BASES DE DATOS

Transacciones: Posibles Situaciones

Profesor : Héctor Gómez Gauchía

http://www.itu.dk/people/pagh/IDB05/Transactions-examples-run.html

Usuario: Pepe (T1)	**Usuario: Pepe2 (T2)**

`SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE NAME ‘T1’;`
 	
`Transaction set.`
	`SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE NAME ‘T2’;`
	
	`Transaction set.`

`SQL> CREATE TABLE Primos (p INT);`
 	
`Table created.`	

`SQL> GRANT SELECT, UPDATE, INSERT ON Primos to Pepe2;`
 	
`Grant succeeded.`	

`SQL> SELECT * FROM Primos;`
 	
`no rows selected`	

	`SQL> SELECT * FROM Pepe.Primos;`
	
	`no rows selected`

`SQL> INSERT INTO Primos VALUES (41);`
 	
`1 row created.`	

	`SQL> INSERT INTO Pepe.Primos VALUES (43);`
	
	`1 row created.`

`SQL> SELECT * FROM Primos;`
 	
`         P`
`----------`
`        41`	

	`SQL> SELECT * FROM Pepe.Primos;`
	
	`         P`
	`----------`
	`        43`

`SQL> COMMIT;`
 	
`Commit complete.`	

	`SQL> SELECT * FROM Pepe.Primos;`
	
	`         P`
	`----------`
	`        43`
	
	`SQL> COMMIT;`
	
	`Commit complete.`

SQL> SELECT * FROM Pepe.Primos; P ---------- 41 43	
	SQL> SELECT * FROM Pepe.Primos; P ---------- 41 43
************ ************	********* *********
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED; Transaction set.	SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED; Transaction set.
SQL> INSERT INTO Primos VALUES (2); 1 row created.	
	SQL> SELECT * FROM Pepe.Primos; P ---------- 41 43 SQL> INSERT INTOPepe.Primos VALUES (2003); 1 row created. SQL> SELECT * FROM Pepe.Primos; P ---------- 41 43 2003
SQL> COMMIT; Commit complete.	
	SQL> SELECT * FROM Pepe.Primos; P ---------- 41 43 2 2003
SQL> SELECT * FROM Primos;	

P ---------- 41 43 2	SQL> COMMIT; Commit complete.
SQL> SELECT * FROM Primos; P ---------- 41 43 2 2003 SQL> INSERT INTO Primos VALUES (3); 1 row created. SQL> ROLLBACK; Rollback complete. SQL> SELECT * FROM Primos; P ---------- 41 43 2 2003	

Usuario:Pepe, (T1)	**Usuario:Pepe, (T2)**
SQL> CREATE TABLE TablaX (pk INT PRIMARY KEY); → por DEF a nivel Instrucción: set transaction read write Table created.	
SQL> INSERT INTO TablaX VALUES(1); 1 row created.	
	SQL> INSERT INTO TablaX VALUES (1);
commit;	→ SE QUEDA ESPERANDO al commit
	INSERT INTO TablaX VALUES (1) * ERROR at line 1: ORA-00001: unique constraint
SQL> INSERT INTO TablaX VALUES (2); 1 row created.	(PAGH.SYS_C0030509) violated

	`SQL> INSERT INTO  TablaX VALUES (3);`
	
	`1 row created.`
	
	`SQL> INSERT INTO  TablaX VALUES (2);`

`SQL> INSERT INTO  TablaX VALUES (3);`
 	
`INSERT INTO  TablaX VALUES (3)`
`            *`
`ERROR at line 1:`
`ORA-00060: deadlock detected while waiting for resource`
➔ No hace la operación 3, pero no aborta Trans	

`SQL> rollback;`
 	
`Rollback complete.`
 	
`SQL> select * from TablaX;`
 	
`        PK`
`----------`
`         1`
	➔ Se queda esperando hasta que T1 termine: Commit o rollback
	
	`1 row created.`

PROBLEMAS DE LA MULTIPROGRAMACIÓN (ver en teoría)

1.1 Example 1 **`NON-REPEATABLE-READ`**

Consistent image at statement level.

