Tema 6. Transacciones y seguridad

Las aplicaciones de bases de datos a gran escala, con bases de datos de gran tamaño y con cientos de usuarios concurrentes, como los sistemas de reservas, los bancos, la bolsa, etc., requieren una alta disponibilidad y un tiempo de respuesta mínimo para cada uno de los usuarios concurrentes que estén utilizando el sistema. En este tema se ofrece una introducción a las *transacciones* como base del procesamiento de peticiones concurrentes en bases de datos, estudiando los conceptos básicos relacionados con el procesamiento de transacciones, analizando el problema del control de la concurrencia que se produce cuando las transacciones de distintos usuarios interfieren entre sí. Asimismo, se realizará una breve introducción a la seguridad, describiendo cómo se puede establecer con SQL un mecanismo de autorización basado en la concesión y retirada de privilegios sobre los objetos de la base de datos.

6.1. Introducción al procesamiento de transacciones

Las aplicaciones de bases de datos suelen ser sistemas a los que acceden concurrentemente varios usuarios por lo que es posible que varios usuarios estén intentando acceder simultáneamente a los mismos datos. Esta situación es especialmente delicada cuando alguno o varios de estos usuarios está intentando modificar los datos que están siendo utilizados por los otros usuarios, de forma que hay que controlar la gestión y el acceso a los datos en este tipo de situaciones, y esto se consigue mediante el uso de *transacciones*.

Una transacción es una unidad lógica de procesamiento que incluye una o más operaciones de acceso a la base de datos (modificación o consulta). Las transacciones se definen especificando sus límites de principio y de fin, y en SQL se denotan con **begin transaction** y **end transaction**.

6.1.1. Modelo de base de datos para el procesamiento de transacciones

Para explicar el procesamiento de transacciones utilizaremos el siguiente modelo de base de datos. Una base de datos está formada por un conjunto de elementos de datos con nombre. El tamaño de un elemento de datos se denomina *granularidad*, y puede ser desde un campo de un registro, hasta un bloque de disco o una o varias tablas completas. Utilizando este modelo simplificado, las operaciones básicas de acceso a la base de datos que puede incluir una transacción son:

- `leer_elemento(X)`: Lee el elemento de la base de datos llamado X.

- `escribir_elemento(X)`: Escribe el elemento de la base de datos llamado X.

Así, una transacción incluye operaciones `leer_elemento` y `escribir_elemento` para tener acceso y modificar la base de datos. La Figura 6.1 ilustra dos transacciones muy sencillas

T_1	T_2
leer_elemento(X) **X=X-N** **escribir_elemento(X)** **leer_elemento(Y)** **Y=Y+N** **escribir_elemento(Y)**	**leer_elemento(X)** **X=X+M** **escribir_elemento(X)**

Figura 6.1. Ejemplo de dos transacciones

Definimos como *conjunto lectura de una transacción* al conjunto de elementos de datos que son leídos de la base de datos en una transacción. Así, en T_1, el conjunto lectura es {X, Y} y el de T_2 es {X}. Definimos como *conjunto escritura de una transacción* al conjunto de elementos de datos que son escritos por una transacción. En el ejemplo, en T_1, el conjunto de escritura es {X,Y} y en T_2 es {X}.

6.1.2. Necesidad del control de la concurrencia

Dado que varios usuarios pueden lanzar operaciones concurrentes contra una base de datos, es necesario establecer mecanismos para que las transacciones no se ejecuten de forma incontrolada y dejen a la base de datos en un estado inconsistente. Para ilustrarlo supongamos que tenemos una base de datos de cuentas corrientes. Como elemento de datos tomamos la cuenta corriente, que entre otra información, cuenta con su saldo. La Figura 6.1 ilustra una transacción T_1 que transfiere de una cuenta X a una cuenta Y N unidades monetarias. Además, de forma paralela se está realizando una transacción T_2 para un ingreso de M unidades monetarias en la cuenta X. Veamos que problemas pueden presentarse cuando se ejecutan concurrentemente estas transacciones sobre algunas ejecuciones o combinaciones concretas.

