CONCURRENCIA Y TRANSFERENCIAS

La motivacion es pensar en que una base de datos real hay multiples usuarios realizando operaciones simultaneamente

Puedo tener sistemas:

- ***Monoprocesador*** donde se puede hacer multitasking eun solo core o procesador (empieza una luego de finalizar o pausar la anterior)
- ***Multiprocesador*** donde hay varias unidades de procesamiento y funcionan en simultaneo.
- ***Sistemas distribuidos*** donde cada nodo tiene su disco y memoria suelen REPLICARSE las tablas, copiandose todas o un fragmento de ellas en los diferentes nodos (puedo tener un 50% de una tabla en un nodo y el resto en otro nodo). La idea es que se minimice el tiempo de respuesta de las tareas o procesos

> **TRANSACCION** es una unidad logica de trabajo o secuencia ordenada de instrucciones atomicas

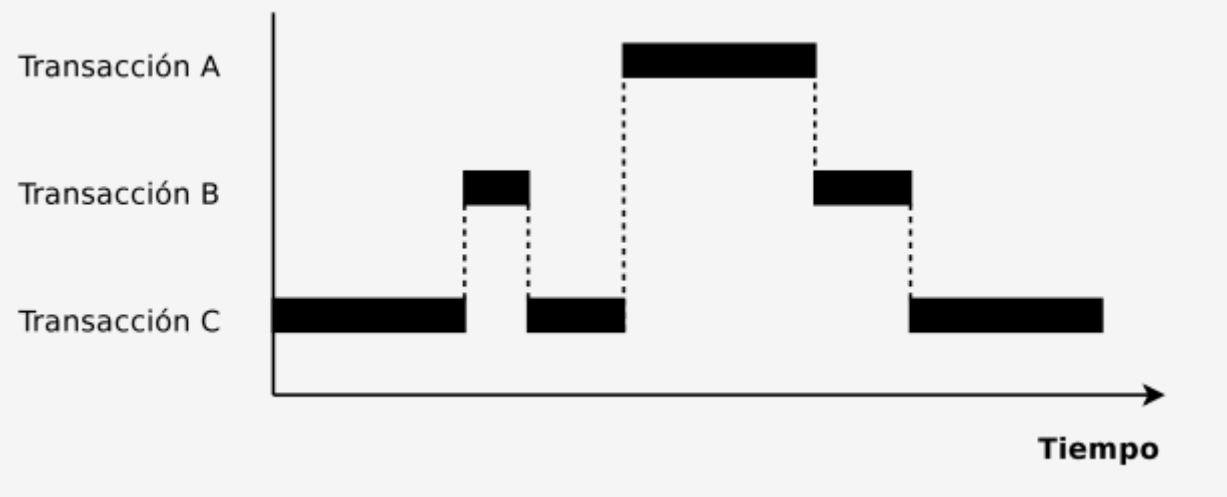
No quiero que las TRANSACCIONES sean interrumpidas. Debe ser ejecutada en su totalidad o no ser ejecutada.

Queremos evitar la serializacion de las tareas ya que es muy ineficiente en tiempo y recursos de hardware.

> La ***CONCURRENCIA*** es la posibilidad de ejecutar multiples transacciones en forma simultanea

El problema de la CONCURRENCIA es como gestionar los recursos compartidos.

Veremos el modelo de **CONCURRENCIA SOLAPADA** donde el procesador ejecuta un proceso a la vez.



Todo lo que veremos de CONCURRENCIA sera en un modelo con un unico procesador. Estas tecnicas se pueden extender a multipleprocesamiento

Voy a pensar en mi base de datos como un conjunto de **ITEMS** que puede ser de distinto tamaño. El ITEM sera mi unidad de dato. Puede ser una fila, el valor de un atributo, un JSON, etc..

Asi tambien voy a definir ***instrucciones atomicas***.
Mis instrucciones se van a basar en:

- **leer_item(X)**: Cargar en memoria el valor del item X.
- **escribir_item(X)**: Ordena escribir en la base de datos el valor que está en memoria del item X.

> **escribir_item(X)** no siempre escribe efectivamente en memoria. A veces puede ejecutar esta transaccion y dejar el valor en un buffer antes de llevarlo a disco.

El tamaño del item es la GRANULARIDAD y es fundamental en cuanto al control de la concurrencia. Cuanto mas chica o fina la GRANULARIDAD, mas voy a permitir la CONCURRENCIA

Las **TRANSACCIONES** deben ser ejecutadas en su totalidad o no ser ejecutadas, mas alla de la interferencia de otras transacciones que se procesen en simultaneo.

