TEMA 6
TRANSACCIONES Y CONTROL DE CONCURRENCIA DISTRIBUIDO

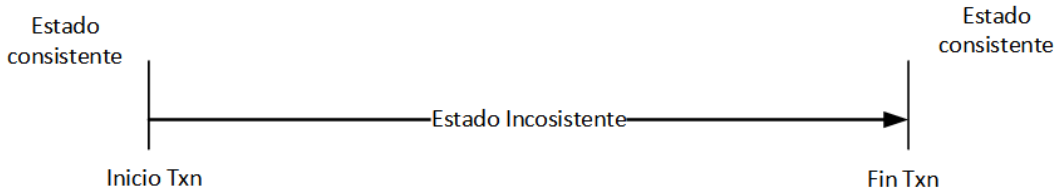
6.1. TRANSACCIONES

6.1.1. Definición de transacción (Txn).

- Unidad básica de procesamiento de datos de forma confiable y consistente.
- Formada por una serie de operaciones de lectura y escritura.

6.1.2. Consistencia de una Base de datos.

- Una BD se encuentra en un estado consistente si todas las restricciones y reglas de integridad definidas en ella se cumplen.
- El estado de una BD cambia al realizar operaciones de INSERT, UPDATE; DELETE.
- Para garantizar integridad, la BD no debería encontrarse en estado inconsistente.
- Sin embargo, la BD adquiere un estado de inconsistencia durante la ejecución de una Transacción., pero al terminarla debe recuperar el estado consistente:



6.1.3. Transacción consistente.

- Se refiere a las acciones realizadas por transacciones concurrentes.
- La BD debe estar en estado consistente, aunque existan peticiones que modifiquen de forma concurrente: lean o escriban datos.
- La condición anterior se complica en ***BD replicadas.***

6.1.4. Confiabilidad.

- Capacidad de una BD para ser flexible o tolerante a fallos, así como su capacidad de recuperación.
- La capacidad de recuperación de una BD le permite regresar de un estado inconsistente a uno consistente.

6.1.5. Formalización del concepto de una Txn

$O_{ij}(x) =$ Operación atómica Oj que pertenece a una transacción Ti que opera sobre un objeto X de la base de datos.

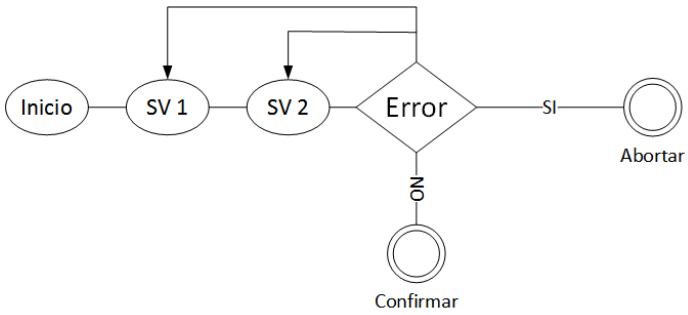
$OS_i = \cup_j O_{ij}$ Conjunto de operaciones que forman a una Txn.
$N_i = commit \mid rollback$

Ejemplo:
$R(x)$
$R(y)$
$x \leftarrow x + y$
$w(x)$
$commit;$

6.2. CONTROL TRANSACCIONAL.

- Se refiere a la administración de cambios realizados por instrucciones DML.
- Para realizar el control de transacciones se emplean las instrucciones `commit, rollback y savepoint`.
- A nivel de programación, el control transaccional debe contar con una estructura que garantice el estado consistente de la BD antes y después de la ejecución de una transacción.

```
begin transaction <txn_name>
   instrucción 1…
   instrucción 2…
   savepoint svp1;
   instrucción 3…
   instrucción 4…
   savepoint svp2;
   instrucción 5…
   commit;
exception
   --manejo del error
   rollback;
end;
```



Ejemplo:

```
connect medicos/medicos@jrcbd_s1
SQL> update paciente set nombre ='Douglas' where paciente_id = 1;
1 row updated.

--En otra terminal como usuario sys
SQL> connect sys/system as sysdba
Connected.
SQL> select xid as  txn_id, xidusn as segmento_undo, status
     from v$transaction;

             TXN_ID    SEGMENTO_UNDO STATUS
---------------- ------------ ----------------
01001E0074070000       1 ACTIVE
```

- Cuando la transacción concluye, el registro se elimina de `v$transaction`
- Una transacción puede concluir de formas alternas, no únicamente a través de la instrucción commit/rollback;

- Al salir de forma normal de sesión. Por ejemplo, al ejecutar el comando `disconnect` o `exit` de SQL *Plus, al cambiar de sesión, etc; Se aplica un `commit` implícito.
- Al terminar una sesión de forma abrupta o anormal, sea aplica un `rollback`.
- Al ejecutar una instrucción DDL. Antes de ejecutarla, se aplica un `commit` implícito a la transacción en curso.

Ejemplo:

- Considerar la siguiente tabla de datos:

PROD

PROD_ID	CANTIDAD
1001	30
1002	20
1003	15
1004	5
1005	12

Revisar la siguiente secuencia de operaciones en la que se ilustra la forma en la que se realiza el control transaccional.

Tiempo	Operación	Descripción
0	`commit;`	Se termina la transacción que estaba en curso en la sesión atual.
1	`set transaction name 'T1';`	Se inicia una Txn con el nombre 'cambio_1001'. Notar que a pesar de no especificar la instrucción `set transaction`, se crea una nueva transacción al ejecutar la siguiente instrucción

2	`update prod`
	`set cantidad = 40`
	`where prod_id=1001;`	

3	`savepoint` `after_update_1001;`	Esta instrucción crea un "punto de guardado". La idea es que la Txn puede hacer `rollback` hasta este punto. Es decir, la Txn puede deshacer todos los cambios desde un Tiempo N sin deshacer los cambios previos a esta instrucción.

4	`update prod`
	`set cantidad = 45`
	`where prod_id=1002;`	

5	`savepoint` `after_update_1002;`	
6	`rollback to savepoint` `after_update_1001;`	La Txn hará `rollback` hasta el tiempo t3: • El update realizado en t4 se revierte. • Cuando se hace un `rollback` sobre un `save point` la Transacción no termina, sigue en curso. • Se conserva el save point en t3, pero los subsecuentes son eliminados. Por ejemplo, en este punto, al hacer rollback al savepoint

		`after_update_1001` el savepoint marcado en t5 se pierde. • Los bloqueos posteriores al savepoint `after_update_1001` se liberan, pero se conservan los bloqueos antes de dicho savepoint.
7	`select * from prod;`	Observar los valores actuales: PROD_ID CANTIDAD ---------- ---------- 1001 40 1002 20 1003 15 1004 5 1005 12

8	`update prod`
	`set cantidad = 49`
	`where prod_id =1003;`	

9	`select * from prod;`	Observar los valores actuals PROD_ID CANTIDAD ---------- ---------- 1001 40 1002 20 1003 49 1004 5 1005 12
10	`rollback;`	La transacción T1 revierte todos los cambios y termina.
11	`select * from prod;`	Observar los valores actuales. PROD_ID CANTIDAD ---------- ---------- 1001 30 1002 20 1003 15 1004 5 1005 12
12	`set transaction name 'T2'`	Inicia la transacción 2

13	`update prod`
	`set cantidad = 40`
	`where prod_id=1001;`	

14	`commit;`	La Txn 2 termina y el cambio se hace permanente.

6.3. PROPIEDADES ACID DE LAS TRANSACCIONES.

- Atomicidad
- Consistencia
- Aislamiento
- Durabilidad.

6.3.1. Atomicidad.

- Una transacción se maneja como si se tratara de una sola operación. Por lo tanto, o todas las operaciones se concluyen o ninguna.
- Fundamental para garantizar la consistencia de los datos una vez que la Txn concluye.

6.3.2. Consistencia

- Capacidad para llevar a la BD de un estado consistente a otro.
- Para realizar esta tarea se requiere de la aplicación de los llamados ***Niveles de aislamiento***.

6.3.3. Aislamiento

- Es la propiedad que requiere cada Txn para ver a la BD consistente en cualquier instante de tiempo.
- Una Txn en ejecución no debe revelar sus cambios a otras Txn hasta confirmar dichos cambios.

Ejemplo:

- ¿Cuánto valdrá x al final de la ejecución de estas instrucciones?

$x = 50$

$Tiempo$	txn_1	txn_2
1	$T_1: R(x)$	
2	$x \leftarrow x + 1$	
3	$w(x)$	
4		$T_2: R(x)$
5	$commit$	$x \leftarrow x + 10$
6		$w(x)$
7		$commit$

$x =$?

6.3.4. Durabilidad.

- Asegura que una vez que la transacción se confirma, los datos son permanentes "***sin posibilidad"*** de pérdida sin importar fallas posteriores al ejecutar la instrucción `commit`.

6.4. NIVELES DE AISLAMIENTO

- Definidos en el estándar SQL-92
- Permiten implementar principalmente la propiedad de consistencia de una transacción.
- Los niveles de aislamiento se definen con base a 3 problemas que pueden ocurrir durante la ejecución de varias transacciones de forma concurrente:
    - Lecturas sucias
    - Lecturas no repetibles.
    - Lecturas fantasmas.

6.4.1. Lecturas sucias.

- Se refiere a la posibilidad de modificar datos de una Txn que aún no ha concluido.
- T1 modifica un dato el cual es leído por una Txn T2 antes de que T1 haga commit o rollback. Si Ti hace rollback, T2 tendrá un valor que ***nunca existió*** en la BD.
- Este problema se resuelve con el primer nivel de aislamiento llamado ***Lecturas confirmadas,*** sin embargo, permite la ocurrencia de lecturas no repetibles y lecturas fantasma.