6.1.2.1. El problema de la actualización perdida

Este problema se produce cuando dos transacciones tienen operaciones intercaladas que acceden a los mismos elementos, haciendo uso de valores incorrectos. Supongamos que a un sistema llegan las transacciones T_1 y T_2 siguientes y que se intercalan como aparece en la Figura 6.2. En este caso el valor de X es incorrecto porque T_2 lee el valor de X antes de que sea escrito por T_1, con lo que se pierde la actualización realizada por T_1 (*actualización perdida*). Por ejemplo, si el saldo inicial de X y de Y era 1000, se realiza una transferencia de 100 de X a Y, y se hace un ingreso de 50 en X, la intercalación de operaciones de la figura da lugar a un saldo en X de 1050 en lugar de 950.

T_1	T_2
leer_elemento(X) **X=X-N**	

	leer_elemento(X) **X=X+M**
escribir_elemento(X) **leer_elemento(Y)**	
	Escribir_elemento(X)
Y=Y+N **escribir_elemento(Y)**	

Figura 6.2. Intercalación de las operaciones de T_1 y T_2 con actualización perdida

6.1.2.2. El problema de la lectura sucia

Este problema se produce cuando una operación de una transacción realiza una actualización y la transacción no llega a completarse con éxito por algún problema (caída del sistema, problemas en la red, etc.), y otra transacción utiliza el valor actualizado antes de que el elemento actualizado por la transacción fallida se restaure a su valor original.

La Figura 6.3 ilustra que la transacción T_1 modifica el valor de X, pero falla antes de completarse la transacción, por lo que el sistema debe restaurar X al valor que tenía X originalmente, antes de que se iniciase la transacción. Sin embargo, cuando se produce el problema en T_1 y se inicia su restauración, la transacción T_2 ya ha empleado el valor modificado de X, que no ha sido guardado como tal en la base de datos. Cuando se vuelva a ejecutar la transacción T_1 dará como saldo 850 en lugar de 950. Al valor de X en la transacción T_1 se le denomina dato sucio, porque está creado por una transacción que no ha finalizado aún, dando lugar al problema de la lectura sucia.

T_1	T_2
leer_elemento(X) **X=X-N** **escribir_elemento(X)**	
	leer_elemento(X) **X=X+M** **escribir_elemento(X)**
leer_elemento(Y) **>>Error en la transacción<<**	

Figura 6.3. Intercalación de las operaciones de T_1 y T_2 con lectura sucia

6.1.2.3. Problema del resumen incorrecto

Este problema se produce cuando una transacción se está realizando una operación de agregación y se están procesando transacciones que están modificando las fuentes sobre las que se está realizando la función de agregación, de forma que tenga en cuenta algunos valores anteriores a las modificaciones y otros posteriores a las modificaciones.

6.1.2.4. Problema de la lectura no repetible

Este problema surge cuando una transacción T incluye dos operaciones de lectura de un mismo elemento y otra transacción T' modifica el elemento entre las dos lecturas. De esta forma, T recibe dos lecturas diferentes del mismo elemento. Esto puede ocurrir al

consultar el saldo en un cajero para saber si se puede retirar una cantidad, y entre la consulta de saldo y la retirada del dinero se procesa otra transacción que modifica el saldo.

6.1.3. Necesidad de la recuperación

Siempre que se envía una transacción al SGBD, se tiene que asegurar que todas las operaciones de la transacción se lleven a cabo correctamente y su efecto quede registrado permanentemente en la base de datos, o bien, la transacción no tenga efecto alguno sobre la base de datos ni sobre otra transacción. Así, el DBMS no debe permitir que se lleven a cabo unas operaciones de una transacción y otras no. Esto podría deberse a un fallo en la ejecución de la transacción como puede ser caídas del sistema, un error en una operación dela transacción, la cancelación de una operación de una transacción, un problema de concurrencia al intercalar las operaciones de las transacciones, un fallo de disco, etc.

