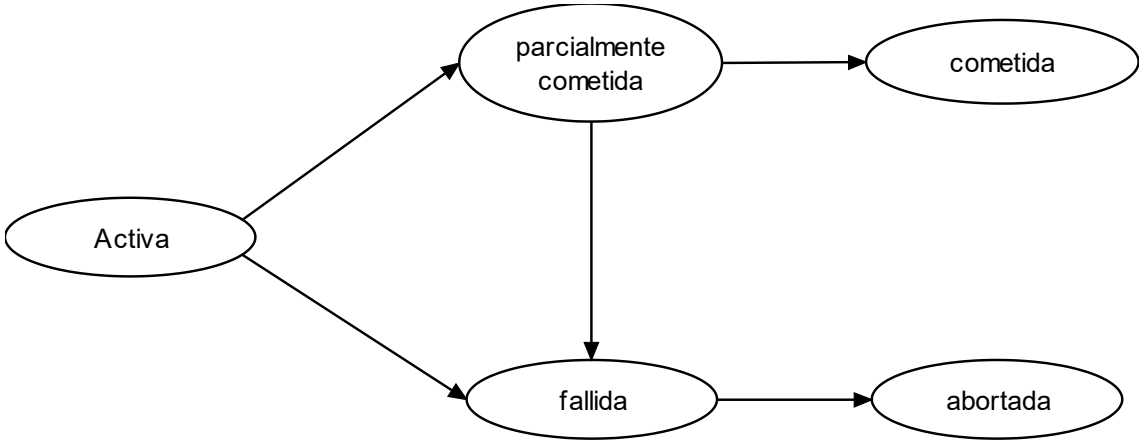
Universidad Nacional de la Patagonia San Juan Bosco
Facultad de Ingeniería

Cátedra: **Bases de datos I**

Administración de Transacciones



- Administración de Transacciones

  - Se denomina **administración de transacciones**, a dos temas muy relacionados entre sí:

  - ✓ **Recuperación de una BD:** significa que el SGBD debe restablecer el estado de la BD a un estado correcto después de que alguna falla haya ocasionado que el estado actual sea inconsistente.
  - ✓ **Concurrencia:** se refiere al hecho de que los SGBD permiten que varias transacciones accedan a una misma BD a la vez.

- Transacción: Definición

  - **Una transacción es una unidad de trabajo lógica**, que accede y posiblemente actualiza varios elementos de datos. Está delimitada por declaraciones de la forma **begin transaction** y **end transaction**. La transacción consiste en todas las operaciones que se ejecutan entre el inicio y el final de la transacción. Dicho de otra forma, **una transacción es una secuencia de operaciones sobre una BD, mediante la cual, un estado consistente de la BD se transforma en otro estado consistente**.

- Las transacciones son conjuntos de instrucciones → como **unidades atómicas.** No son descomponibles en las instrucciones individuales de las que están formadas.
- Gracias a esta atomicidad, las transacciones permiten que se ejecuten operaciones complejas en la BD, **manteniendo la integridad**.
- Una transacción se ejecuta con éxito si y sólo si todas las operaciones que la componen terminan con éxito. Si una de las operaciones falla, o si la transacción se anula explícitamente, todas las operaciones anteriores son también anuladas.
- Las operaciones de una transacción no tienen ningún efecto sobre la BD hasta que la transacción no se completa con éxito.

- Sean A y B dos tablas relacionadas en una BD, de forma que cuando se introduce un dato en la tabla A hay que modificar la tabla B para que se conserve la consistencia del sistema. En este caso, consideramos como una operación cada una de las modificaciones realizadas en cada tabla, la transacción aquí es el conjunto de las modificaciones, que hacen que la BD vuelva a estar en una forma consistente.
- Tras una modificación en la tabla A va a tener lugar la correspondiente modificación en la tabla B. Sin embargo, esto no es siempre posible, ya que el sistema puede *caer* en el momento más inoportuno.

- Sin embargo, si el sistema maneja el procesamiento de transacciones garantizará que si la transacción ejecuta algunas modificaciones (no todas) y se produce un fallo antes del final de la transacción, se anularán las modificaciones realizadas. El componente del sistema encargado de lograr este propósito es el **gestor de transacciones**.

```
BEGIN TRANSACTION
importe := 100;
ctaOrigen  := '2530 10 2000 1234567890';
ctaDestino := '2532 10 2010 0987654321';
UPDATE CUENTAS SET SALDO = SALDO - importe (actualiza la
  cuenta de origen)
WHERE CUENTA = ctaOrigen;
UPDATE CUENTA SET SALDO = SALDO + importe
WHERE CUENTA = ctaDestino;
INSERT INTO MOVIMIENTOS
(CUENTA_ORIGEN, CUENTA_DESTINO, IMPORTE, FECHA_MOVIMIENTO)
VALUES  (ctaDestino, ctaOrigen, importe, SYSDATE);
COMMIT;
```

- En una transacción los datos modificados no son visibles por el resto de usuarios hasta que se confirme la transacción.

- Propiedades ACID
  - Para asegurar la integridad de la BD se necesita que el SGBD mantenga las siguientes propiedades de las transacciones:

  ✓ **Atomicidad**: garantiza que todas las operaciones realizadas durante la transacción deben ser correctas o fallar todas. En estos casos, al confirmar la transacción (COMMIT) o al deshacerla (ROLLBACK) se garantiza que todos los datos quedan en un estado consistente.

  ✓ **Consistencia**: garantiza que la ejecución de una transacción conserva la consistencia de la BD. Es decir, la integridad de los datos se preserva al finalizar una transacción ya sea con éxito o con fallo.

