Curso de SQL Avanzado - Sesión 3

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Contenido

```
Módulo 3: Modelado de Datos Avanzado

1. Normalización y desnormalización
2. Diseño de esquemas avanzados
3. Estrategias de particionamiento
```

Temas

301. Normalización y desnormalización
302. Diseño de esquemas avanzados
303. Estrategias de particionamiento

301. Normalización y desnormalización

La normalización y desnormalización son conceptos fundamentales en el diseño de bases de datos relacionales, y juegan roles cruciales en cómo se estructuran y acceden a los datos. Ambos enfoques buscan optimizar la base de datos, pero desde perspectivas diferentes. Vamos a explorar ambos.

Normalización

La normalización es el proceso de estructurar una base de datos relacional para reducir la redundancia de datos y mejorar la integridad de estos. Esto se logra mediante la organización de los datos en tablas de manera que se minimicen las dependencias, se evite la duplicación de datos, y se facilite el mantenimiento y la actualización de la base de datos. El proceso de normalización se lleva a cabo a través de varias "formas normales" (1NF, 2NF, 3NF, BCNF, 4NF, etc.), cada una con requisitos más estrictos que la anterior.

- **Primera forma normal (1NF):** Asegura que cada columna en una tabla sea atómica y que la tabla tenga una clave única.
- **Segunda forma normal (2NF):** Requiere que la tabla esté en 1NF y que todos los atributos no clave dependan completamente de la clave primaria.
- **Tercera forma normal (3NF):** Se alcanza cuando una tabla está en 2NF y todos sus campos no clave son dependientes solo de la clave primaria, eliminando así las dependencias transitivas.

- **Forma normal de Boyce-Codd (BCNF):** Una versión más estricta de la 3NF, donde cada determinante es una clave candidata.
- **Cuarta forma normal (4NF):** Asegura que las relaciones multivaluadas se identifiquen y se separen en sus propias tablas.

La normalización mejora la consistencia y la integridad de los datos, facilita el diseño de bases de datos y reduce la posibilidad de anomalías de datos. Sin embargo, puede llevar a un mayor número de tablas, lo que puede complicar las consultas y afectar el rendimiento.

Desnormalización

La desnormalización es el proceso de estructurar una base de datos relacional para mejorar el rendimiento de lectura de la base de datos, a expensas de introducir cierta redundancia de datos y complicar la actualización de los datos. Esto se hace típicamente agregando datos redundantes o agrupando datos en tablas más grandes para reducir el número de uniones necesarias en las consultas.

La desnormalización es útil en escenarios donde las operaciones de lectura son mucho más frecuentes que las operaciones de escritura, y donde el rendimiento de las consultas es crítico. Algunas técnicas de desnormalización incluyen:

- **Agregar columnas redundantes** para evitar cálculos costosos o uniones.
- **Combinar tablas** que se consultan juntas con frecuencia.
- **Utilizar tablas de resumen** para consultas agregadas frecuentes.

La desnormalización debe hacerse con cuidado, ya que la introducción de redundancia puede llevar a anomalías de datos y complicar el mantenimiento de la base de datos.

Conclusión

La normalización y desnormalización no son mutuamente excluyentes y a menudo se utilizan juntas en el diseño de bases de datos. La clave es encontrar el equilibrio adecuado que satisfaga los requisitos de integridad y rendimiento de la aplicación. En ambientes como Oracle, estos conceptos son esenciales para aprovechar al máximo las capacidades avanzadas del sistema de gestión de bases de datos, optimizando así tanto el almacenamiento como el acceso a los datos.

Ejemplo de una tabla en la Primera Forma Normal (1NF)

Vamos a comenzar con un ejemplo básico de una tabla que no cumple con la Primera Forma Normal (1NF) y luego cómo transformarla para que sí cumpla.

Tabla No Normalizada

Imagina que tenemos una tabla que registra ventas de productos en una tienda, pero esta tabla no está en 1NF porque algunos de los campos contienen grupos de valores, lo cual viola una de las reglas de 1NF donde cada campo debe contener valores atómicos (indivisibles).

Tabla de Ventas (No en 1NF):

VentaID	Cliente	Productos

1	Juan Pérez	Televisor, Lavadora
2	Ana Gómez	Refrigerador
3	Luis Méndez	Licuadora, Tostadora

En esta tabla, el campo `Productos` no es atómico porque lista más de un producto para algunas ventas.

Transformación a 1NF

Para transformar esta tabla a la Primera Forma Normal, necesitamos asegurarnos de que cada campo contenga solo valores atómicos. Esto implica descomponer el campo `Productos` para que cada producto esté en su propia fila. Además, asegurarse de que la tabla tenga una clave única puede requerir agregar un nuevo campo (por ejemplo, `DetalleVentaID`) si es necesario para mantener la unicidad de cada fila.

