Unidad 5: SQL

Stored Procedures

Un procedimiento almacenado es un programa (o procedimiento) almacenado físicamente en una base de datos. Es un conjunto de instrucciones en SQL que podemos guardar y ejecutar cuando queramos.

Ventajas:

- Reutilización de código (distintos sistemas pueden utilizar los mismos SP). Por ejemplo, un mismo SP se puede disparar desde una página web, desde una aplicación Android o desde una clásica aplicación cliente-servidor.

- Mayor rendimiento (los datos no viajan desde la BD hasta la Aplicación para luego hacer cálculos, sino que todo se calcula dentro de la base y solo se entrega el resultado).

- Tráfico de Red: Pueden reducir el tráfico de la red, debido a que se trabaja sobre el motor (en el servidor), y si una operación incluye hacer un trabajo de lectura primero y en base a eso realizar algunas operaciones, esos datos que se obtienen no viajan por la red.

Desventajas:

- La lógica de la aplicación termina distribuida parte en la base de datos y parte en el código de la aplicación.

- Aumenta la dependencia del repositorio de datos (El lenguaje de los SP suele ser bastante diferente entre los distintos motores de BD, por ej., Transact-SQL para SQL Server y PL/SQL para Oracle.) Si en algún momento, se decide cambiar de Base de Datos, esa migración será más compleja.

Sintaxis SP

```
CREATE PROCEDURE [NAME] [@PAREMETERS]
AS
[sql statement]
```

Triggers

Un trigger es un tipo especial de procedimiento almacenado que se ejecuta automáticamente cuando un evento sucede en la base de datos.

Pueden crearse triggers para eventos DML (INSERT, UPDATE, DELETE), para eventos DDL (CREATE, ALTER, DROP), o incluso para eventos del tipo LOGON (evento disparado cuando un usuario inicia sesión).

Pero además de poder setear el evento que dispara el trigger, podemos indicar cuándo o cómo queremos que se dispare. Es por eso que existen los especificadores BEFORE, AFTER y INSTEAD OF.

Si queremos que nuestro trigger se ejecute luego de un INSERT, podemos especificarlo poniendo AFTER INSERT. De esta forma, primero se hará el INSERT, y luego se llamará a nuestro trigger automáticamente para que ejecute su código.

Nota: en SQL Server no existe BEFORE.

Sintaxis de Trigger

```
CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }
```

Requisitos para poder hacer una vista actualizable

- La vista debe hacerse referencia a una sola tabla.
- La definición de la vista no debe tener campos calculados o funciones de agregación
- No debe tener GROUP BY, HAVING, DISTINCT.
- No se debe, por medio de la actualización de la vista, incumplir alguna restricción de la tabla origen.
- Si es vista de vistas, todas deben ser actualizables.
- La vista debe contener la llave principal (a menos que se genere automáticamente como en una llave que es IDENTITY)

Hints de SQL Server

Los hints en SQL Server (y en la mayoría de las bases de datos relacionales del mercado), son agregados a un comando SQL que indican que debe ejecutarse de manera diferente a la predeterminada por el motor.

Existen 4 tipos de hints diferentes:

- Join Hints: Especifican que tipo de join (merge, hash, loop) vamos a usar en la query.
- Index Hints: Fuerzan el uso de uno o mas índices en la ejecución de la query.
- Lock Hints: Especifican un tipo de lockeo.
- Processing Hints: Especifican una estrategia particular en la ejecución de la query.

En situaciones normales, no se deberían utilizar, pero sin embargo, hay veces que es necesario. Por otra parte, en situaciones donde el volumen de información está en constante cambio, es importante recordar que un hint puede funcionar bien el día de hoy, pero no tan bien la semana siguiente. Es conveniente retestear todos las queries que utilizan hints periódicamente.

Join Hints

- **HASH JOIN:** Se utiliza cuando se quiere forzar el uso de algún método de HASH JOIN entre dos tablas.

  Ejemplo:

  ```
  SELECT title_id, pub_name, title FROM titles
  INNER HASH JOIN publishers ON titles.pub_id = publishers.pub_id
  ```

- **MERGE JOIN:** Se utiliza cuando se quiere forzar el uso del MERGE JOIN entre dos tablas.

  Ejemplo:

  ```
  SELECT title_id, pub_name, title FROM titles
  INNER MERGE JOIN publishers ON titles.pub_id = publishers.pub_id
  ```

- **LOOP JOIN:** Se utiliza cuando se quiere forzar el uso del LOOP JOIN entre dos tablas.

  Ejemplo:

  ```
  SELECT title_id, pub_name, title FROM titles
  INNER LOOP JOIN publishers ON titles.pub_id = publishers.pub_id
  ```

Generalmente no deberíamos usar estos hints, pero si por alguna razón el motor no esta generando el plan de ejecución mas optimo, entonces es valida esta metodología. Pero hay que tener CUIDADO en usarlo.

Index Hints

Este tipo de hints, se utiliza cuando queremos forzar el uso de un índice en particular, para optimizar la consulta. El plan de ejecución generado por el SQL Server suele ser el mas óptimo, pero en algunos casos excepcionales, no. Al igual que con los Join Hints, hay que tener cuidado a la hora de usarlos.

Ejemplo para forzar un Table Scan:

```
SELECT * FROM authors WITH (INDEX(0))
```

Poniendo como ID de INDEX, el valor 0, se fuerza un Table Scan.

Ejemplo para forzar el uso del Clustered Index:

```
SELECT * FROM authors WITH (INDEX(1))
```

Poniendo como ID de INDEX, el valor 1, se fuerza el uso del índice Clustered de la tabla. En caso de que no existe, tira error.

Ejemplo para forzar el uso de un Non Clustered Index:

```
SELECT * FROM authors WITH (INDEX(NOMBRE_DEL_INDICE))
```

Lock Hints

Por lejos, los mas usados. Este tipo de hints especifican que tipo de lockeo se debe efectuar en una operación. Existen varios hints de este tipo, pero el más usado es el NOLOCK y ROWLOCK.

- **NOLOCK (equivalente al READUNCOMMITTED):** Se usa en la sentencia SELECT. Indica al motor que ignore los a lockeos exclusivos de datos y lea directamente de la tabla, lo que suele llamarse "lectura sucia". Con esto ganamos mayor performance y escalabilidad, pero al riesgo de leer datos de una transacción que todavía no finalizo, lo que significa una perdida en la fiabilidad de los datos. Es un riesgo que tenemos que tener en cuenta. Por ejemplo, si queremos sacar un reporte de ventas entre dos periodos que ya terminaron, no tiene sentido realizar y verificar lockeos a la hora de leer, por lo cual un NOLOCK es totalmente valido. Pero si tenemos que leer datos entre periodos vigentes, donde pueden efectuarse transacciones, habría que evaluar el riesgo de leer datos que podrían llegar a ser inválidos.

  Algo importante que hay que aclarar, es que el NOLOCK no ignora TODOS los lockeos, de hecho, adquieren lockeos Sch-S (estabilidad del esquema). Por ejemplo, si se esta corriendo un comando DDL que afecte a la tabla, esta adquiere un lockeos Sch-M (modificación del esquema), y por lo tanto, si se ejecuta una consulta aun teniendo el hint NOLOCK, se bloqueara hasta que no termine la transacción anterior.

  En muchos sitios que requieren alta disponibilidad, se suele usar este hint en prácticamente todas las sentencias SELECT, salvo en aquellas donde se quiere garantizar la integridad de los datos.