  ✓ **Aislamiento**: garantiza que los efectos provocados por una transacción son invisibles para el resto de transacciones que se ejecuten concurrentemente, hasta que la transacción se ha completado. Cada transacción ignora al resto de las transacciones que se ejecuten concurrentemente en el sistema.

  ✓ **Durabilidad o permanencia**: garantiza que una vez que una transacción ha sido completada, los resultados de la transacción se hacen permanentes, incluso frente a fallos del sistema y medios de almacenamiento.

- El acceso a la BD se lleva a cabo mediante dos operaciones:

✓ **Leer(X)**: transfiere el dato X de la BD a la memoria intermedia perteneciente a la transacción que ejecuta la operación leer.

✓ **Escribir(X)**: transfiere el dato X desde la memoria intermedia perteneciente a la transacción que ejecuta la operación escribir a la BD. Actualiza el valor de un dato.

- En un sistema de BD real, la operación escribir(x) no tiene por que producir necesariamente una actualización de los datos en disco, puede almacenarse temporalmente en memoria y llevarse al disco más tarde.

- Un ejemplo bancario: Sea P un programa que transfiere $500 de la cuenta A a la B.

```
T:  leer(A);      1
    A:=A-500;     2
    escribir(A);  3
    leer(B);      4
    B:=B + 500;   5
    escribir(B);  6
```

Si T falla entre 3 y 6, el SGBD debe asegurar que no se hayan reflejado cambios en la BD.

- Supongamos que tenemos una BD con la que gestionamos los pedidos de los productos que vendemos. En concreto, cuando un cliente nos solicita un producto, comprobamos la disponibilidad y, en el caso en que podamos satisfacer el pedido, restamos a la cantidad que tenemos la cantidad que se nos ha pedido. Traduciendo todo esto a SQL, obtenemos la cantidad almacenada con:

```
Instrucción A: SELECT stock FROM productos
               WHERE productoID=1453
```

- La actualización del almacenamiento, una vez comprobada la disponibilidad, se obtiene con la siguiente instrucción:

```
Instrucción B: UPDATE productos
               SET stock=stock-1
               WHERE productoID=1453
```

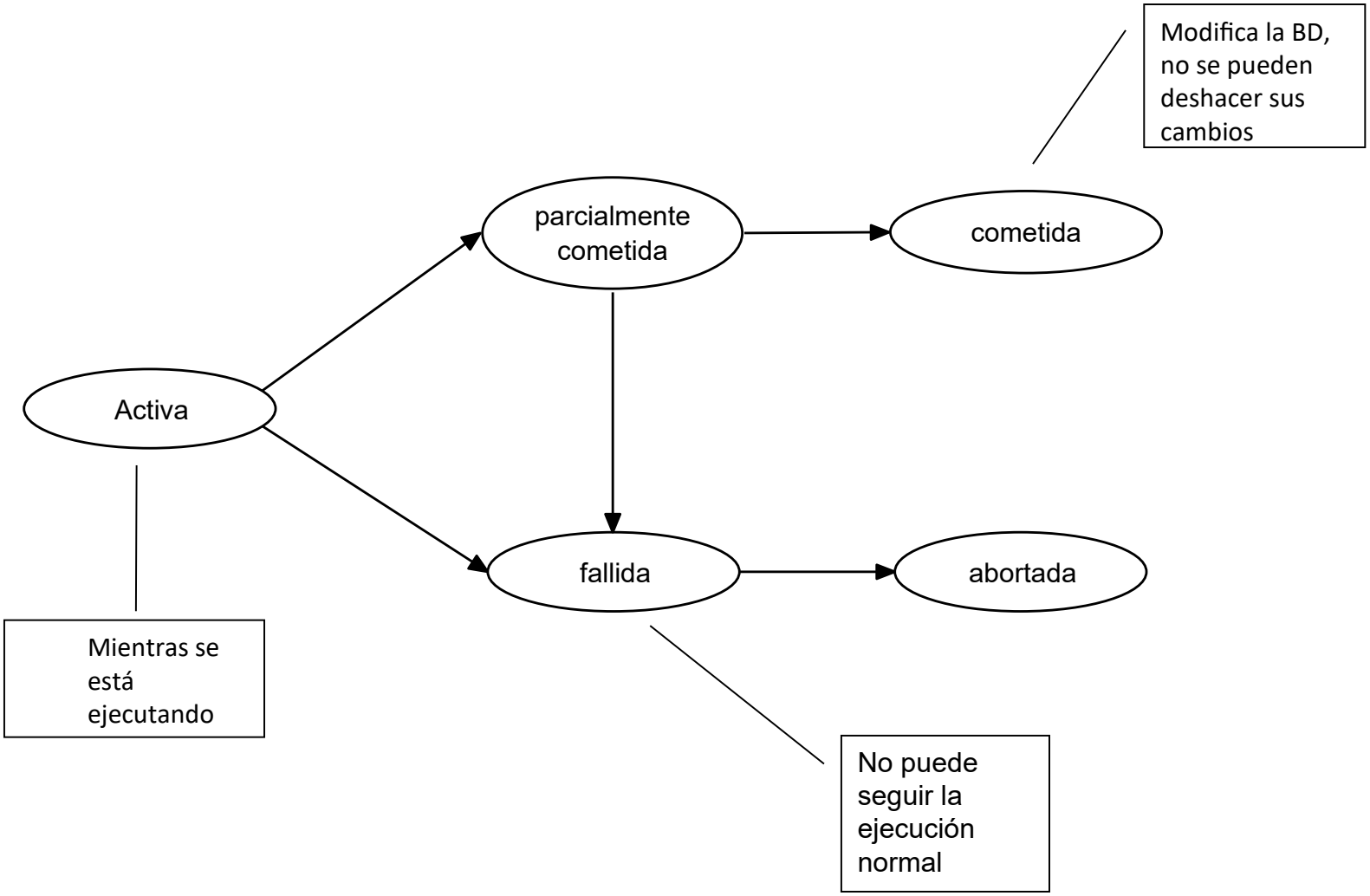
- Si dos usuarios intentan ejecutar esta operación, sin transacción:

1. A por el usuario 1: se devuelve un stock del producto equivalente a 1, por lo que el pedido será aprobado.
2. A por el usuario 2: el stock es 1 y también se aprobará el pedido.
3. B por el usuario 1: en este punto, en la BD el stock para el producto vale 0.
4. B por el usuario 2: ahora el stock vale -1, que es un valor equivocado.

- Estados de una Transacción

  - En ausencia de fallos todas las transacciones se realizan con éxito.
  - Una transacción que termina con éxito se dice que está **cometida**. Cuando una transacción se ha cometido, no se pueden deshacer los cambios.
  - Pero sabemos que los fallos pueden ocurrir, si hay un fallo la transacción no termina con éxito, entonces se dice que la transacción es **abortada**.
  - Si está asegurada la propiedad de atomicidad, una transacción abortada no debe tener efecto sobre el estado de la BD.
  - Existe un estado en el que la transacción está **parcialmente cometida**, es cuando se terminó de ejecutar la última instrucción, a partir de allí puede pasar a **cometida** o a **fallida**, cuando descubre que no puede continuar la ejecución normal. Una transacción se dice que ha terminado si ha llegado al estado cometida o abortada.

- Estados de una Transacción



- Si la transacción fue abortada el sistema tiene dos opciones:

- ✓ **Reiniciarla**: pero sólo si la transacción se ha abortado a causa de algún error hardware o software que no lo haya provocado la lógica interna de la transacción. Una transacción reiniciada se considera una nueva transacción.
- ✓ **Cancelarla**: Normalmente se hace esto si hay algún error interno lógico que sólo se puede corregir escribiendo de nuevo la secuencia de acciones que componen la transacción, debido a una entrada incorrecta, o debido a que no se han encontrado los datos deseados en la BD.

- Los errores que hacen que una transacción no termine exitosamente pueden ser:

- ✓ **Error lógico**: Por ejemplo no existe la cuenta A o la B en la BD.
- ✓ **Caída del sistema**: Se pierden los valores de la memoria principal.
- ✓ **Fallas del disco**: Se dañan varios bloques físicos del disco, lo que resultará en una transferencia errónea a la memoria principal cuando se ejecuta un *leer(A)*.

- Recuperación de transacciones

  - Una transacción comienza con la ejecución de una instrucción **BEGIN TRANSACTION** y termina con la instrucción **COMMIT** o **ROLLBACK**.
  - La operación COMMIT indica la finalización exitosa de una transacción.
  - Por el contrario la operación ROLLBACK indica la finalización de una transacción no exitosa, muestra al gestor de transacciones que algo ha salido mal, que la BD puede estar en un estado inconsistente y que todas las actualizaciones hechas desde el BEGIN TRANSACTION correspondiente deben ser **deshechas**.
  - La operación COMMIT establece lo que es conocido como **punto de verificación** (**checkpoint**), en este punto la BD debe estar en estado consistente.
  - Al establecer un punto de verificación todas las actualizaciones hechas desde el punto de verificación anterior se vuelven permanentes, antes de ese punto todas las actualizaciones deben ser vistas como tentativas, en el sentido de que pueden ser deshechas posteriormente.
  - La capacidad de realizar la recuperación se debe a que **los SGBD mantienen un histórico** de todas las operaciones de actualización realizadas en el sistema. Por tanto, si resulta necesario anular alguna modificación específica, el sistema puede utilizar la entrada concreta en el histórico para restaurar el valor original del objeto modificado.

- Recuperación basada en Bitácoras

  - Una **bitácora** es una estructura (archivo) usada para guardar información sobre las modificaciones que se realizaron a los datos, en la base o en la memoria. Una bitácora es una secuencia de registros. Existen varias implementaciones, una implementación posible tiene los siguientes campos:
  - ✓ **Nombre de la Transacción**: el nombre de la transacción que realiza la operación *escribir(x)*.
  - ✓ **Nombre del Dato**: el nombre único del dato que se va a modificar.
  - ✓ **Valor Antiguo**: el valor del dato anterior a la escritura.
  - ✓ **Valor Nuevo**: el valor que tendrá el dato después de la escritura.

  - Existen registros en la bitácora para guardar sucesos significativos durante la ejecución de una transacción. Algunos de ellos son:
  - [T, begin]: la transacción T inició su ejecución
  - [T, X, x_anterior , x_nuevo]: T realizó una escritura sobre el dato x
  - [T, commit]: T se completó exitosamente
  - [T, abort]: la transacción T ha sido anulada
  - [checkpoint]: el SGBD ha escrito en disco todos los buffers en memoria que han sido modificados