ACID son propiedades de las transacciones

> (A)tomicidad: La transaccion debe ser atomica. Ejecutada o no ejecutada. No hay puntos medios.

> (C)onsistencia: La base de datos debe estar en un estado consistente con las reglas de integridad al finalizar una transaccion. Si se cumplen las demas condiciones de ACID, la Consistencia tambien se cumplirá.

> (I)solation o aislamiento: El resultado de las transacciones realizadas concurrentemente tiene que ser el mismo que si las transacciones fueran hechas aisladamente una detras de otra. Solapamiento serial o serializable.

> (D)urabilidad: Las transacciones deben ser persistentes, independientes de las posibles fallas. EJ: Porque hubo un corte de luz, no se guardo el resultado.

Las ejecuciones hay que tenerlas en cuenta desde la perspectiva de la BASE DE DATOS y del USUARIO Una alta desde la perspectiva de la BDD es el mismo, pero desde la del USUARIO no

Para garantizar los ACID, los SGBD tienen ***MECANISMOS DE RECUPERACION*** que permiten rehacer o deshacer una transaccion por una falla. Esto ayuda con la *A* de ACID.

Es importante que el SGBD pueda deshacer una transaccion. Ej: El cliente se arrepiente de reservar un vuelo

Hay instrucciones especiales ---> BEGIN, COMMIT y ABORT que se aplican a cada transaccion

> **BEGIN** indica el comienzo de la transaccion.

> **COMMIT** indica que la transaccion se realizo. Toda instruccion commiteada tiene que ser durable (ES IRREVERSIBLE). De esta forma la informacion queda en un medio persistente.

> **ABORT** indica que se produjo algun error y que la transaccion queda deshecha. Un ABORT tiene que deshacer todas las operaciones intermedias (rollback).

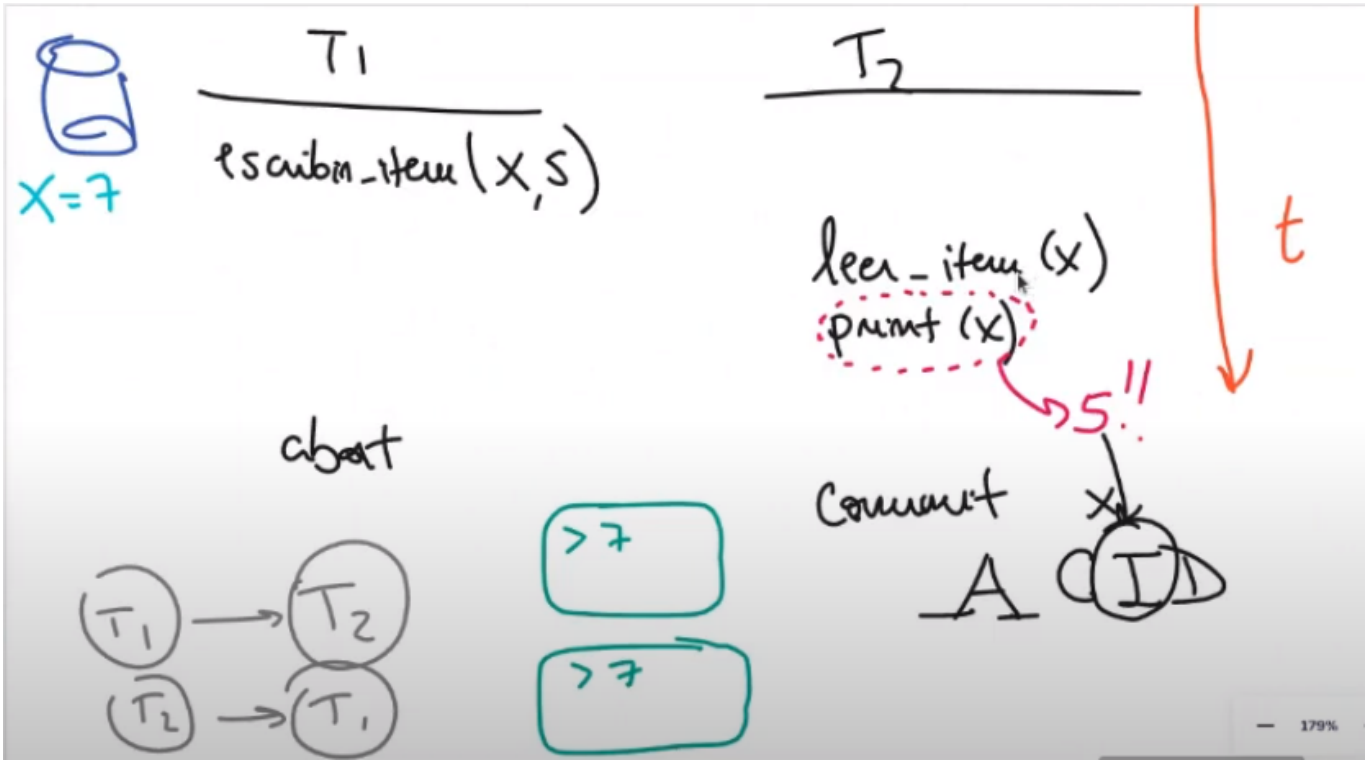
ANOMALIAS de la ejecucion concurrente.

