Preguntas BBDD-2

Pedro Gamallo Fernández

Curso 2018-2019

Abstract

Apuntes con los enunciados y las respuestas de las preguntas de examen de la materia de Bases de Datos II, recogidas del documento *Coñecementos* del Campus Virtual.

1 Acceso a SQL desde lenguajes de programación

1. **¿Por qué es necesario acceder a bases de datos desde lenguajes de programación de propósito general? ¿Qué dos grandes enfoques existen para realizar ese acceso? ¿Cuál es el mayor desafío a abordar en este contexto?**

   SQL proporciona un lenguaje de consultas declarativo muy potente, pero muchas veces no es suficiente para ciertas consultas algo más complejas, que sin embargo si pueden hacerse en lenguajes de propósito general. Además, hay otras acciones no declarativas (como la impresión de informes o la interacción con el usuario) que no se pueden llevar a cabo en SQL, mientras son relativamente simples de manejar desde los demás lenguajes de programación. Por estos dos motivos, principalmente, es necesario que estos lenguajes tengan acceso a las bases de datos.

   Los dos grandes enfoques a la hora de acceder a las bases de datos desde los lenguajes de programación son:

   - **SQL incorporado:** Se utiliza sintaxis SQL dentro del lenguaje *anfitrión*. Es necesario un preprocesador que sustituya las sentencias SQL por código y llamadas a funciones del lenguaje anfitrión antes de la compilación.
   - **SQL dinámico:** Se realizan consultas SQL por medio de funciones del propio lenguaje anfitrión, a las que se le pasa la sintaxis SQL como parámetro de tipo *cadena de caracteres*. Se realizan en tiempo de ejecución.

   Sin embargo, SQL y los lenguajes de programación de propósito general usan tipos de datos muy distintos (SQL opera con relaciones, mientras que los demás lenguajes lo hacen sobre los valores de las variables), y hacer que se pueda pasar de unos a otros supone el mayor desafío en este contexto. Es necesario, por tanto, algún mecanismo para que el lenguaje anfitrión pueda operar con los datos SQL tupla a tupla, obteniendo los valores de las variables que forman la tupla (columnas o atributos).

2. **¿Qué es JDBC? Conexión con la base de datos. Envío de sentencias SQL. Obtención del resultado. Sentencias preparadas. Sentencias que se pueden invocar. Metadatos. Otras características.**

   La norma **JDBC** define una API que pueden utilizar los programas de Java para conectarse a los servidores de bases de datos.

En primer lugar, es necesario abrir una conexión con la base de datos, lo cual se hace mediante la función **getConnection**, que recibe como parámetros, en primer lugar, el tipo de protocolo de comunicación que se va a usar, el nombre de la base de datos, el puerto y el esquema de la base de datos a utilizar, seguidamente el identificador del usuario y, en último lugar, la contraseña.

Para el envío básico de sentencias existen dos funciones:

- **executeUpdate:** Se emplea para ejecutar una sentencia de actualización, inserción o borrado que se pasa como parámetro a la función (como variable de tipo String).
- **executeQuery:** Se emplea para ejecutar consultas. La consulta se pasa como parámetro de tipo String y se recibe el resultado en una variable de tipo *ResultSet*, sobre la que se puede iterar para ir recogiendo los valores de los atributos de las diferentes tuplas devueltas.

La variable de tipo **ResultSet** se iguala a la función **executeQuery** para almacenar así su salida (que será la relación devuelta por la consulta). Sin embargo, en el lenguaje anfitrión interesa acceder a los valores de los datos de los atributos. Por ello, el ResultSet actúa prácticamente como un iterador, que recorre la relación resultado tupla a tupla por medio de la función **next()**, que nos indica si hay más tuplas en la relación, y de ser así, hace que el ResultSet se iguale a la siguiente. Una vez tenemos la tupla actual almacenada en el ResultSet, podemos acceder al valor de los atributos por medio de funciones de tipo **get** (*getString()*, *getInt()*...), a las que le especificamos el número de la columna a la que queremos acceder, o el nombre de la misma.

Sin embargo, la forma de enviar sentencias que se acaba de nombrar puede producir problemas (y hasta fallos de seguridad por medio de la *inyección SQL*) cuando se pasan parámetros de tipo *String* debido al uso de las comillas. Por ello, es preferible usar las *sentencias preparadas* (**PreparedStatement**), en las cuales, en vez facilitar los parámetros, se escribe un *?*, que puede ser sustituído posteriormente por cualquier valor que el usuario facilite. En estos casos, las variables de tipo *String* están protegidas, reduciendo así el problema de la *inyección SQL*. Además, los **PreparedStatement** se pueden compilar una vez y utilizarse despues con parámetros distintos, aumentando así la eficiencia.

Además, existe la posibilidad de invocar sentencias por medio de funciones y procedimientos almacenados gracias a la interfaz **CallableStatement**.

Por otra parte, **JDBC** proporciona también mecanismos para poder conocer los esquemas de la base de datos, conocer el tipo de datos de los atributos, etc. Esta información sobre los datos es lo que se llama *metadatos*, y se consigue mediante las funciones de los objetos **ResultSetMetaData** (para obtener información sobre el conjunto de resultados) y **DatabaseMetaData** (para determinar los metadatos de la base de datos).

A mayores, otras características que ofrece **JDBC** son los **conjuntos de resultados actualizables**, los cuales son tuplas de resultados que al ser actualizadas, actualizan también los valores en la base de datos, la función **setAutoCommit()**, que permite activar o desactivar el *commit* automático de las transacciones, los métodos **getBlob()** y **getClob()**, para acceder a objetos de gran tamaño y trabajar con ellos, y la clase **RowSet**, similar a **ResultSet** pero con varias características adicionales.

3. **¿Qué es ODBC? Conexión con la base de datos. Envío de sentencias SQL. Obtención del resultado. Sentencias preparadas. Metadatos. Transacciones. API ADO.NET**

La norma **ODBC** define el modo de comunicación entre los programas de aplicación y los servidores de bases de datos, por medio de una API que pueden utilizar las aplicaciones para abrir conexiones con las bases de datos, enviar consultas y actualizaciones y obtener los resultados. Es, por tanto, bastante similar a **JDBC**. Se usa la misma API para cualquier servidor, pero cada gestor debe proporcionar una biblioteca que le de soporte. Esta biblioteca se enlaza con el programa cliente para que éste pueda realizar las llamadas a la API.

Para comunicarse con un servidor hay que establecer una conexión. Este proceso requiere primeramente asignar un lugar de entorno SQL, posteriormente un manejador de la conexión, y finalmente abrir la conexión. Estos pasos, en C, se realizan con las funciones **SQLAllocEnv()**, **SQLAllocConnect** y **SQLConnection**.

Una vez creada la conexión, se pueden enviar sentencias por medio de **SQLExecDirect()**, que recibirá como uno de sus parámetros la consulta en formato de cadena de caracteres. Esta función devuelve una variable de tipo **RETCODE**, que valdrá *SQL_SUCCESS* si la sentencia se realizó con éxito.

Para obtener los resultados de una consulta se utilizan las funciones **SQLFetch** (que obtiene los resultados de la consulta tupla a tupla, de manera similar a como lo hace **next()** en **JDBC**) y **SQLBindCol** (que obtiene los valores de los atributos especificando como parámetro el número de columna).

Igual que ocurre en **JDBC**, es posible crear instrucciones SQL cuyos parámetros sean *?*. Estas instrucciones se *preparan* (**sentencias preparadas**) compilándose una vez, y pueden ser ejecutadas repetidamente sustituyendo las interrogaciones por los valores reales.

Otra característica más compartida con **JDBC** es que existen funciones que permiten hallar información sobre la base de datos y sus relaciones, los nombres de los atributos y los tipos de datos, es decir, sobre los **metadatos**.

Por defecto, cada instrucción SQL se trata como una transacción separada que se compromete automáticamente. Mediante **SQLSetConnectionOption(con, SQL_AUTOCOMMIT, 0)** (siendo *con* la variable de la conexión) podemos desactivar el compromiso automático en la conexión actual, por lo que tendremos que comprometer las transacciones explícitamente mediante **SQLTransact(con, SQL_COMMIT)** o retrocederlas con **SQLTransact(con, SQL_ROLLBACK)**.

La API **ADO.NET** es una alternativa a **ODBC** para los lenguajes Visual Basic, .NET y C#, algo más sencilla que esta y muy similar a **JDBC**. Las llamadas a **ADO.NET** se transforman en llamadas a **ODBC**.

4. **¿Qué es el SQL Incorporado? Preprocesamiento. Estructuras necesarias (SQLCA). Conexión. Variables de anfitrión en código SQL. Cursores. Procesamiento del resultado. Transacciones. Construcción de consultas en tiempo de ejecución. SQLJ.**

   El **SQL Incorporado** es un estándar SQL que define la incorporación de sentencias SQL en diversos lenguajes anfitrión (como C, Cobol, Pascal, Java, PL/I y Fortran). Los programas escritos en lenguaje anfitrión pueden usar la sintaxis de SQL incorporado para tener acceso a los datos almacenados en la base de datos y actualizarlos. De esta forma, se amplía la capacidad de los programadores de manipular las bases de datos.

   Cuando se usan sentencias de SQL incorporado es necesario que el programa sea preprocesado por un preprocesador especial antes de la compilación. Este prepocesador sustituye

las peticiones de SQL incorporado por declaraciones escritas en el lenguaje anfitrión y por llamadas a procedimientos que permiten la ejecución de los accesos a la base de datos.

Para que esto funcione, son necesarias una serie de estructuras que conforman el **SQLCA** (área de comunicación SQL). Estas estructuras permiten establecer la conexión con la base de datos y tratar los resultados obtenidos, así como obtener los mensajes de error que se puedan producir en la consulta.

Antes de ejecutar ninguna instrucción SQL se debe realizar las conexión con la base de datos. Esto se puede conseguir mediante la sentencia **EXEC SQL connect to *servidor* user *nombre-usuario* END-EXEC**. Con **EXEC SQL** y **END-EXEC** indicamos al preproccesador que dentro va una sentencia SQL que debe ser preprocesada. Los parámetros *servidor* y *nombre-usuario* se corresponden con el identificador del servidor y el nombre del usuario del mismo, respectivamente.

Una gran ventaja de este sistema es que se pueden usar variables del lenguaje anfitrión dentro de las sentencias SQL. Esto se consigue colocando simplemente *:* delante del nombre de la variable dentro del texto SQL. Por ejemplo, un ***where*** *sueldo = :cantidad* nos indica que la variable cantidad es una variable del lenguaje anfitrión, y en la sentencia SQL se cambiará esta por su valor.

Un aspecto importante es que, aunque las consultas en SQL Incoporado son bastante similares a las de SQL normal, vamos a necesitar siempre definir una variable de tipo **cursor** con la línea ***declare*** *c* ***cursor for*** antes de la instrucción **select**. Con esta variable podemos identificar la consulta a la hora de evaluarla con la función **open** y obtener los valores de una de las tuplas devueltas con la función **fetch**.

Para obtener los valores del resultado de una consulta emplearemos una sentencia del tipo ***EXEC SQL fetch*** *c* ***into*** *:var1, :var2* ***END-EXEC***. De esta forma, estamos obteniendo los valores de dos atributos de una tupla del resultado de la consulta identificada por el cursor *c* y almacenándolos en las variables del lenguaje anfitrión *var1* y *var2*. Sin embargo, de cada vez solo obtendremos los atributos de una tupla, por lo que será necesario ejecutar un bucle para poder obtenerlas todas.

La funciones nombradas anterioremente se refieren al caso del lenguaje C. Para Java existe también un esquema, aunque ya obsoleto, denominado **SQLJ**. En este, se sustituyen los cursores por iteradores que son recorridos con el método **next()** para obtener sus valores con **fetch()**.

2 Gestión de transacciones

1. **¿Qué es una transacción? Pon un ejemplo. Propiedades ACID. ¿Quién es el responsable de que se cumpla cada propiedad?**

   Una **transacción** es un *conjunto de operaciones* (de acceso y actualización de datos) que se entienden como *una única unidad lógica de trabajo*, y por tanto deben ser ejecutadas como una única unidad de ejecución. Para ello, deben escribirse las operaciones requeridas entre las marcas **begin transaction** y **end transaction**. Un ejemplo muy común de transacción es el de la *transferencia bancaria*, que para el usuario supone una única acción, pero se conforma por varias operaciones de actualización de los valores de las cuentas entre las que se efectúa.