El concepto de transacción es fundamental para el control de la concurrencia y de recuperación de fallos.

6.1.4. Estados de una transacción

Una transacción es una unidad atómica de trabajo que se ejecuta por completo o bien no se ejecuta en absoluto. Para fines de recuperación es necesario saber cuándo se inicia, termina y confirma o aborta una transacción. Así, el gestor de concurrencia del SGBD debe controlar las operaciones:

- `BEGIN_TRANSACTION`: Marca el inicio de una transacción.
- `READ o WRITE`: Operaciones de lectura o escritura que forman parte de la transacción.
- `END_TRANSACTION`: Especifica que las operaciones de lectura y escritura han terminado. En este punto es deseable comprobar si los cambios introducidos por la transacción pueden ser aplicados a la base de datos (confirmar) o si la transacción debe desecharse (abortar).
- `COMMIT_TRANSACTION`: Indica que la transacción se ha llevado a cabo con éxito y que cualquier cambio realizado por la transacción se puede almacenar en la base de datos de forma permanente.
- `ROLLBACK` o `ABORT`: Indican que la transacción no se ha llevado a cabo con éxito, por lo que hay que deshacer todos los cambios realizados por las operaciones de la transacción.

Estas operaciones hacen que las transacciones pasen por una serie de estados, denominados estados de una transacción, y que se ilustran en la figura 6.4.

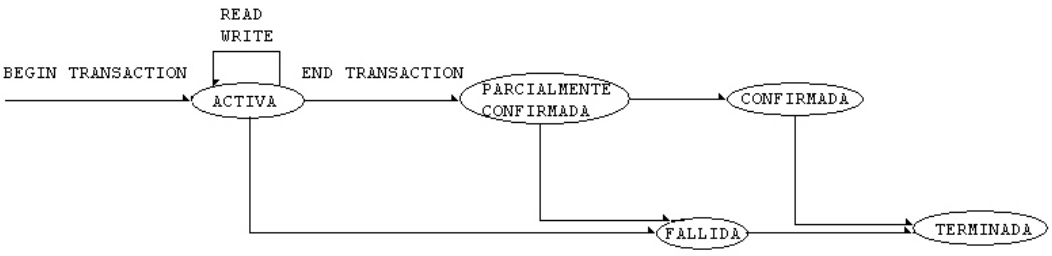

Figura 6.4. Estados de una transacción

- Una operación entra en estado *activo* justo al iniciarse su ejecución. En el estado activo se producen las operaciones de lectura y escritura `READ` y `WRITE`.
- Cuando la transacción acaba, mediante una orden `END TRANSACTION`, la transacción pasa al estado de *parcialmente confirmada*. En este estado se debe comprobar si se ha interferido con otras transacciones concurrentes y se debe asegurar que, en caso de fallo, se podrían guardar todos los cambios introducidos por la transacción.
- Una vez que la transacción ha sido verificada con éxito, la transacción pasa al estado de *confirmada*, indicando que su ejecución ha finalizado con éxito y que todos sus cambios están almacenados en la base de datos.
- En cambio, si se producen fallos en la verificación, o bien se aborta la transacción desde el estado activo, la transacción pasa al estado de *fallida*, haciéndose un `ROLLBACK` de las operaciones de escritura para anular su efecto en la base de datos.
- Una vez que la transacción está en estado de confirmada o de fallida, pasa al estado de *terminada* y el sistema puede pasar a procesar otra transacción.

Si una transacción no se lleva a cabo con éxito, el sistema podrá volver a lanzarla de forma automática. Para ello, se hace uso del *diario* o *registro del sistema* (log), que es lugar donde se almacenan todas las operaciones de modificación de la base de datos, aunque es posible especificar que se registren todas las operaciones realizadas sobre la base de datos (por ejemplo, con el fin de realizar una auditoría). Este registro tiene una serie de entradas como son

- `[start_transaction, T]`: Indica que se ha iniciado la transacción `T`.
- `[write_item, T, valor_anterior, valor_nuevo]`: Indica que la transacción `T` ha cambiado el valor de `X` de *`valor_anterior`* a *`valor_nuevo`*.
- `[commit, T]`: Indica que la transacción `T` se ha llevado a cabo con éxito y por tanto, su efecto se puede almacenar definitivamente en la base de datos.
- `[abort, T]`: Indica que la transacción `T` se abortó.

