LENGUAJES Y
CIENCIAS DE LA
COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

Administración de Bases de Datos

(Ingeniería Informática)

Tema 7. INTEGRIDAD Y CONTROL DE LOS ACCESOS CONCURRENTES A LOS DATOS

E.T.S.I. Informática *Enrique Soler y Jose A. Onieva*

Contenido

- Integridad de los Datos
  - Comprobación de Restricciones
- Control de la Concurrencia/Consistencia
  - Bloqueos
- Gestión de Transacciones
  - Sentencias de Control de transacciones

Integridad de Datos

- Las restricciones de integridad hacen cumplir las reglas del negocio asociadas a la base de datos y previenen la entrada de información incorrecta
- Tipos de Integridad de Datos:
  - NOT NULL. Una columna admite o no valores nulos.
  - UNIQUE. Una columna (o conjunto de columnas) admite valores sólo si son únicos para esa columna (o conjunto de columnas)
  - Primary Key. Definida sobre una columna (o conjunto de columnas) especifica que cada fila en la tabla puede ser identificada unívocamente por los valores de la clave
  - Integridad Referencial. Definida sobre una columna (o conjunto de columnas) en una tabla garantiza que los valores en un atributo son iguales a los valores de una clave en la tabla referenciada. ON DELETE:
    - Set Null: Cuando se borran los datos referenciados, los datos asociados dependientes se ponen a NULL
    - Cascade: Cuando la fila referenciada se borra, los datos asociados, también se borran.
    - **No Action**: Deshabilita el borrado o modificación de los datos referenciados. No Action es el tipo por defecto
  - CHECK. Para restricciones más complejas.

Mecanismo de Comprobación de Restricciones

- Es importante conocer cuándo se hacen las comprobaciones
  - Tabla de empleados con una columna Jefe que referencia al código del empleado (relación reflexiva).

```
CREATE TABLE EMPLEADOS (CODIGO NUMBER PRIMARY KEY,
NOMBRE VARCHAR2(50), JEFE NUMBER REFERENCES EMPLEADOS(CODIGO))
```

  - Cuando se inserta la primera fila, ¿qué valor puede tomar Jefe?
    - NULL (si la columna no tiene restricción NOT NULL)
    - El mismo que el código de empleado:

```
INSERT INTO EMPLEADOS VALUES (10, 'JOSE', 10);
```

Se hace la comprobación DESPUES de insertar la fila

    - INSERT múltiple.

```
UPDATE EMPLEADOS SET CODIGO= CODIGO + 5000,JEFE = JEFE + 5000
```

Se hace la comprobación DESPUES de ejecutar la sentencia

    - INSERT Cruzado. Se inserta una fila con código 20 y jefe 30 y otra al revés.

```
INSERT INTO EMPLEADOS VALUES (20, 'MANUEL', 30);
INSERT INTO EMPLEADOS VALUES (30, 'CARLOS', 20);
```

Restricción DEFERRABLE y SET CONSTRAINTS DEFERRED

Comprobación de Restricciones Postergada

- Por defecto, las restricciones se comprueban al terminar **la instrucción**
- Sin embargo, se pueden postergar hasta que termine la **transacción**
  - Si la restricción postergada es violada, se deshace la transacción
  - Si la restricción inmediata es violada, se deshace la sentencia
- Se puede definir el estado de una restricción al crearla:
  - DEFERRABLE o NOT DEFERRABLE
    - Si es NOT DEFERRABLE (por defecto) siempre se comprueba al terminar la instrucción. Si no, depende de SET CONSTRAINTS.
    - Si es DEFERRABLE despues habremos de indicar si es INITIALLY IMMEDIATE (pode defecto) o INITIALLY DEFERRED.
    - Una restricción NOT DEFERRABLE no puede modificarse a DEFERRABLE. Debe borrarse y volverse a crear.
- Se puede modificar el modo de comprobación de las restricciones de una transacción con la instrucción

```
SET CONSTRAINTS <constraint_names>|ALL DEFERRED|IMMEDIATE
```