6.4.2. Lecturas no repetibles

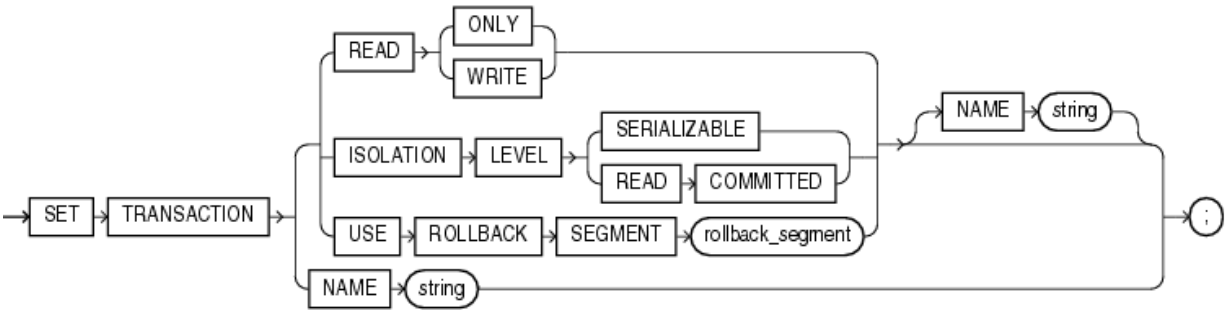
- T1 lee un dato, t2 modifica o elimina un dato y hace `commit`. SI T1 vuelve a leer, T1 va a leer un valor diferente o no va a encontrar el dato. Ambas lecturas regresan resultados diferentes cuando deberían regresar el mismo resultado.
- En resumen: Los datos se mueven mientras T1 se está ejecutando haciendo cálculos y "***asumiendo***" que los datos que se leyeron no han cambiado.
- Este problema se resuelve con el segundo nivel de aislamiento llamado ***Lecturas repetibles,*** sin embargo, permite la ocurrencia de lecturas fantasma.

6.4.3. Lecturas fantasmas.

- Se realiza una consulta con un cierto predicado por una transacción T1. T2 inserta un nuevo registro que satisface el predicado de T1.
- Si T1 vuelve a ejecutar la consulta, existirán más registros de los leídos originalmente lo cuan puede generar problemas.
- Este problema se resuelve con el tercer y último nivel de aislamiento llamado ***Serializable.***

En Oracle, la instrucción SET TRANSACTION se emplea para definir y configurar el comportamiento de una transacción:

```
set transaction
   { { read { only | write }
     | isolation level
       { serializable | read committed }
     | use rollback segment rollback_segment
     } [ name string ]
   | name string
   } ;
```



Existen diversas técnicas y algoritmos para implementar estos niveles de aislamiento en las bases de datos tanto centralizadas como distribuidas. El mecanismo más comúnmente empleado es el basado en ***bloqueos***.

6.5. CLASIFICACIÓN DE LAS TRANSACCIONES.

- Existen varios criterios para clasificar a una transacción.

6.5.1. Clasificación de las transacciones por su duración.

- Transacciones On-Line (De corta duración)
- Transacciones Batch (de larga duración).

6.5.2. Clasificación de las transacciones por el orden de lecturas y escrituras:

- Modelo general. Se aplican lecturas y escrituras sin orden especifico.

T_1: $\{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$

- Transacciones Two Steps (2 pasos). Primero se ejecutan todas las lecturas y después todas las escrituras.

T_2 : $\{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$

- Transacciones restringidas. Todos los datos deben ser leídos antes de una actualización.

T_3 : $\{R(x), R(y), W(y), R(z), W(x), W(z), R(w), W(w), C\}$

- Transacciones basadas por un mecanismo o modelo de ejecución. Cada pareja formada por una lectura y escritura se ejecutan de forma atómica:

T_3 : $\{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$

6.5.3. Clasificación de las transacciones por su estructura

- Transacción plana. Formada por un conjunto de operaciones atómicas delimitadas por un solo inicio y fin (sin anidamientos).

```
begin transaction compraViaje
  operacion 1
  operacion 2

  . . . .
  operacion n

end transaction
```

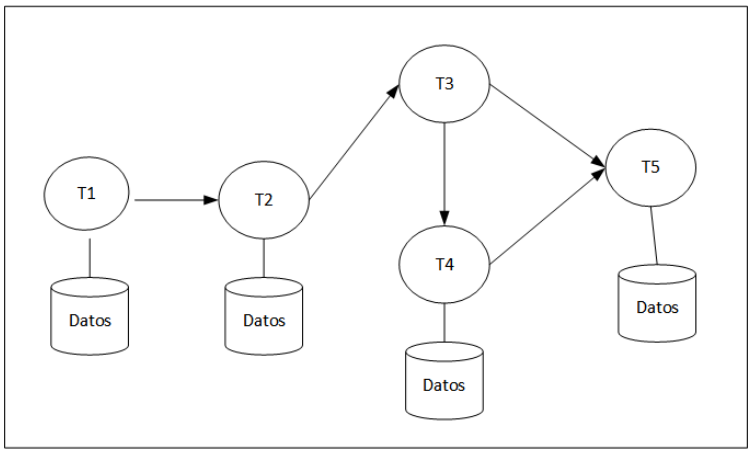
- Transacción Anidada. Una transacción puede incluir como parte de sus operaciones a otras transacciones. Bloques anidados.

```
begin transaction compraViaje
  operacion 1
  operacion 2
    begin transaction reservaHotel
      operacion 1
      operacion 2
      . . . .
    end transaction;
    begin transaction reservaVuelo
      operacion 1
      operacion 2
      . . . .
    end transaction;
  . . . .
  operacion n
end transaction;
```

- Flujos de trabajo. Secuencia de tareas organizadas para ejecutar o implementar un proceso de negocio. Un proceso de negocio es un conjunto de operaciones que son particulares a las reglas de operación de una organización. Estos procesos pueden incluir el uso de diversos recursos transaccionales como son: empleo de más de una base de datos, bases de datos distribuidas, etc.

Ejemplo:

Proceso de negocio que realiza un cliente para reservar un viaje.



Elementos del flujo de trabajo:

- T1: Registro de los datos del cliente
- T2: Reservación del vuelo
- T3: Reservación del hotel
- T4: Renta del auto
- T5: Cargo a tarjeta en banco.

6.5.4. Transacciones anidadas en Oracle: Transacciones autónomas.

- Una transacción autónoma en Oracle es una transacción independiente que puede ser iniciada dentro de otra transacción (transacción principal o transacción padre).
- La transacción padre puede suspenderse mientras que la transacción autónoma se esté ejecutando.
- Una vez que la transacción anidada termina (`commit` o `rollback`), la transacción principal reanuda su ejecución.

- Este tipo de transacciones son útiles para escenarios en donde la transacción anidada debe ejecutarse de forma independiente sin importar si la transacción principal hace `commit` o `rollback`.

La sintaxis general para crear una transacción autónoma se describe a continuación:

```
procedure proc_main is
 -- declaracion de variables
begin  -- Inicia transacción main
  select ...
  insert  ...
  proc_2;      --Observar que se invoca al procedimiento proc_2 que
               --contiene 2 transacciones autónomas TA-1 y TA-2.
  update ....  --Continua Txn principal después de que TA-1 y TA-2
concluyeron
end;

procedure proc_2 is
   pragma autonomous_transaction; --indica que todas las Txn de este
                                  --bloque de código serán autónomas.

begin  -- En este momento la Transacción principal se suspende e inicia una
       --nueva txn autónoma:  TA-1
  insert ...
  update ...
  commit;     -- TA-1 termina.
  insert ...  -- TA-2 inicia
  delete ...
  commit;    ---TA-2 termina
end;   --En este punto las  transacciones autónomas han concluido, por lo
       --tanto la Txn principal reanuda.
```

- En PL/SQL todas las transacciones que se deseen ejecutar como autónomas deben estar contenidas en un bloque PL (BEGIN-END) bajo un contexto de "autonomía "indicado por la instrucción `pragma_autonomous_transaction`.
- Las transacciones autónomas no pueden leer datos que no han sido confirmados por la transacción principal.
- Los cambios que realice una transacción autónoma serán visibles a otras transacciones una vez que esta termine sin importar que la transacción principal aún se encuentre en ejecución
- Dentro del mismo bloque PL pueden crearse N transacciones autónomas. Al terminar una, inicia la otra como se puede apreciar en el pseudo -código anterior.

Ejemplo:

- Independiente de los cambios que se realicen al inventario de productos, se deben registrar los datos de los empleados cada vez que se intente actualizar las existencias de los productos.

```
create table  operacion_usuario(
  operacion_usuario_id number(10,0) generated always as identity,
  fecha_operacion date not null,
  descripcion varchar(4000) not null
);
```

```
--transaccion automona
create or replace procedure p_actualiza_operacion is
pragma autonomous_transaction;
  begin
    insert into operacion_usuario (fecha_operacion,descripcion)
    values(sysdate,'operacion 1');
    commit;
  end;
/
show errors

--procedimiento principal
create or replace procedure p_actualiza_productos is
  begin

    update prod set cantidad = 30 where prod_id =1000;
    --inserta datos del usuario de  forma independiente
    p_actualiza_operacion;

  end;
/
show errors
```

- Para ejecutar el procedimiento:

```
--ejecuta el procedimiento principal y hace rollback
exec p_actualiza_operacion;
rollback;
col  descripcion format A30
select * from operacion_usuario;

OPERACION_USUARIO_ID FECHA_OPE DESCRIPCION
-------------------- --------- ------------
1 23-MAY-17 operacion 1
2 23-MAY-17 operacion 1
3 23-MAY-17 operacion 1
```

- Observar en el código anterior que a pesar de haber hecho `rollback` en la transacción principal, los datos de la tabla `operacion_usuario` se conservan.