Cuando se produce un fallo en el procesamiento de una transacción, los planes de recuperación, que no se estudian en esta asignatura, analizan la información del este

registro y deshacen o rehacen las operaciones necesarias de forma que la base de datos quede en un estado consistente.

6.1.5. Propiedades de las transacciones

Las transacciones deben cumplir cuatro propiedades conocidas como propiedades ACID (por sus iniciales en inglés) y es labor de los métodos de control de concurrencia y recuperación del SGBD asegurar que siempre se cumplen. Las propiedades ACID son:

- Atomicidad: Una transacción es una unidad atómica de procesamiento. O se ejecuta de forma completa, o bien no se ejecuta en absoluto.
- Conservación de la consistencia: Una transacción conserva la consistencia si partiendo de un estado consistente de la base de datos la deja en otro estado consistente.
- Aislamiento (*Isolation*): Una transacción debe parecer que se está ejecutando de forma aislada de las demás transacciones. Es decir, las transacciones no debe interferirse entre sí.
- Durabilidad. Los cambios realizados en la base de datos por una transacción confirmada permanecen en la base de datos.

6.1.6. Serializabilidad

Tal y como hemos visto anteriormente, si se intercalan las operaciones de dos transacciones que contengan lecturas y escrituras sobre el mismo elemento de datos, es posible que la base de datos quede en un estado inconsistente. En cambio esto no ocurriría si una transacción se ejecutase después que la otra de forma secuencial, sin ningún grado de paralelismo. En este caso diríamos que las transacciones se ejecutan *en serie*.

A partir de esta idea surge el concepto de *serializabilidad*, que es una propiedad que indica que las operaciones de dos transacciones pueden intercalarse de forma que se comporten como si se estuviesen ejecutando en serie. Este concepto es interesante porque las operaciones de dos transacciones se pueden intercalar de una gran cantidad de formas, dando lugar a muchas combinaciones. Sin embargo, no todas las combinaciones tienen por que ser serializables, es decir, no tienen por que representar una ordenación correcta de las operaciones de las transacciones. A continuación, estudiaremos el concepto de serializabilidad de forma que podamos determinar si dos transacciones son serializables o no.

6.1.6.1. Serializabilidad de un plan

Una posible forma de evitar los conflictos que provoca la ejecución concurrente de transacciones es ejecutar las transacciones en serie, de forma que sólo haya una transacción activa en cada momento. Así, sólo cuando se confirma o aborta una transacción se pasa a ejecutar la transacción siguiente. De esta forma, no importa que transacción se ejecuta primero, y siempre que las transacciones se ejecuten de forma atómica la base de datos se mantendrá en un estado consistente. Sin embargo, este enfoque presenta el problema que limita la concurrencia y, en general, reduce el

rendimiento del sistema. Por tanto, necesitamos planes de transacciones que permitan la concurrencia pero que produzcan el resultado esperado, sin que se produzcan los problemas vistos anteriormente (actualización perdida, lectura sucia, y demás). La solución está en la serializabilidad de planes.

Un plan de *n* transacciones se *serializable* si es equivalente a un plan en serie de las n transacciones. A partir de las n transacciones podemos obtener *n!* posibles planes en serie, y muchos más no en serie. De estos planes no en serie podemos formar dos conjuntos disjuntos: los que son equivalentes a algún plan en serie, y por tanto son serializables, y los que no lo son.

