

Parcial 2 SID 2

Integrantes:

- Santiago Valencia García (A00395902)
- Danna López Muñoz (A00395625)
- Pablo Fernando Pineda P. (A00395831)

Caso 3 (1.5 puntos):

En este caso, el dueño de la cuenta origen hace un retiro de 60.000 por cajero electrónico simultáneamente con el traslado de fondos. Dicho retiro se identificará como la transacción T_j ejecutada desde otro banco.

Realice las siguientes tareas en el orden indicado:

1. Siguiendo la estrategia A caso 1, ejecute las operaciones de T_i hasta el paso 1
2. Ejecute T_j actualizando el valor directamente con una instrucción UPDATE comprometiéndola inmediatamente (esta simulación implica que dicha transacción consultó el saldo al mismo tiempo que la transacción de traslado de fondos pero terminó antes que la misma)
3. Continúe con el paso 2 de T_i hasta el final.
4. ¿Qué pasa con los valores finales de la cuenta origen y destino?
5. Restaure los valores y repita desde el inicio siguiendo la estrategia B. Varíe el comportamiento de T_j ?
6. Compare los resultados. Explique mediante esquema de planificación de transacciones.
7. Restaure los montos iniciales antes de proseguir con el siguiente caso.

Desarrollo

Saldos iniciales en banco A y B

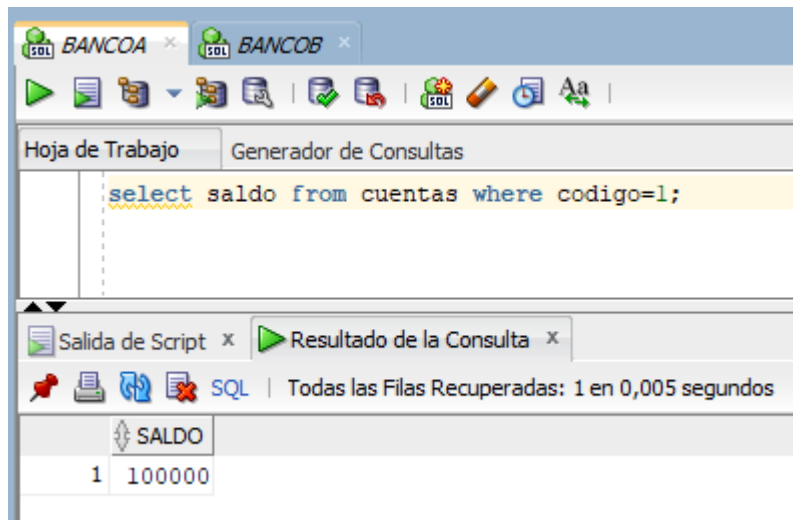
CODIGO	SALDO
1	100000
2	150000

1. Siguiendo la estrategia A caso 1, ejecute las operaciones de T_i hasta el paso 1

1.1 Consultar si existe disponibilidad de fondos en la cuenta origen con una operación SELECT

En banco A:

select saldo from cuentas where codigo=1;



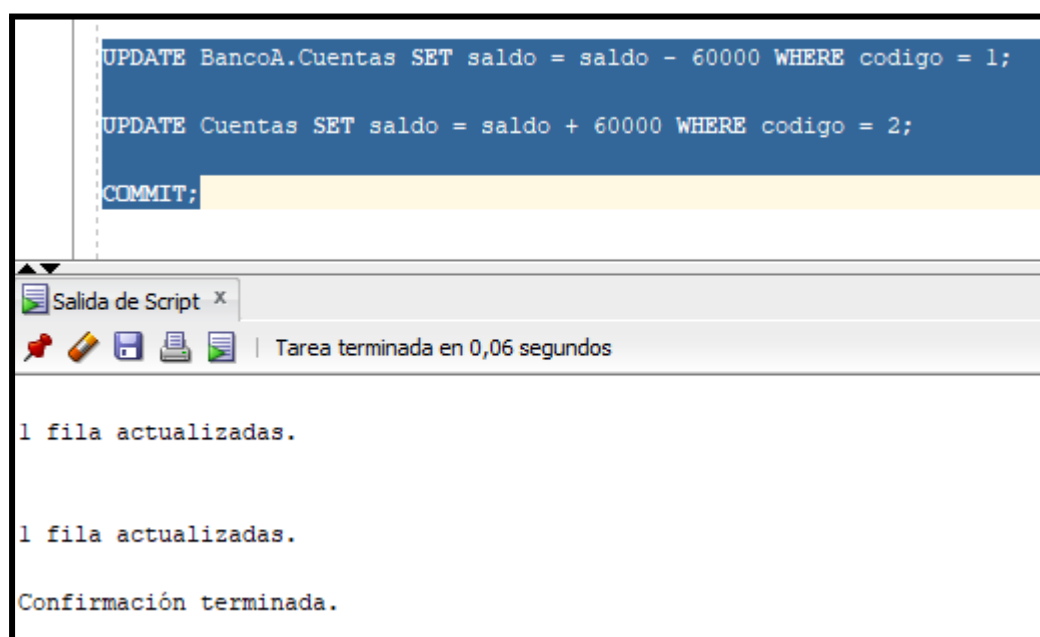
2. Ejecute Tj actualizando el valor directamente con una instrucción UPDATE comprometiéndola inmediatamente

En banco B:

UPDATE BancoA.Cuentas SET saldo = saldo - 60000 WHERE codigo = 1;

UPDATE Cuentas SET saldo = saldo + 60000 WHERE codigo = 2;

COMMIT;



3. Continúe con el paso 2 de Ti hasta el final.

En banco A:

3.1: Transferir el dinero a la cuenta en el otro banco sumando el valor actual más el monto transferido con una operación UPDATE.

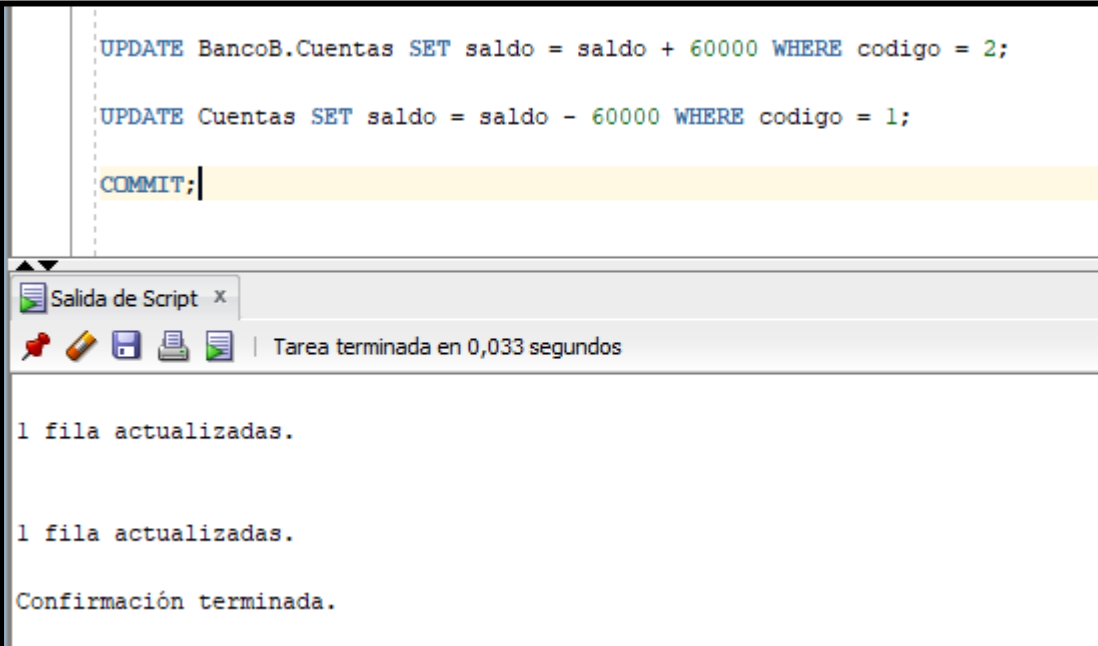
```
UPDATE BancoB.Cuentas SET saldo = saldo + 60000 WHERE codigo = 2;
```

