Manejo de concurrencia y aislamiento de transacciones

Introducción

En entornos de bases de datos, el manejo de **concurrencia** es crucial para garantizar que múltiples transacciones que acceden a los mismos datos no generen inconsistencias ni resultados inesperados. El grado en que una transacción está aislada de otras determina los **niveles de aislamiento** y define la forma en que las transacciones interactúan entre sí.

Este apunte proporciona las bases para entender la importancia del manejo de concurrencia y el aislamiento de transacciones, y su impacto directo en la integridad de los datos. A través de ejemplos prácticos sobre competencia entre transacciones y problemas comunes, se explora cómo manejar en forma adecuada los niveles de aislamiento para evitar conflictos en entornos multiusuario. Además, se introducen los conceptos de bloqueo optimista y bloqueo pesimista, que, en combinación con los niveles de aislamiento, ofrecen una visión completa sobre las estrategias de control de concurrencia.

Niveles de Aislamiento: Definiciones y Comportamiento de Lecturas

Los niveles de aislamiento definidos en el estándar SQL son cuatro: **Read Uncommitted**, **Read Committed**, **Repeatable Read**, y **Serializable**. Sin embargo, cada sistema puede tener variaciones en estos niveles o incluir niveles adicionales específicos.

a. Read Uncommitted (Lectura No Confirmada)

- **Descripción**: Permite que una transacción lea datos no confirmados de otras transacciones, lo que puede resultar en lecturas sucias (“dirty read”).
- **Problemas Potenciales**:
    - **Dirty Read**: Una transacción lee cambios que otra transacción aún no ha confirmado, lo que genera inconsistencia si la segunda transacción realiza un ROLLBACK.
    - **Locks**: En este nivel, se generan pocos o ningún bloqueo, lo que maximiza la concurrencia pero sacrifica la integridad.

b. Read Committed (Lectura Confirmada)

- **Descripción**: Las transacciones solo pueden leer datos confirmados, evitando lecturas sucias.
- **Problemas Potenciales**:
    - **Non-Repeatable Read**: Los datos leídos por una transacción pueden cambiar si otra transacción modifica y confirma esos datos antes de que la primera transacción finalice.
- **Locks**: Se generan bloqueos de lectura compartidos que se liberan una vez que la lectura ha terminado, evitando lecturas de datos no confirmados.

c. Repeatable Read (Lectura Repetible)

- **Descripción**: Asegura que todas las lecturas dentro de una transacción sean consistentes; es decir, los datos no cambiarán durante toda la transacción.
- **Problemas Potenciales**:
    - **Phantom Read**: Nuevas filas que cumplen con una condición pueden aparecer si otra transacción las inserta y confirma.
- **Locks**: Las lecturas son bloqueadas hasta que la transacción finaliza, lo que evita lecturas no repetibles, pero pueden aparecer filas “fantasma”.

d. Serializable (Serializable)

- **Descripción**: El nivel más alto de aislamiento. Asegura que las transacciones se ejecutan de manera secuencial.
- **Problemas Potenciales**: Este nivel evita todos los problemas de concurrencia, pero disminuye el rendimiento debido a los bloqueos.
- **Locks**: Bloqueos completos que aseguran que las transacciones se ejecuten sin interferencia de otras transacciones, evitando cualquier tipo de anomalía.

Problemas Comunes de Concurrencia

- **Lost Update (Actualización Perdida)**: Dos transacciones leen el mismo valor y ambas intentan actualizarlo. La última actualización sobrescribe la primera.
- **Dirty Read (Lectura Sucia)**: Una transacción lee datos que otra transacción aún no ha confirmado.
- **Non-Repeatable Read (Lectura No Repetible)**: Una transacción vuelve a leer datos y encuentra que estos han cambiado debido a otra transacción.
- **Phantom Read (Lectura Fantasma)**: Una transacción consulta varias veces un conjunto de datos y observa la aparición de nuevas filas.

Ejemplo de Competencia entre Transacciones

Imaginemos un escenario en el que tenemos tres transacciones en competencia:

- **Transacción T1 y T2**: Crean nuevas facturas y calculan el precio de productos en función del precio actual.
- **Transacción T3**: Actualiza el precio de algunos productos mientras las facturas están en proceso.