6.6. CONTROL DE CONCURRENCIA.

- Se refiere al problema de sincronizar múltiples transacciones que interactúan de forma simultánea y que en diversos casos intentan acceder a un mismo recurso o dato.
- Lo anterior implica 2 retos a resolver:
  - Mantener la consistencia de la base de datos. Esto se podría resolver fácilmente serializando a cada transacción. De aquí surge el siguiente requerimiento:
  - Mantener un nivel alto de concurrencia para realizar operaciones simultáneas (no degradar desempeño) sin comprometer la consistencia de los datos.

6.6.1. Operaciones en conflicto

- Se dice que 2 operaciones $O_{ij}(x)\ y\ O_{kl}(x)$ que pertenecen a 2 transacciones $T_i\, y\, T_k$ y que intentan acceder al recurso x de forma concurrente estarán en conflicto si al menos una de ellas es una operación de escritura.
- Lo anterior indica que, si en una transacción existen escrituras, pueden existir problemas de consistencia de datos.
- El ***orden*** en el que se deberán ejecutar las operaciones es importante.
- En único caso donde no habría conflicto es si ambas operaciones representan lecturas.

Ejemplo:

- Considerar las 2 siguientes transacciones. ¿Cuál debería ser el orden correcto de ejecución?

$T_1: read(x)$	$T_2: read(x)$
$x \leftarrow x + 1$	$x \leftarrow x - 1$
$w(x)$	$w(x)$
$commit$	$commit$

- Solución:

$R = \{\, R1(x), W1(x), C1, R2(x), W2(x), C2\,\}$

La técnica anterior garantiza la consistencia, pero disminuye la capacidad de procesamiento concurrente. A continuación, se muestran los algoritmos empleados para realizar el control de concurrencia.

6.6.2. Mecanismos de control de concurrencia.

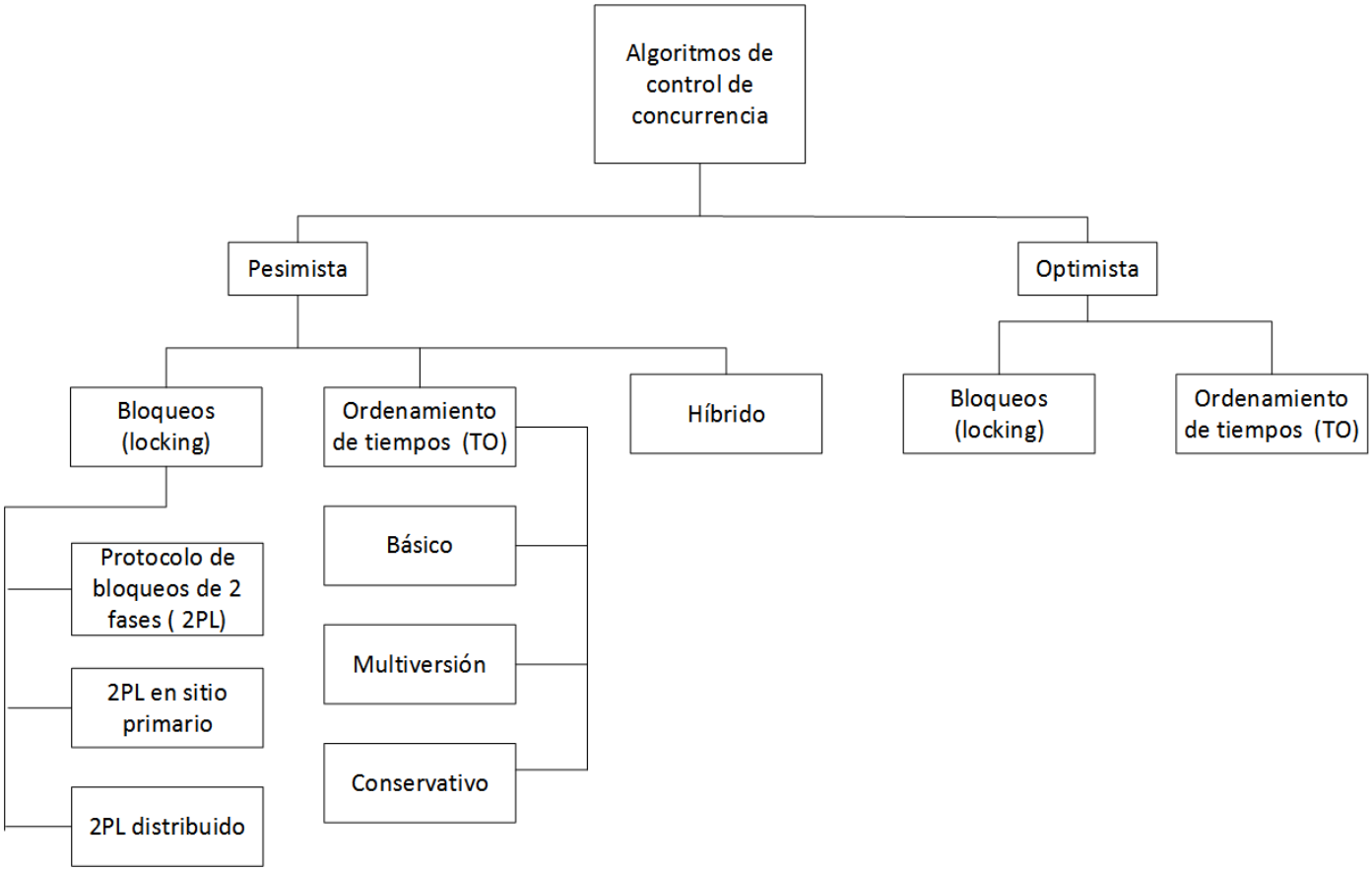
Existen 2 principales estrategias:

- Algoritmos basados en el acceso exclusivo a recursos compartidos llamado también ***mecanismo de bloqueos*** o ***locking***.
- Algoritmos que intentan ordenar la ejecución de transacciones con base a un conjunto de reglas o ***protocolos***.

Ambas estrategias pueden ser empleadas desde 2 puntos de vista:

- Punto de vista ***pesimista***: Asume que existirán múltiples conflictos durante la ejecución de las transacciones por lo que las validaciones para sincronizar la correcta ejecución de múltiples transacciones se realizan desde el principio.
- Punto de vista ***optimista***: Asume que NO existirán conflictos y por lo tanto, la validación para determinar que efectivamente no ocurrieron conflictos se realiza hasta el final, justo antes de que la transacción concluya.

El siguiente diagrama ilustra la clasificación general.



6.7. CONTROL DE CONCURRENCIA BASADO EN BLOQUEOS - ESTILO PESIMISTA.

- Realiza una serialización de operaciones a través del empleo de bloqueos.
- Previene interacciones que pueden alterar o destruir la ***consistencia*** e ***integridad*** de los datos cuando múltiples transacciones intentan modificar o acceder a un mismo dato.
    - Consistencia: Los datos que una Txn T1 lee o modifica no debe ser modificada por otras Txn hasta que T1 termine.
    - Integridad: Los datos que existen en la BD deben ser el reflejo de haber realizado un conjunto de cambios en una secuencia correcta. Es decir, como si no existirá concurrencia.
- Existen diversos tipos de bloqueos dependiendo del tipo de operación a realizar.
- Cuando una transacción requiere acceder a un dato ya sea para leerlo o modificarlo, se debe adquirir una especie de permiso para poder hacerlo. A la acción de obtener este permiso se le conoce como la "***obtención de un bloqueo***" . Cuando una Txn1 adquiere dicho bloqueo, ninguna otra podrá acceder al mismo dato hasta que Txn1 libere el bloqueo.

6.7.1. Tipos de bloqueos.

- Bloqueos exclusivos (Exclusive Locks)
    - Cuando un recurso va a ser accedido por una Txn (un registro, una tabla) dicho recurso puede adquirir ***un solo bloque exclusivo.*** Es decir, no pueden existir más de un bloqueo exclusivo para un mismo recurso.
- Bloqueos compartidos (Shared Locks)

  - A diferencia de los bloques exclusivos, un recurso puede adquirir más de un bloqueo compartido a la vez.
- Recordar: Para que un recurso pueda ser accedido por una transacción, debe existir un bloqueo asignado.
- Los bloqueos afectan la interacción entre lectores y escritores.
  - Lectores: Empleados para leer un dato
  - Escritores: Modifican un dato.
- **=> Un *writer puede bloquear a otro writer*** . Cuando una Txn actualiza un registro, esta debe adquirir y establecer un bloqueo únicamente sobre ese registro. Esto evita que otras Txn actualicen el mismo registro de forma simultánea.
- ***=> Un reader nunca bloquea un writer ni a otro reader***. Lo anterior significa que un writer puede modificar el registro que está siendo leído por el reader.
- Existe una excepción al punto anterior. La cláusula `select … for update` si establece un bloqueo sobre el registro que se está leyendo.
- En Transacciones distribuidas, puede existir otra excepción: En algunos casos, readers deberán esperar a que los writers terminen de modificar el mismo bloque de datos que se intenta leer, el escenario es para lecturas que se intentan ejecutar sobre ***transacciones distribuidas pendientes*** (se verá este tema más adelante).
- Mientras un registro está siendo modificado el manejador proporciona la versión original del registro a los readers para ofrecer consistencia. La versión original se lee del llamado "Undo". El Manejador guarda una copia de la versión original de todas las Txn en curso en un tablespace llamado `undotbs1` el cual es creado al momento de crear la BD. Esta información se emplea entre otras cosas, para que otras Txn puedan leer el dato original.

