SQL desde lenguajes de programación

70-77

Introducción !

- Los programadores deben tener acceso a la base de datos mediante lenguajes de propósito general.
    - No todas las consultas se pueden expresar en SQL
    - Las acciones no declarativas (impresión de informes, interacción con usuarios) no se puede realizar con SQL
- Existen dos enfoques sobre el acceso a SQL desde lenguajes de propósito general:
    - **SQL dinámico:** el programa de propósito general se conecta y comunica con la base de datos utilizando una colección de funciones o métodos
        - Se envía la consulta como una cadena de caracteres.
        - Se pueden enviar consultas en tiempo de ejecución
    - **SQL embebido:** las sentencias de SQL se crean en tiempo de compilación con un preprocesador, que las envía a la base de datos para su compilación y optimización.

Norma JDBC !

- **JDBC (Java Database Connectivity)** es una **interfaz de programación** de aplicaciones que usan los programas de Java para conectarse a base de datos.
- Se configura el driver y se usa getConnection para abrir una conexión a la base de datos, pasándose como parámetros la URL del servidor, un identificador del usuario de la base y una contraseña.

```
this.conexion = java.sql.DriverManager.getConnection("jdbc:" + gestor +
"://" +
configuracion.getProperty("servidor") + ":" +
configuracion.getProperty("puerto") + "/" +
configuracion.getProperty("baseDatos"),
usuario);
```

- Para el envío de datos a la base, se usa un objeto de la clase Statement
    - Un objeto Statement permite que los programas de Java invoquen los métodos que envían una sentencia de SQL como argumento para su ejecución en la base de datos.

    - Para ejecutar una sentencia se invoca stmt.executeQuery o stmt.executeUpdate (consulta/no consulta).

- Para recibir los datos de una consulta, se usa la clase ResultSet (ResultSet ret = stmt.executeQuery(...))
- Se puede declarar un objeto de la clase **PreparedStatement** que permite declarar el formato de un Statement para completar más adelante
    - Para completar uno de los atributos se utilizan los métodos como pStmt.setString o pStmt.setInt
    - Más eficiente si se reutiliza el mismo preparedstatement varias veces.
    - PreparedStatement puede evitar errores/ataques intencionados de **inyección SQL** (incluir caracteres como ‘ en un atributo para romper la sintaxis)
        - Esto es porque la cadena de entrada incluiría caracteres de escape tras cada atributo (que se ha insertado por separado), evitando el error.
    - En el lugar de .executeUpdate(), se puede llamar a pstmt.**addBatch();**, que añade la inserción/consulta que se haya escrito a la cola en lugar de ejecutarla inmediatamente. Cuando se haga .executeUpdate(), se pasarán todos los datos que se hayan añadido a la cola.
        - Esta es la forma correcta de trabajar con muchas filas simultáneamente **(!)**
- **CallableStatement** llama a una función existente en la base de datos, con determinados parámetros.
- La interfaz ResultSet que devuelve executeQuery tiene un método **getMetaData()**, con información sobre los metadatos (nº de columnas, nombre de una columna concreta) de una consulta.

```
ResultSetMetaData meta = rs.getmetadata()
int numeroColumnas = meta.getColumns()
```

- La conexión se debe cerrar para no superar ellímite de conexiones a una base de datos. Se usa conn.close().

Creación de tablas (ejemplo)

```
//Creamos el resultset suponiendo que hay una sentencia
 ResultSet rs = stm.executequery()
 //Cogemos información de los meta datos
 ResultSetMetaData meta = rs.getmetadata()
 int numeroColumnas = meta.getColumns()
 //Creamos la tabla
 DefaultTableModel tablemodel = new Defaulttablemodel()
 //Iteramos a través del result set, como podria ser el siguiente
```

```
while(rs.next){
            //Creamos un obj para almacenar los datos de la final
            Object[] row = new Object[numeroColumnas]
            for(int i = 0; i < numeroColumnas; i++)}
            row[i] = rs.getObject(i)
            }
            tableModel.addrow(row)
            }

            JTable table = new Jtable(tablemodel)
```

Norma ODBC

- Define una API que pueden usar las aplicaciones para abrir conexiones y enviar consultas/actualizaciones a una base de datos.
- Los comandos se declaran como strings (char *) que se pueden enviar con SQLExecDirect..
- La función SQlBindCol permite vincular una columna de la tabla del resultado con una variable del lenguaje de programación
- La función SQLFetch tras usar SQLBindCol guarda los datos de la consulta en las variables.
- El código se guarda en un bucle while(SQLFetch()==SQL_SUCCESS) para obtener todos los datos disponibles.

SQL incorporado

- Consiste en usar sintaxis de SQL en un lenguaje anfitrión (C, C++, Java).
- Para identificar las consultas de SQL se usa EXEC SQL <código SQL>
- Antes de comenzar, se cebe usar EXEC SQL **connect to** servidor **user** usuario **using** contraseña

Seguridad

64-67, 181-185

Autorización (SQL)

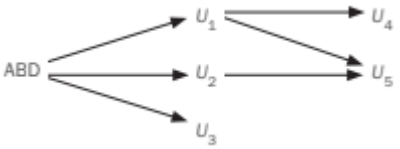
- Los tipos de autorización se denominan **privilegios:** de lectura, de inserción, de actualización y de borrado (select, insert, update, delete)
    - all privileges incluye todos los que se pueden conceder
- Sintaxis: **grant** select **on** *cuenta* **to** Martín, María → Concede permiso de selección en la relación cuenta a Martín y María.
    - Para atributo concreto: **grant update** *(importe*) **on** *préstamo* **to** Martín, María
    - Para retirar: **revoke** en lugar de **grant**
    - public: incluye a todos los usuarios presentes y futuros
- Se pueden conceder a usuarios o a roles

Roles

- Un **rol** define un conjunto de autorizaciones.
- Se crean con **create role** profesor, y luego se pueden conceder varios permisos a un rol. El rol luego se concede a usuarios como si fuese un permiso, y tendrán todos los permisos del rol.
    - Un rol se puede conceder también a otro rol.

Transferencia de privilegios (!)

- Como predeterminado, un usuario que tiene un privilegio no puede concedérselo a otros usuarios. Se puede hacer que un privilegio sea transferible si se especifica **with grant option** en el comando grant.
- Esto crea un **grafo de autorización** donde se incluye una arista $U_i \rightarrow U_j$ si el usuario U_i ha concedido al usuario U_j un permiso concreto.
    - La raíz es el administrador de la base de datos.
    - Varios usuarios pueden concederle el mismo permiso a otro usuario.
    - Un usuario tiene una autorización si y solo si existe un camino desde la raíz hacia su nodo.



- Si, en el caso previo, U_1 pierde sus privilegios:
    - U_5 mantendría siempre su privilegio, ya que hay un camino desde U_2
    - U_4 lo perdería a no ser que se especifique la opción **restrict** que evita revocaciones en cascada (comportamiento predeterminado).
- Si un usuario le concede un rol a otro (un decano le concede a otro usuario el rol de profesor), y luego el decano pierde su rol, el comportamiento predeterminado sería que el profesor también lo pierda, lo cual no es deseable.
    - Si se usa la cláusula **granted by current_role** al conceder el rol, el rol del profesor no dependerá del del decano.
    - El rol de profesor sólo se eliminaría si se retira manualmente del profesor o si se modifica 'decano' para que ya no pueda conceder profesores.
    - Antes de invocar **current_role** es necesario usar **set rol** *nombre_rol* durante una sesión para especificar el rol que tiene concedido el usuario.
- **Ejemplo (2023-2024):** Dadas las siguientes secuencias de comandos, qué les pasa a los roles de Marta y Juan?

Usuario=ADMIN	Usuario=ADMIN
CREATE USER María GRANT Gestor TO María	CREATE USER María GRANT Gestor TO María
Usuario = María	Usuario = María
GRANT Cajero TO Juan GRANT Cajero to Marta	GRANT Cajero TO Juan GRANTED BY current_role GRANT Cajero to Marta GRANTED BY current_role
Usuario = ADMIN	Usuario = ADMIN
REVOKE Gestor TO María DROP USER Maria	REVOKE Gestor TO María DROP USER Maria

- En el primer caso, Juan y Marta son cajeros porque el rol se los otorgó María. Cuando María deja de ser gestor, desaparece la entidad que había otorgado el rol de Cajero. Por lo tanto, Juan y Marta dejan de ser cajeros.
    - A no ser que algún otro gestor le hubiese otorgado también Cajero a Juan y Marta
- En el segundo caso, Juan y Marta son cajeros porque el rol se los otorgó el current_role, es decir, Gestor. Aunque este rol sea retirado de María, mientras que siga existiendo el rol de 'gestor' en la base de datos y este rol tenga permiso de definir cajeros, Juan y Marta seguirán siendo cajeros.

Seguridad de las aplicaciones

- **Cifrado:** Debe ser fácil de cifrar y descifrar para los usuarios autorizados, pero difícil para los intrusos.
    - Depende del algoritmo clave de cifrado
    - **DES:** sustitución y reordenación de caracteres en base a clave de cifrado.
    - **AES:** mejora sobre DES. sigue siendo de clave compartida entre los usuarios autorizados
    - **Cifrado de clave pública:** alternativa a AES. Cada usuario tiene una clave pública y una privada
        - La pública se usa para cifrar, la privada para descifrar
        - Dada la clave pública debe ser dificil calcular la privada. Se aprovecha que es difícil encontrar factores primos de un número
- **Autenticación:** Se puede hacer por contraseñas pero existen alternativas mejores:
    - Evitar conexiones a SGBDs de cualquier IP
    - **Respuesta por desafío:** La base de datos envía una cadena de desafío al usuario
        - El usuario la cifra usando una contraseña secreta como clave y devuelve el resultado. El sistema descifra la cadena usando la clave y si coincide se permite acceso
        - Mejor porque las contraseñas no circulan por la red
    - **Firmas digitales:** La clave privada se usa para firmar los datos y se puede usar la clave pública para autentificarlo. También sirve para garantizar el no repudio (non podes excusarte e decir que NON fixeches algo)
    - **Servidor LDAP:** Almacena usuarios y contraseñas. Las aplicaciones usan el servidor LDAP en lugar del de la base directamente para autenticar.
- **Trazas de auditoría:** Registro de todos los cambios en la base de datos junto con el usuario que lo srealizó.
    - Permite detectar una brecha de seguridad o trazar al usuario que realizó un cambio incorrecto.
    - Una posibilidad es crear un trigger que actualice una tabla de registro de auditoría cada vez que hay un cambio.
    - Es recomendable copiar las trazas de auditoría en una máquina diferente por seguridad.
- **Privacidad:** Se deben tener en cuenta las leyes de protección de datos en cuanto a los datos que se pueden compartir con terceros o almacenar durante un tiempo determinado.

