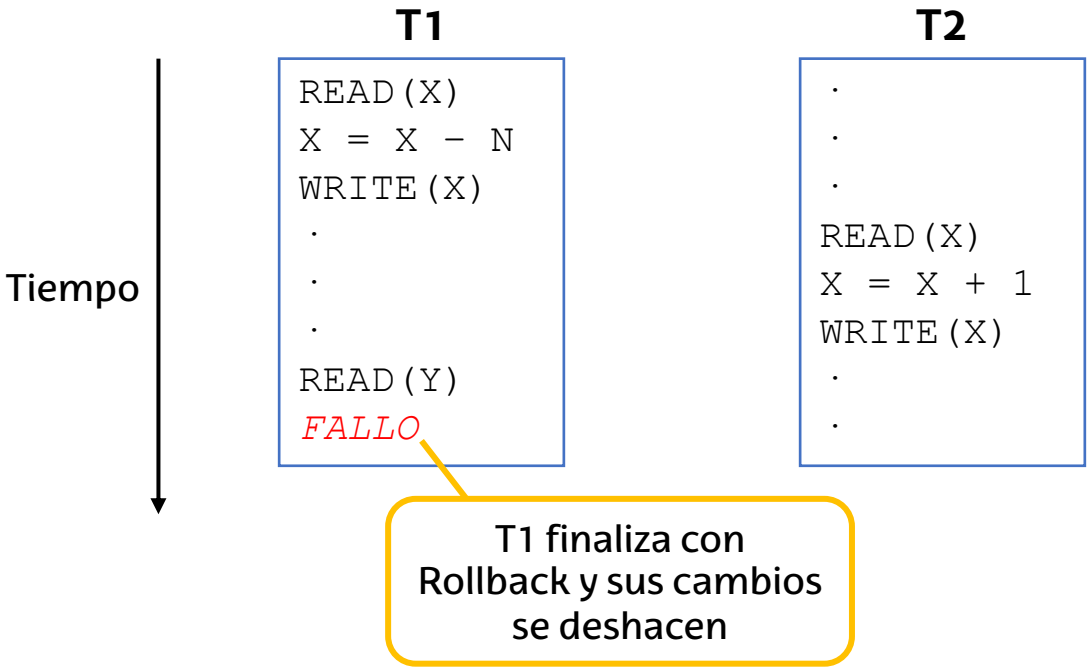
- Situación 1:



- *Problema*: Pérdida de actualizaciones
  - T2 ha leído un valor de X que no refleja la operación `X = X - N`.

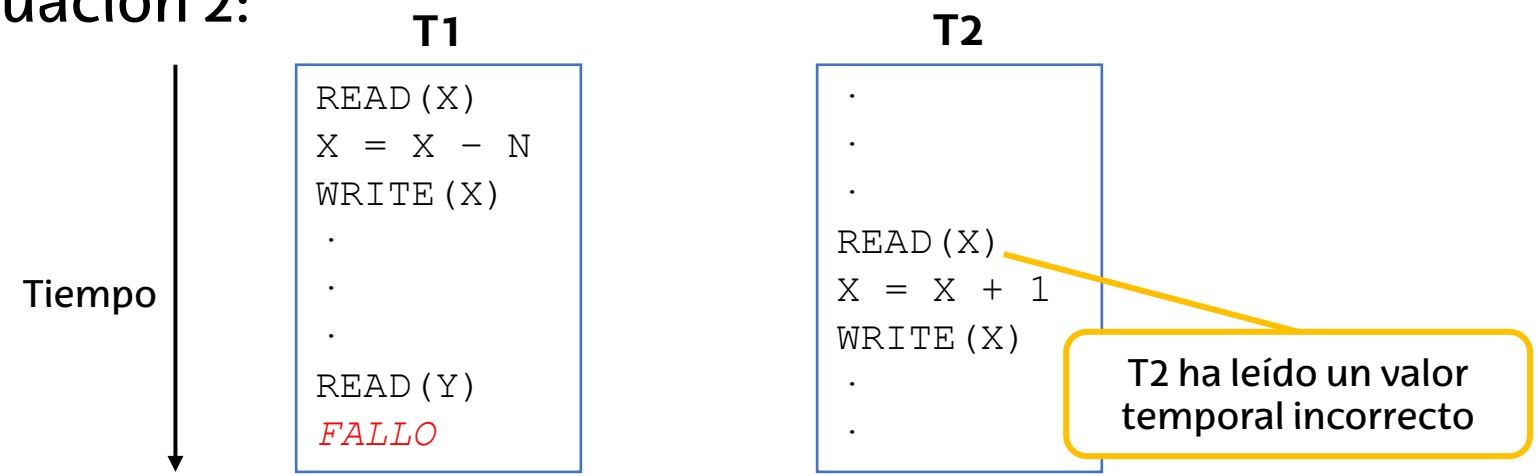
Problemas de concurrencia

- *Suponiendo que ambas transacciones se ejecutan de forma concurrente:*
- Situación 2:



Problemas de concurrencia

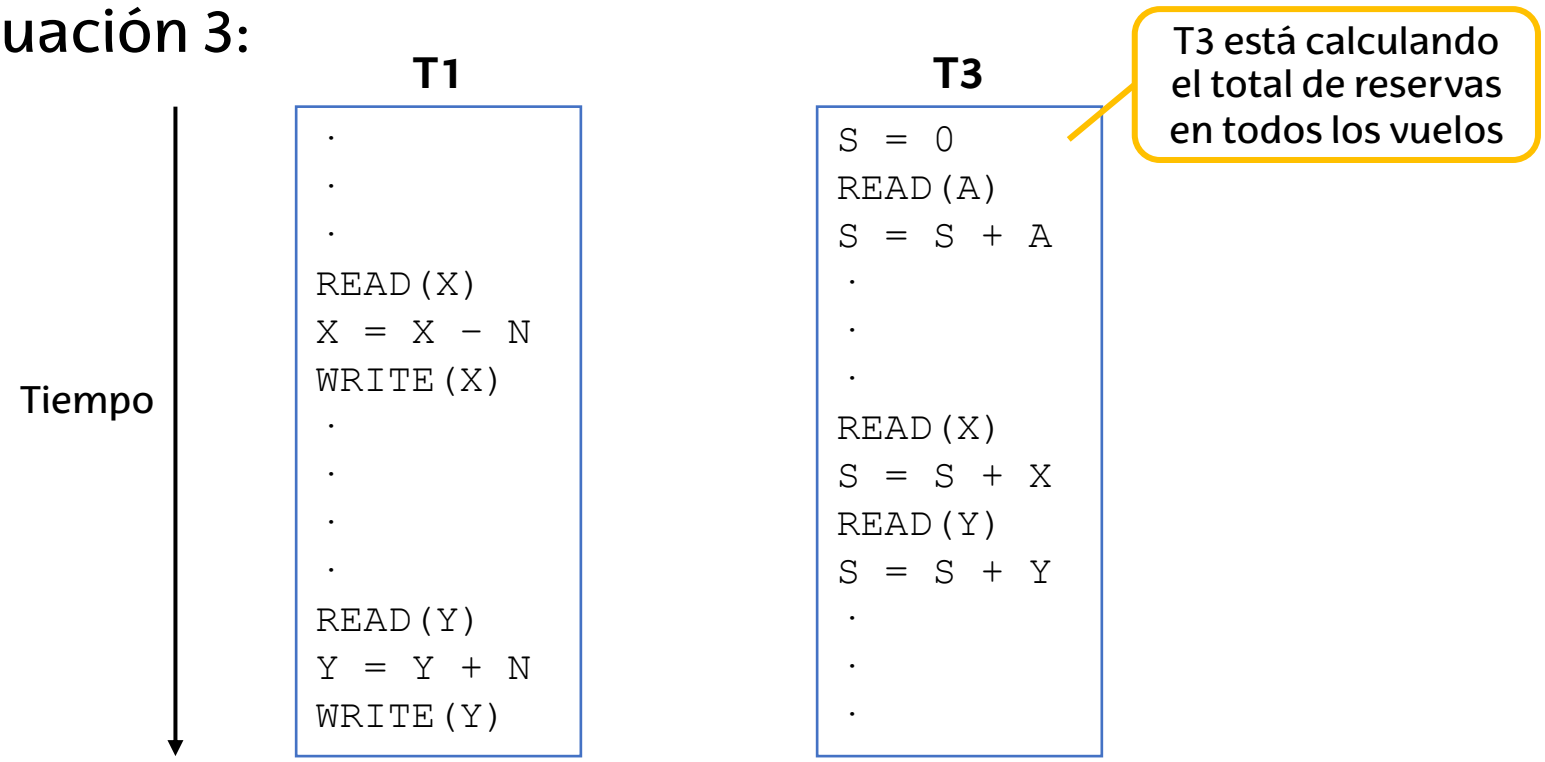
- Situación 2:



- *Problema*: Actualizaciones temporales
  - También llamado "lectura de valores sucios"
  - El final de T1 restaura el valor original de X.

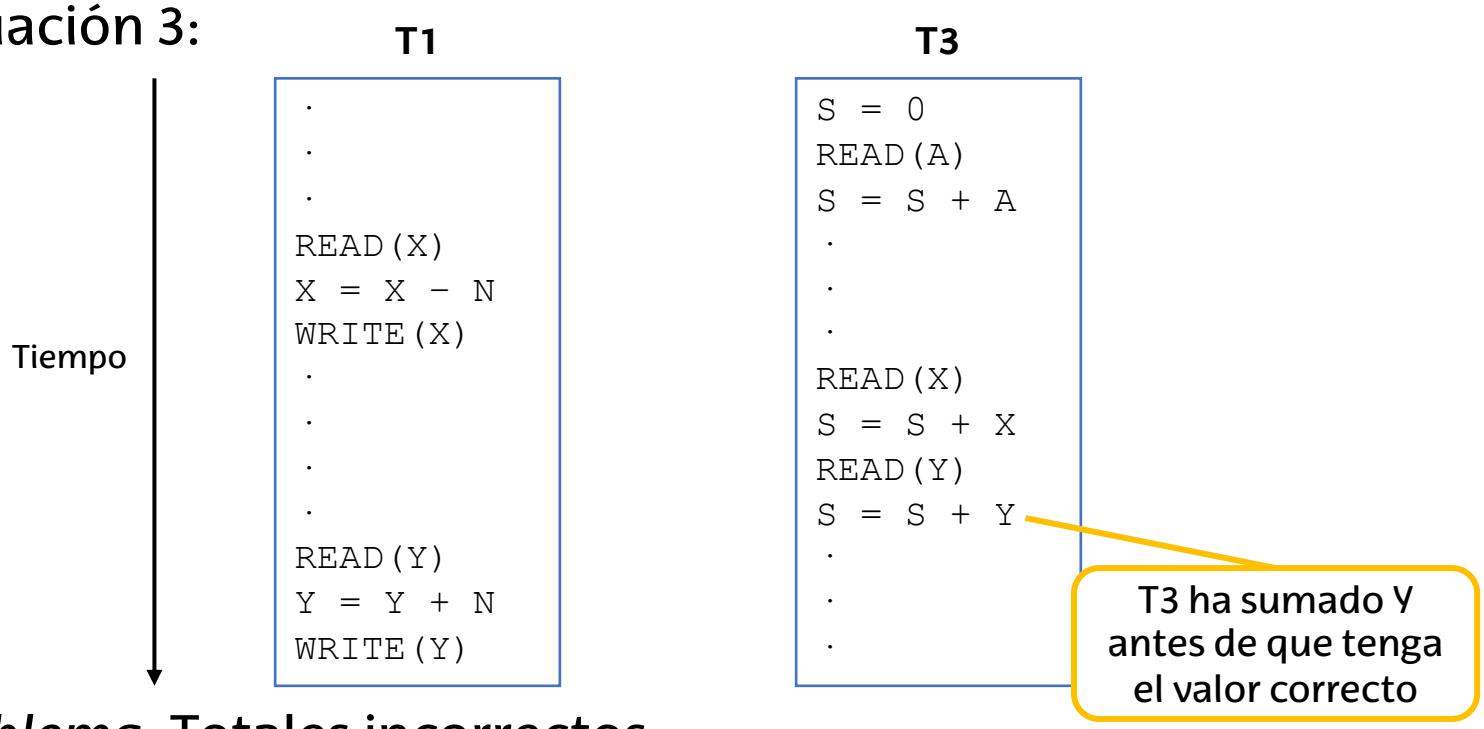
Problemas de concurrencia

- *Suponiendo que ambas transacciones se ejecutan de forma concurrente:*
- Situación 3:



Problemas de concurrencia

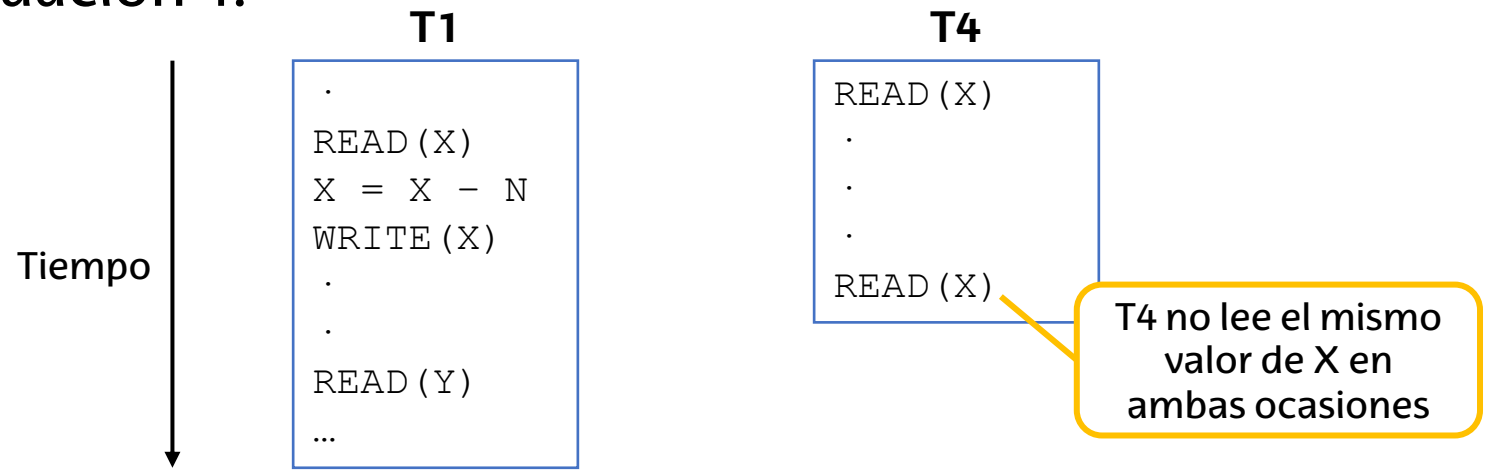
- Situación 3:



- *Problema*: Totales incorrectos
  - También llamado "lectura de valores fantasma"
  - En T3, el valor total de S no incluye N.

