ANEXO I

NIVELES DE AISLAMIENTO

INDICE

1) DIRTY READ

1.1) En ORACLE

1.1.1) READ UNCOMMITTED

No se pueden ejecutar transacciones en Oracle en el nivel de aislamiento READ UNCOMMITTED ya que solo ofrece los niveles de aislamiento READ COMMITTED y SERIALIZABLE.

***Nota**: PostgreSQL ofrece los mismos niveles que Oracle, con la salvedad que permite las instrucciones SET ISOLATION LEVEL READ UNCOMMITTED y SET ISOLATION LEVEL REPEATABLE READ pero estas mapean a READ COMMITTED y SERIALIZABLE respectivamente.*

1.1.2) READ COMMITTED

```
Transaccion 1:
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

UPDATE T1 SET C1 = 0 WHERE C1 = 1;
DBMS_LOCK.sleep(10);
ROLLBACK;
END;

Transaccion 2:
DECLARE
CURSOR C IS SELECT * FROM T1;
RC  C%ROWTYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;

COMMIT;
END;
```

La transacción 2 no se bloquea. Sin embargo dada la naturaleza del MVCC la transacción 2 "ve" los datos comprometidos que se encontraban en el instante en que comenzó. Por lo tanto **No se produce DIRTY READ.**

1.2) En SQL SERVER

1.2.1) READ UNCOMMITED

Transacción 1:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRAN

update t1 set c1 = 0 where c1 = 1;
Waitfor Delay '000:00:10'
rollback tran
```

Transacción 2:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRAN
select * from t1 ;
commit tran
```

Si ejecutamos 1ero la transacción 1 y luego la transacción 2 durante los 10 segundos que dormimos a la transaccion1, la transacción 2 lee el valor 0 que es un valor no comprometido y de hecho al producirse luego el rollback debería considerarse como que nunca existió. **Se produce DIRTY READ.**

1.2.2) READ COMMITED

Transacción 1:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN

update t1 set c1 = 0 where c1 = 1;
Waitfor Delay '000:00:10'
rollback tran
```

Transacción 2:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
select * from t1 ;
commit tran
```

La transacción 2 se bloquea hasta que finaliza la transacción 1. **No se produce DIRTY READ.**

1.3) Conclusión

El fenómeno Dirty Read se produce en el nivel de aislamiento READ UNCOMMITTED, y se evita desde el READ COMMITTED en adelante.

SQL Server evita Dirty Read en el nivel de aislamiento READ COMMITTED mediante los bloqueos que implementa, postergando la transacción que quiere leer el dato "sucio" hasta que la transacción escritora termine. ORACLE evita este fenómeno en su nivel mas bajo, recordemos que no ofrece READ UNCOMMITTED, sin bloquear a ninguna transacción. La transacción lectora siempre lee los datos comprometidos al comienzo de la sentencia con lo que evita leer datos sucios.

2) NON-REPEATABLE READ

2.1) En ORACLE

2.1.1) READ COMMITTED

```
Transaccion 1:
DECLARE
CURSOR C IS SELECT * FROM T1;
RC  C%ROWTYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;
DBMS_LOCK.SLEEP(10);
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;

COMMIT;
END;

Transacción 2:
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE T1 SET C1 = 0 WHERE C1 = 1;
COMMIT;
END;
```

Si ejecutamos primero la transacción 1 y luego la transacción 2 durante los 10 segundos que dormimos a la transacción 1, se producen distintos resultados en la salida de los cursores de la transacción 1.

La primer lectura devuelve 1,2,3,4,5,6,7,8,9,10 y la 2da lectura (luego de ejecutada la transacción 2) devuelve 0,2,3,4,5,6,7,8,9,10. **Se produce Non Repeatable Read.**

2.1.2) REPEATABLE READ

Oracle no ofrece este nivel de aislamiento. Ver 1.1.1.