Session 2's second select statement does see Session 1's update because the update was not committed at the time the 2's select began.

Session 2's third select statement sees the result of the Session 1 update which is now committed. Example of a non-repeatable read.

```
                    Session 1                      Session 2
Time
  |                                      -------BEGIN TRANSACTION-
  |                                      select color from cust
  |                                      where id=500;
  |      -------BEGIN TRANSACTION ----     color
  |      -                                 -----
  |      update cust set color='blue'      red
  |      where id=500;
  |                                      select color from cust
  V                                      where id=500;
                                           color
         -----------COMMIT------------     -----
                                           red

         → usa TABLA cust(id, color)
                                         select color from cust
                                         where id=500;
                                           color
                                           -----
                                           blue
                                         -----------COMMIT--------
```

1.2 Example 2 `Phantom read.`

Session 1 has deleted the row but Session 2 does not see it, because the delete was not committed before either of 2's statements.

```
        T                Session 1                        Session 2
Time
  |                                            -------BEGIN TRANSACTION----
  |                                            select color from cust
  |                                            where id=500;
  |
  |      -------BEGIN TRANSACTION---             color
  |      delete from cust                        -----
  |      where id=500;                           red
  |
  V                                                ...
                                               select color from cust
                                               where id=500;

         -----------COMMIT----------             color
                                                 -----
         → usa TABLA cust(id, color)             red
                                               -----------COMMIT--------------
```

1.3 Example 3 : Session 2 ve sus cambios

A statement see updates made within its transaction. The second select sees changes made by this transaction even though they are not committed.

```
        T                Session 1                     Session 2
Time
  |                                           -------BEGIN TRANSACTION---
  |                                           select color from cust
  |                                           where id=500;
  |
  |                                             color
  |                                             -----
  |                                             red
  |
  V                                           update cust set
                                              color='blue'
                                              where id=500;

                                              select color from cust
                                              where id=500;
         → usa :
         TABLA cust(id, color)                  color
                                                -----
                                                blue
                                              -----------COMMIT----------
```

1.4 Example 4 `Read only transaction Session 2, Foto Fija`

2's second select does NOT see the results of session 1's update even though 1's update was committed before 2's select began. During a read-only transaction, statements use a image consistent to the point-in-time the transaction began, not the statement.

```
                     Session 1                        Session 2
Time
  |     -------BEGIN TRANSACTION-------
  |     update balances set amt=amt+50
  |     where custid=1000;                 -------BEGIN TRANSACTION-------
  |                                        set transaction read only;
  |     insert into trans
  |       (custid, txamt)                  select custid, amt from balances;
  |       values (1000, 50);
  |                                        no ve los cambios de Session 1
  V     -----------COMMIT--------------

                                           select custid, txamt
        → usa TABLAS: balances(custid,amt) from trans;
                      trans(custid,txamt)  no ve los cambios de Session 1
                                            incluso después de commit de
                                           Session 1
                                           -----------COMMIT--------------
```

1.5 Example 5 Muchos commit en la Session 1 no dejan construir imagen

ORA-1555 error. Session 1's numerous updates may prevent Oracle from reconstructing a consistent image for session 2.

```
                     Session 1                          Session 2
ime
  |     -------BEGIN TRANSACTION-------
  |     update cust set color='blue'         -------BEGIN TRANSACTION---
  |     where custid > 0 and custid <=2000;  select color from cust;
  |     -----------COMMIT--------------
  |     -------BEGIN TRANSACTION-------
  |     update cust set color='blue'
  |     where custid > 2000 and custid
  |     <=4000;
  V     -----------COMMIT--------------
        -------BEGIN TRANSACTION-------
        update cust set color='blue'         Possible Error:
        where custid > 400 and custid <=6000; ORA-1555 snapshot too old
        -----------COMMIT--------------      (rollback segment too
                                             small)
        ...

                                             -----------COMMIT----------
        → usa TABLA cust(id, color)
```