Transacciones

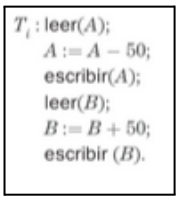
291-305

Definición

- Una **transacción** es una colección de operaciones que forman una única unidad lógica de trabajo.
- **Atomicidad:** Los pasos de una transacción, delimitados por begin transaction y end transaction, deben ser **indivisibles:** se ejecutan todos o ningunos.
- **Aislamiento:** Las sentencias de una transacción se deben ejecutar sin interferencia de otras sentencias que se ejecuten concurrentemente.
- **Durabilidad:** Las acciones de una transacción deben permanecer incluso si el sistema se cae justo después del commit.
- **Consistencia:** Las transacciones deben mantener todas las restricciones y reglas de la base de datos. El estado final de la transacción debe ser válido.

Ejemplo

- Sea una transacción la transferencia de 50€ de A a B como la de la imagen.



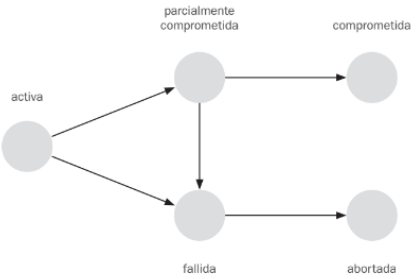
- **Consistencia:** La suma de A+B no debe cambiar tras ejecutar la construcción. La responsabilidad de esto cae en el programador de la aplicación que codifica la transacción.
- **Atomicidad:** Si la transacción fallase después del escribir(A), A perdería 50 euros que no se enviarían a B.
    - Estos estados inconsistentes no deben ser visibles. Para esto, se guarda en el sistema de la base de datos un **log** de los valores previos a la transacción. Si esta falla, se restaura el log.
- **Durabilidad:** Para evitar que se confirme la transacción al usuario pero después un error cause la caída del sistema antes de guardar los datos en disco, estos se deben guardar en disco antes de completar la transacción de forma temporal.
    - Los datos temp guardados son suficientes para restaurar los efectos en caso de reinicio
- **Aislamiento:** Si varias transacciones se ejecutan concurrentemente se pueden producir estados inconsistentes. Esto se solucionaría ejecutándolas todas secuencialmente, pero esto empeora el rendimiento. Existen mecanismos para poder ejecutar transacciones concurrentemente evitando errores.

Almacenamiento

- **Almacenamiento volátil:** No sobrevive a las caídas del sistema. Caché, memoria
- **Almacenamiento no volátil:** Sobrevive a caídas. Discos magnéticos, medios ópticos.
- **Almacenamiento estable:** Nunca se pierde (en teoría). Se consigue replicando la información en varios medios no volátiles con modos de fallo independientes.

Atomicidad y durabilidad

- Cuando se **aborta** una transacción iniciada, se deben revertir los cambios realizados en sus pasos. Tras esto se denomina **retrocedida,** mientras que una que termina correctamente se denomina **comprometida** (**comitted**)
    - Tras ejecutarse la última sentencia pero antes de comittearse la transacción está 'parcialmente comprometida'.



- Esto se consigue guardando un log de las modificaciones.
- Una transacción comprometida no se puede abortar, sólo se puede realizar una transacción compensadora.
    - Una transacción compensadora puede ser útil en casos con escritura externa. Por ejemplo, si un cajero automático falla antes de dispensar el dinero, en lugar de dispensar más dinero cuando se reinicie el sistema se debería realizar una transacción compensadora que devuelva el dinero a la cuenta del usuario.
- Cuando se aborta una transacción se puede reiniciar o cancelar, normalmente dependiendo de si el error fue del sistema o con el programa o entrada.

Aislamiento

- Planificación **secuencial:** Se ejecutan todos los pasos de una transacción y luego todos los pasos de la otra.
    - Asegura que no hay errores de aislamiento
- Es conveniente permitir la concurrencia de transacciones para mejorar el rendimiento, la utilización del procesador y reducir el tiempo medio de respuesta.
- En este caso se realiza **planificación concurrente:** se intercalan los pasos de una con los de otra

- Se debe respetar el orden en el que se solicitaron las transacciones: por ejemplo, si una trabaja sobre el 10% del saldo y otra reduce el saldo en una cantidad, es importante cuál va antes porque el 10% del saldo será distinto.
- Si se deja la planificación al SO por completo, este no tendrá en cuenta estos casos y se pueden causar estados inconsistentes.

Secuencialidad

- Se dice que dos instrucciones I y J están en **conflicto** si afectan al mismo conjunto de datos y al menos unas es de escribir.
- Si dos instrucciones consecutivas de una planificación son de transacciones diferentes y no están en conflicto, se puede cambiar su orden para producir otra planificación S' equivalente a la original.
  - Si una planificación S se puede transformar en otra S' mediante intercambios de instrucciones no conflictivas, S y S' son **equivalentes en cuanto a conflictos**.
- Esto significa que una planificación concurrente que intercala varias transacciones se puede transformar en una secuencial donde los pasos de cada transacción vas juntos, manteniendo la equivalencia.
  - Una planificación es **secuenciable en cuanto a conflictos** si es equivalente en cuanto a conflictos a una secuencial.
- Para determinar la secuencialidad en cuanto a conflictos de una planificación S se puede realizar su grafo de precedencia, donde G = (V,E)
  - V: Conjunto de transacciones de la planificación
  - Vértices: Secuencias $T_i \rightarrow T_j$ que cumple:
    - T_i ejecuta escribir antes que T_j ejecute leer
    - T_i ejecuta leer antes que T_j ejecute escribir
    - T_i ejecuta escribir antes que T_j ejecute escribir
  - Si existe una arista $T_i \rightarrow T_j$, en cualquier planificación S' equivalente a S, T_i debe ir antes que T_j.
  - Si el grafo no contiene un ciclo, la planificación S es secuenciable.

T_1	T_2
leer(A)	
escribir(A)	
	leer(A)
	escribir(A)
leer(B)	
escribir(B)	
	leer(B)
	escribir(B)

Figura 14.6. Planificación 3: solo se muestran las instrucciones leer y escribir.

T_1	T_2
leer(A)	
escribir(A)	
	leer(A)
leer(B)	
	escribir(A)
escribir(B)	
	leer(B)
	escribir(B)

Figura 14.7. Planificación 5: planificación 3 tras intercambiar un par de instrucciones.

T_1	T_2
leer(A)	
escribir(A)	
leer(B)	
escribir(B)	
	leer(A)
	escribir(A)
	leer(B)
	escribir(B)

Figura 14.8. Planificación 6: planificación secuencial equivalente a la planificación 3.

Aislamiento y atomicidad

- Si una transacción T_i falla, se deben abortar todas las transacciones que dependan de ella (que lean los datos que ha escrito), Para poder hacer esto las planificaciones deben cumplir algunos requisitos.
- Una **planificación recuperable** es aquella en la que para todo par de transacciones T_i y T_j tales que T_j lee datos que ha escrito T_i, la operación commit de T_i aparece **antes** que la de T_j.
    - Para conseguir esto, se puede retrasar el commit de T_j hasta que lo haga T_i.
- Una **planificación sin cascada** es aquella en la que para todo par de transacciones T_i y T_j tales que T_j lee datos que ha escrito T_i, la operación de commit de T_i aparece **antes** que la operación **de lectura** de T_j.
    - Toda planificación en cascada es siempre recuperable.
    - Una planificación que no es sin cascada puede ser recuperable, pero puede llevar a que para recuperarla se deba hacer retroceso en cascada, lo cual no es deseable.

Niveles de aislamiento definidos en SQL

- **Secuenciable:** Normalmente asegura la ejecución secuencial
- **Lectura repetible:** Solo permite leer datos comitteados y, entre dos lecturas de un dato, ninguna otra puede actualizarlo.
- **Lectura comprometida:** Solo permite leer datos commiteados, sin lectura repetible. Entre dos lecturas de un dato, otra transacción podría modificarlo y commitear.
- **Lectura no comprometidas:** Permite leer datos no commiteados. Nivel de aislamiento mínimo.
- Ningún nivel permite escrituras sucias: escribir sobre lo que ha escrito otra transacción que no ha commiteado o abortado.
- En postgreSQL se utiliza **change isolation level**.
    - El valor predeterminado en postgreSQL es lectura comprometida, o READ COMMITED.

Niveles de aislamiento (implementación)

- El objetivo es asegurar que las planificaciones generades sean secuenciales en cuanto a conflictos o vistas y que sean sin cascadas, pero manteniendo la máxima concurrencia posible.
- La implementación más sencilla que asegura la secuenciabilidad es el **bloqueo** de la base de datos mientras se ejecuta una transacción.
    - Bloquear toda la base de datos → planificaciones secuenciales, pero mal rendimiento.
    - Bloqueo en dos fases: primero la transacción va adquiriendo bloqueos sin liberarlos, en la segunda fase los va liberando sina adquirir nuevos. Esto asegura que no libere un bloqueo antes de tiempo.
    - Bloqueo compartido: se usa para lectura. Permite que otras transacciones que tengan bloqueo compartido lean el mismo dato.
    - Bloqueo exclusivo: se usa para modificación. Ninguna otra transacción puede leer ni modificar el dato.
- Otra categoría de técnicas es la de asignar a cada transacción una **marca de tiempo**, normalmente de su inicio.
    - En cada dato, se guarda la marca de tiempo de la transacción más reciente que la leyó, y la del dato que escribió el valor actual del dato.
    - Si hay un conflicto entre dos transacciones para acceder a un dato, acceden en el orden de sus marcas de tiempo. Si no es posible, las transacciones se reinician.
- **Aislamiento de instantáneas:** Se guarda más de una versión de cada dato, permitiendo que cada transacción se le da su versión privada (instantánea) de la base de datos.
    - La trans. lee y actualiza su propia versión de la bdd, no la real.
    - Se guardan los datos en la base real sólo cuando se compromete. Si está parcialmente comprometida, solo procede a comprometida si ninguna otra transacción ha modificado el dato que intentaba actualizar.
    - **Problema:** en algunos casos, dos transacciones que modifican el mismo dato no verán los cambios de la otra, y fallarán ambas, cuando se podría secuenciar si no se usase aislamiento.

Transacciones en SQL

- Aunque previamente solo se estudiaban operaciones genéricas read/write, en SQL además de select/update existen insert y delete.

- Evidentemente, realizar un insert simultáneamente a un select puede causar un conflicto, en concreto que ocurre con datos que pueden existir o no en la base de datos (según si el insert fue antes o después). Esto se denomina **fenómeno fantasma**.
- Entonces, no es suficiente considerar las tuplas a las que accede una transacción, sino la información que se utiliza para encontrar tuplas, pues las tuplas a las que accede pueden ser distintas dependiendo del estado de la base.
- Esto significa que si se usa un bloqueo puede no ser suficiente con bloquear ciertas tuplas, sino que se podría realizar **bloqueo de predicado**, pero esto es costoso y no se usa.