- Bitácora incremental con actualizaciones diferidas
  - Durante la ejecución de una transacción, todas las operaciones de COMMIT se postergan hasta que la transacción esté parcialmente cometida. Todas las actualizaciones se graban en la bitácora. Cuando una transacción esté parcialmente cometida, la información de la bitácora se utilizará para ejecutar las operaciones postergadas. Si el sistema se cae antes de completarse la transacción o se aborta, se hace caso omiso de la información en la bitácora, pero puede ser necesario rehacer el efecto de algunas.
  - Ejemplo: Sea T_0 una transacción que transfiere $50 de la cuenta A a la B. Y sea T_1 una transacción que retira $200 de la cuenta C

```
T0:  leer(A)
     A = A-50
     escribir(A)
     leer(B)
     B = B+50
     escribir(B)

T1: leer(C)
    C = C-200
    escribir(C)
```

Cada operación escribir(x), se traduce en la escritura de un nuevo registro en la bitácora, en el commit de T se trasladan los cambios a la BD

- Supongamos que los saldos de las cuentas A, B, y C eran de $1000, $2000 y $800, y que las transacciones se deben ejecutar una después de la otra, la bitácora sería:

  < T_0, begin>

  < T_0, A, 1000, 950> *

  < T_0, B, 2000, 2050> *

  < T_0,commit> **

  < T_1,begin>

  < T_1,C, 800, 600> ***

  < T_1,commit>

- Si falla en * no hay inconveniente, aún no se realizó ningún commit, A y B tienen aún sus valores iniciales. Pueden borrarse de la bitácora los registros de la transacción incompleta T_0.
- Si falla en ** , la BD está consistente con los nuevos valores de A y B.
- Si falla en ***, cuando el sistema vuelve a funcionar debe realizar la operación rehacer (T_0), y puede borrar lo correspondiente a la T_1.

- La operación **rehacer** (Redo) es idempotente, es decir el resultado de ejecutarla una sola vez es el mismo que si se ejecuta varias veces.
- Una transacción debe rehacerse si se cae el sistema y esa transacción tiene los registros <T, begin> y <T, commit>.

- Bitácora incremental con actualizaciones inmediatas
  - **Aplica todas las actualizaciones directamente a la BD** antes de realizar los commit y mantiene una bitácora de todos los cambios que se hacen al estado del sistema.
  - Si se presenta una caída, se utiliza la información de la bitácora para restaurar el sistema a un estado consistente previo. En caso de que ocurra una caída de sistema o fallo de una transacción se debe recurrir a una operación **deshacer** (Undo) que deshace los cambios hechos.
  - La operación deshacer extrae la información de la bitácora para determinar qué datos deben restaurarse. La operación deshacer restaura los valores de los datos a los valores anteriores, y rehacer fija los nuevos valores.

Modificación no cometida, en caso de fallo debo deshacer y dejar la BD con los valores anteriores

< T_0, start >

< T_0, A, 1000, 950 >

< T_0, B, 2000, 2050 > Si existe caída entonces anula T_0 y se realiza un deshacer(T_0)

< T_0, commit >

< T_1, starts >

< T_1, C, 800, 600 > Si existe caída: rehacer(T_0) y deshacer(T_1)

< T_1, commit > Si existe caída: rehacer(T_0) y rehacer(T_1)

- Puntos de Verificación

  – Cuando se presenta una falla del sistema, es necesario consultar la bitácora para determinar cuales son las transacciones que es preciso repetir o anular (revisar toda la bitácora).

  – Existen dos problemas:
  ✓ El proceso de búsqueda consume tiempo
  ✓ Varias de las transacciones que deben rehacerse ya efectuaron sus actualizaciones de BD. Repetirlas no causa daño, pero la recuperación tarda más.

  – Para reducir este tiempo extra existen los **puntos de verificación**. Durante la ejecución, el sistema mantiene la bitácora y ejecuta periódicamente verificaciones. Después de una falla, el proceso de recuperación examina la bitácora para determinar cual fue la última transacción que comenzó a ejecutarse después del último punto de verificación. La transacción puede localizarse si se examina la bitácora hacia atrás para encontrar el primer registro <checkpoint>, localizando después del registro <T_i,begin> que le sigue.

- Recuperación del sistema

  – El sistema debe estar preparado para recuperarse no sólo de fallos puramente locales, sino también de fallos globales.

  ✓ Un **fallo local** afecta sólo a la transacción en la que se presentó el fallo.
  ✓ Mientras que un **fallo global** afecta a todas las transacciones que se estaban realizando en el momento del fallo.

  – Ante un fallo global, el sistema deberá recuperarse de fallos del sistema y de fallos de los medios de almacenamiento:

  ✓ **Fallos del sistema**: que afectan a todas las transacciones que se están realizando, pero no dañan físicamente a la BD. También se conocen como caídas suaves.
  ✓ **Fallos de los medios de almacenamiento**: que causan daños a la BD o a una porción de ella, y afectan al menos a las transacciones que están utilizando esa porción. También se conocen como caídas duras.

