1. What Are Locks?

Locks are mechanisms used to regulate concurrent access to a shared resource. For example, while a stored procedure is executing, the procedure itself is locked in a mode that allows others to execute it, but it will not permit another user to alter that instance of that stored procedure in any way.

In a single-user database, locks are not necessary. There is, by definition, only one user modifying the information. However, when multiple users are accessing and modifying data or data structures, it is crucial to have a mechanism in place to prevent concurrent modification of the same piece of information.

Blocking

Blocking occurs when one session holds a lock on a resource that another session is requesting. As a result, the requesting session will be blocked—it will hang until the holding session gives up the locked resource.

The five common DML statements that will block in the database are INSERT, UPDATE, DELETE, MERGE, and SELECT FOR UPDATE.

a. Blocked Inserts

There are few times when an INSERT will block. The most common scenario is when you have a table with a primary key or unique constraint placed on it and two sessions attempt to insert a row with the same value.

b. Blocked Merges, Updates, and Deletes

In an interactive application—one where you query some data out of the database, allow an end user to manipulate it, and then put it back into the database—a blocked UPDATE or DELETE indicates that you probably have a lost update problem in your code. You are attempting to UPDATE a row that someone else is already updating (in other words, one that someone else already has locked). You can avoid the blocking issue by using the SELECT FOR UPDATE NOWAIT query to

- Verify the data has not changed since you queried it out (preventing lost updates).
- Lock the row (preventing the UPDATE or DELETE from blocking)

2. Lock Types

DML locks: DML stands for *Data Manipulation Language*. In general this means SELECT, INSERT, UPDATE, MERGE, and DELETE statements.

DDL locks: DDL stands for *Data Definition Language*, (CREATE and ALTER statements, and so on).

a. DML Locks

DML locks are used to ensure that only one person at a time modifies a row and that no one can drop a table upon which you are working. Oracle will place these locks for you, more or less transparently, as you do work.

- **Row Locks (TX)**: A **row lock**, also called a **TX lock**, is a lock on a single row of a table. A transaction acquires a row lock for each row modified by one of the following

statements: INSERT, UPDATE, DELETE, MERGE, and SELECT ... FOR UPDATE. The row lock exists until the transaction commits or rolls back.

	Session 1	Session 2	
T1	Update T set x=x+1 where y=2; --1 row updated		Row y=2 is locked
T2		Update T set x=0 where y=2;	Session 2 is blocked until session 1 commit or rollback.

- **TM (DML Enqueue) Locks**: TM locks are used to ensure that the structure of a table is not altered while you are modifying its contents. For example, if you have updated a table, you will acquire a TM lock on that table. This will prevent another user from executing DROP or ALTER commands on that table.

	Session 1	Session 2	
T1	delete from T where y=1 --1 row deleted		
T2		drop table T	Cannot drop table T

b. DDL Locks

DDL locks are automatically placed against objects during a DDL operation to protect them from changes by other sessions.

For example, if I perform the DDL operation ALTER TABLE T, the table T will *in general* have an exclusive DDL lock placed against it, preventing other sessions from getting DDL locks and TM locks on this table

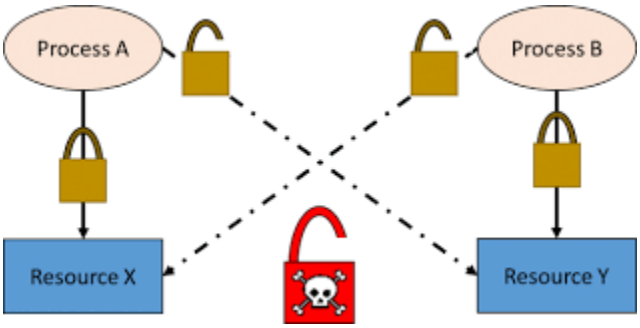
- Exclusive ***DDL locks***: These prevent other sessions from gaining a DDL lock or TM (DML) lock themselves. For example, a DROP TABLE operation is not allowed to drop a table while an ALTER TABLE operation is adding a column to it, and vice versa.
- **Share DDL locks:** These protect the structure of the referenced object against modification by other sessions, but allow modifications to the data.

For example, when a CREATE PROCEDURE statement is run, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and therefore acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table. No transaction can alter or drop a referenced table.

A share DDL lock lasts for the duration of DDL statement execution and automatic commit. Thus, a transaction holding a share DDL lock is guaranteed that the definition of the referenced schema object is constant for the duration of the transaction.

Deadlocks

In a database, a deadlock is a situation in which two or more transactions are waiting for one another to give up locks.



Time	Session 1	Session 2	Explanation

t0	`SQL> UPDATE employees`
	`  SET salary = salary*1.1`
	`  WHERE employee_id = 100;`
	
	`1 row updated.`
		`SQL> UPDATE employees`
		`  SET  salary = salary*1.1`
		`  WHERE employee_id = 200;`
		
		`1 row updated.`
			Session 1 starts transaction 1 and updates the salary for employee 100. Session 2 starts transaction 2 and updates the salary for employee 200. No problem exists because each transaction locks only the row that it attempts to update.

t1	`SQL> UPDATE employees`
	`  SET salary = salary*1.1`
	`  WHERE employee id = 200;`
	
	`-- prompt does not return`
		`SQL> UPDATE employees`
		`   salary = salary*1.1`
		`  WHERE employee id = 100;`
		
		`-- prompt does not return`
			Transaction 1 attempts to update the employee 200 row, which is currently locked by transaction 2. Transaction 2 attempts to update the employee 100 row, which is currently locked by transaction 1.
			
			A deadlock results because neither transaction can obtain the resource it needs to proceed or terminate. No matter how long each transaction waits, the conflicting locks are held.

t2	`UPDATE employees`
	`  *`
	`ERROR at line 1:`
	`ORA-00060: deadlock detected`
	`while waiting for resource`
	
	`SQL>`
			Transaction 1 signals the deadlock and rolls back the `UPDATE` statement issued at t1. However, the update made at t0 is not rolled back. The prompt is returned in session 1.
			
			Note: Only one session in the deadlock actually gets the deadlock error, but either session could get the error.

t3	`SQL> COMMIT;`
	
	`Commit complete.`
			Session 1 commits the update made at t0, ending transaction 1. The update unsuccessfully attempted at t1 is not committed.

t4		`1 row updated.`
		
		`SQL>`
			The update at t1 in transaction 2, which was being blocked by transaction 1, is executed. The prompt is returned.

t5		`SQL> COMMIT;`
		
		`Commit complete.`
			Session 2 commits the updates made at t0 and t1, which ends transaction 2.

3. Concurrency and Multiversioning

a. What Are Concurrency Controls?

Concurrency controls are the collection of functions that the database provides to allow many people to access and modify data simultaneously. The *lock* is one of the core mechanisms by which Oracle regulates concurrent access to shared database resources and prevents interference between concurrent database transactions.