Escenario: T1 y T2 calculan precios basados en un precio "actual" que T3 puede modificar en cualquier momento. Esto puede provocar errores en las facturas si T3 cambia un precio después de que T1 o T2 lo hayan leído pero antes de que confirmen la factura.

Paso a Paso de la Competencia:

Paso 1: T1 y T2 leen el precio actual del producto "A" como $100.

```
-- Transacción T1
BEGIN;
SELECT precio FROM producto WHERE id = 1;  -- Resultado: $100
-- Transacción T2
BEGIN;
SELECT precio FROM producto WHERE id = 1;  -- Resultado: $100
```

Paso 2: T3 inicia y modifica el precio del producto "A" a $120 y confirma (COMMIT).

```
-- Transacción T3
BEGIN;
UPDATE producto SET precio = 120 WHERE id = 1;
COMMIT;
```

Paso 3: T1 y T2 intentan confirmar sus facturas, basándose en el precio de $100 que habían leído inicialmente.

- **Problema**: Las facturas se generan con precios desactualizados (T1 y T2 mantienen el precio de $100 mientras T3 lo actualizó a $120).

Este problema se llama **Non-Repeatable Read**, donde la lectura inicial de un dato cambia debido a una actualización en otra transacción. Si T1 y T2 hubieran ejecutado en el nivel de aislamiento

Serializable, T3 no habría podido confirmar la actualización hasta que T1 y T2 terminaran, evitando la inconsistencia.

El nivel de aislamiento Serializable proporciona la máxima integridad al bloquear tanto la lectura como la modificación de datos y rangos de datos, asegurando que T1 y T2 puedan completar sus operaciones sin que T3 interfiera. Este tipo de bloqueo es útil cuando la consistencia total es crítica, pero puede reducir la concurrencia y el rendimiento en sistemas con alto volumen de transacciones simultáneas.

Situaciones de concurrencia

a. Lost Update

- Dos transacciones (T1 y T2) leen el mismo precio de un producto y ambas intentan actualizarlo. La última transacción que confirma sobrescribe la primera.
- **Gravedad**: Alta. Los cambios de una transacción se pierden por completo.

b. Dirty Read

- T1 lee un precio que T3 ha actualizado, pero que aún no ha confirmado. Si T3 hace ROLLBACK, T1 ha leído datos inconsistentes.
- **Gravedad**: Alta en escenarios críticos, ya que causa lecturas inconsistentes.

c. Non-Repeatable Read

- T1 lee un precio de producto y T3 lo actualiza antes de que T1 termine. Cuando T1 intenta leerlo de nuevo, encuentra un valor diferente.
- **Gravedad**: Media a alta, ya que puede causar errores en cálculos basados en datos temporales.

d. Phantom Read

- T1 consulta productos bajo cierto criterio (precio > $50). Si T3 inserta un nuevo producto que cumple con este criterio, T1 podría ver un “fantasma” al ejecutar la consulta de nuevo.
- **Gravedad**: Media, aunque puede confundir y generar reportes inconsistentes.

Ejemplos de Implementación de Niveles de Aislamiento

PostgreSQL:

```
-- Establecer nivel de aislamiento en PostgreSQL
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
-- Operaciones de transacción aquí
COMMIT;
```

SQL Server:

```
-- Establecer nivel de aislamiento en SQL Server
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
-- Operaciones de transacción aquí
COMMIT TRANSACTION;
```

Configuración de niveles de aislamiento. Bloqueos generados

Este apartado cubre las configuraciones predeterminadas y las herramientas para consultar tanto el nivel de aislamiento como los bloqueos generados en cada nivel y tipo de operación SQL, ya sea en PostgreSQL como en SQL Server.

Niveles de aislamiento por defecto

- **PostgreSQL**:

  El nivel de aislamiento por defecto es **Read Committed**. Esto significa que cada instrucción SQL dentro de una transacción ve solo los datos confirmados por otras transacciones hasta el momento en que se ejecuta esa instrucción.
- **SQL Server**:

  SQL Server también tiene por defecto el nivel de aislamiento **Read Committed**.

  En este nivel, las transacciones solo pueden leer datos que han sido confirmados, y las lecturas crean bloqueos compartidos (S-locks), que se liberan al finalizar la lectura.
  SQL Server permite habilitar Read Committed Snapshot Isolation (RCSI), que implementa versiones de datos usando instantáneas y reduce los bloqueos en lecturas al permitir que las consultas lean versiones previas de datos sin bloquear las actualizaciones concurrentes.