- También se puede modificar el estado de una sesión
- `ALTER SESSION CONSTRAINTS = DEFERRED|IMMEDIATE|DEFAULT`

Estado de las restricciones

- **ENABLE** todos los datos que se introducen cumplen la restricción
- DISABLE permite la entrada de datos sin comprobar la restricción
- **VALIDATE** los datos que existen deben cumplir la restricción
- NOVALIDATE algunos datos existentes pueden no cumplir la restricción
- ENABLE VALIDATE = ENABLE
- ENABLE NOVALIDATE → P.e. tablas con muchos datos que sabemos que son correctos, no necesita comprobar la restricción
- DISABLE NOVALIDATE = DISABLE

Control de Concurrencia/Consistencia

- Una BD puede tener múltiples accesos simultáneos a los mismos datos.
  - Por lo tanto, estos accesos deben estar controlados. Es lo que se conoce como **control de concurrencia (data concurrency)**. Este control para asegurar la **consistencia** (o **data consistency,** estado coherente de los datos que ven los usuarios) es vital.
  - Mientras que el **aislamiento entre las transacciones** es generalmente deseable, el funcionamiento de muchas aplicaciones en este modo puede comprometer seriamente rendimiento.
  - Oracle usa un modelo de consistencia **multiversion** así, como distintos tipos de **locks** para las transacciones.

Control de Concurrencia/Consistencia

- El estándar ANSI/ISO **SQL 92** define **Cuatro Niveles de Aislamiento** de una transacción con diferentes grados de impacto sobre el proceso de transacciones. Estos niveles de aislamiento se definen en relación a **Tres Fenómenos** que deben prevenirse cuando se produce concurrencia de transacciones. Los tres fenómenos referidos son:
  - **Lecturas Erróneas** (*dirty reads*)**:** Una transacción lee datos que han sido escritos por otra transacción que aún no ha sido confirmada (con `COMMIT`). Oracle SIEMPRE evita este fenómeno.
  - **Doble Lectura** (*non-repeatable reads*)**:** Una transacción lee datos que ya había leído, encontrándose que, entre las dos lecturas, los datos han sido modificados o borrados por una transacción que ya ha sido confirmada.
  - **Lectura Fantasma** (*phantom read*)**:** Una transacción reejecuta una consulta encontrando que el conjunto de filas resultantes ha sido ampliado por otra transacción que insertó nuevas filas y que ya ha realizado su `COMMIT`.

Control de Concurrencia/Consistencia

- Los **4 Niveles de Aislamiento de SQL 92** son los siguientes:

Nivel de Aislamiento	Lectura Errónea	Doble Lectura	Lectura Fantasma
Read Uncommited	Posible	Posible	Posible
Read Commited	No posible	Posible	Posible
Repeatable Read	No posible	No posible	Posible
Serializable	No posible	No posible	No posible

- **Oracle ofrece estos Niveles de Aislamiento:**
  - **`READ COMMITED`:** Cada consulta que ejecuta una transacción ve solamente los datos que han sido confirmados con anterioridad al inicio de ejecución de la consulta (no de la transacción): **Consistencia a Nivel de Sentencia**.
    - De esta forma, nunca se leerán datos sin confirmar.
    - Es el nivel de aislamiento por defecto.
    - Si una sentencia DML requiere modificar filas que están bloqueadas por otra transacción, la sentencia esperará hasta que se libere el bloqueo.
    - Es equivalente a **`READ WRITE`**.
  - **`SERIALIZABLE`:** Las transacciones ven solamente los datos que han sido confirmados con anterioridad al inicio de la transacción, exceptuando los cambios que ellas mismas realicen: **Consistencia a Nivel de Transacción**.
    - **`READ ONLY`:** Igual que el **`SERIALIZABLE`**, pero además no permite que la transacción realice ninguna modificación en los datos (a menos que el usuario sea SYS). Este nivel no es SQL 92.
    - Los datos que ve la transacción actual son los datos confirmados (*committed*) antes de que la transacción empezara.
    - Esto es útil para informes que consultan muchas tablas que están continuamente siendo actualizadas.