Transaction Isolation Levels: These isolation levels are defined in terms of three "phenomena" that are either permitted or not at a given isolation level:

- *Dirty reads*:

  A transaction reads data that has been written by another transaction that has not been committed yet.

- *Nonrepeatable (fuzzy) reads*

  A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data. For example, a user queries a row and then later queries the same row, only to discover that the data has changed.

- *Phantom reads*

  A transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

  For example, a transaction queries the number of employees. Five minutes later it performs the same query, but now the number has increased by one because another user inserted a record for a new hire. More data satisfies the query criteria than before, but unlike in a fuzzy read the previously read data is unchanged.

  The SQL standard defines four levels of isolation in terms of the phenomena that a transaction running at a particular isolation level is permitted to experience.

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Oracle Database offers the **read committed** (default) and **serializable** isolation levels. Also, the database offers a read-only mode.

❖ Read Committed Isolation Level

In a read committed transaction, a conflicting write occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction, sometimes called a blocking transaction. The read committed transaction waits for the blocking transaction to end and release its row lock. The options are as follows:

- If the blocking transaction rolls back, then the waiting transaction proceeds to change the previously locked row as if the other transaction never existed.
- If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed row.

❖ **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.

Serializable isolation is suitable for environments:

- With large databases and short transactions that update only a few rows
- Where the chance that two concurrent transactions will modify the same rows is relatively low
- Where relatively long-running transactions are primarily read only

Read Consistency and Serialized Access Problems in Serializable Transactions

Example 1:

Session 1	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.
SQL> UPDATE employees SET salary = 7000 WHERE last_name = '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;	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	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';	Transaction 2 updates the Greene salary successfully because only the Banda row is locked.
SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210,'Hintz', 'JHINTZ', SYSDATE,'SH_CLERK');		Transaction 1 inserts a row for employee Hintz.
SQL> COMMIT;		Transaction 1 commits its work, ending the transaction.

Session 1	Session 2	Explanation
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------ ---------- Banda 7000 Greene 9500 Hintz	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ------------ ---------- Banda 6200 Greene 9900	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;	Transaction 2 commits its work, ending the transaction.

Example 2:

Session 1	Session 2	Explanation
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST NAME SALARY ------------ ---------- Banda 7000 Greene 9900 Hintz	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST NAME SALARY ------------ ---------- Banda 7000 Greene 9900 Hintz	Both sessions query the salaries for Banda, Greene, and Hintz. Each session sees all committed changes made by transaction 1 and transaction 2.
SQL> UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz';		Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is READ COMMITTED.
	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	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;		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	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;	Session 2 rolls back transaction 4, which ends the transaction.
	SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	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	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.	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;	Session 2 commits the update without any problems, ending the transaction.

4. Các vấn đề xảy ra trong tương tác đồng thời, mức cô lập

Hầu hết các hệ thống Cơ sở dữ liệu sử dụng Read Committed làm mức cô lập mặc định (ngoại trừ MySQL sử dụng mức cô lập mặc định là Repeatable Read). Lựa chọn mức cô lập là tìm sự cân bằng phù hợp về tính nhất quán và khả năng mở rộng cho các yêu cầu ứng dụng hiện tại của người dùng.

Bởi vì hầu hết ứng dụng sử dụng mức cô lập mặc định của database, nên sẽ rất quan trọng nếu hiểu các đặc tính của Read committed:

- Các câu truy vấn chỉ nhìn thấy được dữ liệu đã commit trước khi câu lệnh bắt đầu và những dữ liệu được thay đổi trong nội tại transaction (current transaction).
- Trong khi một câu lệnh đang thực thi mà có sự thay đổi dữ liệu ở một nơi khác và dữ liệu này được commit thì câu lệnh đang thực thi vẫn không nhìn thấy được dữ liệu đã thay đổi này.
- Nếu hai transaction cố gắng cập nhật dữ liệu trên cùng một dòng, thì transaction đến sau phải đợi cho đến khi transaction đầu tiên kết thúc giao dịch của nó. Sau đó giao dịch đến sau phải đánh giá lại xem dữ liệu dữ liệu đã bị thay đổi đó còn liên quan đến câu lệnh của mình không rồi mới tiến hành update.

A. Lost update:

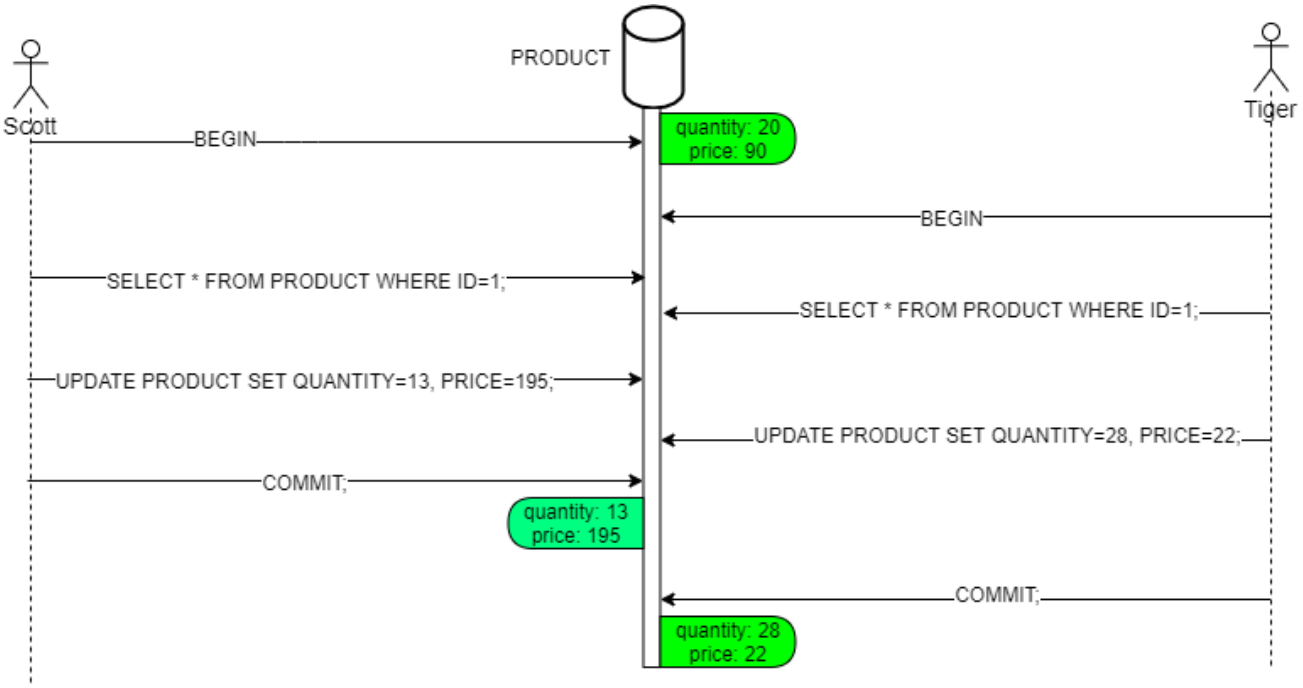
Nếu hai transaction cập nhật hai cột khác nhau của cùng một dòng dữ liệu thì không xảy ra tình trạng xung đột. Giao dịch sau sẽ bị block cho đến khi giao dịch đầu hoàn thành

xong giao tác và nếu cả hai giao tác đều commit thì kết quả cuối cùng phản ánh được các thay đổi trong cả hai giao dịch.