Cuando hablamos que un plan es serializable estamos diciendo que es correcto, ya que es equivalente a un plan en serie, que se considera correcto. Ahora tenemos que ver cuándo dos planes son equivalentes, pero antes necesitamos introducir el concepto de *conflicto de operaciones*.

Dos operaciones de un plan de n transacciones están en conflicto si satisfacen estas tres condiciones:

- Pertenecen a dos transacciones diferentes
- Tienen acceso al mismo elemento `X`
- Al menos una de las dos operaciones es de escritura

De esta forma si denotamos como `r`, `w`, `c`, y a a las operaciones `read_item`, `write_item`, `commit` y `abort`, respectivamente, y utilizamos como subíndice el número de la transacción, podemos ver que en el plan

$$Pa = r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X);\ w_1(Y)$$

las operaciones $r_1(X)$ y $w_2(X)$ están en conflicto, al igual que lo están $r_2(X)$ y $w_1(X)$, así como $w_1(X)$ y $w_2(X)$.

Dos planes son equivalentes (por conflictos) si el orden de cualquier par de operaciones en conflicto es el mismo en ambos planes. Si dos operaciones en conflicto se aplican en un orden diferente, el efecto de los planes puede ser diferente.

Con la noción de equivalencia por conflictos definimos que un plan `P` es *serializable por conflictos* si es equivalente por conflictos a algún plan en serie `P'`. En tal caso, se podrán ordenar las operaciones que no estén en conflicto hasta formar el plan en serie equivalente `P'`. La Figura 6.5 muestra un plan `A` en serie y otro `B` serializable por conflictos, ya que es equivalente a un plan en serie, el `A`.

T_1	T_2
leer_elemento(X) **X=X-N** **escribir_elemento(X)** **leer_elemento(Y)** **Y=Y+N** **escribir_elemento(Y)**	
	leer_elemento(X) **X=X+M** **escribir_elemento(X)**

Plan A

T_1	T_2
Leer_elemento(X) **X=X-N** **escribir_elemento(X)**	
	leer_elemento(X) **X=X+M** **escribir_elemento(X)**
Leer_elemento(Y) **Y=Y+N** **escribir_elemento(Y)**	

Plan B

Figura 6.5. Dos posibles planes de ejecución para las transacciones T_1 y T_2

En este ejemplo, las operaciones en conflicto son $w_1(x)$, $r_2(X)$ y $w_1(x)$, $w_2(x)$, y tal y como se puede observar su orden de ejecución en los planes A y B es el mismo, por lo que el plan B es equivalente por conflictos al plan A, y como A es un plan serie, B es serializable por conflictos.

6.1.6.2. Prueba de serializabilidad por conflictos de un plan

A continuación veremos un algoritmo sencillo que permite determinar si un plan es serializable por conflictos. El algoritmo sólo examina las operaciones read_item y write_item del plan para construir un *grafo de precedencias* o *grafo de serialización*, donde los nodos representan transacciones del plan y un arco desde T_i hasta T_j expresa que una de las operaciones de T_i aparecen en el plan antes que alguna operación en conflicto de T_j.

Algoritmo

1. Para cada transacción T_i que participa en el plan P crear un nodo etiquetado T_i en el grafo de serialización
2. Para cada caso en P en el que T_j ejecuta una orden `read_item(X)` después de que T_i ejecute una orden `write_item(X)` crear un arco T_i->T_j en el grafo.
3. Para cada caso en P en el que T_j ejecuta una orden `write_item(X)` después de que T_i ejecute una orden `read_item(X)` crear un arco T_i->T_j en el grafo.
4. Para cada caso en P en el que T_j ejecuta una orden `write_item(X)` después de que T_i ejecute una orden `write_item(X)` crear un arco T_i->T_j en el grafo.
5. El plan es serializable si, y sólo si, el grafo de precedencia no tiene ciclos.

Para ilustrar el uso del algoritmo, para los dos planes del ejemplo anterior obtendríamos los grafos de la Figura 6.