Example: **Escrituras conflictivas y actualizaciones perdidas en READ COMMITED .**

→Teniendo en cuenta los LOCKS automáticos de Oracle.
→ Controlar por programa estos problemas

Oracle®DatabaseConcepts11*g*Release2(11.2) E40540-01
`http://docs.oracle.com/cd/E11882_01/server.112/e40540/consist.htm#CNCPT020`

Table 9-2 shows a classic situation known as a lost update (see "Use of Locks"). The update made by transaction 1 is not in the table even though transaction 1 committed it. Devising a strategy to handle lost updates is an important part of application development.Table 9-2 Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

Session 1 (read commited)	**Session 2**	**Explanation**

Actualización Perdida
`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');`
 	
`   LAST_NAME          SALARY`
`------------- ----------`
`Banda                6200`
`Greene               9500`
		Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.

`SQL> UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';` **(por defecto Read Commited)** **- Lock BX a fila "banda"** **Multiversión Oracle** **- Lock BS a la tabla Employees**		Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is `READ COMMITTED`.
	`SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;` **Empieza a la fuerza una Trans. nueva**	Session 2 begins transaction 2 and sets the isolation level explicitly to `READ COMMITTED`.

	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');`
	`LAST_NAME          SALARY`
	`------------- ----------`
	`Banda                6200`
	`Greene               9500`
	
	no ve valor nuevo de Banda (actualizado por session1)
		Trans. 2 queries the salaries for Banda, Greene, and Hintz. Oracle uses read consistency to show the salary for Banda before the uncommitted update made by trans.1

	`SQL> UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';` **Actualiza bien, solo bloqueada fila "Banda" por session 1. Ahora BX en Greene**	Transaction 2 updates the salary for Greene successfully because transaction 1 locked

Session 1 (read commited)	**Session 2**	**Explanation**
		only the Banda row (see "Row Locks (TX)").
`SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');` **Lock BX a fila "Hintz"**		Transaction 1 inserts a row for employee Hintz, but does not commit.

	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');`
	`LAST_NAME          SALARY`
	`------------- ----------`
	`Banda                6200`
	`Greene               9900`
	
	no ve valor de "Banda" actualizado por Session 1
		Transaction 2 queries the salaries for employees Banda, Greene, and Hintz.
		
		Transaction 2 sees its own update to the salary for Greene. Transaction 2 does not see the uncommitted update to the salary for Banda or the insertion for Hintz made by transaction 1.

	`SQL> UPDATE employees SET salary = 6300 WHERE last_name = 'Banda';` **- "Banda" bloqueado BX por Session 1** **- Session 2 espera COMMIT de Session 1 por estar en read commited**	Transaction 2 attempts to update the row for Banda, which is currently locked by transaction 1, creating a conflicting write. Transaction 2 waits until transaction 1 ends.
`SQL> COMMIT;` **Libera los BX y BS**		Transaction 1 commits its work, ending the transaction.
	`1 row updated.   SQL>` **- "Banda" se que con valor de session 2 y pone BX de fila** **- "Banda" pierde su valor actualizado por Session 1: 7000** **- <u>Te interesa que sea así</u>?**	The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary

Session 1 (read commited)	**Session 2**	**Explanation**
		for Banda.
	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY - ------------ ---------- Banda 6300 Greene 9900 Hintz **Ve Hinz por estar en Read Committed**	Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. The Hintz insert committed by transaction 1 is now visible to transaction 2. Transaction 2 sees its own update to the Banda salary.
	COMMIT; **Libera los BX y BS**	Transaction 2 commits its work, ending the transaction.
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------- ---------- Banda 6300 Greene 9900 Hintz **Actualización Perdida** **Banda 7000**		Session 1 queries the rows for Banda, Greene, and Hintz. The salary for Banda is 6300, which is the update made by transaction 2. The update of Banda's salary to 7000 made by transaction 1 is now "lost."