Control de concurrencia

309-312, 314-316, 320-328

Protocolos basados en el bloqueo

- **Bloqueos:** Forma más sencilla de control de concurrencia. Existen os tipos de bloqueo sobre un elemento Q:
    - Compartido: Si T_i obtiene un bloqueo en modo compartido (C) sobre Q, puede leer Q pero no puede escribirlo.
    - Exclusivo: Si T_i obtiene un bloqueo en modo exclusivo (X), puede leer y escribir.
- Dado un conjunto de modos de bloqueo se puede definir una función de compatibilidad.
    - Por ejemplo, sean A y B dos tipos de bloqueos teóricos. Si cuando un recurso Q tiene un bloqueo de tipo B y otra transacción solicita un bloqueo de tipo A:
        - Si se le puede conceder a pesar del bloqueo de tipo B, A es compatible con B.

	C	X
C	cierto	falso
X	falso	falso

    - Esto (ver tabla) significa que varios recursos pueden tener bloqueos de modo compartido para leer un recurso sobre el mismo dato. Una petición posterior de bloqueo exclusivo para escribir debería esperar a que todos acabasen.
- Pueden producirse **interbloqueos** si ocurre el caso de que T_1 tiene un bloqueo exclusivo sobre Q y quiere uno compartido sobre Q_2, pero T_2, tiene uno exclusivo sobre Q_2 y quiere uno sobre Q. Esto lleva a que ambos estén esperando a que el otro acabe.
    - Si ocurre esto el sistema debe retroceder una de las dos transacciones.
- Protocolo de bloqueo: conjunto de reglas que indica el momento en que una transacción puede bloquear y desbloquear cada uno de los datos.
    - Algunos protocolos de bloqueo sólo eprmiten planificaciones secuenciales → garantizan aislamiento. Sin embargo, esto limita el nº de planificaciones posibles
    - Se dice que una planificación S es **legal** bajo un protocolo de bloqueo si S es una planificación posible para un conjunto de transacciones que sigan las reglas del protocolo de bloqueo.
    - Se dice que un protocolo de bloqueo **asegura** la secuencialidad si y solo si todas las planificaciones legales son secuenciables en cuanto a conflictos.
- Se dice que T_i **precede** a T_j en la planificación S (T_i—>T_j) si existe un elemento de datos Q tal que:
    - T_i ha obtenido un bloqueo en modo A sobre Q
    - T_i ha obtenido un bloqueo en modo B sobre Q

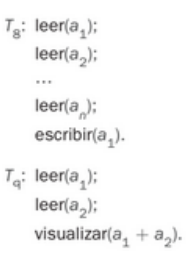
- comp(A,B)=falso → si hay bloqueo B, no se concede bloqueo A.
- Esto quiere decir que si $T_i \rightarrow T_j$, T_i debe aparecer primero en cualquier planificación secuencial.

Concesión de bloqueos

- En general, si una transacción solicita un bloqueo que es compatible con el actual, se puede conceder. Pero se puede producir el caso de que varias transacciones obtengan sus bloqueos compartidos una tras otra y bloqueen mucho tiempo a una que está esperando por un bloqueo exclusivo. Se dice que esta transacción tiene **inanición**.
- Se puede solucionar si sólo se conceden bloqueos cuando no exista otra transacción que haya solicitado bloqueo y lo haya pedido antes.

Protocolo de bloqueo de dos fases !!

- Bloqueo en dos fases: primero la transacción va adquiriendo bloqueos sin liberarlos, en la segunda fase los va liberando sina adquirir nuevos. Esto asegura que no libere un bloqueo antes de tiempo.
- El punto de la planificación donde acaba la fase de crecimiento se denomina **punto de bloqueo** de la transacción. Se pueden ordenar las transacciones según sus puntosde bloqueo.
- Este protocolo no asegura la ausencia de interbloqueos, debido a que puede haber planificaciones con cascada.
  - Los retrocesos en cascada se pueden evitar con una modificación: **protocolo de bloqueo estricto de dos fases:** además de que el bloqueo sea en dos fases, una transacción debe poseer todos los bloqueos en modo exclusivo que tome hasta que se complete la transacción.
  - Esto evita que ninguna transacción lea el dato que está escribiendo una transacción no comprometida.
- El protocolo de bloqueo en dos fases asegura la secuencialidad, siempre y cuando el nivel de aislamiento sea el correcto y
- Otra variante es el **protocolo de bloqueo riguroso de dos fases:** una transacción debe poseer todos los bloqueos que tome de cualquier tipo hasta que se comprometa.
  - Se puede comprobar que con este protocolo se pueden secuenciar las transacciones en el orden en que se comprometen.



- Algunas planificaciones con el protocolo en dos fases se podrían mejorar si se permitiese la **conversión** de un tipo de bloqueo a otro. Por ejemplo, en el ejemplo dado, si

inicialmente T_8 tomase a_1 con bloqueo sólo en modo compartido y luego cuando quisiese escribir lo tomase con modo exclusivo, se obtendría mayor eficiencia.

- Se denomina la conversión de compartido a exclusivo como subir, y al revés bajar.

Ejemplo: planificación, bloqueos (2, Junio 2016)

- **La colocación de los bloqueos en las siguientes transacciones:**
  - **(a) ¿cumplen las reglas del protocolo de bloqueo en dos fases? ¿por qué?**

    No. T1 desbloquea un recurso y luego bloquea uno nuevo. Una vez ha comenzado a liberar recursos, no puede adquirir nuevos recursos.

T1	T2
Bloquear-X(B)	Bloquear-C(A)
Leer(B)	Leer(A)
B:=B-50	Bloquear-C(B)
Escribir(B)	Leer(B)
Desbloquear(B)	Desbloquear(B)
Bloquear-X(A)	Desbloquear(A)
Leer(A)	Visualizar(A+B)
A:=A+50	
Escribir(A)	
Desbloquear(A)	

(b) cambia los bloqueos para que ambas transacciones cumplan las reglas del protocolo de bloqueo en dos fases.

T1	T2
Bloquear-X(B)	Bloquear-C(A)
Leer(B)	Leer(A)
B:=B-50	Bloquear-C(B)
Escribir(B)	Leer(B)
Bloquear-X(A)	Desbloquear(B)
Leer(A)	Desbloquear(A)
A:=A+50	Visualizar(A+B)
Escribir(A)	
Desbloquear(A)	
Desbloquear(B)	

(c) Pon un ejemplo de planificación de las dos transacciones para que sea legal.

T1	T2
Bloquear-X(B)	
Leer(B)	
B:=B-50	
Escribir(B)	
Bloquear-X(A)	
Leer(A)	
A:=A+50	
Escribir(A)	
Desbloquear(A)	
Desbloquear(B)	
	Bloquear-C(A)
	Leer(A)
	Bloquear-C(B)
	Leer(B)
	Desbloquear(B)
	Desbloquear(A)
	Visualizar(A+B)

(d) ¿Es tu planificaci´on secuenciable en cuanto a conflictos? ¿por qu´e?

Sí, porque es secuencial. Además, si realizasemos el grafo de secuencialidad, hay una única arista de T1 a T2, pero ninguna de T2 a T1, así que no hay ciclo.

(e) Propón alguna planificación legal para el protocolo en dos fases que no sea secuenciable en cuanto conflictos.

No es posible. El protocolo de bloqueo en dos fases asegura la secuencialidad en cuanto a conflictos, así que cualquier planificación que sea legal bajo este protocolo es secuenciable en cuanto a conflictos. [1]

[1] Frases do libro: Se dice que un protocolo de bloqueo asegura la secuencialidad en cuanto a conflictos si, y solo si, todas las planificaciones legales son secuenciables en cuanto a conflictos.Se puede demostrar que el protocolo de bloqueo de dos fases asegura la secuencialidad en cuanto a conflictos (pp. 311). tremendísima puta merda de pregunta

Tratamiento de interbloqueos

- Un sistema está en estado de interbloqueo si existe un conjunto de transacciones $[T_1 \ldots T_n]$ tal que cada instrucción T_i está esperando a un elemento que posee T_{i+1}.
    - En esta situación ninguna puede progresar.
- Es posible intentar prevenir los interbloqueos o permitir que ocurran y luego recuperarlos. Ambos pueden provocar un retroceso de transacciones.

Prevención de interbloqueos

- **Primera opción:** Asegurar que no haya esperas cíclicas ordenando las peticiones de bloqueo o exigiendo que todos se adquieran juntos.
    - Un esquema posible es bloquear todos los elementos que use una transacción antes de ejecutarla. Si no se puede bloquear alguno no se bloquea ninguno.
        - Esto produce muy poca utilización de datos y es difícil predecir los elementos.
    - Otro esquema es definir un orden total de elementos de datos, y cuando una transacción ha bloqueado elementos de un orden no puede bloquear a otros que lo preceden.
        - Esto es fácil de implementar pero también requiere conocer el conjunto de datos a los que accede la transacción.
- **Segunda opción**: Realizar retrocesos de transacciones en lugar de esperar a un bloqueo que potencialmente pueda causar un interbloqueo.
    - Cuando T_j solicita un bloqueo poseido por T_i, el bloqueo concedido a T_i puede **expropiarse** retrocediendo T_i y concediéndoselo a T_j.
    - Para controlar la expropiación se asigna una marca temporal a cada transacción.
        - El esquema esperar-morir no usa expropiación. Si T_j solicita lo poseído por T_i, puede esperar solo si T_j es anterior a T_i. Si no, T_i retrocede.
        - El esquema herir-esperar usa expropiación.. Mismo caso, pero si T_j no es anterior a T_i, T_j **hiere** a T_i y esta retrocede.
    - Otro enfoque: bloqueo con límite de tiempo. Si espera demasiado tiempo la transacción retrocede. Fácil de implementar, pero es difícil decidir el tiempo.

Ejemplo - esquemas de expropiación (2023-2024)

T1	T2
Bloqueo-X (B)	
Leer(B)	
B:=B-50	
Escribir(B)	
	Bloqueo-C (A)
	Leer(A)
	Bloqueo-C (B)
Bloqueo-X (A)	

- **¿Qué ocurre si utilizamos el esquema esperar-morir y qué pasa si utilizamos herir-esperar?**
- Si usamos esperar-morir, cuando T2 solicite el bloqueo sobre B;
    - Si T2 tiene una MT previa a la de T1, se quedará esperando hasta que T1 libere el bloqueo.
        - En este caso, T1 continuará con su ejecución hasta llegar al punto donde necesita un bloqueo sobre A.
        - Pero T2 ya tiene un bloqueo sobre A, y establecimos que T2 es previa a T1. Entonces, T1 muere y sus cambios se retroceden. Inmediatamente libera su bloqueo sobre B y T2 puede continuar.
    - Si T2 tiene una MT posterior a la de T1, cuando solicite el bloqueo sobre B morirá instantáneamente, y T1 continuará su ejecución.
- Si usamos herir-esperar, cuando T2 solicite el bloqueo sobre B:
    - Si T2 tiene una MT previa a la de T1, herirá a T1, obligándola a retroceder. T2 adquiere el bloqueo que necesita y continúa con su ejecución.
    - Si T2 tiene una MT posterior a la de T1, cuando solicite el bloqueo tendrá que quedarse esperando.
        - En este caso, T1 continuará con su ejecución hasta llegar al punto donde necesita bloquear A.
        - Pero T2 ya tiene un bloqueo sobre A, y establecimos que T2 es posterior a T1. Entonces, T1 herirá a T2 y la obligará a retroceder. T1 adquiere su bloqueo y continúa con la ejecución.
- Se demuestra que, en cualquier caso, una de las dos transacciones retrocederá para evitar un interbloqueo.

