



Universidad Nacional del Litoral

Facultad de Ingeniería y Ciencias Hídricas

Departamento de Informática

Bases de Datos

SQL: Guía de Trabajo Nro. 8
Configuración de Transacciones

Transacciones en SQL

1. Transacción implícita

Tal como vimos en la Guía de trabajo Nro. 4, toda sentencia SQL aislada representa en sí lo que se denomina una *transacción implícita*. Es decir, el inicio (`BEGIN`) y confirmación (`COMMIT`) de la transacción están implícitamente definidos en toda sentencia SQL.

2. Transacción explícita

Que una transacción implícita llegue a buen término es claramente un problema del DBMS. Como desarrolladores, nuestro problema comienza con las **transacciones explícitas**. Son las que vimos en la *Sección 20- Demarcación de transacciones en SQL* de la *Guía de Trabajo Nro. 4*. Son las transacciones cuyo comienzo es demarcado con la sentencia `BEGIN TRANSACTION` y cuyo final es demarcado con la sentencia `COMMIT TRANSACTION` o `ROLLBACK TRANSACTION`.

Desde el punto de vista de una aplicación, este tipo de transacción constituye una **unidad lógica de trabajo**, algo de nuestro sistema que debe ser llevado a cabo como una unidad.

En relación a este enfoque, C. J. Date define una transacción como “una secuencia de varias operaciones intermedias mediante la cual un estado consistente de la base de datos se transforma en otro estado consistente, sin conservar necesariamente la consistencia en todos los puntos intermedios”.

Date dice que, (definiendo una transacción explícita) “podemos lograr que una secuencia de operaciones, que por esencia no es atómica, aparente serlo desde el punto de vista externo”.

Así, un `COMMIT TRANSACTION` indica que se ha finalizado con éxito la **unidad lógica de trabajo**, que la base de datos está en un nuevo estado consistente.

Por otro lado, un `ROLLBACK TRANSACTION` indica que algo salió mal, que la base de datos podría estar en un estado inconsistente, y que todas las modificaciones realizadas hasta el momento como parte de la **unidad lógica de trabajo** deberían deshacerse.

3. Atomicidad de una transacción

Dijimos que, en una transacción explícita, la serie de pasos se ejecuta como una unidad o no se ejecuta en absoluto.

Se dice que la operación es “todo-o-nada”: si se materializa el **COMMIT**, los cambios provocados persisten. Si se lleva a cabo un **ROLLBACK**, todos los cambios son deshechos.

A esto se denomina “atomicidad” de la transacción, la “A” del *conjunto de características ACID* vistas en teoría.

Como desarrolladores, aseguramos la **Atomicity** definiendo transacciones explícitas y controlando deshacer los cambios ante la ocurrencia de errores del sistema o de violación de reglas de negocios, tal como aprendimos en la *Sección 4. Errores y transacciones* de la *Guía de Trabajo Nro. 6*.

4. Durabilidad de una transacción

Ya sea que se trate de una transacción implícita o de una demarcada explícitamente, los cambios realizados no se deben perder aún en el evento de un crash o falla del sistema.

En el caso de una transacción explícita, si ocurre un error grave que provoca un crash durante el desarrollo de alguno de los pasos de la misma, y algún paso no puede ser completado, el sistema debe garantizar que se pueda recuperar el estado anterior, **sin dejar efectos parciales sobre los datos**.

Este es un tema fuertemente asociado a la **implementación** (los aspectos internos) de los servicios de datos¹.

Esta es la “durability” de la transacción, la “D” del *conjunto de características ACID* vistas en teoría.

¹ Por ejemplo, SQL Server asegura la durabilidad a través de dos características llamadas **Automatic rollback** y **Write-ahead logging**. De una manera simplificada, durante el arranque el motor de base de datos posee una *fase de recovery* que “revisa” el transaction log. Si encuentra transacciones no confirmadas (COMMITeadas), las deshace, recuperando el estado anterior. Esto se denomina Automatic rollback. Si un crash sucede luego del reconocimiento (acknowledgment) del COMMIT de una transacción, la misma también es asegurada ya que aún cuando no haya podido ser hecha permanente en la base de datos, existe completa en data pages “cacheadas”. El transaction log y el motor de base de datos la puede reconstruir durante su reinicio.

5. Consistencia de una transacción

Esta es la "consistency" de la transacción, la "C" del *conjunto de características ACID* vistas en teoría.

La característica de Consistencia es abstracta. Hace referencia a que los datos deben ser siempre "lógicamente correctos" desde un punto de vista de reglas de integridad, que pueden estar asociadas a reglas de negocios.

En el caso de una transferencia bancaria desde una cuenta **A** a una cuenta **B**, una regla lógica sería que no puede acreditarse un monto en una cuenta **B** sin una operación de contra-balance de débito en una cuenta **A**. Dicho en otras palabras, no podremos tener en la cuenta **B** dinero creado de la nada. De manera recíproca, no podremos debitar un monto de una cuenta **A** sin una operación de contra-balance de crédito en una cuenta **B**. En otras palabras, no podremos tener en la cuenta **A** dinero que desaparece sin un movimiento que respalde el débito.