2.1.3) SERIALIZABLE

```
Transaccion 1:
DECLARE
CURSOR C IS SELECT * FROM T1;
RC  C%ROWTYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;
DBMS_LOCK.SLEEP(10);
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;

COMMIT;
END;

Transacción 2:
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE T1 SET C1 = 0 WHERE C1 = 1;
COMMIT;
END;
```

Ejecuto la transacción 1, luego durante los 10 segundos de delay ejecuto la transacción 2. La transacción 2 no se bloquea, sin embargo las 2 lecturas de la transacción 1 son idénticas. **No se produce Non Repeatable Read.**

2.2)En SQL SERVER

2.2.1) READ COMMITTED

Transacción 1:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
select * from t1 ;
Waitfor Delay '000:00:10'
select * from t1 ;
commit tran
```

Transacción 2:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN

update t1 set c1 = 0 where c1 = 1;
commit tran;
```

Si ejecutamos primero la transacción 1 y luego la transacción 2 durante los 10 segundos que dormimos a la transacción 1, la consulta "select * from t1" produce distintos resultados.

La primer lectura devuelve 1,2,3,4,5,6,7,8,9,10 y la segunda lectura (luego de ejecutada la transacción 2) devuelve 0,2,3,4,5,6,7,8,9,10. **Se produce Non Repeatable Read.**

2.2.2) REPEATABLE READ

Transacción 1:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
select * from t1 ;
Waitfor Delay '000:00:10'
select * from t1 ;
commit tran
```

Transacción 2:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN

update t1 set c1 = 0 where c1 = 1;
commit tran;
```

Ejecuto la transacción 1, luego durante los 10 segundo de delay ejecuto la transacción 2 y a diferencia de los niveles anteriores la transacción 2 se bloquea hasta que termina la transacción 1. Por lo tanto las dos lecturas de la transacción 1 son idénticas. **No se produce Non Repeatable Read.**

2.3) Conclusión

El fenómeno Non Repeatable Read o Fuzzy Read se produce en el nivel de aislamiento READ COMMITTED, y se evita desde el REPEATABLE READ en adelante.

SQL Server evita el Non Repeatable Read en el nivel de aislamiento REPEATABLE READ mediante los bloqueos que implementa, postergando la transacción que quiere modificar algún dato que tenga un bloqueo compartido (de lectura). Cuando la transacción de lectura que creo los bloqueos compartidos termina, recién se ejecuta la transacción escritora.

ORACLE evita este fenómeno en su nivel más alto SERIALIZABLE, recordemos que no ofrece REPEATABLE READ, sin bloquear a ninguna transacción. La transacción lectora siempre lee los datos al comienzo de su ejecución con lo que evita leer datos comprometidos pero posteriores a su comienzo. En REPEATABLE READ si se produce Non Repeatable Read ya que se leen solo los datos comprometidos pero al instante de comienzo de cada sentencia.

3) PHANTOM

3.1) En ORACLE

3.1.1) READ COMMITTED

```
Transaccion 1:
DECLARE
CURSOR C IS SELECT * FROM T1 WHERE C1 < 5;
RC  C%ROWTYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;
DBMS_LOCK.SLEEP(10);
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;

COMMIT;
END;


Transaccion 2:
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE T1 SET C1 = 6 WHERE C1 = 1;
COMMIT;
END;
```

Ejecuto la transacción 1, luego durante los 10 segundos de delay ejecuto la transacción 2. Al repetir la transacción 1 la lectura obtiene 1 filas menos, debido a que la transacción 2 cambio el valor 1, que cumplía la condición por el valor 6, que no la cumple. **Se produce Phantom.**

3.1.2) REPEATABLE READ

Oracle no ofrece este nivel de aislamiento. Ver 1.1.1.

3.1.3) SERIALIZABLE

Transaccion 1:

```
DECLARE
CURSOR C IS SELECT * FROM T1 WHERE C1 < 5;
RC  C%ROWTYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;
DBMS_LOCK.SLEEP(10);
 OPEN C;
 LOOP
    FETCH C INTO  RC;
    EXIT WHEN C%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RC.C1);
 END LOOP;
 CLOSE C;

COMMIT;
END;
```

Transaccion 2:

```
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE T1 SET C1 = 6 WHERE C1 = 1;
COMMIT;
END;
```

Ejecuto la transacción 1, luego durante los 10 segundos de delay ejecuto la transacción 2. La transacción 2 no se bloquea, sin embargo las 2 lecturas de la transacción 1 son idénticas. **No se produce Phantom.**