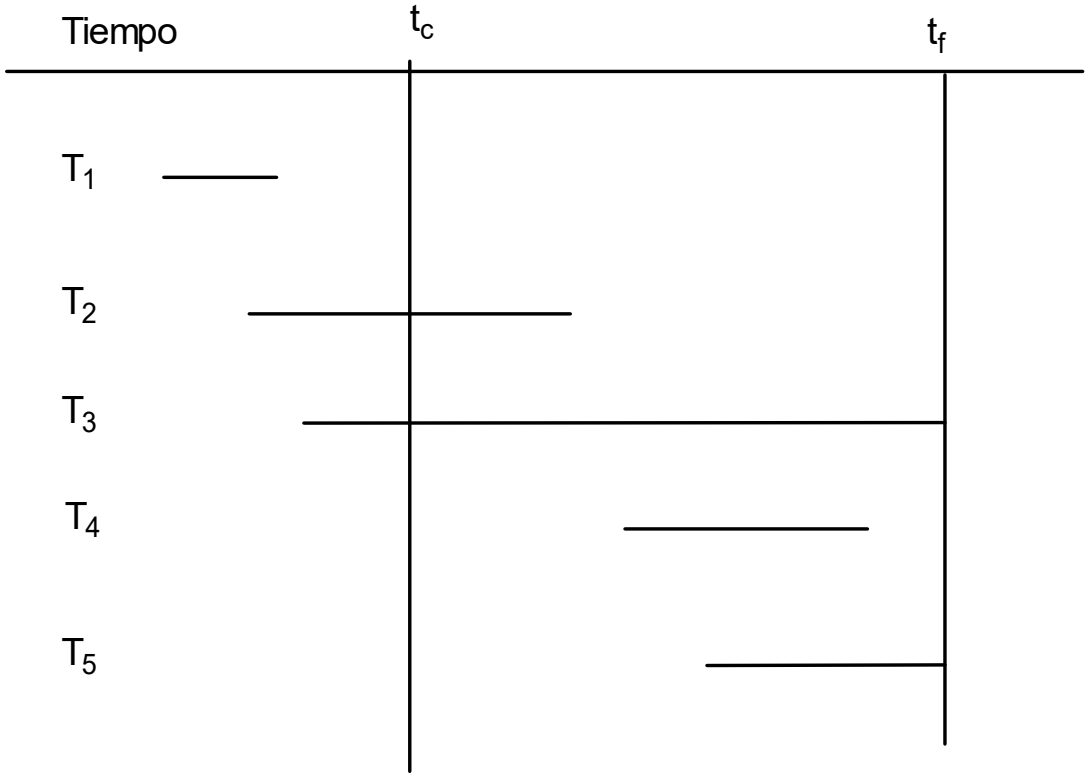
- Fallos del sistema

  - El punto crítico en este tipo de fallos es que **se pierde el contenido de la memoria principal**. Por tanto, ya no se reconocerá el estado preciso de la transacción que se estuviera realizando en el momento del fallo. Esa transacción jamás se podrá finalizar, por lo que será necesario anularla al reiniciar el sistema. Incluso podría ser necesario volver a realizar ciertas transacciones cuya ejecución sí terminó con éxito antes de la caída pero cuyas modificaciones no lograron ser transferidas de los buffers a la BD física.

  - Pero, ¿cómo sabe el sistema qué transacciones debe volver a realizar y cuáles debe anular?. Cada cierto intervalo de tiempo el sistema establece un punto de verificación de forma automática. Esto implica:

    1. Grabar físicamente el contenido de los buffers en la BD física
    2. Grabar físicamente un registro de punto de revisión especial en el histórico o bitácora física

  - El registro de punto de verificación incluye una lista de todas las transacciones que se estaban realizando en el momento de establecerse el punto de revisión.



- En el tiempo t_f sucede una falla en el sistema. El punto de verificación último antes de t_f es t_c. La transacción T_1 terminó satisfactoriamente. La transacción T_2 empezó antes de t_c y terminó satisfactoriamente antes de la falla. La transacción T_3 empezó antes de t_c pero no había terminado al producirse un fallo. T_4 se inició después del punto de verificación pero terminó antes del fallo. T_5 no había terminado al producirse la falla. Cuando se vuelve a iniciar el sistema, deberán deshacerse las transacciones T_3 y T_5 y deberán realizarse de nuevo las transacciones T_2 y T_4. La T_1 no entra en el proceso.

- Por tanto, al reiniciar el sistema se efectúa el siguiente procedimiento para identificar las transacciones T_2 a T_5:

1. Comenzar con dos listas de transacciones: DESHACER y REHACER, e incluir en la lista DESHACER todas las transacciones incluidas en el punto de verificación. Dejar vacía la lista REHACER.
2. Examinar la bitácora hacia delante a partir del registro del punto de revisión.
3. Si se encuentra una entrada de bitácora de "begin transacción" para la transacción T, añadir T a la lista DESHACER.
4. Si se encuentra una entrada de bitácora de COMMIT para la transacción T, pasar T de la lista DESHACER a la de REHACER.
5. Cuando se llegue al final de la bitácora, las listas identifican cuales transacciones debo rehacer y cuales deshacer.

- Cuando finalice este proceso, el sistema estará preparado para aceptar trabajos nuevos.

- Recuperación del medio

  - Un fallo de los medios de almacenamiento es un percance donde se destruye físicamente alguna porción de la BD. Esto implica el restaurar la BD a partir de una copia de seguridad y utilizar después la bitácora para realizar de nuevo todas las transacciones terminadas desde que se realizó dicha copia de seguridad. No será necesario anular las transacciones no finalizadas puesto que por definición estas transacciones se anularon (destruyeron) de todas maneras.

- Propiedades de SQL estándar

  - En SQL no existe una instrucción que haga iniciar explícitamente una transacción. Cada vez que el programa de aplicación inicia una transacción, SQL inicia implícitamente la transacción. Cuando se intenta ejecutar una instrucción que inicia una transacción, si no está ya en marcha una transacción, empieza una. La transacción continúa hasta que una de las instrucciones falla, provocando la anulación de toda la transacción, o hasta que se ejecuten las instrucciones COMMIT WORK o ROLLBACK WORK. La instrucción COMMIT WORK termina la transacción confirmándola, convirtiendo en definitivos los efectos de sus instrucciones sobre la BD. Sin embargo, la instrucción ROLLBACK WORK acaba anulándola (la palabra clave WORK es opcional).