   Para asegurar la integridad de los datos modificados por las transacciones es preciso que estas cumplan las **propiedades ACID**:

- **Atomicidad:** Asegura que o se ejecutan todas las operaciones que componen la transacción o, en caso de fallo en el sistema, no se ejecute ninguna, dando la sensación al usuario de ser una unidad simple e indivisible. Esto se consigue almacenando los valores antiguos en un registro y reestableciéndolos si no se pudieron realizar todas las operaciones. Que se cumpla esta propiedad es **responsabilidad del SGBD**.
- **Consistencia:** Asegura que tras ejecutar de forma aislada una transacción, si la base de datos era consistente antes de ella, lo seguirá siendo. La comprobación de que esto se cumple es **responsabilidad del programador**.
- **Aislamiento:** Asegura que, a pesar de que se ejecuten dos transacciones concurrentemente, una de ellas tendrá la impresión de que se ejecutó antes que la otra y viceversa. Esta característica afecta al rendimiento pero es necesaria en ciertas ocasiones. Su cumplimiento es **responsabilidad del SGBD**.
- **Durabilidad:** Asegura que, tras finalizar exitosamente una transacción, los cambios que se han hecho en la base de datos persistirán incluso aunque se produzca un fallo en el sistema. Este aspecto es **responsabilidad del SGBD**.

2. **Estructura del almacenamiento (volátil, no volátil y estable).**

   Los medios de almacenamiento se pueden estructurar en 3 grandes clases:

   - **Volátil:** La información almacenada en el almacenamiento volátil **no sobrevive a caídas y fallos del sistema**. Sin embargo, son medios de **acceso muy rápido**, tanto por la velocidad del sistema como por el hecho de que se puede acceder de manera directa a cualquier elemento de datos. Son ejemplos de este tipo las *memorias caché* y la *memoria principal*.
   - **No volátil:** La información almacenada en medios de almacenamiento no volátil **si sobrevive a caídas y fallos del sistema** pero, por contra, **son más lentos** en varios órdenes de magnitud que los medios volátiles. Además, son **subsceptibles a fallos**, lo que puede producir una pérdida de información. Ejemplos de este tipo son *los discos duros* y *las cintas magnéticas*.
   - **Estable:** En el almacenamiento estable los datos **nunca se pierden**, pero **su implementación es teóricamente imposible**. Sin embargo, **existen buenas aproximaciones** que reducen el grado de probabilidad de pérdida de datos replicando la información en diversos medios no volátiles diferentes.

3. **Explica los diferentes estados por los que puede pasar una transacción. Cuando una transacción falla, ¿qué debe hacer el SGBD y que técnica se utiliza para eso?**

   Debido a que las transacciones pueden fallar, y por tanto hay que deshacer los cambios efectuados en la base de datos, pero también pueden completarse con éxito, y en ese caso hay que asegurar la durabilidad de los datos actualizados, existen diversos estados que nos indican en que situación se encuentra una transacción en cada caso. Esos estados son los siguientes:

   - **Activa:** Es el estado inicial de las transacciones. Mientras la transacción se esté ejecutando, perdura este estado.
   - **Parcialmente comprometida:** La transacción toma este estado tras finalizar la última operación que la compone. Sin embargo, al no haberse escrito la información en el disco, la transacción todavía puede fallar.

- **Fallida:** La transacción toma este estado tras producirse un error que no le permite continuar. Se puede llegar a este estado desde una transacción activa o desde una parcialmente comprometida.
- **Abortada:** Tras haber pasado por el estado de transacción fallida, se debe retroceder la transacción y reestablecer el estado de la base de datos antes de que esta se ejecutara. Cuando todo esto ocurra, la transacción pasa a estar abortada.
- **Comprometida:** Una vez que la transacción se completa correctamente, incluyendo la escritura de datos en el disco, la transacción está comprometida, y por tanto finalizada.

4. **¿Puede abortarse una transacción comprometida (que solución se puede adoptar y quién puede adoptarla)? ¿Puede fallar una transacción que terminó la ejecución de todas sus instrucciones menos la instrucción de comprometerse? Escrituras externas observables.**

   Tras comprometerse una transacción (es decir, que esta pase al estado de comprometida), los datos ya están escritos y la base de datos se encuentra de nuevo en un estado consistente que perdura incluso si se produce un fallo en el sistema. Por ello, es imposible abortar la transacción como tal, pues esta ya finalizó por completo. La única solución sería ejecutar una **transacción compensadora** compuesta por las operaciones opuestas a las efectudas por la transacción comprometida. Sin embargo, la realización de esta nueva transacción, que es **trabajo del programador**, muchas veces no es posible.

   Una transacción que terminó todas sus operaciones pero todavía no escribió en el disco es, básicamente, una **transacción parcialmente comprometida** y, como se comentó anteriormente, puede fallar (debido a una caída del sistema, por ejemplo) y pasar a un estado de transacción fallida.

   Las **escrituras externas observables** son aquellas que producen una salida por pantalla (por ejemplo) y que por tanto pueden haber sido vistas por el usuario, por lo que **no pueden ser borradas**. Por ello, muchos sistemas las **almacenan en memoria no volátil** y esperan a que la transacción esté comprometida para poder mostrarlas.

5. **¿Por qué es interesante permitir la ejecución concurrente de las transacciones? ¿Quién garantiza el aislamiento en la ejecución concurrente? ¿Qué es una planificación? ¿Qué es una planificación secuencial? ¿Qué es una planificación secuenciable o serializable?**

   Las ejecuciones concurrentes suponen una serie de mejoras bastante evidentes sobre las ejecuciones secuenciales de transacciones, como por ejemplo la **mejora de rendimiento (aumento de la productividad) y uso de los recursos** al poder ejecutar varias operaciones paralelamente y no penalizar a las transacciones cortas a esperar a que terminen las largas o las limitadas por la entrada/salida, produciendo así también una **reducción del tiempo de respuesta** del sistema.

   Sin embargo, la ejecución concurrente provoca un **aumento bastante notable en la probabilidad de llegar a estados inconsistentes** en la base de datos. Para evitar que esto ocurra, **el SGBD se encarga del control de concurrencia**, planificando en que orden se ejecutan las instrucciones.

   Para llevar a cabo este control, el SGBD realiza **planificaciones**, que no son más que secuencias de ejecución que controlan el orden cronológico en el que deben realizarse las operaciones de las transacciones en el sistema (manteniendo el orden entre las operaciones de cada transacción).

El tipo de planificación más básico y sencillo es aquel en el que todas las instrucciones de una misma transacción aparecen juntas, provocando de esta forma un comportamiento idéntico a ejecutar las transacciones secuencialmente. Por ello, estas planificaciones reciben el nombre de **planificaciones secuenciales**.

Sin embargo, con planificaciones secuenciales no estamos aprovechando la ejecución concurrente. Por ello, es necesario buscar planificaciones que no coloquen todas las instrucciones de todas las transacciones juntas, pero si generen el mismo resultado final que el de una ejecución secuencial (que es, a fin de cuentas, el resultado correcto). A aquellas planificaciones, sean de la forma que sean, que cumplan esta última condición de devolver el mismo resultado que una ejecución secuencial, se les llama **secuenciables** o **serielizables**.

6. **Una planificación secuencial, ¿es también secuenciable? ¿Por qué? Define los conceptos de: conflicto entre instrucciones, equivalencia en cuanto a conflictos, secuencialidad en cuanto a conflictos.**

   Si una planificación es **secuencial**, **siempre es secuenciable**. Esto es evidente pues se le llama secuenciable a toda aquella planificación que provoque el mismo resultado que la ejecución secuencial de las transacciones y la planificación secuencial representa básicamente esta ejecución secuencial.

   - **Conflicto entre instrucciones:** Se dice que existe conflicto entre instrucciones cuando dos o más instrucciones acceden al mismo elemento de datos y al menos una de ellas es de escritura. Alterar el orden en el que se ejecuta la escritura con respecto a las demás instrucciones varía el resultado de estas.
   - **Equivalencia en cuanto a conflictos:** Se dice que dos planificaciones son equivalentes en cuanto a conflictos cuando de la primera se puede pasar a la segunda intercambiando el orden de ejecución entre dos operaciones no conflictivas de transacciones diferentes, produciendo por tanto el mismo resultado.
   - **Secuencialidad en cuanto a conflictos:** Se dice que una planificación es secuencial en cuanto a conflictos si esta es equivalente en cuanto a conflictos a una planificación secuencial.

7. **Grafo de precedencia: Nodos y arcos. ¿Cómo se puede utilizar para comprobar que una planificación es secuenciable en cuanto a conflictos? ¿Existen planificaciones que producen el mismo resultado y no son equivalentes en cuanto a conflictos? Ejemplo.**

   El **grafo de precedencia** es un grafo dirigido cuyos **nodos** representan las transacciones de una planificación y cuyos **arcos** representan que, teniendo dos instrucciones conflictivas, la instrucción del nodo origen se ejecuta siempre antes que la instrucción del nodo destino. Gracias a este grafo podemos saber si la planificación es secuenciable en cuanto a conflictos.

   Para **comprobar que una planificación es secuenciable en cuanto a conflictos**, simplemente tenemos que ver que el grafo **no contenga ciclos**. Si el grafo contiene ciclos significa que una instrucción de la primera planificación tiene que ir antes que otra de la segunda y vicerversa, por lo que no es posible realizar intercambios para llegar a una planificación secuencial.

   Sin embargo, es posible encontrar planificaciones secuenciables (es decir, que producen el mismo resultado que el que se obtiene al ejecutar las transacciones de forma secuencial)

pero que no son secuenciables en cuanto a conflictos fallos (porque forman ciclos en el grafo de precedencia). Veamos un ejemplo:

Pongamos el caso de una **transacción 1** que *lee el valor de un segmento de datos A* (que supongamos que es 10), le suma 5 y *escribe el resultado* (es decir, 15). Además, también *lee el valor de otro segmento de datos B* (que digamos que es 100), le resta 20 y *escribe el resultado* (B=80). Posteriomente, una **transacción 2** ejecuta una instrucción que *lee B* (B=80), le suma 10 y *escribe el valor* (B=90) y otra instrucción que *lee A* (A=15), le resta 2 y *escribe el valor* (A=13). Al finalizar la ejecución de esta **planificación secuencial**, los valores escritos en la base son: **A=13** y **B=90**.

Por contra, imaginemos otra planificación en la que primero se ejecuten las *instrucciones sobre el segmento de datos A de la transacción 1* (por tanto A=15), posteriormente las *instrucciones sobre B de la transacción 2* (teniendo B=110), a continuación las *instrucciones sobre B de la transacción 1* (obteniendo **B=90**) y finalmente las *instrucciones sobre A de la transacción 2* (resultando **A=13**). Como vemos, **el resultado es exactamente el mismo**, sin embargo esta planificación formaría **ciclos en el grafo de precedencia** ya que las instrucciones sobre A de la transacción 1 se hacen antes que las de la transacción 2, pero las instrucciones sobre B de la transacción 2 se hacen antes que las de la transacción 1).

8. **Define los conceptos de: Planificación recuperable y Planificación sin cascada. ¿Existen planificaciones sin cascada que no sean recuperables? ¿Por qué?**

   - **Planificación recuperable:** Planificación en la cual todo par de transacciones cumpla la condición de que si una primera transacción depende de la otra, esta última debe comprometerse antes que la primera. De esta forma se asegura que los elementos leídos por la primera (que son los que ha escrito la segunda) son correctos, y si la segunda transacción se tiene que abortar, se aborta también la primera.
   - **Planificación sin cascada:** Planificación en la cual para cada par de transacciones se compromete la transacción de la que depende la otra antes de que ésta lea los datos modificados por ella. Con esto, se evita deshacer transacciones en cascada por el fallo de una de la cual son dependientes las demás.

   Dadas estas definiciones, es fácil de comprender que **toda planificación sin cascada es además recuperable**, pues si falla la transacción de la que dependa la otra, lo hará antes de comprometerse, y por tanto antes de que esta otra lea los datos escritos por ella.

9. **¿Por qué es interesante dar soporte a distintos niveles de aislamiento? Describe los niveles de aislamiento del estándar: secuenciable, lectura repetible, lectura comprometida, lectura no comprometida. ¿Qué es una escritura sucia? ¿En qué niveles puede darse una escritura sucia? ¿Cómo se cambia en JDBC el nivel de aislamiento? ¿En qué nivel de aislamiento tendrá el SGBD más rendimiento? Ventaja y desventaja de usar niveles de aislamiento altos o bajos.**

   La secuencialidad garantiza el aislamiento y por tanto la consistencia, pero disminuye mucho el rendimiento. Por tanto, es una decisión de compromiso la que se debe tomar en cada caso adaptándose a la situación. Por ello, es interesante que se ofrezcan diversos **niveles de aislamiento** que se adapten a estas situaciones. Los niveles que se definen en el estándar son los siguientes:

   - **Secuenciable:** Normalmente asegura la ejecución secuenciable de la transacción.