3.2) En SQL SERVER

3.2.1) REPEATABLE READ

Transacción 1:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
select * from t1 where c1 < 5;
Waitfor Delay '000:00:10'
select * from t1 where c1 < 5;
commit tran
```

Transaccion 2:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN

update t1 set c1 = 6 where c1 = 1;
commit tran
```

Ejecuto la transacción 1, luego durante los 10 segundos de delay ejecuto la transacción 2. Al repetir la transacción 1 la lectura obtiene 1 filas menos, debido a que la transacción 2 cambio el valor 1, que cumplía la condición por el valor 6, que no la cumple. **Se producen Phantom.**

3.2.2) SERIALIZABLE

Transaccion 1:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
select * from t1 where c1 < 5;
Waitfor Delay '000:00:10'
select * from t1 where c1 < 5;
commit tran
```

Transaccion 2:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN

update t1 set c1 = 6 where c1 = 1;
commit tran
```

Ejecuto la transacción 1, luego durante los 10 segundos de delay ejecuto la transacción 2. La transacción 2 se bloquea hasta que termina la transacción 1. **No se producen Phantom.**

3.3) Conclusión

El fenómeno Phantom se produce en el nivel de aislamiento REPEATABLE READ y los niveles menores, y se evita solamente en el SERIALIZABLE.

SQL Server evita Phantom en el nivel de aislamiento SERIALIZABLE mediante los bloqueos que implementa, postergando la transacción que quiere modificar algún dato sobre la o las tablas involucradas en la transacción de lectura. Cuando la transacción de lectura termina, recién se ejecuta la transacción escritora. Es indistinto si la transacción escritora modifica datos que no están involucrados en la lectura; si estos pertenecen a la o las tablas de la condición de búsqueda de la transacción de lectura, entonces serán bloqueados.

ORACLE evita este fenómeno en su nivel más alto SERIALIZABLE, recordemos que no ofrece REPEATABLE READ, sin bloquear a ninguna transacción. La transacción lectora siempre lee los datos al comienzo de su ejecución con lo que evita leer datos comprometidos pero posteriores a su comienzo. En REPEATABLE READ si se produce Phantom ya que se leen solo los datos comprometidos pero al instante de comienzo de cada sentencia. No es relevante el hecho de que la lectura de la transacción sea sobre un conjunto determinado por una condición de búsqueda, ya que en Oracle lo que interesa es el momento de creación del dato; por lo tanto para Oracle es lo mismo evitar Non Repeatable Read que Phantom.

4) DIRTY WRITE

4.1) En ORACLE

4.1.1) READ UNCOMMITTED

Oracle no ofrece este nivel de aislamiento. Ver 1.1.1.

4.1.2) READ COMMITTED

```
Transaccion 1:
DECLARE
DATO INT;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT X INTO DATO FROM T2 ;
DBMS_OUTPUT.PUT_LINE(DATO);
UPDATE T2 SET X = 10;
DBMS_LOCK.SLEEP(10);
ROLLBACK;
DBMS_LOCK.SLEEP(1);

SELECT X INTO DATO FROM T2 ;
DBMS_OUTPUT.PUT_LINE(DATO);
END;


Transaccion 2:
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE T2 SET X = 20;
END;
```

El Update de la Transacción 2 se bloquea hasta que la transacción 1 termina. Este es el único caso en que Oracle se bloquea en Read Committed. Por lo tanto **No se produce DIRTY WRITE.**

4.2) En SQL SERVER

4.2.1) READ UNCOMMITTED

Transaccion 1:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRAN
SELECT X FROM T2;
UPDATE T2 SET X = 10;
Waitfor Delay '000:00:10'
ROLLBACK TRAN
Waitfor Delay '000:00:01'
SELECT X FROM T2;
```

Transaccion 2:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRAN
UPDATE T2 SET X = 20;
```

El Update de la Transacción 2 se bloquea hasta que la transacción 1 termina, en este caso con ROLLBACK TRAN. Por lo tanto **No se produce DIRTY WRITE.**

4.3) Conclusión

Tanto en Oracle como en SQL Server no se permiten 2 escritores a la vez en la base sobre los mismos datos. Ambos prohíben el Dirty Write en su nivel mas bajo de aislamiento. El MVCC provisto por Oracle maneja bloqueos sobre los ítems de datos para evitar estos casos.