Detección

- Los interbloqueos se pueden describir con un **grafo de espera** dirigido, siendo los vértices las transacciones, y las aristas indican qué transacción espera a cuál.
    - Existe un interbloqueo si el grafo contiene un ciclo.
- El tiempo que tarda en ejecutarse el algoritmo de detección dependerá del tiempo que haya entre interbloqueos y del nº de transacciones afectadas por el interbloqueo.

Recuperación

- **Selección de una víctima:** Escoger la transacción que se va a retroceder.
    - Se escoge aquella cuyo retroceso implique un coste mínimo, dependiendo del cómputo que lleve hecho la transacción, el nº de elementos y cantidad ed atos que usa, el nº de transacciones que se vean involucradas por su retroceso…
- **Retroceso:** Determinar hasta donde se retrocederá en la transacción: no siempre es necesario un retroceso total.
    - Dependerá de cuáles bloqueos se deban liberar para romper el interbloqueo.
- **Inanición:** Ocurre si se escoge muchas veces a la misma transacción como víctima. Entonces, se debe asegurar que cada transacción solo se escoge un número finito de veces.

Protocolos basados en validación

- Una alternativa a los protocolos de bloqueos, que son pesimistas (asumen que puede haber interferencias) es un protocolo **optimista**, que asume que no va haber conflictos y luego valida (comprueba si se puede confirmar la operación de forma secuencial)
    - Los protocolos basados en validación son mejores que los de bloqueos en casos donde hay pocos conflictos, como si la mayoría de transacciones son lecturas.
- El **protocolo de validación** requiere que cada transacción se ejecute en dos o tres fases diferentes, en orden:
    - Fase de lectura: Se ejecuta la transacción T_i sobre datos locales sin actualizar la bdd.
    - Fase de validación: Se realiza una prueba de validación para determinar si puede pasar a la fase de copiar a la bdd sin causar una violación de la secuencialidad. Si no, se aborta.
    - Fase de escritura: Si T_i satisface la prueba de validación, escribe lo que tenga que escribir en la bdd. Si es de lectura sólo se salta esta fase.
- Para realizar la prueba de validación se deen conocer tres instantes temporales de cada T_i: Inicio(T_i), Validacion(T_i) (fin de fdl) y Fin(T_i) (fin de fde)
    - Se determina el orden de secuencialidad ordenando según la marca temporal
    - MT(T_i) = Validación(T_i)
- **Comprobación de validación** de T_i: Para toda T_k con MT(T_K)<MT(T_i) se debe cumplir una de las dos:
    - Fin(T_k) < Inicio(T_i)
    - Fin(T_k) < Validación(T_i), y todos los elementos de datos que escribe T_k no tiene intersección con los datos que lee T_i,

- Este esquema se denomina **control de concurrencia optimista,** ya que en lugar de forzar una espera cuando hay un retroceso como ordenación por marcas temporales o bloqueos, asumen que las transacciones se ejecutarán y luego se validan.
    - Evita los retrocesos en cascada, ya que las escrituras reales solo ocurren después de que la escritura se haya comprometido.
    - Sin embargo, si hay varias transacciones cortas que fallan, pueden producir que una transacción larga se reinicie muchas veces. Para solucionarlo se bloquean las cortas.

Aislamiento de instantáneas

- Cada transacción obtiene una instantánea de la base de datos y trabaja sólo sobre ella. Existe conflicto en cuanto a las transacciones concurrentes de actualización.
- Bajo el esquema **primer compromiso gana**, cuando una T entra en parcialmente comprometida pero otra transacción concurrente ya ha escrito alguna actualización en los mismos elementos de datos, T se debe abortar.
- Bajo el esquema **primera actualización gana**, cuando T intenta actualizar un elemento de datos, debe solicitar antes un bloqueo sobre el elemento. Si el elemento ya ha sido actualizado por otra T_i, T debe abortar.
    - Si T_i está esperando y otra T_j obtiene el bloqueo, T_i debe esperar a que T_j comprometa (T_i aborta) o T_j aborte (T_i puede obtener el bloqueo, y debe volver a comprobar si el elemento ha sido actualizado).
- El aislamiento de instantáneas no asegura la secuencialidad (a diferencia del protocolo de bloqueo en dos fases o el protocolo optimista de validación). Por ejemplo, T_i lee A y B y modifica A, y T_j lee A y B y modifica B. Ambas modifican datos distintos, pero su grafo de precedencia tiene un ciclo (atasco de escritura).
- Añadiendo la sentencia for update al final de una consulta SQL, el sistema la tratara como si fuese una operacion de escritura a la hora del control de concurrencia. Esto evita las anomalías que pueda producir lo previo, aunque de por si no son tan problemáticas en la mayoría de casos.

Inserción y borrado

- **Borrado:** Si I_i es borrar(Q) y I_j es concurrente a I_i, sin importar el tipo de operación de I_j, estará en conflicto con I_i.
    - En el protocolo de bloqueo de dos fases, para borrar se necesita un bloqueo **exclusivo.**
    - En el protocolo de marcas temporales, se rechazará T_i (borrar(Q)) si MT-L(Q) **o** MT-E(Q) son mayores que MT(T_i). EN caso contrario se puede ejecutar el borrar.
- **Inserción:** Se puede tratar de forma similar a escribir.
    - Requiere, igual que escribir(Q) un bloqueo exclusivo sobre el nuevo Q creado.
    - Si T_i realiza insertar(Q), se fijan los valores MT-L(Q) y MT-E(Q) a MT(T_i)

Fenómeno fantasma, bloqueos

- En ocasión, una inserción entra en conflicto con una lectura sobre una tupla que aún no existe en el sistema porque es parte de la inserción. Esto se denomina **fenómeno fantasma** y puede llevar a que el sistema no detecte e impida planificaciones no secuenciables.
- El **protocolo de bloqueo de índices** convierte las apariciones del fenómeno fantasma en conflictos de los bloqueos sobre los nodos hoja índice.
    - Toda relación debe tener al menos un índice. Se usa un árbol de tipo B+.
    - Si una transacción está realizando una búsqueda, requiere un bloqueo en modo compartido sobre todos los nodos hojas del índice que lee.
    - Cuando una transacción modifica, inserta, o elimina datos en la tabla, debe actualizar todos los índices afectados. Para esto, obtiene un bloqueo exclusivo en los nodos hojas relevante.
    - Además de bloquear los índices, las tuplas en si se bloquean de forma normal. Se sigue el bloqueo en dos fases.
- El **protocolo de bloqueo de predicado** no se usa en la práctica porque es más costoso. Consiste en bloquear las tuplas que cumpan una condición determinada (sueldo>9000)

Control de concurrencia en interacciones de usuario

- Las transacciones en la práctica pueden implicar entrada del usuario en distintos puntos.

- Por ejemplo, si se tienen en cuenta todos los pasos entre que el usuario ve la lista de asientos disponibles hasta que finalmente selecciona uno como una única transacción, usar el bloqueo sería muy ineficiente.
- Los protocolos de marca de tiempo evitan bloqueos, pero pueden llegar a abortar transacciones que no tienen conflictos. Ejemplo: si un usuario selecciona el asiento 1 y otro que entra más tarde selecciona el A2, podría abortarse su transacción porque modifica la misma tabla que el primer usuario, aunque no haya conflicto.
- El **aislamiento de instantáneas** es una alternativa mejor y no tiene el mismo problema, preo puede consumir muchos recursos.
- Una solución alternativa es separar la transacción en sus pasos de leer y confirmar, pero se debe volver a leer al confirmar para asegurar que no haya errores.
- **Control de concurrencia optimista con nº de versión:** añade un nº de versión a cada tupla en la base de datos.
    - Cuando un usuario actualiza una tupla, aumenta su número de versión. Entonces, si otro usuario intenta modificar esa misma tupla (reservar su asiento), observará que el nº de la tupla es mayor que el que tiene el usuario en su sistema local porque otro usuario lo aumentó. En este caso se aborta la transacción.
    - Se denomina optimista porque no realiza bloqueos.
    - Sólo valida que pueda hacer sus cambios sin sobreescribir otros, no se preocupa sobre si lo que leyó está actualizado.

Sistema de recuperación

335-342, 345

Clasificación de los fallos

- **Fallo en la transacción:** Puede ser un error lógico (entrada incorrecta, datos no encontrados) o del sistema (bloqueo). Sólo en el segundo caso se puede reintentar la trans.
- **Caída del sistema:** Causa la pérdida de la memoria volátil.
    - Supuesto de fallo-parada: se asume que el contenido de la memoria no volátil no se corrompe, gracias a los mecanismos de hardware y software de seguridad.
- **Fallo de disco:** Se soluciona con archivos de seguridad y almacenamientos secundarios.

Almacenamiento y bloques

- Para asegurar la estabilidad del almacenamiento se deben utilizar múltiples sistemas de almacenamiento no volátiles con modos de fallo independientes.
    - Como medida de seguridad adicional, se pueden guardar estos sistemas adicionales en lugares remotos para asegurar contra desastres físicos.
- La base de datos se divide en unidades de longitud fija denominadas **bloques**.
    - Los bloques del disco se denominan **bloque físicos**, y los que residen temporalmente en la mem. principal son **bloques de memoria intermedia.**.
    - entrada(B) transfiere el bloque físico B a la mem. principal
    - salida(B) transfiere el bloque de memoria intermedia B al disco
- Un fallo en una transferencia de datos es **parcial** si se llega a enviar información incorrecta y **total** si se produce tan pronto que el bloque de destino permanece intacto.
    - Se guardan dos bloques físicos por cada bloque lógico. Se actualiza primero un bloque y luego el segundo. Si hay un error durante la transacción, se restaura el valor del primer bloque con el del segundo.
- Si se ejecuta una operación que utilice el elemento X y el bloque B_x que contiene X no está en mem., se ejecuta primero entrada(B_x)
- La operación salida(B_x) no tiene que ir necesariamente inmediatamente después de ejecutar escribir(X), ya que el bloque puede contener más elementos que X.

Recuperación y atomicidad

Registro histórico !

