6. TRANSACCIONES Y CONTROL DE CONCURRENCIA DISTRIBUIDO

6.1. Transacciones antecedentes.

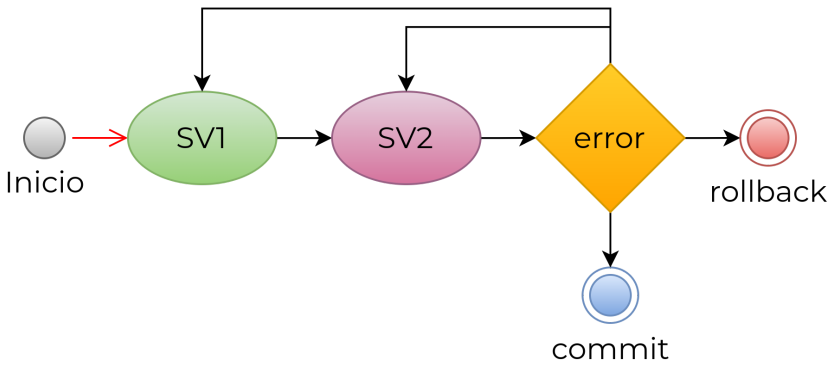
6.1.1. Definición de transacción.

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

6.1.2. Instrucciones de control.

- 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;
```



6.1.3. Propiedades ACID de las transacciones

Propiedad	Descripción
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.
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***.
Aislamiento	• Es la propiedad que requiere cada Txn para ver a la BD consistente en cualquier instante de tiempo. • Define cómo y cuándo los cambios producidos por una Txn se hacen visibles para otras transacciones en ejecución. El grado de visibilidad

Propiedad	Descripción
	depende del ***nivel de aislamiento*** configurado. En la siguiente sección se revisan estos niveles.
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. Esta garantía se cumple siempre y cuando la BD sea configurada de forma adecuada empleando los mecanismos de recuperación a fallas.

6.1.4. 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.

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**.

Modo de bloqueo	Descripción
Exclusivo	• Previene que el recurso sea compartido. Este modo se adquiere cuando una transacción modifica datos. • La primera transacción en acceder al recurso es la única que puede modificarlo. El bloqueo se libera hasta que la transacción termina.
Compartido	• Permite que el recurso sea compartido dependiendo el tipo de operación (lecturas).

	• Múltiples usuarios pueden leer el mismo dato. • Múltiples transacciones pueden adquirir un bloqueo compartido sobre el mismo recurso.

Resumen, niveles de aislamiento.

Problema	Nivel que lo resuelve	descripción
Lecturas sucias	*Lecturas confirmadas* Sin embargo, permite la ocurrencia de lecturas no repetibles y lecturas fantasma. `set transaction isolation level read committed`	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 T1 hace rollback, T2 tendrá un valor que ***nunca existió*** en la BD.
Lecturas no repetibles	*Lecturas repetibles* Sin embargo, permite la ocurrencia de lecturas fantasma. `set transaction isolation level serializable`	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.
Lecturas fantasma	*Lecturas serializables* `set transaction isolation level serializable`	Se realiza una consulta con un cierto predicado para 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 cual puede generar problemas.

Para los siguientes ejemplos:

Discutir lo que pasaría al intentar ejecutar cada instrucción con base a los tiempos T0, .., Tn, indicar el tipo de bloqueo que se genera y el resultado de cada instrucción.

Ejemplo 1
Considerar los siguientes datos.

PROD

prod_id	cantidad
1001	300
1002	500
1003	700

T	Sesión 1	Sesión 2
0	`update prod set cantidad = 100 where prod_id = 1001;`	
1	`commit;`	
2		`select cantidad from prod where prod_id=1001;`
	inicia otra transacción: Txn3	
	`update prod set cantidad = 150 where prod_id = 1002;`	
3		`select cantidad from prod where prod_id=1002;`
4	`rollback;`	
5		`select cantidad from prod where prod_id=1002;` • ¿Qué se obtiene? • ¿Qué valor se obtendría si txn3 hubiera hecho `commit`? • ¿Qué ocurre si se permitieran lecturas sucias?

Ejemplo 2

Considerar los siguientes datos.

PROD

prod_id	cantidad
1001	100

T	Sesión 1	Sesión 2
0	`set transaction isolation level serializable;`	
1	`select cantidad from prod where prod_id = 1001;`	
2		`update prod set cantidad = 35 where prod_id = 1001;`
3	`select cantidad from prod where prod_id = 1001;`	

T	Sesión 1	Sesión 2
4		`commit;`
5	`select cantidad from prod where prod_id = 1001;`	
6	`commit;`	
7	`select cantidad from prod where prod_id = 1001;`	

- ¿Qué efecto produce la cláusula `isolation level serializable`?

Ejemplo 3:
Considerar los siguientes datos.

PROD

prod_id	cantidad
1001	35

T	Sesión 1	Sesión 2
0	`set transaction isolation level serializable;`	
1	`select * from prod where cantidad = 35;`	
2		`insert into prod values(1006,35);`
3		`commit;`
4	`select * from prod where cantidad = 35;`	
5	`commit;`	
6	`select * from prod where cantidad = 35;`	

¿Qué efecto produce la cláusula `isolation level serializable`?

6.1.5. Control de concurrencia.

A pesar de existir estos niveles de aislamiento pueden ocurrir otros problemas cuando 2 o más transacciones concurrentes intentan acceder a un mismo dato.

Ejemplo 1
Considerar las siguientes condiciones iniciales:

PROD

PROD_ID	CANTIDAD
1001	5

Suponer la siguiente secuencia de eventos que ocurren entre 2 transacciones Txn1 y Txn2, considerar el nivel de aislamiento por default en Oracle: Read Commited.

T	Sesión 1	Sesión 2
0	`select * from prod where prod_id =1001;`	
1		`select * from prod where prod_id = 1001;`
2	`update prod set cantidad = 50 where prod_id = 1001;`	
3		`update prod set cantidad = 20 where prod_id = 1001;`
4	`commit;`	
5	¿Qué evento ocurrirá en este tiempo, Inmediatamente después de que Txn1 hizo `commit`?	
6		`commit;`
7	`select cantidad from prod where prod_id = 1001;`	
8		`select cantidad from prod where prod_id = 1001;`

- En este ejemplo ha ocurrido un problema llamado ***lost update*** ¿por qué razón y en dónde?
- ¿Cómo solucionar este problema?

Existen 2 técnicas empleadas:
- Control de concurrencia pesimista.
- Control de concurrencia optimista.

Ejemplo
Aplicar ambas técnicas para el siguiente escenario: Asignación de boletos.

CONCIERTO

num_asiento	ocupado	nombre
1	0	
2	0	
3	0	

Suponer que 2 clientes compiten por ganar un mismo asiento.

T	Sesión 1	Sesión 2
0	`select ocupado from concierto`	

T	Sesión 1	Sesión 2
	`where num_asiento=1;`	
1		`select ocupado from concierto` `where num_asiento=1;`

2	`update concierto`
	`set ocupado = 1, nombre ='cliente1'`
	`where num_asiento=1;`	

3	`commit;`	

4		`update concierto`
		`set ocupado = 1,nombre ='cliente2'`
		`where num_asiento=1;`

5	`select *`
	`from concierto`
	`where num_asiento = 1;`	

6		`commit;`

7		`select *`
		`from concierto`
		`where num_asiento = 1;`

- Al ejecutar esta secuencia de instrucciones, tanto la Txn1 como la Txn2 habrán adquirido el asiento número 1. Evidentemente hay un problema de ***lost update***. La sentencia `update` de Txn2 en T4 sobrescribe el cambio realizado por Txn1 en T2.

6.1.5.1. Control de concurrencia pesimista.

- En el esquema pesimista, la bd bloquea tanto lecturas como escrituras de una transacción que está procesando un registro.
- Este comportamiento se logra haciendo uso de la instrucción `select for update`.
- Desventaja: serialización total de la base de datos. Txn2 no podrá ni leer ni escribir al registro reservado por Txn1 hasta que sea liberado.
- La reservación o bloqueo del registro se aplica desde la primera lectura del registro.

T	Sesión 1	Sesión 2

0	`select ocupado from concierto`
	`where num_asiento=1`
	`for update;`	

1		`select ocupado from concierto`
		`where num_asiento=1`
		`for update;`

- Bajo este esquema, cuando Txn2 intente realizar la consulta para revisar si el asiento está disponible, la sesión entrará en modo de espera ya que Txn1 ha reservado el mismo registro para ser modificado previamente.
- Hasta que Txn1 haga `commit`, Txn2 podrá reservar el registro y ejecutar la consulta.

- Debido a que Txn1 actualizó el registro en T2 y al hacer `commit`, Txn2 reanuda su operación, la consulta se realiza y obtendrá como resultado que el asiento ha sido ocupado por `'cliente1'`, es decir, obtendrá `ocupado = 1`.
- El resultado anterior permite a Txn2 percatarse que el asiento ya fue ocupado por alguien más y por lo tanto ya no deberá sobrescribir el cambio realizado por Txn1.
- Esta estrategia resuelve el problema de un **lost update** pero el costo es muy elevado: transacciones que intenten consultar o escribir sobre el mismo registro serán bloqueadas desde el inicio.

6.1.5.2. Control de concurrencia optimista.

- Esta es la técnica comúnmente empleada ya que no realiza bloqueo de sesiones desde el inicio.
- El esquema se llama optimista ya que asume que no existirán problemas de concurrencia por lo que permite que las transacciones lean los registros de forma concurrente.
- Para validar que una transacción no sobrescriba un dato previamente actualizado por otra, se realiza una validación justo al intentar realizar la actualización del registro. Esto se logra agregando una condición en la sentencia `update` que permita verificar si un dato ha cambiado o si ha sido modificado por otra transacción.
- Para el ejemplo, bastaría con comprobar si el valor del campo ocupado ha cambiado al valor 1. De ser así, la sentencia `update` regresaría "*0 rows updated*".

T	Txn1	Txn2

2	`update concierto`
	`set ocupado = 1, nombre ='cliente1'`
	`where num_asiento=1`
	`and ocupado = 0;`	

3	`commit;`	

4		`update concierto`
		`set ocupado = 1,`
		`nombre ='cliente2'`
		`where num_asiento=1`
		`and ocupado = 0;`

- Este resultado permite que Txn2 se percate que el cambio no fue exitoso, detectando así que el dato ha sido actualizado por otra transacción.
- Dependiendo de las reglas de negocio, el tratamiento de esta condición se maneja de forma diferente. Por ejemplo, en un sistema de asignación de asientos, seguramente el sistema le notificará al cliente que el asiento seleccionado ha sido ocupado, y por lo tanto debe seleccionar otro. Para otros sistemas, el problema podría solucionarse ejecutando nuevamente la instrucción `select` esto con la finalidad de actualizar el valor del campo cuyo valor ha cambiado.

Ejemplo

Suponer que existen múltiples transacciones que desean actualizar el número de existencias en inventario de un producto que es vendido en un sitio web.

- Cada transacción es un cliente que realiza una compra. La transacción resta N número de existencias al inventario que corresponden a la cantidad de productos comprados por un cliente.
- Para actualizar las existencias, el grupo de desarrollo de software ha diseñado un procedimiento almacenado:

```
create or replace procedure disminuyeExistencias(
  p_prod_id number, p_cantidad number) is
  v_existencias number;
begin
  select existencias into  v_existencias
  from prod
 where prod_id = p_prod_id;
 --actualiza el inventario con base a la compra realizada.
 update prod
 set existencias = v_existencias - p_cantidad
 where prod_id = p_prod_id;
end;
/
```

- El código anterior tiene un problema de ***lost update***.
- Para ilustrar el problema, suponer la siguiente secuencia de eventos en la que 2 clientes Txn1 y Txn2 realizan compras en el sitio web de forma simultánea. Txn1 va a descontar 2 unidades, y txn2 va a descontar 5 unidades. En el inventario existen 10 unidades, por lo que el resultado final esperado debería ser 10-2-5 = 3.

PROD

Prod_id	Existencias
1	10

T	Sesión 1	Sesión 2
1	`exec disminuyeExistencias(1,2);` • Se invoca al procedimiento. • `v_existencias` tendrá el valor 10. • Al realizar la resta, la sentencia `update` actualizará 10-2 = 8.	
2		`exec disminuyeExistencias(1,5);` • Se invoca al procedimiento. • `v_existencias` tendrá el valor 10 ya que Txn1 aún no hace `commit`.

T	Sesión 1	Sesión 2
		• Al realizar la resta, la sentencia update intentará actualizar 10-5 = 5. • La sentencia `update` entra en modo de espera ya que no puede actualizar el registro hasta que Txn1 termine.
3	`commit;` • Actualiza las existencias con el valor 8.	
		• Justo después que txn1 hace commit, txn2 reanuda y actualiza el campo de existencias con el valor 5.
		`commit;` • ¡Al final del proceso, el dato que estará registrado en la BD será el valor 5 en lugar del valor 3!

- Para solucionar el problema, el procedimiento deberá validar que al momento de aplicar la sentencia `update`, el valor de la variable `v_existencias` debe ser el mismo para garantizar que ninguna otra transacción haya modificado su valor posterior a su lectura.