Problemas de concurrencia

- *Suponiendo que ambas transacciones se ejecutan de forma concurrente:*
- Situación 4:



- Problema: lectura inconsistente / irrepetible.

Plan de transacciones

- *Definición*: Serie intercalada de operaciones de distintas transacciones que se ejecutan de forma concurrente.

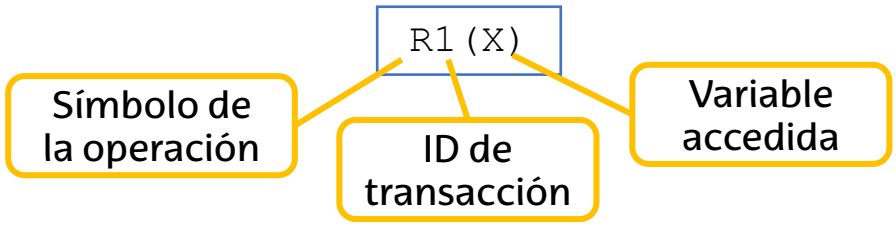
- Un plan P para un conjunto de transacciones ‹T1, T2,...,Tn›, es una serie en la que, para cada transacción Ti se cumple:
  - Las operaciones aparecen en P en el mismo orden que en Ti.
  - Las operaciones de otras transacciones pueden intercalarse con las de Ti.

Plan de transacciones

- En un plan, consideramos las siguientes operaciones:

Operación	Símbolo
READ	R
WRITE	W
COMMIT	C
ROLLBACK	B

- Cada operación se identifica de la siguiente forma:
  - *Ejemplo*: READ(X) en la Transacción T1



Plan de transacciones

- Ejemplo de plan:
  - Dada la ejecución:

T1

```
READ(X)
X = X - N
.
.
WRITE(X)
READ(Y)
.
Y = Y + N
WRITE(Y)
```

T2

```
.
.
READ(X)
X = X + 1
.
.
WRITE(X)
.
.
```

  - El plan es:

```
R1(X); R2(X); W1(X); R1(Y); W2(X) W1(Y);
```

Plan de transacciones: Conflictos

- En un plan, puede haber conflictos: situaciones que pueden derivar en problemas de concurrencia.

- Dos operaciones de un plan están en conflicto si:
  - Las operaciones pertenecen a transacciones diferentes.
  - Acceden a la misma variable/elemento X.
  - Al menos una de las operaciones es *write(X)*.

Plan de transacciones: Conflictos

- Ejemplo:
  - Dado el plan:

```
R1(X); R2(X); W1(X); R1(Y); W2(X); C2; W1(Y); C1;
```

  - Operaciones en conflicto:

```
R1(X); R2(X); W1(X); R1(Y); W2(X); C2; W1(Y); C1;
```

```
R1(X); R2(X); W1(X); R1(Y); W2(X); C2; W1(Y); C1;
```

```
R1(X); R2(X); W1(X); R1(Y); W2(X); C2; W1(Y); C1;
```

  - Operaciones sin conflicto:

```
R1(X); R2(X); W1(X); R1(Y); W2(X); C2; W1(Y); C1;
```

```
R1(X); R2(X); W1(X); R1(Y); W2(X); C2; W1(Y); C1;
```

Plan de transacciones: Conflictos

- Al permitirse intercalar operaciones de varias transacciones, existen muchos órdenes posibles.
  - Alto riesgo de conflictos.

- Una opción es utilizar planificaciones serie:
  - Plan donde las operaciones de cada transacción se ejecutan consecutivamente.
  - Las operaciones de las transacciones no se intercalan.
  - Libres de conflictos.

Planificación en serie

- Ejemplo:

Planificación en serie:

T1	T2
READ(X) X = X − N WRITE(X) READ(Y) Y = Y + N WRITE(Y) · · ·	· · · · · · READ(X) X = X + 1 WRITE(X)

Planificación no serie:

T1	T2
READ(X) X = X − N · · WRITE(X) READ(Y) · Y = Y + N WRITE(Y)	· · READ(X) X = X + 1 · · WRITE(X) · ·

Planificación en serie

- Toda planificación serie es correcta.
  - Se preserva la consistencia.
- *Sin embargo:*
- Una planificación serie no aprovecha la posibilidad de concurrencia en un sistema moderno.
  - P.e. si una transacción tiene que esperar por una lectura/escritura en disco, no se pueden ejecutar otras operaciones.
  - El resultado el poco eficiente.

Planificación serializable

- Una planificación es serializable si, para las mismas transacciones, produce un *resultado equivalente* a una planificación en serie.
  - Si una planificación "no serie" es serializable, su resultado será correcto.

- Dos planificaciones producen un *resultado equivalente* si:
  - Producen el mismo estado final en la BD.
  - Las operaciones sobre cada dato se aplican en el mismo orden en ambos planes.

Planificación serializable

- Ejemplo para las mismas operaciones:
  - Plan en serie P1:

```
R1(X); W1(X); R1(Y); W1(Y); R2(X); W2(X);
```

  - Plan en serie P2:

```
R2(X); W2(X); R1(X); W1(X); R1(Y); W1(Y);
```

  - Plan no serializable:

```
R1(X); R2(X); W1(X); R1(Y); W2(X); W1(Y);
```

  - Plan serializable:

```
R1(X); W1(X); R2(X); W2(X); R1(Y); W1(Y);
```

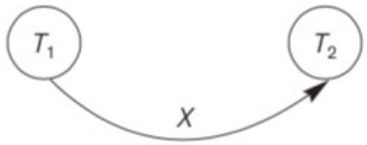
Planificación serializable

- Se puede verificar si una planificación es serializable utilizando un grafo de precedencias y el siguiente algoritmo:

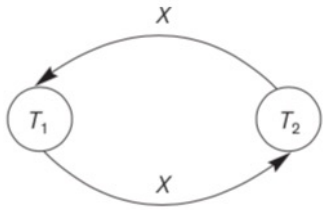
1) Crear un nodo por cada transacción Ti del plan.
2) Añadir un arco de la transacción Ti a Tj si existe un par de operaciones Oi (de Ti) y Oj (de Tj) que:
   - Oi y Oj están en conflicto.
   - Oi aparece antes que Oj en el plan.

Planificación serializable

- Si no hay ciclos en el grafo, el plan es serializable
  - Es posible crear un plan en serie equivalente reordenando las operaciones de las transacciones.
  - Ejemplo:
    - El plan serie equivalente es ‹T1, T2›



- Si hay un ciclo en el grafo, el plan **no** es serializable
  - No hay un orden serie que permita satisfacer las dependencias.
  - Ejemplo:



Planificación serializable

- Comprobar si un plan es serializable a través de grafos es correcto.
  - Pero no realista.
- En un sistema real, nuevas transacciones se crean constantemente
  - No es viable hacer comprobaciones estáticas con grafos.
  - No es fácil definir cuando empieza y acaba un plan.
- En un sistema real, se utilizan protocolos de serialización.