Nếu hai giao dịch muốn thay đổi cùng một dữ liệu (trên cùng dòng và cột) thì dữ liệu của giao dịch thực hiện sau sẽ ghi đè lên dữ liệu của giao dịch trước, do vậy sẽ gây nên tình trạng mất dữ liệu của lần update trước đó.

Do vậy, một sự kiện được gọi là lost update khi một giao dịch ghi đè dữ liệu lên dữ liệu trước đó mà không nhận ra là đã có một giao dịch khác thay đổi dữ liệu đó.

Ví dụ 1: Hãy xem SCOTT và TIGER cùng cập nhật một sản phẩm



Hai transaction được thể hiện như sau:

```
CREATE TABLE PRODUCT (ID NUMBER PRIMARY KEY, QUANTITY INT, PRICE NUMBER);
INSERT INTO PRODUCT VALUES (1, 20, 90);
COMMIT;
```

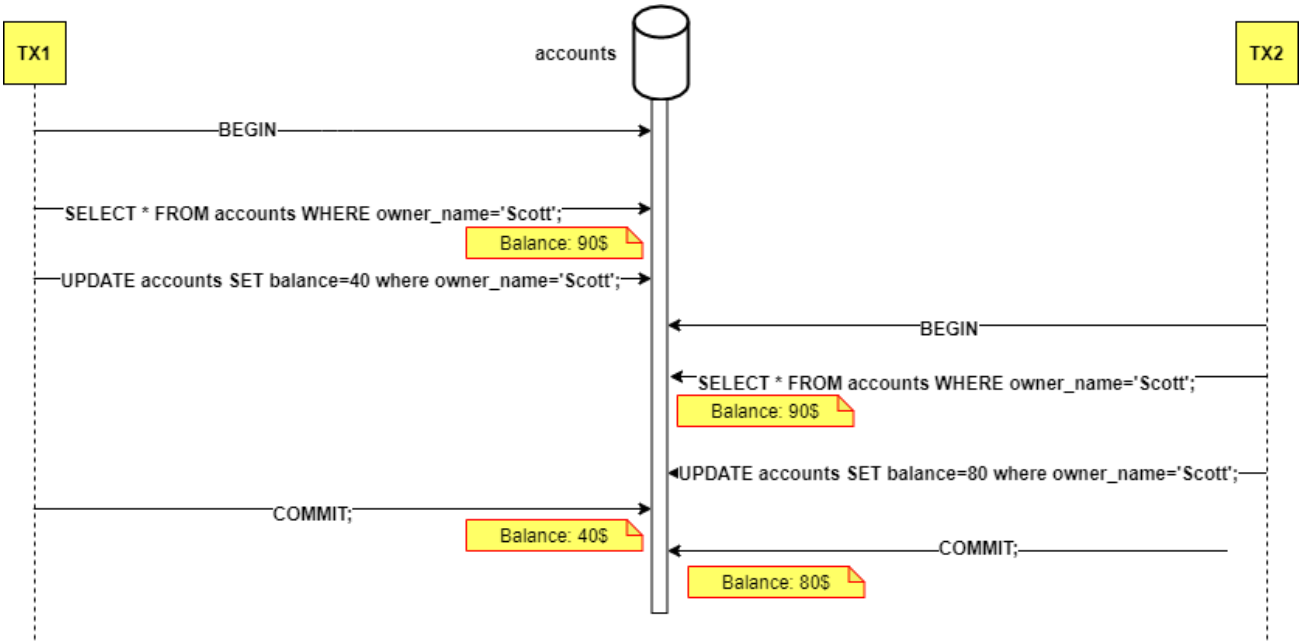
SCOTT	TIGER
SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 20 90	SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 20 90
UPDATE PRODUCT SET QUANTITY=13, PRICE=195 WHERE ID=1; --1 row updated	
	UPDATE PRODUCT SET QUANTITY=28, PRICE=22 WHERE ID=1;
COMMIT;	
	--1 row updated
SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 13 195	
	COMMIT; SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 28 22

```
SELECT * FROM PRODUCT WHERE ID=1;

        ID   QUANTITY      PRICE
---------- ---------- ----------
         1         28         22
```

Dữ liệu mà SCOTT cập nhật đã bị mất và TIGER không hề biết là mình đã ghi đè lên dữ liệu này của SCOTT.

Ví dụ 2: Hãy xem hai session cùng cập nhật một tài khoản



Trong ví dụ này, hai transaction đang rút 50$ và 10$ từ tài khoản của Scott. Sau khi cả hai kết thúc thì chúng ta mong chờ sẽ nhìn thấy 30$ trong tài khoản, nhưng rốt cuộc số tiền trong tài khoản là 80$. Bởi vì giao dịch thứ hai chỉ nhìn thấy dữ liệu đã commit, nó không nhận ra được tương tác đồng thời và vận hành như thể giao dịch trước đó chưa bao giờ xảy ra. Kết quả là giao dịch thứ hai ghi đè lên lần cập nhật trước đó của transaction 1 và hệ thống mất 50$.

Ngăn chặn Lost Update
Có rất nhiều cách để ngăn chặn lost update đó là:

- Increase transaction isolation level (Tăng mức cô lập của giao dịch)
- Optimistic Locking
- Pessimistic Locking

a. Increase transaction isolation level

Một ưu điểm của cách tiếp cận này là cơ sở dữ liệu có thể thực hiện việc kiểm tra một cách hiệu quả cùng với mức cô lập repeatable read. PostgreSQL’s repeatable read, Oracle’s serializable, và SQL Server’s snapshot isolation level sẽ tự động phát hiện khi nào xảy ra

bản cập nhật bị mất và hủy bỏ giao dịch vi phạm. Tuy nhiên, mức cô lập repeatable read của MySQL / InnoDB không phát hiện ra lost update.

Ví dụ: Sử dụng mức cô lập Serializable có thể ngăn chặn tình trạng lost update (Oracle không có mức cô lập Repeatable read).