Cómo ver el nivel de aislamiento actualmente aplicado

- **PostgreSQL**:

```
SHOW TRANSACTION ISOLATION LEVEL;

SET TRANSACTION ISOLATION LEVEL <nivel>;  -- Ejemplo: SET TRANSACTION
ISOLATION LEVEL SERIALIZABLE;
```

- **SQL Server**:

```
DBCC USEROPTIONS;
```

  Este comando muestra las opciones de configuración de la sesión actual, incluyendo el nivel de aislamiento activo.

Niveles de Aislamiento en PostgreSQL y SQL Server

No todos los motores de bases de datos ofrecen los mismos niveles de aislamiento, y la implementación de cada nivel puede variar ligeramente según el motor. Los niveles de aislamiento definidos en el estándar SQL son cuatro: **Read Uncommitted**, **Read Committed**, **Repeatable Read**, y **Serializable**. Sin embargo, cada SGBD puede tener variaciones en estos niveles o incluir niveles adicionales específicos.

PostgreSQL

Los tipos de bloqueo y su comportamiento dependen tanto de la instrucción SQL como del nivel de aislamiento:

1. **Read Uncommitted:** no implementado.
2. **Read Committed (Predeterminado)**:
    - **SELECT**: No genera bloqueos de lectura exclusivos; las lecturas no bloquean a otras transacciones.
    - **INSERT/UPDATE/DELETE**: Generan un **Row-Level Exclusive Lock (ROW EXCLUSIVE)** en las filas afectadas, bloqueando cualquier otra modificación de esas filas.
3. **Repeatable Read**:
    - **SELECT**: Evita lecturas no repetibles bloqueando los datos leídos por la transacción. Sin embargo, pueden ocurrir lecturas fantasma.
    - **UPDATE/DELETE**: Generan un **Row-Level Exclusive Lock** que asegura que otras transacciones no puedan modificar las filas leídas hasta que la transacción actual se complete.
    - **Bloqueos de Sesión**: Utiliza pg_stat_activity junto con pg_locks para ver bloqueos y sesiones activas.
4. **Serializable**:
    - **SELECT**: Bloquea toda lectura de filas si la transacción puede potencialmente interferir con otras, evitando tanto lecturas fantasma como lecturas no repetibles. Este nivel implementa **Serializable Snapshot Isolation (SSI)** en PostgreSQL.
    - **UPDATE/DELETE**: Genera bloqueos de fila y de rango en todos los datos leídos o escritos, asegurando la máxima integridad.

SQL Server

SQL Server utiliza distintos tipos de bloqueos (compartidos, exclusivos, de actualización) según el nivel de aislamiento.

1. **Read Committed (Predeterminado)**:
    - **SELECT**: Genera **Shared Locks (S-locks)**, que se liberan al finalizar la instrucción de lectura.

- **INSERT/UPDATE/DELETE**: Genera un **Exclusive Lock (X-lock)**, que impide que otras transacciones lean o modifiquen las filas hasta que se complete la transacción.

2. **Repeatable Read**:
    - **SELECT**: Mantiene los bloqueos compartidos durante toda la transacción, evitando que otras transacciones realicen cambios en los datos leídos.
    - **UPDATE/DELETE**: Mantiene bloqueos exclusivos en las filas afectadas durante la transacción.
3. **Serializable**:
    - **SELECT**: Bloquea tanto las filas leídas como los rangos de valores, evitando lecturas no repetibles y lecturas fantasma.
    - **UPDATE/DELETE**: Genera bloqueos de rango y exclusivos en todos los datos leídos o escritos, garantizando que ninguna otra transacción pueda interferir con los datos durante la transacción.

4. **Read Committed Snapshot Isolation (RCSI)**
    - **SELECT**: En este modo, los SELECT no crean bloqueos compartidos, ya que las consultas leen versiones de datos previas sin interferir con actualizaciones en curso.
    - **INSERT/UPDATE/DELETE**: Operan de forma habitual, generando bloqueos exclusivos donde sea necesario.