Protocolos de serialización

- Conjuntos de reglas basados en la teoría de la serialización.
  - El plan de transacciones es serializable si todas las transacciones cumplen las reglas.

- 4 tipos de protocolos:
  - Basados en reservas.
  - Basados en marcas de tiempo.
  - De concurrencia multiversión.
  - Optimistas.

Protocolos basados en reservas

- La serialización se asegura a través de reservas.
  - También llamados candados o bloqueos (“*locks*”).
- Un *lock* permite reservar el uso de una variable/elemento para una transacción concreta.
  - El *lock* es gestionado por el SGBD.
- Como protocolo, una transacción:
  - Antes de acceder a un elemento, hace una petición de reserva.
  - Si el elemento está reservado por otra transacción, espera hasta que sea liberado.

Protocolos basados en reservas

- *Lock* binarios
  - Controlan el acceso exclusivo a una variable
  - 2 estados posibles: bloqueado (1) o desbloqueado (0)
  - Comandos:
    - Reservar un elemento X:

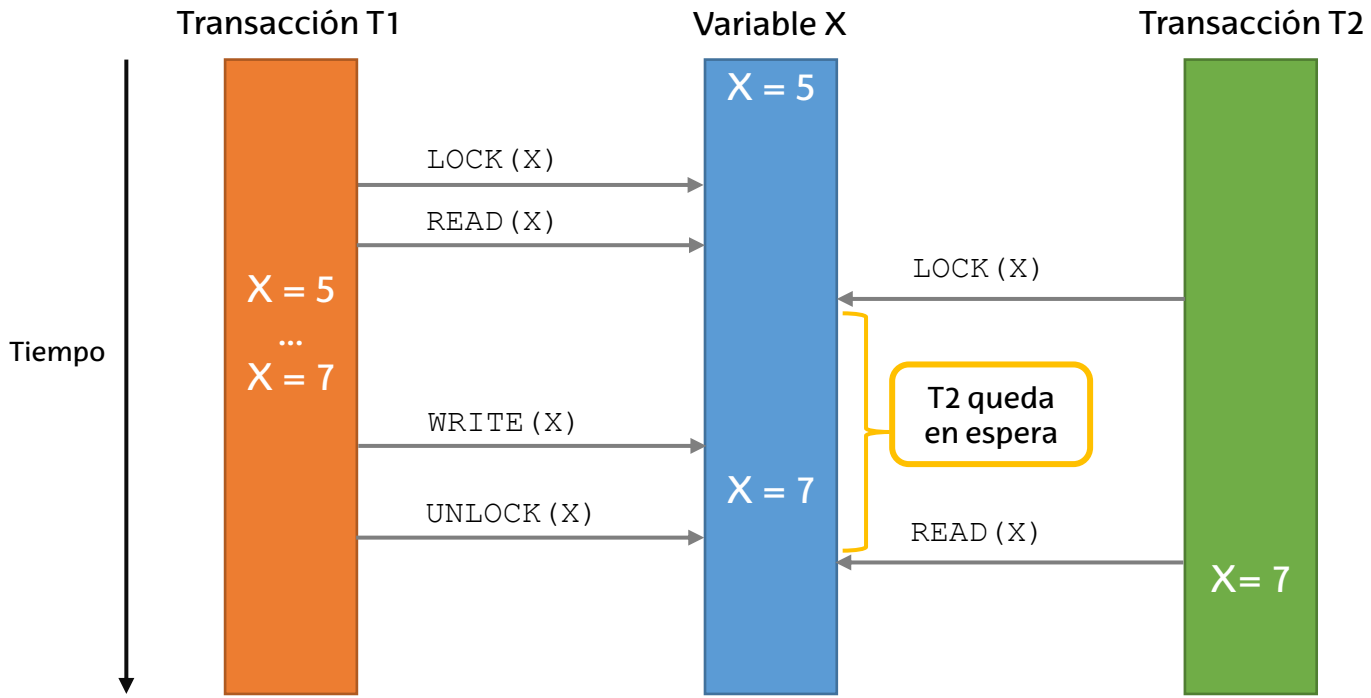
```
LOCK(X)
```

    - Liberar un elemento X:

```
UNLOCK(X)
```

Protocolos basados en reservas

- *Lock* binarios
  - Ejemplo:

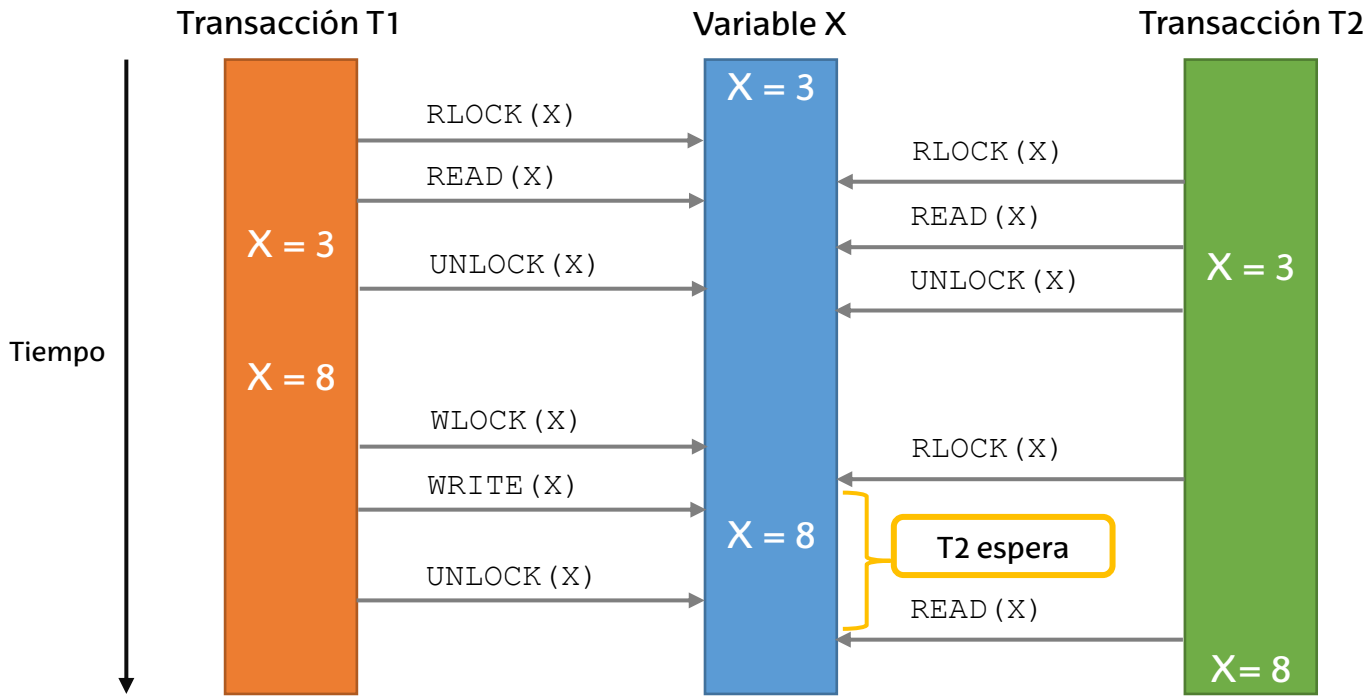


Protocolos basados en reservas

- *Lock* ternarios
  - Permiten el acceso concurrente a un elemento X si todas las transacciones sólo quieren leerlo.
  - Si alguna transacción quiere modificar X, el acceso es exclusivo.
  - 3 estados posibles:
    - Reservado para lectura (“Shared”)
    - Reservado para escritura (“Exclusive”)
    - Liberado o disponible
  - 3 comandos
    - Reservar un elemento X en modo lectura: `RLOCK(X)`
    - Reservar un elemento X en modo escritura: `WLOCK(X)`
    - Liberar un elemento X: `UNLOCK(X)`