5) LOST UPDATE

5.1) En ORACLE

5.1.1) READ COMMITTED

```
Transacción 1:
DECLARE
DATO INTEGER;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT X INTO DATO FROM T2;
DBMS_OUTPUT.PUT_LINE ( 'DATO= ' || DATO );
DBMS_LOCK.SLEEP(10);
UPDATE T2 SET X = 130;
COMMIT;
END;

Transacción 2:
DECLARE
 DATO INTEGER;
 BEGIN
 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
 SELECT X INTO DATO FROM T2;
 UPDATE T2 SET X = 120;
 COMMIT;
 END;
```

Se ejecuta primero la Transacción 1, luego durante el delay de ésta se ejecuta la Transacción 2. El update de la Transacción 2 se pierde y es sobrescrito por el update de la Transacción 1 (X = 130). **Se produce LOST UPDATE.**

5.1.2) REPEATABLE READ

Oracle no ofrece este nivel de aislamiento. Ver 1.1.1.

5.1.3) SERIALIZABLE

```
Transacción 1:
DECLARE
DATO INTEGER;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT X INTO DATO FROM T2;
DBMS_OUTPUT.PUT_LINE ( 'DATO= ' || DATO );
DBMS_LOCK.SLEEP(10);
UPDATE T2 SET X = 130;
COMMIT;
END;

Transacción 2:
DECLARE
 DATO INTEGER;
 BEGIN
 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
 SELECT X INTO DATO FROM T2;
 UPDATE T2 SET X = 120;
 COMMIT;
 END;
```

Se ejecuta primero la Transacción 1, luego durante el delay de esta se ejecuta la Transacción 2. La Transacción 2 termina correctamente, mientras que la Transacción 1 aborta con el error **ORA-08177,** no puede serializar la Transacción 1 con la Transacción 2. En oracle en su nivel serializable si una transacción intenta modificar un dato que fue modificado por otra transacción concurrente, entonces cancela con el error de que no puede serializarse cuando intente comprometerse o abortar. Es por eso que la Transacción 1 aborta. **No se produce LOST UPDATE.**

```
Error on line 1
DECLARE
DATO INTEGER;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SE

ORA-08177: can't serialize access for this transaction
ORA-06512: at line 8
```

5.2) En SQL SERVER

5.2.1) READ COMMITTED

```
Transaccion 1:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
SELECT X FROM T2;
Waitfor Delay '000:00:10'
UPDATE T2 SET X = 130
COMMIT

Transaccion 2:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
SELECT X FROM T2;
UPDATE T2 SET X = 120
COMMIT
```

Se ejecuta primero la Transacción 1, luego durante el delay de esta se ejecuta la Transacción 2. El update de la Transacción 2 se pierde y es sobrescrito por el update de la Transacción 1 (X = 130). **Se produce LOST UPDATE.**

5.2.2) REPEATABLE READ

```
Transaccion 1:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT X FROM T2;
Waitfor Delay '000:00:10'
UPDATE T2 SET X = 130
COMMIT

Transaccion 2:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT X FROM T2;
UPDATE T2 SET X = 120
COMMIT
```

Se ejecuta primero la Transacción 1, luego durante el delay de esta se ejecuta la Transacción 2. La misma aborta con el error 1205 (DEADLOCK) debido a que cuando intenta actualizar el ítem X, no puede hacerlo ya que la transacción 1 modifico este ítem

y empezó con anterioridad a la Transacción 2. Se produce un deadlock por el recurso compartido X. **No se produce LOST UPDATE.**

```
(1 filas afectadas)

Servidor: mensaje 1205, nivel 13, estado 5, línea 1
Error 1205

Nivel de gravedad 13

Texto del mensaje

La transacción (Id. de proceso %1!) quedó en interbloqueo en {%2!}
recursos con otro proceso y fue elegida como sujeto del interbloqueo.
Ejecute de nuevo la transacción.
```

Explicación

Este error se produce cuando Microsoft® SQL Server™ encuentra un interbloqueo. Un interbloqueo tiene lugar cuando dos (o más) procesos intentan tener acceso a un recurso sobre el que el otro proceso mantiene un bloqueo. Dado que cada proceso tiene una petición de otro recurso, no se puede completar ningún proceso. Cuando se detecta un interbloqueo, SQL Server deshace el comando con el menor tiempo de proceso y devuelve el mensaje de error 1205 a la aplicación cliente. Este error no es fatal y no hará que se termine el proceso por lotes.