3.2 Si la operación anterior es exitosa restar de la cuenta la suma transferida ejecutando una operación UPDATE.

```
UPDATE Cuentas SET saldo = saldo - 60000 WHERE codigo = 1;
```

3.3 Comprometer la transacción Ti con la instrucción COMMIT.

```
COMMIT;
```



```
UPDATE BancoB.Cuentas SET saldo = saldo + 60000 WHERE codigo = 2;

UPDATE Cuentas SET saldo = saldo - 60000 WHERE codigo = 1;

COMMIT;
```

Salida de Script x

Tarea terminada en 0,033 segundos

1 fila actualizadas.

1 fila actualizadas.

Confirmación terminada.

4. ¿Qué pasa con los valores finales de la cuenta origen y destino?

Banco A:

CODIGO	SALDO
1	-20000
2	150000

Banco B:

CODIGO	SALDO
1	100000
2	270000

En la cuenta del banco A, el saldo final queda en -20000 debido a que la transacción de transferencia al banco B se procesó después de la validación inicial del saldo. Esto significa que, aunque el saldo ya estaba comprometido por un retiro simultáneo, la transferencia también fue ejecutada, lo que permitió que el banco B recibiera los fondos antes de que el retiro se completara. Como resultado, el banco A terminó con un saldo negativo, ya que ambas operaciones utilizaron el mismo dinero de forma concurrente.

5. Restaure los valores y repita desde el inicio siguiendo la estrategia B. ¿Varió el comportamiento de Tj ?

Saludos restaurados en los dos bancos:

CODIGO	SALDO
1	100000
2	150000

5.1 Ejecute Tj actualizando el valor directamente con una instrucción UPDATE comprometiéndola inmediatamente

En Banco B:

```
UPDATE BancoA.Cuentas SET saldo = saldo - 60000 WHERE codigo = 1;
```

```
UPDATE Cuentas SET saldo = saldo + 60000 WHERE codigo = 2;
```

```
COMMIT;
```

```
1 fila actualizadas.  
  
1 fila actualizadas.  
  
Confirmación terminada.
```

Continúe con el paso 2 de Ti hasta el final.

5.2 Restar el dinero a transferir de la cuenta origen con una operación UPDATE.

en Banco A:

```
UPDATE Cuentas SET saldo = saldo - 60000 WHERE codigo = 1;
```

5.3 Si la operación no es exitosa devolver a la cuenta origen a su estado inicial. De lo contrario sumar al saldo de la cuenta destino el valor a transferir con una instrucción UPDATE.

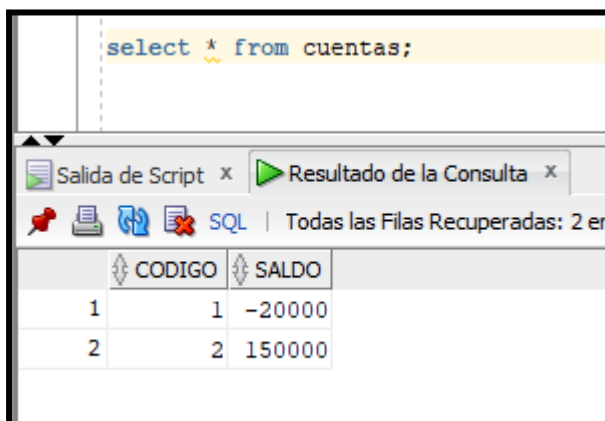
```
UPDATE BancoB.Cuentas SET saldo = saldo + 60000 WHERE codigo = 2;
```

5.4 Comprometer la transacción Tj con la instrucción COMMIT.

en banco B:

```
COMMIT;
```

en banco A consultamos y queda:



The screenshot shows a SQL query editor with the query `select * from cuentas;` and its results. The results are displayed in a table with two columns: CODIGO and SALDO. The first row has CODIGO 1 and SALDO -20000. The second row has CODIGO 2 and SALDO 150000.

	CODIGO	SALDO
1	1	-20000
2	2	150000

Se debe revertir la transacción porque no puede haber valores negativos:

en banco A:

```
ROLLBACK;
```

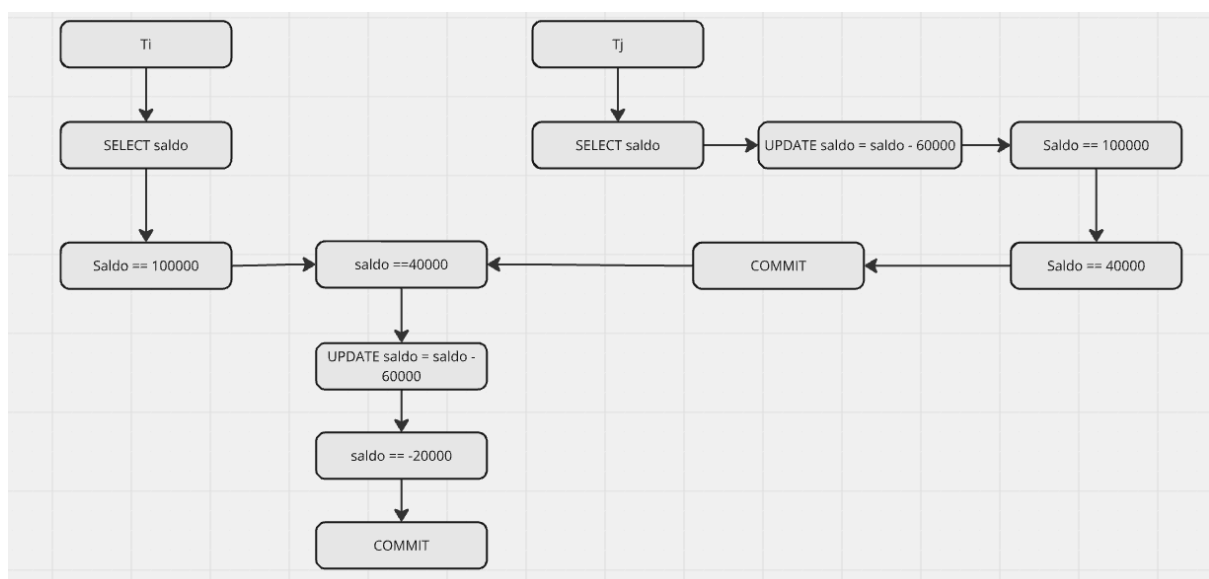
<pre>ROLLBACK; select * from cuentas;</pre>	
Salida de Script x	Resultado de la Consulta x
<div> <div>SQL</div> <div>Todas las Filas Recuperadas: 2 en 0,002 segundos</div> </div>	
CODIGO	SALDO
1	40000
2	150000

Con la estrategia B, si el retiro no se puede completar debido a la falta de fondos suficientes, la operación en la cuenta origen se revierte, lo que impide que se lleve a cabo la transferencia. Esto garantiza que la cuenta 1 mantenga un saldo de 40.000 en lugar de quedar en -20.000, lo que habría sucedido si no se hubiera realizado la verificación de fondos antes de proceder con la transacción.