Protocolos basados en reservas

- *Lock* ternarios
  - Ejemplo:



Protocolos basados en reservas

- Uso de *Lock* ternarios (I)
  - Siendo T una transacción y X un elemento a leer/escribir.

1) T debe reservar X usando RLOCK(X) o WLOCK(X) antes de hacer cualquier operación READ(X).
2) T debe reservar X usando WLOCK(X) antes de realizar cualquier operación WRITE(X).
3) T debe liberar X usando UNLOCK(X) después de haber completado todas las operaciones READ(X) y WRITE(X).

Protocolos basados en reservas

- Uso de *Lock* ternarios (II)
  - Siendo T una transacción y X un elemento a leer/escribir.

4) T no reservará X con RLOCK(X) si ya lo tiene reservado.
   - Excepción: si T tiene X reservado en modo escritura/exclusivo puede convertir la reserva en compartida con RLOCK(X).
5) T no reservará X con WLOCK(X) si ya lo tiene reservado.
   - Excepción: si T tiene X reservado en modo lectura/compartido puede convertir la reserva en exclusiva con WLOCK(X).
6) T no liberará X a menos que lo tenga reservado.

Locks en MySQL

- Hay 2 formas de gestionar *locks* en las transacciones de forma manual.

- A nivel de fila
  - Forma aconsejada.

- A nivel de tabla
  - Forma desaconsejada.

Locks en MySQL: Nivel de fila

- Se utiliza la operación *Select* para marcar las filas a bloquear.
  - Las filas se desbloquean cuando la transacción hace *Commit*.

- *Lock* compartido para lectura:

```
SELECT ... FROM ... FOR SHARE
```

- *Lock* exclusivo para escritura:

```
SELECT ... FROM ... FOR UPDATE
```

[1]Locks Set by Different SQL Statements in InnoDB: https://dev.mysql.com/doc/refman/8.0/en/innodb-locks-set.html

Locks en MySQL: Nivel de tabla

- Aplicar un cerrojo a una tabla[1]:

```
LOCK TABLES <tabla> <modo>
```

  - donde ‹modo›: READ (lectura) o WRITE (escritura)

- Liberar un cerrojo de una tabla:

```
UNLOCK TABLES <tabla>
```

- No se recomienda usar estos cerrojos con InnoDB[2].
  - No proveen mejor aislamiento y empeoran el rendimiento.

[1]LOCK TABLES and UNLOCK TABLES Statements: https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html
[2]Table Locking Issues: https://dev.mysql.com/doc/refman/8.0/en/table-locking.html

Ejercicio 1

- *Realizar este ejercicio en parejas, seréis A y B.*
- Usando MySQL, A crea una BD llamada *GestionVuelos*, que contiene una tabla *Vuelos* con los siguientes datos:

id (int)	descripcion (text)	pasajeros (int)
1	BIO-BCN	40
2	BIO-MAD	50
3	EAS-BCN	20
4	MAD-VIT	30

- *Continúa en la siguiente diapositiva*

Ejercicio 1

- A crea un procedimiento almacenado llamado *TransferReservas(N)*, el cual:
  - Reduce en N el nº de pasajeros del vuelo con id=1
  - Incrementa en N el nº de pasajeros del vuelo con id=2

- En el servidor de A, B crea un procedimiento almacenado llamado *Incrementar(N)*, el cual:
  - Incrementa en N el nº de pasajeros del vuelo con id=2.

- *Ambos procedimientos deben usar una transacción y cerrojos para hacer cambios sobre Vuelos.*
- A y B ejecutan los procedimientos a la vez.
  - Verificar que los pasajeros de la tabla Vuelos son correctos.

Protocolos basados en reservas: 2PL

- El uso de reservas en transacciones no garantiza la serialización de los planes.
- Es necesario seguir un protocolo que indique dónde colocar las operaciones de reserva y liberación dentro de las transacciones.
- *Two-Phase-Locking* (2PL) es el protocolo más conocido para emitir operaciones reserva y liberación en transacciones.

Protocolos basados en reservas: 2PL

- Dentro de toda transacción, se distinguen 2 fases

1) Fase de expansión (o crecimiento)
   - Se pueden realizar (o promover) reservas
   - No se pueden liberar reservas

2) Fase de contracción
   - Se pueden liberar reservas
   - No se pueden realizar reservas

Protocolos basados en reservas: 2PL

- Ejemplo:

Escenario A

T1	T2
RLOCK(Y)	RLOCK(X)
READ(Y)	READ(X)
RLOCK(X)	RLOCK(Y)
READ(X)	READ(Y)
X = X + Y	Y = Y + X
WLOCK(X)	WLOCK(Y)
UNLOCK(Y)	UNLOCK(X)
WRITE(X)	WRITE(Y)
UNLOCK(X)	UNLOCK(Y)

Fase de expansión

Fase de contracción

Las transacciones siguen 2PL.
Su plan es serializable.

Escenario B

T1	T2
RLOCK(Y)	RLOCK(X)
READ(Y)	READ(X)
RLOCK(X)	RLOCK(Y)
READ(X)	READ(Y)
X = X + Y	Y = Y + X
UNLOCK(Y)	UNLOCK(X)
WLOCK(X)	WLOCK(Y)
WRITE(X)	WRITE(Y)
UNLOCK(X)	UNLOCK(Y)

Se hace WLOCK después de UNLOCK

Las transacciones no siguen 2PL.

Protocolos basados en reservas: 2PL

- Ventajas de 2PL:
  - Se puede demostrar que si toda transacción de un plan sigue 2PL, el plan es serializable.
    - Ya no es necesario comprobar la serialización de los planes.
    - El SGBD gestiona las reservas y liberaciones.

- Inconvenientes de 2PL:
  - Puede limitar el grado de concurrencia de un plan.
    - Hay un sobrecoste por utilizar reservas en todo momento.
  - Puede generar otros problemas de concurrencia:
    - Bloqueo mortal (*deadlock*).
    - Espera indefinida (*starvation*).