5.3) Conclusión

El fenómeno LOST UPDATE se produce tanto en Oracle como en SQL Server, en el nivel de aislamiento READ COMMITED. SQL Server lo evita en su nivel de aislamiento REPEATABLE READ mediante el bloqueo y cancelación de la Transacción 2, que quiere acceder al recurso que ya tiene adquirido la Transacción 1. En cambio, Oracle lo evita en su nivel de aislamiento SERIALIZABLE (recordemos que no posee REPEATABLE READ) mediante la cancelación de la Transacción 1 ya que al intentar comprometer se da cuenta que la Transacción 2 modifico antes que ella el mismo recurso, y después del timestamp de comienzo de la Transacción 1. En Oracle cancela la transacción que modifica un valor que ha sido modificado con anterioridad por otra transacción concurrente, y en SQL Server, simplemente el que gana es el primero que accedió al recurso.

Para decirlo de una manera más simple, en ORACLE cuando dos transacciones en nivel de aislamiento serializable modifican un mismo recurso, la primera que actualiza gana.

Es importante notar la diferencia en Oracle que es el primero que actualiza y no el primero que compromete quien gana. Esta prueba se realiza intercambiando las instrucciones en el ejemplo 5.1.3) de la siguiente manera:

```
DBMS_LOCK.SLEEP(10);
UPDATE T2 SET X = 130;

Por

UPDATE T2 SET X = 130;
DBMS_LOCK.SLEEP(10);
```

6) READ SKEW

6.1) En ORACLE

6.1.1) READ COMMITTED

```
Transacción 1:
DECLARE
DATO T3.X%TYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT X INTO DATO
FROM T3;
DBMS_OUTPUT.PUT_LINE ( 'DATO= ' || DATO );
DBMS_LOCK.SLEEP(10);
SELECT Y INTO DATO
FROM T3;
DBMS_OUTPUT.PUT_LINE ( 'DATO= ' || DATO );
COMMIT;
END;

Transacción 2:

BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE T3 SET X = 10;
UPDATE T3 SET Y = 20;
COMMIT;
END;
```

Se ejecuta primero la Transacción 1, luego durante el delay de ésta se ejecuta la Transacción 2. Ambas se ejecutan sin ningún bloqueo entre si (la transacción 1 es de solo lectura). Las lecturas de la transacción 1 son inconsistentes con la restricción de la tabla que no permite que X sea mayor que Y, ya que la primer lectura nos devuelve un valor de X = 50 y la segunda un valor de Y = 20. Oracle en su nivel de aislamiento READ COMMITTED tiene consistencia a nivel de sentencia, por lo que lee los datos generados por sentencias de transacciones concurrentes, posteriores al comienzo de la transacción. **Se produce READ SKEW.**

6.1.2) REPEATABLE READ

Oracle no ofrece este nivel de aislamiento. Ver 1.1.1.

6.1.3) SERIALIZABLE

```
Transacción 1:
DECLARE
DATO T3.X%TYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT X INTO DATO
FROM T3;
DBMS_OUTPUT.PUT_LINE ( 'DATO= ' || DATO );
DBMS_LOCK.SLEEP(10);
SELECT Y INTO DATO
FROM T3;
DBMS_OUTPUT.PUT_LINE ( 'DATO= ' || DATO );
COMMIT;
END;

Transacción 2:

BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE T3 SET X = 10;
UPDATE T3 SET Y = 20;
COMMIT;
END;
```

Se ejecuta primero la Transacción 1, luego durante el delay de ésta se ejecuta la Transacción 2. La Transacción 1 ejecutándose en el nivel de aislamiento serializable ve los datos como estaban al comienzo de la transacción por lo tanto no ve los valores inconsistentes (de acuerdo a las restricciones existentes) que pudieran existir generados por la transacción 2. **No se produce READ SKEW.**