- Secuencia de registros que almacena todas las actividades de actualización de la bdd.
    - Permite deshacer todos los cambios de una transacción si se realizaron algunos antes de causar un error.
    - La creación del registro precede a una modificación de la base de datos.
- Cada registro de actualización del registro histórico almacena identificadores de la transacción y del elemento de datos, además del valor anterior y nuevo del dato ($<T_i, X_j, V_1, V_2>$)
    - Existen otros registros especiales para identificar sucesos como el inicio, compromiso o aborto de una transacción.
- Debe residir en almacenamiento estable. Ocupa mucho almacenamiento.
- La transacción está comprometida una vez su registro histórico está en memoria estable.

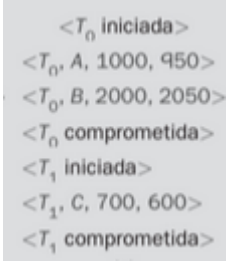
Modificación de la base de datos

- Una transacción se puede separar en tres etapas: cálculos internos, escritura en memoria intermedia y finalmente escritura en disco. Las dos últimas se consideran modificaciones de la base de datos.
- Si una transacción no modifica la bdd hasta estar comprometida, usa **modificación diferida**
- Si la modificación se produce mientras la transacción está activa, usa **modificación inmediata.** Los algoritmos de recuperación que se verán la admiten.

Control de concurrencia y recuperación

- Si dos transacciones modifican un elemento de datos simultáneamente (antes de que una de ellas comprometa), deshacer una de ellas por recuperación requeriría revertir el elemento al valor previo, deshaciendo las dos en lugar de sólo una.
- Esto se puede evitar usando un bloqueo de dos fases estricto.
- Tanto la modificación inmediata como la diferida son válidas con el bloqueo en dos fases.

Rehacer y deshacer



- Sea T_0 una transacción que transfiere 50€ de A a B, y T_1 una que resta 100€ de C. El registro histórico completo sería (imagen).
- **rehacer(T_i)** establece el valor de todos los elementos modificados por T_i a los valores nuevos (derecha).
    - Se debe mantener el orden de ejecución de las transacciones.
    - Una transacción debe rehacerse si el registro contiene su registro de 'iniciada' y también el de 'comprometida' o 'abortada'.
        - en realidad, cuando se 'rehace' una transacción abortada, los cambios que se quedarán al final serán los de su operación 'solo-deshacer' (ver abajo).
- **deshacer(T_i)** restaura el valor de todos los elementos de datos actualizados por T_i a los valores previos (izquierda)
    - También escribe registros del registro histórico para esta restauración. Estos registros **solo-deshacer** no almacenan el valor anterior de los datos actualizados.
    - Una transacción debe deshacerse si el registro contiene su registro de 'iniciada' pero no el de 'comprometida' ni 'abortada'.
    - Cuando se completa la operación deshacer, se escribe un registro <T_i aborta>.

Puntos de revisión

- Cuando ocurre un fallo se debe consultar el registro histórico para determinar cuáles transacciones rehacer y cuales deshacer, lo cual consume tiempo.
- Para solucionar esto, periódicamente se realiza una escritura en almacenamiento estable de todo el registro histórico en memoria junto con una escritura en disco de todos los bloques modificados en memoria, y esta revisión se marca con un registro histórico **<revisión L>**
    - Siendo L una lista de transacciones activas en ese punto.
    - Mientras la revisión está en curso ninguna transacción puede realizar cambios.
        - Un punto de reivsión difuso si permite estas transacciones.
- Cuando se produce un fallo, se ira al último registro <revisión L>, y las operaciones de rehacer y deshacer se aplicarán solo a transacciones de L y a transacciones iniciadas después de este registro.
    - Todas las transacciones de dentro de L (que estaban en proceso cuando se cayó el sistema, pero no se comprometieron) serán revertidas y abortadas cuando se recupere el sistema (no se continúa con su ejecución).

Algoritmo de recuperación

Retroceso de transacciones sin fallo del sistema

- El algoritmo descrito funciona para planificaciones sin retrocesos en cascada.
- Para retroceder una transacción se realizan los siguientes pasos:
    - Se explora el registro desde atrás, y para cada registro T_i de la forma $<T_i,X_j,V_1,V_2>$:
        - Se escribe V_1 en X_j
        - Se escribe un registro **solo-deshacer**[2] $<T_i,X_j,V_1>$
    - Se deja de explorar al encontrar un $<T_i$ iniciada>, y se escribe un $<T_i$ abortada>

Recuperación tras fallo del sistema

- Se divide en dos fases. En la **fase rehacer**, se vuelven a realizar modificaciones de todas las transacciones hacia delante desde el último punto de revisión.
    - Se identifica además la lista de transacciones que hay que retroceder. Estas son inicialmente las de la lista L del registro <revisión L>, y se le van añadiendo todas las iniciadas, pero las que se encuentran abortadas o comprometidas se eliminan de la lista.
        - Es decir, se deben retroceder las que estaban en proceso en la revisión y también las que iniciaron pero no comprometieron/abortaron.
    - Cuando se encuentra un registro normal o 'solo-deshacer', se escribe V_2 (o V_1, en el caso de sólo-deshacer) en X_j.
- En la **fase deshacer** se retroceden todas las transacciones que se hayan identificado en la fase previa.
    - El recorrido se realiza recorriendo el registro hacia atrás, desde el final.
    - Cuando se encuentra un registro histórico perteneciente a una transacción de la lista-deshacer se retrocede según los pasos descritos previamente.
    - La fase termina cuando el sistema haya encontrado los $<T_i$ iniciada> de todas las T de la lista-deshacer.

---

[2] no libro intercambian 'solo-rehacer' e 'solo-deshacer' como se fosen o mismo. tiven que mirar aquí (ARIES é o algoritmo de recuperacion especifico que mencionan no libro) para ver a diferencia, pero polo que entendo, ambos son basicamente registros donde sólo se garda o valor final do dato. a unica diferencia é que os solo-deshacer usanse para marcar un cambio que se desfixo durante a recuperacion e os solo-rehacer usanse, opcionalmente, para marcar un cambio que se refixo pero que xa está confirmado na bdd, ou como sustituto de un registro de actualizacion normal cando non che interesa gardar o valor previo para aforrar memoria. asi que é distinto o motivo polo que se crean, pero funcionalmente son iguales

Ejemplo - Recuperación tras fallo del sistema (4, Junio 2016)

- **Teniendo en cuenta la planificación de la figura (primeras dos columnas), describe los valores de los elementos A y B en la base de datos (columnas añadidas a la derecha), y el contenido del registro histórico antes y después de la realización de un punto de revisión en el instante 7, utilizando modificación inmediata.**

Asumimos valores iniciales A=50 y B=50. Por usar modificación inmediata, los cambios se producen al ejecutar 'escribir', no al ejecutar 'commit'.

T1	T2	A	B
Leer(B)		50	50
	Leer(A)	50	50
B:=B-50		50	50
Escribir(B)		50	0
	A:=A*2	50	0
	Escribir(A)	100	0
	Visualizar(A)	100	0
	Commit	100	0
Leer(A)		100	0
A:=A+50		100	0
Escribir(A)		150	0
Visualizar(A+B)		150	0
Commit		150	0

Asumimos valores iniciales A=50 y B=50. Por usar modificación inmediata, los cambios se producen al ejecutar 'escribir', no al ejecutar 'commit'. En cuanto al registro histórico:

Antes del punto de revisión	Después del punto de revisión
[Inicio T1] [Inicio T2] [T1, B, 50, 0] [T2, A, 50, 100]	[Inicio T1] [Inicio T2] [T1, B, 50, 0] [T2, A, 50, 100] [Revisión T1, T2]

Lo único que hace el punto de revisión es marcar, en un momento dado, las transacciones que están activas y no comprometidas aún en la base de datos.

- **¿Qué tenía que hacer el sistema para recuperarse si hay un fallo en el instante 12?**

Tras la caída del sistema, el registro histórico queda así:

```
[Inicio T1]
[Inicio T2]
[T1, B, 50, 0]
[T2, A, 50, 100]
[Revisión T1, T2]
[T2 Commit]
[T1, A, 100, 150]
```

El sistema va al último punto de revisión (asumimos que fue el del instante 7) y observa la lista L de transacciones que estaban en proceso [T1, T2].

Comienza la fase rehacer desde este punto de revisión (instante 7) hacia delante. Desde este punto, va rehaciendo todos los cambios que encuentre.

- **Encuentra [Commit T2]**: Descubre que T2 ya fue comprometida y sus cambios se deben mantener. Elimina T2 de la lista de transacciones a deshacer, dejando sólo [T1].
- **Encuentra [T1, A, 100, 150]:** Rehace el cambio que se ha encontrado, ejecutando otra vez esta actualización

Llega al final del registro histórico y comienza con la fase solo-deshacer. En todo el registro histórico no se encontró el commit de T1, por lo que los cambios de T1 se deben deshacer.

Comienza a leer el registro desde el final.

- **Encuentra [T1, A, 100, 150]:** Deshace el cambio de T1. Escribe un registro solo-deshacer **[T1, A, 100].**
- **Encuentra [T1, B, 50, 0]**: Deshace el cambio de T1. Escribe un registro solo-deshacer **[T1, B, 50].**
- **Encuentra [Inicio T1]:** Ya ha deshecho todos los cambios de T1. Escribe un registro **[T1 abortada]** para indicar esto. Retira T1 de la lista de transacciones a deshacer.

No hay más transacciones en la lista: se han acabado de corregir los cambios.

- **¿En qué estado quedaría la base y el registro antes y después de la recuperación?**

Antes de la recuperación, A tiene el valor 150 y B el valor 0.

Después de la recuperación, observamos el valor resultante que guardan los registros solo-deshacer. A tiene el valor 100 y B el valor 50 (se han cancelado los cambios de T1).

En cuanto al registro histórico:

Antes de la recuperación	Después de la recuperación
[Inicio T1] [Inicio T2] [T1, B, 50, 0] [T2, A, 50, 100] [Revisión T1, T2] [Commit T2] [T1, A, 100, 150]	[Inicio T1] [Inicio T2] [T1, B, 50, 0] [T2, A, 50, 100] [Revisión T1, T2] [T2 Commit] [T1, A, 100, 150] [T1, A, 100] [T1, B, 50] [T1 Abortada]

Fallo con pérdida de almacenamiento no volátil

- Para protegerse de fallos en los que se pierda información de disco, se podrái **volcar** periódicamente la base de datos en un almacenamiento más estable, como cintas magnéticas.
- Requiere que ninguna transacción esté activa durante el volcado, similar a la revisión.
    - Para la recuperación de datos se restituye la bdd utilizando el último volcado realizado.
    - Se consulta el registro histórico y se rehacen todas las transacciones que se hubieran comprometido después de ese volcado.
- Se puede volcar el contenido de la bdd (**volcado de archivo**) o usar **volcado SQL:** escribe sentencias SQL DDL y insert en un archivo, que se pueden ejecutar para reconstruir la base de datos.
- Existe también volcado difuso, similar a la revisión difusa.