Técnicamente, la consistencia es redundante con las otras tres características.

6. El problema de la concurrencia.

Hasta aquí, el tema de transacciones parecería agotado: encerramos los conjuntos de operaciones en una transacción y solamente la confirmamos si todas las condiciones son las adecuadas. Además, si sucediera algo fuera de nuestro control (un corte de energía eléctrica, por ejemplo), el DBMS usaría su "sistema de emergencia" para restablecer la base de datos a su estado anterior.

Sin embargo, hay un problema. Esta situación ideal sería posible si solamente existieran **nuestras** transacciones. Sin embargo, en el mundo real muchas otras conexiones a la base de datos estarán compitiendo por los mismos recursos, y esto incorpora el concepto de **Concurrencia**.

En aplicaciones tales como web Services, operaciones bancarias o reservaciones de líneas aéreas, sobre la base de datos se lleven a cabo *cientos de operaciones por segundo*. Es muy probable que tengamos dos operaciones afectando en el mismo momento la misma cuenta bancaria o vuelo. Si esto sucede, éstas operaciones pueden interactuar de maneras extrañas.

Ejemplo

Usaremos el siguiente ejemplo como base para nuestras pruebas: una aerolínea proporciona a los usuarios una aplicación Web a través de la cual pueden elegir un asiento para su vuelo. La aplicación muestra un “mapa” de asientos disponibles:



El siguiente es el modelo de base de datos:



Para simplificar, trabajaremos con un único vuelo, el 'GA4561B', y con solo cuatro asientos (los que aparecen identificados en el mapa de asientos):

```
CREATE TABLE Vuelo
(
    idVuelo VARCHAR (8),
    fecha DATETIME,
    PRIMARY KEY (idVuelo)
)
```

```
INSERT INTO vuelo
SELECT 'GA4561B', '2015/10/12';
```

```
CREATE TABLE VueloAsiento
(
    idVuelo VARCHAR (8),
    idAsiento VARCHAR(6),
    estadoAsiento CHAR(1) CHECK (estadoAsiento IN ('O', 'D'))
    PRIMARY KEY (idVuelo, idAsiento)
)
```

```

INSERT INTO VueloAsiento
  SELECT 'GA4561B', 'AL01', 'D';

INSERT INTO VueloAsiento
  SELECT 'GA4561B', 'AL02', 'D';

INSERT INTO VueloAsiento
  SELECT 'GA4561B', 'AR01', 'D';

INSERT INTO VueloAsiento
  SELECT 'GA4561B', 'AR02', 'D';

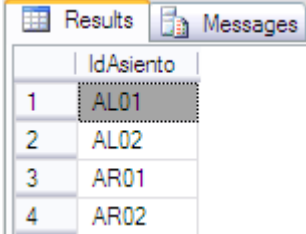
```

El mapa está mostrando los asientos actualmente desocupados:

```

SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';

```



	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

Si esta situación es desatendida, puede suceder lo siguiente:

El cliente **1** elige reservar el asiento 'AR02'.

La base de datos es modificada a través de la siguiente sentencia **UPDATE**:

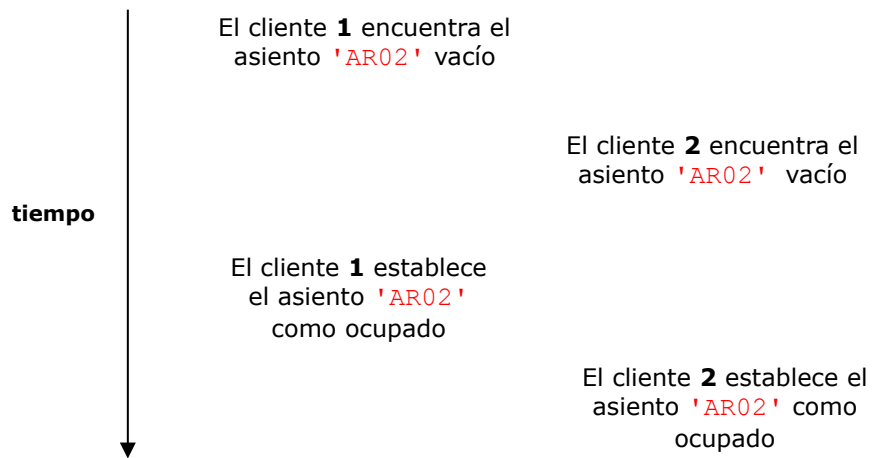
```

UPDATE VueloAsiento
  SET estadoAsiento = 'O'
  WHERE idVuelo = 'GA4561B' AND
        idAsiento = 'AR02';

```

En ese momento exacto el cliente **2** también está observando el mapa y elige reservar el asiento 'AR02'.