- SQL especifica también que el **sistema debe garantizar la secuencialidad de las transacciones concurrentes**. Esto es, si se van a ejecutar varias transacciones en forma concurrente, el estado final de la BD debe ser igual al que hubiera quedado si las ejecuciones de las transacciones hubieran sido en serie.

- A pesar de que SQL no incluye ninguna instrucción BEGIN TRANSACTION explícita, algunos SGBD (p. e. SQL Server) introducen una sentencia para indicar el comienzo de una transacción: **BEGIN TRAN**.
- Si alguna de las operaciones de una transacción falla, hay que deshacer la transacción en su totalidad para volver al estado inicial en el que estaba la BD antes de empezar. Esto se realiza con la sentencia **ROLLBACK TRAN**.
- Si todas las operaciones de una transacción se completan con éxito hay que marcar el fin de una transacción para que la BD vuelva a estar en un estado consistente con la sentencia **COMMIT TRAN**.
- Hay una cuarta sentencia para trabajar con transacciones: **SAVE TRAN**, esta sentencia crea un punto de almacenamiento dentro de una transacción. Esta marca sirve para deshacer una transacción en curso sólo hasta ese punto.

- Control de Concurrencia

  - El término concurrencia se refiere al hecho de que los SGBD permiten que varias transacciones accedan a una misma BD a la vez. Por esto es que se necesita algún mecanismo de control de concurrencia para asegurar que las transacciones concurrentes no interfieran entre si, dejando a la base en un estado inconsistente.

- Tres problemas de concurrencia

  - La mayoría de los SGBD son multiusuario. Como dijimos, en estos sistemas se necesita un mecanismo de control de concurrencia. Sin este tipo de mecanismos pueden surgir muchos problemas, veremos los más significativos.
  - Son tres los errores que pueden presentarse, es decir, tres situaciones en las que una transacción, aunque correcta en sí, puede producir de todos modos un resultado incorrecto debido a una interferencia por parte de alguna otra transacción:

  1. El problema de la actualización perdida
  2. El problema de la actualización no cometida
  3. El problema del análisis inconsistente

- El problema de la actualización perdida

  - Este problema se presenta cuando dos transacciones acceden a los mismos datos y sus instrucciones se intercalan de forma incorrecta.
  - Consideremos la siguiente situación:

Transacción A	Tiempo	Transacción B
Leer registro R	T_1	
	T_2	Leer registro R
Actualizar registro R	T_3	
	T_4	Actualizar registro R

  - Es evidente que la actualización realizada por la transacción A se pierde porque la transacción B sobreescribe el resultado sobre el resultado anterior.

- El problema de la actualización no cometida

  – Este problema se presenta cuando se permite a una transacción leer (o peor aún: modificar) un registro que ha sido actualizado por otra transacción, y esta última todavía no la ha cometido. Es evidente que existe la posibilidad de que nunca se cometa, lo cual indicaría que los datos transitorios podrían no ser correctos y generar un error encadenado en otras transacciones.

Transacción A	**Tiempo**	**Transacción B**
	T_1	Actualizar registro R
Leer registro R	T_2	
	T_3	rollback

Transacción A	**Tiempo**	**Transacción B**
	T_1	Actualizar registro R
Actualizar registro R	T_2	
	T_3	rollback

- El problema del análisis inconsistente

  - Se realiza alguna operación sobre un conjunto de tuplas mientras otra transacción las está actualizando. Supongamos el siguiente caso, con valores iniciales de CTA1 = 40, CTA2 = 50, y CTA3 = 30:

Transacción A	Tiempo	Transacción B
Recuperar $CTA_1(40)$ Suma = 40	T_1	
Recuperar $CTA_2(50)$ Suma = 90	T_2	
	T_3	Recuperar $CTA_3(30)$
	T_4	Actualizar $CTA_{3:}$ $CTA_3 = 20$
	T_5	Recuperar $CTA_1(40)$
	T_6	Actualizar $CTA_{1:}$ $CTA_1 = 50$
Recuperar $CTA_3(20)$ Suma = 110	T_7	COMMIT

  - Nos encontramos con la situación de que el resultado de una transacción ha interferido claramente en el resultado de otra de duración mayor. El resultado 110 es incorrecto, y por tanto decimos que A ha realizado un análisis inconsistente.

- Bloqueo

  - El bloqueo es la técnica más usual que se utiliza para **resolver problemas de concurrencia**. La noción básica de bloqueo es simple: cuando una transacción requiere la seguridad de que el objeto (p. e. una o varias tuplas de la BD) en la cual está interesada no cambiará mientras lo está usando, entonces adquiere un bloqueo sobre ese objeto, con lo que ninguna transacción podrá leer ni modificarlo.

  - El bloqueo funciona del siguiente modo:

1. Primero suponemos la existencia de dos tipos de bloqueo, bloqueos exclusivos (X) y bloqueos compartidos (S), definidos en los siguientes dos puntos.
2. Si la transacción A tiene un bloqueo exclusivo (X) sobre el registro R, una solicitud por parte de una transacción B de cualquier tipo de bloqueo sobre R hará que B entre en un estado de espera. B espera hasta que A libere el bloqueo.
3. Si la transacción A tiene bloqueo compartido (S) sobre el registro R:

✓ Una solicitud por parte de una transacción B de bloqueo X sobre R hará que B entre en un estado de espera, y B espera hasta que se libere el bloqueo de A;

✓ Una solicitud por parte de una transacción B de un bloqueo S sobre R será concedida, y B tendrá también ahora un bloqueo sobre R.