6.2) En SQL SERVER

6.2.1) READ COMMITTED

Valores iniciales X=50 e Y= 100 y T3 tiene una check constraint donde X < Y
Transaccion 1:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRAN
SELECT X FROM T3;
Waitfor Delay '000:00:10'
SELECT Y FROM T3;
COMMIT;
```

Transaccion 2:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRAN
UPDATE T3 SET X = 10
UPDATE T3 SET Y = 20
COMMIT;
```

Se ejecuta primero la Transacción 1, luego durante el delay de esta se ejecuta la Transacción 2. Ambas se ejecutan sin ningún bloqueo entre si (la transacción 1 es de solo lectura). Las lecturas de la transacción 1 son inconsistentes con la restricción de la tabla que no permite que X sea mayor que Y, ya que la primer lectura nos devuelve un valor de X = 50 y la segunda un valor de Y = 20. **Se produce READ SKEW.**

6.2.2) REPEATABLE READ

Valores iniciales X=50 e Y= 100 y T3 tiene una check constraint donde X < Y
Transaccion 1:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRAN
SELECT X FROM T3;
Waitfor Delay '000:00:10'
SELECT Y FROM T3;
COMMIT;
```

Transaccion 2:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

```
BEGIN TRAN
UPDATE T3 SET X = 10
UPDATE T3 SET Y = 20
COMMIT;
```

Se ejecuta primero la Transacción 1, luego durante el delay de ésta se ejecuta la Transacción 2. A pesar de ser la Transacción 1 de solo lectura, la transacción 2 se bloquea, de esta manera evita que la transacción 1 lea un estado inconsistente de la base de datos (un valor X mayor que Y). **No se produce READ SKEW.**

6.3) Conclusión

El fenómeno de Read Skew se produce tanto en Oracle como en SQL Server en el nivel de aislamiento READ COMMITED y se evita en REPEATABLE READ en SQL Server y en Serializable en ORACLE. SQL Server evita el READ SKEW bloqueando la Transacción 2 a pesar de que ésta sea de solo lectura, y ORACLE lo evita sin bloqueos dando consistencia a nivel de transacción, por lo que las transacciones ven los datos como se encontraban al comienzo de su ejecución.

7) WRITE SKEW

7.1) En ORACLE

7.1.1) READ COMMITTED

```
Transacción 1:

DECLARE
dummy padre.col_padre%TYPE;
sigo boolean := true;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
 BEGIN
    select col_padre into dummy from padre where col_padre = 3;
 EXCEPTION
      WHEN NO_DATA_FOUND THEN
            sigo := FALSE ;
     END;
       -- si existe inserto el hijo
       IF (SIGO) THEN
             DBMS_LOCK.SLEEP(10);
             INSERT INTO hijo VALUES (3,'D');
             COMMIT;
      END IF;
END;


Transaccion 2;

DECLARE
dummy hijo.col_ref_padre%TYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  select col_ref_padre into dummy from hijo where col_ref_padre = 3;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      -- si no existen hijos borro el padre
            delete from padre where col_padre = 3;
              commit;
END;
```

Tanto la Transacción 1 como la Transacción 2 realizan chequeos a nivel de aplicación para mantener integridad referencial. La transacción 1 chequea la existencia del padre para ver si puede insertar un hijo, si esto sucede lo inserta y se compromete. La Transacción 2 chequea si no existen hijos para poder borrar un padre, si esto sucede borra el padre. Ambas transacciones asumen que los datos leídos no cambiarán y como esto puede no suceder en ORACLE se pueden producir inconsistencias. **Se produce WRITE SKEW.**

7.1.2) REPEATABLE READ

Oracle no ofrece este nivel de aislamiento. Ver 1.1.1.

7.1.3) SERIALIZABLE

```
Transacción 1:

DECLARE
dummy padre.col_padre%TYPE;
sigo boolean := true;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
 BEGIN
    select col_padre into dummy from padre where col_padre = 3;
 EXCEPTION
      WHEN NO_DATA_FOUND THEN
            sigo := FALSE ;
     END;
       -- si existe inserto el hijo
       IF (SIGO) THEN
             DBMS_LOCK.SLEEP(10);
             INSERT INTO hijo VALUES (3,'D');
             COMMIT;
      END IF;
END;


Transaccion 2;