Almacenamiento y estructura de archivos

206-212

Organización de los archivos !

- Una base de datos se hace corresponder con un número de **archivos**.
- Los archivos se dividen lógicamente en secuencias de registros, los cuales se corresponden con los **bloques** del disco.
- Bloques: Unidades de tamaño fijo en las que se dividen los archivos. Normalmente 4-8kb
    - Un bloque puede contener varios registros, pero un registro nunca es mayor que un bloque, y cada registro está completo en un único bloque.
    - Se comenzará estudiando archivos con registros de longitud fija.

Registros de longitud fija

```
type profesor = record
        ID varchar (5);
        nombre varchar(20);
        nombre_dept varchar (20);
        sueldo numeric (8,2);
    end
```

- Dado el registro (pseudocódigo) de la imagen, supongamos que cada char ocupa un byte y que el numeric ocupa 8. Entonces podemos fijar el tamaño del registro 'profesor' a 53 bytes.
- Esto significaría que los primeros 53 bytes del archivo corresponden al registro 1 y así sucesivamente, pero causa problemas.:
    - Si el tamaño del bloque no es múltiplo de 53, será necesario guardar un registro parcialmente en dos bloques → requiere dos accesos a bloque para leer/escribir
        - En lugar de hacer esto se deja libre el espacio al final de bloque
    - Si se borra un registro, hay que rellenar el espacio.
        - En lugar de hacer esto se usan cabeceras de archivo: el archivo indica la direccion del primer registro borrado, y esa ubicacion de memoria guardará la ubicación del proximo registro borrado. Así, quedan marcadas las posiciones que tienen registros borrados y son inutilizables para lectura. Esto forma una lista enlazada de posiciones libres (**lista libre**)
        - Esta posición de la cabecera también será donde se inserte el primer dato nuevo que busque espacio libre.

Registros de longitud variable

- Son utilizados si el archivo contiene tipos de datos distintos, si los registros tienen campos de longitud variable o si permiten campos repetidos (arrays, multiconjuntos)
- El **registro** se divide en dos partes:
    - Parte fija: Para los atributos de tamaño fijo (ints, dates) guarda su valor. Para los de tamaño variable guarda un nº de offset (posición respecto al inicio del registro donde empieza el dato variable) y de longitud.
    - Parte variable: Valor de los atributos de tamaño variable.
- Cada **bloque** contiene una cabecera, que contiene:
    - Nº de elementos del registro de la cabecera
    - Final del espacio vacío del bloque
        - Los registros se almacenan de forma contigua en el bloque, empezando por el final. El espacio libre será entre el último registro real del array y el final de la cabecera del bloque.
    - Un array que contiene la ubicación y tamaño de cada registro
- Los datos demasiado grandes para caber en un bloque no se guardarán en registros sino en archivos especiales. La mayoría de sistemas impiden que un registro sea más grande que un bloque.

Organización de (conjuntos de) registros en archivos

- **Organización de archivos en montículos:** Cualquier registro se coloca, desordenado, en cualquier parte del archivo.
- **Organización secuencial de los archivos:** Los registros se guardan en orden secuencial, según su clave de búsqueda.
    - La clave de búsqueda es un atributo o conjunto de atributos cualquiera, no tiene por que ser clave primaria.
    - El puntero de cada registro indica la ubicación del próximo registro según orden de cdb
    - Al insertar un dato, se localiza el registro del archivo que lo precede según orden de cdb.
        - Si dentro de su mismo bloque hay algún registro vacío se inserta ahí. De lo contrario, se insertará en un bloque de desbordamiento.
        - En ambos casos se actualizan los punteros.

10101	Srinivasan	Informática	65000
12121	Wu	Finanzas	90000
15151	Mozart	Música	40000
22222	Einstein	Física	95000
32343	El Said	Historia	60000
33456	Gold	Física	87000
45565	Katz	Informática	75000
58583	Califieri	Historia	62000
76543	Singh	Finanzas	80000
76766	Crick	Biología	72000
83821	Brandt	Informática	92000
98345	Kim	Electrónica	80000

        - Una vez se acumulan demasiados datos en bloques de desbordamiento, se reorganiza por completo el archivo para restaurar la secuencialidad, lo cual es costoso.

- **Organización asociativa (hash) de los archivos:** Se calcula una función hash para algún atributo de cada registro, que indica la posición donde se guarda.
- **Organización de archivos en agrupaciones de varias tablas:** Consiste en almacenar registros de diferentes tablas (si están relacionadas entre si, como con una FK) dentro del mismo archivo o bloque.
    - En muchas implementaciones se guarda cada relación en un archivo, lo cual para bases pequeñas simplifica el código porque las tuplas de cada relación son de un tamaño fijo.
    - Pero para bdd's de mayor tamaño puede ser mejor una implementación más compleja.
        - Generalmente se pueden guardar distintas relaciones en el mismo archivo, pero también es posible guardarlas en el mismo bloque.
        - Esto mejora el rendimiento de algunas transacciones (las que usan join para leer las distintas relaciones que están en el mismo bloque) pero empeora el de otras (las de selección de muchas tuplasd e una única relación)

Diccionario de datos

- Hasta ahora se ha estudiado el almacenamiento de las tuplas, pero es también necesario almacenar los **metadatos** de las relaciones (nombre, esquema, dominio, vistas, restricciones de integridad)
    - También nombre de usuarios autorizados, autorizaciones, contraseñas…
    - También información estadística, de almacenamiento y de organización de las tuplas.
- La implementación exacta depende del diseñador del sistema.

Gestor de la memoria intermedia

- Se establece un nivel de memoria intermedia al que se accede primero que al disco, de forma similar que una caché.
- La mayor parte de sistemas de bdd's utilizan la estrategia **LRU** (least recently used) para reemplazar.
    - Sin embargo, en una bdd se pueden predecir los futuros accesos con mayor precisión que en un SO.

    - Se denomina como **extracción inmediata** cuando se elimina de la memoria intermedia un bloque prematuramente porque se predice que no se utilizará.
    - En un caso en el que se va a iterar por todos los bloques de una relación en orden, en realidad el que haya sido usado hace más tiempo es el primero que probablemente volvamos a utilizar. Entonces sería mejor utilizar MRU a LRU para desalojar.
- Para mejorar la recuperación en caso de caída, los bloques que están siendo actualizados en caché se encuentran **clavados**, y mientras lo están no pueden escribirse en disco.
- Cuando se debe escribir en disco un bloque aunque no sea por falta de espacio en caché se denomina **salida forzada** del bloque.

Indexación y asociación

230-241

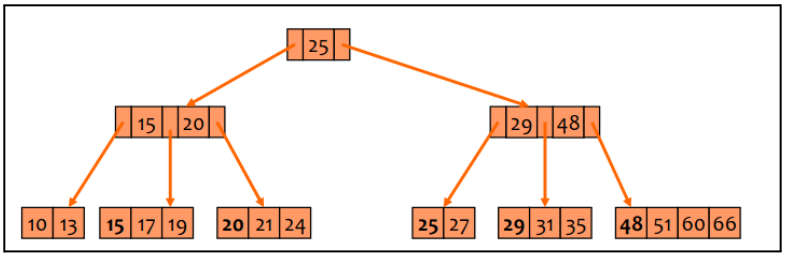
Indexación

Árboles B+

- Todas las claves están representadas en las hojas. Los nodos internos solo sirven como guías para la búsqueda.
- Las hojas están **vinculadas**, permitiendo una trayectoria secuencial rápida de las claves del árbol.
- Todos los caminos desde la raíz hasta cualquier dato son de la misma longitud.
- Ocupan más espacio porque algunas claves están duplicadas entre hojas y nodos internos (si son necesarias para definir los caminos de búsqueda).
    - Sin embargo, evitan la operación de reorganización del árbol, que es costosa.

Árboles B+ (definición)

- Siendo un árbol de orden m, cada página excepto la raíz contiene entre m/2 y m-1 elementos.
    - La raíz puede tener entre 1 y m-1 elementos. Debe tener al menos 2 descendientes.
- Todas las hojas están al mismo nivel.
- Las páginas internas contienen un vector de claves (índices)

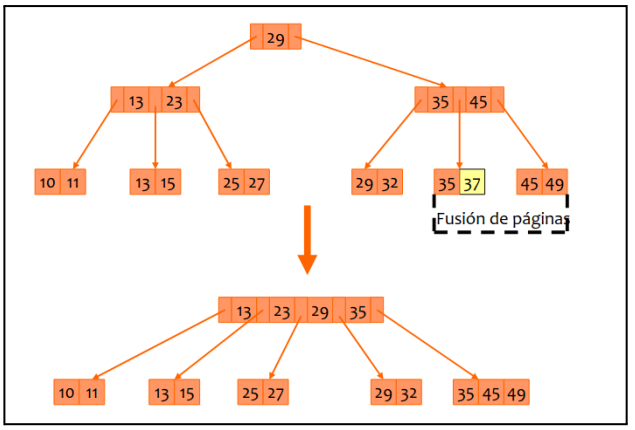


Inserción en árboles B+

- Al insertar en una página llena, se divide en dos, con las n/2 primeras claves en la mitad izquierda y las demás en la derecha.
- Una copia de la clave mediana sube a la antecesora. Si esta está llena, se repite el proceso. Sin embargo, el desbordamiento de una página que no es hoja no crea más duplicidad de claves.

Eliminación en árboles B+

- Siempre se elimina de las hojas. Si la hoja no queda por debajo del mínimo, termina la operación y no es necesario actualizar las antecesoras.
- Si queda por debajo del mínimo es necesario reestructurar las claves fusionando hojas.
    - También se deben modificar las hojas interiores, pudiendo reducir la altura del árbol.
- Al eliminar una hoja, no desaparece este elemento de los nodos interiores si sigue sirviendo como separador correcto.



Índices secundarios y reubicación de registros

- El índice primario es la estructura de datos que en base a la clave primaria de una tupla nos permite encontrar la ubicación exacta de esa tupla en disco. Se implementa con una tabla hash o con un árbol B+.
- Un índice secundario es una estructura auxiliar que cumple la misma función que el índice principal pero nos permite buscar tuplas por un atributo que no es clave primaria.
    - Al igual que el primario, mapea valores del atributo secundario que sea con una o varias ubicaciones del archivo de datos.
- Para implementar el índice secundario se puede mapear directamente un valor del atributo con uno o más punteros.
    - Sin embargo en estructuras como los arboles B+ es frecuente tener que cambiar la ubicación de registros incluso sin cambiar los valores (ej si se divide un nodo en hojas) y esto requeriría volver a mapear todos los índices secundarios
- En su lugar se puede mapear cada valor del atributo secundario con una lista de claves primarias de registros que tienen ese valor, y luego se usa el índice primario para hallar el registro.
    - Es más escalable y mejor para reubicación, pero requiere dos accesos.