– Veamos todo esto en una matriz de compatibilidad:

	X	**S**	**-**
X	no	no	si
S	no	si	si
-	si	si	si

– **Interpretación**: Consideremos un registro R, y que una transacción A tiene algún bloqueo sobre R, tal y como lo indican las entradas en las cabeceras de las columnas. Supongamos que una transacción B emite una solicitud de bloqueo sobre R tal y como indica la primera columna:

✓ Un **no** indica un conflicto → la petición de B no puede ser satisfecha y B pasa a un estado de espera

✓ Un **si** indica compatibilidad → la petición de B es satisfecha

- A continuación presentaremos un protocolo de bloqueo que utiliza los bloqueos X y S definidos anteriormente.

1. Una transacción que desea **recuperar una relación** debe primero adquirir un bloqueo S sobre esa relación.
2. Una transacción que desea **actualizar una relación** debe primero adquirir un bloqueo X sobre esa relación. Si ya tiene un bloqueo S sobre la relación debe adquirir un bloqueo X.
3. Si una petición de bloqueo de una transacción B es rechazada porque entra en conflicto con un bloqueo que ya tiene una transacción A, B pasa a un estado de espera y permanece así hasta que se libere el bloqueo de A. Cuando una transacción lee con éxito un registro, adquiere de forma automática un bloqueo S sobre él, y cuando lo actualiza, adquiere un bloqueo X.
4. Los bloqueos X se mantienen hasta el final de la transacción (COMMIT o ROLLBACK). Lo normal es que los bloqueos S se mantengan de igual forma.

- Veamos ahora cómo se resolverían los tres problemas de concurrencia que hemos descrito con anterioridad utilizando los bloqueos. En cada caso podremos obtener la secuencia de acciones llevadas a cabo por cada transacción, y los periodos de espera en los que se encuentran cada una, así como el tipo de bloqueos utilizado en cada caso.

- El problema de la actualización perdida

Transacción A	Tiempo	Transacción B
Leer registro R Adquirir bloqueo S	T_1	
	T_2	Leer registro R Adquirir bloqueo S
Actualizar registro R Adquirir bloqueo X Esperar Esperar	T_3	
	T_4	Actualizar registro R Adquirir bloqueo X Esperar Esperar ....

  - Las operaciones de lectura llevan implícito un bloqueo S, y las de escritura un bloqueo X, con lo que siguiendo la secuencia en el tiempo, las dos transacciones quedan en un tiempo de espera infinito, y no se pierde ninguna actualización (porque no se realiza). Esto, sin embargo, nos lleva a un problema que trataremos un poco más adelante: el **bloqueo mortal**.

- El problema de la actualización no comprometida

Transacción A	Tiempo	Transacción B
	T_1	Actualizar registro R Adquirir bloqueo X
Leer registro R Solicitar bloqueo S Esperar....	T_2	
	T_3	Rollback Liberar bloqueo X
Continuar: leer R Adquirir bloqueo S	T_4	

Transacción A	Tiempo	Transacción B
	T_1	Actualizar registro R Adquirir bloqueo X
Actualizar registro R Solicitar bloqueo X Esperar....	T_2	
	T_3	Rollback Liberar bloqueo X
Continuar: ActualizarR Adquirir bloqueo X	T_4	

  - Como se puede apreciar en las nuevas tablas que reproducen el caso del problema descrito en un apartado anterior, los bloqueos implícitos hacen que, dado que B ha realizado un bloqueo X sobre R, A no pueda leer el registro, y deba esperar hasta que se llegue a un final de transacción (en este caso un rollback), y se libere el bloqueo. El problema queda de este modo resuelto.

- El problema del análisis inconsistente
  - Analicemos ahora este problema, siguiendo los bloqueos implícitos que se han definido:

Transacción A	Tiempo	Transacción B
Recuperar $CTA_1(40)$ Adquirir bloqueo S sobre CTA_1 Suma = 40	T_1	
Recuperar $CTA_2(50)$ Adquirir bloqueo S sobre CTA_2 Suma = 90	T_2	
	T_3	Recuperar $CTA_3(30)$ Adquirir bloqueo S sobre CTA_3
	T_4	Actualizar CTA_3: Adquirir bloqueo X sobre CTA_3 CTA_3= 20
	T_5	Recuperar $CTA_1(40)$ Adquirir bloqueo S sobre CTA_1
	T_6	Actualizar CTA_1: Solicitar bloqueo X sobre CTA_1 Esperar....
Recuperar $CTA_3(20)$ Solicitar bloqueo X sobre CTA_3 Esperar....	T_7	

  - De nuevo, se soluciona el problema del análisis inconsistente, puesto que ninguna transacción finaliza y no genera resultados erróneos. Sin embargo, vuelve a aparecer el problema del bloqueo mortal que a continuación analizamos.

- El bloqueo mortal

  – Hemos visto que los bloqueos son capaces de resolver los principales problemas de concurrencia, pero que al mismo tiempo pueden ocasionar problemas adicionales de transacciones que entran en un tiempo de espera infinito. Este problema, que ha sido claramente ilustrado en los ejemplos, se denomina **bloqueo mutuo**. Sin embargo, no todas las posibilidades de bloqueos mutuos quedan representadas en los ejemplos ya vistos. Supongamos el siguiente caso, que también origina un bloqueo doble:

Transacción A	**Tiempo**	**Transacción B**
Adquirir bloqueo X sobre R_1	T_1	
	T_2	Adquirir bloqueo X sobre R_2
Solicitar bloqueo X sobre R_2 Esperar...	T_3	
	T_4	Solicitar bloqueo X sobre R_1 Esperar...