La sucesión de eventos en el tiempo sería:



Sin un tratamiento adecuado, aún cuando todas las operaciones se llevaron a cabo sin errores, sucede que ambos clientes "creen" que han reservado el asiento 'AR02'.

6.1. Inviabilidad de la ejecución serial (Serializability)

La solución al problema de la concurrencia es la ejecución secuencial (en serie) de las transacciones.

Lamentablemente, en la práctica es virtualmente imposible requerir que las operaciones se ejecuten en serie. Las operaciones son tantas que se requiere de algún paralelismo.

Las transacciones son entonces normalmente ejecutadas solapadas en el tiempo ya que este es el único enfoque viable en cuanto a eficiencia.

6.2. Isolation

Las transacciones poseen una tercer propiedad que está relacionada a la Concurrencia.

Cuando una transacción **A** y una transacción **B** se están ejecutando de manera concurrente, la transacción **A no debe poder ver** los cambios incompletos (estados intermedios) realizados por **B**, y la transacción **B** no debe poder ver los cambios incompletos (estados intermedios) realizados por **A**.

El "todo-o-nada" que enuncia el conjunto de criterios ACID tiene como vemos una **doble significación**:

I. Son "todo-o-nada" en el sentido del efecto (los cambios) que provocan en la base de datos.

II. Son "todo-o-nada" en términos de visibilidad. Las demás transacciones ven el resultado completo o no ven nada.

Lo que estamos enunciando aquí es justamente la "Isolation" de la transacción, la "I" del *conjunto de características ACID* vistas en teoría, y está directamente relacionada a la "A" (Atomicity).

Una transacción que satisface la "I" (Isolation) se ejecuta como si fuese la única en ejecución en ese momento.

6.3. Situaciones derivadas de la concurrencia

La concurrencia puede provocar efectos colaterales que pueden ser vistos como problemas de consistencia a simplemente como situaciones posibles que el desarrollador está dispuesto a asumir o no.

La elección de un determinado *Isolation Level* determina cuales de éstas situaciones “admite” el desarrollador.

6.3.1. Lost updates

Los lost updates ocurren cuando dos procesos leen los mismos datos y los manipulan, alterando su valor, intentando ambos actualizar un valor original al nuevo valor. El segundo proceso puede sobrescribir completamente la actualización llevada a cabo por el primero, En nuestro caso de Ejemplo, dos clientes están tratando de reservar el mismo asiento. Si estuviésemos guardando el ID del cliente que finalmente lo reserva podría darse de que el segundo cliente sobrescribiera el UPDATE del primero.

Lost updates es el único comportamiento que se desea evitar en todos los casos.

6.3.2. Dirty reads

Cuando una transacción se ejecuta provoca **cambios tentativos** sobre la base de datos.

En dependencia del Isolation level elegido, esos cambios tentativos pueden haber sido “vistos” por alguna otra transacción. A esos cambios tentativos vistos por otra transacción se les denomina **Dirty data**. Son datos escritos por una transacción que aún no ha llevado a cabo un [COMMIT](#).

Si una transacción **A** escribe datos tentativos (dirty data) y otra transacción **B** los lee, se dice que la transacción **B** está haciendo una *dirty read*.

Las dirty reads traen aparejadas grandes riesgos:

1. Una transacción **A** escribe datos tentativos (dirty data) y otra transacción **B** los lee.
2. La transacción **A** hace un [ROLLBACK](#) de su modificación tentativa.
Cuando esto sucede, la modificación tentativa es eliminada de la base de datos, y se supone que el resto de las aplicaciones se comporten como si nunca hubiese existido.
3. La transacción **B**, que tomó como “buena” la dirty data, puede hacer su propio [COMMIT](#) o realizar una acción secundaria que involucre esta dirty data.

Isolation levels y dirty reads

Algo que es crucial es que un proceso que está actualizando datos no tiene en absoluto control sobre otros procesos que pretendan leer esos datos antes de que los mismos sean COMMITeados.

Es responsabilidad del proceso que lee el hacer una lectura de datos que no es garantía que sean COMMITeados.

En otras palabras, un proceso que actualiza datos no tiene manera de impedir dirty reads.

6.3.3. Non-repeatable reads

Un non-repeatable read es la situación en la cual una transacción realiza una lectura en un momento $T1$ y lleva a cabo la misma lectura en un momento $T2$ sin garantías de obtener los mismos valores para la misma.

Algunas tuplas pueden haber cambiado o simplemente desaparecido.

Esto sucede porque otra transacción puede haber actualizado los datos en el espacio de tiempo que transcurrió entre las dos lecturas.

6.3.4. Phantom read

Una phantom read es la situación en la cual una transacción realiza una lectura en un momento $T1$ y lleva a cabo la misma lectura en un momento $T2$ y obtiene como resultado que se han agregado nuevas tuplas.

Esto sucede porque otra transacción puede haber insertado nuevas tuplas en el espacio de tiempo que transcurrió entre las dos lecturas.

Las tuplas que se "agregan" como resultado de la segunda lectura se denominan "phantom tuples".