Son situaciones que pueden violar las propiedades ACID

DIRTY READ (lectura sucia): La transaccion T2 lee un item que ha sido modificado por T1, pero T1 luego deshace la operacion (ABORT) y por lo tanto la lectura de T2 es invalida Se la conoce tambien como *"Read*

uncommited data". El resultado de esta lectura puede ser que caiga en un caso de transacciones no serializables, por eso es anomalo.

Fenomeno ---> Wt1(x) Rt2(x) ct2(x) at1---> quiere decir que T1 escribio (Write) un dato X y que T2 leyo (Read) ese dato. Luego T2 commitea, pero T1 aborta. VIOLACION DE Isolated de ACID



LOST UPDATE (actualizacion perdida) : Cuando una transaccion lee un item que fue leido antes por una transaccion que aun no termino. Es decir que entre las dos lecturas, va a haber un cambio de valores impercepitble por las transacciones.
Aqui, cuando la T1 vuelve a leer el item y se encuentra con el valor que puso T2, se llama ANOMALIA DE LECTURA NO REPETIBLE, porque esta leyendo dos veces el mismo item y tiene valores distintos (sin que T1 lo haya modificado) (https://www.youtube.com/watch?v=pVmqhLnZoSc&feature=youtu.be min 1:10

https://youtu.be/S1SyPDvIZv0?t=4964)
Fenomeno ---> Rt1(x)...Wt2(x)...(at1 o ct1)

. DIRTY WRITE : Analogo a la lectura sucia. Un item X es escrito por T1 y luego por T2, pero luego T1 aborta y deshace los cambios. Se pierden ambos valores: el de T1 y T2.

. PHANTOM (fantasma): Ocurre cuando una transaccion T1 observa un grupo de items que cumple una condicion (ej: SELECT * WHERE Table.a1 = x) pero luego otra transaccion T2 modificado esos items afectando la transaccion T1

GRAFO DE PRECEDENCIAS

Herramienta que sirve para la serializacion. Ahora tendremos los siguientes elementos:

- Rt(x): La transaccion T lee el item X
- Wt(x): La transaccion T escribe el item X
- bT : Comienza T

- cT : Se hace el commit de T
- aT : Se aborta T

> Un ***SOLAPAMIENTO*** es una lista de (mT1+mT2) instrucciones, donde mTi son las instrucciones de la transaccion i, y donde cada instruccion aparece una sola vez y respetan el orden original entre ellas dentro del solapamiento

> Una ejecucion **SERIAL** es aquella en que las transacciones se ejecutan por completo una detras de otra y donde se puede generar ordenes distintos de ellas Dadas N transacciones, puedo tener N! ejecuciones seriales

Un **SOLAPAMIENTO de N transacciones es SERIALIZABLE** cuando la ejecucion de sus instrucciones en dicho orden deja a la base de datos en un estado equivalente a otra ejecucion SERIAL cualquiera El estado hace referencias al valor de sus items (como quedan las filas, tablas, etc..)