Protocolos basados en reservas: 2PL

- Variante I: 2PL conservador / estático
  - Cada transacción debe reservar todos los elementos a los que quiere acceder antes de comenzar a ejecutarse.
  - Si no es posible reservar algún elemento, no se reservará ninguno y esperará a reintentar más tarde.
  - Esta variante está libre de bloqueo mortal (deadlock)

Protocolos basados en reservas: 2PL

- Variante II: 2PL estricto
  - Una transacción T no libera ninguna reserva de escritura hasta finalizar.
  - Ninguna otra transacción lee o escribe un elemento modificado por T, salvo si T se ha completado.
  - No está libre de bloqueo mortal (deadlock).
- Variante III: 2PL riguroso
  - Una transacción T no libera ninguna reserva hasta que finaliza.
  - Es más restrictivo que 2PL estricto.

Ejercicio 2

- Utilizando reservas ternarias, propón diferentes implementaciones de la siguiente transacción para que:
  - No cumpla 2PL
  - Cumpla 2PL normal
  - Cumpla 2PL estático
  - Cumpla 2PL estricto
  - Cumpla 2PL riguroso

Transacción T1

```
READ(X)
READ(W)
X = X + W
WRITE(X)
READ(Z)
W = W + Z
WRITE(W)
COMMIT
```

Ejercicio 3

- Utilizando reservas ternarias, propón diferentes implementaciones del siguiente procedimiento para que:
  - No cumpla 2PL
  - Cumpla 2PL normal
  - Cumpla 2PL estático

Procedimiento A

```
X = Y + 1
Y = Z + 1
Z = X + 1
```

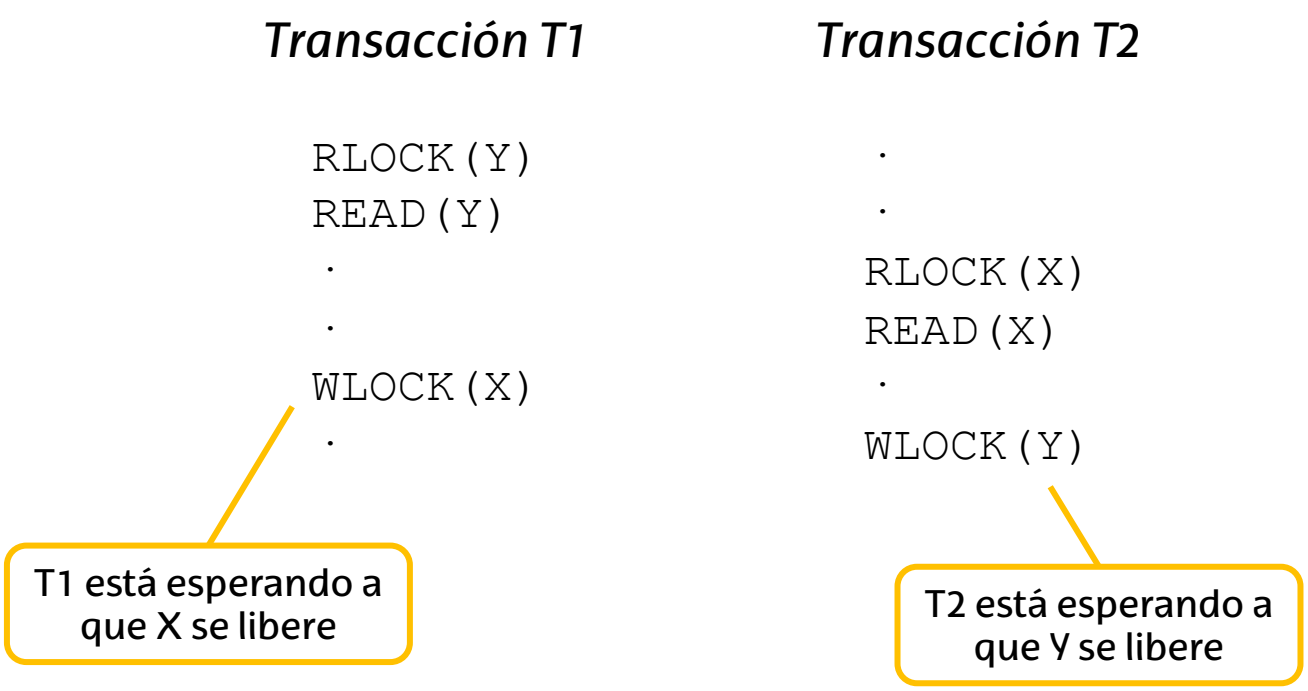
Deadlock

- El bloqueo mortal / interbloqueo / *deadlock* es el bloqueo permanente de varios procesos que compiten por los mismos recursos.

- En un SGDB, puede suceder cuando varias transacciones esperan a que se liberen recursos entre sí.

Deadlock

- Ejemplo:



Deadlock

- Condiciones necesarias para que suceda un *deadlock*:

1) Exclusión mutua:
   - Se impide que una transacción use un recurso reservado por otra.
2) Reserva parcial:
   - Se permite que se hagan reservas simultaneas sobre diferentes partes de un mismo recurso.
3) No expropiación:
   - El sistema no puede retirar las reservas concedidas.
4) Espera circular:
   - T1 espera a T2, T2 a T3, ..., TN espera a T1.

Deadlock

- Si una transacción está implicada en un posible deadlock, puede:
  - Esperar
  - Abortar
  - Hacer que otra transacción aborte
- En un SGBD, qué hacer está determinado por el protocolo de control de concurrencia.

Deadlock: Protocolos

- En este tema veremos los siguientes protocolos:

1) Reservado por adelantado
2) Marcas de tiempo de transacción
3) Uso de *timeouts*
4) Marcas de tiempo
5) Optimistas

Reserva por adelantado

- Empleado por 2PL conservador.
- Toda transacción reserva por adelantado todos los recursos que quiere leer o escribir.
- Si no puede reservarlos todos, se cancela y reintenta más tarde.
- Concurrencia muy limitada.

Marcas de tiempo de transacción

- Cada transacción T tiene un identificador único t(T), asignado según el instante de comienzo.
  - Las transacciones se numeran de forma cronológica.
  - La transacción más antigua tiene el menor t(T).
  - La transacción más reciente tiene el mayor t(T).

- Se considera:
  - T_i es una transacción que puede generar un deadlock al reservar.
  - T_j es la transacción que tiene reservada la variable que quiere T_i.

- Si T_i intenta reservar X, pero X ya está reservado por T_j, se pueden aplicar 2 esquemas para resolver el posible deadlock.
  - Esperar – Morir.
  - Herir – Esperar.

Marcas de tiempo de transacción

- Esquema Esperar – Morir.
  - Se aplica el siguiente algoritmo.

```
IF t(Ti) < t(Tj)
  Ti espera
ELSE
  Ti aborta y reinicia con la misma marca de tiempo
```

  - Una T_i más antigua espera a que una T_j más nueva termine
  - Una T_i más joven que intente reservar un elemento reservado por un T_j más antigua es abortada y reiniciada.

Marcas de tiempo de transacción

- Esquema Herir – Esperar.
  - Se aplica el siguiente algoritmo.

```
IF t(Ti) < t(Tj)
  Tj es abortada y reinicia con la misma marca de tiempo
ELSE
  Ti espera
```

  - Una T_i más antigua que solicite un elemento reservado por una T_j más joven, provoca que T_j aborte y sea reiniciada.
  - Una T_i más joven espera a que una T_j más antigua termine.