Plan A **Plan B**
a) **b)**

Figura 6.6. Grafos de serializabilidad de dos planes de ejecución para T_1 y T_2

Como podemos observar, en ninguno de los dos grafos hay ciclos por lo que los planes son serializables. Sin embargo, si disponemos del plan de transacciones de la Figura 6.7, podemos comprobar que no es serializable porque su grafo de serializabilidad contiene ciclos, como muestra la Figura 8.

T_1	T_2
leer_elemento(X) **X=X-N**	
	leer_elemento(X) **X=X+M**
escribir_elemento(X) **leer_elemento(Y)**	
	escribir_elemento(X)
Y=Y+N **escribir_elemento(Y)**	

Figura 6.7. Plan de ejecución no serializable para T_1 y T_2

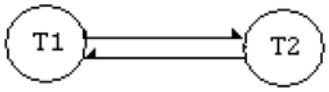

Figura 6.8. Grafo de serializabilidad para el plan de la Figura 7

6.1.7. Soporte de transacciones en SQL

La definición de una transacción en SQL es similar al concepto de transacción que estamos empleando: es una unidad lógica de trabajo atómica. Con SQL no hay una sentencia explícita para begin_transaction, sino que el inicio de una transacción se realiza implícitamente cuando se encuentran determinadas sentencias de SQL. Sin embargo, todas las transacciones deben tener una sentencia explícita de final que puede ser `COMMIT` o `ROLLBACK`.

Las transacciones tienen unas características que se especifican con la sentencia `SET TRANSACTION`, entre las que cabe destacar el modo de acceso y el nivel de aislamiento.

El modo de acceso puede especificarse como `READ ONLY` o `READ WRITE`, siendo éste el valor predeterminado. El modo `READ WRITE` permite que se ejecuten sentencias de modificación de la base de datos, mientras que el modo `READ ONLY` se usa solamente para recuperación de datos.

La opción nivel de aislamiento se especifica con `ISOLATION LEVEL <aislamiento>` donde `<aislamiento>` puede ser `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`. El nivel de aislamiento predeterminado suele ser `SERIALIZABLE`. El uso en SQL de `SERIALIZABLE` indica que no se pueden permitir violaciones que causen lecturas sucias, lecturas no repetibles

ni fantasmas. Si una transacción se ejecuta con un nivel de aislamiento inferior a `SERIALIZABLE` pueden ocurrir algunas de estas anomalías:

- **Lectura sucia**. Una transacción T_1 puede leer una escritura realizada por la transacción T_2 pero que todavía no ha sido confirmada. Si T_2 no se completa, T_1 habrá leído un valor incorrecto.
- **Lectura no repetible**. Una transacción T_1 lee un valor de la base de datos. Si otra transacción T_2 actualiza más tarde dicho valor y T_1 vuelve a leer el valor se encuentra con un valor diferente.
- **Fantasmas**. Una transacción T_1 lee un conjunto de valores mediante la ejecución de una consulta. Si otra transacción T_2 introduce un nuevo valor que podría salir como resultado de la consulta, si T_1 vuelve a ejecutar la consulta verá una fila que antes no existía (un *fantasma*)

La tabla siguiente muestra las violaciones que se pueden cometer en función del tipo de aislamiento definido.

	Permite la violación		
Nivel de aislamiento	Lectura sucia	Lectura no repetible	Fantasma
`SERIALIZABLE`	No	No	No
`REPEATABLE READ`	No	No	Sí
`COMMITED READ`	No	Sí	Sí
`UNCOMMITED READ`	Sí	Sí	Sí

Por tanto, los administradores de la base de datos y los programadores de aplicaciones deberán utilizar estas características del procesamiento de transacciones para mejorar el rendimiento de la base de datos de forma que las transacciones puedan tener cierto grado de intercalación sin dejar de ser serializables.