Control de Concurrencia/Consistencia

- **Puede cambiarse de Nivel:**
  - Al **Principio de una Transacción**, con la orden:

```
SET TRANSACTION
  ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED | READ
ONLY | READ WRITE};
```

  - Para todas las transacciones siguientes en una **Sesión:**

```
ALTER SESSION SET
  ISOLATION_LEVEL = {SERIALIZABLE | READ COMMITTED};
```

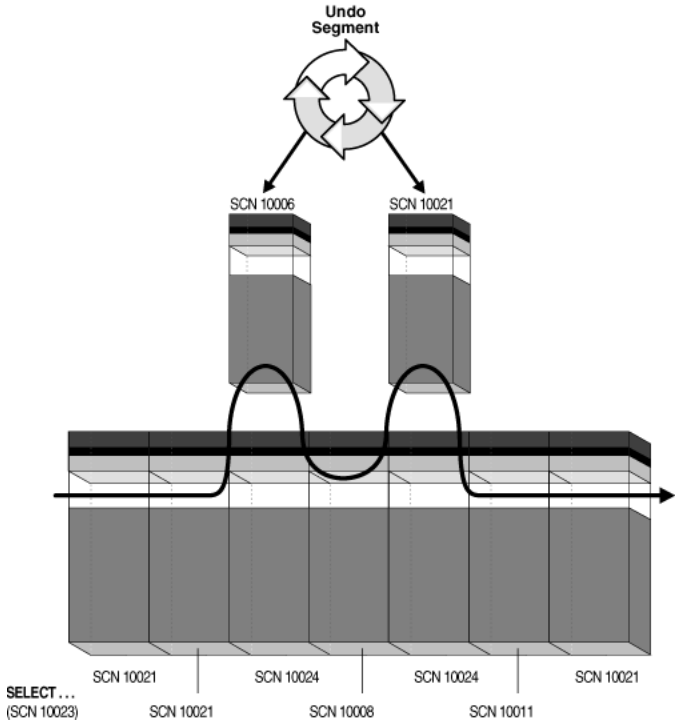
- **Puede cambiarse de segmento de *Rollback* en una Transacción:**

```
SET TRANSACTION USE ROLLBACK SEGMENT <nombre_seg_Rollback>;
```

  - Implícitamente establece esta transacción como **`READ WRITE`**, ya que el modo **`READ ONLY`** no genera información en los segmentos de *rollback*.
  - Es útil en transacciones que modifican muchos datos, para las que posiblemente no haya espacio en el segmento de *rollback* por defecto.
- La sentencia **`SET TRANSACTION`** sólo puede usarse al principio de una transacción.

Control de Concurrencia/Consistencia

- Para conseguir la Consistencia a Nivel de **Sentencia** o de **Transacciones**, Oracle utiliza la información incluida en los Segmentos de *Undo*.
  - **Oracle** asigna a cada transacción un número **SCN** (*System Change Number*) que graba junto con los datos en el segmento de undo. El SCN es un timestamp.
  - Los datos cambiados más recientemente (SCN más reciente que la **consulta** o **transacción** en curso) son reconstruidos usando los datos de los segmentos de *undo*.
  - En la **Figura**, la consulta en ejecución tiene el **SCN** 10023, por lo que esa consulta sólo leerá los datos cambiados con anterioridad.
  - Así que los datos con SCN mayor a 10023 son leídos de los segmentos de *rollback/undo*.



Ventajas y desventajas READ COMMITED

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 a transaction by updating the Banda salary. The default isolation level for transaction 1 is READ COMMITTED.
	SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	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	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 salary for Greene successfully **because transaction 1 locked only the Banda row**.

Ventajas y desventajas READ COMMITED

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, 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	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'; -- prompt does not return	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;		Transaction 1 commits its work, ending the transaction.
	1 row updated. SQL>	The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary for Banda.

Ventajas y desventajas READ COMMITED

	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz'); LAST_NAME SALARY ---------- ---------- Banda 6300 Greene 9900 Hintz	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;	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		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."**

Ventajas y desventajas SERIALIZABLE

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 SERIALIZABLEisolation 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**.

Ventajas y desventajas SERIALIZABLE

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.
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.