Que los solapamientos sean serializables garantizan el aislamiento de ACID, por eso buscamos que los solapamientos sean de este tipo. Lo ideal es no realizar el solapamiento para ver si es serializable, sino tratar de verlo antes. Debemos considerar a la BDD desde cualquier estado inicial posible

Equivalencias: Como detectar que dos solapamientos serializados son equivalentes:

1. Equivalencias de resultados: Cuando los ordenes de ejecucion dejan a la BDD con el mismo estado, partiendo del mismo inicio

2. Equivalencias de conflictos: Cuando ambos ordenes de ejecucion poseen los mismos conflictos entre instrucciones. ES LA MAS FUERTE DE LAS TRES PORQUE NO DEPENDE DEL ESTADO INICIAL Un conflicto es un par de instrucciones I1 e I2 ejecutadas por distintas transacciones Ti, Tj tales que I2 se encuentra mas tarde que I1, y sucede algunas de las siguientes situaciones:

    - RTi(x), WTj(x) : Una transaccion escribe un item que otra leyo
    - WTi(x), RTj(x) : Una transaccion lee un item que otra escribio
    - WTi(x), WTj(x) : Dos transacciones escriben el mismo item En otras palabras, tenemos un conflicto cuando dos transacciones distintas ejecutan instrucciones sobre un mismo item X, y al menos una de las dos es una escritura (W(x))

> Dos instrucciones de un solapamiento que no constituyen un conflicto pueden ser invertidas en su ejecucion obteniendo un solapamiento equivalente por conflictos.

3. Equivalencias de vistas: Aca se pide que no solo los estados sean iguales, sino que los usuarios vean lo mismo en un momento determinado.

> ***ALGO IMPORTANTE ES NO SWAPEAR INSTRUCCIONES DE UNA MISMA TRANSACCION YA QUE ESO ROMPE EL CRITERIO DE TRANSACCION***

- **GRAFO DE PRECEDENCIAS** se forma con las transacciones T1...Tn como nodos y las aristas van a ser los conflictos entre ellas, donde los conflictos pueden ser unos de los 3 que vimos antes (RW, WR o WW). *Cada arco en el grafo va a representar una precedencia*. Si no hay conflictos, el grafo queda simplemente representado por sus nodos.

> TEOREMA: Un orden de ejecucion es serializable por conflictos si y solo si su grafo de PRECEDENCIAS no tiene ciclos

- ALGORITMO PARA VER SI UN GRAFO TIENE CICLOS (https://youtu.be/S1SyPDvIZv0?t=8554)

1. OPCION 1: Ver cuales de sus nodos no tiene ningun arco entrante. Eliminar el nodo en caso de que exista y buscar otra vez. Si todos tienen grado entrante, entonces hay un ciclo
2. OPCION 2: Lo mismo, pero viendo el grado de salida El orden en que los nodos fueron eliminados determinara el orden topologico que a su vez determinara el orden de ejecucion serial por conflictos

---

CONTROL DE CONCURRENCIAS

Busca garantizar el aislamiento (La I de ACID). Puedo tener dos tipos de enfoques:

Enfoque	**Tipo de control de concurrencia**
Pesimista: Se que van a haber conflictos y quiero garantizar que no van a ocurrir.	Control basado en locks
Optimista: Dejo hacer a las transacciones y en caso de haber un conflicto, hago un rollback de alguna de ellas. Esto sirve cuando se que la probabilidad de conflicto es baja	Control basado en timestamps / Snapshot Isolation

BASADO EN LOCKS (enfoque pesimista): Consiste en bloquear los recursos que vamos a utilizar (lock) y no permitir que se use mas de una vez simultaneamente por las transacciones. Estos locks se insertan como instrucciones especiales del SGBD.

Un lock es una variable que permite regular el acceso a los recursos.

Primitivas: LOCK(X) y UNLOCK(X)

Resuelve la exclusion mutua. El lock es bloqueante.

Es fundamental que las Primitivas sean ATOMICAS, es decir, que ninguna otra transaccion debe tener solapada una ejecucion semejante

Puede haber locks de escritura (de acceso exclusivo) y de lectura (de acceso compartido). De acceso exclusivo son los locks genericos en los cuales ninguna transaccion puede acceder al recurso lockeado por otra Pero de acceso compartido implica que muchas transacciones puedan poseer locks sobre un mismo item simultanea

```
PROTOCOLO DE 2PL --> Una transaccion no puede tomar un lock despues de
liberar uno que habia adquirido.
```

La transaccion se divide en dos fases:

- la etapa de adquisicion de locks: donde la cantidad de locks crece
- la de liberacion: donde la cantidad de locks decrece

> El cumplimiento de 2PL garantiza que la ejecucion de transacciones es serializable.