6.2. Introducción a la seguridad y autorización en bases de datos

Las bases de datos normalmente son accedidas por varios usuarios por lo que el SGBD debe proporcionar técnicas que permitan restringir a algunos usuarios o grupos de usuarios el acceso a determinadas partes de la base de datos. Esto es especialmente importante cuando se trata de organizaciones que cuentan con una gran base de datos que almacena los datos de distintas secciones de la organización. Para ello, los SGBD suelen contar con un subsistema de seguridad y autorización que asegura la seguridad de la base de datos frente accesos no autorizados. Estos mecanismos de seguridad suelen ser de dos tipos:

- *Mecanismos de seguridad discrecionales*. Permiten conceder privilegios a los usuarios de la base de datos.
- *Mecanismos de seguridad obligatorios*. Permiten definir varios niveles de seguridad para los objetos de la base de datos, de forma que los usuarios

pueden utilizar sólo los elementos que están en su nivel de seguridad o inferior.

Otro problema de seguridad es el de evitar que personas no autorizadas tengan acceso al sistema. El mecanismo de seguridad de un SGBD debe incluir formas que permitan restringir el acceso al sistema, Esa función se denomina *control de acceso* y está basado en el uso de cuentas de usuario.

6.2.1. La seguridad de la base de datos y el DBA.

El administrador de la base de datos (DBA) es la persona responsable de la base de datos y entre sus obligaciones se encuentra la de conceder privilegios a los usuarios que necesitan utilizar el sistema. El DBA tiene una cuenta de superusuario que ofrece privilegios no disponibles para las cuentas convencionales. Entre estos privilegios se encuentran el de creación de cuentas, concesión y revocación de privilegios, y asignación de niveles de seguridad.

De esta forma, con la figura del DBA, todas las personas que deseen tener acceso a la base de datos, tendrán que solicitar la DBA la creación de una cuenta de usuario con un nombre de usuario y una contraseña. El usuario utilizará estos datos y podrá acceder a la base de datos siempre que el SGBD valide dicha conexión. El SGBD podrá guardar todas las operaciones que se realicen sobre la base de datos, de forma que se puedan detectar anomalías en el acceso o en la modificación de los datos. Estas operaciones son guardadas en el diario del sistema (*log*) y el número de entradas correspondientes a operaciones realizadas sobre la base de datos dependerá del grado de detalle con el que se quieran registrar las operaciones sobre la base de datos.

6.2.2. Control de acceso discrecional basado en la concesión y revocación de privilegios

La forma más común de imponer un control de acceso discrecional es mediante la concesión y revocación de privilegios. El control de acceso se puede realizar a nivel de cuenta o a nivel de objeto de la base de datos (en un SGBD será a nivel de tabla), especificando en cada caso las operaciones que es posible realizar.

- A nivel de cuenta, el DBA especifica los privilegios que tiene cada cuenta, independientemente de los objetos de la base de datos. En este nivel se encuentran privilegios como el de crear o modificar esquemas, crear o modificar tablas, consultar y modificar tablas, etc.
- A nivel de tabla, se pueden controlar los privilegios de acceso a cada tabla o vista de la base de datos para cada usuario, obteniendo una matriz bidimensional con una dimensión para los objetos de la base de datos y la otra para las cuentas de usuarios. Las celdas de la matriz especifican los privilegios de cada usuario sobre cada objeto.

6.2.3. Privilegios en SQL

En SQL hay definidos seis tipos de privilegios: `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `REFERENCES` y `USAGE`. Los cuatro primeros se aplican a una tabla o vista e indican si el propietario de esos privilegios puede consultar, insertar, eliminar o actualizar, respectivamente, los datos de una tabla o vista. El privilegio `REFERENCES` especifica si es posible hacer referencia a una relación en una restricción de integridad. Por último, el privilegio `USAGE` se utiliza sobre dominios o elementos del esquema que no sean relaciones o aserciones, y especifica el derecho del usuario a utilizar ese elemento.