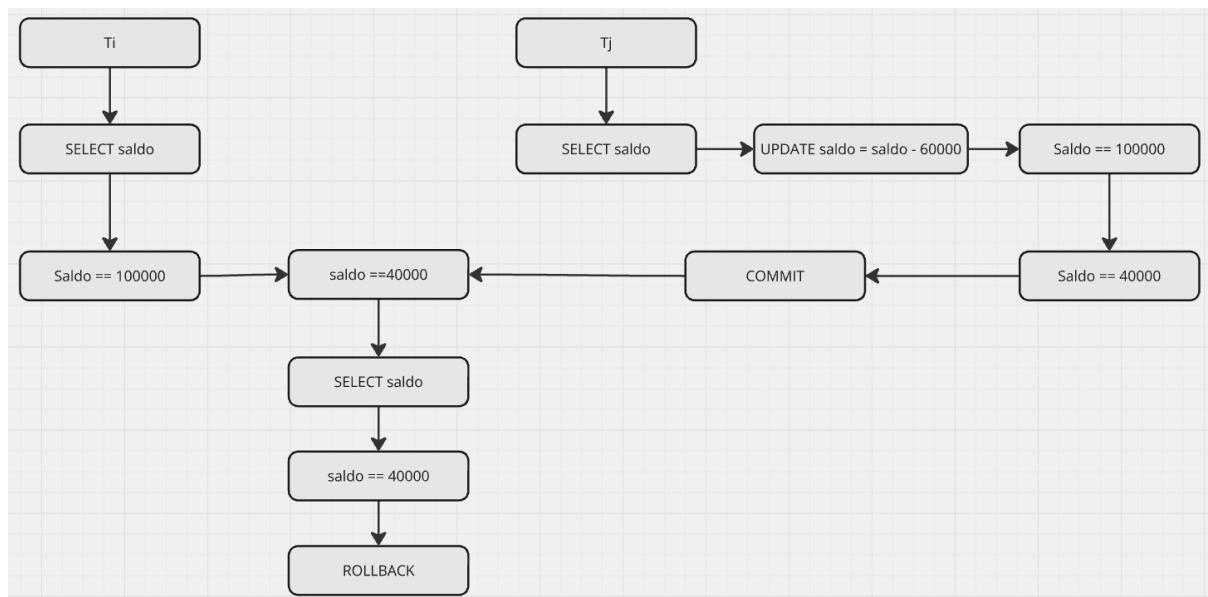
6. Compare los resultados. Explique mediante esquema de planificación de transacciones.

La diferencia radica en que, al realizar el retiro, la validación de la transacción se efectúa al finalizar. Si la validación falla, la transacción se revierte. Esto significa que no se permite completar el retiro, ya que el sistema es capaz de detectar el cambio realizado por la transacción Tj.

estrategia A:



estrategia B:



7. Restaure los montos iniciales antes de proseguir con el siguiente caso.

en ambos bancos:

UPDATE Cuentas SET saldo = 100000 WHERE codigo = 1;

UPDATE Cuentas SET saldo = 150000 WHERE codigo = 2;

Conclusiones generales

Atomicidad:

- Estrategia A:

La atomicidad puede verse comprometida porque Ti consulta el saldo, luego transfiere el dinero al otro banco, y finalmente resta de la cuenta origen.

Si Tj se ejecuta entre estos pasos, podría llevar a un estado inconsistente o a un saldo negativo.

- Estrategia B:

Mejora la atomicidad al restar primero de la cuenta origen antes de transferir al banco destino.

Si Tj se ejecuta después de la resta, no afectará la transferencia de Ti.

Si Tj se ejecuta antes, Ti podría fallar por falta de fondos, preservando la atomicidad.

Consistencia:

La ejecución concurrente de Ti y Tj puede llevar a un estado inconsistente si no se manejan adecuadamente. Por ejemplo, si el saldo inicial es 100,000, una transferencia de 60,000 (Ti) y un retiro de 60,000 (Tj) simultáneos podrían resultar en un saldo negativo, violando la regla de negocio de los saldos positivos.

La Estrategia B tiene más probabilidades de mantener la consistencia al verificar y actualizar el saldo de la cuenta origen antes de proceder con la transferencia.

Aislamiento:

El aislamiento es débil, ya que Ti y Tj pueden interferir entre sí. Ti podría leer un saldo que ya no es válido porque Tj lo ha modificado, o viceversa. Esto puede llevar a decisiones incorrectas sobre la disponibilidad de fondos. Ni la Estrategia A ni la B proporcionan un aislamiento fuerte sin mecanismos adicionales como bloqueos o control de concurrencia.

Durabilidad:

Ambas estrategias utilizan COMMIT para hacer los cambios permanentes. Sin embargo, la durabilidad podría verse afectada si el sistema falla entre la ejecución de Ti y Tj. En la Estrategia A, si el sistema falla después de transferir al banco destino pero antes de actualizar la cuenta origen, podría haber una inconsistencia duradera. La Estrategia B es más robusta en términos de durabilidad, ya que actualiza primero la cuenta origen.

Caso 4 (1.5 puntos):

Se desea mejorar el funcionamiento de las transacciones para el caso anterior. Para esto se hará uso de las instrucciones LOCK y SELECT ... FOR UPDATE (ver Anexo A para la sintaxis de LOCK).

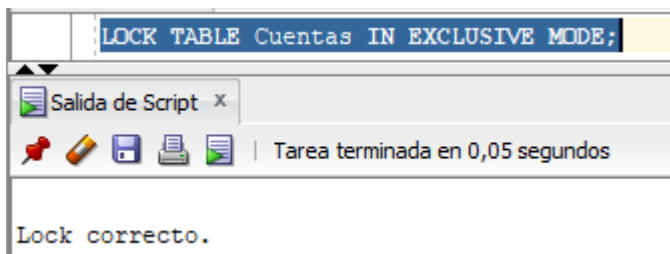
Realice las siguientes tareas en el orden indicado:

1. Un bloqueo en modo exclusivo para la tablas de cuentas utilizando la instrucción LOCK sin la opción final NOWAIT.
2. Ejecute T_i hasta el paso 3, es decir, sin comprometerla.
3. Inicie T_j desde el otro banco intentando bloquear la tabla de cuentas con la opción NOWAIT. Que pasa con esta transacción?
4. Comprometa T_i .
5. Intente nuevamente iniciar T_j . Que pasa ahora?
6. Comprometa T_j desde el otro banco.
7. Consulte el valor de saldo de la cuenta desde su banco.
8. Describa, mediante el esquema de planificación de transacciones lo que ocurrió.
9. Cómo denominaría el tipo de bloqueo que utilizó?