Example: **Escrituras conflictivas en Serializable Isolation Level**

In the serialization isolation level, a transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself. A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.

Table 9-3 Read Consistency and Serialized Access Problems in Serializable Transactions

Session 1 (→read commited)	**Session 2**	**Explanation**
`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');` `LAST_NAME SALARY` `------------- ----------` `Banda 6200` `Greene 9500`		Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.
`Empieza Trans T.1` `SQL> UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';` **`BX en fila 'Banda'`**		Session 1 begins transaction 1 by updating the Banda salary. The default isolation level for is `READ COMMITTED`.
	`SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;` **`Empieza Trans T.2`**	Session 2 begins transaction 2 and sets it to the `SERIALIZABLE` isolation level.
	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');` `LAST_NAME SALARY` `------------- ----------` `Banda 6200` `Greene 9500` **`no ve valor de "Banda" de la Session 1`**	Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1.
	`SQL> UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';` **`Va bien; solo Banda bloqueada BX a Greene`**	Transaction 2 updates the Greene salary successfully because only the

Session 1 (→read commited)	**Session 2**	**Explanation**
		Banda row is locked.
SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK'); **crea Hintz (y BX)**		Transaction 1 inserts a row for employee Hintz.
SQL> COMMIT; **Termina trans T.1**		Transaction 1 commits its work, ending the transaction.
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------- ---------- Banda 7000 Greene 9500 Hintz **No ve valores modificados por Session 2 que no ha hecho commit**	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------- ---------- Banda 6200 Greene 9900 **No ve valores modificados por Session 1: todavía Session 2 no ha hecho commit** **- INTERESA QUE FUNCIONE ASÍ ?**	Session 1 queries the salaries for employees Banda, Greene, and Hintz and sees changes committed by transaction 1. Session 1 does not see the uncommitted Greene update made by transaction 2. Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. Oracle Database read consistency ensures that the Hintz insert and Banda update committed by transaction 1 are not visible to transaction 2. Transaction 2 sees its own update to the Banda salary.
	COMMIT; **Termina Trans. T.2**	Transaction 2 commits its work, ending the transaction.
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------- ----------	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------- ----------	Both sessions query the salaries for Banda, Greene, and Hintz. Each session sees all committed

Session 1 (→read commited)	**Session 2**	**Explanation**
Banda 7000 Greene 9900 Hintz	Banda **(de s.1)** 7000 Greene **(de s.2)** 9900 Hintz **(de s.1)**	changes made by transaction 1 and transaction 2.
Trans nueva - T3 - SQL> UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz'; **BX en la fila**		Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is `READ COMMITTED`.
	Trans nueva - T4 - SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Session 2 begins transaction 4 and sets it to the `SERIALIZABLE` isolation level.
	SQL> UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz'; -- prompt does not return **se queda esperando porqué session 1 bloqueó Hintz**	Transaction 4 attempts to update the salary for Hintz, but is blocked because transaction 3 locked the Hintz row (see "Row Locks (TX)"). Transaction 4 queues behind transaction 3.
SQL> COMMIT; **Fin Trans, libera bloqueo de Hintz**		Transaction 3 commits its update of the Hintz salary, ending the transaction.
	UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz' * ERROR at line 1: ORA-08177: can't serialize access for this transaction **No puede hacer el plan secuenciable: no ejecuta esa instrucción, sigue en T.4 Porque está secuenciable: Evita pérdida de datos actualizados por T.3**	The commit that ends transaction 3 causes the Hintz update in transaction 4 to fail with the `ORA-08177` error. The problem error occurs because transaction 3 committed the Hintz update after transaction 4 began.
	SQL> ROLLBACK; **Termina Trans T.4**	Session 2 rolls back transaction 4, which ends the transaction.