6.4. Isolation levels ANSI SQL

La Isolation ideal correspondería a la ejecución serial, que dijimos es inviable en el mundo real.

ANSI SQL define diferentes niveles de Isolation, desde el más cercano a la ejecución serial hasta el más alejado de la misma.

6.4.1. Isolation level Serializable

En este Isolation level las transacciones **se comportan** como si fuesen ejecutadas "en serie"².

El resultado final se presenta a los usuarios como si las operaciones hubiesen sido ejecutadas de manera secuencial.

Es el nivel más alto de **Isolation**, y es el Isolation level por omisión en el estándar ANSI SQL.

Decimos que dos transacciones **A** y **B** concurrentes se ejecutan en Isolation Level Serializable si el estado de base de datos resultado es igual al resultado que hubiésemos obtenido si cada transacción se hubiese ejecutado de manera secuencial, sin superposición ni solapamiento en el tiempo.

El Isolation level Serializable no admite lost updates, ni dirty reads. Tampoco admite situaciones de non-repeatable reads o phantom reads.

O sea, si una transacción **A** posee un **Isolation Level Serializable**, impedirá que otra transacción **B** modifique los datos que están siendo leídos por la transacción **A**, hasta que ésta finalice, e impedirá que una transacción **C** inserte tuplas, las cuales podrían coincidir con la condición de lectura de la transacción **A**.

Establecemos este isolation level a través de la siguiente sentencia:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Plantearemos alternativas de solución a nuestro caso de Ejemplo usando Isolation level Serializable:

² Decimos que el DBMS hace que las transacciones se comporten como si fuesen ejecutadas "en serie". Esto es implementado a través de locks.

Ejemplo S1	
Isolation Level	Serializable
Situación a evaluar	Dirty reads

Supongamos el siguiente escenario:

- A.** La sesión **1** corresponde a un Cliente que reserva el asiento '**AR02**' del vuelo '**GA4561B**'.
- B.** La sesión **2** corresponde a otro cliente que lista los asientos desocupados del vuelo '**GA4561B**' a fin de reservar uno.

Sesión 1

```
BEGIN TRANSACTION;

UPDATE VueloAsiento
SET estadoAsiento = 'O'
WHERE idVuelo = 'GA4561B' AND
      idAsiento = 'AR02';
WAITFOR DELAY '00:00:09.000' ;

COMMIT TRANSACTION;
```

Sesión 2

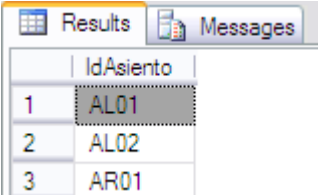
```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION

SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';

COMMIT TRANSACTION
```

Se percibe un retraso en la ejecución de la Sesión **2**, ya que la Sesión **1** está bloqueando filas y la Sesión **2** debe esperar a que finalice esa transacción a fin de ejecutar la lectura. Obtenemos:



	IdAsiento
1	AL01
2	AL02
3	AR01

En la Sesión **2**, el Cliente que está buscando asientos libres no ve al asiento '**AR02**' como una opción, ya que éste está ocupado.

Conclusión: No hay dirty reads.

Ejemplo S2	
Isolation Level	Serializable
Situación a evaluar	Non-repeatable reads

Supongamos el siguiente escenario:

- A.** La sesión **1** lista los asientos disponibles del vuelo 'GA4561B' como parte de una transacción.
- B.** La sesión **2** ocupa uno de los asientos disponibles y hace un COMMIT de la transacción.
- C.** La sesión **1** vuelve a listar los asientos disponibles del vuelo 'GA4561B' como parte de la misma transacción.

Sesión 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN TRANSACTION
```

```
SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';
```

```
WAITFOR DELAY '00:00:10.000' ;
```

```
SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';
```

```
COMMIT TRANSACTION
```

Sesión 2


```
BEGIN TRANSACTION;
```

```
UPDATE VueloAsiento  
SET estadoAsiento = 'O'  
WHERE idVuelo = 'GA4561B' AND  
      idAsiento = 'AR02';
```

```
COMMIT TRANSACTION;
```

Podemos notar que el **UPDATE** de la Sesión 2 tarda en ejecutarse el tiempo que tarda en ejecutarse la Sesión 1. Esto se debe a que no puede proceder con el **UPDATE** ya que la Sesión 1 ha establecido locks sobre las tuplas que está leyendo a fin de obtener siempre el mismo resultado mientras dure su propia la transacción.

A continuación el resultado de las dos lecturas de la Sesión 1:



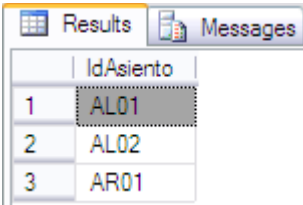
	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

Como vemos, la sesión 1 obtiene siempre cuatro asientos disponibles, cuando en realidad hay solo tres. Lo que obtenemos es justamente un **Repeatable Read**: un asiento que está disponible para el primer query permanecerá disponible en queries subsiguientes mientras dure la transacción.

Por supuesto, una vez finalizada la ejecución de la transacción de la Sesión 1, si disparamos una nueva sentencia **SELECT** obtenemos el resultado correcto:

```
SELECT IdAsiento  
FROM VueloAsiento  
WHERE idVuelo = 'GA4561B' AND  
      estadoAsiento = 'D';
```



	IdAsiento
1	AL01
2	AL02
3	AR01

Conclusión: No hay Non-repeatable reads.

Ejemplo S3	
Isolation Level	Serializable
Situación a evaluar	Phantom reads

Supongamos el siguiente escenario:

- A.** La sesión **1** lista los asientos disponibles del vuelo **'GA4561B'** como parte de una transacción.
- B.** La sesión **2** inserta un nuevo asiento disponible para el vuelo **'GA4561B'** y hace un **COMMIT** de la transacción.
- C.** La sesión **1** vuelve a listar los asientos disponibles del vuelo **'GA4561B'** como parte de la misma transacción.

Sesión 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION

    SELECT IdAsiento
        FROM VueloAsiento
        WHERE idVuelo = 'GA4561B' AND
              estadoAsiento = 'D';

    WAITFOR DELAY '00:00:10.000' ;

    SELECT IdAsiento
        FROM VueloAsiento
        WHERE idVuelo = 'GA4561B' AND
              estadoAsiento = 'D';

COMMIT TRANSACTION
```

Sesión 2

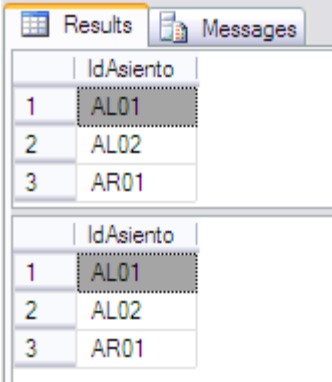
```
BEGIN TRANSACTION;

    INSERT INTO VueloAsiento VALUES ('GA4561B', 'ZZ01', 'D')

COMMIT TRANSACTION;
```


Podemos notar que el `INSERT` de la Sesión 2 tarda en ejecutarse el tiempo que tarda en ejecutarse la Sesión 1. Esto se debe a que no puede proceder con el `INSERT` porque la Sesión 1 ha establecido locks sobre las tuplas que está leyendo a fin de obtener siempre el mismo resultado mientras dure su propia la transacción.

A continuación el resultado de las dos lecturas de la Sesión 1:



	IdAsiento
1	AL01
2	AL02
3	AR01

	IdAsiento
1	AL01
2	AL02
3	AR01

Conclusión: No hay Phantom reads.

6.4.2. Isolation level Read-Uncommitted

Que una transacción posea un **Isolation Level Read-Uncommitted** significa que le es permitido realizar **dirty reads**.

El Isolation level Read uncommitted es el más permisivo de todos: admite situaciones de dirty reads, non-repeatable reads y phantom reads.

Establecemos este isolation level a través de la siguiente sentencia:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

Ejemplo RU1	
Isolation Level	Read-Uncommitted
Situación a evaluar	Dirty reads

Supongamos el siguiente escenario:

- A.** La sesión **1** corresponde a un Cliente que reserva el asiento **'AR02'** del vuelo **'GA4561B'**.
- B.** La sesión **2** corresponde a otro cliente que lista los asientos desocupados del vuelo **'GA4561B'** a fin de reservar uno.
- C.** La sesión **1** cancela la reserva.

Sesión 1

```
BEGIN TRANSACTION;

UPDATE VueloAsiento
SET estadoAsiento = 'O'
WHERE idVuelo = 'GA4561B' AND
      idAsiento = 'AR02';

WAITFOR DELAY '00:00:09.000' ;

ROLLBACK TRANSACTION;
```

Sesión 2

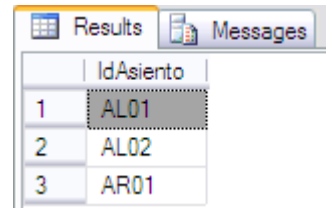
```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

BEGIN TRANSACTION

SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';

COMMIT TRANSACTION
```

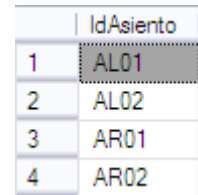
No se percibe un retraso en la ejecución de la Sesión **2**, ya que la Sesión 1 no ha lockeado filas. La Sesión **2** obtiene el siguiente resultado:



	IdAsiento
1	AL01
2	AL02
3	AR01

Lo que ha sucedido es que la sesión **2** ha realizado una dirty read de los asientos desocupados. El cliente que desea reservar **no ve al asiento 'AR02' como una opción**, ya que éste está ocupado, lo cual es falso.

Por supuesto, una vez finalizada la ejecución de la transacción de la Sesión **1**, si volvemos a repetir el código de la Sesión **2**, obtenemos el resultado correcto:



	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

Conclusión: Hay dirty reads.