Desarrollo

1. bloqueo en modo exclusivo para la tablas de cuentas utilizando la instrucción LOCK sin la opción final NOWAIT

LOCK TABLE Cuentas IN EXCLUSIVE MODE;



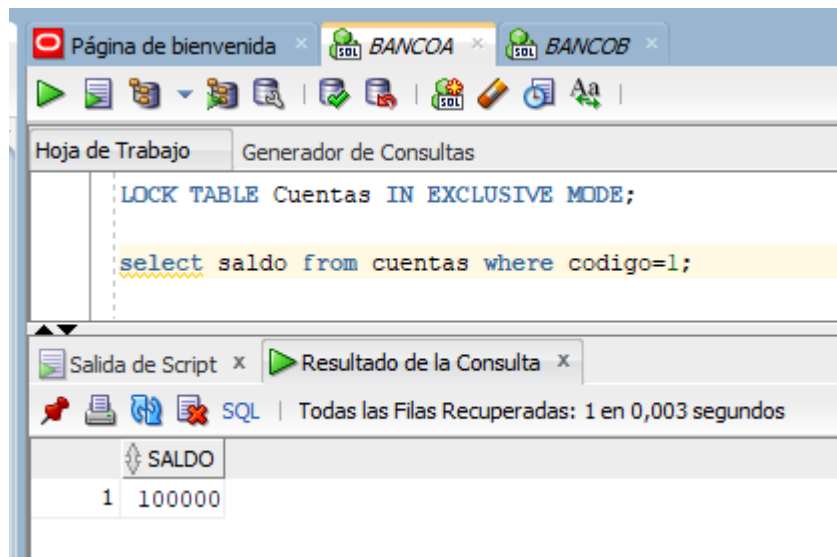
(Se bloquea la tabla cuentas para que nadie más acceda a ella)

2. Ejecute T_i hasta el paso 3, es decir, sin comprometerla.

En banco A:

2.1 Consultar si existe disponibilidad de fondos en la cuenta origen con una operación SELECT 2.

select saldo from cuentas where codigo=1;

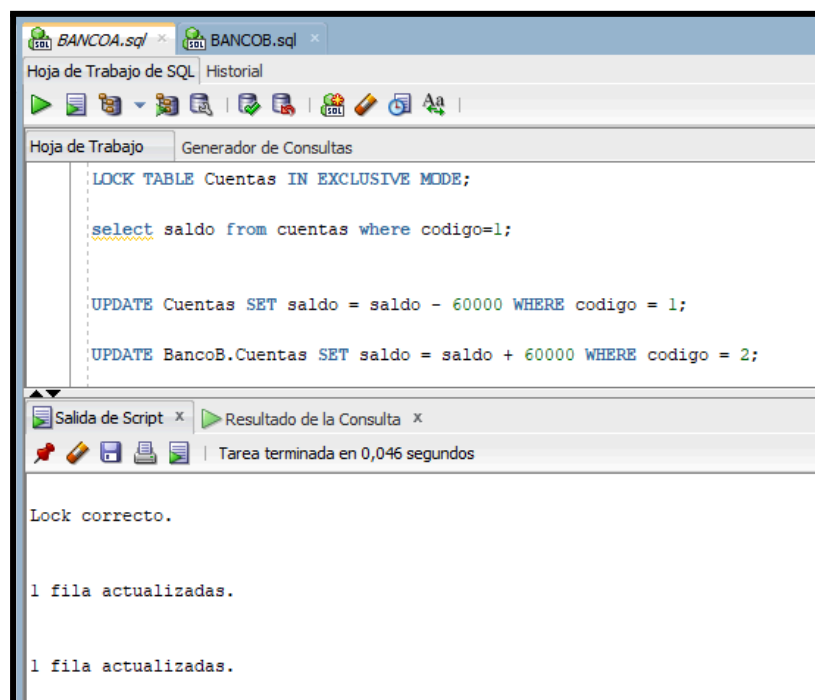


2.2 Restar el dinero a transferir de la cuenta origen con una operación UPDATE.

UPDATE Cuentas SET saldo = saldo - 60000 WHERE codigo = 1;

2.3 Si la operación no es exitosa devolver a la cuenta origen a su estado inicial. De lo contrario sumar al saldo de la cuenta destino el valor a transferir con una instrucción UPDATE.

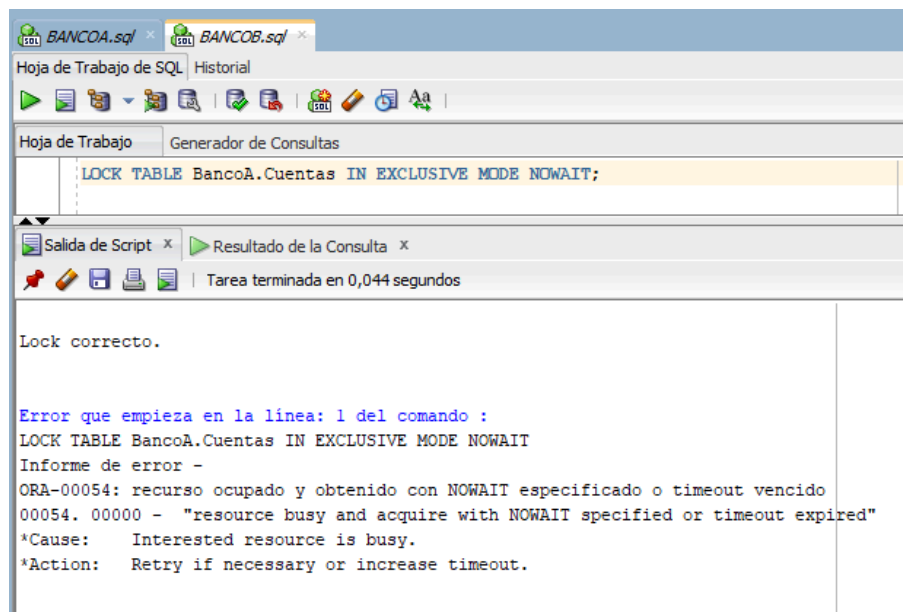
UPDATE BancoB.Cuentas SET saldo = saldo + 60000 WHERE codigo = 2;



3. Inicie Tj desde el otro banco intentando bloquear la tabla de cuentas con la opción NOWAIT. ¿Qué pasa con esta transacción?

En Banco B:

LOCK TABLE Cuentas IN EXCLUSIVE MODE NOWAIT;

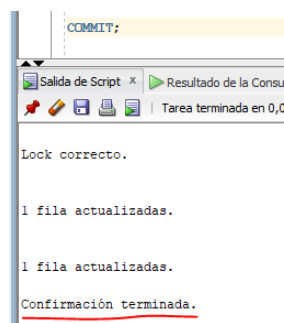


Sale un error, porque la tabla la tiene bloqueada otro usuario el cual es el banco A, así que no se ejecuta la instrucción en el banco B.