Session 1 (→read commited)	Session 2	Explanation

	`SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
	`Empieza Trans T.5`
		Session 2 begins transaction 5 and sets it to the `SERIALIZABLE` isolation level.

	`SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');`
	
	`  LAST_NAME         SALARY`
	`------------- ----------`
	`Banda               7100`
	`Greene              9500`
	`Hintz               7100`
	`Se ve valor de Hintz de T.3 de Session 1`
		Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible.

	`SQL> UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz';`
	` 1 row updated.`
	
	`Como T.3 ya terminó se puede actualizar Hintz`
		Transaction 5 updates the Hintz salary to a different value. Because the Hintz update made by transaction 3 committed before the start of transaction 5, the serialized access problem is avoided.
		
		Note: If a different transaction updated and committed the Hintz row after transaction transaction 5 began, then the serialized access problem would occur again.

	`SQL> COMMIT;`
	`Termina Trans T.5`
		Session 2 commits the update without any problems, ending the transaction.

Example: **Efectos de los Locks Automáticos de Oracle .**

Table 9-4 Row Locking Example

Time	Session 1: usuario "hr1"	Session 2: usuario "hr2"	Explanation

t0	`SELECT employee_id, email, phone_number FROM hr.employees WHERE last_name = 'Himuro';`
	`EMPLOYEE_ID EMAIL PHONE_NUMBER`
	`----------- ------- ------------`
	`118 GHIMURO 515.127.4565`
	
	no se bloquea ninguna fila
			In session 1, the `hr1` user queries `hr.employees` for the Himuro record and displays the employee_id (`118`), email (`GHIMURO`), and phone number (`515.127.4565`) attributes.

t1		`SELECT employee_id, email, phone_number FROM hr.employees WHERE last_name = 'Himuro';`
		`EMPLOYEE_ID EMAIL PHONE_NUMB`
		`----------- ------- -------`
		`118 GHIMURO 515.127.4565`
		
		no se bloquea ninguna fila
			In session 2, the `hr2` user queries `hr.employees` for the Himuro record and displays the employee_id (`118`), email (`GHIMURO`), and phone number (`515.127.4565`) attributes.

t2	`UPDATE hr.employees SET phone_number='515.555.1234' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.127.4565'; 1 row updated.` **Se bloquea fila (BX) GHIMURO**		In session 1, the `hr1` user updates the phone number in the row to `515.555.1234`, which acquires a lock on the `GHIMURO` row.
t3		`UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.127.4565';` **No contesta nada: se queda esperando por esa fila**	In session 2, the `hr2` user attempts to update the same row, but is blocked because `hr1` is currently processing the row. The attempted update by `hr2` occurs almost simultaneously with the `hr1` update.
t4	`COMMIT; Commit complete.` **Desbloquea la fila GHIMURO Ha cambiado el num. Teléfono termina en 1234**	**El num. Teléfono ya no coincide con la condición: ninguna fila cumple la condición, luego no se actualiza ninguna fila**	In session 1, the hr1 user commits the transaction. The commit makes the change for Himuro permanent and unblocks session 2, which has been waiting.
t5		`0 rows updated.`	In session 2, the `hr2`

Time	Session 1: usuario "hr1"	Session 2: usuario "hr2"	Explanation
			user discovers that the `GHIMURO` row was modified in such a way that it no longer matches its predicate. Because the predicates do not match, session 2 updates no records.

t6	`UPDATE hr.employees  SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.555.1234';`
	`1 row updated.`
	
	Vuelve a actualizar el num. Tel. termina en 1235 de nuevo
			In session 1, the `hr1` user realizes that it updated the `GHIMURO` row with the wrong phone number. The user starts a new transaction and updates the phone number in the row to `515.555.1235`, which locks the `GHIMURO` row.

t7		`SELECT employee_id, email, phone_number  FROM hr.employees  WHERE last_name = 'Himuro';`
		