Índices primarios y secundarios - ejemplo de uso

- Tenemos una tabla de 'Persona', con los datos:

id_persona (PK)	**nombre**	**Ciudad**
101	Ana	Madrid
102	Juan	Sevilla
103	Lucía	Madrid
104	Pedro	Cee

- **Consulta sobre el índice primario:** Digamos que queremos buscar id_persona = 103. Utilizamos este id como índice de búsqueda en el árbol B+ del índice primario (se va bajando de nivel en nivel, comparando 103 con la clave de cada nodo) hasta llegar a la hoja que contiene el valor.
    - Dentro de la hoja, habrá una serie de parejas de claves primarias con la dirección donde esa tupla está guardada. Buscamos el índice 103 por búsqueda lineal o binaria, y obtenemos su dirección en disco.

Índice	Dirección
101	página 12, offset 3
102	página 12, offset 4
103	página 12, offset 5

- **Consulta sobre el índice secundario**: Digamos que queremos buscar Ciudad="Madrid". Las dos opciones que vimos antes:
    - Mapear directamente valores del atributo a direcciones. Básicamente, hacer lo mismo que en el caso previo del índice primario.
        - Sin embargo, en el disco físico los datos se almacenan ordenados por su PK, no por su atributo 'ciudad'. Al llegar a la hoja donde la ciudad es 'Madrid':

Ciudad	Dirección
Madrid	página 12, offset 3
Madrid	página 12, offset 5
Madrid	página 18, offset 1

    - Las direcciones no están necesariamente contiguas. Además, si se divide el árbol B+ principal por inserción de datos, habría que acceder a este árbol secundario y actualizar sus direcciones.

- Lista de claves primarias: En lugar de guardar las direcciones de memoria, se guardan las claves primarias que se corresponden con las tuplas. Se usa esta PK para acceder al índice primario.

Ciudad	Clave
Madrid	101
Madrid	103
Madrid	109

Índices sobre cadenas de caracteres

- Si tomamos como índice un dato de tipo **varchar** nos encontramos con problemas por su tamaño variable y potencialmente grande, causando árboles de gran altura.
- Podría ser una solución la compresión del prefijo: En lugar de almacenar el nombre 'Silberchatz' guardamos 'Silb' en un nodo interno, siempre y cuando no hay otros nombres en los dos subárboles bajo esta clave que también empiecen por Silb y sean nombres distintos.

Inserción eficiente en árboles B+

- Al construir un índice sobre una relación muy grande tal que no cabe en memoria principal, las inserciones en el árbol B+ requieren muchos accesos aleaotrios al disco.
- Solución: **carga en bruto** del índice.
  - Consiste en crear un archivo temporal con las entradas del índice, ordenar por la clave de búsqueda del índice y luego insertar las entradas ya ordenadas en el árbol B+ en el orden correcto.
  - Insertarlas en el orden que corresponde lleva a operaciones de E/S secuenciales, más rápidas.
- Si el árbol está vacío, es más eficiente construírlo **de abajo hacia arriba** que insertar entrada por entrada.
  - Se ordenan las entradas por la clave del índice como en el método previo, se dividen las entradas en bloques llenos y luego se van calculando sus claves mínimas para ir subiendo.

Árboles B

- Se elimina la redundancia de almacenar valores repetidos en claves de búsqueda.

- Esto lleva a que en teoría se necesiten menos nodos del árbol, y el tiempo de acceso a un nodo puede ser más rápido si ya está en un nodo intermedio (en el B+ hay que ir siempre a hoja)
- Sin embargo, presenta inconvenientes en cuanto al borrado y la inserción por lo que B+ es mejor.

Memorias flash

- Alternativa a los discos magnéticos. Se estructuran en bloques y pueden funcionar como un árbol B+.
- Las búsquedas son mucho más rápidas, pero no permiten actualizaciones de bloques (Requieren copia+escritura de un bloque completo)

Acceso bajo varias claves

- Supongamos que queremos buscar *profesores* con un nombre_dept y un sueldo específico, y que ambos son índices. Podemos:
    - Usar el índice nombre_dept y luego examinar cuáles de los resultados tienen el sueldo
    - Hacer al revés, usando sueldo como índice y nombre_dept como criterio
    - Usar el índice de nombre_dept para encontrar punteros a los registros del departamento, luego el índice de sueldo para hallar los punteros buscados, y hacer la intersección.
    - La tercera estrategia de estas es la mejor, pero es ineficiente si hay muchos datos con el nombre buscado, muchos con el sueldo buscado, pero muy pocos en la intersección.
- Una alternativa mejor es crear y utilizar un **índice con una clave compuesta** (nombre_dept, sueldo).
    - Este índice funcionaría igualmente bien si queremos buscar sólo por nombre_dept, o si buscamos con nombre_dept='finanzas' y sueldo<8000.
    - Sin embargo, hay problemas con consultas del tipo nombre_dept<'finanzas' and sueldo<8000.
        - Ordenamos las tuplas respecto a nombre_dept, tomamos las que alfabeticamente son menores y luego buscamos las de sueldo menor a 8000, pero como estamos ordenando primero por nombre_dept, estas tuplas podrían estar en archivos distintos, llevando a muchas E/S.
- Los **índices de cobertura** almacenan, además de la clave de búsqueda, valores de otros atributos.
    - De esta forma, para la consulta previa, no sería necesario ni siquiera acceder a la relación. Buscamos por nombre_dept y ya tenemos directamente el sueldo a buscar.ç

Asociación

- La **asociación** (hashing) permite evitar el acceso a la estructura de índice, pero también proporciona una forma de construir indices.
- Se usa el término **cajón** para referirse a una unidad de almacenamiento que puede guardar uno o más registros. Puede ser:

Asociación estática

- Sea K el conjunto de todos los valores de la clave de búsqueda y sea B el conjunto de todas las direcciones de cajón. Una **función de asociación** h es una función de K a B.
    - B es un número fijo y limitado.
- Para insertar un registro con clave de búsqueda K_i se calcula $h(K_i)$, lo que proporciona la dirección del cajón para ese registro.
- La función de asociación debe distribuir los datos de forma uniforme (no todos al mismo cajón) y aleatoria (cada cajón tiene más o menos los mismos datos, sin importar la distribución real de las claves)
- Se denomina **atasco** si un cajón está desbordado cuando otros aun tienen espacio libre.
    - Se pude solventar con cajones de desbordamiento, que crea el sistema en la posición donde se quería guardar un dato pero el cajón ya está lleno. Esto puede llevar a una lista enlazada de cajones de desbordamiento (encadenamiento → asociación cerrada)
    - Por otra parte, el método de recolocación en otro cajón ya existente se denomina asociación abierta.
- Un **índice asociativo** organiza las claves de búsqueda, con sus punteros asociados, en una estructura de archivo asociativo. Esto permite crear estructuras de índice utilizando asociación.
    - Se aplica una función hash sobre la clave de búsquead para identificar un cajónm y se almacenan las claves y los punteros asociados en ese cajón.

Asociación dinámica

- La función hash y también el espacio de aplicación B se adapta dinámicamente al nº de registros. Se usa para evitar los problemas de desbordamiento y atasco.
- En este apartado se describe un tipo concreto, la asociación extensible.
    - Se escoge una función de asociación estándar que genere valores dentro de un rango amplio, como los binarios de 32 bits.
    - Se utilizan solo i bits de esos 32, y el valor de i aumenta o disminuye con el tamaño de la base de datos.

- A la hora de insertar en un cajón j, si se encuentra que está lleno, hay que dividir el cajón y redistribuir sus registros actuales. Para esto hay que determinar también si aumentar el nº de bits a utilizar.
    - Si $i=i_j$, entonces solo apunta al cajón j una entrada de la tabla de direcciones. Entonces, al dividir el cajón j en 2 será necesario aumentar en 1 el valor de i, lo que duplica el tamaño de la tabla de direcciones.
    - Si $i>i_j$, más de una entrada de la tabla apunta al cajón j, por lo que podemos dividirlo en 2 sin aumentar el tamaño de la tabla de direcciones.
    - En ambos casos solo se recalcula la función de asociación para el cajón j.

Comparación estática - dinámica (2023-2024)

- Asociación dinámica: el rendimiento no se degrada con aumentos de tamaño. No requiere reservar cajones prematuramente.
    - El coste medio de acceso se mantiene cercano a O(1)
    - Adecuado cuando se requiere un rendimiento estable para lecturas y escrituras, aunque aumente el volumen de datos.ç
- Asociación estática: no requiere un nivel adicional de indirección → implementación más sencilla
    - El coste de acceso aumenta al aumentar la carga
    - Adecuado cuando se puede predecir anticipadamente la cardinalidad de las claves e importa la sencillez de la implementación

Comparación indexación - asociación !

- Indexación: mejor para acceder a registros ordenados/en un rango de un atirbuto, mantienen orden lógico, sin colisiones, mayor uso de memoria
- Asociación: mejor para acceder a valor concreto, sin orden, coste de actualización bajo (en dinámica), produce colisiones.

Procesamiento y optimización de consultas

249-253, 269-279 (SIN REPASAR)

Procesamiento de consultas

Análisis y traducción

- Antes de nada, el sistema debe traducir SQL a un lenguaje apropiado para la representación interna de la consulta, como álgebra relacional.
    - Cada consulta SQL puede tener varias traducciones válidas.
- Además, no basta con dar la sentencia en álgebra relacional, se pueden también especificar las **primitivas de evaluación** que son las operaciones concretas a utilizar (por ejemplo, especificar si usar árboles B+ o indexación)
    - La secuencia de primitivas a ejecutar son el **plan de ejecución de la consulta**.
    - El **motor de ejecución de consultas** es el que escoge el plan, evaluando el coste de las operaciones para elegir la de menor coste.

Estimación de coste !

- El factor más importante es el coste de acceso a los datos en disco, no de procesamiento de CPU. Se usará el nº de transferencias de bloque de disco y el nº de búsquedas en disco.
- Sea t_t el tiempo de transferir un bloque de datos y t_s el tiempo de acceso a bloque. El tiempo de una operación que transfiere b bloques y ejecuta S búsquedas es de $b*t_t + S*t_s$.
    - Los valores usuales son de t_s=4ms y t_T = 0.1 ms.
    - No se incluye el tiempo de escribir en disco, y se ignora la posibilidad de que el dato esté en memoria intermedia.

Operación selección

- El **explorador de archivo** es el operador de nivel más bajo para acceder a los datos. Se trata de algoritmos de búsqueda que localizan y recuperan los registros que cumplen una condición de selección.
- **A1:** Búsqueda lineal. Se explora cada bloque del archivo y se comprueban todos los registros para determinar si satisfacen o no la condición de selección. T
    - $t_s + b_r * t_r$ (peor caso)
- **A2:** Índice primario con igualdad basada en la clave. Recupera una única tupla.