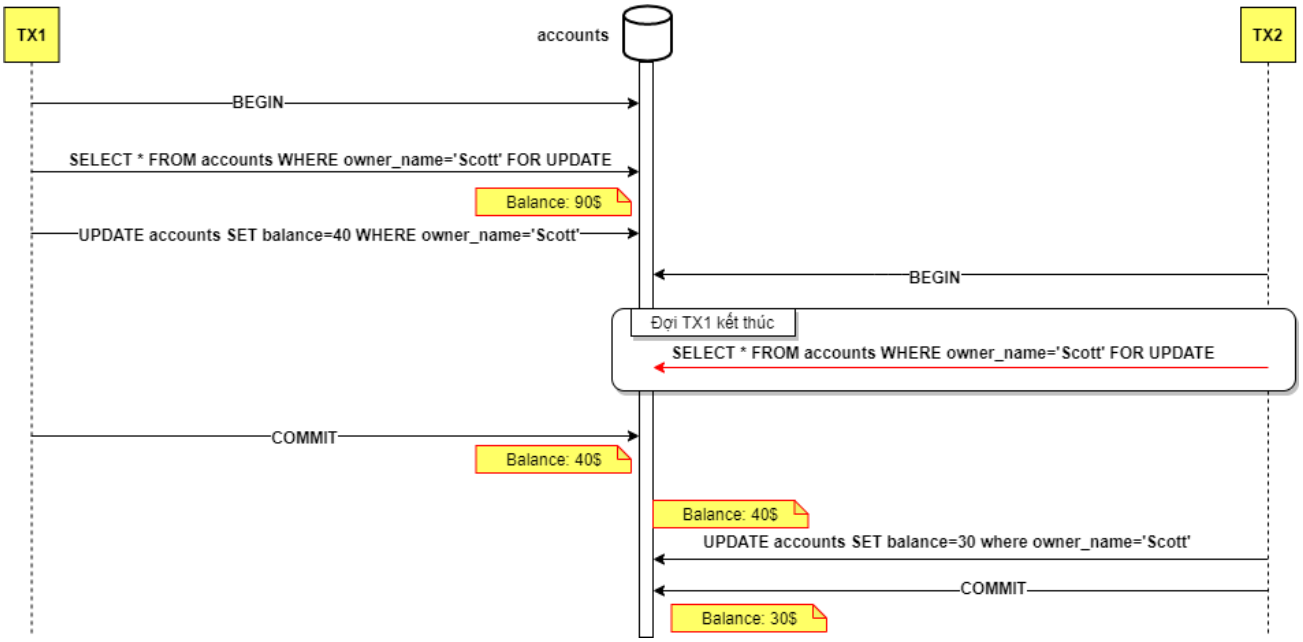
SCOTT	TIGER
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 28 22	SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 28 22
UPDATE PRODUCT SET QUANTITY=15, PRICE=17 WHERE ID=1; --1 row updated	
	UPDATE PRODUCT SET QUANTITY=21, PRICE=26 WHERE ID=1; --1 row updated
COMMIT;	
SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 15 17	Error starting at line : 3 in command - UPDATE PRODUCT SET QUANTITY=21, PRICE=26 WHERE ID=1 Error report - ORA-08177: can't serialize access for this transaction

Lần này, Tiger không thể ghi đè dữ liệu của mình lên của Scott, và câu lệnh update của Tiger bị hủy bỏ. Trong Serializable, một câu lệnh truy vấn chỉ có thể nhìn thấy dữ liệu tại thời điểm bắt đầu transaction hiện tại. Những thay đổi đã được commit bởi những transaction khác thì không được nhìn thấy từ giao dịch hiện tại.

Nếu hai giao dịch cố gắng thay đổi cùng một dòng dữ liệu, thì giao dịch thứ hai sẽ đợi đến khi giao dịch thứ nhất kết thúc (commit hoặc rollback). Nếu giao dịch đầu commit thì giao dịch thứ hai sẽ bị hủy bỏ và không xảy ra lost update.

b. Pessimistic Locking

Một tùy chọn khác để ngăn lost update là khóa tường minh các đối tượng sẽ được cập nhật. Sau đó, ứng dụng có thể thực hiện chu trình đọc-sửa-ghi và nếu bất kỳ giao dịch nào khác cố gắng đọc đồng thời cùng một đối tượng, nó buộc phải đợi cho đến khi hoàn thành chu kỳ đọc-sửa-ghi đầu tiên. Cách tiếp cận này được gọi là khóa bi quan.



Vì giao dịch thứ hai chỉ đọc dữ liệu sau khi hoàn thành giao dịch đầu tiên, không có ghi đè dữ liệu, do đó không thể xảy ra lost update.

Ví dụ: sử dụng FOR UPDATE với mức cô lập mặc định Read Committed.

SCOTT	TIGER
SELECT * FROM PRODUCT WHERE ID=1 FOR UPDATE; ID QUANTITY PRICE ---------- ---------- ---------- 1 15 17	
	SELECT * FROM PRODUCT WHERE ID=1 FOR UPDATE;
UPDATE PRODUCT SET QUANTITY=19, PRICE=18 WHERE ID=1; COMMIT; SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 19 18	
	ID QUANTITY PRICE ---------- ---------- ---------- 1 19 18
	UPDATE PRODUCT SET QUANTITY=23, PRICE=25 WHERE ID=1; COMMIT; SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE ---------- ---------- ---------- 1 23 25

Tiger không thể tiến hành với câu lệnh SELECT vì Scott đang nắm giữ lock trên dòng này. Tiger sẽ phải đợi Scott kết thúc transaction và khi đó Tiger sẽ không còn bị 'treo' (block) nữa, lúc này Tiger sẽ tự động nhìn thấy dữ liệu mà Scott đã thay đổi, do đó những thứ mà Scott update sẽ không bị mất.

Cả hai transaction nên sử dụng FOR UPDATE locking. Nếu transaction đầu tiên mà không yêu cầu write lock thì lost update có thể vẫn xảy ra.

Ghi chú:

Trong kiến trúc "client/server" cũ, mỗi người dùng giữ cùng một session để truy vấn dữ liệu và cập nhật dữ liệu. Để tránh xung đột ở mức dòng dữ liệu (row level), chỉ cần thực hiện SELECT FOR UPDATE thay vì câu lệnh SELECT đơn giản. Bằng cách đó, người