```
select existencias into v_existencias
from prod
where prod_id = p_prod_id;

--actualiza el inventario con control de concurrencia optimista
update prod
set existencias = v_existencias - p_cantidad
where prod_id=p_prod_id
and existencias = v_existencias;
```

- Si el campo `extistencias` fue modificado por otra transacción, la sentencia `update` regresaría 0 registros actualizados.
- Esta condición deberá ser detectada por el procedimiento.
- De ocurrir, significaría que el valor de `v_existencias` ya no es el más reciente, por lo tanto, su valor debe actualizarse. Es decir, la sentencia `select` debe ejecutarse nuevamente, y la resta debe recalcularse.
- Notar que, al volver a leer, puede suceder que una tercera transacción vuelva a actualizar el valor de la variable, la sentencia `update` nuevamente indicaría “0 registros actualizados” y se tendría que volver a leer, y así sucesivamente. Esto puede ocurrir si el nivel de concurrencia es alto: ¡miles de clientes están comprando un producto de súper oferta de 10 min!

- La solución: La sentencia `select`, `update` y la validación deberán estar incluidas en un ciclo que se detendrá cuando la sentencia `update` sea exitosa.

La versión correcta del procedimiento almacenado que el equipo de sistemas debió haber desarrollado es:

```
create or replace procedure disminuyeExistencias(
  p_prod_id number, p_cantidad number) is
  v_existencias number;
  v_actualizados number;
begin
  loop
    select existencias into  v_existencias
    from prod
    where prod_id = p_prod_id;
    --actualiza el inventario con base a la compra realizada.
    update prod
    set existencias = v_existencias - p_cantidad
    where prod_id = p_prod_id
    and existencias = v_existencias;
    v_actualizados := sql%rowcount;
    if v_actualizados > 0 then
      exit;
    else
      --no se sale del ciclo, reintenta.
      null;
    end if;
  end loop;
end;
/
```

Ventajas del control 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.

6.1.6. Clasificación de las transacciones.

- Existen varios criterios para clasificar a una transacción.

6.1.6.1. Clasificación de las transacciones por su duración.

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

6.1.6.2. 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;
```

6.1.7. 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`.
- 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

- Independientemente 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 autónoma
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
```

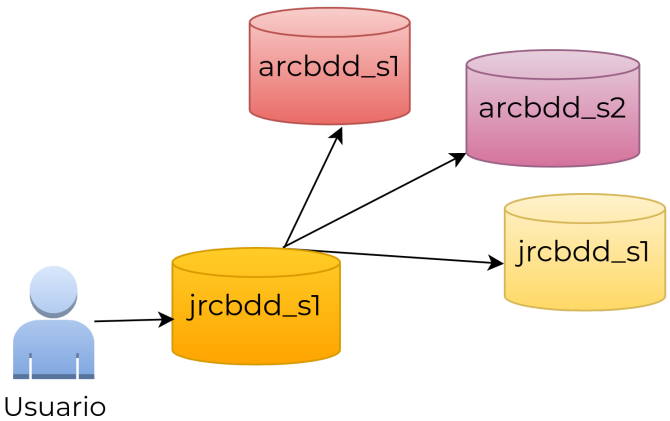
- Observar en el código anterior que a pesar de haber hecho `rollback` en la transacción principal, los datos de la tabla `operacion_usuario` se conservan.

6.2. Transacciones distribuidas.

Formada por una o más sentencias que de forma individual o en conjunto actualizan datos de 2 o más nodos que forman parte de una BDD.

Ejemplo:

```
begin
  --local
  update aqencia_1 set clave ='c1'
  where agencia_id = 1;
  --remota
  update agencia_2 set clave ='c2'
  where agencia_id = 2;
  --remota
  update cliente_1
  set num_tarjeta ='55893'
  where cliente_id = 12;
```



```
  --remota
  update cliente_2
  set email ='m@m.com'
  where cliente_id = 15;
  --confirma transacción distribuida
  commit;
exception
  when others then
  --manejo del error
  rollback;
  raise;
end;
```

- En este caso la transacción distribuida inicia en el nodo `jrcbd_s1`, realiza un cambio local e invoca un cambio en los otros 3 nodos
- Si todas las sentencias SQL de la transacción apuntan a un mismo nodo, entonces a la transacción se le conoce como transacción ***remota*** en lugar de ser ***distribuida***.

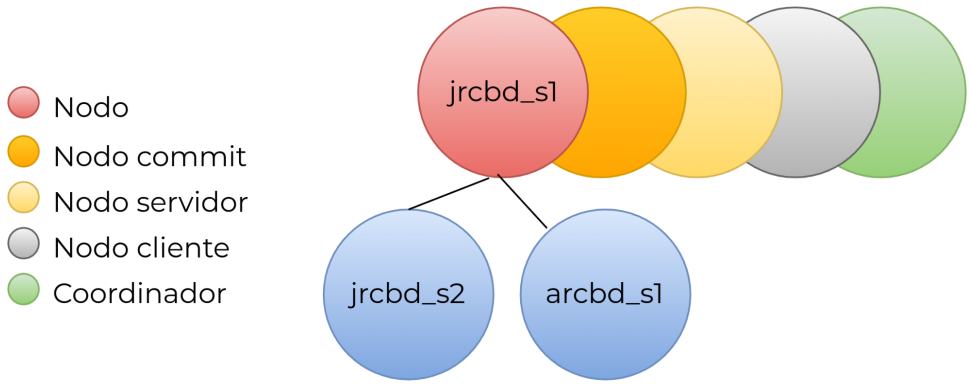
6.2.1. DML y DDL en una transacción distribuida.

- Las siguientes operaciones DML y DDL son permitidas en una transacción distribuida (Oracle)

