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

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. Table 9-1 shows the levels.

Table 9-1 Preventable Read Phenomena by Isolation Level

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.

`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.