A los privilegios `INSERT`, `UPDATE` y `REFERENCES` es posible asignarles un atributo como argumento, de forma que el privilegio definido sólo se aplique a dicho atributo. Así, es posible autorizar el acceso a cualquier subconjunto de las columnas de una relación. No obstante, para limitar el acceso a las columnas o filas de una tabla también se pueden utilizar las vistas. Para ello, se crea una vista sobre una o varias tablas seleccionando ciertas columnas y las filas que cumplan un predicado y se ofrece a los usuarios la vista en lugar de la tabla.

Estos privilegios determinan las acciones que se pueden llevar a cabo sobre los elementos de la base de datos, pero es necesario contar con un mecanismo que permita la asignación y revocación de privilegios a los usuarios sobre los objetos de la base de datos. Decir antes de describir cómo se realiza la concesión y revocación de privilegios que cuando se crea un elemento de la base de datos, éste es propiedad del usuario que lo crea, y por tanto, tiene todos los privilegios sobre dicho objeto. No obstante, también es posible transmitir estos privilegios a un usuario denominado `PUBLIC` que representa a todos los usuarios.

Por tanto, por ahora sólo conocemos una forma de tener privilegios sobre los objetos de la base de datos, que es creando los objetos. Para evitar esto, de forma que otros usuarios diferentes al creador de los objetos puedan utilizarlos, se define el mecanismo de concesión de privilegios. Asimismo, es posible retirar a un usuario los privilegios concedidos anteriormente.

6.2.3.1. Concesión de privilegios

SQL ofrece la sentencia `GRANT` para la concesión de privilegios, donde uno de estos privilegios puede ser a su vez el de concesión de privilegios. De esta forma es posible conceder privilegios de selección y concesión a un usuario sobre una tabla, de forma que este usuario puede conceder a su vez estos privilegios y no más a un tercer usuario. La sintaxis general de `GRANT` es la siguiente:

`GRANT` *`privilegio`* `ON` *`elemento`* `TO` *`usuario`* `[WITH GRANT OPTION]`
donde *`privilegio`* puede ser alguno de los vistos al principio de este apartado (`SELECT`, `INSERT`, etc.) o `ALL PRIVILEGES` como abreviatura de todos los privilegios que el usuario puede conceder, y *`elemento`* es básicamente una tabla, vista, aunque puede ser otra cosa más (dominio, atributo, y demás).

Por ejemplo, el creador de una tabla A podría conceder permisos de consulta, inserción y concesión a los usuarios `Usuario1` y `Usuario2` con

```
GRANT SELECT, INSERT ON A TO Usuario1, Usuario2 WITH GRANT OPTION
```

Así, el creador de la tabla ha transmitido los privilegios de selección e inserción a los usuarios 1 y 2 con la opción de concesión, por lo que estos usuarios podrían conceder estos mismos privilegios a otros usuarios.

Por ejemplo, el usuario 1 ahora podría otorgar derecho de consulta sobre `A` a un tercer usuario, como se muestra a continuación

```
GRANT SELECT ON A TO Usuario3
```

pero este tercer usuario ya no podría transmitir su único privilegio a ningún otro usuario de la base de datos.

6.2.3.2. Revocación de privilegios

Los privilegios concedidos a otros usuarios pueden ser revocados en cualquier momento, e incluso pueden ser revocados en cascada, de forma que se revoquen los privilegios concedidos por un usuario al que se concedieron privilegios de concesión. La sintaxis para la revocación de privilegios en SQL es esta

```
REVOKE privilegios ON elemento FROM usuario [CASCADE | RESTRICT]
```

donde *`privilegios`* es la lista de privilegios que se quieren retirar a un usuario, *`elemento`* es el elemento sobre el que se retiran los privilegios y *`usuario`* es el nombre de usuario al que se retiran los privilegios. `CASCADE` propaga la retirada de privilegios a los usuarios, mientras que `RESTRICT` impide que se revoquen los privilegios de un usuario si éste a su vez los concedió a terceros. Para facilitar esta tarea de retirada de privilegios en conveniente utilizar un grafo de concesión de privilegios y analizarlo cuando se vayan a retirar los privilegios para ver el efecto antes de proceder a la retirada de los privilegios.