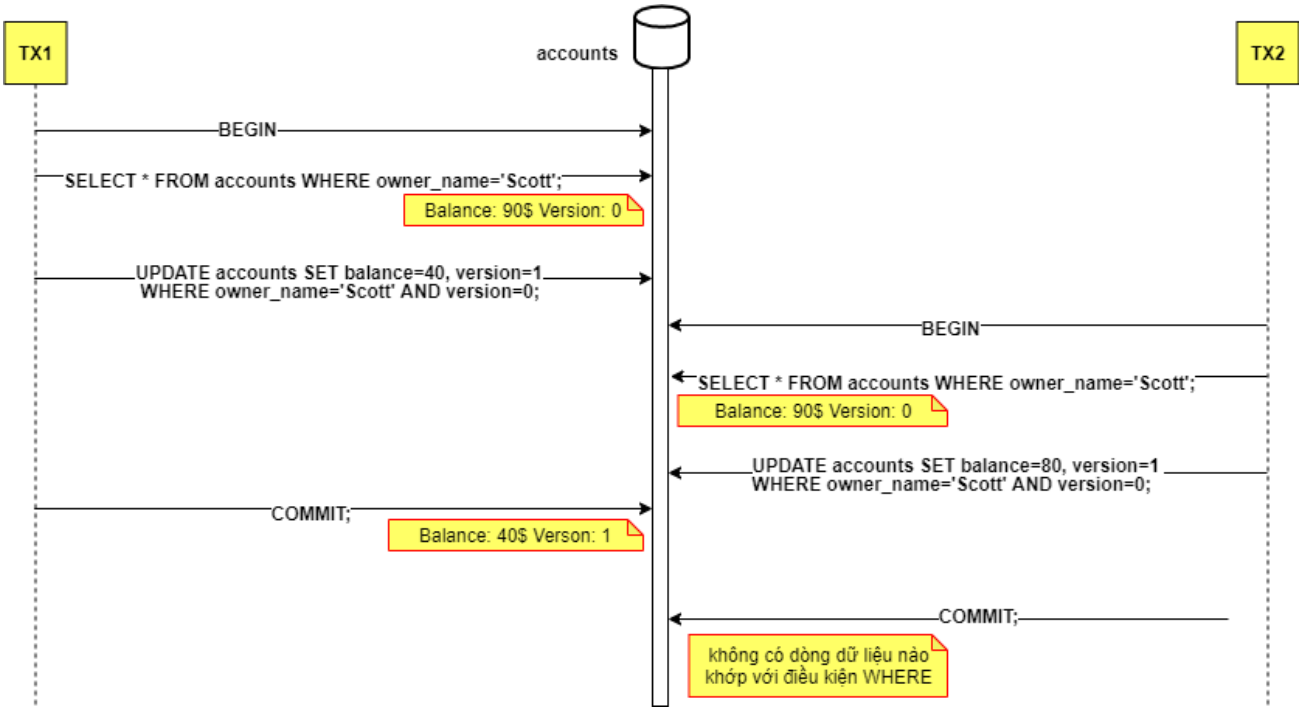
dùng hiện tại đã thấy tất cả các thay đổi trước đó và những người dùng sau sẽ thấy tất cả các thay đổi hiện tại. "Lost update" hoàn toàn tránh được. Điều này được gọi là "pessimistic locking".

Trong các kiến trúc "stateless architectures"ngày nay, người dùng có thể nhận được một session để truy vấn dữ liệu và một session khác để cập nhật nó. Không session nào sẽ nhớ bất cứ điều gì về những gì đã xảy ra khi người dùng truy cập vào cơ sở dữ liệu trước đó. Do đó, SELECT FOR UPDATE đã lỗi thời. Điều này dẫn đến các chiến lược "optimistic" khác nhau, một số phương pháp là:

- column compare
- hash compare
- timestamp compare
- version compare

c. Optimistic Locking

Optimistic Locking còn được gọi là 'cập nhật có điều kiện' hoặc 'so sánh và thiết lập' giúp tránh lost update bằng cách chỉ cho phép một cập nhật xảy ra nếu giá trị không thay đổi kể từ lần cuối đọc nó. Nếu giá trị hiện tại không khớp với những gì đã đọc trước đó, thì bản cập nhật sẽ không có hiệu lực và chu trình đọc-sửa-ghi phải được thử lại. Có nhiều cách khác nhau để triển khai Optimistic locking, tuy nhiên, có lẽ chúng ta sẽ sử dụng cách phổ biến nhất.



Vì phiên bản theo từng dòng đã được cập nhật bởi giao dịch đầu tiên, nên bản cập nhật thứ hai không có hiệu lực.

Ví dụ:

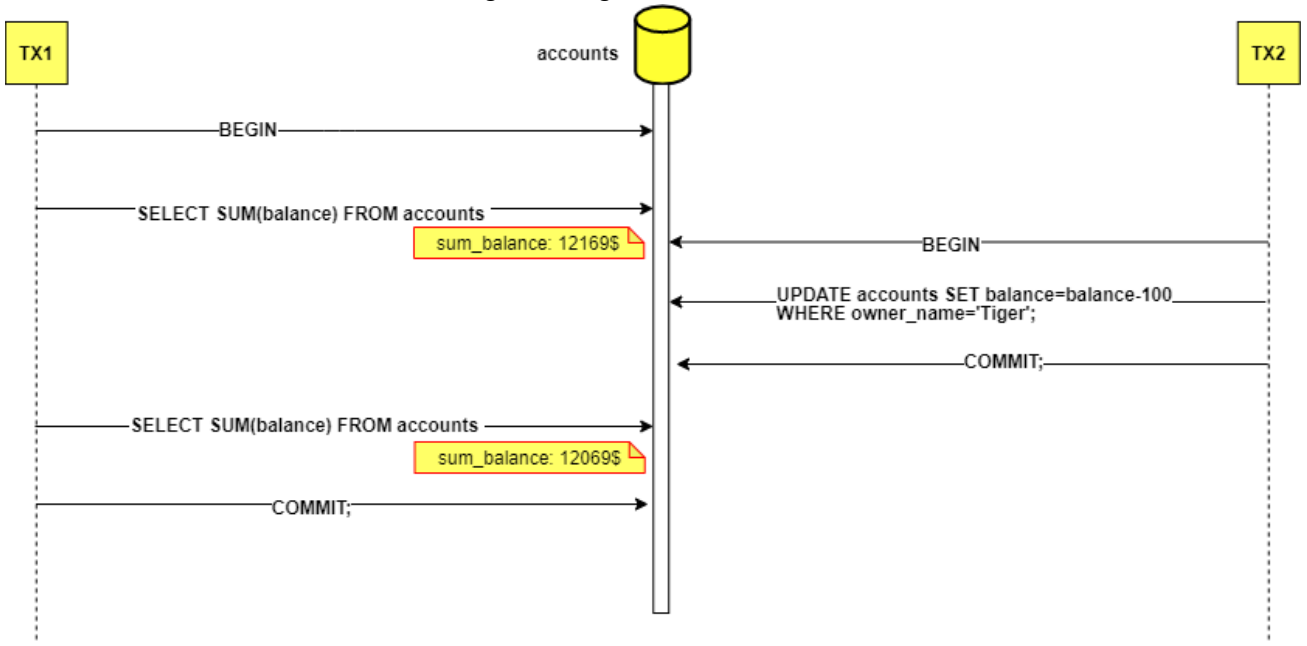
```
Create table product (id number primary key, quantity int, price number, version int);
insert into product values (1, 20, 90, 0);
commit;
```