Ejemplo RU2

Isolation Level	Read-Uncommitted
Situación a evaluar	Dirty reads con escritura

En este ejemplo se presenta el mismo escenario con un agravante, la transacción que lleva a cabo el dirty read escribe en la base de datos en base a una lectura incorrecta:

Supongamos la siguiente:

A. La sesión **1** corresponde a un Cliente que reserva el asiento **'AR02'** del vuelo **'GA4561B'**.

B. La sesión **2** mantiene en una tabla la cantidad de asientos libres de cada vuelo. Realiza una lectura de los asientos libres del vuelo **'GA4561B'** y encuentra tres. **Ve como ocupado al asiento 'AR02'**, que fue ocupado por la transacción de la Sesión **1**, pendiente de **COMMIT**.

C. La sesión **1** cancela la reserva.

```
CREATE TABLE Cantidad
(
    idVuelo VARCHAR (8) PRIMARY KEY,
    cantLibres Integer NULL
)
```

```
INSERT INTO Cantidad (idVuelo) VALUES ('GA4561B')
SELECT * FROM cantidad
```

idVuelo	cantLibres
GA4561B	NULL

Sesión 1

```
BEGIN TRANSACTION;
```

```
UPDATE VueloAsiento
SET estadoAsiento = 'O'
WHERE idVuelo = 'GA4561B' AND
      idAsiento = 'AR02';
```

```
WAITFOR DELAY '00:00:10.000' ;
```

```
ROLLBACK TRANSACTION;
```

Sesión 2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

```
DECLARE @cantidad Integer;
```

```
BEGIN TRANSACTION
```

```
SELECT @cantidad = COUNT(IdAsiento)
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';
```

```
UPDATE Cantidad
SET cantLibres = @cantidad
WHERE idVuelo = 'GA4561B'
```

```
COMMIT TRANSACTION
```

No se percibe un retraso en la ejecución de la Sesión **2**, ya que la Sesión 1 no ha lockeado files.

```
SELECT * FROM cantidad
```

idVuelo	cantLibres
GA4561B	3

```
SELECT * FROM VueloAsiento
```

idVuelo	idAsiento	estadoAsiento
GA4561B	AL01	D
GA4561B	AL02	D
GA4561B	AR01	D
GA4561B	AR02	D

La cantidad real de asientos disponibles es cuatro. La Sesión **2** ha llevado a cabo un `UPDATE` en base a un resultado incorrecto.

6.4.3. Isolation level Repeatable-Read

Que una transacción posea un **Isolation Level Repeatable-Read** significa que, si en una transacción **A** una tupla es recuperada la primera vez, podemos asegurar que la misma tupla será recuperada nuevamente si el query es repetido.

Esto se logra impidiendo que otra transacción **B** modifique los datos que están siendo leídos por la transacción **A**, hasta que ésta finalice.

El Isolation level Repeatable-Read no admite situaciones de dirty reads, y, por supuesto, no admite non-repeatable reads. Sin embargo, admite phantom reads.

Esto sucede porque se permite que otra transacción **B** inserte tuplas, y éstas pueden coincidir con la condición de lectura de la transacción **A**.

Establecemos este isolation level a través de la siguiente sentencia:

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

Ejemplo RR1	
Isolation Level	Repeatable-Read
Situación a evaluar	Dirty reads

Supongamos el mismo escenario del ejemplo **S1**:

Sesión 1

```
BEGIN TRANSACTION;

UPDATE VueloAsiento
SET estadoAsiento = 'O'
WHERE idVuelo = 'GA4561B' AND
      idAsiento = 'AR02';

WAITFOR DELAY '00:00:09.000' ;

ROLLBACK TRANSACTION;
```

Sesión 2

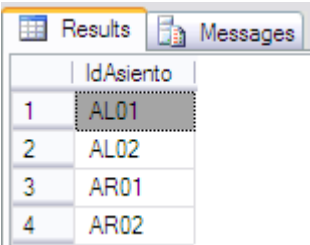
```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRANSACTION

SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';

COMMIT TRANSACTION
```

Se percibe un retraso en la ejecución de la Sesión **2**, ya que la Sesión **1** está lockeando filas y la Sesión **2** debe esperar hasta que finalice esa transacción. Obtenemos:



	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

Conclusión: No hay dirty reads.

Ejemplo RR2	
Isolation Level	Repeatable-Read
Situación a evaluar	Non-repeatable reads

Supongamos el mismo escenario del ejemplo **S2**:

Sesión 1

```

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION

    SELECT IdAsiento
        FROM VueloAsiento
        WHERE idVuelo = 'GA4561B' AND
            estadoAsiento = 'D';

    WAITFOR DELAY '00:00:10.000' ;

    SELECT IdAsiento
        FROM VueloAsiento
        WHERE idVuelo = 'GA4561B' AND
            estadoAsiento = 'D';

COMMIT TRANSACTION


```

Sesión 2

```
BEGIN TRANSACTION;  
  
UPDATE VueloAsiento  
SET estadoAsiento = 'O'  
WHERE idVuelo = 'GA4561B' AND  
      idAsiento = 'AR02';  
  
COMMIT TRANSACTION;
```

Podemos notar que el `UPDATE` de la Sesión 2 tarda en ejecutarse el tiempo que tarda en ejecutarse la Sesión 1. Esto se debe a que no puede proceder con el `UPDATE` ya que la Sesión 1 ha establecido locks sobre las tuplas que está leyendo a fin de obtener siempre el mismo resultado mientras dure su propia la transacción.