Ejemplos:
Considerar la siguiente tabla de datos:

PROD

PROD_ID	CANTIDAD
1001	30
1002	20
1003	15
1004	5
1005	12

Para todos los ejercicios siguientes, determinar los valores del campo cantidad al terminar de ejecutar las transacciones concurrentes.

Ejemplo 1:
- Txn1 Agrega 11 productos 1001.
- Txn2 Agrega 15 productos a 1001.
- Ocurre un error y Txn1 aborta la transacción
- Al final se espera que en total de registros 1001 debe ser 30+15 = 45.
- Considere que la BD está configurada con el nivel de aislamiento read commited.

Tiempo	Txn1	Txn2	Descripción
1	`Select cantidad into v_cantidad from prod where prod_id = 1001`		Txn1 adquiere un bloqueo compartido para accede al renglón 1001 Obtiene `v_cantidad = 30`
2	`Update prod Set cantidad =v_cantidad+ 11 Where prod_id=1001`		Txn1 adquiere un bloqueo exclusive sobre el registro 1001 para modificarlo. Al aplicar el update, se actualiza un registro con `cantidad = 11+30 = 41`
3		`Select cantidad into v_cantidad from prod where prod_id = 1001;`	Txn2 adquiere un bloqueo compartido para accede al renglón 1001 Obtiene `cantidad = 30` debido al nivel de aislamiento configurado por default READ COMMITED. En este momento el renglón 1001 tiene 2 bloqueos compartidos generados.
4	`Rollback;`		Txn1 termina, no se aplica cambio alguno.
5		`Update prod Set cantidad =v_cantidad+ 15 Where prod_id=1001;`	Txn2 adquiere un bloqueo exclusivo sobre el registro 1001 para modificarlo. Al aplicar el update, se actualiza un registro con `cantidad = 30+15 = 45`
6		`Commit;`	Txn2 termina.
7		`Select cantidad From prod where prod_id = 1001;`	Txn2 lee el registro y obtiene `cantidad = 45`

- El resultado es el esperado. Txn1 hizo rollback en T4. La lectura realizada en T3 permitió que Txn2 no fuera afectada por los cambios de Txn1 y el valor final es el esperado.
- Sin embargo, ¿Qué problema existiría si la BD permitiera y estuviera configurada con el nivel 0, es decir, sin niveles de aislamiento?
    - En este caso, el valor leído en T3 sería 41. Por lo tanto, en T6 Txn2 actualiza a 41+15 = 56.
    - El valor leído en T7 es 56. Esto es incorrecto ya que Txn2 realizó `rollback`. El valor 41 leído en T3, en realidad nunca existió en la BD.
- Existe otro problema: ¿Qué hubiera pasado si Txn1 hace `commit`?. Revisar el siguiente ejemplo

Ejemplo 2:

Suponer que 2 peticiones simultaneas están actualizando las existencias de los productos que han comprado los clientes:

- Txn1 va a agregar 11 productos para prod_id 1001
- Txn2 va a agregar 15 productos para prod_id = 1001
- Al final se esperaría que el total de productos 1001 es 30+11+15 = 56
- Observar lo que sucede cuando ambas transacciones se ejecutan de gorma paralela o concurrente:

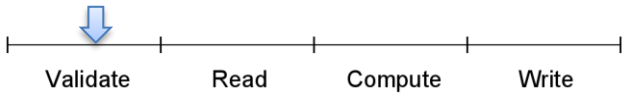
Tiempo	Txn1	Txn2	Descripción
1	`Select cantidad into v_cantidad from prod where prod_id = 1001`		Txn1 adquiere un bloqueo compartido para acceder al renglón 1001 Obtiene `v_cantidad = 30`
2		`Select cantidad into v_cantidad from prod where prod_id = 1001`	Txn2 adquiere un bloqueo compartido para acceder al renglón 1001 Obtiene `v_cantidad = 30` En este momento el renglon 1001 tiene 2 bloqueos compartidos generados.
3	`Update prod Set cantidad = v_cantidad+ 11 Where prod_id=1001`		Txn1 adquiere un bloqueo exclusive sobre el registro 1001 para modificarlo. Al aplicar el `update`, se actualiza un registro con `cantidad = 11+30 = 41`
4		`Update prod Set cantidad = v_cantidad + 15 Where prod_id=1001`	Txn2 intenta adquirir un bloqueo exclusivo sobre el registro 1001. Sin embargo, la petición entra en modo de espera ya que no es posible generar uno nuevo hasta que Txn1 libere el existente.
5	`Commit;`		Txn1 termina y por lo tanto se libera el bloqueo exclusivo que adquirió en T3
6		`1 registro actualiazado.`	Txn2 sale del estado en espera ya que el bloqueo exclusivo sobre el registro ha sido liberado por Txn1 y actualiza el registro con `cantidad = 30+15 =45`
7		`Commit;`	La Txn2 termina.
8	`Select cantidad From prod`		Txn1 lee 45

	Where Prod_id =1001;		
9		Select cantidad From prod Where Prod_id =1001;	Txn2 lee 45

- Al final se tienen solo 45 productos. ¿Qué sucedió?
- A pesar de los bloqueos que ofrece la BD para garantizar consistencia e integridad, el resultado obtenido no es correcto.
- El problema generado se le conoce como ***lost update***. En T6 Txn2 actualiza el registro 1001 sin considerar que ese registro ya había sido modificado por otra transacción.
- Para solucionar el problema se pueden aplicar 2 técnicas:
  - Aumentar el nivel de aislamiento continuando con el enfoque pesimista.
  - Cambiar al enfoque optimista.

Ventajas del enfoque pesimista:

- Permite realizar un control adecuado del orden de ejecución de operaciones concurrentes que intentan acceder a un mismo recurso a través del uso de bloqueos, es decir, este enfoque valida que el recurso se encuentre libre antes de ser accedido:



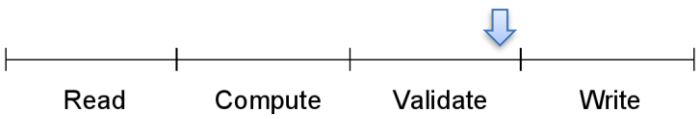
- Otra ventaja es que el usuario final no interviene en lo absoluto. El control es completamente automático.

Desventajas del enfoque pesimista:

- Reduce la capacidad de procesamiento concurrente, en especial si se aumenta el nivel de aislamiento como se propuso anteriormente.
- Al aumentar el número de bloqueos se tiene latente la ocurrencia de Dead Locks.

6.8. CONTROL DE CONCURRENCIA BASADO EN BLOQUEOS - OPTIMISTA

- Como se mencionó anteriormente, en esta técnica se asume que no habrá problemas y por lo tanto se mantiene un nivel de aislamiento relativamente bajo.
- En el ejemplo anterior, se generó un "Lost Update" debido a que el nivel de aislamiento READ COMMITED no evita este problema, las lecturas no repetibles, ni las lecturas fantasmas.
- En esta técnica la validación de un posible "Lost Update" se realiza justo antes de que la transacción actualice un dato:



- Considerando el ejemplo anterior, el problema se genera en el tiempo T4 a T6 al ejecutar la sentencia update:

```
Update prod
Set cantidad = v_cantidad + 15
Where prod_id=1001;
```

- La validación consiste en verificar que el valor de la columna cantidad no ha cambiado desde que fue leído anteriormente (siendo optimistas se esperaría que el valor no ha cambiado).
- Para implementar esta validación típicamente se agrega una condición a la sentencia `update` para que el cambio se realice únicamente si el valor del campo cantidad tiene el mismo valor desde la primera lectura:

```
Update prod
Set cantidad = v_cantidad + 15
Where prod_id=1001
and cantidad = v_cantidad;
```

- Si el campo cantidad fue modificado por otra transacción, la sentencia `update` regresaría 0 registros actualizados.
- Esta condición deberá ser detectada empleando programación.
- La acción a seguir al detectar esta condición es totalmente dependiente de cada aplicación.
- Por ejemplo, en un sistema de asignación de asientos, el sistema debería mostrar un mensaje de error al usuario indicando que ya alguien más ha ocupado el asiento.
- Para el ejemplo anterior, bastaría con volver a leer el dato `cantidad` y reintentar la actualización:
- El siguiente fragmento de código muestra la forma de implementar la validación empleando la estrategia optimista:

```
v_num_registros_actualizados := 0;

while v_num_registos_actualidzados = 0
loop
  Update prod
  Set cantidad = v_cantidad + 15
  Where prod_id=1001
  and cantidad = v_cantidad;
  num_registros_actualizados := sql%rowcount;
  if  num_registros_actualizados  = 0 then
    -- vuelve a leer el dato
    select cantidad into v_cantidad
    from prod where prod_id = 1001;
  end if
end loop;
```

- Como se puede observar, se emplea un loop para reintentar las lecturas hasta que esta sea repetible.