DECLARE
dummy hijo.col_ref_padre%TYPE;
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  select col_ref_padre into dummy from hijo where col_ref_padre = 3;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      -- si no existen hijos borro el padre
            delete from padre where col_padre = 3;
              commit;
END;
```

Obtenemos el mismo resultado al ejecutar esto en el nivel de aislamiento READ COMMITTED y SERIALIZABLE. Ver 7.1.1.

7.2) En SQL SERVER

7.2.1) REPEATABLE READ

Transaccion 1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN
select col_padre from padre where col_padre = 3
insert into secuencia values (1,getdate(),'select col_padre from padre where col_padre =
3')
IF @@rowcount > 0 begin
-- si existe el padre inserto un hijo
Waitfor Delay '000: 00:10'
INSERT INTO hijo VALUES (3,'D')
insert into secuencia values (1,getdate(),'INSERT INTO hijo VALUES (3,"D")')
Waitfor Delay '000: 00:10'
COMMIT TRAN
insert into secuencia values (1,getdate(),'COMMIT TRAN')
END
ELSE ROLLBACK TRAN
```

Transaccion 2

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN
select col_ref_padre from hijo where col_ref_padre = 3
insert into secuencia values (2,getdate(),'select col_ref_padre from hijo where
col_ref_padre = 3')
Waitfor Delay '000: 00:10'
IF @@rowcount = 0 begin
-- si no existen hijos borro el padre
delete from padre where col_padre = 3
insert into secuencia values (2,getdate(),'delete from padre where col_padre = 3')
COMMIT TRAN
insert into secuencia values (2,getdate(),'COMMIT TRAN')
END
ELSE ROLLBACK TRAN
```

***Nota**: los inserts en la tabla secuencia son para ver el orden en el cual se ejecutan las sentencias.*

Tanto la Transacción 1 como la Transacción 2 realizan chequeos a nivel de aplicación para mantener integridad referencial. La transacción 1 chequea la existencia del padre para ver si puede insertar un hijo, si esto sucede lo inserta y se compromete.

La Transacción 2 chequea si no existen hijos para poder borrar un padre, si esto sucede borra el padre. En SQL Server en el nivel de aislamiento REPEATABLE READ el bloqueo de lectura de la Transacción 1 demora el delete de la Transacción 2 pero cuando este se libera deja que la Transacción 2 continúe normalmente. La Transacción 2 se durmió en el delete pero ya realizo el chequeo si existían hijos antes que la Transacción 1 los inserte. Por lo tanto como el chequeo fue anterior, la transacción 2 realiza el delete. Se pueden producir inconsistencias. Al demorar la transacción 1 a la transacción 2, lo que sucede ya no es WRITE SKEW.

Más formalmente Write Skew se define de la siguiente manera en el paper: ftp://ftp.research.microsoft.com/pub/tr/tr-95-51.pdf

A5B: r1[x]...r2[y]...w1[y]...w2[x]...(c1 and c2 occur) (Write Skew)

En nuestro ejemplo
r1[x] = select col_padre from padre where col_padre = 3

r2[y] = select col_ref_padre from hijo where col_ref_padre = 3

w1[y] = INSERT INTO hijo VALUES (3,'D')

w2[x]... = delete from padre where col_padre = 3

Los delays son para forzar la historia que se quiera.

Con la transacciones 1 y 2 si se produce error pero no es write skew ya que los bloqueos que realiza REPEATABLE READ fuerzan en realidad la siguiente historia

r1[x]...r2[y]...w1[y]...**C1 …** w2[x]...C2 …

No se produce Write Skew.

7.2.2) SERIALIZABLE

Transacción 1:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
        select col_padre from padre where col_padre = 3
                IF @@rowcount > 0 begin
                  -- si existe el padre inserto un hijo
                  Waitfor Delay '000:00:10'
                  INSERT INTO hijo VALUES (3,'D')
                  COMMIT TRAN
                END
                ELSE ROLLBACK TRAN
```

