Solución Interrogación 2 - IIC2413

CAL

Octubre 2025

Pregunta 1: Lógica en la BD

a) Vista vw_activos_detalle

```
CREATE VIEW vw_activos_detalle AS
SELECT a.nombre, a.categoria, a.valor, a.estado,
       u.nombre AS ultima_ubicacion,
       m.fecha_movimiento AS fecha_ultimo_movimiento
FROM Activo a
JOIN Movimiento m ON a.id_activo = m.id_activo
JOIN Ubicacion u ON m.id_ubicacion = u.id_ubicacion
WHERE m.fecha_movimiento = (
    SELECT MAX(fecha_movimiento)
    FROM Movimiento
    WHERE id_activo = a.id_activo
);
```

b) Procedimiento sp_registrar_movimiento

```
CREATE OR REPLACE FUNCTION sp_registrar_movimiento(
    p_id_activo INT,
    p_id_ubicacion INT,
    p_tipo_movimiento VARCHAR
) RETURNS VOID AS $$
BEGIN
    INSERT INTO Movimiento(id_activo, id_ubicacion, fecha_movimiento, tipo_movimiento
    VALUES (p_id_activo, p_id_ubicacion, CURRENT_DATE, p_tipo_movimiento);

    IF p_tipo_movimiento = 'ENTRADA' THEN
        UPDATE Activo SET estado = 'EN USO' WHERE id_activo = p_id_activo;
    ELSE
        UPDATE Activo SET estado = 'FUERA DE USO' WHERE id_activo = p_id_activo;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

c) Trigger `trg_valor_activo`

```
CREATE OR REPLACE FUNCTION fn_validar_valor() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.valor <= 0 THEN
        RAISE EXCEPTION 'El valor del activo debe ser mayor a 0';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_valor_activo
BEFORE INSERT ON Activo
FOR EACH ROW EXECUTE FUNCTION fn_validar_valor();
```

Pregunta 2: ACID, Transacciones y Locks

a) Conceptos

- **Lectura sucia:** Una transacción lee datos modificados por otra transacción que aún no hace commit.

Table 1: Ejemplo de Lectura Sucia

Orden	T1	T2
1	W(X)	
2		R(X)
3	ABORT	
4		COMMIT

- **Lectura no repetible:** Una transacción lee el mismo dato dos veces y obtiene valores distintos porque otra transacción lo modificó y confirmó entre ambas lecturas.

Table 2: Ejemplo de Lectura No Repetible

Orden	T1	T2
1	R(X)	
2		W(X)
3		COMMIT
4	R(X)	

- **Actualización perdida:** Dos transacciones actualizan el mismo dato y una sobrescribe el cambio de la otra sin advertirlo.

Table 3: Ejemplo de Actualización Perdida

Orden	T1	T2
1	R(X)	
2		R(X)
3	W(X)	
4		W(X)
5		COMMIT
5	COMMIT	

b) Cronograma S

Conflictos presentes:

- Lectura sucia: Ninguna de las 3 transacciones lee datos que han sido modificadas por otra transacción, por lo tanto no ocurre.
- Lectura no repetible: Ninguna transacción lee 2 veces una variable, por lo tanto no ocurre.
- Actualización perdida: Sucede con la transacción *T1* que sobreescribe (`W(Y)`) datos de la transacción *T2*, con la transacción *T3* que sobreescrube (`W(X)`) datos de la transacción *T1* y datos (`W(Z)`)de la transacción *T2*.

c) Grafo de precedencia

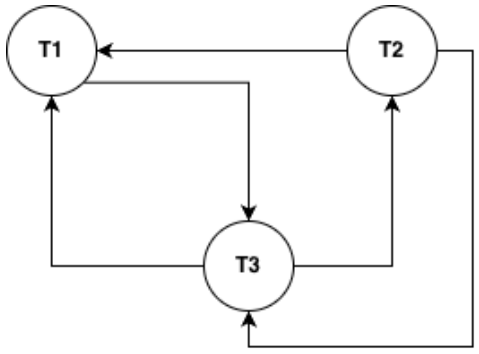


Figure 1: Grafo de precedencia

El grafo tiene ciclos, por lo tanto **no es Conflict Serializable**.

d) Permutaciones para S2

Uno de los posibles schedules que se puede llegar es el que se muestra en el cuadro 4, el cual no es posible continuar por las operaciones sobre `X` entre *T1* y *T3*. Por lo tanto **no es Conflict Serializable**

Table 4: Permutaciones S2

T1	T2	T3
R(X)		
		R(Z)
		W(X)
		R(X)
		W(Z)
	W(Y)	
	R(Y)	
	W(Z)	
W(X)		
W(Y)		

Pregunta 3: Log y Recuperación

a) Checkpoints

Respuesta correcta: **iii. Sólo I, II, IV y V son verdaderas.** La afirmación **III** es falsa porque en *Redo Logging* todas las transacciones que habian realizado `Commit` deben finalizar escribiendo `END` y después se escribe `END CKPT`

b) Undo Logging

En *Undo Logging* las transacciones que han realizado `Commit` ya tienen sus datos escritos en disco, por lo tanto si hay falla no se pierde su información. Las transacciones que no han realizado *Commit* no hay seguridad que los datos están en disco y lor lo tanto en caso de falla debe deshacerse.

De acuerdo a lo anterior, se deshace T2 (porque no hizo commit). Se restauran valores originales de Z.

c) Redo Logging

En *Redo Logging* las transacciones que no han realizado `Commit` no han finalizado, por lo tanto no se hace nada en caso de falla. Las transacciones que han realizado *Commit* y no han realizado *End* se deben rehacer. Finalmente las transacciones que han realizado `Commit` y `End` no se hace nada porque sus datos ya están escritos en disco.

En este caso la única transacciones que se debe rehacer es la transacción *T1* que ha realizado `Commit` y no ha realizado `End`. *T2* ya realizó `End` y *T3* no ha realizado `Commit.`

Pregunta 4: Índices y Costos

a. B+ unclustered sobre c: Costo $\approx h + 21 \div P + 21$

b. B+ clustered sobre b: Consulta por d=5 no usa índice, costo $\approx 2.000.000 \div M$ (escaneo completo).

c. Hash clustered sobre a: Costo ≈ 1 (búsqueda exacta).

Bonus

a. Correcta: **C. Sólo I es verdadera.**

- II es falsa porque el módulo que es responsable de los errores es *Recovery Manager*
- III es falsa porque *RAID* sirve para fallas de almacenamiento, pero en caso de fallas catastróficas se usan copias distribuidas o respaldos totates/incrementales

b. Correcta: **B. Sólo II y III son verdaderas.**

- I es falsa, porque la *Atomicidad* es que una transacción se ejecuta completa o no se ejecuta.

c. Correcta: **D. Hash ideal para exactas, B+ para rangos.**

Los índices *Hash* son ideales para búsquedas exactas porque son de $O(1)$ y en cambio los $B+$ es $O(h)$. En cambio para búsqueda de rango los índices *Hash* es $O(Tuplas + Tuplas * BúsquedaListaLigada)$ y en $B+$ es $O(h + PáginasRango)$