  Ejemplo:

  ```
  SELECT COUNT(*) FROM Usuarios WITH (NOLOCK)
  INNER JOIN MenuUsuario WITH (NOLOCK) ON Usuarios.UsuarioID = MenuUsuario.UsuarioID
  ```

- **ROWLOCK:** Especifica que se apliquen bloqueos de fila cuando normalmente se aplicarían bloqueos de página o de tabla. Aplica solo a sentencias UPDATE, DELETE e INSERT. También, como en el caso del NOLOCK, este hint sirve para ganar mayor performance en entornos muy concurrentes. Algo que hay que tener cuidado acá, es que si ocurren muchos update sobre la misma tabla, se puede saturar el servidor de tantos lockeos por fila

  Ejemplo:

  ```
  UPDATE Usuarios WITH (ROWLOCK) SET UsuarioID = 20 WHERE UsuarioID = 1
  ```

Unidad 6: Procesamiento y optimización de consultas

Método de junta SQL Server

- **Nested Loops:** recorre primero una de las dos tablas (las más pequeña o la que no tenga índice) y para cada fila de esa tabla, busca en la otra tabla (mediante el índice, Index Seek) las filas que tienen igual valor en el atributo de junta, para combinarlas.

  Se utiliza generalmente cuando una tabla es muy pequeña y la otra es muy grande y tiene índice en el atributo de junta.

  Aplicable a joins por igualdad, comparación de mayor, menor y diferencia.

  El único método de junta que permite condiciones de desigualdad (mayor, menor o distinto) es el Nested Loops. Los otros métodos necesitan al menos una condición de igualdad (equijoin).

- **Merge Join:** También conocido como Sort Join o Sort-Merge Join.

  Este método se utiliza cuando ambas tablas se encuentran ordenadas físicamente por el atributo de junta (con índice cluster) o bien cuando solo una de ellas esta ordenada y la otra es relativamente pequeña y se puede ordenar rápidamente antes de hacer la junta (también se pueden ordenar ambas tablas antes de hacer la junta, pero en ese caso este método suele no ser tan conveniente).

  El Merge Join lee simultáneamente una fila de cada tabla y las compara por el atributo de junta. Si el valor coincide, esas filas son incluidas en el resultado. Si no coinciden, la fila con el menor valor es descartada, ya que como ambas tablas están ordenadas, esa fila no podrá combinarse con ninguna otra fila de las restantes.

  De esta forma se va avanzando y recorriendo ambas tablas desde la primer fila hasta la última, sin necesidad de retroceder. Como resultado, el costo total de aplicar este método es el correspondiente a leer una vez cada tabla (lo cual es muy eficiente y produce un costo muy bajo).

  Este método requiere al menos una condición de igualdad en las condiciones de junta (equijoin). Si tiene más de una condición de junta, resuelve la junta con la condición de igualdad y las otras condiciones las resuelve luego en memoria, por ejemplo: “T1.a = T2.a and T1.b > T2.b.”

- **Hash Match:** se usa generalmente cuando las tablas no están ordenadas ni tienen índices. Es decir, que este método se suele aplicar cuando no es posible utilizar ninguno de los dos métodos anteriores. Cuando SQL Server elije el método Hash join pueden ser una mala señal porque indica que probablemente algo se podría mejorar (por ejemplo, crear un índice).

  En este método consiste en:
  1. Construir una tabla hash en memoria con el contenido de la tabla menor.
  2. Recorrer la segunda tabla, comparando con el hash de memoria.

  Para que este método sea útil debe haber memoria suficiente para construir el hash.

El costo es el de leer una vez cada tabla (una para construir el hash y la otra para buscar coincidencias).

Al igual que el Merge, este método requiere al menos una condición de igualdad en las condiciones de junta (equijoin).

Si tiene más de una condición de junta, resuelve la junta con la condición de igualdad y las otras condiciones las resuelve luego en memoria, por ejemplo: “T1.a = T2.a and T1.b > T2.b.”

Table Spool: es un almacenamiento temporal de datos que van a ser leídos más de una vez. Es como una Tabla Temporal y sirve para mejorar la performance.

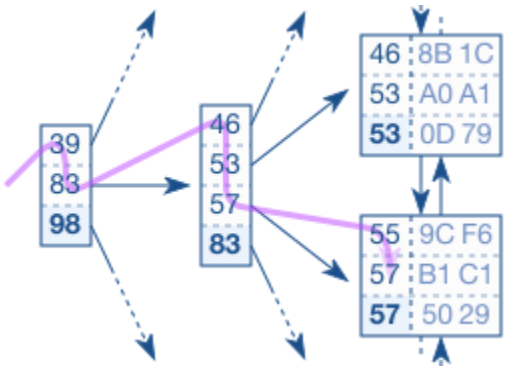
Índices

Un índice es una estructura diferente dentro de la base de datos; creado con el comando CREATE INDEX. Requiere su propio espacio en disco y contiene una copia de los datos de la tabla. Eso significa que un índice es una redundancia. Crear un índice no cambia los datos de la tabla; solamente establece una nueva estructura de datos que hace referencia a la tabla. De hecho, un índice de base de datos se parece mucho a un índice de un libro: ocupa su propio espacio, es redundante y hace referencia a la información actual almacenada en otro lugar.

El concepto clave es que todos los datos estén ordenados según un orden bien definido. Encontrar datos dentro de un conjunto de datos ordenados es rápido y fácil debido a que el orden de clasificación determina la posición de cada dato.

Generalmente los índices utilizan una estructura de árboles binarios y listas doblemente enlazadas para agilizar la búsqueda de los índices al momento de necesitarlos.

Una búsqueda por un índice requiere tres etapas: (1) el recorrido del árbol; (2) seguir la cadena de los nodos hojas; (3) devolver los datos de la tabla. El recorrido del árbol es la única etapa que tiene acceso a un número limitado de bloques, corresponde a la profundidad del árbol. Las otras dos etapas deberían tener acceso a muchos bloques que pueden ser la causa de la lentitud durante una búsqueda a través de un índice.



<u>Creación de un índice óptimo</u>

Para definir un índice óptimo, se debe saber no sólo cómo funcionan los índices; también se tiene que aprender cómo consultan las sentencias los datos de la aplicación. Eso significa que se han de conocer todas las combinaciones posibles de las columnas, cuyas aparecen como filtro en el where.

Esto más que nada es importante cuando tenemos índices concatenados/compuestos. No es lo mismo tener un índice (tipodni, dni) y hacer consultas que sean del estilo dni = X, que tener el índice (dni, tipodni) y realizar la misma consulta. En el primer caso, la consulta no va a poder hacer uso del índice para buscar el DNI más rápido.

Se dice que los índices deben definirse empezando por el campo con mayor variabilidad y terminando con el de menor variabilidad.

De todas formas, más que nada hay que fijarse en el tipo de consultas que ejecutaremos. Es el tipo de consulta el que nos dirá qué índice será más benéfico.