Resumen comparativo

Nivel de Aislamiento	Comando	PostgreSQL	SQL Server
Read Committed	SELECT	Sin bloqueos, lectura de datos confirmados	Shared Lock (S-lock), liberado al finalizar
	UPDATE	Row-Level Exclusive Lock	Exclusive Lock (X-lock)
Repeatable Read	SELECT	Mantiene bloqueo de lectura durante la transacción	Mantiene S-lock hasta finalizar la transacción
	UPDATE	Row-Level Exclusive Lock	Exclusive Lock (X-lock)
Serializable	SELECT	Serializable Snapshot Isolation	Range Lock y S-lock hasta finalizar
	UPDATE	Row-Level y Range Lock	Exclusive Lock (X-lock y Range Lock)

Cómo visualizar bloqueos

PostgreSQL

```
SELECT
    pg_stat_activity.datname AS database_name,
    pg_stat_activity.pid AS process_id,
    pg_stat_activity.usename AS user_name,
    pg_stat_activity.query AS current_query,
    pg_locks.locktype AS lock_type,
    pg_locks.mode AS lock_mode,
    pg_locks.granted AS is_granted,
    pg_locks.relation::regclass AS locked_table
FROM
    pg_locks
JOIN
    pg_stat_activity ON pg_locks.pid = pg_stat_activity.pid
```

Tanto pg_locks como pg_stat_activity son vistas del sistema que permiten monitorear el estado y la actividad de la base de datos.

SQL Server

```
SELECT
    tl.request_session_id AS session_id,
    tl.resource_type AS resource_type,
    tl.resource_associated_entity_id AS resource_id,
    tl.request_mode AS lock_mode,
    tl.request_status AS lock_status,
    es.login_name AS login_name,
    es.host_name AS host_name,
    ISNULL(er.command, 'N/A') AS command,
    er.blocking_session_id AS blocking_session_id
FROM
    sys.dm_tran_locks AS tl
JOIN
    sys.dm_exec_sessions AS es ON tl.request_session_id = es.session_id
LEFT JOIN
    sys.dm_exec_requests AS er ON tl.request_session_id = er.session_id
ORDER BY
    tl.request_session_id;
```

En SQL Server sys.dm_tran_locks, sys.dm_exec_sessions y sys.dm_exec_requests son vistas de administración dinámica que proporcionan información en tiempo real sobre la actividad y el estado del sistema.

Qué nivel de aislamiento usar?

La elección del nivel de aislamiento depende del tipo de transacción y del equilibrio deseado entre consistencia de datos y rendimiento.

1. **Requerimientos de Consistencia**:
    - **Transacciones críticas** (como aquellas que manejan pagos, actualizaciones de inventario, o cambios de precios) suelen requerir un nivel de aislamiento más alto para evitar problemas de consistencia, como lecturas sucias o actualizaciones perdidas.
    - **Transacciones de lectura intensiva** o aquellas en las que cierta inconsistencia temporal no es crítica (como reportes o análisis exploratorios) pueden optar por niveles más bajos de aislamiento, como `Read Committed` o incluso `Read Uncommitted`.
2. **Impacto en el Rendimiento**:
    - Los niveles de aislamiento altos, como `Serializable`, pueden reducir la concurrencia y afectar el rendimiento en sistemas con alto volumen de transacciones simultáneas debido a los bloqueos más restrictivos.
    - Transacciones que requieren acceso rápido y concurrente a los datos, como consultas frecuentes y ligeras, se benefician de un nivel de aislamiento más bajo.
3. **Tipo de Operación en la Transacción**:
    - **Lecturas y reportes**: Pueden usar `Read Committed` o `Repeatable Read` si necesitan coherencia en el tiempo, pero no requieren evitar por completo efectos de concurrencia.
    - **Actualizaciones o escrituras críticas**: Requieren mayor aislamiento para evitar efectos de concurrencia que puedan causar inconsistencias.

Ejemplos de uso

Imaginemos un sistema de ventas e inventario.

- **Actualización de Precios (Transacción Crítica)**: Podría establecerse a nivel `Serializable` para evitar que otros procesos lean precios inconsistentes durante la actualización.
- **Generación de Facturas** : Puede realizarse a nivel `Repeatable Read`, ya que requiere consistencia en el precio y la cantidad de inventario, pero es menos estricta que `Serializable`.
- **Consultas de Reporte de Ventas (Transacción de Solo Lectura)**: Se beneficiarían de un nivel de aislamiento `Read Committed`, que proporciona consistencia básica y permite un alto grado de concurrencia.