```
create table as select
delete
insert
update
lock table
select
select for update
```

6.2.2. Árboles de sesiones en transacciones distribuidas.

- Un árbol de sesión representa un modelo jerárquico que describe las relaciones entre un conjunto de sesiones y los roles que puede desempeñar durante la ejecución de una transacción distribuida.
- Cuando una Txn distribuida inicia, la BD define o construye un árbol de sesiones formado por todos los nodos que participan en la transacción.



En el ejemplo anterior, el sitio `jrcbd_s1` tiene asignados 4 roles. Los otros 2 nodos, solo tienen un rol asignado. La txn inicia en `jrcbd_s1`.

Rol	Descripción
Cliente	Nodo que hace referencia a datos que pertenecen a otro nodo. En el ejemplo `jrcbd_s1` es cliente ya que la txn que inicia en ese nodo modifica datos que pertenecen a los otros 2.
Nodo servidor	Nodo que recibe una petición de otro nodo para realizar alguna operación con sus datos.
Coordinador global.	El nodo que coordina la ejecución. Corresponde al nodo donde la txn se origina. Representa al nodo raíz del árbol de sesiones.
Nodo commit	Nodo en el que se realiza `commit` o `rollback`. Estas operaciones son coordinadas por el coordinador global.
Coordinador local	Nodo que recibe una petición de otro nodo para realizar alguna operación. La diferencia con un nodo servidor es que a su vez este nodo necesita hacer referencia a otro nodo para poder completar la tarea asignada. Notar que `jrcbd_s1` actua tanto como coordinador global como local ya que coordina toda la txn pero también recibe instrucciones para realizar cambios que a su vez requiere la ayuda de otros nodos.

6.2.3. Nodo commit

- Su principal tarea es iniciar con el proceso de `rollback` o `commit` ordenado por el coordinador global.
- El nodo seleccionado para ser el ***nodo commit*** es el que guarda la información más crítica.
- Este nodo nunca entra en el estado de “Preparación”.
- Los datos de este nodo nunca están en el estado “In-doubt” aunque ocurra una falla.
- Cuando ocurre una falla en ciertos nodos, estos permanecen en el estado de “Preparación” conservando los ***locks*** necesarios sobre los datos hasta que las transacciones “in doubt” o “en duda” sean resueltas.

- El nodo commit es el primero en hacer `commit` , los demás nodos lo siguen.
- El coordinador global se encarga de verificar que todos los nodos terminen la transacción en la misma forma que lo hace el nodo commit

6.2.4. Proceso para realizar commit en una txn distribuida.

- Una transacción distribuida se considera confirmada cuando todos los nodos que no son nodo commit están en estado de preparación y la transacción ha hecho `commit` en el nodo commit.
- El Online redo log en el nodo commit es actualizado en cuanto se hace `commit` en dicho nodo.
- Notar que puede que ya se haya hecho `commit` en el nodo commit, pero los otros nodos aún pueden estar en el estado de preparación sin haber hecho `commit`.
- La transacción distribuida se considera como no confirmada cuando no existe el `commit` en el nodo commit.

6.2.5. Commit point strength

- Este concepto se puede entender como un valor de “relevancia” o “intensidad” que se le asigna a cada nodo. El nodo que tenga el mayor valor es designado como nodo commit.
- El mayor valor es asignado al nodo que comparte la mayor cantidad de datos con otros nodos.
- Este valor se especifica a través del parámetro `commit_point_strength`. Puede ser actualizado de forma manual.
- Cuando una transacción distribuida hace `rollback`, no se necesitan las fases de preparación ni la fase de commit. En este caso el coordinador global instruye a todos los nodos ejecutar `rollback`.

6.3. Mecanismo Two-phase commit

- El procesamiento de una transacción distribuida es más complicado que una centralizada justamente por esta tarea de coordinación de actividades en cada nodo como una sola unidad de trabajo “Se hace todo o se hace nada”.
- Este mecanismo está formado por 2 fases que se emplean para garantizar la integridad de los datos.
- En la fase de ***preparación*** el coordinador global solicita a los demás nodos realizar ya sea un `commit` o un `rollback`.
- En la fase de ***commit*** el nodo que inicia la transacción solicita a todos los nodos realizar `commit`.
- Si estas 2 fases no se cumplen, se solicita a todos los nodos hacer `rollback`.

Fase	Descripción

Fase de preparación	• El nodo donde se inicia la txn (coordinador global) le solicita a todos los nodos excepto al nodo commit prepararse para aplicar ya sea un `commit` o un `rollback`.
	• Se escriben los cambios en los Online Redo logs, pero aun sin agregar la marca del `commit` (`commit record`).
	• Establece locks en tablas modificadas para prevenir ***lecturas***.
	• El nodo responde al coordinador que está preparado y se ***compromete*** a realizar el `commit` o `rollback` cuando sea requerido.
	• Esta promesa consiste en que si llegara a ocurrir una falla en el nodo, este podrá hacer un ***recovery*** para regresar al estado de ***preparación***.
	• El bloqueo sobre las tablas termina al terminar todas las fases. Si una Txn2 intenta leer o escribir sobre los mismos datos posterior al inicio de esta fase, Txn2 tendrá que esperar. Este tiempo es realmente insignificante a menos que ocurra una falla. Su duración es muy corta.
	• Tipos de respuestas:
	  ○ *Preparado*.
	  ○ *Read Only* (no se hicieron cambios, no se requiere preparación por lo que el nodo no participará en la fase `commit`).
	  ○ *Abort* (el nodo no pudo prepararse).
	• El nodo espera la señal del coordinador para realizar el `commit` o el `rollback`.

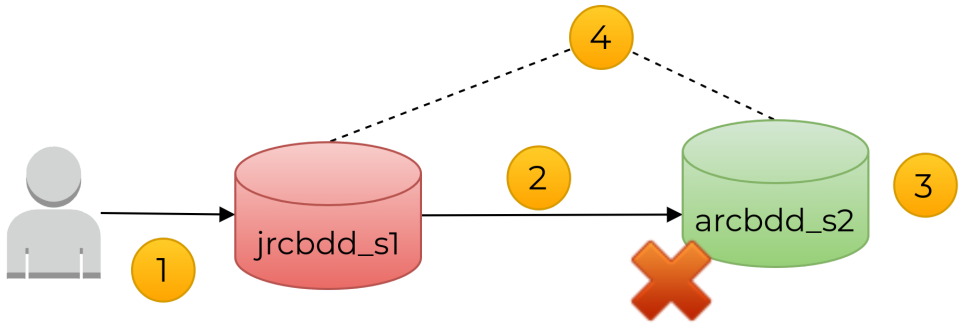
Fase de commit	• La fase de commit inicia hasta que todos los nodos diferentes al ***nodo commit*** han concluido la etapa de preparación. • En esta fase se realizan las siguientes acciones: 1. El coordinador global le instruye al ***nodo commit*** hacer commit. 2. El *nodo commit* realiza commit. 3. El ***commit site*** le notifica al coordinador global que ya ha realizado `commit`. 4. El coordinador envía mensajes a los demás nodos para que realicen commit. 5. En cada nodo se realiza el `commit` de la parte proporcional de cambios de la txn distribuida que fueron aplicados y se liberan los ***locks***. 6. En cada nodo se escriben datos adicionales en los Online Redo Logs indicando que la txn distribuida ha hecho `commit`. 7. Los nodos le comunican al coordinador global que han hecho `commit`. Al terminar esta fase la transacción se considera como concluida y el estado consistente de la base de datos se conserva.

- Cada transacción confirmada cuenta con su SCN (Sytem Change Number) que de forma única identifican los cambios realizados por sentencias SQL dentro de la transacción.
- Durante la etapa de preparación todos los sitios envían sus SCNs al coordinador. Se selecciona el de mayor valor y es el que se emplea para hacer commit en el nodo commit.
- Este SCN mayor es enviado a los demás nodos para hacer su `commit` parcial.

6.3.1. Transacciones en duda (In - doubt Transactions)

- Una transacción puede considerarse "en duda" si el mecanismo two phase commit falla por algún error no esperado: errores de sistema, errores de red, errores a nivel del software que accede a la BD.
- El proceso de background RECO es el encargado de resolver transacciones ***"in doubt"*** posterior a que la falla se haya resuelto. Mientras la transacción se encuentre en este estado, los datos son bloqueados tanto para lecturas como para escrituras. Las lecturas también son bloqueadas ya que no se sabe con certeza qué versión de los datos debe ser mostrada para una determinada consulta.

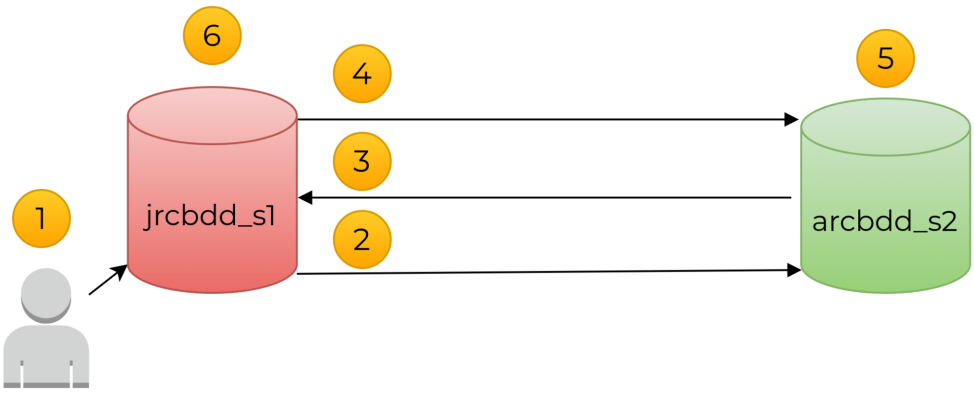
Ejemplo



1. El usuario se conecta al nodo `jrcbd_s1` forma local e inicia una transacción distribuida.
2. El coordinador global `jrcbd_s1` designado también como nodo commit, solicita a los demás nodos prepararse para hacer `commit`.
3. `arcbd_s2` falla y no envía su respuesta de preparación al coordinador global.
4. La transacción distribuida termina con una operación `rollback` cuando el nodo se recupere empleando al proceso de background RECO.

Ejemplo

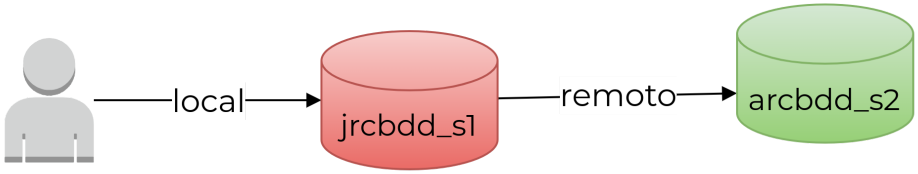
Falla en la fase commit.



1. El usuario se conecta al nodo `jrcbd_s1` forma local e inicia una transacción distribuida.
2. El coordinador global `jrcbd_s1` designado también como nodo commit, solicita a los demás nodos prepararse para hacer `commit`.
3. El nodo commit recibe la respuesta de `arcbd_s2` indicando que está preparado.
4. El nodo commit confirma la transacción localmente, e instruye a `arcbd_s2` realizar `commit`.
5. `arcbd_s2` recibe la señal para realizar commit, pero este no puede enviar la respuesta de regreso por una falla de red.
6. A pesar de este error, la transacción distribuida hace `commit`, y el proceso de background RECO realizará el `commit` una vez que la comunicación sea restablecida.

6.3.2. Procesamiento de una transacción distribuida.

La siguiente lista de pasos resume todos los eventos que ocurren cuando se procesa una transacción distribuida. Asumir que el sitio `jrcbd_s1` inicia la transacción:




```
insert into t1 values (...);
update t2@jrcbd_s2 set ...
insert into t3 values (...);
update t4@jrcbd_s2 set ...
insert into t5 values (...);
update t6@jrcbd_s2 set ...
commit;
```

1. Nodo `jrcbd_s1` ejecuta una serie de instrucciones DML locales y remotas.
    a. Se define el árbol de sesiones.
    b. `jrcbd_s1` se asigna como coordinador global.
    c. Debido a que los 2 nodos reciben instrucciones para realizar operaciones DML, son considerados nodos servidor.
    d. `jrcbd_s1` también actúa como cliente ya que hace referencia a datos de `jrbd_s2`.
    e. Este paso termina con la adquisición de todos los ***locks*** (bloqueos) necesarios en cada nodo necesarios para realizar los cambios solicitados. Estos bloqueos permanecen hasta que terminen las 2 fases.
2. Se determina el nodo commit.
    a. Este proceso inicia justo después de que el usuario ejecuta la instrucción `commit`.
    b. En este ejemplo, `jrcbd_s1` es asignado como en nodo commit.
3. El coordinador global envía mensajes a los nodos para prepararse.
    a. El coordinador global envía el mensaje de preparación a todos los nodos a los que hace referencia directa excepto al nodo commit. En este caso `jrcbd_s2`
    b. `jrcbd_s2` inicia el proceso de preparación y envía su respuesta al nodo que se lo solicitó indicando que está preparado.
    c. Dependiendo las respuestas enviadas puede ocurrir lo siguiente:
        i. Si alguno de los nodos responde con “abort”, el coordinador global ordena hacer `rollback`.
        ii. Si todos los nodos responden “Prepared”, el coordinador global ordena al nodo commit ejecutar la instrucción `commit`.
4. Nodo commit realiza `commit`.
    a. El nodo commit, en este caso `jrcbd_s1` hace commit local y guarda el commit record en su Online Redo Log. Notar que en este momento el otro nodo aun no hace `commit`.
5. Nodo commit le notifica al coordinador la ejecución de la instrucción commit.
    a. El nodo commit le comunica al coordinador que ha realizado `commit`. En este ejemplo, se trata del mismo nodo, por lo que omite esta comunicación.
6. Coordinadores globales y locales notifican a los demás nodos realizar `commit`.
    a. El coordinador global envía mensaje para hacer `commit` a todos los nodos directamente referenciados
    b. El coordinador local Instruyen a sus nodos servidor hacer `commit` y así sucesivamente.
    c. Los nodos hacen `commit` y liberan los recursos bloqueados.
7. Coordinador global y el nodo commit completan la operación `commit`.
    a. Cuando todos los nodos han hecho commit incluyendo el coordinador global, este le notifica al nodo commit el evento.
    b. El nodo commit recibe el mensaje, y elimina los datos del status actual de la transacción.
    c. Notifica al coordindor que ha terminado.
    d. El coordinador global finaliza la transacción.

6.4. Administración de transacciones distribuidas.

6.4.1. Determinar el valor del parámetro commit_point_strength.

- Se puede establecer de forma manual en el archivo de parámetros. Si valor varía entre [0,255]