4. Comprometa a Ti.

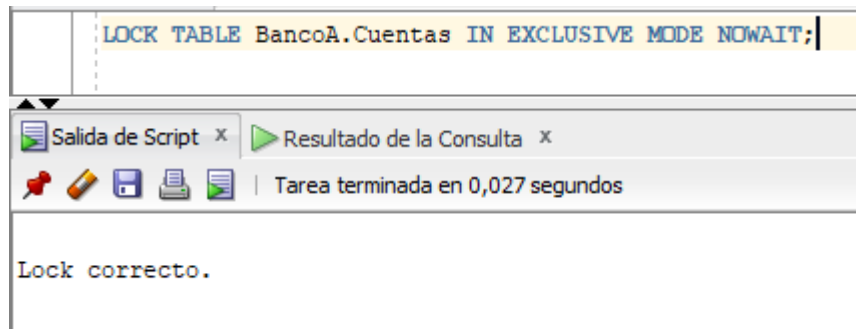
En Banco A:

COMMIT;



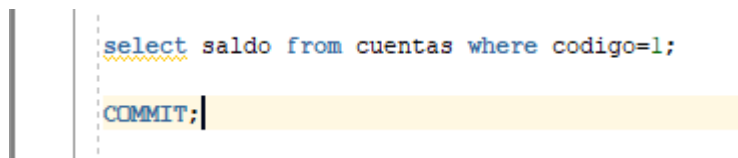
5. Intente nuevamente iniciar Tj. ¿Qué pasa ahora?

En Banco B:

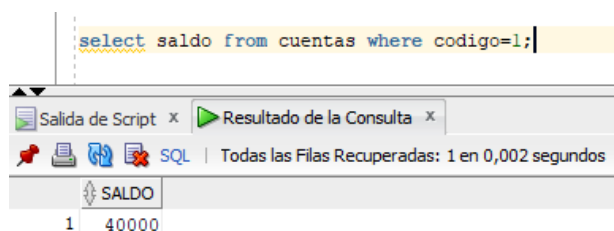


Ejecuta correctamente el LOCK debido a que la otra transacción ya terminó y al comprometer Ti se liberó el bloqueo de la tabla.

6. Comprometa Tj desde el otro banco.



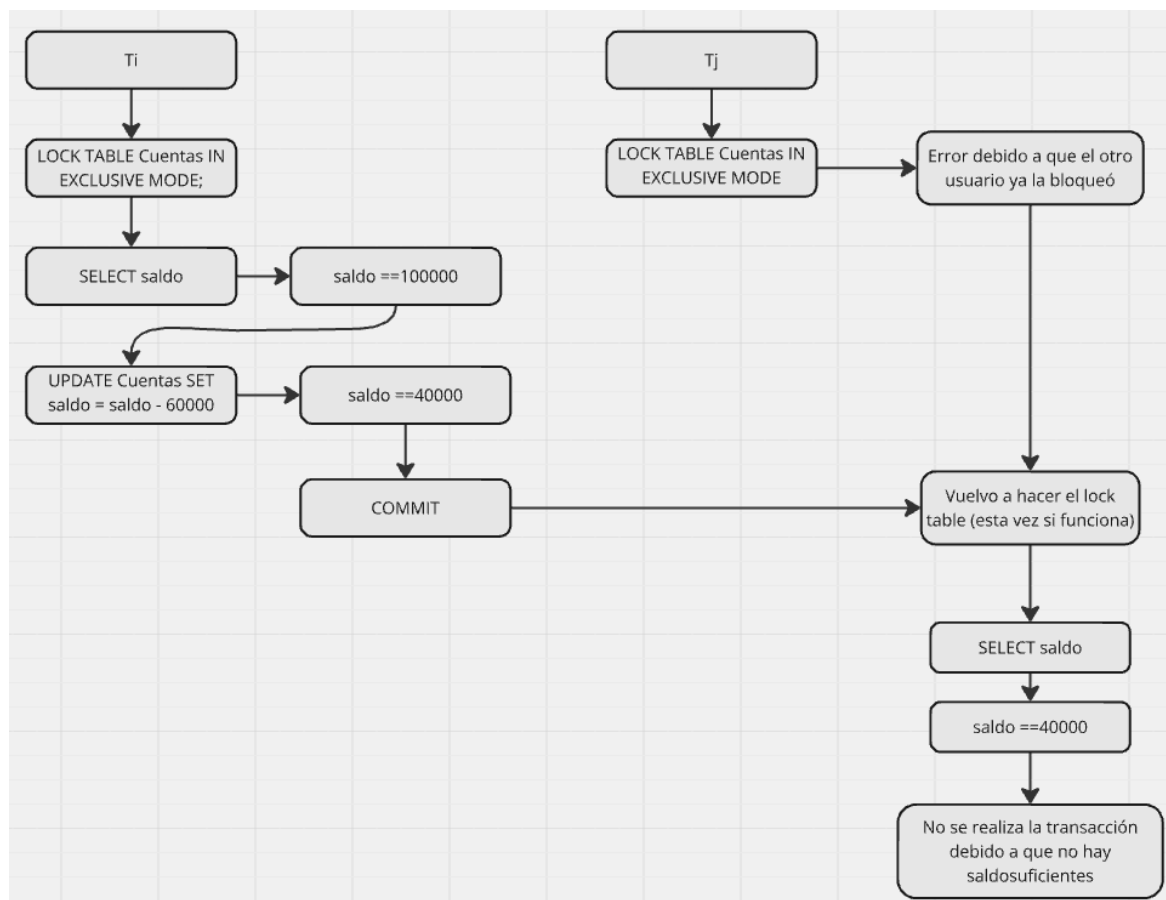
7. Consulte el valor de saldo de la cuenta desde su banco.



Por un lado, se detectó que la transacción no se realiza por saldo insuficiente y que el saldo de es válido a pesar de que la transacción se ejecutó al mismo tiempo.

8. Describa, mediante el esquema de planificación de transacciones, lo que ocurrió.

El bloqueo exclusivo establecido por la transacción T_i asegura que ningún otro usuario o sesión pueda interactuar con la tabla CUENTAS hasta que T_i haya finalizado, lo que obliga a los demás a esperar o abandonar sus transacciones. En consecuencia, la transacción T_j debe detenerse hasta que T_i concluya, o simplemente desistir de continuar. Al final, T_j no realiza modificaciones, ya que desde el inicio identificó que no había suficiente saldo para llevar a cabo la operación.



9. ¿Cómo denominaría el tipo de bloqueo que utilizó?

El tipo de bloqueo utilizado es un bloqueo exclusivo (exclusive lock). Este tipo de bloqueo impide que cualquier otra transacción pueda leer o modificar los datos bloqueados mientras la transacción que lo aplicó no haya finalizado, garantizando que los recursos sean utilizados por una sola transacción a la vez.

Conclusiones generales

Aislamiento:

El LOCK TABLE en modo exclusivo previene que otras transacciones modifiquen o incluso lean la tabla mientras la transacción actual está en progreso.

Esto garantiza un alto nivel de aislamiento

Atomicidad:

El bloqueo exclusivo permite que todas las operaciones dentro de Ti se ejecuten como una unidad atómica.

Ninguna otra transacción puede interferir entre los pasos de Ti, asegurando que o bien todas las operaciones se completen, o ninguna lo haga.

Si Tj intenta obtener un bloqueo con NOWAIT y falla, se asegura que no interferirá con la atomicidad de Ti.

Una vez que Ti se compromete (COMMIT), libera el bloqueo, permitiendo que Tj proceda, manteniendo así la atomicidad de ambas transacciones.

Consistencia:

El uso de LOCK TABLE en modo exclusivo ayuda a mantener la consistencia de los datos al prevenir que otras transacciones modifiquen la tabla durante la ejecución de Ti.