Sintaxis Index SQL Server

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
  ON <object>
```

Los índices UNIQUE son automáticamente creados en Primary Keys por el motor de BD.

Índices Clúster vs No Clúster

Una tabla o vista puede contener los siguientes tipos de índices:

- **Clúster:**
    - Los índices clúster ordenan y almacenan las filas de datos en la tabla o vista basándose en los valores de sus claves. Estas son las columnas que son incluidas en la definición del índice. Solamente puede haber un índice clúster por tabla, porque las filas solo pueden ser almacenadas en un único orden.
    - El único momento en que las filas de una tabla se encuentran almacenadas de forma ordenada es cuando la tabla contiene un índice clúster. Cuando una tabla posee uno de estos índices, recibe el nombre de tabla clúster. Si una tabla no posee un índice clúster, entonces sus filas se encuentran almacenadas en una estructura sin orden llamada “pila” o heap.
    - En SQL Server, por defecto, las claves primarias son índices clúster, a menos que se especifique lo contrario utilizando NONCLUSTERED en la definición de la clave. Esto puede llegar a ser un problema si no lo tenemos en cuenta, ya que la tabla quedará ordenada por la clave primaria, y si la clave primaria no es secuencial, cada vez que ingresemos un registro deberá reordenarse toda la tabla…
- **No clúster:**
    - Los índices no clúster tienen una estructura separada de las filas de la tabla. Esta estructura es una lista de punteros que apuntan directamente a las filas

físicas. El puntero recibe el nombre de “localizador de fila”.

- Una tabla puede tener varios índices no clúster, pero tener varios de ellos puede afectar al rendimiento.

Una analogía para entender estos índices es la siguiente: los índices clúster se asemejan a un diccionario. Un diccionario como tal no necesita un índice que indique en dónde se encuentra cada letra, porque ya está ordenado por letras de por sí.

En cambio, podemos imaginarnos al índice no clúster como el índice de un libro de Biología, el cual nos indicará en qué página se encuentra la sección de anatomía, o cualquier otro tema deseado. Sin embargo, el libro no se encuentra necesariamente ordenado por este índice.

Índices compuestos

Los índices compuestos trabajan de la misma forma que los índices normales, solo que están compuestos por varias claves/valores en vez de uno solo.

Por ejemplo, un índice compuesto por los atributos (a,b,c) será ordenado primero por a, luego por b, y luego por c.

```
| A | B | C |
-------------
| 1 | 2 | 3 |
| 1 | 4 | 2 |
| 1 | 4 | 4 |
| 2 | 3 | 5 |
| 2 | 4 | 4 |
| 2 | 4 | 5 |
```

Es de notar que, como los índices se guardan en un árbol binario, un índice de este tipo hará las búsquedas más eficientes cuando se busque (a) o (a,b), pero no en otras búsquedas como (b) o (b,c).

Acceso a tabla y índice

SQL Server tiene una terminología sencilla: las operaciones “Scan” leen el índice o la tabla entera mientras las operaciones “Seek” usan el B-tree o la dirección física (RID, como ROWID en Oracle) para tener acceso a una parte especifica del índice o de la tabla.

- **Index Seek, Clustered Index Seek:** Index Seek realiza un recorrido del B-tree y lee los nodos hoja para encontrar todas las entradas que coinciden.

- **Index Scan, Clustered Index Scan:** lee el índice entero (todas las filas) en el orden del índice. Dependiendo de varias estadísticas del sistema, la base de datos podría realizar esta operación si se necesitan todas las filas en el orden del índice, por ejemplo, debido a la cláusula del order by.

- **Key Lookup (Clustered):** recupera una sola fila desde una agrupación de índice. Es similar a la operación de Oracle INDEX UNIQUE SCAN por una tabla organizada según

índice (IOT).

- **RID Lookup (Heap):** recupera una sola fila desde una tabla, como la operación de Oracle TABLE ACCESS BY INDEX ROWID.

- **Table Scan:** es también conocido como un escaneo entero de la tabla. Lee la tabla entera (todas las filas y las columnas) tal y como se almacenan sobre el disco. Aunque las operaciones de lectura de multi-bloques pueden mejorar la velocidad de Table Scan, es todavía una de las operaciones más costosas. Además de la tasa alta de I/O, Table Scan debe también leer todas las filas de la tabla así que también se puede consumir una grande cantidad de tiempo CPU.

Índices e INSERT

Si bien los índices son buenos para agilizar las consultas SQL, tener muchos índices o un índice mal puesto puede afectarnos. Más que nada, los índices afectan a las operaciones de INSERT.

Cada vez que se inserta una nueva fila en la tabla, los índices que esa tabla posea deben actualizarse y agregar una nueva entrada que haga referencia a esa fila. Pero hay un problema, y es que los índices deben permanecer ordenados. Por lo que todos los índices que tenga esa tabla ahora deberán ser re-ordenados… Esto consume tiempo y hace que los tiempos de escritura sean mucho más lentos.

Ni hablar de la redundancia que estaríamos generando.

Índices y OR

Si por ejemplo tenemos una consulta así, donde tanto company como city son índices:

```
SELECT username FROM users WHERE company = 'bbc' or city = 'London';
```

Generalmente, el motor de base de datos utiliza solo un índice por tabla en una consulta SQL. Si en esta consulta utilizase el índice que está en compañía, de igual forma tendría que hacer un table-scan para hallar las filas donde la ciudad es London. Si utiliza el índice en ciudad, también tendría que hacer un table-scan para hallar las filas donde la compañía es bbc.

En definitiva, el motor elegiría por no usar ninguno de los índices y haría un table-scan directamente.

Índices y LIKE

Solo existe un caso en el que los índices son utilizados cuando se utiliza la sentencia LIKE:

- WHERE Column1 LIKE '%cde%' --can't use an index
- WHERE Column1 LIKE 'abc%' --can use an index
- WHERE Column1 Like '%defg' --can't use an index

Índices y NULL

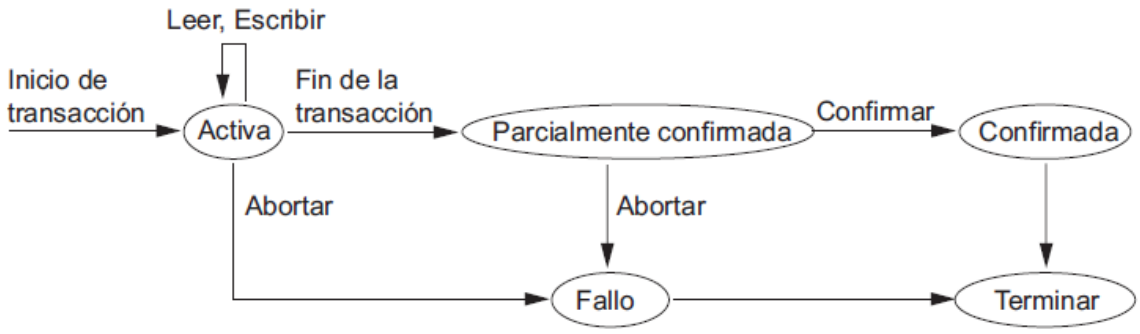
Por defecto, los índices no almacenan valores NULL. Por lo que no nos sirven en consultas que utilicen o busquen valores NULL.

Unidad 7: Transacciones y control de concurrencia

Una transacción en sí es una unidad lógica que posee una o un conjunto de operaciones atómicas. Todo conjunto de instrucciones SQL en una base de datos es ejecutado como una transacción. Las transacciones comienzan y terminan, **siempre terminan**.

Si una transacción no puede completarse, ya que ocurrió un fallo o hubo una excepción, los cambios realizados hasta el momento son deshechos. Las transacciones son todo o nada, o se ejecutan todos los cambios, o no se hace ninguno.

Estados de una transacción



El gestor de recuperación hace un seguimiento de las siguientes operaciones:

- **BEGIN TRANSACTION:** marca el inicio de la ejecución de una transacción
- **READ o WRITE:** especifican operaciones de lectura o escritura en los elementos de la base de datos que se ejecutan como parte de una transacción.
- **END TRANSACTION:** especifica que las operaciones READ y WRITE de la transacción han terminado y marca el final de la ejecución de la transacción. En este punto es necesario comprobar si los cambios pueden ser introducidos de forma permanente o si la transacción se suspendió por un error. Se dice que la transacción se encuentra **parcialmente confirmada**.
- **COMMIT TRANSACTION:** señala una finalización satisfactoria de la transacción. Los cambios pueden enviarse con seguridad a la base de datos y no se desharán.
- **ROLLBACK o ABORT:** señala que la transacción no ha terminado satisfactoriamente, por lo que deben deshacerse los cambios.

En el estado de confirmación parcial se pueden realizar chequeos y comprobaciones para asegurarse de que la transacción es correcta. Si pasa los chequeos, la transacción se confirma. Si no los pasa, es abortada y se genera un rollback.

Log de Transacciones

Cada base de datos de SQL Server tiene un log de transacciones que mantiene registro de todas las transacciones y las modificaciones que las mismas hicieron sobre la base de datos.

El log de transacciones un componente crítico de la base de datos. Si hay una falla en el sistema, el log de transacciones puede ayudar a devolver la base de datos a un estado consistente.

El log de transacciones está implementado como un archivo o un conjunto de archivos separados en la base de datos. Este archivo puede estar almacenado en disco, aunque una parte del mismo es llevada a memoria principal para agilizar la escritura o lectura.

Al ser un log, y al ser un archivo, el mismo puede crecer demasiado. Esto puede llevar a que consuma todo el espacio disponible en disco. Por esta misma razón, el archivo debe ser truncado con regularidad.

El truncamiento del archivo ocurre automáticamente luego de los siguientes eventos:

- Si se está usando el “simple recovery model”, siempre se trunca luego de un checkpoint.
- Si se está utilizando un “full recovery model” o un “bulk-logged recovery model”, si ocurrió un checkpoint desde el último backup, el truncamiento se hará cuando se realice un backup del log.

Además, el log de transacciones puede funcionar de tres formas diferentes:

- **Log circular:** un log de transacciones circular es un archivo con un tamaño máximo definido. Una vez que el archivo llega a su tamaño límite, las transacciones que ya han sido commiteadas son “pisadas” o sobre-escritas por transacciones nuevas, re-utilizándose el espacio.
- **Log secuencial:** los archivos secuenciales guardan todas las transacciones hasta quedarse sin espacio en disco para almacenar el archivo de log.
- **Log secuencial archivado:** los logs secuenciales archivados tienen un tamaño máximo. Al llegar a ese tamaño máximo, se cierra el archivo de log actual y se guarda. Luego se crea un nuevo archivo de log donde se empiezan a guardar las nuevas transacciones y el proceso vuelve a repetirse cuando llega a su tamaño límite.

Checkpoints

Un checkpoint marca el inicio del proceso que lleva los cambios de las transacciones al archivo de base de datos que se encuentra en disco. Es decir, refleja los cambios del log físicamente en disco. Se puede configurar el tiempo entre checkpoints.

Es de notar que el checkpoint solo vuelca las transacciones que fueron commiteadas. Aquellas que no fueron commiteadas son ignoradas.

Se debe elegir un tiempo entre checkpoints que sea “balanceado”. Un tiempo muy corto nos da la ventaja de recuperar rápido la base de datos en caso de una falla, pero estaríamos constantemente accediendo a disco (lo cual es muy costoso). Uno tiempo muy alto nos daría tiempo de recuperación alto, pero no accederíamos tanto a disco.

Simple Recovery Model

El modelo simple de recuperación es el modelo más básico de recuperación que hay en SQL Server. Todas las transacciones son escritas en el log, pero una vez que la transacción está completa y los datos fueron escritos al archivo de datos (fueron escritos en disco), el espacio que se estaba usando en el log de transacciones ahora está disponible para ser re-usado por nuevas transacciones (puede ser reescrito).

Debido a que las transacciones son re-escritas, no hay forma alguna de hacer una recuperación de un punto exacto en el tiempo. Por lo tanto, el punto de restauración más reciente será o un backup completo, o el último backup que se haya hecho.

Además, ya que las transacciones son re-escritas, el log de transacciones no crecerá por siempre, a diferencia de lo que sucede en el “full recovery model”.

Full Recovery Model

El modelo completo de recuperación le dice a SQL Server que guarde todos los datos relacionados a transacciones en log, hasta que ocurra un backup del log, o hasta que el log de transacciones sea truncado. Como todas las transacciones son guardadas, y como no se permite la re-escritura de las mismas, es posible hacer una restauración en cualquier punto del tiempo.

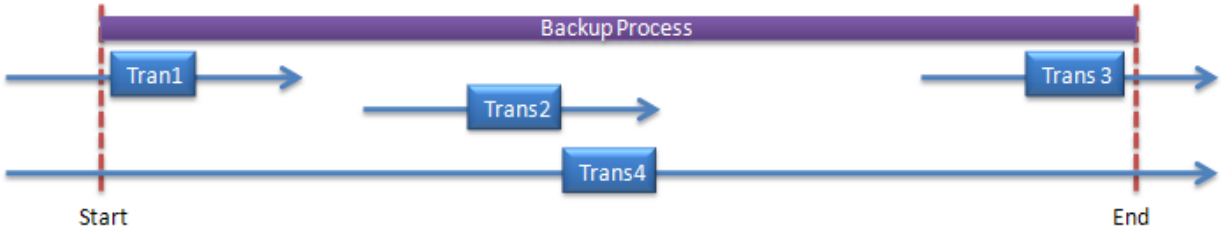
Backups en frío y en caliente

- **Cold:** Un backup en frío se hace cuando no hay actividad alguna de usuarios presente en el sistema. Suele llamarse un backup offline, ya que se realiza mientras la base de datos no está ejecutándose y no hay usuarios loggeados, ni transacciones activas. Si tenemos un log de transacciones circular, solo podemos hacer un backup en frío.

  Como no podemos tener usuarios ni sesiones activas, presenta una baja de servicio.

- **Hot:** Otra forma de hacer backups es el backup en caliente. Estos backups permiten hacer un respaldo de la base de datos mientras se encuentra en ejecución y recibiendo transacciones. Hace uso de los logs secuenciales archivados.

  Este backup, primero que nada, realiza un checkpoint del log de transacciones actual para que los cambios de dichas transacciones se vean reflejados en el archivo de datos



por las transacciones entrantes.

Una vez que termina de respaldar la base de datos entera (con todos sus bloques y elementos), el proceso de backup marca otro log de transacciones como el log final. Haciendo uso de estas dos marcas (log inicial y log final), el motor de base de datos puede aplicar los cambios de las transacciones en el archivo de backup, terminando así el proceso de backup en caliente.

La ventaja es que no tenemos baja de servicio en lo que dure el backup.

Transacción ideal

Se dice que una transacción es ideal si cumple con las propiedades ACID:

- **Atomicidad:** la transacción es atómica de principio a fin.
- **Consistencia:** la transacción parte de un estado consistente de la base de datos, y termina dando como resultado otro estado consistente. Con estado consistente nos referimos a un estado que no viola ninguna de las restricciones de la BD (por ejemplo, claves duplicadas).
- **Isolation (Aislamiento):** las transacciones están aisladas una con otras. Es decir que son independientes en su ejecución.
- **Durabilidad:** las transacciones no deben perderse en el tiempo. No podemos perder rastro de una transacción y sus cambios. Deben estar registradas.

Las propiedades de consistencia y aislamiento pueden ser flexibilizadas un poco en situaciones especiales.

Por ejemplo, podemos realizar una o varias transacciones que nos dejen la BD en un estado inconsistente (claves duplicadas, claves foráneas inexistentes), ya que quizás estamos recibiendo por partes un lote de datos desorganizado. Luego podemos hacer un chequeo de la integridad y verificar que no se violen las restricciones.

Planificaciones de transacciones

Todo motor de base de datos posee una planificación de las transacciones, que no es más que el orden en el cual se van a ejecutar las operaciones de dichas transacciones. El motor de base de datos debe elegir una planificación que evite **conflictos** entre las transacciones, y que maximice la concurrencia de las mismas.

Se dice que dos operaciones de una planificación entran en conflicto si satisfacen estas condiciones:

1. Pertenecen a transacciones diferentes.
2. Acceden al mismo elemento X.
3. Al menos una de las operaciones es write_item(x).

Planificaciones en serie, no serie, y serializables por conflicto

En una planificación en serie, las transacciones se ejecutan enteras y en serie. Por ejemplo, primero T1, y después T2. Es decir, todas las operaciones de T1 se ejecutan consecutivamente en la planificación, y luego se ejecutan todas las de T2. En caso contrario, se dice que la planificación es no serie.

Las planificaciones en serie siempre generan estados correctos en la base de datos, ya que no hay lugar a conflictos (no hay interpolación/solapamiento de transacciones).

El problema con las planificaciones en serie es que limitan la concurrencia o la interpolación de operaciones. La transacción T2 debe esperar hasta que T1 se termine de ejecutar completamente para recién iniciar su ejecución, lo cual ralentiza mucho las cosas.

Decir que una planificación no serie S es serializable es equivalente a decir que es correcta, porque es equivalente a una planificación en serie (que son siempre correctas).

Hay dos formas de equivalencia de una planificación:

- **Equivalentes por resultado:** dos planificaciones son de **resultado equivalente** si producen el mismo estado final de la base de datos. Sin embargo, dos planificaciones diferentes pueden producir accidentalmente el mismo estado final. Por lo tanto, la equivalencia de resultado por sí sola no puede utilizarse para definir la equivalencia de planificaciones.