```
commit_point_strength=200
```

- Si se realiza de forma manual considerar los siguientes puntos:
  - Debe ser un sitio que tenga la mejor disponibilidad
  - Debe ser el sitio que comparte la mayor cantidad de datos críticos.
  - Recordar que el nodo commit guarda información del estado de la txn distribuida y por lo tanto esta información debe estar disponible.

6.4.2. Nombrar transacciones

- Útil para transacciones distribuidas

```
set transaction name 'my distributed txn';
```

- Este nombre es incluido en las vistas `v$transaction` y en `dba_2pc_pending` cuando se realiza `commit`.

6.4.3. Mostrar información de transacciones distribuidas

El diccionario de datos de cada nodo guarda información acerca de todas las transacciones distribuidas abiertas. La vista `dba_2pc_pending` contiene el detalle de las transacciones "en duda".

Columna	Descripción
`local_tran_id`	Identificador local de la transacción con formato `integer.integer.integer` Si el valor es similar al de la siguiente columna, el nodo es el designado como el coordinador.
`global_tran_id`	Identificador global de la transacción con formato `global_db_name.db_hex_id.local_tran_id`, `db_hex_id` es una cadena hexadecimal de 8 caracteres para identificar de forma única a la base de datos.
`state`	El status de la transacción: • ***Collecting***: aplica únicamente al coordinador global, o al coordinador local. El nodo se encuentra recolectando

	información de otras bases de datos antes de decidir si el nodo puede prepararse para hacer commit o no. • ***Prepared***: El nodo está preparado, sin importar si su coordinador ha recibido el mensaje de notificación. Aún no se recibe mensaje para hacer commit, los datos permanecen bloqueados. • ***Commited***: El nodo ha hecho commit sin importar si otros nodos aún no lo han hecho. • ***Forced commited***: Transacción que fue manualmente confirmada en el nodo local por el administrador. • ***Forcerd termination (rollback)***: El administrador ha forzado un rollback en el nodo local.
mixed	Si su valor es YES, significa que la transacción ha hecho `commit` en algún nodo y también rollback en otro.
tran_comment	Comentario de la transacción. Si se emplea un nombre, este se mostrará en esta columna cuando la transacción hace `commit`.
host	Nombre de host.
commit#	Número de commit global de la transacción.

6.4.4. Administrar transacciones en duda

- Como se mencionó anteriormente, una transacción se encuentra en “duda” cuando ocurre una falla durante la aplicación del proceso commit de 2 fases.
- Es posible forzar un commit o un rollback de una transacción local “en duda”. Los siguientes mensajes de error indican un problema al durante alguna de las 2 fases para realizar commit:

```
ORA-02050: transaction ID rolled back,
  some remote dbs may be in-doubt
ORA-02053: transaction ID committed,
  some remote dbs may be in-doubt
ORA-02054: transaction ID in-doubt
```

- Generalmente la bd resuelve de forma automática estas fallas, en especial ante una intermitencia o falla temporal.
- Las siguientes condiciones pueden sugerir forzar el término de la transacción:
  - La transacción “en duda” está bloqueando datos que son requeridos por otras transacciones.
  - Una transacción “en duda” previene que las extensiones reservadas de los segmentos undo sean utilizadas por otras transacciones (no se pueden leer).
  - Si la falla se prolonga por tiempo indefinido.
- La decisión de forzar el término de la transacción debe tomarse con precaución ya que de tomar la decisión equivocada, resultará complicado regresar al estado consistente.

- El siguiente análisis puede ser considerado para tomar la decisión:
  - Determinar algún otro nodo que haya resuelto la transacción. Una vez que se ha encontrado se pueden aplicar las mismas acciones en el nodo que está "en duda". Esto se puede consultar en la vista `dba_2pc_pending`.
  - Revisar la información contenida en la columna `tran_comment` de la vista `dba_2pc_pending`. Este comentario indica el origen de la transacción y su tipo.
  - Auxiliarse del llamado transaction advisor.

6.4.4.1.Realizar commit manual de una transacción en duda

- Se requiere contar con el privilegio `force transaction`.
- Ejecutar la instrucción:

```
commit force 'transaction_id';
rollback force 'transaction_id';
```

- El id de la transacción se especifica en la columna local_tran_id o global_tran_id de la vista `dba_2pc_pending`.
- Es posible hacer uso de SCN asignado cuando la transacción hizo `commit` en otros nodos.