		`EMPLOYEE_ID EMAIL PHONE_NUM`
		`----------- ------- -------`
		`118 GHIMURO 515.555.1234`
		
		no ve el valor 1235 porque session 1 no ha hecho commit
			In session 2, the `hr2` user queries `hr.employees` for the Himuro record. The record shows the phone number update committed by session 1 at t4. Oracle Database read consistency ensures that session 2 does not see the uncommitted change made at t6.

t8		`UPDATE hr.employees  SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.555.1234';` **Se queda esperando porque esa fila la bloqueó session 1**	In session 2, the `hr2` user attempts to update the same row, but is blocked because `hr1` is currently processing the row.
t9	`ROLLBACK;  Rollback complete.` **Deshace update y queda el num. Tel de antes 1234**		In session 1, the `hr1` user rolls back the transaction, which ends it.
t10		`1 row updated.` **Pone el num. Tel. a 1235**	In session 2, the update of the phone number succeeds because the session 1 update was rolled back. The `GHIMURO` row matches its predicate, so the update succeeds.
t11		`COMMIT;  Commit complete.` **Queda permanentemente 1235**	Session 2 commits the update, ending the transaction.

Oracle Database automatically obtains necessary locks when executing SQL statements. For example, before the database permits a session to modify data, the session must first lock the data. The lock gives the session exclusive control over the data so that no other transaction can modify the locked data until the lock is released.

Because the locking mechanisms of Oracle Database are tied closely to transaction control, application designers need only define transactions properly, and Oracle Database automatically manages locking. Users never need to lock any resource explicitly, although Oracle Database also enables users to lock data manually.

1.5.1.1.1 Row Locks and Concurrency

Example: **Oracle usa los Locks Automáticos para la Concurrencia**

Table 9-6 illustrates how Oracle Database uses row locks for concurrency. Three sessions query the same rows simultaneously. Session 1 and 2 proceed to make uncommitted updates to different rows, while session 3 makes no updates. Each session sees its own uncommitted updates but not the uncommitted updates of any other session.

Table 9-6 Data Concurrency Example

Time	Session 1	Session 2	Session 3	Explanation
t0	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY ----------- ------ 100 512 101 600 **Las tres Sesiones devuelven lo mismo**	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLO_ID SALARY ----------- ---- 100 512 101 600	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLO_ID SAL ------ ------ 100 512 101 600	Three different sessions simultaneously query the ID and salary of employees 100 and 101. The results returned by each query are identical.
t1	UPDATE hr.employees SET salary=salary+100 WHERE employee_id=100; **Bloquea fila 100 para impedir actualizaciones de otras Trans.**			Session 1 updates the salary of employee 100, but does not commit. In the update, the writer acquires a row-level lock for the updated row only, thereby preventing other writers from modifying this row.
t2	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY ----------- ------ 100 612 101 600	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY --------- ------ 100 512 101 600	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY ---- ---- 100 512 101 600	Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update. The readers in session 2 and 3 return rows immediately and do not wait for session 1 to end its transaction. The database uses multiversion read consistency to show the salary as it existed before the update in session 1.

Time	Session 1	Session 2	Session 3	Explanation
t3		UPDATE hr.employees SET salary=salary+100 WHERE employee_id=101; **Bloquea fila 101 para impedir actualizaciones de otras Trans.**		Session 2 updates the salary of employee 101, but does not commit the transaction. In the update, the writer acquires a row-level lock for the updated row only, preventing other writers from modifying this row.
t4	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY ----------- ------ 100 612 101 600 **Cada Trans ve una version de la BD**	SELECT employee_id, salaryFROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY --------- ------ 100 512 101 700	SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101); EMPLOYEE_ID SALARY -- ------ 100 512 101 600	Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update, but not the salary update for employee 101 made in session 2. The reader in session 2 shows the salary update made in session 2, but not the salary update made in session 1. The reader in session 3 uses read consistency to show the salaries before modification by session 1 and 2.