La elección de niveles de aislamiento debe realizarse intentando balancear ***integridad de datos y rendimiento****, adaptándose a los requisitos y el entorno de operación de cada sistema de información.*

Bloqueo optimista

Los servidores de aplicaciones modernos como **WildFly** (sucesor de JBoss) usan con frecuencia una técnica de **bloqueo optimista** para manejar la concurrencia. Este tipo de bloqueo es, efectivamente, **complementario a los niveles de aislamiento** de la base de datos, ya que se implementa en la capa de aplicación y no depende de los bloqueos de la base de datos. Esta estrategia es útil para minimizar bloqueos y mantener la concurrencia.

El **bloqueo optimista** asume que las colisiones de datos son raras. En lugar de bloquear los registros de base de datos cuando se leen, permite que múltiples transacciones lean los datos al mismo tiempo. Al momento de actualizar, la aplicación verifica si los datos han cambiado desde la última lectura. Esto se hace comúnmente con un campo de **versión** que se incluye en la condición WHERE de la sentencia UPDATE.

1. **Campo de versión**:
    - La tabla tiene un campo adicional, típicamente llamado version o timestamp, que se incrementa automáticamente con cada actualización de la fila.
    - Este campo actúa como un indicador del estado de la fila y permite a la aplicación verificar si alguien más ha modificado la fila desde la última vez que fue leída.
2. **Actualización con bloqueo optimista**:
    - Cuando la aplicación lee un registro, también lee el valor del campo version.
    - Antes de realizar un UPDATE, la aplicación incluye este valor version en la condición WHERE del UPDATE.
    - Si el UPDATE tiene éxito (es decir, si version no ha cambiado desde la lectura), la transacción se confirma. Si no, el UPDATE no afecta ninguna fila, lo que indica una colisión, y la transacción debe ser reintentada o se lanza un error.

Ejemplo bloqueo optimista (tabla productos)

Paso 1: Lectura del Registro y Campo de Versión

La aplicación lee el producto y el campo version actual:

```
SELECT id, nombre, precio, version
FROM producto
WHERE id = 1;
```

Supongamos que el valor de version es 5.

Paso 2: Intento de Actualización con el Campo de Versión

Cuando la aplicación intenta actualizar el precio, usa el valor version en la condición WHERE:

```
UPDATE producto
SET precio = 120, version = version + 1
WHERE id = 1 AND version = 5;
```

1. Si el UPDATE afecta la fila (es decir, version todavía es 5), entonces la actualización es exitosa, y version se incrementa a 6.
2. Si el UPDATE no afecta ninguna fila (otro proceso cambió el version a 6), la aplicación detecta una colisión de concurrencia, lo que indica que alguien más ha modificado el registro. En este caso, la aplicación puede:
    - **Reintentar** la transacción con el nuevo valor de version.
    - **Abortar** la operación e informar al usuario que los datos fueron modificados por otro proceso.

¿Por qué usar bloqueo optimista?

El bloqueo optimista es útil en sistemas donde las colisiones son poco frecuentes, ya que permite a múltiples transacciones leer los mismos datos simultáneamente sin los costos de bloqueo asociados con el manejo de concurrencia en el nivel de base de datos. Este enfoque es especialmente eficaz en aplicaciones con **alta concurrencia de lectura** y **pocas actualizaciones simultáneas**.

Bloqueo optimista y niveles de aislamiento: ¿Cómo se Complementan?

- **Independencia de la base de datos**: El bloqueo optimista funciona a nivel de aplicación, por lo que no depende de los mecanismos de bloqueo de la base de datos. Esto permite un control de concurrencia adicional que es independiente de los niveles de aislamiento de la base de datos.
- **Complemento a niveles de aislamiento bajo**: En muchos casos, el bloqueo optimista permite que las transacciones funcionen con niveles de aislamiento más bajos, como Read Committed, sin riesgos de inconsistencias. El control en la aplicación asegura que los datos no hayan cambiado entre la lectura y la actualización, minimizando problemas como **lecturas no repetibles** o **lecturas sucias**.

Limitaciones del Bloqueo Optimista

1. **No previene "Phantom Reads"**: Aunque evita actualizaciones concurrentes inconsistentes, el bloqueo optimista no impide que nuevos registros que cumplan con una condición de consulta aparezcan después de la lectura inicial.
2. **Criterio de Reintento**: Si las colisiones de datos son frecuentes, el sistema puede tener que reintentar múltiples veces, lo que afecta el rendimiento y puede anular las ventajas de evitar bloqueos.
3. **Complementariedad**: En aplicaciones con alta concurrencia y donde la precisión transaccional es crítica, el bloqueo optimista debe ser complementado con un nivel de aislamiento adecuado.