Transacción 2:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
          select col_ref_padre from hijo where col_ref_padre = 3
          IF @@rowcount = 0 begin
            -- si no existen hijos borro el padre
            delete from padre where col_padre = 3
            COMMIT TRAN
           END
           ELSE ROLLBACK TRAN
```

Tanto la Transacción 1 como la Transacción 2 realizan chequeos a nivel de aplicación para mantener integridad referencial. La transacción 1 chequea la existencia del padre para ver si puede insertar un hijo, si esto sucede lo inserta y se compromete. La Transacción 2 chequea si no existen hijos para poder borrar un padre, si esto sucede borra el padre. En SQL Server en el nivel de aislamiento SERIALIZABLE la Transacción 2 aborta con el error 1205 (DEADLOCK). Se produce un deadlock por un recurso compartido. La transacción 1 bloquea el recurso padre con un bloqueo de lectura e intenta bloquear el recurso hijo con un bloqueo de escritura mientras que la transacción 2 bloquea el recuso hijo con un bloqueo de lectura y espera por el recurso padre para realizar una escritura.

En SQL Server en el nivel de aislamiento SERIALIZABLE cuando la Transacción 1 intenta realizar el insert se produce el error abortándola. Dado que la transacción 2 realizo una lectura con anterioridad de la tabla hijo y si dejara insertar a la Transacción 1 y luego la transacción 2 repitiera la lectura podría producirse PHANTOM. **No se produce WRITE SKEW.**

NOTA:

SQL SERVER EN EL NIVEL DE AISLAMIENTO SERIALIZABLE BLOQUEA TODA LA TABLA PARA ESCRITURA DE LAS TRANSACCIONES CONCURRENTES SI SE HACE UN SELECT SOBRE LA MISMA. MIENTRAS QUE EN EL NIVEL DE AISLAMIENTO REPEATABLE READ BLOQUEA TODOS LOS DATOS DE LA TABLA PERO PERMITE INSERT DE LAS TRANSACCIONES CONCURRENTES SI SE HACE UN SELECT SOBRE LA MISMA.

7.3) Conclusión

Dado que Oracle no usa bloqueos de lectura, incluso con el nivel de aislamiento SERIALIZABLE, los datos leídos por una transacción pueden ser sobrescritos por otra. Las transacciones que realizan chequeos de consistencia a nivel de aplicación no deberían asumir que los datos que leen no van a cambiar durante la ejecución de la transacción (aunque esos cambios no sean visibles durante la ejecución de la transacción). Pueden ocurrir inconsistencias en la base de datos a menos que los chequeos a nivel de aplicación sean codificados cuidadosamente, incluso en transacciones SERIALIZABLES.

Por lo dicho anteriormente en Oracle se produce Write Skew mientras que en SQL Server, al evitar el fenómeno Phantom, en el nivel de aislamiento serializable, mediante bloqueos, evita inserciones de transacciones concurrentes sobre la tabla bloqueada por una lectura, por lo tanto en el ejemplo no se pueden crear hijos sin padre. En SQL Server no se produce Write Skew.

8) Scripts de Creación de tablas para los ejemplos

8.1) En ORACLE

```
CREATE TABLE T1
(
C1 INT
)
------------------------------

DECLARE
i int;
BEGIN
       FOR I IN 1..10 LOOP
            INSERT INTO T1 VALUES(I);
      END LOOP;
COMMIT;
END;
```

```
CREATE TABLE PADRE (
COL_PADRE INT);

CREATE TABLE HIJO (
COL_REF_PADRE INT, COL_HIJO VARCHAR(2));


INSERT INTO PADRE VALUES (1);
INSERT INTO PADRE VALUES (2);
INSERT INTO PADRE VALUES (3);

INSERT INTO HIJO VALUES (1,'A');
INSERT INTO HIJO VALUES (1,'B');
INSERT INTO HIJO VALUES (2,'C');

COMMIT;
```

8.2) En SQL SERVER

```
Create table T1
(
C1 int
)
--------------------------------
Declare @i As Int
Set @i = 1
While @i <= 10
Begin
        --Print 'Loop número' + Cast(@i As Varchar(2))
        Insert Into T1 Values(@i)
        Set @i = @i + 1
End
```

```
Create table T2
(
X int,
Y int
)
Insert Into T2 Values(50,50)
```

```
Create table T3
(
X int,
Y int,
CHECK (x < y)
)
Insert Into T3 Values(50,100)
```

```
Create table T4
(
X int,
Y int
)
Insert Into T4 Values(50,100)
```

```
Create table Secuencia
 (
idtran varchar(2),
fecha datetime,
operacion varchar(255)
)
```