Ventajas del control de optimista:

- Mejora el nivel de concurrencia al dejar que las validaciones se hagan al final.
- Permite conservar el nivel de aislamiento recomendado READ COMMITED y mantener consistencia.

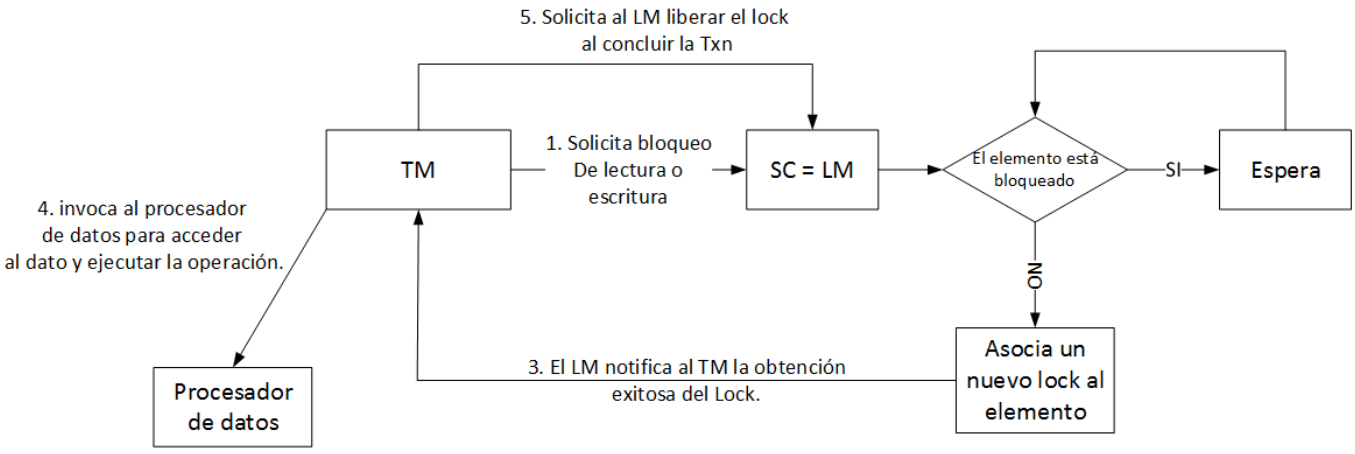
Desventajas del control optimista:

- Como se puede observar, requiere programación.
- Las acciones que se deben aplicar cuando se detecte una condición "Lost Update" o "lectura no repetible" pueden ser costosas. En algunos casos, podría representar hacer un `rollback` de una gran cantidad de operaciones, y reintentarlas aumenta el costo.

A nivel general, el esquema optimista suele ser la opción más adecuada.

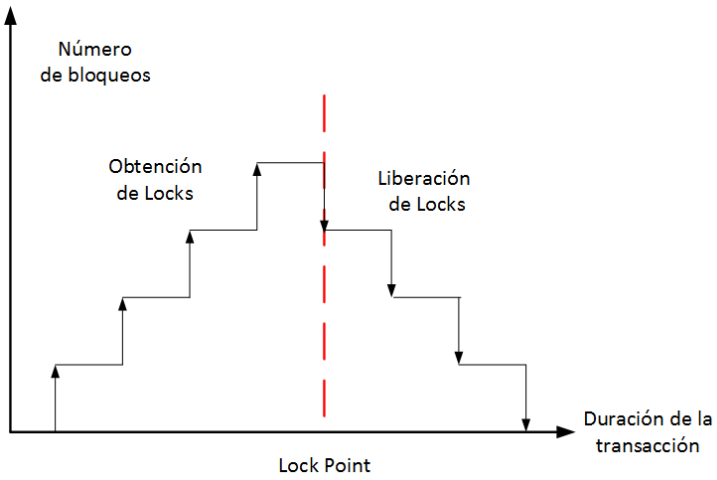
6.9. CONTROL DE CONCURRENCIA BASADO EN BLOQUEOS PARA BD DISTRIBUIDAS.

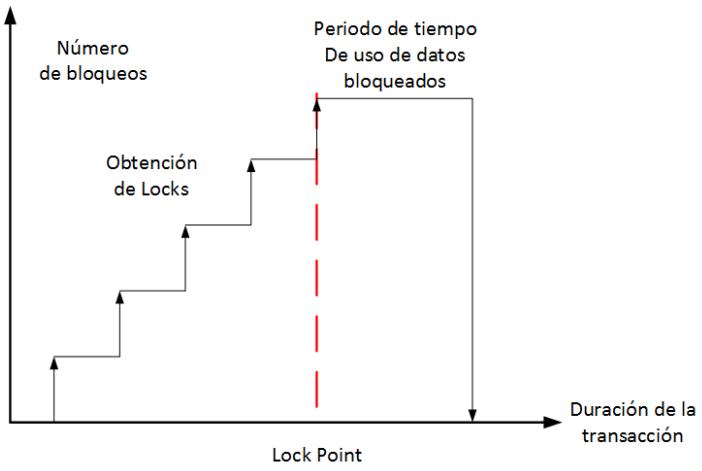
- En una BDD se requiere un control adicional para realizar la administración de bloqueos.
- Existen 3 principales componentes que participan en esta tarea:
  - Administrador de transacciones (TM: Transaction manager)
    - Responsable de coordinar las operaciones requeridas para la correcta ejecución de una transacción.
  - Sincronizador (SC: Scheduler)
    - Encargado de implementar algoritmos de control de concurrencia con el objetivo de sincronizar los accesos hacia la base de datos.
  - Administrador local de recuperación (LRM: Local Recovery Manager).
    - Participa en la administración de transacciones distribuidas, en especial para cuestiones de recuperación (este componente se revisará en el siguiente tema).
- Una transacción distribuida se inicia en un sitio, y sus operaciones son administradas por el TM.
- El TM recibe comandos como `begin_transaction`, `read`, `write`, `commit`, `abort`. En una BDD estos comandos deberán propagarse y aplicarse en cada uno de los sitios.
- Típicamente el TM se comunica con SCs y con los procesadores de datos ubicados en otros sitios.
- En la estrategia basada en bloqueos, el SCs actúa como un "***Lock manager (LM)***" ya que es el encargado de verificar si una operación de lectura o escritura puede ejecutarse en ese momento. El siguiente diagrama muestra e este proceso:



6.9.1. Protocolo de bloqueos de 2 fases (2PL: Two Phase Locking).

- Complementando las características del mecanismo de control de concurrencia basada en bloqueos que aplica tanto para BD centralizadas como para BDD, existe el protocolo de bloqueos de 2 fases (2PL) que indica la forma en la que el LM debe administrar los bloqueos de una transacción:
    - Ninguna transacción debe solicitar un nuevo bloqueo posterior a la liberación de algún bloqueo otorgado anteriormente.
    - Visto de otra forma: Una transacción no debe liberar sus bloqueos hasta tener la certeza que no se requieren más.
- Las características de este protocolo permiten la ejecución de una transacción en 2 fases:
    - Fase de crecimiento: Obtiene todos los bloqueos necesarios.
    - Fase de reducción: Los bloqueos comienzan a liberarse.
    - Entre estas 2 faces existe un punto llamado "Punto de bloqueo" que marca el fin de la fase de crecimiento y el inicio de la fase de liberación. Posterior a este punto, ya no es posible adquirir o solicitar nuevos bloqueos.
- En la práctica, los bloqueos se liberan hasta que la transacción se termina ya que es complicado conocer en qué momento una transacción ya no va a necesitar más bloqueos durante su ejecución. Por otro lado, si la transacción pudiera ser capaz de liberar bloqueos antes de concluir, ¿Qué sucedería si la transacción hace rollback? R: Se tendrían que deshacer cambios aplicados a datos cuyo bloqueo ya fue liberado y que posiblemente exista un nuevo bloqueo por parte de otra Transacción. Esto representaría un problema.
- Para resolver este inconveniente, existe una variante del algoritmo llamado: ***Protocolo estricto de bloqueos de 2 fases***. La diferencia entre las 2 versiones se muestra en las siguientes imágenes (Izquierda 2PL general, derecha 2PL estricto).