Ejemplo Completo: Bloqueo Optimista con Transacciones

Supongamos un escenario donde:

- **T1 y T2** son transacciones que intentan actualizar el mismo producto simultáneamente usando bloqueo optimista.

- **T3** es una transacción que intenta leer y actualizar este producto usando solo el nivel de aislamiento Read Committed.

Escenario:

1. **T1 y T2** leen el producto y su campo version es 5.
2. Ambas transacciones T1 y T2 intentan actualizar el precio y la version:
    - T1 actualiza el precio a $110 y version a 6.
    - T2 intenta actualizar, pero la condición WHERE version = 5 no se cumple, lo que indica una colisión. T2 debe abortar o reintentar con la nueva version = 6.
3. **T3** lee el precio y lo usa en una operación de facturación, sin riesgo de inconsistencia porque el valor de version asegura que siempre usa el valor actualizado más reciente.

El bloqueo optimista permite una solución eficiente para la concurrencia de datos sin depender de bloqueos de base de datos. Funciona bien en aplicaciones de lectura intensiva y cuando las colisiones de actualización son raras. Sin embargo, en entornos donde las actualizaciones concurrentes son frecuentes o donde la precisión transaccional es crítica, el bloqueo optimista debe complementarse con un adecuado nivel de aislamiento y una estrategia de manejo de concurrencia en la base de datos.

Bloqueo Pesimista

El **bloqueo pesimista** es una estrategia de control de concurrencia en la cual una transacción **bloquea el acceso a los datos** desde el momento en que los lee o modifica, impidiendo que otras transacciones puedan leer o actualizar esos mismos datos hasta que la primera transacción haya terminado. Este tipo de bloqueo garantiza que ninguna otra transacción interfiera con los datos en uso, eliminando el riesgo de inconsistencias, pero a costa de la concurrencia.

En el bloqueo pesimista:

1. **Bloqueo al leer o modificar**: la transacción que lee o actualiza los datos los bloquea de manera exclusiva. Esto evita que otros procesos los lean o cambien hasta que el bloqueo se libere, generalmente después de un COMMIT o ROLLBACK.
2. **Anticipación de problemas**: el bloqueo pesimista **anticipa conflictos** y los evita bloqueando el recurso antes de que ocurra cualquier conflicto.
3. **Bloqueo Exclusivo (Exclusive Lock)**: la transacción donde hay actualización mantiene un bloqueo exclusivo sobre los registros, asegurando que ninguna otra transacción pueda leer o escribir en ellos hasta que el bloqueo se libere.

Ejemplo

Supongamos que una aplicación de inventario utiliza bloqueo pesimista para evitar problemas al actualizar la cantidad de productos en stock.

1. **Transacción T1**: Lee el stock del producto “A” y establece un bloqueo pesimista (bloqueo exclusivo).

   ```
   BEGIN TRANSACTION;
   SELECT cantidad FROM inventario WHERE id_producto = 1 FOR UPDATE;
   ```

   - Aquí, FOR UPDATE indica a la base de datos que bloquee el registro seleccionado para escritura. Ninguna otra transacción puede leer ni modificar cantidad en este registro hasta que T1 finalice.
2. **Transacción T2**: Intenta leer el stock del producto “A”.
   - Como T1 tiene un bloqueo exclusivo sobre el registro, T2 queda en espera hasta que T1 finalice con un COMMIT o ROLLBACK.
3. **Actualización y Liberación de Bloqueo en T1**: Una vez que T1 realiza la actualización de cantidad y ejecuta COMMIT, el bloqueo se libera y T2 puede acceder al registro.

¿Cuándo usar bloqueo pesimista?

El bloqueo pesimista es útil en situaciones en las que se espera que los datos tengan un alto riesgo de conflicto entre transacciones concurrentes. Es una buena opción cuando:

- **Las transacciones son largas o tienen muchas lecturas/escrituras en los mismos datos**.
- **La integridad de los datos es crítica** y los errores de concurrencia (como lecturas sucias o actualizaciones perdidas) son inaceptables.
- **Las actualizaciones concurrentes son frecuentes** y no se pueden manejar solo con la estrategia de bloqueo optimista.