- **Equivalentes por conflicto:** dos planificaciones son equivalentes por conflicto si el orden de cualquier par de operaciones en conflicto es el mismo en las dos planificaciones.

Utilizando el concepto de equivalencia por conflicto, decimos que una planificación S es serializable por conflicto si es equivalente (por conflicto) con alguna planificación en serie S'. En tal caso, podemos reordenar las operaciones no conflictivas de S para formar la planificación serie equivalente S'.

Algoritmo de comprobación de serialización por conflicto

1. Por cada transacción Ti participante en la planificación S, crear un nodo etiquetado como Ti en el gráfico de precedencia.
2. Por cada caso de S donde Tj ejecute una operación read_item(x) después de que Ti ejecute write_item(x), crear un arco (Ti → Tj) en el gráfico de precedencia.
3. Por cada caso de S donde Tj ejecute una operación write_item(x) después de que Ti ejecute read_item(x), crear un arco (Ti → Tj).
4. Por cada caso de S donde Tj ejecute una operación write_item(x) después de que Ti ejecute write_item(x), crear un arco (Ti → Tj).
5. La planificación es serializable sí y solo sí el gráfico de precedencia no tiene ciclos.

Si no hay ningún ciclo en el gráfico de precedencia, podemos crear una planificación en serie equivalente a S, ordenando de tal modo las transacciones que participan en S: siempre que exista un arco en el gráfico de precedencia de Ti a Tj, Ti debe aparecer antes que Tj en la planificación en serie equivalente.

Usos de la serialización

Una planificación serializable proporciona los beneficios de la ejecución concurrente sin ninguna corrección. En la práctica, es muy difícil probar la serialización de una transacción. Por tanto, la metodología que se toma en la mayoría de los sistemas prácticos es determinar los métodos que garantizan la serialización, sin tener que probar las propias planificaciones.

La metodología tomada en la mayoría de los DBMS comerciales es diseñar protocolos que garantizaran la serialización de todas las planificaciones en las que las transacciones participan.

Nivel de aislamiento en SQL

Una transacción tiene ciertas características atribuibles a ella, que se especifican en SQL con una sentencia SET TRANSACTION. Una de dichas características es el nivel de aislamiento.

El motor de base de datos maneja un nivel de aislamiento por defecto para todas las transacciones. No se recomienda utilizar nunca el nivel que esté seteado por defecto. Siempre tenemos que utilizar el nivel que nosotros creamos necesario para nuestra base de datos, y en todo caso, modificar el valor por defecto, o escribir las transacciones con un nivel de aislamiento propio.

Cada nivel de aislamiento provee ventajas y desventajas. Los niveles menos estrictos pueden incurrir en una o más violaciones (haciendo que los datos en nuestra BD pierdan confianza), pero poseen un gran nivel de concurrencia entre transacciones. Los niveles más estrictos, a costa de la concurrencia, producen menos violaciones (y por lo tanto los datos de la BD son más confiables).

La opción de nivel de aislamiento se especifica con la sentencia ISOLATION LEVEL <aislamiento>, donde el valor para <aislamiento> puede ser READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ o SERIALIZABLE. El nivel de aislamiento predeterminado es SERIALIZABLE, aunque algunos sistemas utilizan READ COMMITTED como su valor predeterminado. El uso del término SERIALIZABLE aquí está basado en no permitir violaciones que provoquen lecturas sucias, irrepetibles y fantasmas. Si una transacción se ejecuta a un nivel de aislamiento más bajo que SERIALIZABLE, entonces se pueden producir una o más de estas violaciones.

1. **Lectura sucia:** Una transacción T1 puede leer la actualización de una transacción T2, que todavía no se ha confirmado. Si T2 falla y es cancelada, entonces T1 habría leído un valor que no existe y es incorrecto.

2. **Lectura irrepetible:** Una transacción T1 puede leer un valor dado de una tabla. Si otra transacción T2 actualiza más tarde ese valor y T1 lee de nuevo el valor, T1 verá un valor diferente.

3. **Fantasmas:** Una transacción T1 puede leer un conjunto de filas de una tabla, quizá basándose en alguna condición especificada en la cláusula WHERE de SQL. Ahora, suponga que una transacción T2 inserta una fila nueva que también satisface la condición de la cláusula WHERE utilizada en T1, en la tabla utilizada por T1. Si T1 se repite, entonces T1 verá un fantasma, una fila que anteriormente no existía.

Tabla 17.1. Posibles violaciones según el nivel de aislamiento.

	Tipo de violación		
Nivel de aislamiento	**Lectura sucia**	**Lectura irrepetible**	**Fantasma**
READ UNCOMMITTED	Sí	Sí	Sí
READ COMMITTED	No	Sí	Sí
REPEATABLE READ	No	No	Sí
SERIALIZABLE	No	No	No

Aunque parezca que sí, el nivel de aislamiento no funciona por arte de magia. Al setear un nivel de aislamiento, le estamos indicando al motor de base de datos que utilice ciertos tipos de bloqueos. Por ejemplo, READ UNCOMMITED ni siquiera utiliza Shared Locks para hacer lecturas. En cambio, un nivel más estricto como READ COMMITTED sí los utiliza, y por eso no caemos en la violación de lecturas sucias.

Tipos de Bloqueos:
La selección de un nivel de aislamiento determina el tipo y el alcance de los bloqueos que va a adquirir en cada sentencia para proteger las modificaciones de datos. Estos bloqueos se mantienen hasta que se completa la transacción.

	Read uncommitted	Read committed	Repeatable read	Serializable
SELECT	No genera ningún bloqueo sobre los datos consultados e ignora los bloqueos de otras	No genera ningún bloqueo sobre los datos consultados.	**Alcance:** solo las filas del resultado. **Tipo de Bloqueo:** Compartido (de lectura) Se bloquean solo las filas	**Alcance:** toda la tabla. **Tipo de Bloqueo:** Compartido (de lectura) Se bloquea toda la tabla consultada.

Si bien esta tabla es muy linda y todo, no es 100% precisa. No siempre se bloquea TODA la tabla. Eso depende de la consulta y de la cantidad de rows afectadas. Por ejemplo, un UPDATE de una sola row no va a bloquear toda la tabla… Pero es posible que un UPDATE de varias rows se consolide en un table block.

Bloqueos

Los bloqueos (o lockeos) son técnicas que se utilizan para controlar la ejecución concurrente de transacciones. Un bloqueo es una variable asociada a un elemento de datos que describe el estado de ese elemento respecto a las posibles operaciones que se le puedan aplicar.

Generalmente, hay un bloqueo por cada elemento de datos de la base de datos. Los bloqueos se utilizan como medio para sincronizar el acceso de las transacciones concurrentes a los elementos de la base de datos.

Tipos de bloqueos

- **Bloqueos binarios:** un bloqueo binario puede tener solo dos estados: bloqueado y desbloqueado. Si es valor de bloqueo sobre X es *bloqueado* nadie podrá acceder a dicho elemento. Si es *desbloqueado* es posible acceder al elemento solicitado.

  Los bloqueos binarios son una forma de exclusión mutua. Las operaciones de bloquear() y desbloquear() deben ser atómicas.

  En la práctica, los bloqueos binarios no son utilizados, ya que son demasiado restrictivos. Como máximo, solo una transacción puede tener un bloqueo sobre un elemento dado. Esto limita muchísimo la concurrencia.

- **Bloqueos compartidos/exclusivos:** los bloqueos compartidos/exclusivos actúan de forma diferente a los binarios para evitar dañar la concurrencia de las transacciones.

  En un bloqueo compartido/exclusivo, varias transacciones pueden tener acceso al elemento X si todas ellas acceden para leer. Sin embargo, si una transacción va a escribir un elemento X, debe tener acceso exclusivo a X.

  Hay tres operaciones de bloqueo: bloquear_lectuar(X), bloquear_escritura(X) y desbloquear(X). Lo que permite que los elementos posean tres posibles estados: bloqueados para lectura, bloqueado para escritura, o desbloqueado.

  Un elemento bloqueado para lectura también se denomina **lectura compartida**, porque otras transacciones pueden leer el elemento, mientras que un elemento bloqueado para escritura se denomina **escritura exclusiva**, porque solo una transacción posee en exclusiva el bloqueo de un elemento.

  Cuando utilizamos el esquema de bloqueo compartido/exclusivo, el sistema debe implementar las siguientes reglas:

  1. Una transacción T debe emitir la operación bloquear_lectura(X) o bloquear_escritura(X) antes de que se ejecute cualquier operación leer_elemento(X) de T.

  2. Una transacción T debe emitir la operación bloquear_escritura(X) antes de que se ejecute cualquier operación escribir_elemento(X) de T.

  3. Una transacción T debe emitir la operación desbloquear(X) una vez que se hayan completado todas las operaciones leer_elemento(X) y escribir_elemento(X) de T.

  4. Una transacción T no emitirá una operación bloquear_lectura(X) si ya posee un bloqueo de lectura (compartido) o de escritura (exclusivo) para el elemento X. Esta regla

se puede hacer menos estricta.

5. Una transacción T no emitirá una operación bloquear_escritura(X) si ya posee un bloqueo de lectura (compartido) o de escritura (exclusivo) para el elemento X. Esta regla se puede hacer menos estricta.

6. Una transacción T no emitirá una operación desbloquear(X) a menos que ya posea un bloqueo de lectura (compartido) o de escritura (exclusivo) sobre el elemento X.

Conversión/promoción de bloqueos

Las reglas 4 y 5 son un poco estrictas. Podemos, bajo ciertas condiciones, convertir el bloqueo de un estado a esto. Por ejemplo, si T es la única transacción que posee un bloqueo de lectura sobre X en el momento de emitir la operación bloquear_escritura(X), el bloqueo puede promocionarse (convertirse a uno exclusivo); en caso contrario, la transacción debe esperar. También es posible para una transacción T emitir una operación bloquear_escritura(x) y más tarde degradar el bloqueo emitiendo una operación bloquear_lectura(x).

Garantía de la serialización por el bloqueo en dos fases

Los bloqueos (de cualquier tipo, binarios, o compartidos/exclusivos) no garantizan por sí solos la serialización de las planificaciones, por lo que se puede caer nuevamente en los problemas de concurrencia.

Se dice que una transacción obedece el protocolo de bloqueo en dos fases si todas las operaciones de bloqueo preceden (están antes de) la primera operación de desbloqueo de la transacción.

Es así como una transacción puede dividirse en dos fases: una **primera fase de expansión**, durante la cual pueden adquirirse bloqueos nuevos sobre los elementos, pero no pueden liberarse; y una **segunda fase de reducción**, en la que los bloqueos existentes se pueden liberar, pero no se pueden adquirir bloqueos nuevos.

La promoción de bloqueos (de bloqueado para lectura a bloqueado para escritura) debe realizarse durante la fase de expansión, y la degradación de bloqueos debe realizarse en la segunda fase.