SCOTT -- Mức cô lập mặc định Read Committed.	TIGER -- Mức cô lập mặc định Read Committed.
SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE VERSION ---------- ---------- ---------- ---------- 1 20 90 0	SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE VERSION ---------- ---------- ---------- ---------- 1 20 90 0
UPDATE PRODUCT SET quantity=13, version=1 WHERE ID=1 AND version=0; -- 1 row updated.	
	UPDATE PRODUCT SET QUANTITY=28, version=1 WHERE ID=1 AND version=0;
COMMIT;	
SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE VERSION ---------- ---------- ---------- ---------- 1 13 90 1	-- 0 rows updated.
	SELECT * FROM PRODUCT WHERE ID=1; ID QUANTITY PRICE VERSION ---------- ---------- ---------- ---------- 1 13 90 1

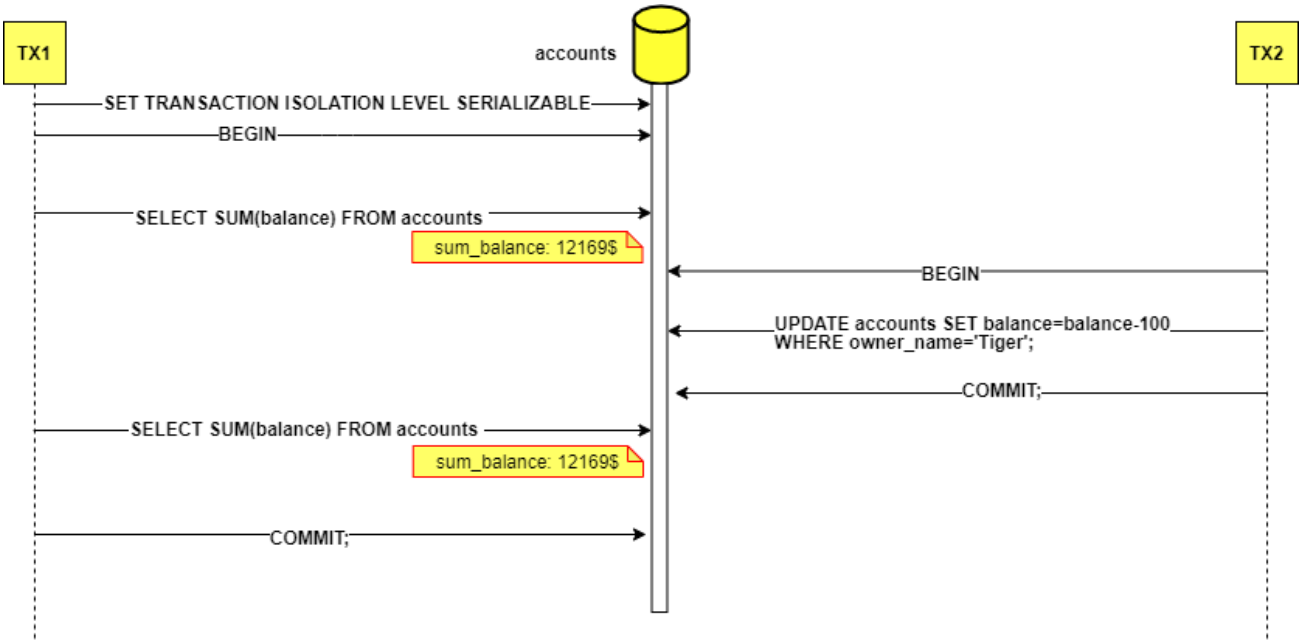
Lúc đầu cả scott và tiger đều đọc cùng một dòng dữ liệu và version của dòng dữ liệu này là 0. Sau đó, scott quyết định cập nhật quantiy dòng dữ liệu này và thay đổi version thành 1 nhưng chưa commit. Tiger không biết và vẫn tiến hành cập nhật dòng dữ liệu này với điều kiện là version bằng 0. Tiger phải đợi cho đến khi scott commit dữ liệu và kết quả trả về của tiger là '0 rows updated'. Điều này có nghĩa là dòng dữ liệu này đã bị thay đổi nên Tiger không được phép cập nhật và tránh tình trạng lost update.

B. Non-Repeatable read

Trong lúc hệ thống đang xuất báo cáo thông tin tài khoản của khách hàng thì có một giao dịch rút tiền diễn ra làm sai lệch thông tin trong báo cáo.



Để giải quyết tình trạng không đồng nhất khi đọc dữ liệu trong cùng một transaction thì nên chuyển mức cô lập thành Serializable.