Uso de *timeouts*

- El SGBD puede definir un tiempo máximo de espera por una reserva (llamado *timeout*).
- Si una transacción espera un tiempo superior al definido, el SGBD asume que existe un *deadlock* y la aborta.
  - No comprueba que haya *deadlock* real.
  - Alternativa a usar un esquema tipo Esperar-Morir/Herir-esperar.
- En algunos SGDBs, se usa en combinación con otras técnicas.

Protocolos

- Usar reservas/*locks* y 2PL se puede considerar un enfoque pesimista de gestión de concurrencia.
  - Se asume que van a suceder problemas constantemente.
  - Se abortan transacciones que podrían provocar *deadlocks*, pero tal cosa podría no ocurrir nunca.

Protocolo: marcas de tiempo

- Protocolo que permite asegurar el cumplimiento de la propiedad *Isolation* de ACID sin usar reservas/*locks*.

- Cada transacción T_i tiene asociada una marca de tiempo $t(T_i)$, asignada al comienzo de su ejecución.
  - Si T1 comenzó antes que T2, t(T1) ‹ t(T2).

- Cada elemento X tiene 2 marcas de tiempo:
  - TSREAD(X): marca de tiempo de lectura más reciente.
  - TSWRITE(X): marca de tiempo de escritura más reciente.

Protocolo: marcas de tiempo

- La planificación serie equivalente al usar este protocolo sería la resultante de ordenar las transacciones por su marca de tiempo.
  - Las operaciones de las transacciones debe seguir ese orden.
  - Ejemplo: t(T1) ‹ t(T2) ‹ t(T3).

- Cuando una transacción T_i accede al elemento X, $t(T_i)$ se compara con TSREAD(X) y TSWRITE(X) para verificar que cumple la planificación serie equivalente.
  - Si la operación se fuese a realizar en un orden incorrecto, aborta Ti.

Protocolo: marcas de tiempo

- **Situación:** T realiza `WRITE(X)`
  - Aplicar:

```
IF (TSREAD(X) > t(T)) or (TSWRITE(X) > t(T))
  T es abortada
ELSE
  WRITE(X)
  TSWRITE(X) = t(T)
```

  - Si una transacción más joven que T ha leído o escrito X antes de que T haya podido escribirlo: violación de la ordenación.
    - T debe abortar/hacer Rollback.

Protocolo: marcas de tiempo

- **Situación:** T realiza `READ(X)`
  - Aplicar:

```
IF (TSWRITE(X) > t(T))
  T es abortada
ELSE
  READ(X)
  TSREAD(X) = max(t(T), TSREAD(X))
```

  - Si una transacción más joven que T ha escrito el valor X antes de que T haya podido leerlo: violación de la ordenación.
    - T debe abortar/hacer Rollback.

Protocolos optimistas

- Protocolos que asumen pocos problemas de concurrencia.
  - Se basan en los protocolos en marcas de tiempo.
  - Mejor rendimiento, no interfieren en la ejecución de las transacciones.
- Durante la ejecución de la transacción:
  - No se realiza ninguna comprobación.
  - No se realiza ninguna modificación sobre la BD, sino sobre copias locales.
- Al finalizar la ejecución, se realizan las comprobaciones.
  - Si son correctas, se escriben los cambios en la BD.
  - Si no lo son, se aborta la transacción.

Control de concurrencia

- Hasta ahora, hemos considerado que las transacciones sólo leen y modifican objetos.

- En la realidad, las transacciones también pueden crear y eliminar objetos.
  - Esto genera problemas diferentes.

Control de concurrencia

- Suponiendo una BD con la siguiente tabla:

```
CREATE TABLE Usuarios(id int,
  nombre text, edad int);
```

Tiempo ↓

T1

```
START TRANSACTION;
.
SELECT COUNT(edad)
  FROM Usuarios
  WHERE nombre="Unai";
.

.
SELECT COUNT(edad)
  FROM Usuarios
  WHERE nombre="Unai";
COMMIT;
```

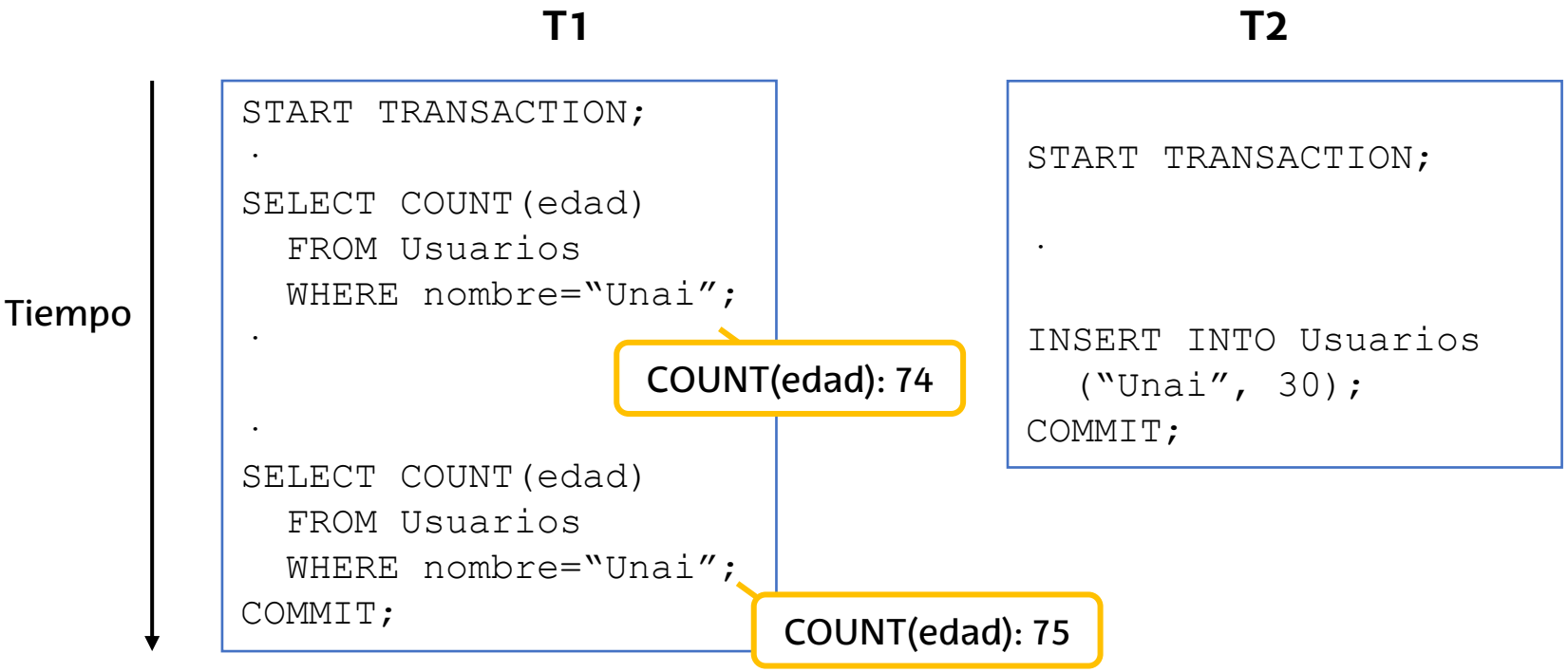
T2

```
START TRANSACTION;

.

INSERT INTO Usuarios
  ("Unai", 30);
COMMIT;
```

Control de concurrencia



- El valor de MAX(edad) es diferente en ambas ocasiones.