Si cada transacción de una planificación obedece el protocolo de bloqueo en dos fases, la planificación es serializable, obviando la necesidad de comprobar la serialización de las planificaciones. Es el mismo mecanismo de bloqueo, a través de sus reglas, que implementa la serialización.

El bloqueo en dos fases puede limitar la concurrencia que puede darse en una planificación. Este es el precio por garantizar la serialización de todas las planificaciones sin tener que comprobar las propias planificaciones.

Bloqueo en dos fases básico, conservador, estricto, y riguroso

- **2PL Básico:** es el que acabás de leer más arriba.

- **2PL Conservador:** requiere una transacción para bloquear todos los elementos a los que tendrá acceso antes de comenzar a ejecutarse, mediante la declaración previa de los conjuntos de lectura y escritura. Si no es posible bloquear cualquiera de los elementos predeclarados necesarios, la transacción no bloqueará ningún elemento; en cambio, esperará a que puedan bloquearse todos los elementos. Este protocolo no tiene deadlocks, pero es difícil de utilizar en la práctica debido a la necesidad de predeclarar los conjuntos de lectura y escritura, algo que no siempre es posible.

- **2PL Estricto:** una transacción T no libera ninguno de sus bloqueos exclusivos hasta después de confirmarse o abortar. Por tanto, ninguna otra transacción puede leer o escribir un elemento que es escrito por T a menos que esta se confirme. 2PL estricto no está libre de deadlocks.

- **2PL Riguroso:** una transacción T no libera ninguno de sus bloqueos (exclusivos o compartidos) hasta después de su confirmación o cancelación.

La diferencia entre conservador y riguroso es que el primero debe bloquear todos sus elementos antes de iniciarse, de modo que una vez que la transacción empieza se encuentra en la segunda fase. El riguroso no desbloquea ninguno de sus elementos hasta después de terminar (por confirmación o cancelación), de modo que la transacción se encuentra en la primera fase hasta que termina.

Nota: en la cátedra “usamos” el bloqueo estricto, ya que es el usado por la mayoría de los motores de base de datos.

<u>Granularidad de los bloqueos</u>

El tamaño de los elementos de datos se denomina con frecuencia granularidad del elemento de datos. La granularidad fina se refiere a los tamaños de elemento pequeños, mientras que la granularidad gruesa se refiere a los tamaños de elementos más grandes.

Cuanto mayor es es el tamaño del elemento de datos, más bajo es el grado de concurrencia permitido.

Por otro lado, cuanto más pequeño es el tamaño del elemento de datos, mayor número de elementos hay en la base de datos. Como cada elemento está asociado con un bloqueo, el sistema tendrá una mayor cantidad de elementos activos que el gestor de bloqueos deberá controlar. Se realizarán más operaciones de bloqueo y desbloqueo, lo que provocará una sobrecarga más alta.

Además, el sistema posee una lista enlazada en memoria (locklist) que guarda información sobre todos los bloqueos. Si nuestra granularidad es muy pequeña, esa lista podría llenarse, y nos quedaríamos sin bloqueos (en caso de que una transacción requiera nuevos bloqueos, no podrá pedirlos, y deberá abortarse).

Si una transacción accede a pocos registros, lo mejor es tener una granularidad equivalente a un registro. Por otro lado, si una transacción accede normalmente a muchos registros del mismo fichero, lo mejor es tener una granularidad equivalente a un bloque o a un fichero, de

modo que la transacción considerará todos los registros como uno (o unos cuantos) elementos de datos.

¿Qué se puede bloquear?

- **Database:** bloquear toda la base de datos.
- **Table:** bloquear toda una tabla entera.
- **Extend / Pages / Blocks:** bloquear varias páginas o conjunto de bloques.
- **Page / Block:** bloquear solo un bloque.
- **Key:** bloqueo a nivel de clave primaria / índice. Bloquea entradas de un índice.
- **Row/s:** bloqueo a nivel de filas / registros.

Lock-Scalation en la granularidad

Existe también algo llamado **lock-scalation**. El escalamiento de bloqueos es algo que realiza el motor de base de datos cuando nota que una transacción posee demasiados bloqueos. En vez de dejar que todos esos bloqueos consuman recursos, consolida todos esos un bloqueo de mayor nivel, ahorrando así los recursos.

Por ejemplo, si una transacción bloquea casi todos los registros de una tabla, el motor de base de datos puede decir convertir ese bloqueo de registros a un bloqueo de tabla entera. Se sube el nivel de bloqueo, haciendo que la granularidad sea más grande, y ahorrándose recursos en el proceso.

Lock-Wait

A ninguna transacción se le puede otorgar un lock que haga conflicto con el modo de un lock que ya ha sido otorgado en otra transacción para esa misma data (tabla, fila, etc). Si una transacción pide un modo de lockeo que entre en conflicto con el lock que ya ha sido ortogado para la misma data, la instancia de SQL Server pausará la transacción que hace la petición hasta que el primer lock sea liberado.

La duración de esta pausa puede ser configurada. Por defecto es infinita.

Key-Range Locking

Es un bloqueo de un rango de claves usado SOLO en transacciones serializables.

Los Key-range locks protegen un rango de filas implícitamente incluidas en una tabla que está siendo leída por una transacción mientras se usa el nivel de aislamiento serializable.

El nivel de aislamiento serializable requiere que cada consulta ejecutada durante una transacción debe obtener el mismo conjunto de filas cada vez que es ejecutada en la transacción. Un key-range lock protege este requerimiento al prevenir que otras transacciones inserten nuevas filas cuyas claves entrasen en el rango de claves que está siendo leído por la transacción serializable.

De esta forma se prevén las phantom reads. Al proteger los rangos de claves entre las filas, también prevé inserciones fantasmas en un conjunto de registros que está siendo accedido por una transacción.

Los locks de este tipo se aplican a un índice, especificando un valor de clave de inicio y de fin. Este lock bloquea cualquier intento para insertar, actualizar, o borrar cualquier fila con un valor de clave que caiga dentro de ese rango, porque esas operaciones primero debería obtener un lock en el index (y no podrá obtenerlo ya que lo tiene la transacción que pidió el key-range lock).

Deadlocks

Ocurren cuando dos transacciones quedan bloqueadas porque T1 necesita un recurso que tiene T2, y viceversa.

Muchas veces los deadlocks son causados por cómo están programadas las transacciones, y su solución es tan fácil como cambiar o respetar el mismo orden de bloqueo en todas las transacciones.

También un nivel de aislamiento muy estricto puede causar deadlocks.

Existen protocolos para prevenir el deadlock, así como formas de detectarlo, y algoritmos para solucionarlo (donde solucionarlo significa matar a una transacción víctima), pero esto no es sistemas operativos y no vamos a ahondar tanto en eso.

Deadlocks con un solo recurso

https://docs.actian.com/ingres/10.2/index.html#page/DatabaseAdmin/Different_Access_Paths_as_a_Source_of_Deadlock.htm

Cabe destacar que, aunque sea extraño, es posible causar un deadlock con un solo recurso, independientemente del nivel de aislamiento.

El ejemplo siguiente ilustra un deadlock que ocurre al utilizar una misma tabla.

T1	T2
Begin transaction	Begin transaction
Insert Into Tabla Values (1, ‘asd’, 2)	
	Insert Into Tabla Values (2, ‘AAA’, 3)
Update Tabla Set campo2 = ‘asd Where id = 2	
	Update Tabla Set campo2 = ‘AAA Where id = 1
Commit transaction	Commit transaction

En este caso, las filas nuevas que son ingresadas son bloqueadas de forma exclusiva: no pueden ser leídas, ni modificadas. El update de T1 esperará a que se libere el bloqueo exclusivo (que sucederá en el commit de T2). Lo mismo sucederá con el update de T2: va a esperar al commit de T1. Entonces se genera un deadlock sobre la misma tabla.

Otro ejemplo:

T1	T2
Begin transaction	Begin transaction
Insert Into Tabla Values (1, ‘asd’, 2)	
	Insert Into Tabla Values (2, ‘AAA’, 3)
Insert Into Tabla Values (2, ‘AAA’, 3)	
	Insert Into Tabla Values (1, ‘asd’, 2)
Commit transaction	Commit transaction

El problema que surge es el siguiente: INSERT bloquea toda la tabla/filas y no permite que la misma sean consultadas ni modificadas por otras transacciones, pero sí permite que se puedan añadir nuevas filas.

Lo que sucede es que se presenta una violación de clave primaria, ya que estamos ingresando el mismo valor de clave en la tabla dos veces.

Todavía estoy tratando de entender si es plausible este caso en MS SQL Server. Aparentemente sí lo es en MySQL, pero…

Unidad 8: Seguridad

Mecanismos de control de acceso

La mayoría de los DBMS aplican estos tres tipos de controles de acceso:

- **DAC:** control de acceso discrecional, cada objeto tiene un dueño y el dueño es el encargado de otorgar permisos sobre el objeto. El DAC se basa en la identificación de un usuario mediante sus credenciales en el momento de autenticación (es decir, utiliza usuarios). Es discrecional porque el dueño puede, a discreción, indicar quién recibe permisos y quién no.

- **RBAC:** la idea básica es que los permisos están asociados a roles y a los usuarios se les asignan los roles apropiados. Los roles se pueden crear mediante los comandos CREATE ROLE y DESTROY ROLE. Los comandos GRAN t REVOKE puede ser usados para conceder y revocar privilegios a los roles.

- **MAC:** control de acceso mandatorio, se utiliza para reformar la seguridad a varios niveles mediante la clasificación de los datos y de los usuarios en varias clases de seguridad (o niveles) para después implementar la política de seguridad adecuada a la organización. Por ejemplo, una política de seguridad habitual es permitir a los usuarios de un determinado nivel de clasificación ver solo los elementos de datos clasificados en el mismo (o menor) nivel de clasificación que el del usuario. Esto es en contraste al DAC, ya que los usuarios no definen las políticas de seguridad, ni pueden otorgar o revocar permisos.

Seguridad desde el punto de vista del DBA

El administrador de la base de datos (DBA) es quien debe determinar que la información almacenada es la que debería ser y quien debe impedir que los ataques maliciosos modifiquen u obtengan información de la base de datos que no debería ser accesible. Para lograr esto, el DBA debe:

- Manejar cuentas de usuario
- Asignar privilegios a esas cuentas
- Definir niveles de seguridad

Con estas tres acciones el DBA maneja las diferentes escalas de acceso al conjunto de datos.

Para realizar estas tareas, se definió un subconjunto de instrucciones ANSI-SQL que se dio en llamar Lenguaje de Control de Datos (DCL).

Administrar cuentas de usuario

El DBA puede crear, inhabilitar o borrar un usuario.

De forma genérica, para acceder a una base de datos tenemos al usuario y al grupo al que pertenece. Los permisos pueden asignarse a este usuario y/o al grupo al que pertenece.

Algunos Motores de bases de datos, dividen al usuario como Usuario de Acceso y usuario de Base de datos, en lugar de usuario y grupo. Pudiendo además llamarse ambos de

distinta manera.

Esto ocurre, por ejemplo en MSSQL Server, cuando para crear un usuario debemos hacer lo siguiente:

```
CREATE LOGIN usuario WITH PASSWORD = 'clave'
CREATE USER mi_usuario FOR LOGIN usuario
```

Donde el “Login” es el usuario de acceso. Luego el “User” será la identidad dentro de la base de datos.

Para borrar usuarios, siempre podemos hacerlo mediante la instrucción DROP.

Entonces, podemos definir accesos o denegarlos para un usuario o para un grupo, aplicándolo a todos los que pertenecen a este grupo.

Privilegios de la base de datos

Cada uno de los usuarios en la base de datos tiene privilegios y solo pueden operar con esos datos cuando estén autorizados para hacerlo.

Los permisos pueden darse según:

- El tipo de Acción
  - Select
  - Update
  - Delete
  - Insert

- El objeto usado
  - Toda la tabla
  - Algún campo

Asignando y revocando permisos

Para trabajar con permisos, se definió un lenguaje, conocido como lenguaje de control de datos (**DCL**). Este lenguaje tiene tres palabras reservadas:

- **GRANT:** es la que nos permite asignar permisos a un usuario o a un grupo de usuarios.
- **REVOKE:** es aquella que nos sirve para eliminar un permiso dado. Puede eliminar tanto un GRANT como un DENY.
- **DENY:** sirve para denegar permisos sobre un objeto de la base de datos. Cabe destacar, que la instrucción DENY no forma parte del estándar, lo cual hace que solo se incluya en algunos motores de base de datos.

Sintaxis GRANT