Tener cuidado con los deadlocks (cuando una transaccion lockea un item y no lo libera) y livelocks ()

Mecanismos de prevencion de deadlocks
Que la transaccion obtenga todos los locks que vaya a necesitar al principio
Definir un ordenamiento de los recursos
Analizar el grafo de alocacion de recursos
Definir un timeout para la adquisicion del lock(x)

NIVELES DE AISLAMIENTO

No siempre voy a querer el mismo nivel de aislamiento, por ejemplo no voy a querer que siempre sea serializable porque no me interesa. Pueden ser cuatro niveles segun el estandar: - READ UNCOMMITED: No hay aislamiento. No hay locks y se acceden a los items sin precauciones - READ COMMITED: Evita la lectura sucia - REPETEABLE READ: Evita la lectura sucia y la lectura no repetible - SERIALIZABLE: Evita todas las anomalias y asegura que el resultado de las transacciones sea equivalente a alguna ejecucion con orden serial

---

CONTROL DE CONCURRENCIAS (cont.)

BASADO EN TIMESTAMPS Se asigna a cada transaccion Ti un timestamp TS(Ti). Los *timestamps* deben ser unicos y determinaran el orden serial del solapamiento.

Ej: TS(Ti) = 52 TS(Tj) = 76

Se ejecutara entonces Ti -> Tj.

Pueden aparecer conflictos pero siempre que respeten el orden de los timestamps.
Si una transaccion se ejecuta en un orden erroneo de acuerdo a su timestamp, se abortara y se hará un rollback de dicha transaccion.

En todo momento se tiene la siguiente informacion:

- ***read_TS(X)*** : Es el TS(T) correspondiente a la transaccion mas joven (de mayor TS(T)) que leyo el item X

- ***write_TS(X)*** : Es el TS(T) correspondiente a la transaccion mas joven (de mayor TS(T)) que escribio el item X

- Lógica de funcionamiento:
  1. Cuando una transacción T_i quiere ejecutar un $R(X)$:
     - Si una transacción posterior T_j modificó el ítem, T_i deberá ser abortada *(read too late)*.
     - De lo contrario, actualiza `read_TS(X)` y lee.
  2. Cuando una transacción T_i quiere ejecutar un $W(X)$:
     - Si una transacción posterior T_j leyó ó escribió el ítem, T_i deberá ser abortada *(write too late)*.
     - De lo contrario, actualiza `write_TS(X)` y escribe.
- Observación: La ejecución de los $R(X)$ y $W(X)$, y actualización de los `read_TS(X)` y `write_TS(X)` se centraliza en el *scheduler*.

EJ:

> R58(X) --> W56(X)

HAY QUE ABORTAR W56(X) YA QUE ALGUIEN POSTERIOR LEYÓ EL MISMO ITEM

> **REGLA GENERAL: Abortar siempre que se lea un item con una transaccion de orden menor a la ultima lectura del mismo item**

Esto quiere decir que si mi T58 quiso escribir X y despues vino una T56 a querer escribir X tambien, puedo descartar T56 sin problema ya que no afecta a la serializacion

---

SNAPSHOT ISOLATION

Cada transaccion ve "una foto" de la BBDD commiteada al momento en que inició. Entonces cada transaccion, al momento de iniciar, sabe en que estado estan los datos de la BBDD.

Ventajas: Permite mayor solapamiento, ya que se puede realizar la mayoria de las lecturas.
Desventajas: Requiere mayor espacio en memoria o disco. Ademas, en conflictos WW siempre va a querer deshacer una.

Sigue la regla de *first-commited-win*: si dos transacciones quieren modificar el mismo item, gana la que realice el commit primero.

Esto, sumado a VALIDACION PERMANENTE DEL GRAFO DE PRECEDENCIAS Y LOCKS DE PREDICADOS garantizan la seriablidad.

Solución a la anomalía del fantasma

1. Bajo control de concurrencia basado en *locks*:
   - *Locks* de tablas: cuando se produce un $R(\{X|cond\})$, bloquear el acceso a toda la tabla a la que X pertenece. Es una solución casi trivial, y poco eficiente.
   - *Locks* de predicados: bloquear todas aquellas tuplas que podrían cumplir la condición, evitando incluso futuras inserciones de tuplas que también la cumplan.
2. Bajo *Snapshot Isolation*
   - No puede ocurrir, porque la transacción ve la misma snapshot a lo largo de su vida.
3. Bajo control de concurrencia basado en *timestamps*:
   - Utilizar índices de tipo árbol, y mantener registros `read_TS(I)` y `write_TS(I)` también para los nodos del árbol.