Esto asegura que todas las reglas de integridad y restricciones de la base de datos se mantengan durante la transacción.

Al finalizar Ti y liberar el bloqueo, la base de datos pasa de un estado consistente a otro estado consistente, cumpliendo así con esta propiedad.

Durabilidad:

El LOCK por sí mismo no garantiza la durabilidad, pero trabaja en conjunto con el COMMIT para asegurarla.

Caso 5 (2 puntos):

Proponga un caso de uso en donde se evidencie y pueda demostrar las 4 propiedades ACID sobre este mismo esquema. Demuestre rigurosamente.

Un sistema de reservas de vuelos permite a los pasajeros seleccionar asientos disponibles en vuelos. Cada vuelo tiene una lista de asientos con dos posibles estados: '**libre**' o '**reservado**'. Se desea garantizar que:

1. Un asiento solo puede ser reservado una vez.
2. Si se produce un fallo durante la reserva, el sistema debe revertir los cambios para evitar inconsistencias.
3. Dos personas no pueden reservar el mismo asiento al mismo tiempo.
4. Una vez confirmada una reserva, debe ser permanente

Configuración del ambiente de trabajo

1. Cree dos tablas, una de vuelos donde cada registro cuenta con el id del vuelo y el destino de este, y otra tabla de asientos, sus registros cuentan con el id del asiento, el id del vuelo (llave foránea) y estado ('libre' o 'reservado').
2. Insertar los siguientes registros:

Tabla vuelos:

1, 'Bogotá - Medellín'

Tabla asientos:

1, 1, 'libre'

2, 1, 'libre'

3. Cada esquema de usuario en la base de datos representa a una agencia de viajes (User1 y User2) que gestiona las reservas. Asigne permisos para que ambos usuarios puedan modificar las tablas de asientos con la operación GRANT SELECT, UPDATE ON asientos TO UserY;

Se desea reservar el asiento con id **1** en el vuelo de Bogotá a Medellín. Consideraremos dos estrategias para la transacción, identificadas como T_1 .

Estrategia A:

1. Consultar la disponibilidad del asiento con id '1' con una operación SELECT.
2. Cambiar el estado del asiento a 'reservado' utilizando una operación UPDATE.
3. Si la operación es exitosa, confirmar la transacción utilizando la instrucción COMMIT.
4. Si la operación no es exitosa (por que ocurre un fallo, por ejemplo), utilizar ROLLBACK para revertir los cambios.

Estrategia B:

1. Consultar la disponibilidad del asiento con id '1' con la operación SELECT.
2. Intentar reservar el asiento cambiando su estado a 'reservado' mediante UPDATE.
3. Si la operación falla, revertir la transacción con la instrucción ROLLBACK.
4. Si la operación es exitosa, comprometerla con la instrucción COMMIT.

Realiza las siguientes tareas en el orden indicado:

1. Desde cualquiera de las agencias de viajes, realiza la estrategia B y verifique nuevamente con la operación SELECT que el asiento con el id '1' sigue disponible. Esto con el objetivo de apreciar que al no poder terminar una transacción, la coherencia de los datos se mantiene.
2. A continuación, desde cualquiera de las agencias realice la estrategia A y verifique que el asiento haya quedado reservado.
3. Restaure los valores de la tabla asientos a los iniciales antes de continuar.
4. Se sabe que dos personas no pueden reservar un asiento al mismo tiempo, en el esquema User1 realizará los dos primeros pasos de la estrategia A.
5. Luego, en el esquema User2 intente reservar el mismo asiento.
6. Comprometa la transacción desde el User1.
7. Intente comprometer la transacción desde el User2

8. Realiza una nueva transacción llamada T_2 , consultar la disponibilidad del asiento con id '2' con una operación SELECT y cambia el estado del asiento a 'reservado' con una operación UPDATE.
9. Comprometa la transacción T_2 utilizando la instrucción COMMIT; luego cierre la aplicación de Oracle.
10. Vuelva a ingresar y verifique que disponibilidad del asiento con id '2' con una operación SELECT. Debe aparecer que su estado es 'reservado'.

Desarrollo del caso:

Configuración del ambiente:

Se crean 3 usuarios: User1, User2 y Reservas. En Reservas estarán las tablas con los registros y User1 y User2 tendrán acceso a estos datos para realizar las operaciones correspondientes:

Código inicial de Reservas:

-- Crear la tabla de vuelos

```
CREATE TABLE vuelos (  
    id_vuelo NUMBER PRIMARY KEY,  
    destino VARCHAR2(50)  
);
```

-- Crear la tabla de asientos

```
CREATE TABLE asientos (  
    id_asiento NUMBER PRIMARY KEY,  
    id_vuelo NUMBER,  
    estado VARCHAR2(10),  
    CONSTRAINT fk_vuelo FOREIGN KEY (id_vuelo) REFERENCES  
vuelos(id_vuelo)  
);
```

-- Insertar los registros iniciales en las tablas

```
INSERT INTO vuelos (id_vuelo, destino) VALUES (1, 'Bogotá - Medellín');  
INSERT INTO asientos (id_asiento, id_vuelo, estado) VALUES (1, 1, 'libre');  
INSERT INTO asientos (id_asiento, id_vuelo, estado) VALUES (2, 1, 'libre');
```

-- Confirmar los cambios

```
COMMIT;
```

– Otorgar permisos

```
GRANT SELECT, UPDATE ON asientos TO User1;  
GRANT SELECT, UPDATE ON vuelos TO User1;
```

```
GRANT SELECT, UPDATE ON asientos TO User2;  
GRANT SELECT, UPDATE ON vuelos TO User2;
```

```
Table VUELOS created.
```

```
Table ASIENTOS created.
```

```
1 row inserted.
```

```
1 row inserted.
```

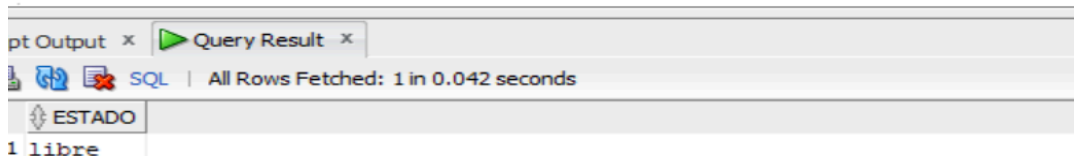
```
1 row inserted.
```

```
Commit complete.
```

Atomicidad: Esta propiedad asegura que las transacciones se completan en su totalidad o no se realizan en absoluto. Si una parte de la transacción falla, el sistema revertirá todos los cambios realizados en la transacción.

Ahora, desde **User1** se ejecutan los siguientes comandos:

```
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;  
  
UPDATE Reservas.asientos SET estado = 'reservado' WHERE id_asiento = 1;  
  
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;  
  
ROLLBACK;  
  
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;
```



The screenshot shows a database client window with a tab labeled 'Query Result'. Below the tab, it indicates 'All Rows Fetched: 1 in 0.042 seconds'. A table with one column named 'ESTADO' is displayed, containing a single row with the value '1 libre'.

```
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;
```

```
UPDATE Reservas.asientos SET estado = 'reservado' WHERE id_asiento = 1;
```

```
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;
```

```
ROLLBACK;
```

```
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;
```

En esta parte, podemos ver el cumplimiento de una de las propiedades ACID: la Atomicidad. Aquí, al reservar un asiento y luego hacer ROLLBACK (al simular un fallo), vemos que el estado del asiento vuelve a ser “libre”, debido a que si una parte de la transacción falla, el sistema revertirá los cambios realizados por la transacción.