- **Lectura repetible:** Solo permite leer datos escritos por transacciones comprometidas (es decir, nunca lee datos escritos por transacciones todavía no comprometidas) y entre dos lecturas de un dato por parte de una transacción no permite que ninguna otra lo modifique.
- **Lectura comprometida:** Solo permite leer datos escritos por transacciones comprometidas, pero a diferencia del caso anterior, entre dos lecturas de un mismo dato por parte de una transacción, este puede haber sido modificado por otra (por lo que no garantiza lecturas repetibles).
- **Lectura no comprometida:** Es el nivel de consistencia más bajo. Permite leer incluso datos escritos por transacciones no comprometidas todavía.

Se denomina **escritura sucia** al hecho de que una transacción escriba un dato que ya había sido escrito por otra transacción no comprometida todavía. Sin embargo, esto **no es permitido por ninguno de los niveles del estándar**.

Para cambiar el nivel de aislamiento en **JDBC** tenemos que usar la función **Connection.setTransactionIsolationLevel(***nivel***)**, donde *nivel* representa el nivel de aislamiento que queremos configurar.

Como se comentó anteriormente, niveles de aislamiento más bajo suponen un mayor rendimiento, al permitir mayor concurrencia. Por tanto, el **SGBD tendrá más rendimiento en el nivel de lectura comprometida**.

Además, como también se dijo antes, sabemos que las ventajas de usar un nivel de aislamiento alto es que aseguran los resultados exactos y consistentes, pero reducen el rendimiento del sistema al secuencializar la ejecución de las transacciones, y justo lo opuesto para los niveles de aislamiento bajo.

10. **Describe brevemente las características principales de los siguientes mecanismos de implementación del aislamiento: Bloqueos, Marcas de tiempo, versiones múltiples y aislamiento de instantáneas.**

    - **Bloqueos:** Consisten en bloquear los elementos de datos cuando una transacción acceda a ellos (exclusión mútua). Se pretende que duren lo necesario para garantizar la secuencialidad, pero sin excederse para no afectar demasiado al rendimiento.
    - **Marcas de tiempo:** A cada transacción se le asigna una marca de tiempo en función del momento en el que entró en el sistema. Cada elemento de datos almacena la marca de la transacción que lo ha leído en último lugar y la marca de la transacción que escribió el valor actual. Con ello se garantiza que las transacciones accedan a los datos en el orden preestablecido por las marcas de tiempo, y si una intenta acceder fuera de orden se aborta, se reinicia y se le da una nueva marca de tiempo.
    - **Versiones múltiples:** Se almacenan varias versiones de un dato, de forma que una transacción puede leer un dato antiguo en lugar del escrito por otra transacción que debería ejecutarse después pero lo hizo antes.
    - **Aislamiento de instantáneas:** Se almacenan diversas "versiones" (*instantáneas*) de la base de datos con las cuales trabajan las transacciones para leer y escribir. Cuando las transacciones pasan a parcialmente comprometidas, los cambios de estas versiones se escriben en la base de datos.

11. **Explica brevemente el problema del fenómeno fantasma y sus posibles soluciones.**

El problema del **fenómeno fantasma** aparece cuando dos transacciones están en conflicto sin acceder a ninguna tupla común. El problema en estos casos reside en que si cambiamos el orden de ejecución de las transacciones, el resultado obtenido es distinto, y esto se debe a que una de ellas produce una inserción en un elemento de datos en el cual la otra realiza una consulta. Si la consulta se realiza antes de la inserción, no comparten ninguna tupla y es donde se produce el fenómeno fantasma, pero si la inserción se realiza antes de la consulta, vemos que si y por tanto el resultado de la consulta cambia.

Para solucionar el problema del fenómeno fantasma existe el **protocolo de bloqueo de índices**, que bloquea tanto los datos como las estructuras a las que pertenecen estos datos (lo que supone también una reducción de la concurrencia). Otra alternativa es el **bloqueo del predicado**, la cual es más costosa, pero es la utilizada por *PostgreSQL* desde su versión 9.1.

12. **Considera el caso de una aplicación de reserva de asientos en un avión. ¿Se puede generar un estado inconsistente derivado de su ejecución concurrente? ¿Cuál sería la solución más sencilla? ¿Qué problema tiene esta solución más sencilla? ¿Cuál sería la solución adoptada normalmente?**

    Se genera un estado inconsistente si a dos usuarios se les muestra la misma disponibilidad de asientos pero, tras realizar la reserva de un asiento el primero de ellos, el otro (al que no se le modifica la imagen porque ésta es fija) selecciona el mismo asiento que el usuario anterior, ya que a él le aparece libre.

    La solución más sencilla para este caso sería bloquear el acceso a la reserva de asientos cada vez que haya un usuario en ella (exclusión mútua). Sin embargo, esto provocará grandes esperas para los demás usuarios, especialmente si el usuario que se encuentra en la reserva deja la página abierta y se marcha a hacer otras cosas. Por tanto, esta solución es inaceptable.

    Por tanto, la solución adoptada normalmente consiste en separar la interacción con el usuario de la interacción con la base de datos. Al usuario se le presenta la imagen del avión con los asientos disponibles en el momento en el que entró a la página de la reserva, selecciona uno y posteriormente se comprueba en la base de datos que ese asiento sigue disponible y se le asigna a él.

3 Control de concurrencia

1. **Bloqueos: Modos de bloqueo. Instrucciones para solicitar y liberar bloqueos. ¿Se pueden utilizar bloqueos y obtener planificaciones no secuenciables? Pon un ejemplo. ¿Qué es un interbloqueo? Pon un ejemplo de planificación con interbloqueo. ¿Qué es peor, tener interbloqueos o inconsistencias?**

    Los bloqueos consisten en, al acceder una transacción a un elemento de datos, no permitir que ninguna otra pueda hacerlo. Existen dos modos de bloqueo:

    - **Compartido:** Permite la lectura del elemento de datos, pero no su escritura. Es compatible con más bloqueos compartidos sobre el mismo elemento de datos.
    - **Exclusivo:** Permite tanto leer como escribir en el elemento de datos. No es compatible con ningún tipo de bloqueo sobre el mismo segmento de datos.

    Por tanto, existen dos instrucciones para solicitar los bloqueos (una para cada tipo) y una para desbloquear. Llamaremos a estas funciones **Bloquear-c(Q)** (bloquea en modo

compartido un elemento de datos Q), **Bloquear-x(Q)** (bloquea en modo exclusivo un elemento de datos Q) y **Desbloquear(Q)** (desbloquea un elemento de datos Q).

Si una transacción realiza el desbloqueo de un segmento de datos inmediatamente después de acabar de trabajar con él, puede **no garantizarse la secuencialidad**. Como ejemplo, supongamos el caso de una transacción que **resta *y* de un elemento de datos *Q*** y se **los suma a un elemento de datos *R***. Otra transacción, simplemente **lee *Q*+*R***. De forma secuencial, el resultado obtenido por la segunda transacción siempre será **A+B = x**. Sin embargo, si se ejecutan de forma concurrente y la transacción 1 **desbloquea Q justo después de la resta** y en ese momento pasa a ejecutarse la transacción 2, esta leerá **x - y**, que no coincide con el resultado anterior y por tanto hace que la planificación no sea secuenciable.

Se produce un **interbloqueo** cuando una transacción no puede avanzar porque espera a que otra desbloquee un segmento de datos, mientras que esta segunda también está esperando a que la primera desbloquee otro segmento de datos para poder avanzar. Por tanto, ninguna puede avanzar por los bloqueos activos de la otra. Esto puede deberse al hecho de esperar bastante hasta desbloquear los elementos de datos bloqueados por una transacción. Un **ejemplo** es el caso de una planificación en la que una transacción bloquea un elemento de datos Q para leer su contenido y realizar una actualización (por lo tanto un bloqueo exclusivo), posteriormente otra transacción bloquea un elemento de datos R para leerlo o actualizarlo y, sin desbloquear éste, solicita el bloqueo sobre el segmento de datos Q que se encuentra bloqueado por la transacción anterior, por lo que tiene que esperar. Al continuar la primera transacción, y sin desbloquear Q, solicita el bloqueo de R, por lo que también tiene que esperar. En este punto, ambas transacciones están esperando por la otra, por lo que tenemos un interbloqueo.

Sin embargo, cuando se produce un **interbloqueo**, si el SGBD es capaz de detectarlo, la solución es tan simple como **retroceder una transacción** y continuar. Por contra, **ante un estado inconsistente no se puede hacer nada**. Por tanto, **es preferible un interbloqueo a las inconsistencias**.

2. **¿Qué es un protocolo de bloqueo? ¿Cuándo decimos que una transacción Ti precede a otra T_j (T_i -> T_j) en una planificación? ¿Qué es una planificación legal para un protocolo de bloqueo? ¿Cuándo podemos decir que un protocolo de bloqueo asegura la secuencialidad en cuanto a conflictos?**

   Se denominda **protocolo de bloqueo** al conjunto de reglas que indican el momento en el que una transacción puede bloquear y desbloquear cada segmento de datos. Esto supone una restricción en el número de planificaciones posibles.

   Se dice que una transacción T_i precede a otra T_j en una planificación cuando T_i realiza un bloqueo sobre un elemento de datos Q y posteriormente T_j también realiza un bloqueo sobre ese elemento de datos pero en un modo no compatible.

   Denominamos **planificación legal** bajo un protocolo de bloqueo a aquella que es posible para un conjunto de transacciones que sigan las reglas del protocolo de bloqueo.

   Se dice que un protocolo **asegura la secuencialidad en cuanto a conflictos** si, y sólo si, todas las planificaciones legales son secuenciables en cuanto a conflictos.

3. **¿Cuándo se puede conceder un bloqueo solicitado por una transacción? Describe el problema de la inanición en la concesión de bloqueos. ¿Cómo puede solucionarse?**

Un bloqueo sobre un elemento de datos solicitado por una transacción se le puede conceder cuando no haya activo ningún otro bloqueo **conflictivo** con el que se solicita sobre el mismo elemento de datos.

Sin embargo, existe un problema denominado **inanición** sobre la concesión de bloqueos. Al existir un bloqueo de tipo **compartido** sobre un elemento de datos solicitado por una transacción, si otra distinta quiere solicitar uno de tipo **exclusivo**, tendrá que esperar que se desbloquee éste. Pero, por otra parte, sí se pueden conceder más bloqueos de tipo *compartido*, pues no son conflictivos. Debido a esto, puede llegarse a una situación en la que la transacción que solicitó el bloqueo exclusivo se encuentre en una **espera interminable**, que es la que se conoce como **inanición**.

Una forma de resolver el problema anterior es, básicamente, no permitir que se concedan bloqueos sobre el elemento de datos (aunque sean compatibles) si ya hay otra solicitud de bloqueo esperando.

4. **Describe las reglas del protocolo de bloqueo en dos fases. ¿Asegura la secuencialidad en cuanto a conflictos? ¿Pueden ocurrir interbloqueos? ¿Pueden darse planificaciones con retrocesos en cascada? ¿Existe algún protocolo de bloqueo que conozcas que evite los retrocesos en cascada? Si existe descríbelo.**

   El **protocolo de bloqueo en 2 fases** es un tipo de protocolo que separa en dos fases las peticiones de bloqueo y desbloqueo de una transacción. Esas fases son las siguientes:

   - **Fase de creciemiento:** Fase en la que la transacción puede solicitar bloqueos pero no liberarlos.
   - **Fase de decrecimiento:** Fase en la que la transacción puede liberar bloqueos pero no solicitar nuevos.

   Este tipo de protocolo, además, **asegura la secuencialidad en cuanto a conflictos**. Esto es fácilmente demostrable ya que, en base a los puntos en los que finalizan la fases de crecimiento (conocidos como **puntos de bloqueo**), las transacciones se pueden ordenar asegurando que no se producirán conflictos y se obtendrá el mismo resultado que el de una planificación secuencial.

   No obstante, el protocolo **no asegura que no se puedan producir interbloqueos**. Aunque separemos en 2 fases los bloqueos y las liberaciones de cada transacción, si intercalamos instrucciones de la primera de las fases de dos transacciones diferentes todavía podemos obtener un interbloqueo.

   Además, también **pueden darse planificaciones con retrocesos en cascada**, pues en ningún momento se impide que una de las transacciones lea los datos escritos por otra si ésta ya liberó su bloqueo en la fase de decrecimiento.

   Una forma de **evitar los retrocesos en cascada** es con el **protocolo de bloqueo estricto en dos fases**. Este bloqueo impide que se libere un bloqueo de tipo exclusivo hasta que se complete la transacción. De esta forma, ninguna otra transacción puede leer el dato modificado hasta que se haya comprometido la transacción que lo escribió.