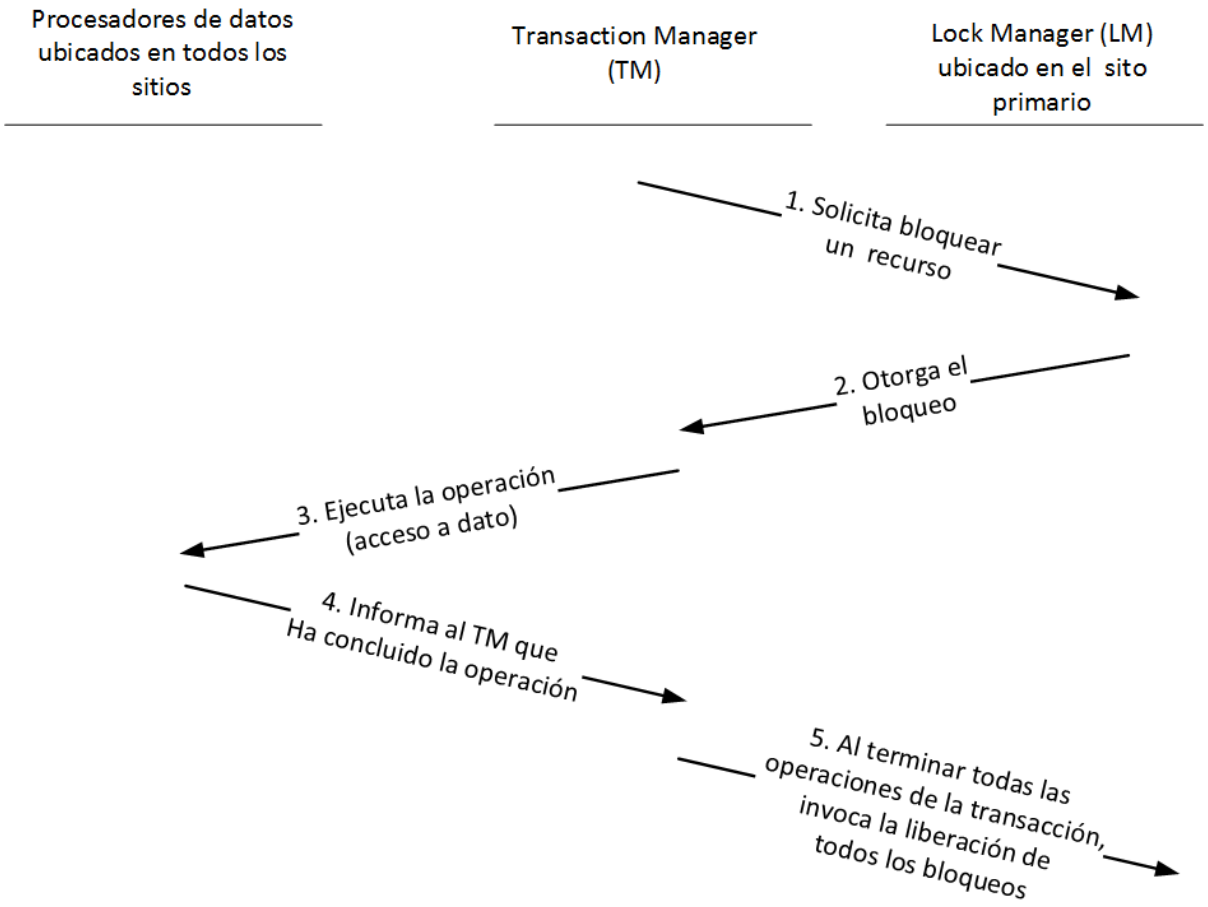


6.9.2. Protocolo de bloqueos de 2 fases en sitio primario.

- Representa la versión del protocolo anterior para BDD.
- En una BDD, los datos que requieren bloquearse se encuentran en distintos sitios. Se requiere de una administración de bloqueos distribuida.
- La idea principal es delegar la administración de los bloqueos a un solo sitio llamado "***sitio primario***".

- A este protocolo se le conoce también como "***Protocolo de 2 faces centralizado*** (C2PL)". No confundir la palabra "Centralizado" con el concepto de bases de datos centralizadas. En este caso la palabra "Centralizado" se refiere a que la administración de los bloqueos se realiza de forma centralizada en un solo sitio (el sitio primario).
- En este algoritmo la comunicación inicia en el sitio donde se origina la transacción. En este sitio se inicia un nuevo TM (transaction manager) encargado de coordinar dicha comunicación.
- En resumen, en este proceso de comunicación participan 3 componentes:
  - Transaction manager (TM) Inicia y coordina la comunicación
  - Lock Manager (LM) ubicado en el "sitio primario".
  - Procesadores de datos. Componente de cada sitio encargado de realizar el acceso a datos.

- La siguiente figura ilustra este proceso. Cada componente mostrado en la parte superior se única en un sitio diferente.

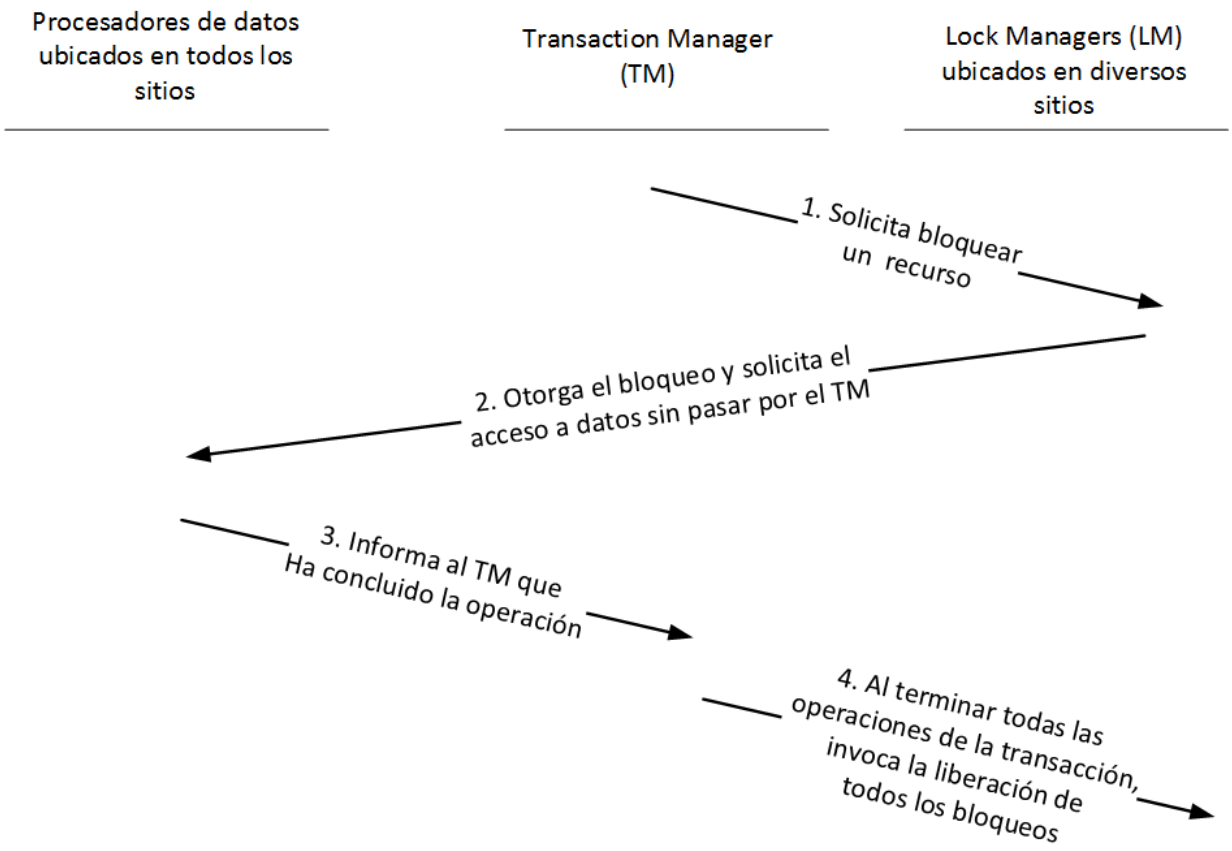


6.9.3. Protocolo de 2 fases distribuido.

- Una desventaja del protocolo de 2 fases centralizado es que pueden generarse cuellos de botella en el ***sitio primario*** sobre todo en situaciones de alta concurrencia, es decir, el LM podría caer en una situación de alta demanda de peticiones provenientes de otros sitios.
- Por otro lado, si el sitio designado como sitio primario falla o no está disponible, no habría manera de ejecutar transacciones distribuidas.
- Una alternativa a esta problemática llamado ***Protocolo de 2 fases distribuido***:

- Distribuir la administración de Lock en varios sitios. Ahora existirán varios sitios en donde se puede realizar la administración de bloqueos. Existirán varias instancias del Lock manager (LM).

- Cuando se requiere solicitar una operación de bloqueo, esta se envía a todos los sitios donde existe un LM
- Otra diferencia es que la respuesta de éxito al obtener un bloqueo ya no se envía al TM. Ahora el LM es el encargado de enviar la petición de acceso a datos a los procesadores, liberando al TM de hacer esta solicitud.
- Observar las diferencias de la siguiente imagen con la anterior.



6.10. CONTROL DE CONCURRENCIA BASADO EN TIEMPOS PARA BD DISTRIBUIDAS.

- A diferencia del mecanismo de bloqueos esta técnica no realiza una serialización a través del uso de bloqueos exclusivos.
- La serialización en esta técnica se realiza al principio (antes de iniciar la transacción) para posteriormente ejecutar las operaciones ya ordenadas.
- Para establecer este orden, a cada transacción T_i se le asocia un tiempo único (timestamp) que representa el instante en el que inicia: $ts(T_i)$. Este timestamp es asignado por el transaction manager (TS).
- En una BDD cada sitio genera sus propios valores, y para evitar coincidencias entre sitios, se emplea la pareja $(valor\ local, identificdor\ del\ sitio)$.

- A nivel general este algoritmo emplea la siguiente regla para determinar la operación que debe ejecutarse primero:

6.10.1. Regla de ordenamiento: Timestamp Ordering (TO)

Dadas 2 operaciones en conflicto O_{ij} y O_{kl} que pertenecen a las transacciones T_i y T_k , O_{ij} se ejecutará primero si $ts(T_i) < ts(T_k)$. Bajo esta condición a T_i se le conoce como la ***transacción vieja*** (inicia antes) y a T_k se le conoce como la ***transacción nueva***.

- Si llega una nueva operación que pertenece a una Txn con un timestamp(ts) mayor o totalmente nuevo con respecto a los existentes (transacciones que iniciaron antes) , la nueva operación se acepta.