```
GRANT [ All | Priv ] ON <objeto> TO <USER | ROLE> [ WITH GRANT OPTION ]
```

WITH GRANT OPTION especifica que el usuario o rol que reciba dicho privilegio, tendrá la capacidad de otorgar ese mismo privilegio a otros usuarios.

Sintaxis REVOKE

REVOKE [PRIV] ON <objeto> FROM <[USUARIO|ROLE]>

Sintaxis DENY

DENY [PRIV] ON <objeto> TO <[USUARIO|ROLE]>

Jerarquía de privilegios

Los privilegios poseen una jerarquía. Es decir que hay diferentes niveles de privilegios.

Un DENY puede anular un GRANT si se encuentra en un nivel mayor o igual al del GRANT.

Por ejemplo, si denegamos el privilegio de SELECT para un usuario en una base de datos entera, incluso aunque el usuario tenga GRANT en varias tablas de esa base de datos, no podrá realizar ningún SELECT porque el DENY tiene mayor nivel.

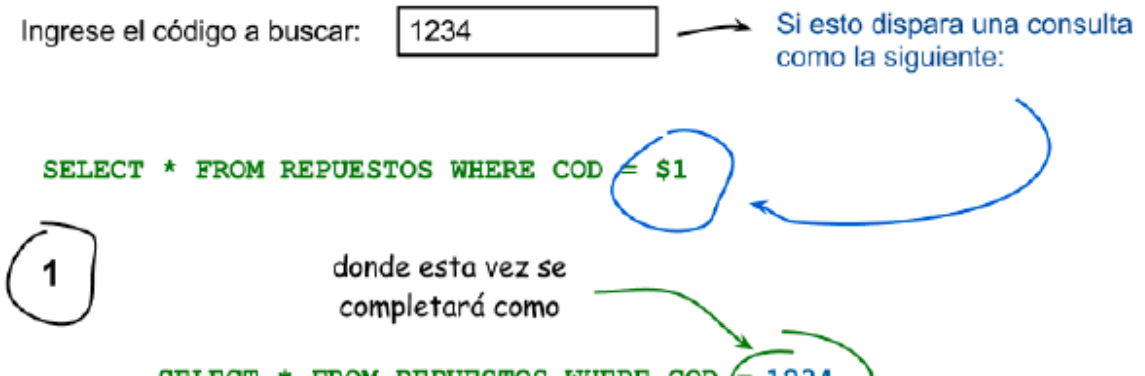
Seguridad a través de vistas

Las vistas son una buena herramienta para la seguridad discrecional. Por ejemplo, podemos utilizar vistas para limitar los campos que un usuario puede ver de una tabla. También podemos crear vistas actualizables, y darles a un usuario permisos sobre esa vista, limitando así su poder de actualización sobre una tabla.

SQL Injection

El método de ejecución de código SQL llamado SQL injection, refiere a la ejecución de código no deseado en base a la explotación de una vulnerabilidad en un sistema informático.

Esto ocurre cuando el contenido de controles, tanto en sistemas escritorio o web, utilizan el contenido generado por el usuario dentro de un control para realizar consultas a la base de datos.



Como vemos, en el sistema, se tiene un cuadro de texto (o textbox) donde se pide ingresar un valor numérico. Este valor numérico se incluye en un texto que se ejecutará en la base de datos para obtener resultados.

Si en lugar de ponerse el valor buscado, se utiliza dicho cuadro de texto para cualquier otro fin, se puede incurrir en graves errores de seguridad, como se plantea en el ejemplo.

Hay muchas formas de evitar esto. Algunas de ellas son:

- Validar todo el contenido de los cuadros de texto o lugares donde el usuario realice un ingreso de texto.

- Parametrizar los ingresos de consultas en SQL.

- Filtrado del ingreso, comprobando tipos de datos y utilizando transacciones.