Ventajas y desventajas SERIALIZABLE

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. 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.

Ventajas y desventajas SERIALIZABLE

	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.

Control de Concurrencia/Consistencia

- En **Consultas de Larga Ejecución** (*long-running queries*), puede ocurrir (pero es raro), que **Oracle no pueda devolver unos resultados coherentes**, reconstruyendo la información con los segmentos de *undo*, porque esos datos ya no estén allí por haber sido reescritos a causa de la forma circular del segmento de *undo*.
  - **Esto ocurrirá con mayor probabilidad:**
    1. Si los **segmentos de *undo*** son demasiado **pequeños** y/o
    2. Si el **nivel de modificaciones** de la BD es **muy alto**.
  - En estos casos se produce un mensaje de **error** informando al usuario de este hecho: `ORA-1555: snapshot too old (rollback segment too small)`
  - **Soluciones:**
    - Cambiar apropiadamente la variable undo_retention con alter system (mínimo tiempo que la BD retiene en el undo segment los datos antes de sobreescribirlos).
    - Ejecutar esas operaciones cuando haya pocas transacciones en curso.
    - Obtener un bloqueo compartido (`SHARE`) sobre la tabla de la consulta, para evitar bloqueos largos de otros usuarios durante la transacción.
- Si tenemos **Consistencia a Nivel de Transacciones**, esto también puede ocurrir, con mayor probabilidad, en **Transacciones Largas**.

Bloqueos DML (DML Locks)

- **BLOQUEOS (*locks*):** Mecanismos para prevenir los problemas de interacción entre usuarios accediendo al mismo recurso.
  - **Dos niveles de Bloqueo en una BD multiusuario:**
    - **Bloqueos Exclusivos (*exclusive locks*):** No permite compartir recursos. La primera transacción que bloquea un recurso exclusivamente es la única que puede usarlo hasta que lo libere.
    - **Bloqueos Compartidos (*share locks*):** Permite compartir recursos dependiendo del tipo de operaciones involucradas.
  - **Dos Tipos de Bloqueos:**
    - **Bloqueo de Fila (TX):** Este es el menor (en cuanto a tamaño) bloqueo posible.
      - Se produce cuando se ejecuta una instrucción `INSERT`, `UPDATE`, `DELETE` o `SELECT` con la cláusula `FOR UPDATE`.
      - No permite la modificación de la fila **misma** por otra transacción.
      - Cuando una transacción obtiene este tipo de bloqueo exclusivo de una fila, también obtiene un **bloqueo de tabla**.
    - **Bloqueo de Tabla (TM):**
      - Una transacción obtiene este tipo de bloqueo de toda la tabla cuando realiza una operación **DML** de las enumeradas anteriormente o usa la sentencia `LOCK TABLE`.
      - Evita que otra transacción consiga un **bloqueo DDL**, el cual es necesario para efectuar alguna operación de **DDL** sobre la tabla.

Bloqueos DML (DML Locks)

- Por tanto un lock afecta a la interacción entre lecturas y escrituras. Las reglas que resumen el comportamiento de Oracle al respecto:
  - Una fila es bloqueada sólo cuando es modificada.
    - La escritura de una fila bloquea a otra escritura **sobre la misma fila**
  - Una lectura no bloquea una escritura.
    - La única excepción es la sentencia SQL SELECT ... FOR UPDATE.
  - Una escritura no bloquea a una lectura
    - Como hemos visto anteriormente, se usan los segmentos undo para proporcionar a la lectura una vista consistente.
  - Existen excepciones cuando se trata de transacciones sobre BD distribuidas.