A continuación el resultado de las dos lecturas de la Sesión 1:



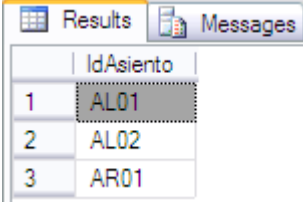
	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

Como vemos, la sesión 1 obtiene siempre cuatro asientos disponibles, cuando en realidad hay solo tres, o sea que obtenemos un **Repeatable Read**.

Por supuesto, una vez finalizada la ejecución de la transacción de la Sesión 2, si disparamos una sentencia `SELECT` obtenemos el resultado correcto:

```
SELECT IdAsiento  
FROM VueloAsiento  
WHERE idVuelo = 'GA4561B' AND  
      estadoAsiento = 'D';
```



	IdAsiento
1	AL01
2	AL02
3	AR01

Conclusión: No hay non-repeatable reads.

Ejemplo RR3	
Isolation Level	Repeatable-Read
Situación a evaluar	Phantom reads

Supongamos el mismo escenario del ejemplo **S3**:

Sesión 1

```

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION

    SELECT IdAsiento
    FROM VueloAsiento
    WHERE idVuelo = 'GA4561B' AND
           estadoAsiento = 'D';

    WAITFOR DELAY '00:00:10.000' ;

    SELECT IdAsiento
    FROM VueloAsiento
    WHERE idVuelo = 'GA4561B' AND
           estadoAsiento = 'D';

COMMIT TRANSACTION

```

Sesión 2

```

BEGIN TRANSACTION;

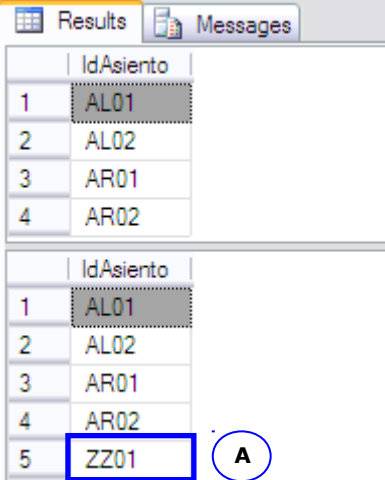
    INSERT INTO VueloAsiento VALUES ('GA4561B', 'ZZ01', 'D')

COMMIT TRANSACTION;

```

Podemos notar que el `INSERT` de la Sesión **2** se ejecuta inmediatamente. Esto se debe a que puede proceder con el `INSERT` ya que la Sesión **1** no ha establecido locks sobre las tuplas que está leyendo a fin de obtener siempre el mismo resultado mientras dure su propia la transacción.

A continuación el resultado de las dos lecturas de la Sesión 1:



	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02
5	ZZ01

En el segundo `SELECT`, la Sesión **1** obtiene una phantom tuple (**A**).

Conclusión: Hay phantom reads.

6.4.4. Isolation level Read Committed

Las transacciones con Isolation Level Read Committed son más permisivas que las con Isolation Level Repeatable Read. No admiten dirty reads, pero, además de phantom reads, admiten non-repeatable reads.

Read Committed es el Isolation level por omisión en SQL Server, PostgreSQL y Oracle.

Establecemos este isolation level a través de la siguiente sentencia:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Ejemplo RC1	
Isolation Level	Read Committed
Situación a evaluar	Dirty reads

Supongamos el mismo escenario del ejemplo **S1**:

Sesión 1

```
BEGIN TRANSACTION;

UPDATE VueloAsiento
SET estadoAsiento = 'O'
WHERE idVuelo = 'GA4561B' AND
      idAsiento = 'AR02';

WAITFOR DELAY '00:00:09.000' ;

ROLLBACK TRANSACTION;
```

Sesión 2


```
SET TRANSACTION ISOLATION LEVEL READ COMMITED

BEGIN TRANSACTION

SELECT IdAsiento
FROM VueloAsiento
WHERE idVuelo = 'GA4561B' AND
      estadoAsiento = 'D';

COMMIT TRANSACTION
```

Se percibe un retraso en la ejecución de la Sesión **2**, ya que la Sesión **1** está bloqueando filas y la Sesión **2** debe esperar a que finalice esa transacción a fin de ejecutar la lectura. La Sesión **2** obtiene:



	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

Conclusión: No hay dirty reads.