5. **¿Existe alguna variante del protocolo de bloqueo en dos fases que aumente el nivel de concurrencia? Si existe descríbela. ¿Puede haber planificaciones secuenciales en cuanto a conflictos que no se puedan obtener con el protocolo de bloqueo en dos fases? ¿Pueden las peticiones de bloqueo y desbloqueo generarse de forma automática? ¿Cómo?**

Simplemente con el **protocolo en dos fases se pierde concurrencia** debido a que algunas transacciones deben esperar hasta que otras realicen desbloqueos para poder ejecutarse. Sin embargo, existen las **conversiones de bloqueo** que permiten cambiar el nivel de un bloqueo existente para facilitar la compatibilidad entre instrucciones de varias transacciones. Pongamos por ejemplo el caso en el que una transacción **primero lee** un segmento de datos Q y **posteriormente lo escribe**, por lo que realiza un **bloqueo exclusivo**, mientras que otra transacción simplemente necesita **leer** el valor inicial de Q, por lo que solicita un **bloqueo compartido**. Como el bloqueo de la primera transacción es exclusivo, la segunda transacción tiene que esperar hasta que ésta llegue a la fase de decrecimiento y lo desbloquee. Sin embargo, con las **conversiones de bloqueo** podemos hacer que la primera transacción inicialmente solo realice un bloqueo compartido (permitiendo que ambas transacciones puedan leer concurrentemente Q) y cuando vaya a escribir, **suba el bloqueo** al modo exclusivo. Para respetar el protocolo en dos fases, la operación de **subir** solo está disponible en la **fase de crecimiento**, y su opuesta (**bajar**), solo lo esté en la **etapa de decrecimiento**.

Todas las planificaciones que cumplan el protocolo en dos fases son secuenciales en cuanto a conflictos, pero existen **planificaciones secuenciales en cuanto a conflictos que no se obtengan por medio del protocolo en dos fases**. Sin embargo, para obtener estas por medio de protocolos de bloqueo que no sean el de *en dos fases* es necesario tener información adicional sobre las transacciones o imponer un orden sobre el conjunto de elementos de datos de la base de datos.

Un esquema simple, pero de uso extendido, **genera las peticiones de bloqueo y desbloqueo de forma automática** para una transacción, basándose en sus solicitudes de lectura y escritura. El funcionamiento es el siguiente:

- Cuando una transacción solicita **leer** un elemento de datos, el sistema genera un **bloqueo compartido** sobre el elemento de datos y posteriormente la instrucción de lectura.
- Cuando una transacción solicita **escribir** en un elemento de datos, el sistema comprueba si ya había un bloqueo en modo compartido, generando la instrucción **subir** a dicho bloqueo, o genera directamente el **bloqueo exclusivo** y posteriormente la instrucción de escritura.
- Para **desbloquear** los bloqueos, el sistema espera a que la transacción se haya **comprometido** o haya sido **abortada**.

6. **Describe, en todo el detalle que puedas, las alternativas que conozcas para la prevención de interbloqueos.**

   Una forma simple de prevenir interbloqueos es que cada transacción realice **bloqueos sobre todos los elementos de datos que va a tratar antes de comenzar su ejecución**, asegurándose además que o se bloquean todos o no se bloquea ninguno (**bloqueo de forma atómica**). Sin embargo, esta estrategia tiene **grandes desventajas** ya que muchas veces no se conocen de antemano todos los elementos que va a necesitar bloquear una transacción y la utilización de los elementos de datos puede ser baja, teniendo elementos bloqueados mucho tiempo sin ser usados y disminuyendo claramente la concurrencia.

   Otro esquema para prevenir los interbloqueos es **ordenar las peticiones de bloqueo** en función de los elementos de datos que vayan a bloquear, al imponer un orden parcial sobre los elementos de datos de la base de datos. No obstante, esto supone un **gran coste** al tener que ordenar todos los elementos de la base de datos.

Por último, otra aproximación para evitar interbloqueos consiste en utilizar **expropiación y retroceso de transacciones**. Cuando una transacción solicita un bloqueo incompatible con otro vigente en otra transacción, se puede **expropiar** este bloqueo vigente **retrocediendo dicha transacción** y asignando el bloqueo a la transacción que lo solicitó. Para controlar esta expropiación se usan **marcas temporales** asignadas a cada transacción, que indican si la transacción debe **esperar o retroceder**. Existen dos esquemas en los que se usan estas marcas temporales:

- **Esperar-Morir:** Evita la expropiación. Si la transacción que solicita el bloqueo (habiendo ya uno conflictivo activo) tiene una **marca de tiempo menor** (es decir, es anterior) que la de la transacción que posee el bloqueo, entonces **espera**. Por contra, si tiene una **marca temporal mayor** (y por tanto es más reciente), entonces **retrocede** (*muere*).
- **Herir-Esperar:** Utiliza la expropiación. Cuando una transacción solicita un bloqueo conflictivo con el de otra transacción, si la que lo solicita tiene una **marca temporal mayor** (es más reciente), entonces **espera**. En caso contrario, si tiene una **marca temporal menor** (es más antigua), **hace retroceder** a la transacción que posee el bloqueo.

7. **Describe, en todo el detalle que puedas, algún método que conozcas para la detección y recuperación de un estado de interbloqueo.**

   Cuando el sistema no es capaz de garantizar la ausencia de interbloqueos, es necesario contar con un esquema de **detención y recuperación de bloqueos** que permita averiguar cuando se producen y pueda actuar para recuperarse del mismo. Para ello, el sistema básicamente tiene que cumplir 3 pasos:

   - **Mantener información** sobre la asignación de elementos de datos a las transacciones (**bloqueos activos**) y también sobre las peticiones pendientes (**solicitudes de bloqueo**).
   - Disponer de un **algoritmo**, que se ejecute reiteradamente, capaz de **determinar si se ha producido un interbloqueo**.
   - Ser capaz de **recuperarse del interbloqueo** cuando el algoritmo anterior lo determine.

   Para resolver los dos primeros puntos, que se engloban dentro de la **detención de interbloqueos**, es muy común describir un **grafo dirigido** donde los nodos representan las transacciones y los arcos entre ellos simbolizan las esperas de una transacción (de la que sale el arco) debido a que está solicitando un elemento de datos reservado por otra (a la que se dirige el arco). Por tanto, detectar un interbloqueo es tan simple como **buscar ciclos en el grafo**. Así pues, el sistema deberá implementar alguno de los múltiples algoritmos que existen para detectar ciclos en un grafo y **ejecutarlo reiteradamente**. Sin embargo, ejecutar el algoritmo de detención de ciclos continuamente sobre un grafo cada vez mayor puede suponer un **coste bastante grande**. Por ello, hay que prever con que probabilidad se puede dar un interbloqueo y sobre que conjunto de elementos de datos se puede dar para ajustar la regularidad con la que se invoca al algoritmo.

8. **Protocolos basados en validación: ¿Para qué conjuntos de transacciones son idóneos? Describe sus fases. Describe el algoritmo que se utiliza en la fase de validación. ¿Pueden darse retrocesos en cascada? ¿Puede haber inanición? ¿Por qué se llaman protocolos de control de concurrencia optimistas, y cuáles serían los pesimistas?**

Cuando tenemos un conjunto de transacciones donde la mayoría son de **sólo lectura** la **probabilidad de conflictos es baja**. El **sistema de concurrencia impone una sobrecarga** en la ejecución del código que **para estos casos puede parecer inútil** ya que en la mayor parte de los casos, incluso sin él, se llegará a un estado consistente. En el intento de reducir esta sobrecarga necesitamos saber de antemano que transacciones estarán involucradas en conflictos, por lo que necesitamos **supervisar** el sistema. De ello se encargan los **protocolos basados en validación**, pensados para este tipo de conjuntos de transacciones, que asumen que cada transacción se ejecutará en **dos o tres fases** dependiendo si es de sólo lectura o solo escritura. Las fases son:

- **Fase de lectura:** La transacción lee los valores de la base de datos y los almacena en una variable local. Todas las operaciones de escritura se hacen sobre estas variables locales, no sobre la base de datos.
- **Fase de validación:** Se realiza una prueba de validación para comprobar si se pueden escribir los valores de las variables locales actualizadas en la base de datos sin causar una violación de la secuencialidad.
- **Fase de escritura:** Si la validación fue positiva, se escriben los valores en la base de datos. Evidentemente, las transacciones de sólo lectura no ejecutan esta fase.

Para cada una de estas fases se impone un **marca temporal**. Por tanto tenemos la marca de **inicio**, cuando se empieza a ejecutar la transacción, la marca de **validación**, al terminar la fase de lectura y antes de realizar la prueba de validación, y la marca de **fin**, cuando la transacción finaliza la fase de escritura. Para fijar el orden de secuencialidad se escoge como **marca global de la transacción** la **marca de validación**.

Para la **prueba de validación** de la segunda fase se comprueba que, para toda transacción T_i cuya marca temporal es menor que T_j (y por tanto irá antes en un orden secuencial, **o** la marca **Fin de** T_i es **menor** que la marca **Inicio de** T_j (y por tanto se cumple el orden secuencial y no hay ningún problema), **o** que los **datos escritos por** T_i **no intersecan con los leídos por** T_j **y** además se cumple que **Fin de** T_i **es menor que Validación de** T_j (y por tanto la primera acaba antes de que empiece la validación de la segunda).

Este esquema de validación asegura que **no habrá retrocesos en cascada** ya que las escrituras en la base de datos tienen lugar después de que se comprometa la transacción que la ejecuta.

Sin embargo, si **puede haber inanición** ya que **transacciones cortas** pueden causar el continuo **fallo de validación** de una **transacción larga**, haciendo que esta no prosiguiese nunca. Una forma de superarlo sería bloquear temporalmente las transacciones cortas hasta hacer avanzar a la larga.

A los protocolos basados en validación se les denomina **protocolos optimistas** ya que supone que las transacciones son capaces de finalizar y se realiza la **validación al final**. Por contra estarían los **protocolos pesimistas**, como los **bloqueos** o la **ordenación por marcas temporales**, que suponen que las transacciones no van a ser capaces de finalizar y toman medidas **preventivas** que fuerzan esperas y retrocesos.

9. **Aislamiento de instantáneas: Describe de la forma más detallada que puedas su funcionamiento. ¿Qué tipos de conjuntos de transacciones son los más favorecidos por este protocolo? Describe el problema de la actualización perdida. Describe las dos variantes del protocolo que solucionan el problema de la actualización perdida.**

Una forma de **mejorar la concurrencia** es realizar diversar copias (**instantáneas**) de la base de datos y hacer que las transacciones trabajen con ellas en lugar de provocar esperas. Esto es básicamente lo que hace el **aislamiento de instantáneas**, el cual proporciona una instantánea de la base de datos a cada transacción en el momento en el que ésta comienza su ejecución. La transacción trabajará sobre esta copia, y solo se escribirán los cambios realizados de vuelta a la base de datos cuando la transacción se haya comprometido.

Por tanto, parece evidente que esta es una solución muy buena para las transacciones de **sólo lectura**, ya que de esta forma nunca tendrán que esperar ni serán retrocedidas.

Sin embargo, para transacciones de **actualización** puede ocurrir un problema si varias actúan sobre el mismo elemento de datos, cada una en su copia. La primera transacción se comprometerá y escribirá los cambios realizados en su copia en la base de datos pero, posteriormente, la segunda también se comprometerá y escribirá su modificación sobre los datos originales en la base de datos, haciendo que se pierda la actualización de la primera transacción. A este problema se le conoce como **problema de la actualización perdida**.

Para solucionar este problema de la actualización perdida existen dos variantes del protocolo de aislamiento de instantáneas:

- **Primer compromiso gana:** Solo se escriben los datos de la primera transacción que se comprometa dentro del conjunto de transacciones que realicen actualizaciones sobre un elemento de datos común. Por tanto, todas las demás transacciones abortan. Para controlar quien se compromete primero se usan marcas temporales.
- **Primera actualización gana:** Cada transacción solicita un bloqueo sobre el elemento de datos que quiere escribir y espera hasta que se le conceda. Si al serle concedido ve que los datos ya han sido actualizados por otra transacción, aborta. En caso contrario, realiza su actualización y posteriormente lo escribe en la base de datos.