Bloqueos DML (DML Locks)

- Un **Bloqueo de una Tabla** se puede producir de varios **MODOS** que determinan las operaciones que se permiten a las otras transacciones sobre la misma tabla**:**
  - ***RS:*** **`ROW SHARE`** ***Table Lock:***
    - Indica que la transacción ha bloqueado filas de la tabla y va a modificarlas.
    - Permite compartir la tabla con otras transacciones en cualquier modo salvo que soliciten un bloqueo de las mismas filas, o bien, un bloqueo exclusivo de la tabla.
    - Sentencias SQL que bloquean RS:
      ```
      SELECT ... FROM ... FOR UPDATE OF ... ;
      LOCK TABLE <Tabla> IN ROW SHARE MODE;
      ```
  - ***RX:*** **`ROW EXCLUSIVE`** ***Table Lock:***
    - Indica que la transacción ha realizado bloqueos sobre algunas filas de la tabla. Se produce con las instrucciones **`INSERT`**, **`UPDATE`**, **`DELETE`**.
    - Sólo compatible con otros RX y RS.
  - ***S:*** **`SHARE`** ***Table Lock:***
    - Se adquiere cuando se ejecuta la instrucción **`LOCK TABLE`** asociada.
    - No permite realizar modificaciones en la tabla, aunque sí lecturas, así como otros bloqueos del mismo tipo S, o de tipo RS.
    - Si varias transacciones tienen este bloqueo para la misma tabla, ninguna podrá modificar la tabla.
  - ***SRX:*** **`SHARE ROW EXCLUSIVE`** ***Table Lock:***
    - Se adquiere mediante la instrucción **`LOCK TABLE`** asociada y, en un momento dado, sólo puede conseguir este tipo de bloqueo una transacción para una tabla concreta.
    - Solamente permite a otras transacciones realizar lecturas de la tabla o bloquear ciertas filas con **`SELECT ... FOR UPDATE`**, pero no permite modificar la tabla.
  - ***X:*** **`EXCLUSIVE`** ***Table Lock:***
    - Es el modo más restrictivo de bloqueo y permite, a la transacción que lo obtiene, un acceso en exclusiva para escribir en la tabla.
    - Sólo permite a otras transacciones consultas normales.

Bloqueos DML (DML Locks)

– **Resumen de los MODOS de Bloqueo de una Tabla:**
  - Operaciones que producen cada modo.
  - Operaciones permitidas o no cuando otra transacción tiene cierto modo de bloqueo.

Instrucción SQL	Bloqueo de Fila	Bloqueo de Tabla	¿Modos Permitidos? RS	RX	S	SRX	X
`SELECT ... FROM table`			Sí	Sí	Sí	Sí	Sí
`INSERT INTO table ...`	TX	RX	Sí	Sí	No	No	No
`UPDATE table ...`	TX	RX	Sí*	Sí*	No	No	No
`DELETE FROM table ...`	TX	RX	Sí*	Sí*	No	No	No
`SELECT...FROM table ...FOR UPDATE OF...`	TX	RS	Sí*	Sí*	Sí*	Sí*	No
`LOCK TABLE table IN ...`							
`ROW SHARE MODE`		RS	Sí	Sí	Sí	Sí	No
`ROW EXCLUSIVE MODE`		RX	Sí	Sí	No	No	No
`SHARE MODE`		S	Sí	No	Sí	No	No
`SHARE ROW EXCLUSIVE MODE`		SRX	Sí	No	No	No	No
`EXCLUSIVE MODE`		X	No	No	No	No	No

* Sí, si no existe otra transacción con bloqueos de fila conflictivos; En otro caso se produce una espera.

– **Sentencia `LOCK TABLE`:** Bloquea una tabla o vista en el modo indicado.

```
LOCK TABLE <Tabla> IN <Modo> MODE [NOWAIT];
```

- **`NOWAIT`:** Indica que si no puede hacer el bloqueo, que no espere y produzca un error indicando que la tabla está bloqueada por otra transacción.

Deadlocks

- **Un Deadlock ocurre cuando dos o más usuarios están esperando, respectivamente, datos que están bloqueados por el otro usuario.**
  - **Ejemplo:**

Sin problemas: Cada transacción tiene un bloqueo a distinta fila.

Problema: Ninguna transacción puede conseguir el recurso que necesita.