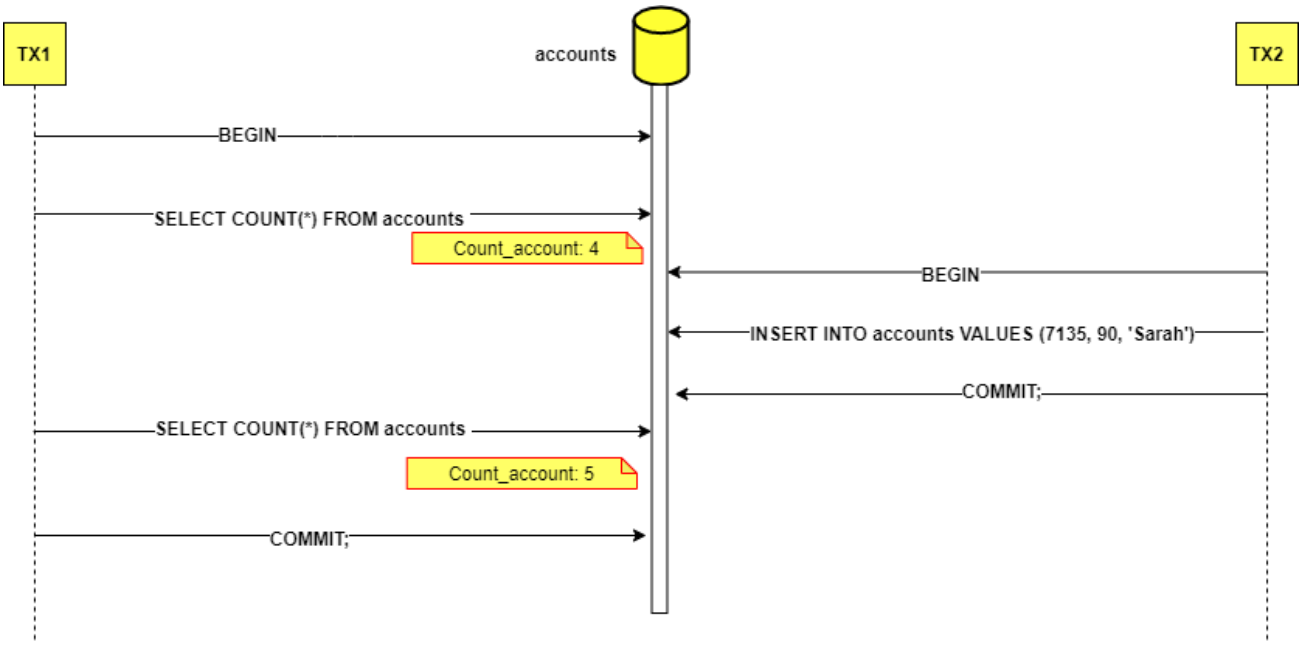


Session1	Session2

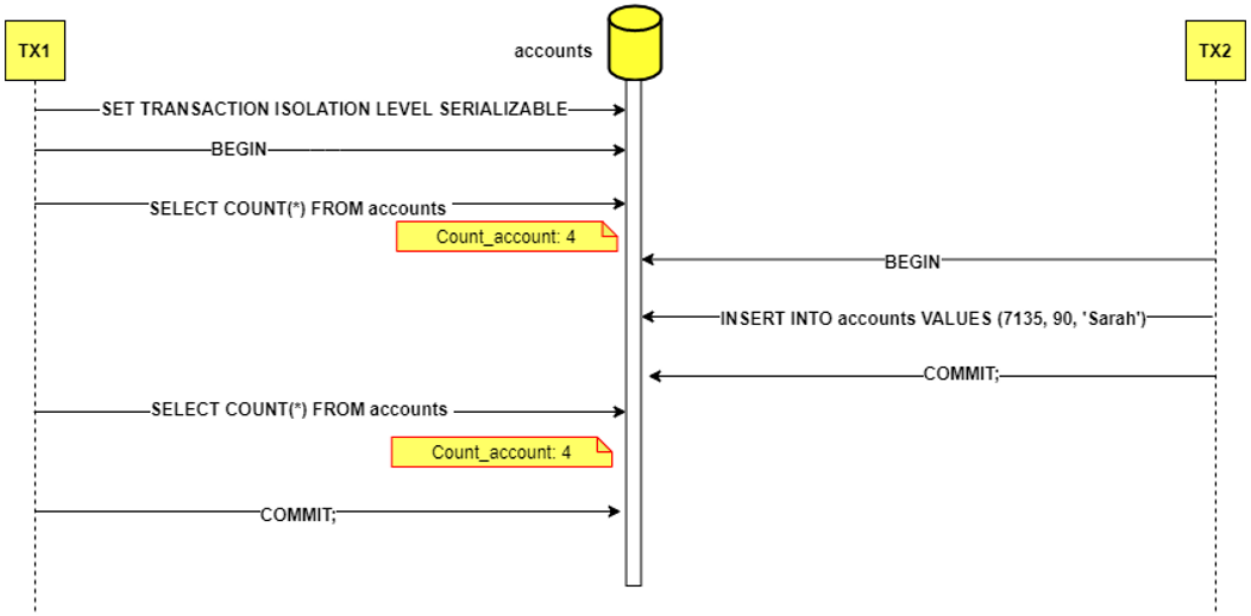
set serveroutput on
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
DECLARE
sum_balance NUMBER :=0;
BEGIN
FOR i IN (select * from accounts) LOOP
sum_balance:=sum_balance+i.balance;
DBMS_OUTPUT.PUT_LINE (i.accid || ' ' || i.balance || ' ' || i.owner_name);
END LOOP;
DBMS_OUTPUT.PUT_LINE('Tổng tiền sau vòng lặp for: ' || sum_balance);
 	
DBMS_LOCK.SLEEP(10);
 	
SELECT SUM(balance) INTO sum_balance FROM accounts;
DBMS_OUTPUT.PUT_LINE ('Tổng tiền đang có trong hệ thống:' ||sum_balance);
END;
COMMIT;
	SET SERVEROUTPUT ON
	BEGIN
	UPDATE accounts SET balance=balance-100 WHERE owner_name='Tiger';
	END;
	commit;

C. Phantom Read

Hệ thống đang xuất báo cáo thông tin về tài khoản của khách hàng, ngay lúc đó có một giao dịch thêm một tài khoản mới và làm sai lệch báo cáo.



Để tránh tình trạng thiếu nhất quán khi đọc dữ liệu ở trong cùng một transaction, thì nên chuyển mức cô lập của transaction thành Serializable.

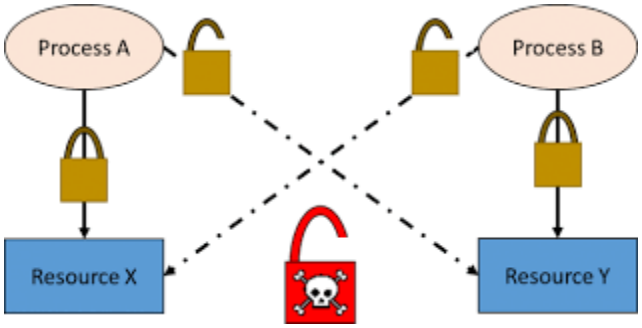


Session1	Session2

set serveroutput on
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
DECLARE
count_member NUMBER :=0;
BEGIN
FOR i IN (select * from accounts) LOOP
count_member:=count_member+1;
DBMS_OUTPUT.PUT_LINE (i.accid || ' ' || i.balance || ' ' || i.owner_name);
END LOOP;
DBMS_OUTPUT.PUT_LINE('số lượng tài khoản trong hệ thống: ' || count_member);
 	
DBMS_LOCK.SLEEP(10);
	
	SET SERVEROUTPUT ON
	BEGIN

SELECT count(*) INTO count_member FROM accounts; DBMS_OUTPUT.PUT_LINE('số lượng tài khoản trong hệ thống: ' || count_member); END; COMMIT;	INSERT INTO accounts VALUES (7135, 90, 'Sarah'); END; commit;

D. DEADLOCK: xảy ra khi hai transaction cùng đợi chờ lẫn nhau để hoàn thành giao tác của mình.




```
DROP TABLE accounts;

CREATE TABLE accounts (accid NUMBER(6) primary key,

                       balance NUMBER (10,2),

                       owner_name varchar2(30),

                       check (balance>=0));

INSERT INTO accounts VALUES (7715, 7000, 'Scott');

INSERT INTO accounts VALUES (7720, 5100, 'Tiger');

INSERT INTO accounts VALUES (7725, 20, 'Helen');

INSERT INTO accounts VALUES (7730, 49, 'John');

COMMIT;
```