Lecturas fantasma

- La creación y borrado de elementos en una transacción puede crear lecturas fantasma en otras.

- **Motivo**: se pueden establecer reservas para elementos existentes, pero no para los no existentes aún.
  - *Nota*: aunque se pueden establecer reservas a nivel de tabla, bloquear una tabla completa puede degradar el rendimiento.

Lecturas fantasma

- Soluciones que el SGBD puede implementar:

1) Re-ejecutar lecturas completas

2) Reservas/*locks* de predicados

3) Reservas/*locks* sobre índices.

Lecturas fantasma

1) Re-ejecutar lecturas completas
   - El SGBD registra las cláusulas WHERE de todas las operaciones que la transacción ejecuta.
     - Y retiene el resultado.
   - Al hacer *Commit*, el SGDB ejecuta las porciones WHERE de todas las operaciones y verifica que coinciden con lo registrado anteriormente.
   - *Problema*: Degradación de rendimiento al hacer *Commit*.
     - Especialmente, si se usan tablas grandes que están en disco.

Lecturas fantasma

2) Reservas de predicados

- Consiste en hacer una reserva para la expresión que se define en la cláusula WHERE.
- La reserva se realiza sobre el espacio representado por la expresión, no sobre unas filas concretas.
- *Problema*: muy difícil de implementar.
  - Determinar si los predicados son compatibles es NP-completo.
- Más información:
  - https://www.geeksforgeeks.org/predicate-locking/

Lecturas fantasma

2) Reservas de predicados

- Ejemplo:
  - Una transacción T_x realiza:

```
SELECT nombre FROM Usuarios WHERE edad > 29;
```

    - Se concede una reserva sobre el espacio de la expresión *edad > 29*.

  - Si una transacción T_y intenta realizar:

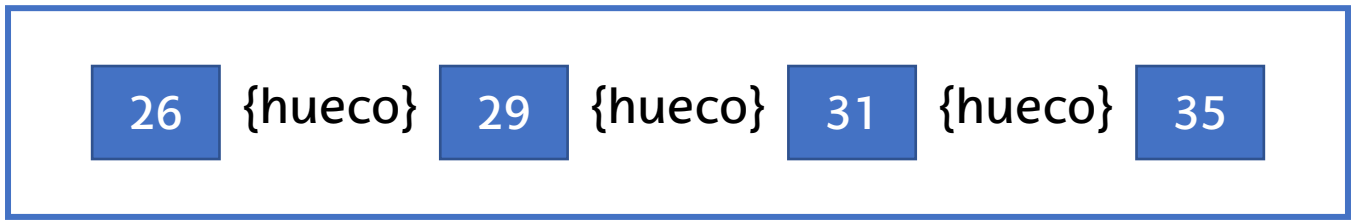
```
INSERT INTO Usuarios VALUES("Mikel",38);
```

    - Esta operación modificaría el espacio *edad > 29*, reservado por T_x.
    - T_y debe esperar hasta que T_x finalice (o lo que indique el protocolo en uso).

Lecturas fantasma

3) Reservas sobre índices

- Los índices contienen meta-información sobre el contenido de las tablas.
- Se pueden realizar reservas sobre índices para evitar su modificación.
  - Técnica usada por la mayoría de SGBDs.
- Ejemplo de índice:
  - *Hueco* representa la ausencia de valores entre elementos.



Lecturas fantasma

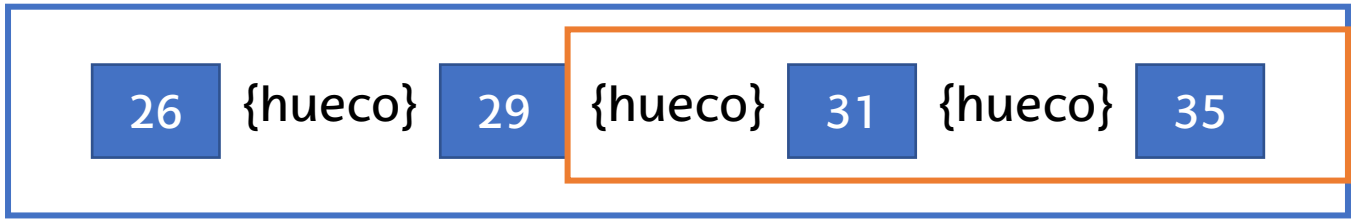
3) Reservas sobre índices

- Se pueden hacer reservas que ocupen un rango de huecos.
  - En inglés, *gap lock*.

- Ejemplo:
  - Para la consulta:

```
SELECT nombre FROM Usuarios WHERE edad > 29;
```

  - El SGBD puede reservar el rango ›29 en el índice:



Niveles de aislamiento

- El estándar SQL 1992 define diferentes niveles de aislamiento que un SGBD puede ofrecer.
- Permite ajustar cómo de estricto es el SGBD al asegurar que los planes son serializables.
  - 100% estricto asegura que no habrá problemas de concurrencia.
  - ... pero puede limitar el rendimiento del sistema.

Niveles de aislamiento

- Controla el grado de exposición de una transacción a las acciones de otras transacciones concurrentes.

- Permitir mayor concurrencia puede provocar que las transacciones estén expuestas a problemas:
  - Lecturas fantasma.
  - Lecturas irrepetibles.
  - Lecturas sucias.

Niveles de aislamiento

- En cada nivel de aislamiento pueden aparecer diferentes problemas de concurrencia:

	Lecturas sucias	Lecturas irrepetibles	Lecturas fantasma
Serializable	No	No	No
Repeatable Reads	No	No	Posible
Read Committed	No	Posible	Posible
Read Uncommitted	Posible	Posible	Posible

  - Debemos elegir el nivel adecuado para nuestro entorno.

Niveles de aislamiento

- Serializable
  - Equivalente a usar 2PL estático + riguroso, y usar reservas sobre índices.
- Repeatable Reads
  - Igual que *Serializable*, pero sin usar reservas sobre índices.
- Read Committed
  - Igual que *Repeatable Reads*, pero las reservas tipo RLOCK se liberan inmediatamente.
- Read Uncommitted
  - Igual que *Read Committed*, pero no se utilizan reservas tipo RLOCK.

Niveles de aislamiento

- Niveles por defecto en varios SGBDs[1]:

SGBD	Por defecto
CockroachDB	SERIALIZABLE
Microsoft SQL Server	READ COMMITTED
MySQL	REPEATABLE READS
Oracle	READ COMMITTED
PostgreSQL	READ COMMITTED

[1]Fuente: Andy Pavlo, CMU 15-445: https://15445.courses.cs.cmu.edu/fall2022/

Bibliografía

- R. Elmasri, S. B. Navathe. “Fundamentals of Database Systems”, 7th edition, Pearson, 2017.
- A. Silberschatz, H. F. Korth, S. Sudarshan. “Database Systems Concepts”, 7th edition, McGraw-Hill, 2019.
- A. Pavlo et al. “Intro to Database Systems”.
  - Carnegie Mellon University, 15-445/645, Fall 2022, https://15445.courses.cs.cmu.edu/fall2022/

- Sección “Motivación”:
  - Ver pies de página.