10. **¿Asegura la secuencialidad el protocolo de aislamiento de instantáneas? Describe lo que es un atasco de escritura y pon un ejemplo ¿Existe alguna forma de evitar estos problemas? ¿Tiene algún coste? Enumera sistemas que implementen el aislamiento de instantáneas para algún nivel de aislamiento concreto.**

    El protocolo de aislamiento de instantáneas **no asegura la secuencialidad** ya que puede darse el caso en el que una transacción lea un elemento de datos antes de que otra lo escriba, y a su vez esta otra lea un elemento de datos distinto antes de que la primera lo actualice. Como podemos ver, el resultado obtenido no será el mismo que si las transacciones se ejecutasen de forma secuencial.

    Otro problema surge cuando cada transacción lee un dato de la otra pero no escriben nada en común. De hecho, podemos comprobar que no existe ningún conflicto entre ambas transacciones y, sin embargo, aunque pueden ser comprometidas debido a esto, puede que su ejecución concurrente viole alguna restricción. A este problema se le denomina **atasco de escritura**.

    Un ejemplo lo tenemos en el caso de un banco. Imaginemos que una restricción nos indica que la suma de los valores de las cuentas de una persona no puede ser negativo. Supongamos una persona con una cuenta A con 100 euros y una cuenta B con 200 euros. Una transacción retira 200 euros de A y tras comprobar que la suma de $A+B$ sigue siendo positiva, escribe el resultado. Concurrentemente, en otra instantánea, otra transacción

retira 200 euros de B, y tras comprobar que en esa instantánea $A+B$ sigue siendo positivo, escribe el valor actualizado en la base de datos. Ambas se comprometen sin problema al haber escrito elementos de datos distintos. Sin embargo, en la base de datos tenemos ahora los valores *A=-100 euros* y *B=0 euros*, lo que da una suma negativa, que viola la restricción inicial.

Una forma de evitar este problema es el uso de cláusulas **for update**. Con estas cláusulas, el sistema **trata a los datos leídos como si hubieran sido actualizados**, de forma que con dos transacciones concurrentes que leen un elemento de datos común pero no lo escriben, una de ellas tendría que retroceder, evitando así violar restricciones como en el caso anterior. Sin embargo, el principal problema de esta solución es que al tratar las lecturas como escrituras se **pierde concurrencia**.

Ejemplos de sistemas que implementan el aislamiento de instantáneas para algún nivel de aislamiento concreto son:

- **SQLServer:** Implementa un nivel específico para este protocolo de aislamiento de instantáneas.
- **PostgreSQL:** En el nivel de aislamiento secuenciable solo ofrece asilamiento de instantáneas.
- **Oracle:** Igual que *PostgreSQL*, el nivel de aislamiento secuenciable solo ofrece aislamiento de instantánea.

11. **Describe los conflictos que se generan entre las instrucciones de inserción, borrado, lectura y escritura.**

    Teniendo las operaciones **insertar**, **borrar**, **leer** y **escribir** aplicadas sobre un mismo elemento de datos, parece obvio que no vamos a obtener el mismo resultado si movemos el orden de ejecución de las mismas.

    Por un lado, tendremos **errores lógicos** si **primero leemos** un segmento de datos y posteriormente lo **insertamos**, o si primero lo **borramos** y a continuación lo intentamos **leer**. Podemos sustituir la operación de lectura por la de **escritura** y seguiremos teniendo los mismos errores, pues el segmento de datos al que queremos acceder no existe en la base de datos en ese momento.

    Por tanto, podemos concluir que **borrar** está en **conflicto** con **leer** y **escribir**, pero también con **insertar** (por exactamente el mismo problema que el descrito para el caso *borrar-escribir*, pues si no existe el elemento de datos, porque no se ha insertado todavía, no se puede borrar). A su vez, **insertar** está en **conflicto** con **leer**, **escribir** y **borrar** por lo visto arriba, pero no está en conflito con otra instrucción de **insertar**, pues esta puede considerarse como un caso de *insertar-escribir*, la cual no da ningún problema en ese orden.

12. **Describe con el máximo detalle que puedas el problema del fenómeno fantasma y sus soluciones: Bloqueos de relación, bloqueos de índice y bloqueos de predicado.**

    Supongamos que tenemos una instrucción de **lectura** sobre un elemento de datos y otra de **inserción de una tupla adicional** sobre el mismo elemento de datos. Si se ejecuta primero la de lectura y posteriormente la de inserción obtendremos un resultado distinto a si se ejecutan en orden inverso. Sin embargo, a pesar de que es evidente que por lo tanto esta planificación no es secuenciable, en el primero de los casos, no comparten ningún dato entre ellas. Por lo tanto, el conflicto se genera debido a **una tupla que todavía**

no existe en la base de datos. A este fenómeno se le conoce como el **fenómeno fantasma**. Existen 3 posibles soluciones:

- **Bloqueo de relación:** Se asocia el elemento de datos con la relación. Por tanto, las transacciones que leen solicitan un bloqueo compartido sobre el elemento de datos, mientras que las que actualicen la información acerca de que tuplas pertenecen a la relación solicitan uno exclusivo. De esta forma la actualización y la lectura entran en conflicto en un elemento de datos real (la relación) en vez de en uno fantasma (la tupla). Sin embargo, el bloqueo de este elemento de datos no impide poder acceder a una tupla específica de forma concreta, sino que solo evita transacciones acerca de que tuplas pertenecen a la relación. Por ello, es posible que se generen inconsistencias igualmente. Además, baja considerablemente la concurrencia al producir bloqueos sobre elementos de datos tan grandes.
- **Bloqueo de índice:** Cada relación mantiene al menos un índice. Estos índices, que se representan en un árbol B+, son los que se usan para acceder a un elemento de datos. Cuando se necesita solicitar un bloqueo, se hace sobre el índice en el árbol. De esta forma, se evita el fenómeno fantasma (ahora el conflicto es sobre los elementos asociados a ese índice), y también el problema al acceso de los elementos que deberían estar bloqueados que suponía el bloqueo de relación.
- **Bloqueo de predicado:** Es el empleado por SQL desde su versión 9.1. Se realizan bloqueos compartidos sobre los predicados en una consulta. En el caso del borrado o la inserción, se comprueba si entra en conflicto con el predicado bloqueado. Si es así, espera; en caso contrario, se realiza.

13. **Describe con el máximo detalle que puedas el problema del control de concurrencia en interacciones de usuario. ¿Puede solucionarse con protocolos basados en bloqueo (problemas de la solución)? ¿Puede solucionarse con aislamiento de instantáneas (problemas de la solución)? ¿Qué tipo de solución se adopta en la práctica?**

    Cuando existe iteración con el usuario los problemas de la concurrencia se hacen más notorios. Pongamos por ejemplo el caso de la reserva de de asiento en un avión. Al usuario se le muestra el una imagen estática con los asientos libres que representa el estado de la base de datos al momento en el que accedió a la página. Sin embargo, otro usuario que acceda justo después puede encontrarse con exactamente la misma imagen. Por tanto, estos dos usuarios pueden querer reservar el mismo asiento, generando un estado de inconsistencia.

    Una solución sencilla sería **bloquear** todos los asientos mientras un usuario está en la página de reserva. De esta forma, se asegura que no se producirá ningún tipo de inconsistencia. Sin embargo, el usuario puede hacer que otros esperen indefinidamente antes de poder realizar su reserva, perdiendo completamente la concurrencia, lo cual es **muy ineficiente**.

    Otra solución puede ser aplicar **aislamiento de instantáneas**, en la que nuevamente volvemos a permitir que los usuarios accedan a la página de la reserva y vean una imagen de los asientos libres (**instantánea** de la base de datos) que había en el momento en el que entraron. Cuando dos usuarios seleccionen el mismo asiento, a la transacción iniciada por uno de los dos se le hará retroceder. Por el hecho de aplicar el protocolo de aislamiento de instantáneas tendremos que guardar una copia de los datos modificados mientras haya transacciones concurrentes activas, y puede ser que un usuario tarde mucho en finalizar su transacción, lo que obliga a alargar el tiempo que se mantiene la copia, incluso aunque la transacción que trabajó con ella se haya comprometido ya.

En la práctica se opta por dividir la **transacción en 2 partes**. En la primera se leen los asientos disponibles y se le muestran al usuarios, dejando que este pase todo el tiempo que quiera escogiendo su asiento favorito. Posteriormente, una vez lo haya seleccionado, se ejecuta la segunda parte, en la cual se comprueba si el asiento sigue estando disponible.

14. **Describe el protocolo de control de concurrencia optimista sin validación de lectura.**

    El **protocolo de control de concurrencia optimista sin validación de lectura** supone un esquema de control de concurrencia **similar al de aislamiento de instantáneas**, sin embargo, la lectura que realiza la transacción puede no corresponderse con la de la instantánea de la base de datos y, al contrario que el protocolo basado en validación, **las lecturas realizadas por la transacción no se validan**.

    Este esquema de control de concurrencia emplea **números de versión** que se almacenan en las propias tuplas y evitan el problema de la **actualización perdida**. Al insertar la tupla, este número de versión toma el valor 0. Cuando una transacción lee por primera vez la tupla para actualizarla, guarda su número de versión. De forma local, la transacción actualiza la tupla y, al momento de escribirla, comprueba que el número de versión de la tupla es el mismo que cuando lo leyó anteriormente. Si coincide, actualiza la tupla en la base de datos e incrementa en 1 el valor del número de versión. En caso contrario, aborta y retrocede las actualizaciones. Si la comprobación es correcta para todas las tuplas, la transacción es comprometida.

4 Sistema de recuperación

1. **Clasificación de los fallos: Fallo en la transacción (error lógico y error en el sistema) y caída del sistema (fallo del hardware o del software y fallo del disco). ¿Qué dos grandes partes tiene cualquier algoritmos de recuperación?**

    En un sistema de bases de datos pueden producirse distintos tipos de fallos, cada uno de los cuales requiere un tratamiento diferente. Los más comunes se pueden agrupar en los dos siguientes grupos:

    - **Fallo en la transacción:** Fallo que se produce durante la ejecución de una transacción por algún motivo relacionado con esta. Hay dos tipos de errores que hacen que una transacción falle:
        - **Error lógico:** Error en la propia transacción. Producido por una entrada errónea, un fallo de sintaxis o un dato no encontrado. Si se repite la ejecución de la misma transacción en el mismo contexto, esta volverá a fallar.
        - **Error en el sistema:** Error que impide que la transacción se ejecute por un problema en el sistema, como un interbloqueo. Se puede repetir la ejecución de la misma transacción más adelante.
    - **Caída del sistema:** Fallos producidos en el sistema (externos a las transacciones), que colapsan el funcionamiento de éste.
        - **Fallo de hardware o software:** Fallos producidos por un mal funcionamiento del hardware o software que soporta la base de datos, que suponen una pérdida de la memoria volátil y abortan el procesamiento de las transacciones. La memoria no volátil, sin embargo, permanece intacta. Existen, además, una serie de fallos que fuerzan a la parada del sistema pero no corrompen el almacenamiento no volátil. A estos se les conoce como **supuestos fallos-parada**.

– **Fallo del disco:** Fallos en los discos que componen la memoria no volátil y que suponen una pérdida de datos. Para recuperarse de estos se usan las copias de seguridad.

Para recuperarse ante estos fallos existen **algoritmo de recuperación** que analizan el error ocurrido y devuelven a la base de datos a su último estado consistente. Estos algoritmos constan de dos partes:

(a) **Comprobaciones durante la ejecución normal del sistema** para asegurarse de que está preparado para afrontar un fallo (como tener la información suficiente sobre el estado de la base de datos, por ejemplo).

(b) **Acciones llevadas a cabo tras ocurrir un fallo**, con las que reestablecen el contenido de la base de datos a un estado que asegure el cumplimiento de las propiedades *ACID*.

2. **Tipos de almacenamiento. Implementación del almacenamiento estable y acceso a los datos (operaciones de lectura, escritura, entrada y salida).**

Los elementos que componen una base de datos pueden ser almacenados en diferentes medios de almacenamiento. Estos se clasifican en tres grandes grupos:

- **Almacenamiento volátil:** La información almacenada en este tipo de almacenamiento no sobrevive a caídas del sistema. Sin embargo, el acceso a este es muy rápido. Son ejemplos de este tipo la memoria caché y la memoria principal.
- **Almacenamiento no volátil:** La información que reside en el almacenamiento no volátil sobrevive a caídas del sistema. Non obstante, es varios órdenes de magnitud más lento que el almacenamiento volátil, y además también está sujeto a fallos del propio hardware, que pueden producir una pérdida de la información. Son ejemplo de este tipo los discos duros y las cintas.
- **Almacenamiento estable:** La información almacenada en él nunca se pierde. Sin embargo, es teóricamente imposible de alcanzar. Existen, por tanto, buenas aproximaciones que reducen enormente la probabilidad de pérdida de información.