***Deadlock*:** Oracle informa del problema a una transacción y la deshace (*rollback*).

Usuario 1	Usuario2

`UPDATE Pieza`
`SET Nombre='TUERCA'`
`WHERE Codigo = 1000;`
	`UPDATE Pieza`
	`SET Peso=10`
	`WHERE Codigo=2000;`

`UPDATE Pieza`
`SET Precio=Precio*1.1`
`WHERE Codigo= 2000;`
	`UPDATE Pieza`
	`SET Peso=10`
	`WHERE Codigo=1000;`

	`ORA-00060: Deadlock detected while waiting for resource.`

  - **Los deadlocks son raros en Oracle**, ya que sus bloqueos son por cada fila y no por páginas.
    - Además, Oracle sólo bloquea por defecto las filas necesarias y no bloquea una tabla completamente aunque muchas de sus filas estén bloqueadas (no hay *lock escalation*).
  - **Los *Deadlocks* entre varias tablas pueden evitarse si las transacciones que acceden a esas tablas las bloquean en el mismo orden, a través de bloqueos implícitos o explícitos**

Gestión de Transacciones

- Una transacción es una unidad lógica y atómica de trabajo que contiene una o más sentencias SQL. Agrupa las sentencias de modo que, o se realizan todas o se deshacen todas.
- Cada transacción tiene un identificador único llamado ID de transacción.
- Las transacciones cumplen las propiedades ACID (Atomicidad, Consistencia, alslamiento y Durabilidad)
- Toda transacción tiene un comienzo y un fin:
  - Comienzo:
    - Con la primera sentencia SQL encontrada (DML, DDL)
    - Cuando comienza se le asigna un segmento de undo y, a continuación, un ID de transacción.
      - UPDATE alumnos set nombre = nombre;
      - SELECT XID AS "id", XIDUSN AS "undo seg", STATUS AS "estado" FROM V$TRANSACTION;

Gestión de Transacciones

  - Fin de una transacción:
    - COMMIT
    - ROLLBACK sin la clausula SAVEPOINT
    - Ejecución de sentencia DDL (Oracle ejecuta un commit implícito antes y después de cada sentencia DDL)
    - Terminación de la sesión (por defecto se hace commit, pero es configurable)
    - Un proceso de cliente aborta (se hace rollback implícito)
- Cuando una transacción acaba, la siguiente sentencia SQL(*) en esa sesión inicia otra transacción (con otro ID)
- Atomicidad de Sentencia
  - Si una sentencia aborta se hace rollback de la sentencia, no de la transacción. El efecto del rollback de la sentencia es como si nunca se hubiera ejecutado (incluso lo ejecutado por triggers provocados por la sentencia)
  - Es por defecto en Oracle

Sentencias de Control de Transacciones

- COMMIT.
  - Termina la transacción
  - Borra todos los savepoints
  - Se liberan los bloqueos realizados por la transacción
- ROLLBACK
  - Todos los cambios desde el ultimo COMMIT o ROLLBACK se deshacen
- ROLLBACK TO SAVEPOINT.
  - Deshace los cambios desde el SAVEPOINT
- SAVEPOINT nombre
  - Identifica un punto de la transacción al cual poder volver
- SET TRANSACTION NAME
  - Opcional, pero es útil para ver el estado de las transacciones
  - Se pueden encontrar más fácil en V$TRANSACTION

Sentencias de Control de Transacciones

time	Session	Explicación
T0	SET TRANSACTION NAME ‘mis_notas’;	Comienzo
T1	UPDATE notas set nota = 5 where nombre =‘Maria’;	Aprueba a Maria
T2	SAVEPOINT aprobar_a_maria;	
T3	UPDATE notas set nota = 5 where nombre =‘Luis’;	Aprueba a Luis
T4	SAVEPOINT aprobar_a_luis;	
T5	ROLLBACK TO SAVEPOINT aprobar_a_maria;	María sigue aprobada, Luis no
T6	ROLLBACK;	Deshace todos los cambios
T7	SET TRANSACTION NAME ‘otras_notas’;	Comienza otra transacción
T8	UPDATE notas set nota = 10 where nombre =‘Enrique’;	Un 10 a Enrique
T9	UPDATE notas set nota = 10 where nombre =‘Manuel’;	Otro para Manuel
T10	COMMIT;	Se graban los cambios de la transacción otras_notas. Los cambios también se graban en el redo log.