Ventajas y desventajas del bloqueo pesimista

Ventajas:

- **Evita conflictos de concurrencia**: Como los datos están bloqueados para otras transacciones, el riesgo de inconsistencias se elimina.
- **Garantiza integridad**: Ideal para situaciones críticas en las que los errores de concurrencia no son tolerables.

Desventajas:

- **Reducción de concurrencia**: dado que los datos están bloqueados, el acceso concurrente se limita, lo que puede afectar el rendimiento en sistemas de alta concurrencia.
- **Riesgo de deadlocks**: Como varias transacciones intentan bloquear recursos, puede ocurrir un deadlock si dos o más transacciones quedan bloqueadas en espera de recursos que la otra tiene.

Ejemplos

Se utiliza el comando SELECT ... FOR UPDATE para bloquear filas para lectura y escritura en una transacción.

```
BEGIN;
SELECT cantidad FROM inventario WHERE id_producto = 1 FOR UPDATE;
-- Realizar actualización
UPDATE inventario SET cantidad = cantidad - 1 WHERE id_producto =
1;
COMMIT;
```

Comparación: Bloqueo Pesimista vs. Bloqueo Optimista

Característica	Bloqueo Pesimista	Bloqueo Optimista
Estrategia	Bloquea de antemano, anticipando conflictos	Verifica conflictos al confirmar la transacción
Concurrencia	Baja, ya que bloquea acceso	Alta, solo falla si ocurre una colisión
Uso Ideal	Transacciones críticas con alto riesgo de conflicto	Aplicaciones de lectura intensiva, pocas actualizaciones concurrentes
Implementación	FOR UPDATE	Campo version en condiciones WHERE
Impacto en Rendimiento	Puede causar bloqueo prolongado y deadlocks	Generalmente mejor rendimiento, requiere reintentos si falla

*La elección de una u otra técnica depende del balance necesario entre **consistencia, concurrencia y rendimiento** en el sistema.*

¿Qué estrategia aplicar?

La elección entre niveles de aislamiento, bloqueo optimista y bloqueo pesimista depende de varios factores, incluyendo el tipo de aplicación, el perfil de concurrencia, la criticidad de los datos y el rendimiento esperado. A continuación, se intenta resumir cuándo es recomendable utilizar cada uno y cuándo evitarlos, así como la posibilidad de usarlos en combinación.

1. Niveles de Aislamiento

Usar en:

- **Sistemas con transacciones frecuentes de lectura y escritura** donde es importante mantener la integridad de los datos, pero la tolerancia a inconsistencias depende de la criticidad de cada operación.
- **Control de concurrencia básico en bases de datos relacionales**. Los niveles de aislamiento son efectivos para resolver problemas comunes de concurrencia (lecturas sucias, lecturas no repetibles, lecturas fantasma).
- **Situaciones de alto tráfico** donde se necesita gestionar la consistencia sin sacrificar el rendimiento global.

Cuándo usar cada nivel:

- **Read Uncommitted**: Rara vez es recomendable, ya que permite lecturas sucias y genera datos inconsistentes. Puede considerarse en reportes o análisis donde los datos exactos no son críticos.
- **Read Committed**: Ideal como nivel predeterminado, ya que evita lecturas sucias y balancea rendimiento y consistencia. Recomendado para la mayoría de aplicaciones comerciales y sistemas OLTP (Online Transaction Processing).
- **Repeatable Read**: Aumenta la consistencia en operaciones donde es crítico que los datos no cambien durante la transacción. Es útil en sistemas financieros y de inventario donde la precisión en el tiempo es crucial.
- **Serializable**: Útil cuando se requiere máxima integridad de datos, como en sistemas bancarios o de facturación. Sin embargo, puede reducir la concurrencia y el rendimiento, por lo que debe usarse solo cuando es esencial evitar cualquier inconsistencia.

Minimizar el uso de niveles de aislamiento altos (Repeatable Read o Serializable):

- En aplicaciones de **alta concurrencia** donde el rendimiento es prioritario, y la posibilidad de inconsistencias menores no representa un riesgo importante.
- Cuando los **bloqueos prolongados** puedan reducir el rendimiento y causar problemas de escalabilidad en el sistema.