La forma de aproximarse al almacenamiento estable (y, por así decirlo, *implementarlo*), consiste en replicar la información en diversos medios de almacenamiento no volátil con modos de fallo diferentes, e ir actualizando dicha información de forma controlada. Un ejemplo son los sistemas **RAID** (*Redundant Array of Independet Disks*), que en su variante más sencilla guardan dos copias de cada bloque en distintos discos, garantizando que un fallo en un disco, aunque sea durante una transacción, no suponga la pérdida de los datos (por tener estos replicados).

Sin embargo, no previenen desastres como incendios o inundaciones. Para abordar estos es preciso guardar copias en cintas en diferentes lugares remotos. El inconveniente de esto reside en que es costoso trasladar las cintas y, en caso de un fallo, todos los datos actualizados después del último traslado son perdidos.

Por otra parte, los sistemas más seguros guardan copias tanto en el almacenamiento local como en sistemas remotos, a los que acceden por medio de redes de computadoras. En estos, los bloques se envían a los sistemas remotos al mismo tiempo que son actualizados en las copias locales, estando así siempre actualizados.

Por otra parte, todos estos sistemas que replican los datos suponen una mayor complejidad a la hora de realiza una transferencia de datos. En estos casos, las transferencias pueden finalizar de tres formas:

- **Éxito:** La información llegó a su destino con seguridad.
- **Fallo parcial:** Ocurre un fallo en el medio de la transferencia y el bloque destino contiene información incorrecta.
- **Fallo total:** Ocurre un fallo al principio de la transferencia, por lo que el bloque destino permanece intacto.

Si ocurre uno de estos fallos, el sistema debe ser capaz de detectarlo e invocar un procedimiento de recuperación que restaure el bloque a un estado estable. Para ello se necesitan dos bloques físicos por cada bloque lógico.

En una operación de salida convencional primero se escribe un bloque, y tras completarse con éxito dicha escritura, se escribe el otro bloque. La operación solo está completada cuando finalice correctamente la segunda escritura. Si se produce un fallo durante la operación, uno de los bloques estará en estado inconsistente y se podrá comprobar por medio de códigos de redundancia.

Todos estos bloques, que conforman las bases de datos, pueden residir en la memoria no volátil, como los discos (y se les conoce como **bloques físicos**) o, temporalmente, en la memoria principal (a los que se les llama **bloques de memoria intermedia**). Los bloques pasan de disco a memoria principal por medio de operaciones de **entrada** y realizan el camino inverso con operaciones de **salida**, volviendo al disco, donde reemplazan el bloque físico correspondiente.

Además, cada transacción posee un **área de trabajo** privada que se crea cuando comienza y se elimina cuando finaliza, en la que almacena copias de los elementos de datos con los que trabaja. Sobre estos datos, se pueden realizar dos operaciones:

- **leer(X):** La transacción asigna el valor del elemento de datos X a una variable local. Si el bloque que contiene a X no está en memoria principal, habrá que realizar una operación de *entrada* para traerlo.
- **escribir(X):** La transacción escribe el valor de su variable local en el elemento de datos X. Si el bloque que contiene a X no está en memoria principal, habrá que realizar una operación de *entrada* para traerlo.

3. **Describe la técnica del registro histórico, con sus variantes de modificación inmediata y diferida.**

El **registro histórico** constituye una de las técnicas más ampliamente utilizada para guardar las modificaciones efectuadas sobre una base datos. Consta de una secuencia de registros que documentan todas las actividades de actualización sufridas por la base de datos. Estos registros pueden documentar una escritura en la base de datos (se llaman **registros de actualización**), los cuales incluyen un identificador de la transacción, otro del elemento de datos, el valor antiguo de este, y el nuevo; o indicar cuando una transacción se inicia, se aborta o se compromete.

Cuando tienen lugar las escrituras es fundamental que se añada el registro al registro histórico antes de realizar la escritura sobre la base de datos. Además, este registro histórico debe residir en **almacenamiento estable**, ya que no se debe perder nunca.

Existen dos alternativas para actualizar el registro histórico:

- **Actualización inmediata:** La transacción puede modificar la base de datos mientras esté activa. Si ocurre un fallo, se ejecuta una operación ***deshacer(T)*** y se restaura el valor anterior recogido en el registro correspondiente del registro histórico.

- **Actualización diferida:** La transacción solo modifica la base de datos cuando esté comprometida. De esta forma, en el registro histórico solo es necesario guardar el nuevo valor, y no el actual, ya que si se produce un fallo durante la escritura, esto es después de haber finalizado la transacción, y se sabe cual es el valor que hay que escribir en la base de datos.

4. **Describe la interacción entre el control de concurrencia y la recuperación. Explica cuando podemos asegurar que una transacción está comprometida, si se usa la técnica de registro histórico.**

   El esquema de recuperación depende en gran medida del esquema de control de concurrencia que se use. Para poder llevar a cabo la recuperación, se necesita que el esquema de control de concurrencia evite planificaciones no recuperables, impidiendo así que si una transacción modifica un elemento de datos ninguna otra pueda modificar el mismo hasta que la primera se comprometa o se retroceda. Esto puede conseguirse con facilidad utilizando un **bloqueo estricto en dos fases**, donde la transacción solo libera los bloqueos exclusivos al final, y usando protocolos de **aislamiento de instantáneas** y **validación**, que se pueden implementar junto con **modificación inmediata** o **modificación diferida**.

   Por otro lado, usando la técnica de **registro histórico**, podremos afirmar que una transacción está realmente comprometida cuando se haya escrito el registro $<T$ *comprometida*$>$ en el bloque de memoria correspondiente y éste sufra una *salida* de la memoria principal.

5. **Describe como se utilizaría el registro histórico para deshacer y rehacer transacciones. ¿Cuál es el problema que justifica la necesidad de los puntos de revisión? Describe la creación y el uso de los puntos de revisión para deshacer y rehacer transacciones.**

   Cuando se produce una caída, el sistema produce dos listas, una para las transacciones que debe deshacer y otra para aquellas que tiene que rehacer. Para rellenarlas, se recorre el **registro histórico** desde el final hasta que se encuentra el primer **punto de revisión**. Durante este recorrido, si se encuentra un registro del tipo $<T$ *comprometida*$>$, se añade a la lista de las que se tienen que rehacer. Por otro lado, si se encuentra un registro $<T$ *iniciada*$>$ y esa transacción no está ya en la lista de rehacer, se añade a lista de las que hay que deshacer.

   Finalizada esta clasificación, se vuelve a recorrer el registro histórico desde el final hacia atrás, deshaciendo las transacciones correspondientes hasta encontrar el correspondiente registro $<T$ *iniciada*$>$ de cada una. Una vez deshechas todas las transacciones de la lista deshacer, se localiza el último punto de revisión y se recorre hacia delante el registro histórico rehaciendo las transacciones marcadas en la lista de rehacer.

   Si no existiesen los **puntos de revisión** habría que recorrer todo el registro histórico, lo que supone un coste temporal muy grande, y además se estarían rehaciendo transacciones que en muchos casos ya tienen sus datos escritos en la base de datos.

   Cuando se **crean** estos puntos de revisión, todos los bloques del registro histórico y los bloques modificados de la base de datos salen de la memoria principal a disco, y además se escribe el registro histórico en almacenamiento estable. De esta forma, no habrá que rehacer ninguna transacción abortada o comprometida antes del punto de revisión.

6. **Describe el algoritmo utilizado para: i) retroceder transacciones y ii) recuperarse tras un fallo del sistema.**

Para **retroceder transacciones** se tiene que explorar el registro histórico desde el final hacia el principio (es decir, hacia atrás) y para cada registro de actualización, establecer su valor antiguo. Tras esto, se escribe en el registro histórico un registro ***solo-rehacer***, que indica que se deshizo la operación. Este tipo de registro solo contiene el valor que se acaba de reescribir, y no el que ha sido eliminado. Esto se debe a que estos registros de *deshacer* nunca son deshechos. El retroceso de una transacción finaliza cuando se encuentra el registro $<T$ *iniciada*$>$ correspondiente. En ese momento se escribe $<T$ *abortada*$>$.

Para **recuperarse tras un fallo del sistema**, el sistema construye dos listas, una para las transacciones que se deben rehacer y otra para las que se tienen que deshacer. Estas listas se van rellenando mientras se recorre el registro histórico hacia atrás hasta encontrar un **punto de revisión**. Durante este recorrido, para cada registro $<T$ *comprometida*$>$ se añade la transacción correspondiente a la lista de las que hay que rehacer, mientras que para cada registro $<T$ *iniciada*$>$, se añade la transacción a la lista de transacciones a deshacer, siempre y cuando esta no estuviese ya en la lista de rehacer. Una vez finalizada la clasificación, se vuelve a recorrer el registro histórico hacia atrás deshaciendo las transacciones correspondientes. Al terminar, se cola en el punto de revisión y se procede a rehacer las transacciones desde ese punto hasta el final del registro.

7. **Fallo con pérdida de almacenamiento no volátil: Describe la idea básica, y los pasos para la realización de volcados y su uso para recuperar el sistema. ¿Qué gran problema tiene el proceso de volcado?**

   Aunque es bastante menos frecuente, también es posible que ocurran fallos en los sistemas de almacenamiento no volátil. Por lo tanto, es necesario estar preparados para afrontarlos. La **idea básica** para prevenirlos consiste en **volcar** periódicamente el contenido de la base de datos en almacenamiento estable. Cuando se produzca un fallo en el almacenamiento no volátil, se utilizará el volcado más reciente para hacer que la base de datos recupere un estado consistente. Para llevar a cabo estos volcados es necesario seguir una serie de **pasos**:

   - Escribir en almacenamiento estable todos los registros del registro histórico que estuvieran en memoria principal.
   - Escribir en disco todos los bloques de la memoria intermedia.
   - Copiar el contenido de la base de datos en memoria estable.
   - Escribir un registro de tipo $<$*Volcar*$>$ en el registro histórico en almacenamiento estable que indique que en ese punto se ha realizado un volcado.

   Sin embargo, como parece evidente, escribir toda la base de datos en almacenamiento estable es un proceso **muy costosto**. Además, **pierden ciclos de CPU** porque hay que detener el procesamiento de transacciones mientras se realiza el volcado.

   Existen formas alternativas de realizar el volcado que permiten reducir estos costes, como el **volcado difuso**.

5 Seguridad

1. **Concesión y revocación de privilegios en SQL (privilegios, usuario public, sentencias de concesión y revocación). Uso de roles en SQL.**

   En SQL se distinguen varios tipos de privilegios: **select**, **instert**, **update**, **delete**. Cada uno de ellos permite, al usuario que lo posee, realizar sobre la base de datos la operación

que indica su nombre. El creador, por supuesto, tiene todos los privilegios por defecto (**all privileges**).

Estos privilegios pueden ser concedidos por medio de la sentencia **GRANT** y revocados con **REVOKE**. Además, existe una sentencia (o tipo de usuario, depende de como se quiera mirar), que permite, por defecto asignar o revocar privilegios a todos los usuarios actuales y futuros de la base de datos. Se trata del **usuario public**.

En SQL existen también los **roles**, que permiten agrupar a usuarios de forma que se les asignen a todos los mismos privilegios por pertenecer al mismo grupo. De esta forma, se asignan los privilegios a los roles y cada vez que entre un nuevo usuario solo habría que especificar de que rol es, asignándosele así directamente los privilegios correspondientes.

2. **Autorización sobre vistas, procedimientos y funciones. Autorización sobre esquemas (manipulación del esquema y permiso references). Transferencia de privilegios ("with grant option" y grafo de autorización). Revocación de privilegios (cascade y restrict, "grant option", problema de revocación en cascada de privilegios y solución con cláusula "granted by current role").**

   En una **vista**, el creador debe tener privilegios **select** sobre todos los objetos que la compongan. Además, para evaluarla se utilizan los privilegios de este.

   En el caso de las **funciones y procedimientos**, existe un privilegio **execute** ejecutarlas. Cuando se vayan a ejecutar, se hace siempre con los privilegios de creador, a no ser que en la sentencia de creación de las mismas se especifique ***SQL SECURITY INVOKER***, que permite que se ejecuten con los privilegios del invocador.

   En cuanto a los **esquemas**, solo el creador de la base de datos tiene privilegios sobre ellos. Estos privilegios incluyen su creación, modificación y borrado de objetos (como relaciones o vistas). Es, por tanto, el único que puede manipular los esquemas. Existe también un **privilegio references**, el cual permite crear claves foráneas en otras tablas que referencien a la primera. Este privilegio es necesario para que el creador de la tabla pueda decidir si permite que se referencien las tuplas de su tabla desde otras pues, si esto ocurre, y las restricciones de borrado de la otra tabla lo impiden, él no va a poder borrar una tupla de su tabla solo porque está referenciada en otra ajena.