Ejemplo:

$\boldsymbol{ts(O_{i5}) = 5}$

$ts(O_{i4}) = 4$
$ts(O_{i3}) = 3$
$ts(O_{i2}) = 2$
$ts(O_{i1}) = 1$

- En este caso $\boldsymbol{ts(O_{i5})}$ se acepta ya que su timestamp (ts) es el mayor de todos. Es decir, llegó al último, llegó en orden.
- De lo contrario, la operación se rechaza, su correspondiente transacción se reinicia asignándole un nuevo ts tantas veces como se requiera hasta que su timestamp sea el mayor o más joven con respecto a las operaciones existentes.
- En la práctica estas operaciones pueden llegar al Sincronizador (Scheduler) en un orden diferente. Para verificar si una operación llega en un orden diferente se emplea la siguiente consideración:
- A cada dato x que se desea acceder se le asocian 2 timestamps:
  - $rts(x)$ Corresponde al timestamp más grande o más joven de una transacción que hizo una lectura sobre x
  - $wts(x)$ Corrresponde al timestamp más grande o más joven de una transacción que hizo una escritura sobre x
- Para validar si una operación se acepta o se rechaza, el scheduler realiza las siguientes operaciones:
- Para una lectura $R_i(x)$:
  - Si $ts(T_i) < wts(x)$, T_i intenta leer el valor de x que fue modificado posterior a su inicio, => El scheduler aborta a T_i
    - Esta situación representaría un ejemplo de lectura no repetible. Por tal razón se debe abortar. El valor de X fue modificado por una transacción mas reciente.
  - Si $ts(T_i) \geq wts(x)$, T_i inició después de la última actualización de x, => El scheduler la acepta.
    - T_i puede leer el valor de x ya que el último cambio que se le aplicó ocurrió antes del inicio de T_i

- Las comparaciones con $rts(x)$ no son necesarias ya que 2 operaciones de lectura concurrente no causan conflicto

- Para una escritura $W_i(x)$
  - Si $ts(T_i) < rts(x)$, T_i intenta modificar un valor de x que fue leído posterior al inicio de T_i , => El scheduler la rechaza.
    - Otra forma de entenderlo es: X fue leída por otra transacción anteriormente, por lo tanto, X no debe ser actualizada.
  - Si $ts(T_i) \geq rts(x)$, T_i inició después de la última lectura de x , => El scheduler la acepta.
    - Con esta validación, el scheduler protege a otras transacciones que los valores leídos no serán
  - Si $ts(T_i) < wts(x)$, T_i intenta modificar un valor que fue a su vez modificado posteriormente, => El scheduler la rechaza.
    - Otra forma de entenderlo es: T_i intenta escribir un valor viejo de X, hay una escritura más reciente.
  - Si $ts(T_i) < rts(x)$, T_i intenta modificar un valor que fue a su vez leído posteriormente, => El scheduler la rechaza.
  - Si $ts(T_i) \geq wts(x)$, T_i inició después de la última actualización de x , => El scheduler la acepta.
  - Si $ts(T_i) \geq rts(x)$, T_i inició después de la última lectura, => El scheduler la acepta.

6.10.2. Algoritmo de ordenamiento por tiempos básico: TO básico.

- Básicamente representa la implementación de la regla de ordenamiento TO aplicada a BDD.
- Como se mencionó anteriormente, el Scheduler reiniciará una transacción si una Transacción más joven que la actual ha sido aceptada. Por ejemplo, si T1 con ts = 4 fue aceptada y llega T2 con ts = 1, T2 es más vieja , T2 es reiniciada.
- Si una operación es rechazada por el Scheduler, la Txn se reinicia asignándole un nuevo timestamp. Esto tiene como efecto que las operaciones de la Txn pudieran aceptarse en el siguiente intento.

Ventaja:

- Al no existir bloqueos se elimina la posibilidad de dead locks.

Desventaja:

- El costo de reiniciar una Txn puede ser elevada, sobre todo si al momento de obtener un rechazo de alguna operación, ya se contaban con un número importante de operaciones previas realizadas, lo que implica un alto costo reiniciar o deshacer cambios.
- El scheduler debe esperar a que el procesador de datos termine de procesar la actual operación antes de enviar una nueva, en especial si ambas operaciones tienen conflicto. Por ejemplo: 2 escrituras. Esto se realiza para garantizar el orden de acceso. Se emplea una cola de operaciones para garantizar este orden.

6.10.3. Algoritmo de ordenamiento por tiempos conservativo: TO conservativo.

- La diferencia con el algoritmo anterior es la reducción de la probabilidad para reiniciar una transacción.
- Una desventaja del algoritmo anterior es que el número de transacciones que se reinician puede ser elevado ya que trata de ejecutar las operaciones conforme van llegando, es decir, la ejecución de operaciones es "***agresiva***".
- Para minimizar esta condición, el algoritmo hace uso de un "buffer" o "cola de peticiones". En dicho buffer se almacenan las peticiones ordenadas por el valor "timestamp" de su correspondiente transacción. Las operaciones se agregan a este buffer por lo menos hasta que se pueda establecer un orden de ejecución.
- Este ordenamiento permite reducir el reinicio de una Txn.

Ventaja:

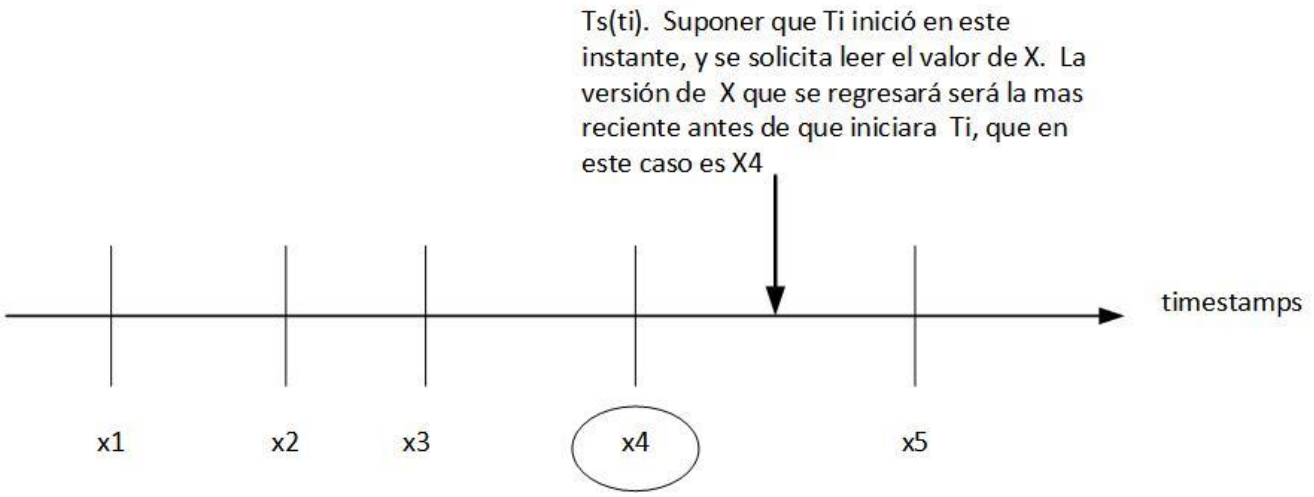
- Reduce o elimina por completo el rechazo de transacciones.

Desventaja:

- Como consecuencia de incorporar un tiempo de espera para ejecutar operaciones, se puede caer en el riesgo de DeadLocks.

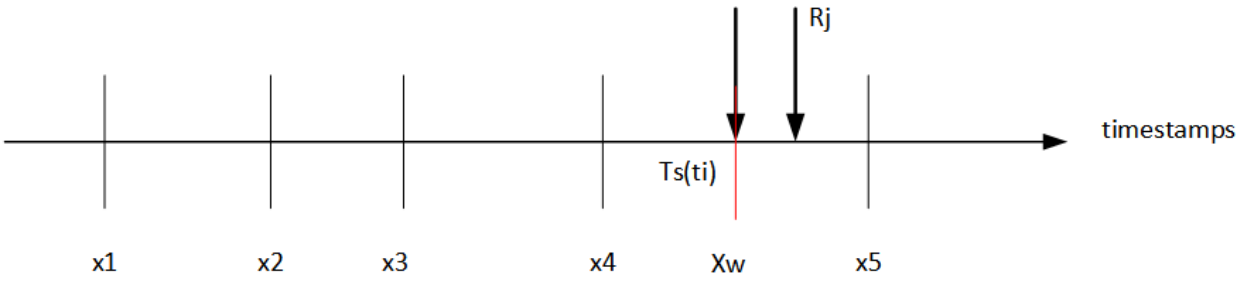
6.10.4. Algoritmo de ordenamiento por tiempos multi-versión TO multi-versión

- En este algoritmo, las operaciones de escritura no modifican la base de datos En su lugar se crean versiones de los datos. Por ejemplo, para un dato x , pueden existir versiones x_1 , x_2 , x_3 , … , x_n
- Una operación de lectura realizada por una transacción T_i siempre es exitosa. Funciona de la siguiente manera:
- $x = x_v$, donde x_v corresponde a la versión más reciente $wts(x)$ que fue escrita antes del inicio de la transacción T_i, es decir $ts(x_v)$ tiene el mayor valor anterior a $ts(T_i)$



- Para una operación de escritura, $w_i(x)$ invocada por una transacción T_i produce una nueva versión de x llamada x_v en el que $ts(x_v) = ts(T_i)$
- La escritura puede ser rechazada si se detecta una lectura realizada posterior al tiempo $ts(T_i)$.