2. Bloqueo Optimista

Usar en:

- **Aplicaciones de lectura intensiva** y con baja frecuencia de actualizaciones, donde las colisiones entre transacciones son infrecuentes (por ejemplo, sistemas de consulta de información, aplicaciones de reportes y análisis).

- **Sistemas distribuidos** o **entornos con múltiples capas:** en la capa de aplicación, donde el objetivo es reducir la dependencia en bloqueos y minimizar el uso de transacciones largas en la base de datos.
- **Sistemas que priorizan la concurrencia** y pueden tolerar la necesidad de reintentos en actualizaciones concurrentes.

No aplicable en los siguientes casos:

- En aplicaciones donde las **actualizaciones concurrentes son frecuentes**, ya que esto puede llevar a conflictos frecuentes que causen reintentos constantes y reduzcan el rendimiento.
- En situaciones de alta criticidad donde no se toleran conflictos de actualización y los datos deben estar siempre actualizados y bloqueados mientras son manipulados.

*Es común usar **bloqueo optimista en combinación con niveles de aislamiento bajos** (como Read Committed), especialmente cuando se busca un equilibrio entre rendimiento y consistencia en aplicaciones que toleran reintentos ocasionales.*

3. Bloqueo Pesimista

Usar en:

- **Aplicaciones con alta frecuencia de actualizaciones** y en las que las colisiones son probables y deben evitarse (por ejemplo, sistemas de inventario, sistemas bancarios, control de stock en almacenes).
- **Transacciones largas y complejas** que requieren mantener la integridad de los datos mientras múltiples operaciones se ejecutan en el mismo conjunto de datos.
- **Escenarios críticos en los que las colisiones de datos no son aceptables**, como cuando se actualizan saldos bancarios o se registran transacciones financieras.

Cuándo evitar el bloqueo pesimista:

- En sistemas de **alto volumen de transacciones concurrentes** donde el rendimiento y la concurrencia son prioritarios, ya que los bloqueos prolongados pueden causar cuellos de botella y afectar la experiencia del usuario.
- En aplicaciones donde los conflictos son raros, ya que el bloqueo pesimista impone una sobrecarga innecesaria y reduce la escalabilidad.

Cuándo usarlo en combinación:

- En aplicaciones donde es esencial bloquear ciertos datos críticos, el bloqueo pesimista se puede combinar con niveles de aislamiento altos (Serializable), de modo que solo las transacciones críticas mantengan bloqueos, mientras que otras transacciones utilicen bloqueos menos estrictos.
- Es posible combinar el bloqueo pesimista en áreas críticas del sistema con bloqueo optimista en áreas de consulta, optimizando tanto el rendimiento como la consistencia donde es más necesario.

Ejemplo de combinación de estrategias

Imaginemos un **sistema de ventas e inventario**:

1. **Transacciones de Venta**: Estas transacciones pueden beneficiarse de un **nivel de aislamiento Read Committed con bloqueo optimista**. Aquí, el bloqueo optimista reduce los bloqueos al leer precios y calcular el total.
2. **Actualización de Inventario Crítico**: Para operaciones de ajuste y actualización de inventario, ya sea al confirmar una entrega, recibir nuevos productos o realizar auditorías de stock, un **nivel de aislamiento Repeatable Read o Serializable o el uso de bloqueo pesimista** asegura que no se registren cambios mientras la transacción está en curso, garantizando que el stock se mantenga consistente.
3. **Consultas y Reportes de Ventas**: Las transacciones de solo lectura, como reportes mensuales de ventas o análisis de tendencias, pueden usar el nivel de aislamiento **Read Uncommitted o Read Committed**, sin bloquear los registros que leen. Esto maximiza el rendimiento sin afectar la integridad, ya que la consistencia perfecta no es crítica en estos reportes.

Sugerencias:

- **Usar niveles de aislamiento** como primera línea de control de concurrencia en bases de datos relacionales, ajustando el nivel según los requisitos de consistencia de cada operación.
- **Aplicar bloqueo optimista** en la capa de aplicación cuando la concurrencia sea una prioridad y las actualizaciones concurrentes sean poco frecuentes, para evitar bloqueos innecesarios y mejorar el rendimiento.
- **Reservar el bloqueo pesimista** para transacciones críticas o datos de alta concurrencia donde los conflictos sean frecuentes y la integridad no pueda comprometerse.