   Generalmente, cuando a un usuario se le da un privilegio, este no está autorizado a darle el mismo privilegio a otro usuario. Sin embargo, con la sentencia ***with grant option***, permitimos que el usuario sea también capaz de conceder el privilegio a terceros. Para controlar como se distribuyen los privilegios y estas autorizaciones existe un **grafo de autorización**, cuya raíz es el administrador (tiene grado de salida, pero no de entrada). En dicho grafo, para que un usuario pueda tener permisos debe existir un camino hacia el administrador.

   Cuando se le revocan los privilegios, por defecto se establece la opción **cascade** (en cascada), con la cual, al eliminarle los privilegios a un usuarios, también son revocados todos aquellos que concedió a terceros, y se siguen eliminando recursivamente siempre y cuando los descendientes no tengan otro camino hacia el administrador. Por otra parte, si se establece la opción **restrict**, si se le quiere revocar permisos a un usuario que a su vez ha concedido permisos a otros, se genera una excepción y se impide la revocación. Sin embargo, si especificamos ***grant option***, revocaremos la opción del usuario de conceder el permiso. En estos casos, se les revocará el permiso a todos aquellos terceros a los que se lo hubiera concedido, si estos no tienen otro camino que los conecte con el administrador.

3. **Inyección de SQL: ejemplos. Fuga de contraseñas: medidas a tomar.**

La **inyección SQL** se trata de una fuga de seguridad por la cual el atacante consigue que la aplicación ejecute una sentencia SQL. Este fuga se da cuando es posible concatenar la entrada del usuario con cadenas SQL. Un ejemplo común se da en las aplicaciones antiguas de conexión con alguna base de datos, en las que, en una ventana, se pide que el usuario rellene un *textfield* para especificar algún parámetro en una búsqueda. Si no está correctamente configurada la aplicación, es posible que el usuario introduzca un ***";"*** para finalizar la parte correspondiente a la consulta, y a continuación escriba una sentencia del tipo ***drop table*** (por ejemplo), la cual será interpretada por el gestor de la base de datos. Una forma de evitar que esto ocurra es, por un lado, si se usa *JDBC*, utilizar las **sentencias preparadas** o, de forma más genérica, utilizar carácteres de escape para procesar la entrada del usuario.

Otro problema común es la **fuga de contraseñas**, que se da cuando las contraseñas de los usuarios se almacenan sin cifrar, especialmente en entornos web. Estas contraseñas pueden ser accesibles a través de un servidor HTTP. Una buena solución es almacenar las contraseñas en archivos cifrados con claves eficaces. Otra, a mayores, es no permitir el acceso al gestor de la base de datos desde cualquier IP (es decir, controlar quien puede acceder a la base de datos).

4. **Autenticación de las aplicaciones: Objetivo, forma más simple, autenticación en dos pasos, servicios de autenticación, firma única, SAML y OpenID.**

   La **autenticación** consiste en verificar la identidad de la persona o software que se conecta a una aplicación. La **forma más simple** es por medio de una contraseña secreta, sin embargo esta puede ser adivinada por terceros o filtrada si no está cifrada. Una forma alternativa es la **autenticación en dos pasos**, que consiste en mezclar la autenticación por medio de una contraseña personal con otra como medidas biométricas (huella dactilar, escáner del iris...). A veces incluso se mezcla el uso de una contraseña personal con otra generada aleatoriamente por el sistema en el momento, aunque este caso puede ser vulnerable ante ataques *man in the middle*. Otra solución son los **servicios de autentificación** centralizados, que permite evitar el tener que estar autentificándose constantemente en distintas aplicaciones. Este servidor sigue un protocolo que almacena los usuarios y sus contraseñas, y las aplicaciones acceden a él para la autenticación. Muy similar es el sistema de acceso con **firma única**, que consta simplemente de un único servicio central de autenticación.

   El **SAML** (*Security Assertion Markup Language*, lenguaje de marcado para confirmaciones de seguridad) supone un estándar para intercambiar información de autenticación y autorización.

   Por último, la norma **OpenID** supone una forma alternativa de firma única, con la cual el usuario puede verificarse sin tener que crear cuentas ni registrarse, solo con un identificador único que le sirve para todas aquellas aplicaciones y páginas que soporten la norma.

5. **Autorización a nivel de aplicación: Problemas para autorizar usuarios de aplicación con SQL, problemas de la autorización a nivel de aplicación, soluciones de autorización SQL de grano fino.**

   En SQL no es posible realizar una autenticación personal de manera que cada usuario solo pueda acceder a las tuplas que le corresponden, principalmente porque lo más problable es que la mayoría de los usuarios no tengan identificador propio e individual en la base de datos. Es por ello por lo que este tipo de autenticaciones tienen que hacerse a nivel de aplicación. Sin embargo, la autenticación a nivel de aplicación supone una serie de

problemas. Por un lado está el hecho de mezclar el código de la autenticación con el código del resto del problema, lo que complica su comprensión. Por otro, y también ligado con lo anterior, aparece el problema de la dificultad de garantizar la ausencia de agujeros de seguridad. Si una aplicación tiene problemas de seguridad, puede comprometer la seguridad de las demás, y verificar la seguridad de todas es un gran esfuerzo.

Sin embargo, si la base de datos tiene **autorización de grano fino**, se pueden especificar políticas de autorización de forma declarativa, lo que conllevaría menos errores.

6. **Trazas de auditoría: ¿Qué son? ¿Para qué sirven? ¿Cómo pueden implementarse dentro de un SGBDs? ¿Qué ocurre cuando la autorización se realiza en el nivel de aplicación? Privacidad: ¿Cuál es el problema? ¿Qué soluciones proporcionan los países? Comenta lo que sepas sobre la distribución de datos agregados. Discute el tratamiento de datos privados en aplicaciones web.**

   Las **trazas de auditoría** son registros de cambios producidos en la base de datos que almacenan quien fue el usuario que lo realizó y en que momento. Gracias a ellas, cuando hay problemas de seguridad se puede saber que ha pasado y quién realizó los cambios que llevaron a esa situación, así como ayudar a recuperarse de dicho fallo.

   Pueden ser usadas para seguir los cambios hechos por un usuario y descubrir nuevos problemas, o para detectar brechas en la seguridad (como cuando un intruso está usando las credenciales de un usuario).

   Una forma de implementarlas en los SGBD es usando disparadores para añadir la información de auditoría de forma automática, utilizando variables del sistema como ***current_user*** o ***current_time***. En otros casos, los propios sistemas implementan mecanismos más fáciles de usar.

   Si la autentificación se realiza en el nivel de aplicación, sin embargo, las trazas de auditoría no pueden hacerse en la base de datos, pues esta no dispone de la información del usuario.

   Por otro lado, la **privacidad** supone una preocupación cada vez mayor actualmente debido al auge de las redes sociales, el *Big Data*, y los intereses económicos de las empresas sobre los mismos. Por ello, los países regulan leyes de privacidad, cuyo incumplimiento supone penas criminales.

6 Implementación

1. **Organización de los archivos: Almacenamiento de registros en bloques, registros de tamaño fijo (¿Qué hacemos cuando un registro no cabe en el espacio libre de un bloque? ¿Qué hacemos para borrar registros?), registros de tamaño variable (problemas a solucionar, representación de un registro de tamaño variable). Representación de registros de tamaño variable en un bloque (estructura de páginas con ranuras). Almacenamiento de registros de gran tamaño.**

   Los archivos se organizan como secuencias de registros divididos en bloques.

   Si estos registros son de **tamaño fijo**, se irán asignando en los bloques según quepan. **Si un registro no cabe** en el espacio libre de un bloque, se almacena en el siguiente bloque, dejando un desperdicio de espacio en el bloque anterior. **Cuando se borra un registro** es necesario desplazar todos los registros para compactar, evitando dejar espacios en el medio. Sin embargo, esta operación es muy costosa y requiere accesos adicionales. Por ello, otra solución consiste en mantener una lista que enlace los huecos libres desde la cabecera del archivo hasta el final.

Si, por contra, los registros son de **tamaño variable**, aparecen un par de problemas a resolver. El primero supone el hecho de representar cada registro diferente de manera que sus campos individuales puedan extraerse de forma sencilla. El segundo, almacenar los registros en los bloques de forma que sea sencillo acceder a un registro concreto.

La **representación común de estos registros de tamaño variable** consiste en unos campos al comienzo que indican información acerca de los datos de tamaño variable (la posición en la que empieza cada uno y su desplazamiento), seguido de los campos de tamaño fijo, el bitmap para representar valores nulos y por último los valores de los campos de tamaño variable.

Para representar estos registros en un bloque, se sigue una **estructura de páginas con ranuras** mediante la cual se almacena al principio un array cuyas entradas contienen la ubicación (punteros) y el tamaño de cada registro, mientras los registros reales se almacenan de manera contigua empezando desde el final.

Para los **registros de gran tamaño**, que no caben enteramente en un bloque, se usan estructuras de datos especiales como **BLOB** y **CLOB**. Estos normalmente se almacenan en archivos especiales, de forma que cada campo largo se almacena en un archivo nuevo y el registro guarda una referencia al archivo especial. Estos archivos se organizan por medio de árboles *B+*.

2. **Organización de los registros en archivos: Almacenamiento de una tabla en un archivo (montículos, secuencial y asociativa). Archivos en agrupaciones.**

   Se pueden seguir distintas estrategias de almacenamiento y organización de los registros en archivos:

   - **Organización de archivo en montículo:** No se sigue un orden para la organización de los archivos en el archivo, por lo que se dice que se sigue la estructura de un montículo.
   - **Organización de archivo secuencial:** Se ordenan los registros según una clave de búsqueda. Estos se referencian con punteros, de forma que tras cada inserción no sea necesario realizar un ordenamiento (que sería costoso). Se precisan bloques de desbordamiento para poder seguir almacenando registros una vez se acabe el espacio en el bloque actual.
   - **Organización de archivo asociativa:** Se siguen estrategias *hash* para organizar los registros.

   Otra de forma de organizar los archivos es por medio de agrupaciones de varias tablas. Cuando hay *joins* muy frecuentes entre algunas tablas, estas pueden almacenarse conjuntamente en un archivo, poniendo además muy cerca aquellas tuplas que cumplan la condición del *join*, para así ganar eficiencia.

3. **Almacenamiento del diccionario de datos.**

   A parte de almacenar las propias relaciones, un sistema de bases de datos relacionales necesita también mantener la información sobre estas. Esta información conforma el **diccionario de datos**.

   En el diccionario de datos se almacena conjuntamente información sobre los nombres de las tablas, los nombres de los atributos, el dominio y la longitud de cada uno, nombres y definiciones de las vistas, y definiciones de las restricciones de integridad (claves primarias, foráneas...). Además, muchos sistemas guardan también datos de los usuarios y sus permisos, datos estadísticos y descriptivos de las relaciones (como el número de tuplas,

la forma en que se almacenan...) y, en algunos casos, referencias a la localización de los datos en el sistema de archivos y su organización (montículos, secuencial o asociativa), datos de cada índice (nombre, relación, atributos, tipo de índice...), etc.

4. **Extensiones de los árboles B+: Organización de archivo con arbol B+, índices secundarios y reubicación de registro, indexación de cadenas de caracteres, carga en bruto, árboles B y memorias flash.**

   El problema principal de organizar la información en un archivo secuencial indexado reside en el coste de tener que hacer una búsqueda secuencial cada vez que se busque un índice o un dato. Por ello, una mejor alternativa es organizarlos siguiendo las estructuras de los **árboles B+**. Al seguir esta organización, se penaliza ligeramente temporal y espacialmente el rendimiento en las inserciones y en los borrados, pero se gana bastante eficiencia en las búsquedas (que, al fin y al cabo, va a ser la operación más repetida).

   Además, con el uso de árboles B+, estos pueden ser usados como índices (para saber localizar cada dato) pero también como organizadores de los propios datos. En estas estructuras, los nodos hoja ya contienen los propios registros, en lugar de punteros a los mismos.

   En las estructuras de árboles B+, cuando un nodo hoja se llena, este se divide en dos, reubicando entonces los registros correspondientes (lo cual ocurre aunque no se hayan modificado estos). Estas reorganizaciones requieren actualizar los índices primarios de los registros, pero también todos los secundarios (que almacenan punteros a esos registros). Esta operación es muy costosa, pues cada nodo hoja puede tener muchos registros, y cada uno de ellos estar en distintas ubicaciones de cada índice secundario. Una forma de solucionarla es almacenar en los índices secundarios los valores de los índices primarios correspondientes (y, a través de estos, poder hacer ya la búqueda directamente en el árbol), en lugar de almacenar punteros que, aunque permiten acceder directamente al registro, hay que actualizarlos en cada reubicación.