Ejemplo RC2	
Isolation Level	Read Committed
Situación a evaluar	Non-repeatable reads

Supongamos el mismo escenario del ejemplo **S2**:

Sesión 1

```

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION

    SELECT IdAsiento
       FROM VueloAsiento
      WHERE idVuelo = 'GA4561B' AND
             estadoAsiento = 'D';

    WAITFOR DELAY '00:00:10.000' ;

    SELECT IdAsiento
       FROM VueloAsiento
      WHERE idVuelo = 'GA4561B' AND
             estadoAsiento = 'D';

COMMIT TRANSACTION

```

Sesión 2

```

BEGIN TRANSACTION;

    UPDATE VueloAsiento
       SET estadoAsiento = 'O'
      WHERE idVuelo = 'GA4561B' AND
             idAsiento = 'AR02';

COMMIT TRANSACTION;

```

Lo primero que notamos es que la Sesión **2** se ejecuta inmediatamente. Esto se debe a que no encuentra restricciones para llevar a cabo el **UPDATE**.

Esto se debe a que la Sesión **1** no ha establecido locks sobre las tuplas que está leyendo mientras dura su propia la transacción. La sesión **1** obtiene:

Como vemos, la Sesión **1** obtiene dos resultados diferentes para el mismo query, aún dentro de la misma transacción.



	IdAsiento
1	AL01
2	AL02
3	AR01
4	AR02

	IdAsiento
1	AL01
2	AL02
3	AR01

Conclusión: Hay non-repeatable reads.

Ejemplo RC3	
Isolation Level	Read Committed
Situación a evaluar	Phantom reads

Supongamos el mismo escenario del ejemplo **S3**:

Sesión 1

```

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION

    SELECT IdAsiento
       FROM VueloAsiento
      WHERE idVuelo = 'GA4561B' AND
             estadoAsiento = 'D';

    WAITFOR DELAY '00:00:10.000' ;

    SELECT IdAsiento
       FROM VueloAsiento
      WHERE idVuelo = 'GA4561B' AND
             estadoAsiento = 'D';

COMMIT TRANSACTION

```

Sesión 2

```

BEGIN TRANSACTION;

    INSERT INTO VueloAsiento VALUES ('GA4561B', 'ZZ01', 'D')

COMMIT TRANSACTION;

```

Podemos notar que la Sesión **2** se ejecuta inmediatamente. Esto se debe a que no encuentra restricciones para llevar a cabo el `INSERT`.

A continuación el resultado de las dos lecturas de la Sesión **1**:

Results		Messages	
	IdAsiento		
1	AL01		
2	AL02		
3	AR01		
4	AR02		

	IdAsiento		
1	AL01		
2	AL02		
3	AR01		
4	AR02		
5	ZZ01		A

En el segundo `SELECT`, la Sesión **1** obtiene una phantom tuple (**A**).

Conclusión: Hay phantom reads.

Ejercicio 1

Realice exhaustivamente todos los ejercicios bajo PostgreSQL.
Para ello cree dos scripts:

Script 1

```
-----
----- Sesion 1 -----
-----

----- Sesion 1 -Ejemplo S1 -----
-----
....

----- Sesion 1 -Ejemplo S2 -----
-----
...

etc.
```

Script 2

```
-----
----- Sesion 2 -----
-----

----- Sesion 2 -Ejemplo S1 -----
-----
...

----- Sesion 2 -Ejemplo S2 -----
-----

etc.
```

Prerequisitos

```
CREATE TABLE Vuelo
(
    idVuelo VARCHAR (8),
    fecha DATE,
    PRIMARY KEY (idVuelo)
)
```

```
INSERT INTO vuelo
SELECT 'GA4561B', '2015/10/12';
```

```
CREATE TABLE VueloAsiento
(
    idVuelo VARCHAR (8),
    idAsiento VARCHAR(6),
    estadoAsiento CHAR(1) CHECK (estadoAsiento IN ('O', 'D'))
    PRIMARY KEY (idVuelo, idAsiento)
)
```

```
INSERT INTO VueloAsiento
SELECT 'GA4561B', 'AL01', 'D';
```

```
INSERT INTO VueloAsiento
SELECT 'GA4561B', 'AL02', 'D';
```

```
INSERT INTO VueloAsiento
SELECT 'GA4561B', 'AR01', 'D';
```

```
INSERT INTO VueloAsiento
SELECT 'GA4561B', 'AR02', 'D';
```

```
CREATE TABLE Cantidad
(
    idVuelo VARCHAR (8) PRIMARY KEY,
    cantLibres Integer NULL
)
```

```
INSERT INTO Cantidad (idVuelo) VALUES ('GA4561B')
```

Scripts de utilidad

----- Reset -----

```
UPDATE VueloAsiento
  SET estadoAsiento = 'D'
  WHERE idVuelo = 'GA4561B' AND
        idAsiento = 'AR02';
```

```
DELETE VueloAsiento WHERE idAsiento = 'ZZ01'
```

Entregue los scripts y las conclusiones.