ESTADO
1 reservado

Rollback complete.

Luego del ROLLBACK (fallo), el estado del asiento vuelve a ser “libre”, que era su estado original.



The screenshot shows a SQL IDE interface. At the top, a query is entered in a text area: `SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;`. Below the text area, there are tabs for 'Script Output' and 'Query Result'. The 'Query Result' tab is active, displaying a table with one row and one column. The table has a header 'ESTADO' and a single data row with the value '1 libre'. Above the table, a status bar indicates 'All Rows Fetched: 1 in 0.007 seconds'.

ESTADO
1 libre

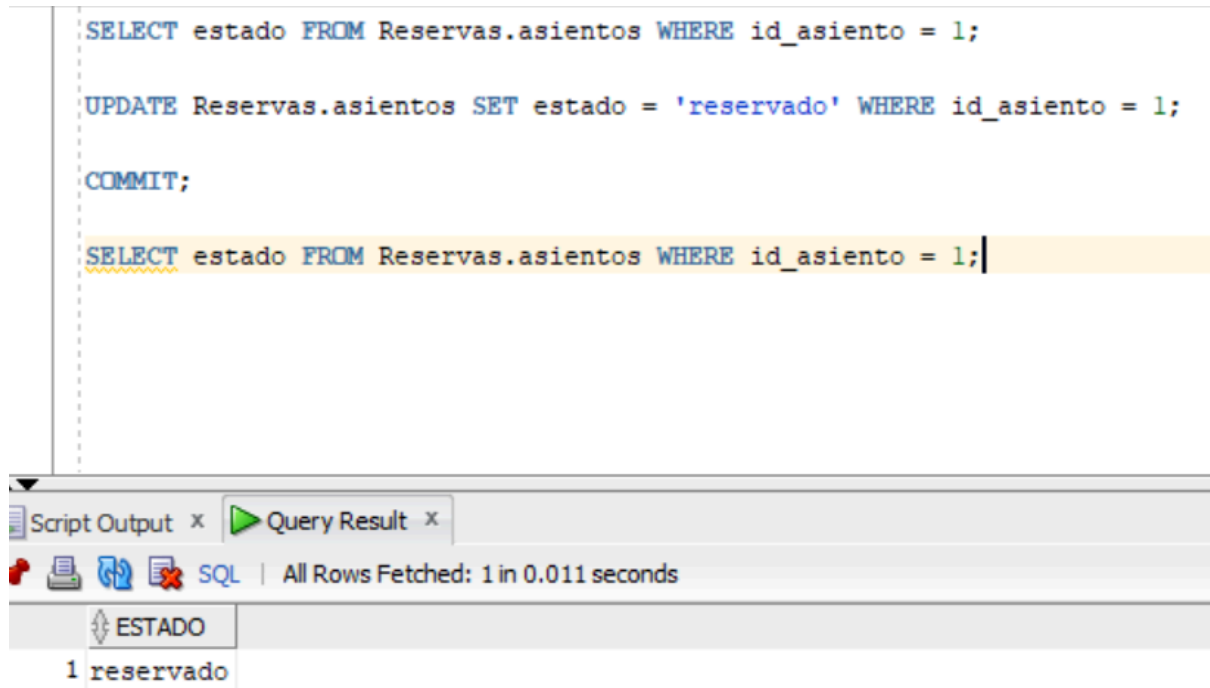
Ahora, para seguir con el caso, realizamos los mismos pasos, pero esta vez comprometemos la transacción:

```
SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;

UPDATE Reservas.asientos SET estado = 'reservado' WHERE id_asiento = 1;

COMMIT;

SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;
```



The screenshot shows a database query tool interface. The top pane contains four SQL statements: a SELECT query to check the state of seat 1, an UPDATE query to set it to 'reservado', a COMMIT statement, and a final SELECT query to verify the update. The bottom pane shows the 'Query Result' tab with a table titled 'ESTADO' containing one row: '1 reservado'. The status bar indicates 'All Rows Fetched: 1 in 0.011 seconds'.

ESTADO
1 reservado

Con lo anterior, aseguramos que este asiento ya esté en estado “reservado”.

Consistencia: Esta propiedad asegura que una transacción lleve al sistema de un estado válido a otro estado válido. Todas las reglas y restricciones de la base de datos deben cumplirse.

Cuando insertamos un registro de asiento como "libre" y se establece una relación de clave foránea con la tabla de vuelos, estamos garantizando que no se insertan asientos sin un vuelo asociado.

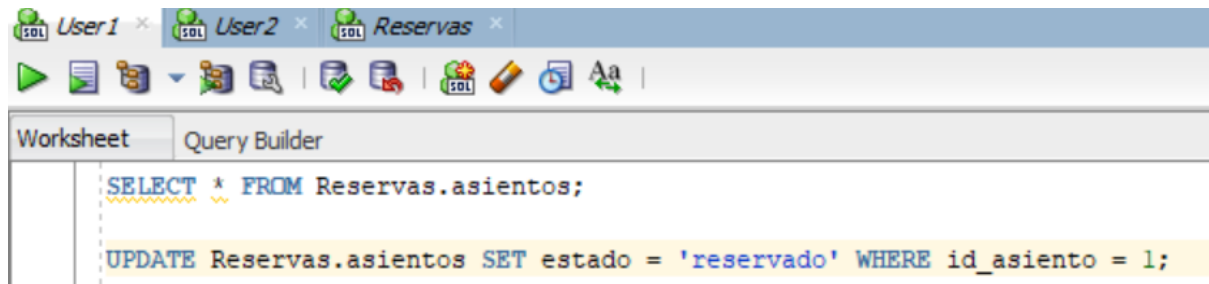
```
INSERT INTO asientos (id_asiento, id_vuelo, estado) VALUES (1, 1, 'libre');
```

Al no permitir que un asiento esté reservado sin una relación válida con un vuelo, se mantiene la consistencia de los datos.

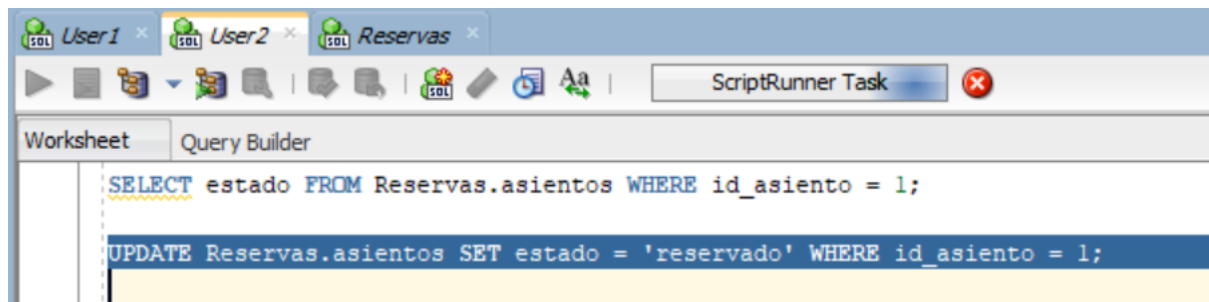
Aislamiento: Esta propiedad asegura que las transacciones concurrentes no interfieran entre sí. Cada transacción debe verse como si fuera la única que se ejecuta en el sistema.