Observar que en el tiempo ts(Ti) se produce una nueva versión del dato x llamada Xw.
Sin embargo existe una lectura Rj que se realizó después.
Originamente Rj obtuvo el valor X4 por ser la más reciente.
Al agregar esta nueva versión, Rj debió haber leído Rj ya que es más reciente que X4. Esto provocaría una incosistencia y por tal razón, la escritura se rechaza.



Ventajas:

- No se requiere aplicar ordenamiento ni bloqueo de operaciones. La existencia de versiones permite ejecutar operaciones sin preocuparse de la concurrencia o el ordenamiento.

Desventajas:

- Espacio de almacenamiento de las versiones.
- En cierto momento dichas versiones se tienen que consolidar y desecha una vez que se detecte que no se requiere acceder a versiones anteriores.

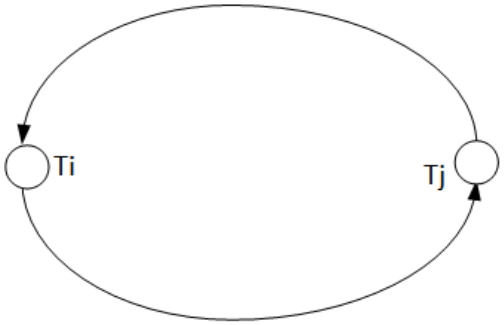
6.11. ADMINISTRACIÓN DE DEADLOCKS

- Un deadlock ocurre cuando varias transacciones esperan mutuamente a que concluyan o liberen sus bloqueos adquiridos.
- Los algoritmos basados en bloqueos y los que requieren esperar a que una transacción concluya, puede caer en un estado de dead lock.

6.11.1. Graficas Wait for Graph (WFG)

- Útil para identificar un deadlock:

Cuando Tj debe esperar a que termine Ti se traza un arco que sale de Tj hacia Ti

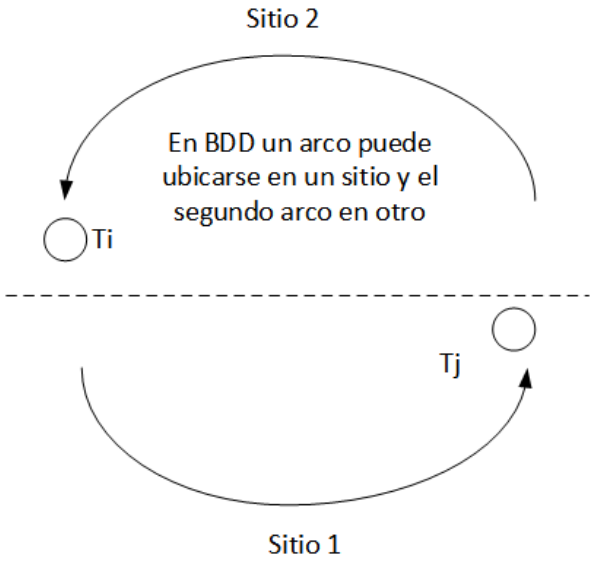


Cuando Ti debe esperar a que termine Tj se traza un arco que sale de Ti hacia Tj
Si la gráfica se cierra, se produce un deadlock!

Ejemplo:

Tiempo	Txn1	Txn2	Observaciones
1	`update emp set puesto_id = 1 where emp_id = 100`		Adquiere lock para el registro emp_id =100
2		`update emp set puesto_id = 1 where emp_id = 200`	Adquiere lock para el registro emp_id = 200
3	`update emp set puesto_id = 1 where emp_id = 200`		Txn1 se bloquea ya que el lock para emp_id =200 fue adquirido por Txn2, se traza un arco de Txn1 a Txn2
4		`update emp set puesto_id = 1 where emp_id = 100`	Txn2 se bloquea ya que el lock para emp_id = 100 fue adquirido por Tnx1 en T1 Se traza un arco de Txn2 a Txn1 Y en este momento se genera Deadlock.

- En BDD se emplea el concepto de ***Gráfica global de deadlocks***.
- La gráfica se puede cerrar con arcos que se producen en diferentes sitios.
- Existen algoritmos que permiten detectar la formación de estas gráficas, en especial cuando ocurren en diferentes sitios:



- Como se puede observar en la figura anterior, en BDD la detección de deadlocks se realiza a partir de la unión de las gráficas locales de cada sitio.
- Lo anterior implica que los sitios deben comunicarse entre si para intercambiar o compartir sus gráficas locales.

6.11.2. Métodos para realizar el manejo de deadlocks

- Prevención
- Anulación
- Detección y solución.

6.11.2.1. Prevención

- Esta técnica garantiza que nunca van a ocurrir.
- Se realiza una validación antes de iniciar una transacción. Si se comprueba que no existirán deadlocks, la transacción inicia, de lo contrario, se espera hasta que existan las condiciones adecuadas para su ejecución.
- Todos los recursos que requiere la transacción deben estar libres para poder iniciar.

Ventajas:

- No existe la necesidad de aplicar una operación de `rollback` cuando se detecte un deadlock debido a que estos nunca ocurrirán.
- No se requiere de un mecanismo externo de detección y solución para resolver deadlocks.

Desventajas:

- Reduce concurrencia
- Complejidad para determinar los recursos que va a emplear una transacción.

6.11.2.2. Anulación

- En esta técnica los deadlocks se detectan antes de ocurrir.
- Las transacciones inician. Cuando una de ellas solicita un recurso ocupado se aplican ciertas acciones para evitar el deadlock. Existen 2 principales técnicas:
  - ***Ordenamiento de datos y sitios***. Bloqueos solo pueden ser otorgados en dicho orden.
  - ***Priorización de transacciones***. Se emplea el valor timestamp (ts) de cada transacción. Asumir 2 transacciones Ti, Tj, suponer que Ti solicita un recurso bloqueado por Tj. Tradicionalmente Ti deberá esperar a que termine Tj, pero en esta técnica se aplica una validación para evitar la ocurrencia de un deadlock. Si la validación es exitosa, Ti espera, y si no es exitosa, Tj hace rollback.
    - Regla WAIT-DIE: $si\ ts(T_i) < ts(T_j),\ T_i\ espera;\ en\ otro\ caso, T_i\ hace\ aborta\ y\ se\ reinicia\ con\ el\ mismo\ timestamp.$ Este reinicio permite intentar N veces hasta que Tj libere el recurso.
    - Regla WOUND-WAIT: $si\ ts(T_i) < ts(T_j),\ T_j\ aborta\ y\ reinicia;\ en\ otro\ caso, T_i\ espera.$

Ventajas:

- No es necesario determinar la lista de recursos a ocupar antes de que inicie.
- No ocurrirán deadlocks.

Desventajas

- Requiere soporte para manejo de timestamp para priorizar transacciones.

6.11.2.3. Detección y solución

- Se permite que las transacciones inicien
- No existe validación alguna para evitar deadlocks. Se permite su generación.
- Se revisan las gráficas globales para detectar la ocurrencia de un deadlock
- Si se detecta un deadlock, se aborta una de las 2 transacciones.
- Representa la técnica que comúnmente se emplea por los manejadores.

Ventajas:
- Permite un mayor nivel de concurrencia.
- Es el método más popular y estudiado en la práctica.

Desventajas
- Costo de aplicar `rollback` a una de las 2 transacciones, puede llegar a ser elevado.

6.11.3. Técnicas para detectar deadlocks en BDD

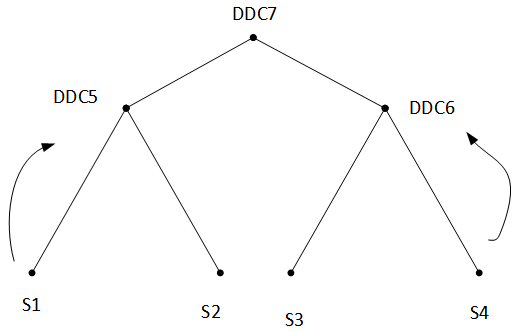
- Derivado a que la técnica de detección y solución es la más común y la mejor implementada. Existen 3 principales algoritmos de detección de deadlocks en BDD:
  - Detección centralizada (no confundir con BDs centralizadas)
  - Detección jerárquica
  - Detección distribuida.

6.11.3.1. Detección de deadlocks centralizada

- En esta técnica un solo sitio es designado como ***detector de deadlocks centralizado (DDC)***. De aquí el nombre de detección centralizada (No confundir con BD centralizada).
- Cada Scheduler envía sus gráficas locales al DDC.
- En el DDC se realiza la unión de gráficas locales y determina la existencia o formación de deadlocks.
- Si se detecta un deadlock, el DDC lo rompe seleccionando una de las 2 transacciones para hacer rollback.
- Esta técnica resulta adecuada si el control de transacciones emplea también una administración centralizada.
- Esta técnica tiene el inconveniente del único punto de falla y del clásico cuello de botella al existir un solo sitio que actúa como DDC.

6.11.3.2. Detección de deadlocks por jerarquía

- En esta estrategia los sitios son organizados en una jerarquía.
- Cada sitio envía su gráfica al sitio padre que le corresponde en la Jerarquía.
- En cada sitio padre se realiza la detección. Es decir, en esta técnica existen múltiples DDCs
- Reduce la dependencia de un solo sitio.



6.11.3.3. Detección distribuida de deadlocks

- En esta técnica todos los sitios cooperan en la detección
- Las gráficas globales se forman en cada sitio y se propagan a los demás.
- Deadlocks locales se manejan en el mismo sitio.
- Si se detectan ciclos en otros sitios, se reporta un posible deadlock global.