   Estos índices, si se crean sobre atributos de tipo *cadenas de caracteres*, presentan un par de problemas. Por un lado, la longitud puede ser variable entre unas y otras, por lo que será difícil almacenarlas. Por otro, estas pueden ser directamente bastante largas, lo que reduce el número de claves que caben en el nodo. Una forma de arreglar estos problemas es usar la técnica de **comprensión del prefijo**. Con ella, todas las cadenas de caracteres se reducen a una longitud fija, quedándose con sus primeras letras hasta completar dicha longitud (el prefijo). Es importante fijar un prefijo lo suficientemente largo para evitar que coincidan dos prefijos en el árbol.

   A la hora de insertar todos los datos en un archivo que sigue la estructura de árbol B+, utilizar un algoritmo de inserción convencional no es eficiente. Una alternativa es el proceso de **carga en bruto**, mediante el cual se crea un archivo temporal que almacena todas la entradas del índice, se ordena este archivo, y posteriormente se crea el índice de abajo a arriba.

   Otra alternativa de organización es el uso de estructuras de **árboles B**, que se parecen a los *árboles B+*, pero con una serie de cambios importantes. En los árboles B, los nodos internos también tienen punteros a datos, no solo los nodos hoja. Además, a diferencia de los árboles B+, en los árboles B solo se almacena cada clave una vez. En ellos, puede ser que se las búsquedas sean más rápidas al encontrar un dato antes de llegar al nodo hoja. Sin embargo, también es probable que se necesiten más niveles para representar las misma información. Por otra parte, el borrado también es más difícil en estos árboles, ya que se puede querer borrar un nodo de datos que en este caso puede ser un nodo interno, lo que requiere una reestructuración.

Cuando se usan estas estructuras de árboles, cada nodo supone un bloque en disco. Al tener que acceder a varios nodos, el proceso de búsqueda puede demorarse Por ello, una mejor solución es almacenarlos en **memorias flash**, las cuales son más rápidas en los accesos. Por contra, los borrados son más lentos, por lo que se está investigando para reducir el número de los mismos.

5. **Accesos bajo varias claves: uso de varios índices de clave única, índices sobre varias claves, índices de cobertura.**

   A veces resulta ventajoso usar varios índices para procesar ciertos tipos de consultas en una relación o construir uno nuevo sobre la clave de búsqueda.

   Un ejemplo es el **uso de varios índices de clave única**. Imaginemos que queremos realizar una consulta que cumpla dos condiciones, es decir, determinamos dos condiciones, cada una de las cuales debe ser satisfecha por un atributo. Una estrategia para realizar la consulta puede ser usar, por separado, el índice de cada atributo para encontrar los conjuntos que cumplen cada una de las condiciones y, posteriormente, realizar la intersección sobre ellos para obtener el resultado final. Sin embargo, esta estrategia puede tener una baja eficiencia si las búsquedas con los índices devuelven muchos registros, pero finalmente solo unos poco pertenecen al resultado final, puesto que habría que analizar un gran número de punteros para obtener un resultado muy pequeño.

   Otra opción, más eficiente en este caso, es crear **índices sobre varias claves**. Con esta solución, se crearía un índice nuevo cuya clave de búsqueda se constituye de la concatenación de los atributos que deben cumplir las condiciones de la consulta (**clave de búsqueda compuesta**). Este método sirve, y es eficiente, para consultas en las que se da una condición de igualdad en ambos atributos, igualdad en la primera y un rango (*mayor que* o *menor que*) en la segunda e incluso cuando solo hay condiciones sobre un único atributo.

   Por último, existe la estrategia de los **índice de cobertura**, los cuales almacenan valores de otros atributos junto con los punteros a sus registros correspondientes. De esta forma, se puede obtener el valor de esos atributos que se almacenan sin tener que acceder al registro. Se puede obtener el mismo resultado que con los índices sobre varias claves (si se almacena uno de los atributos que lo forman), pero reduciendo el tamaño de las claves de búsqueda, lo que incrementa el número de claves por nodo (al ser más pequeñas las claves, caben más) y por tanto se reduce la altura (al haber más claves por nodo se precisan menos nodos, por lo que se generan menos niveles).

6. **Asociación dinámica: problemas de la asociación estática, asociación extensible.**

   La asociación estática emplea técnicas de hashing que organizan los registros en los bloques. Sin embargo, puede ser que, por culpa de la función hash usada, muchos registros sean destinados a la misma posición, mientras que otras posiciones del bloque quedan libres. Esta degradación supone uno de los mayores problemas de la técnica de asociación estática, que se hace todavía más patente cuando el número de registros aumenta considerablemente, suponiendo un gran desperdicio de espacio.

   Una solución es la **asociación dinámica**, que permite modificar dinámicamente la función hash para adaptarse al aumento o disminución del tamaño de la base de datos.

   Una de las formas de asociación dinámica más utilizada es la **asociación extensible**. La asociación extensible hace frente a los cambios del tamaño de la base de datos dividiendo y fusionando los bloques a medida que la base de datos aumenta o disminuye. En consecuencia, se conserva la eficiencia espacial. Además, puesto que la reorganización sólo

se lleva a cabo en un bloque de cada vez, la degradación del rendimiento resultante es aceptablemente baja.

7. **Comparación de la asociación estática y dinámica. Comparación entre indexación y asociación.**

   La principal ventaja de la asociación dinámica es el rendimiento no se degrada con el incremento del tamaño (lo cual suponía el gran inconveniente de la asociación estática). Además, aunque es cierto que es necesario más espacio para su implementación, debido a la presencia de una tabla que controla los *cajones* (bloques), este incremento es realmente mínimo, pues se trata de una tabla muy simple que solo contiene punteros a los bloques.

   Mientras, la mayor desventaja reside en la búsqueda, pues se requiere un nivel más de indirección, al tener que acceder primeramente a la tabla, y posteriormente a los bloques. No obstante, esta referencia adicional solo tiene una mínima repercusión en el rendimiento.

   Por otra parte, a día de hoy la mayor parte de los SGBD soportan tanto indexación como asociación pero escoger cual usar es decisión del diseñador de la base de datos. Este diseñador deberá escoger la que más le convenga es función de varios aspectos como, por ejemplo, el tipo de consultas que se supone que van a formular los usuarios con mayor frecuencia. Si generalmente se van a realizar consultas con condiciones de igualdad, es mejor emplear la asociación, pues presenta un tiempo constante frente al tiempo logarítmico de la indexación. Sin embargo, si se van a realizar consultas por rangos (*mayor que* o *menor que*), la asociación no funciona, puesto que no es posible pasar de un bloque al siguiente para buscar los datos (ya que la organización de los bloques se hace de forma aleatoria con el hashing).

8. **Procesamiento de consultas: descripción general y medidas del coste de una consulta.**

   El procesamiento de consultas hace referencia a una serie de actividades implicadas en la extracción de datos de una base de datos. Estas actividades engloban el **análisis y traducción** de la consulta de un lenguaje como SQL a otro más adecuado para la representación interna, como el álgebra relacional extendida, la optimización de la misma, que produce varios planes de ejecución donde se escoge el de menor coste estimado, y su evaluación final para obtener el resultado.

   Para obtener las **medidas del coste de una consulta** se debe ejecutar un algoritmo de estimación de coste para cada primitiva que la conforma. Este coste puede estimarse en función de estadísticas de la base de datos y de posibles medidas como el tiempo de CPU, el número de accesos a disco, etc. Estimar el tiempo de CPU es bastante complicado, pero con respecto al tiempo de acceso a disco es varios órdenes inferior, por lo que suele ser obviado. Por otra parte, para medir el coste de acceso a datos en disco se debe conocer el tiempo medio de búsqueda de un bloque y el tiempo medio de transferencia de un bloque. Multiplicando cada uno de estas medidas por el número de bloques que se quieren buscar y transferir y sumando los tiempos, se puede obtener la estimación del coste total en acceso a disco. Sin embargo, para ser más exactos también habría que conocer el tamaño de la memoria intermedia, puesto que los bloques que la conformen serán bloques que ya no hay que ir a buscar a disco. Por supuesto, esto es casi imposible de estimar sin haberlo ejecutado.

9. **Procesamiento de la operación de selección: Escaneos, uso de índices (igualdad y comparaciones), selecciones complejas.**

   Para las operaciones de selección deben llevarse a cabo algoritmos que sean capaces de buscar las tuplas en un archivo. Estos algortimos se dividen en cuatro bloques:

- **Algoritmos básicos (de escaneo):**
  - **Algoritmo 1 (búsqueda lineal):** Se exploran todos los bloques y se comprueba para cada uno si cumplen la condición de búsqueda. No es muy eficiente, pero funciona siempre.
  - **Algortimo 2 (búsqueda binaria):** Solo funciona si el archivo está ordenado según el atributo buscado. Reduce la complejidad lineal a logarítmica.
- **Algoritmos de selecciones con índices en condiciones de igualdad:**
  - **Algoritmo 3 (índice primario, igualdad basada en la clave):** Para condiciones de igualdad, utiliza el índice para recuperar el único registro que cumple la condición.
  - **Algoritmo 4 (índice primario, igualdad basada en un atributo no clave):** Para condiciones de igualdad, al no ser sobre atributos clave, permite recuperar varia tuplas que cumplan la condición.
  - **Algoritmo 5 (índice secundario, igualdad):** Si la condición de igualdad es sobre una clave, tiene un comportamiento idéntico al *algoritmo 3*. Si la condición de igualdad no es sobre una clave, puede que cada registro resida en un bloque diferente, lo que puede provocar operaciones de *E/S* que decrementen el rendimiento.
- **Algoritmos de selecciones con índices en condiciones de comparación:**
  - **Algoritmo 6 (índice primario, comparación):** Sirve cuando hay índices primarios ordenados (como en el caso de los árboles B+) y la condición es una comparación. Si la comparación es del tipo *mayor que*, se busca el índice primario que tenga el valor que delimite el rango (es decir, el valor que aparezca al otro lado de *mayor que*) y desde él se recorren todas las demás tuplas del archivo hasta el final. Por contra, si la comparación es de tipo *menor que* no se precisan índices, simplemente se recorren las tuplas del archivo desde el principio hasta encontrar la que tenga el valor que marque la comparación.
  - **Algoritmo 7 (índice secundario, comparación):** Se utiliza un índice secundario ordenado para guiar la recuperación bajo las condiciones impuestas por la comparación. Sin embargo, los índices secundarios solo proporcionan un puntero a los registros, pero para extraerlos será necesario acceder a ellos de cada vez, lo que puede suponer operaciones de *E/S*. Si el número de registros extraídos es muy grande, este algoritmo puede ser incluso más costoso que la búsqueda lineal.
- **Algoritmos de selecciones complejas:**
  - **Algoritmo 8 (selección cojuntiva utilizando un índice):** Se recuperan los registros que cumplan una de las condiciones con algún algoritmo del 2 al 7 y, una vez estos son llevados a la memoria intermedia, allí se comprueban el resto de condiciones.
  - **Algoritmo 9 (selección conjuntiva utilizando un índice compuesto):** Si se dispone de un índice compuesto formado precisamente por los atributos que componen la selección conjuntiva, se puede buscar dicho índice directamente. En función del índice que se use, el tipo de este determina cual de los algoritmos 3, 4 o 5 utilizar en cada caso.
  - **Algoritmo 10 (selección conjuntiva mediante intersección de identificadores):** Se explora cada índice en busca de las tuplas que cumplan una condición concreta y, con todos los conjuntos obtenidos, se realiza la intersección para obtener el resultado.

- **Algoritmo 11 (selección disyuntiva mediante unión de identificadores):** Se explora cada índice en busca de punteros cuyas tuplas cumplan una condición individual. La unión de todos estos punteros proporciona el conjunto de punteros cuyas tuplas conforman el resultado final. El úlltimo paso será utilizar estos punteros para recuperar las tuplas reales.

10. **Optimización de consultas: visión general.**

    La **optimización de consultas** es el proceso de seleccionar el plan de ejecución más eficiente de entre todos los posibles para resolver una consulta dada. Este proceso requiere tres etapas:

    (a) La generación de expresiones que sean equivalentes lógicamente a la obtenida en el análisis de la consulta, usando *reglas de equivalencia*.

    (b) La estimación del coste de cada expresión basándose en datos estadísticos sobre las tablas que intervienen (como el número de tuplas) y estimaciones de medidas como el tiempo de CPU y el acceso a disco. Se selecciona la de menor coste.

    (c) Los expresiones alternativas se anotan para generar planes de ejecución alternativos.