- $(h_i+1) * (t_r + t_s)$

- **A3:** Índice primario con igualdad basada en un atributo no clave. Todos los registros estarían almacenados consecutivamente debido a que el archivo se ordena por la clave de búsqueda.
  - $h_i * (t_r + t_s) + b * t_r$
- **A4:** Índice secundario con igualdad. Si son varios registros a devolver, cada registro puede estar en un bloque distinto, aumentando el tiempo de acceso. SI no, es igual a A2.
  - $(h_i + n) * (t_r + t_s)$
- **A5:** Índice primario con comparación. Para comparaciones de buscar una tupla A>v, se busca la primera que tenga A>c y sigue hasta el final del archivo. Si la búsqueda es A<v, comienza desde el principio hasta llegar a una donde A=v.
  - $h_i * (t_r + t_s) + b * t_r$ Idéntico a A3.
- **A6:** Índice secundario con comparación.
  - $(h_i + n) * (t_r + t_s)$. Idéntico a A4.
- **A7:** selección conjuntiva (AND) usando índice. Se obtienen las tuplas de una de las condiciones con uno de los previos algoritmos y luego se comprueba que cumplan el resto de condiciones.
- **A8:** Selección conjuntiva (AND) usando índice compuesto de varios atributos. Se utiliza A2, A3 o A4 sobre este índice.
- **A9:** Selección conjuntiva (AND) mediante intersección de identificadores. Se exploran los punteros a los registros que cumplan las condiciones, se obtiene la intersección de los conjuntos hy luego se recuperan los registros reales.
- **A10:** Selección disyuntiva (OR) mediante unión de identiifcadores. Mismo caso pero con la unión

Optimización de consultas !

Visión general

- Se considera la consulta ‘nombres de todos los profesores del departamento de música junto con los nombres de todas las asignaturas que enseñan’. Requiere trabajar sobre tres relaciones distintas (profesor→enseña→asignatura), produciendo una relación intermedia de gran tamaño, pero de los 10 atributos totales que cargamos, sólo dos nos son útiles.
- Además, si realizamos primero las operaciones de reunión y al final la de selección que elige solo los profesores del departamento buscado, estamos cargando muchos datos innecesarios. Se muestran abajo dos árboles de expresión distintos que representan la

misma consulta (se leen de abajo a arriba), pero el de la derecha realiza primero la selección sobre profesor.

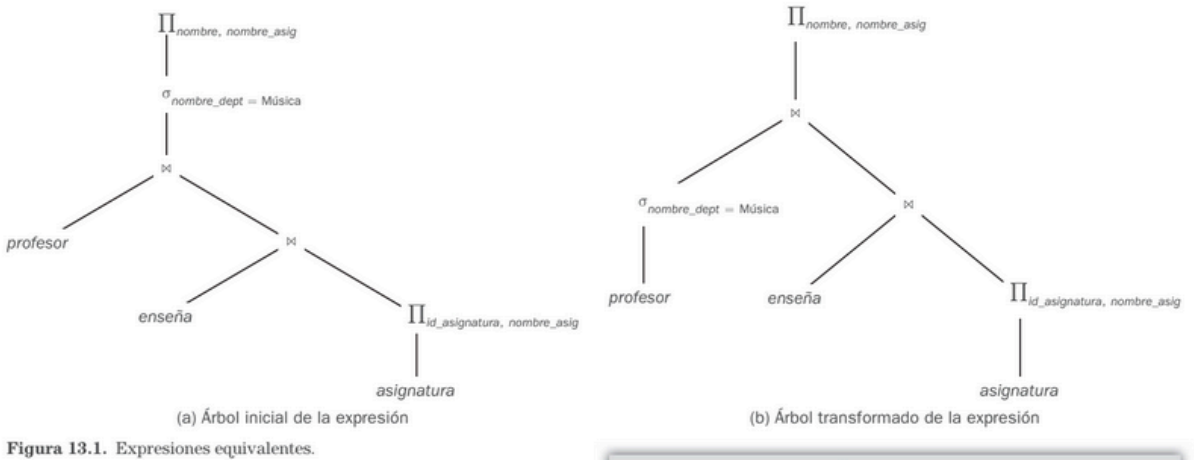
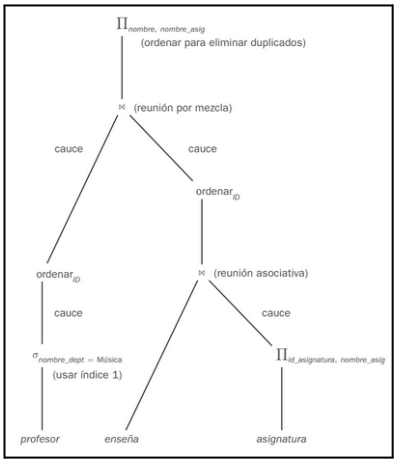


Figura 13.1. Expresiones equivalentes.

- El árbol de la izquierda representa profesor ⋈ enseña ⋈ $\Pi_{id\_asignatura,nombre\_asig}$ (asignatura), y[3] $\Pi_{nombre,nombre\_asig}$ (($\sigma_{nombre\_dept = «Música»}$ (profesor))⋈(enseña ⋈$\Pi_{id\_asignatura,nombre\_asig}$ (asignatura)))

- Un **plan de evaluación (!)** es tomar una de estas posibles expresiones y completarlas con las operaciones concretas a realizar. Por ejemplo, un posible plan de evaluación para el árbol (b) sería: (imagen)



Transformación de expresiones relacionales !

- Dos expresiones son equivalentes si para cada ejemplar legal de la bdd devolverán las mismas tuplas. El orden es irrelevante.

- Ejemplos de reglas de equivalencia:

  - $\sigma_{\theta 1 \wedge \theta 2}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(E))$
  - $\sigma_{\theta 1}(\sigma_{\theta 2}(E)) = \sigma_{\theta 2}(\sigma_{\theta 1}(E))$
  - $\Pi_{L1}(\Pi_{L2}(... (\Pi_{Ln}(E)) ...)) = \Pi_{L1}(E)$[4]
  - $\sigma_{\theta}(E1 \times E2) = E1 \bowtie_{\theta} E2$ ($\bowtie_{\theta}$ → reunión zeta)
  - $E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$
  - $(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$
  - $(E1 \bowtie_{\theta 1} E2) \bowtie_{\theta 2 \wedge \theta 3} E3 = E1 \bowtie_{\theta 1 \wedge \theta 3} (E2 \bowtie_{\theta 2} E3)$, donde θ2 implica únicamente atributos de E_2 y E_3.

[3] el de la derecha,

[4] La proyección es la operación que selecciona columnas concretas. Esto quiere decir que si haces muchas proyecciones una tras otra, sólo importa la última, que será la que tenga menos columnas.

- $\sigma_{\theta 0}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 0}(E1)) \bowtie_{\theta} E2$, donde θ0 implica únicamente atributos de una de las dos expresiones (en este ejemplo E1)
- $\sigma_{\theta 1 \wedge \theta 2}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 1}(E1)) \bowtie_{\theta} (\sigma_{\theta 2}(E2))$, donde θ1 implica únicamente atributos de E1 y θ2 de E2.
- $\Pi_{L1 \cup L2}(E1 \bowtie_{\theta} E2) = (\Pi_{L1}(E1)) \bowtie_{\theta} (\Pi_{L2}(E2))$, donde L1 es un atributo de E1 y L2 uno de E2.
- $\Pi_{L1 \cup L2}(E1 \bowtie_{\theta} E2) = \Pi_{L1 \cup L2}((\Pi_{L1 \cup L3}(E1)) \bowtie_{\theta} (\Pi_{L2 \cup L4}(E2)))$, donde L1 son atributos de E1 y L2 de E2. Además, L_3 son los atributos de E_1 que están implicados en la condición de reunión $\bowtie_{\theta}$, pero no entran en $L_1 \cup L_2$, y L_4 lo mismo para E_2.
- $(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$, $(E1 \cap E2) \cap E3 = E1 \cap (E2 \cap E3)$
- $\sigma_P (E1 - E2) = \sigma_P (E1) - \sigma_P (E2)$, y $\sigma_P (E1 - E2) = \sigma_P (E1) - E2$
- $\Pi_L(E1 \cup E2) = (\Pi_L(E1)) \cup (\Pi^L(E2))$

- En general, es recomendable:
  - Priorizar los joiins que filtren más tuplas primero.
  - Aplicar las proyecciones lo antes posible.

Catálogo, histograma

- Los catálogos de las bases de datos almacenen información estadística de las relaciones:
  - **n_r** → nº de tuplas de r
  - **b_r** → nº de bloques que contienen tuplas de r
  - **l_r** → tamaño de cada tupla de r en bytes
  - **f_r** → cantidad de tuplas de r que caben en un bloque
    - $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$
  - **V(A,r)** → nº de valores distintos en la relación r para A. Si A es clave, el valor es n_r.
- La mayoría de bases de datos almacenan la distribución de valores de cada atributo en un **histograma**.
  - Los histogramas habitualmente registran tanto el nº de valores distintos dentro de cada intervalo de valores para cada atributo como el nº de tuplas en ese intervalo.
  - Un histograma de equianchura divide los valores en intervalos de igual tamaño.
  - Un histograma de equiprofundidad divide los valores en intervalos de igual nº de valores.

- nada de esto entrou nunca nin vai entrar nunca. non sei por qué coño estou facendo apuntes. o puto sebas vai chegar e decir ‘escribeme un codigo enteiro de java compilable a boligrafo’ e vaise reir

Estimación de tamaño de selección

- **Selección con igualdad:** Si los valores de A están uniformemente distribuidos, $n_r/V(A,r)$. Sin embargo, esto no se suele cumplir.
  - En su lugar, si tenemos histogramas del atributo, lo podemos utilizar.
- **Selección con desigualdad:** siendo v el valor contra el que se compara,
  - $\frac{(v-min(A,r))}{(max(A,r)-min(A,r))} \cdot n_r$. Si no se conoce v se tomaría $n_r/2$.
  - También se pueden usar los histogramas.

Estimación de tamaño de reunión

- Si R ^ S forman una clave, el tamaño máximo será n_s o n_r.
- El caso general es $\frac{n_r * n_s}{max(V(A,r),V(A,s))}$, asumiendo una distribución uniforme de valores.

Elección de planes de evaluación

- Un **optimizador basado en el coste** explora el espacio de todos los planes de evaluación de consultas equivalentes a la consulta dada y selecciona la de menor coste estimado.
  - Explorar el espacio de todos los planes pueden ser muy costoso, por lo que se siguen heurísticas.
- **Selección del orden basada en coste:** Determina el orden een el que se realizan las múltiples reuniones de una consulta.
  - Se usa programación dinámica, guardando los mejores resultados parciales o aquellos que pueden ser interesantes, y trabajando sobre ellos para no explorar órdenes redundantes o poco interesantes.
- **Optimización basada en reglas de equivalencia:** Mejora la eficiencia de una consulta al usar reglas de equivalencia para cambiar el orden en el que se aplican las expresiones. Por ejemplo, aplicar antes las proyecciones o las reuniones que descarten más datos.