Para justificar esta propiedad, vamos a realizar dos transacciones, una desde User1, el cual intentará reservar un asiento, y desde el User2 se intentará reservar el mismo asiento al mismo tiempo.

En **User1**:



Ahora, desde **User2**:



Al intentar reservar el mismo asiento al mismo tiempo, observamos que una de las transacciones no pudo finalizar hasta que la otra transacción fue confirmada. Esto se debe a que ambas transacciones intentaban modificar el mismo registro simultáneamente, lo que resultó en un bloqueo temporal. De este modo, se cumple la propiedad de aislamiento, ya que ninguna transacción puede interferir con otra mientras no se haya completado (con COMMIT o ROLLBACK), garantizando que dos usuarios no puedan realizar la misma acción sobre el mismo asiento al mismo tiempo.

Durabilidad: Esta propiedad asegura que una vez que una transacción ha sido confirmada (commit), los cambios son permanentes y sobrevivirán a fallos del sistema.

Al hacer COMMIT luego de reservar el asiento 2 por el User2, se asegura que el cambio en la información quede permanentemente dentro del sistema. Esto significa que una vez que se confirma la reserva, no se puede revertir, y el estado del asiento se mantendrá como "reservado", lo que es crítico para la funcionalidad del sistema de reservas.

```

SELECT estado FROM Reservas.asientos WHERE id_asiento = 1;

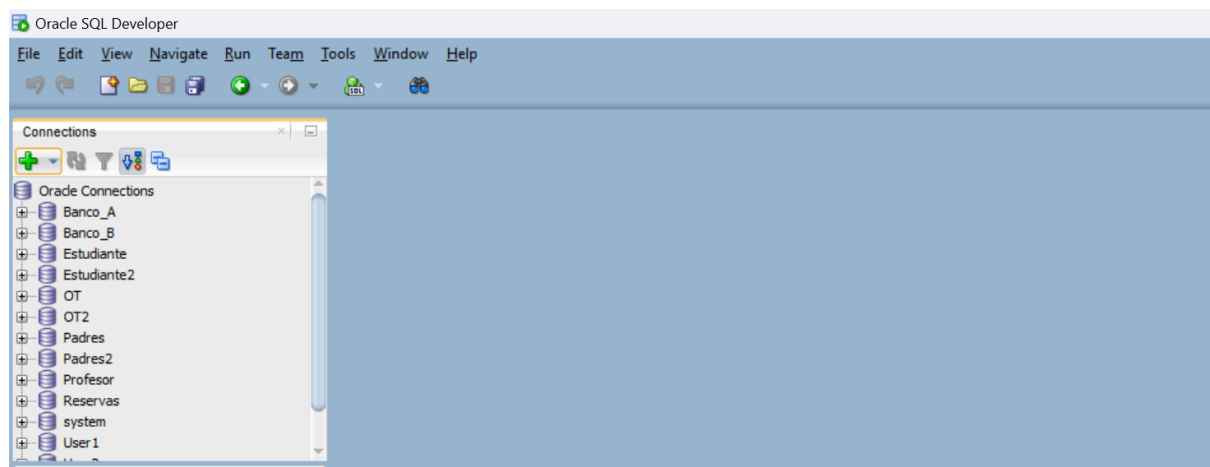
UPDATE Reservas.asientos SET estado = 'reservado' WHERE id_asiento = 2;

COMMIT;

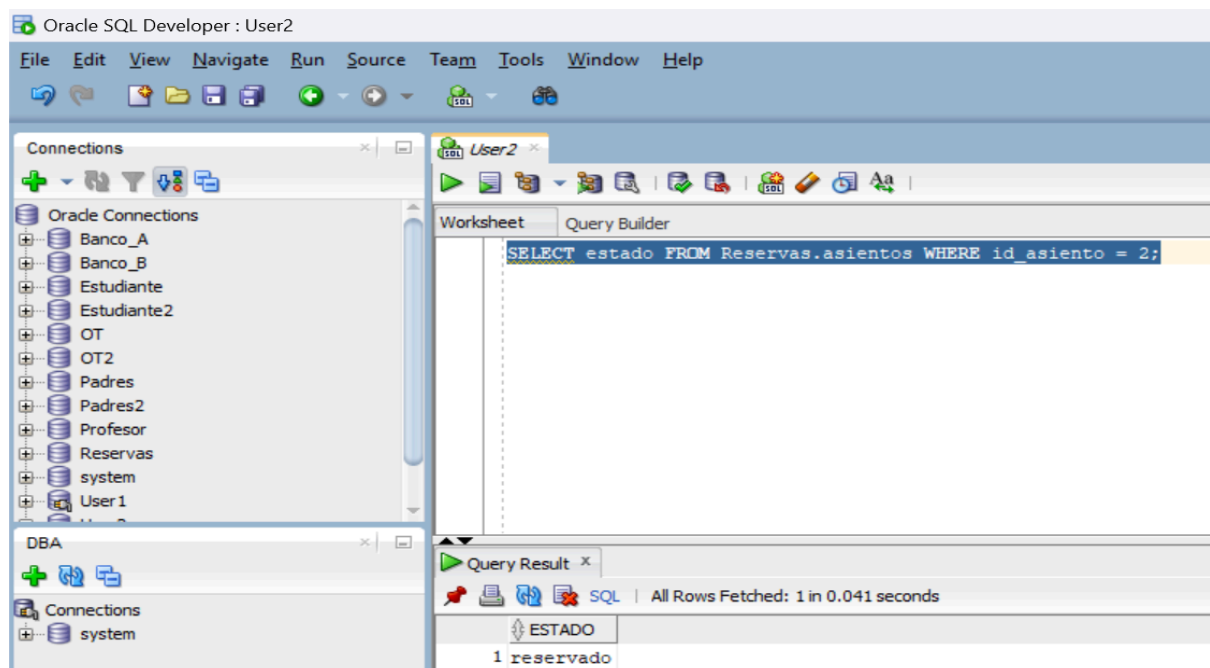
SELECT * FROM Reservas.asientos;

```

ID_ASIENTO	ID_VUELO	ESTADO
1	1	1 reservado
2	2	1 reservado



Luego de cerrar SQL Server y volver a verificar la información, vemos que se mantuvo correctamente en el sistema:



En esta práctica se demostraron las propiedades ACID a través de la simulación de transacciones en un sistema de reservas de vuelos:

Atomicidad: Se comprobó que las transacciones son indivisibles. Si una transacción falla, como en el caso de un ROLLBACK, todos los cambios se revierten, garantizando que el sistema no quede en un estado parcial.

Consistencia: La consistencia se mantuvo cuando las tablas de asientos y vuelos se crearon con claves foráneas. Esto asegura que un asiento no pueda existir sin estar asociado a un vuelo válido, cumpliendo las restricciones definidas en la base de datos.

Aislamiento: Cada transacción fue ejecutada de manera independiente, y los cambios no fueron visibles para otros usuarios hasta que se aplicó el COMMIT, evitando interferencias entre usuarios concurrentes.

Durabilidad: Los cambios confirmados mediante COMMIT persistieron en la base de datos incluso después de cerrar la sesión, garantizando que la información permanezca intacta ante fallos.