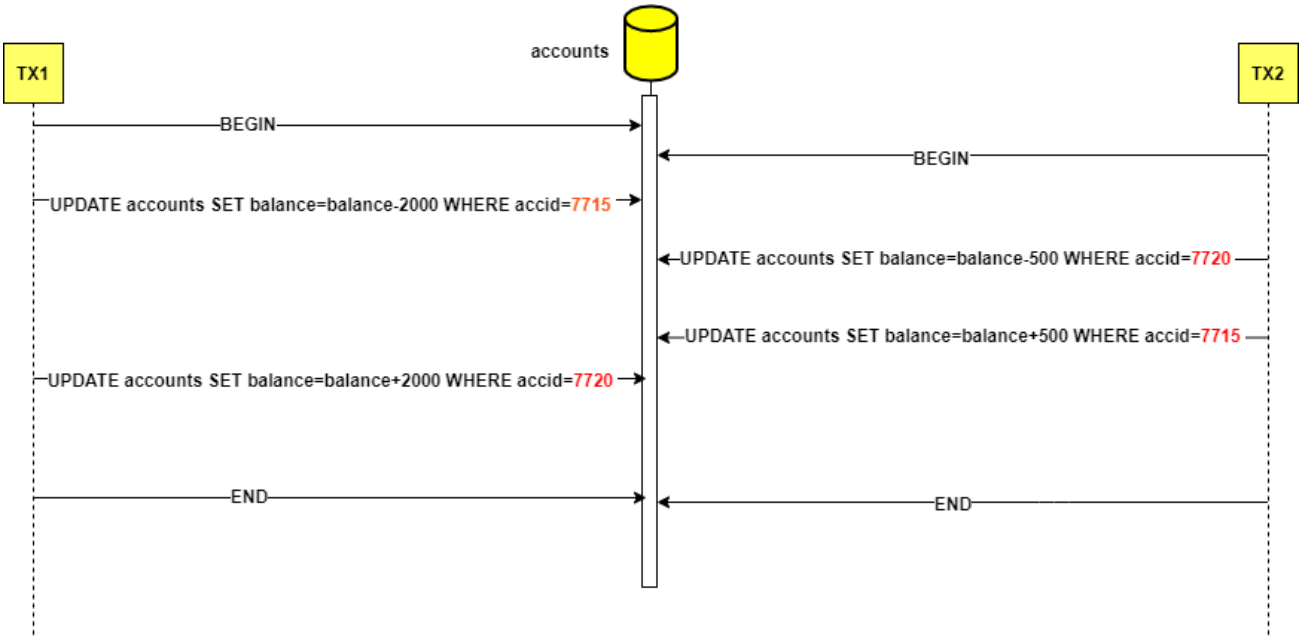
Giả sử có hai giao dịch cùng chuyển tiền

- Chuyển 2000 từ account 7715 sang 7720.
- Chuyển 500 từ account 7720 sang 7715.

Giả sử hai giao dịch này diễn ra đồng thời với kịch bản như sau:

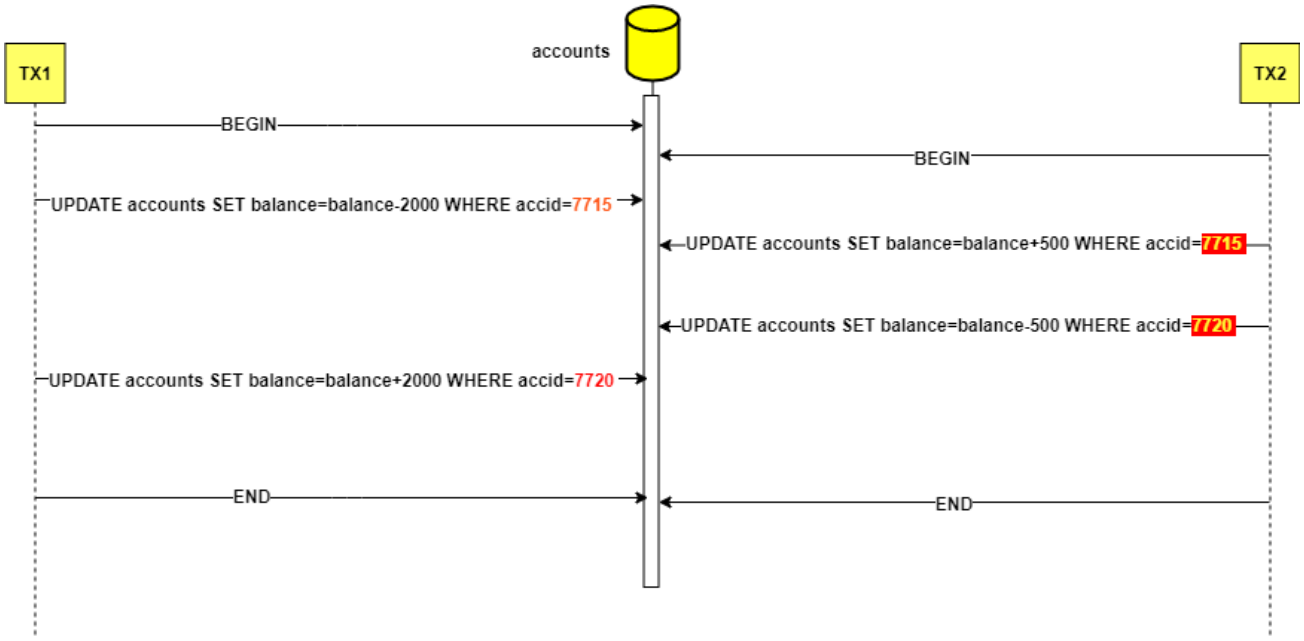
Time	SCOTT	TIGER

	acc 7715: -2000$	
		Acc 7720: -500$
		Acc 7715: +500$
	Acc 7720: +2000$	



Nếu việc chuyển tiền được thực hiện theo thứ tự trên, thì hệ thống sẽ xảy ra tình trạng deadlock. Và khi xảy ra deadlock thì database sẽ phát hiện, một session sẽ bị kết thúc ‘kill’ và session còn lại thực hiện tiếp công việc chuyển tiền của mình. Session mà bị database ‘kill’ này có thể đợi và thực hiện lại việc chuyển tiền sau. Tuy nhiên, để tránh hiện tượng deadlock thì ta có thể thay đổi thứ tự thực hiện việc chuyển tiền như sau

Time	SCOTT	TIGER
	acc 7715: -2000$	
		Acc 7715: +500$
		Acc 7720: -500$
	Acc 7720: +2000$	



Do đó để tránh deadlock thì:

- Khi sửa đổi nhiều bảng trong một transaction, hoặc nhiều dòng của cùng một bảng, hãy thực hiện các thao tác đó theo thứ tự nhất quán. Do đó, transaction tạo thành hàng đợi được tổ chức tốt và không bị deadlock. Như ví dụ trên, việc chuyển tiền có thể đi theo thứ tự từ account có mã số nhỏ đến account có mã số lớn hơn.
- Sử dụng lock dữ liệu (SELECT … FOR UPDATE). Như trong ví dụ trên, trước khi chuyển tiền sẽ lock hai dòng account 7715 và 7720, rồi sau đó mới thực hiện thao tác chuyển.

Nếu các cách trên không giải quyết được thì có thể LOCK TABLE.