```
commit force 'transaction_id' scn;
```

Ejemplo

```
commit force 'sales.example.com.55d1c563.1.93.29', 829381993;
```

6.5. Simulando fallas en transacciones distribuidas.

- Para comprender el funcionamiento y la forma en la que una txn distribuida puede resolverse de forma automática o manual, existe una funcionalidad que permite simular errores empleando la siguiente sentencia al invocar el `commit` de una transacción distribuida:

```
commit comment 'ora-2pc-crash-test-n';
```

- El valor de N describe el error a similar (se conservan las descripciones en inglés).

N	Efecto
1	Crash commit point after collect
2	Crash non-commit-point site after collect
3	Crash before prepare (non-commit-point site)

N	Efecto
4	Crash after prepare (non-commit-point site)
5	Crash commit point site before commit
6	Crash commit point site after commit
7	Crash non-commit-point site before commit
8	Crash non-commit-point site after commit
9	Crash commit point site before forget
10	Crash non-commit-point site before forget

- Para poder hacer uso de estas simulaciones de error, tanto el usuario local como los usuarios remotos que están participando en la transacción distribuida deben contar con el privilegio `force any transaction`.

6.5.1. Habilitar / deshabilitar el proceso de background RECO.

- Recordando las secciones anteriores, el proceso de background RECO es el encargado de resolver transacciones en duda de forma automática.
- RECO intenta reanudar el trabajo que quedó pendiente al ocurrir una falla al transcurrir un cierto periodo de tiempo el cual se incrementa entre intentos.
- Una vez que la transacción ha sido resuelta. El registro de la transacción en duda desaparece de la vista `bda_2pc_pending`.
- Para efectos de pruebas y de aprendizaje, es posible deshabilitar al proceso RECO y permitir que la transacción sea resuelta de forma manual. Las siguientes instrucciones permiten habilitar o deshabilitar la acción automática de este proceso:

```
alter system disable distributed recovery;
alter system enable distributed recovery;
```

Nota: Para efectos del curso en el que se emplean PDBs, estas instrucciones deben ser ejecutadas en la PDB root.

6.5.2. Provocando errores en el mecanismo two phase commit..

Para ilustrar la administración de fallas durante la aplicación del protocolo two phase commit, considerar las siguientes configuraciones.

- En los ejemplos participarán 4 nodos: `jrcbd_s1`, `jrcbd_s2`, `arcbd_s1`, `arcbd_s2`.
- Considerar al usuario `control_agencia` empleado en temas anteriores.
- Considerar que existe una tabla llamada `prod` con la siguiente estructura y con los siguientes datos en los 4 nodos:

```
--crear esta tabla en los 4 nodos
create table  prod(
  id number constraint prod_pk primary key,
  nombre varchar2(20),
  existencias number
);

--insertar en los 4 nodos.
insert into prod values(1,'monitor HD1',10);
insert into prod values(2,'monitor HD2',20);
insert into prod values(3,'monitor HD3',30);
insert into prod values(4,'monitor HD4',40);
commit;
```

- Considerar que el usuario `control_agencia` se le ha otorgado el privilegio `force any transaction`.

Ejemplo 1

Crear una transacción en la que no se genere error alguno, el `commit` deberá ser exitoso. Ejecutar en `jrcbd_s1`. La transacción realizará un cambio en cada nodo.

```
--cambio local
update prod set existencias = 15 where id = 1;
--cambio remoto
update prod@jrcbd_s2 set existencias = 15 where  id = 1;
--cambio remoto
update prod@arcbd_s1 set existencias = 15 where  id = 1;
--cambio remoto
update prod@arcbd_s2 set existencias = 15 where  id = 1;
--commit distribuido
commit;
```

Ejemplo 2

Crear una transacción distribuida que realice cambios en los 4 nodos. La fase de preparación será exitosa, pero en la fase `commit` ocurrirá una falla. El coordinador global nunca recibe la señal por parte del nodo commit en la que se indica que este ha realizado `commit` de forma exitosa.
La consulta se lanza en `jrcbd_s1`.

```
update prod set existencias          =  50 where  id = 1;
update prod@jrcbd_s2 set existencias =  50 where  id = 1;
update prod@arcbd_s1 set existencias =  50 where  id = 1;
```

```
update prod@arcbd_s2 set existencias =  50 where  id = 1;
--Crash commit point site after commit
commit comment 'ORA-2PC-CRASH-TEST-6';
```

- Observar el mensaje de error que se empleará para simular el error: ***"commit point site after commit"***. El término ***commit point site*** se refiere al nodo commit, en el que ocurre una falla justo después de que se ha realizado `commit`.
- El mensaje recibido al ejecutar estas instrucciones es:

```
control_agencia@jrcbd_s1> commit comment 'ORA-2PC-CRASH-TEST-6'
*
ERROR at line 1:
ORA-02053: transaction 2.7.1113 committed, some remote DBs may be in-doubt
ORA-02059: ORA-2PC-CRASH-TEST-6 in commit comment
```

- Notar que la transacción distribuida ha sido marcada como confirmada.
- Al consultar los datos en la vista `dba_2pc_pending` en los 4 nodos se obtiene lo siguiente:

`jrcbd_s1`

	LOCAL_TRAN_ID	GLOBAL_TRAN_ID	STATE	MIXED
1	2.7.1113	JRCBD_S1.FI.UNAM.3788c56c.2.7.1113	committed	no

`jrcbd_s2`

	LOCAL_TRAN_ID	GLOBAL_TRAN_ID	STATE	MIXED	ADVICE
1	10.19.1428	JRCBD_S1.FI.UNAM.3788c56c.2.7.1113	prepared	no	

`arcbd_s1`

	LOCAL_TRAN_ID	GLOBAL_TRAN_ID	STATE	MIXED	ADVICE
1	6.0.1122	JRCBD_S1.FI.UNAM.3788c56c.2.7.1113	prepared	no	

`arcbd_s2`

	LOCAL_TRAN_ID	GLOBAL_TRAN_ID	STATE	MIXED	ADVICE
1	3.20.1275	JRCBD_S1.FI.UNAM.3788c56c.2.7.1113	prepared	no	

- Observar los valores del campo `local_tran_id`. Contiene el identificador de cada una de las transacciones locales que participan en la transacción distribuida.
- El campo `global_tran_id` muestra el identificador global de la transacción. La última parte permite identificar al nodo commit: `2.7.113` corresponde al valor del identificador de la transacción local del sitio `jrcbd_s1`. Por lo tanto, este sitio fue el designado como nodo commit. A nivel general la estructura de este campo es:
- `global_database_name.hhhhhhhh.local_transaction_id`

- Observar el campo `state`. Sus posibles valores antes que esta sea resuelta de forma manual son `prepared`, `commited`, o `collecting`. Una vez que se realiza un `commit` o `rollback` forzado, su estado puede cambiar a `forced commit` o `forced rollback`.
- Solo las transacciones en status `prepared` pueden ser consideradas para aplicar un `forced commit` o un `forced rollback`.

Al intentar acceder a los datos de la tabla desde cualquier otro sitio se obtiene el siguiente error:

```
control_agencia@arcbd_s2> select * from prod;
select * from prod;
              *
ERROR at line 1:
ORA-01591: lock held by in-doubt distributed transaction 3.20.1275
```

- El error anterior se origina debido a que el status de la transacción en este sitio es `prepared`. El nodo está listo para hacer `commit` o `rollback`, y solo está esperando el mensaje por parte de su coordinador. Sin embargo, esta notificación no llegó por el error provocado.
- Recordando los conceptos revisados, mientras la transacción se encuentre en este estado, los datos involucrados en la transacción son bloqueados tanto para leer como para escribir. La transacción deberá ser resuelta manualmente en caso que el proceso RECO no esté habilitado o la falla persista por tiempo indefinido.
- En este ejemplo, una nueva transacción está intentando leer los datos del registro involucrado en la transacción distribuida cuya transacción local está en ***"duda"***, recibe el error de bloqueo de lecturas ya que la transacción continua en status `prepared`.
- Al consultar la vista `dba_2pc_neighbors` en cada nodo se obtiene lo siguiente:

`jrcbd_s1`

	LOCAL_TRAN_ID	IN_OUT	DATABASE	DBUSER_OWNER	INTERFACE	DBID
1	2.7.1113	in	(null)	CONTROL_AGENCIA	N	(null)
2	2.7.1113	out	JRCBD_S2.FI.UNAM	CONTROL_AGENCIA	N	5108825c
3	2.7.1113	out	ARCBD_S1.FI.UNAM	CONTROL_AGENCIA	N	f99218cd
4	2.7.1113	out	ARCBD_S2.FI.UNAM	CONTROL_AGENCIA	N	607180af

`jrbd_s2`

	LOCAL_TRAN_ID	IN_OUT	DATABASE	DBUSER_OWNER	INTERFACE	DBID	SESS#
1	10.19.1428	in	JRCBD_S1.FI.UNAM	CONTROL_AGENCIA	N	3788c56c	1

`arcbd_s1`

	LOCAL_TRAN_ID	IN_OUT	DATABASE	DBUSER_OWNER	INTERFACE	DBID
1	6.0.1122	in	JRCBD_S1.FI.UNAM	CONTROL_AGENCIA	N	3788c56c

`arcbd_s2`

	LOCAL_TRAN_ID	IN_OUT	DATABASE	DBUSER_OWNER	INTERFACE	DBID	SESS#
1	3.20.1275	in	JRCBD_S1.FI.UNAM	CONTROL_AGENCIA	N	3788c56c	1

Observar los valores de la columna `in_out`.

- `in`: El nodo actúa como server (recibe peticiones).
- `out`: El nodo actual como cliente (hace referencia a datos en otro nodo).
- La columna `interface` indica si el nodo fue designado como nodo commit (C) o (N) Nodo que no fue designado como tal.

Resolución manual de la transacción.

- Al observar y analizar la salida de las vistas anteriores, los 3 nodos remotos muestran status `prepared`, el nodo commit ha hecho `commit`. La falla solo se refiere a un error de comunicación, puede aplicarse un `commit` manual en los nodos restantes. Las siguientes instrucciones deberán ser ejecutadas como `sysdba` en los nodos locales.

```
--jrcbd_s2
commit force '10.19.1428';
--arcbd_s1
commit force '6.0.1122';
--arcbd_s2
commit force '3.20.1275';
```

- Al realizar el commit forzado el status en la vista `dba_2pc_neighbors` cambia a `forced commit`.

Limpieza de vistas.

- Finalmente, ejecutar la siguiente instrucción para limpiar las vistas del diccionario de datos.

```
exec dbms_transaction.purge_lost_db_entry('2.7.1113');
exec dbms_transaction.purge_lost_db_entry('3.20.1275');
exec dbms_transaction.purge_lost_db_entry('6.0.1122');
exec dbms_transaction.purge_lost_db_entry('3.20.1275');
```

- Cuando la resolución de errores se realiza en automático empleando el proceso RECO, los registros de estas vistas son eliminadas de forma automática. La excepción ocurre cuando el valor del campo `mixed = 'y'`.
- Si la resolución de transacciones se realiza de forma manual, los datos de estas vistas deben eliminarse también de forma manual.
- Existe otro procedimiento `dbms_transaction.purge_mixed` que se emplea para realizar la limpieza de datos, en especial cuando la columna `mixed` en

`dba_2pc_pending` tiene el valor `'Y'`. Este valor indica una resolución manual de transacciones incorrecta. Algunos nodos fueron resueltos con un `commit` forzado y otros con un `rollback` forzado.

Ejemplo 3:

Provocar error en el siguiente evento: ***Crash commit point site before commit***. Lanzar en `jrcbd_s2`.

```
update prod@jrcbd_s2 set existencias =  60 where  id = 2;
update prod@arcbd_s1 set existencias =  60 where  id = 2;
update prod@arcbd_s2 set existencias =  60 where  id = 2;
commit comment 'ORA-2PC-CRASH-TEST-5';
```

El error a provocar se refiere a la ocurrencia de una falla en el nodo commit justo antes de hacer `commit`.

La salida de la vista es:

```
control_agencia@jrcbd_s2> commit comment 'ORA-2PC-CRASH-TEST-5';
commit comment 'ORA-2PC-CRASH-TEST-5'
*
ERROR at line 1:
ORA-02050: transaction 1.12.960 rolled back, some remote DBs may be in-doubt
ORA-02059: ORA-2PC-CRASH-TEST-5 in commit comment
```

`jrcbd_s2`

	LOCAL_TRAN_ID	GLOBAL_TRAN_ID	STATE	MIXED	ADVICE
1	1.12.960	JRCBD_S2.FI.UNAM.5108825c.1.12.960	collecting	no	

`arcbd_s1`

	LOCAL_TR...	GLOBAL_TRAN_ID	STATE	MIXED	ADVICE
1	10.23.1444	JRCBD_S2.FI.UNAM.5108825c.1.12.960	prepared	no	

`arcbd_s2`

	LOCAL_TRAN_ID	GLOBAL_TRAN_ID	STATE	MIXED	ADVICE
1	2.9.1119	JRCBD_S2.FI.UNAM.5108825c.1.12.960	prepared	no	

Solución propuesta:
La transacción hizo `rollback` debido a que el nodo commit no le fue posible confirmar la transacción.

Los demás nodos están preparados, se realizará `rollback` forzado ya que la transacción distribuida realizó `rollback`. Las siguientes instrucciones deberán ser ejecutadas como `sysdba` en los nodos locales.

```
rollback force '10.23.1444';
rollback force '2.9.1119';
```

Limpieza de vistas.

```
exec dbms_transaction.purge_lost_db_entry('10.23.1444);
exec dbms_transaction.purge_lost_db_entry('2.9.1119');
```