- Si se presenta un bloqueo mutuo, **es deseable que el sistema lo detecte y lo rompa**. Para ello, es necesario elegir una de las transacciones que están paralizadas como víctima y cancelarla, con lo que se liberan sus bloqueos y permite continuar a la otra transacción.

- El tratamiento que los sistemas dan a la transacción víctima una vez cancelada es variable:

✓ En algunos casos se reinician de forma automática las transacciones desde el principio, suponiendo que no se va a volver a repetir el caso de bloqueo mutuo.

✓ En otros casos, simplemente se avisa a la aplicación que maneja la transacción víctima que ha sido objeto de una cancelación debido a un bloqueo mutuo, y es responsabilidad del programador el tratar convenientemente estos problemas.

- Seriabilidad

  - La seriabilidad es el criterio de corrección aceptado comúnmente para la ejecución de un conjunto dado de transacciones.
  - **Se considera que la ejecución de un conjunto dado de transacciones es correcta cuando es seriable**, es decir cuando produce el mismo resultado que una ejecución en serie de las transacciones, ejecutando una a la vez.

  - ¿Cómo justificar la aseveración anterior?

1. Las transacciones individuales son consideradas correctas, es decir se da por hecho que transforman un estado correcto de la base de datos en otro estado correcto.
2. Por lo tanto ejecutar varias transacciones en serie también es correcto.
3. Por lo tanto una ejecución intercalada es correcta cuando equivale a alguna ejecución serie.

- Propiedades de SQL
  - El estándar de SQL **no proporciona ninguna posibilidad de bloqueo explícito**.
  - Requiere que **la implementación proporcione las garantías** para las transacciones concurrentes.
  - En particular requiere que las actualizaciones hechas por una transacción T1 no sean visibles ante ninguna otra transacción hasta que T1 termine con una confirmación (COMMIT).
  - Terminar con una confirmación ocasiona que todas las modificaciones hechas por la transacción se tornen visibles ante las demás transacciones.
  - Terminar con un ROLLBACK ocasiona que todas las actualizaciones hechas por la transacción sean canceladas.

- Niveles de Aislamiento

  - Vimos que el aislamiento es una propiedad (ACID) que los SGBD deben hacer cumplir a las transacciones: como si una transacción fuera la única que se estuviera ejecutando en el sistema, sus efectos se verán solo cuando se confirma la transacción.
  - El nivel de aislamiento determina en qué grado la transacción se aísla de las demás. Un **nivel alto** de aislamiento permite **bases más íntegras** pero disminuye el número de transacciones concurrentes. Un **nivel bajo** de aislamiento permite un mayor número de transacciones concurrentes pero **bases menos correctas**.

  - SQL maneja cuatro niveles de aislamiento:

✓ READ UNCOMMITTED (lectura no confirmada)
✓ READ COMMITTED (lectura confirmada)
✓ REPEATABLE READ (lectura repetible)
✓ SERIALIZABLE

  - El nivel **Serializable** es el nivel de mayor aislamiento. Si todas las transacciones son ejecutadas a este nivel, queda garantizado que la **ejecución intercalada de cualquier conjunto de transacciones concurrentes es seriable**.

- El nivel más bajo de aislamiento es **lectura no confirmada**, y el aislamiento que éstas poseen es el de garantizar que no se lean datos erróneos. El nivel siguiente es el de **lectura confirmada**, permite lecturas de datos confirmados, o sea de datos que se convirtieron permanentes en la BD. A continuación **lectura repetible**, el que garantiza que lecturas repetidas de uno o mas datos actualizados por una transacción, siempre tengan el mismo resultado.

- El estándar define tres formas en que se puede violar la seriabilidad:

1. **Lectura desactualizada (sucia)**: Supongamos que la transacción T1 realiza una actualización sobre alguna fila, luego la transacción T2 recupera esa fila y la transacción T1 termina con una instrucción ROLLBACK. Entonces T2 ha visto una fila que no existe.
2. **Lectura no repetible**: Supongamos que la transacción T1 recupera una fila, luego la transacción T2 actualiza esa fila y después la transacción T1 recupera nuevamente "la misma" fila. Entonces T1 ha recuperado la misma fila dos veces pero ve dos valores diferentes de ella.
3. **Fantasmas**: Supongamos que la transacción T1 recupera el conjunto de todas las filas que satisfacen una condición y la transacción T2 inserta una nueva fila que satisface esa condición. Si la transacción T1 repite ahora su petición de recuperación, verá una fila que antes no existía → un fantasma.

- Los diferentes niveles de aislamiento están definidos en términos de las violaciones de seriabilidad que permiten. **Si** significa que puede ocurrir la violación indicada y **No** que no puede ocurrir.

Nivel de aislamiento	Lectura sucia	Lectura no repetible	Fantasma
READ UNCOMMITTED	Si	Si	Si
READ COMMITTED	No	Si	Si
REPEATABLE READ	No	No	Si
SERIALIZABLE	No	No	No

- Desde el momento en que a una BD pueden acceder diferentes usuarios al mismo tiempo, en cada instante podremos tener distintas transacciones que manipulen la BD a la vez. El estándar SQL prevé que normalmente las transacciones se ejecuten en el **nivel de aislamiento serializable** (isolation level SERIALIZABLE), o sea en una modalidad de ejecución que **garantice la serializabilidad** de las transacciones.
- Ejemplo:

```
SET TRANSACTION ISOLATION LEVEL
< NIVEL DE AISLAMIENTO >
```

- SQL Server utiliza por defecto **READ COMMITTED**, que funciona correctamente la mayoría de las veces.