Tabla de Ventas Detallada (En 1NF):

DetalleVentaID	VentaID	Cliente	Producto
1	1	Juan Pérez	Televisor
2	1	Juan Pérez	Lavadora
3	2	Ana Gómez	Refrigerador
4	3	Luis Méndez	Licuadora
5	3	Luis Méndez	Tostadora

En esta versión normalizada de la tabla:

- Cada fila tiene una clave única (`DetalleVentaID`).
- Cada campo (columna) contiene solo valores atómicos, cumpliendo con la definición de 1NF.
- La relación entre `VentaID` y `Cliente` se repite para cada producto involucrado en la misma venta, lo que es aceptable y necesario en este diseño para mantener la relación entre ventas y productos.

Esta estructura facilita la gestión de los datos y permite realizar consultas más precisas y eficientes, como buscar todos los productos que compró un cliente específico o todas las ventas donde se vendió un producto específico.

Ejemplo de una tabla en la Segunda Forma Normal (2NF)

Para llevar una tabla a la Segunda Forma Normal (2NF), primero debe cumplir con la Primera Forma Normal (1NF), y además, debe asegurarse de que todos los atributos no clave dependan completamente de la clave primaria de la tabla. Esto significa que la tabla no debe tener dependencias parciales, es decir, ningún atributo debe depender solo de una parte de una clave primaria compuesta.

Vamos a usar el ejemplo de la tabla de ventas detallada de la sección anterior y transformarla para que cumpla con 2NF. La tabla ya está en 1NF, pero si consideramos `DetalleVentaID` como nuestra clave única, no hay dependencias parciales porque cada atributo no clave depende de toda la clave. Sin embargo, para ilustrar mejor el concepto de 2NF, supongamos que `VentaID` y `Producto` juntos forman una clave primaria compuesta en una

nueva tabla que registra los detalles de los productos vendidos, pero ahora incluimos un nuevo campo `PrecioProducto`, que es dependiente de `Producto`, pero no de `VentaID`.

Tabla DetalleVentas (Casi 1NF, no 2NF debido a dependencias parciales)

VentaID (PK)	Producto (PK)	Cliente	PrecioProducto
1	Televisor	Juan Pérez	800
1	Lavadora	Juan Pérez	400
2	Refrigerador	Ana Gómez	1000
3	Licuadora	Luis Méndez	150
3	Tostadora	Luis Méndez	100

En esta tabla, `PrecioProducto` depende solo de `Producto` y no de `VentaID`, lo que representa una dependencia parcial respecto a la clave primaria compuesta (`VentaID`, `Producto`). Esto viola la 2NF.

Transformación a 2NF

Para corregir esto y llevar la tabla a 2NF, necesitamos eliminar las dependencias parciales dividiendo la tabla en dos: una tabla que capture la relación de ventas (incluyendo el cliente y la venta) y otra que capture los productos y sus precios. Así, cada tabla tiene claves primarias a las cuales todos los atributos no clave están completamente relacionados, eliminando las dependencias parciales.

Tabla Ventas

VentaID	Cliente
1	Juan Pérez
2	Ana Gómez
3	Luis Méndez

Tabla ProductosVendidos

VentaID	Producto	PrecioProducto
1	Televisor	800
1	Lavadora	400
2	Refrigerador	1000
3	Licuadora	150
3	Tostadora	100

En este diseño:

- La tabla **Ventas** tiene una clave primaria (`VentaID`) que identifica de forma única cada venta.
- La tabla **ProductosVendidos** utiliza una clave compuesta (`VentaID`, `Producto`) para identificar de forma única cada producto vendido en cada venta. Aquí, `PrecioProducto` depende completamente de la clave primaria compuesta, satisfaciendo así los requisitos de 2NF.
- Se eliminan las dependencias parciales, ya que ahora `PrecioProducto` está en una tabla donde depende completamente de la clave primaria.

Este enfoque mejora la estructura de la base de datos al asegurar que todos los atributos en cada tabla dependan completamente de la clave primaria, facilitando la actualización de los datos y reduciendo la redundancia.

Ejemplo de una tabla en la Tercera Forma Normal (3NF)

Para alcanzar la Tercera Forma Normal (3NF), una tabla debe primero cumplir con la Segunda Forma Normal (2NF). Además, 3NF requiere que todos los atributos no clave sean mutuamente independientes entre sí y que solo dependan de la clave primaria. Esto significa que no debe haber dependencias transitivas, donde un atributo no clave dependa de otro atributo no clave.

Vamos a expandir el ejemplo de las tablas **Ventas** y **ProductosVendidos** para llevarlas a 3NF, suponiendo que hemos identificado una dependencia transitoria.

Tabla ProductosVendidos (En 2NF, pero con una dependencia transitoria)

Recordemos la tabla **ProductosVendidos**:

VentaID	Producto	PrecioProducto	CategoriaProducto
1	Televisor	800	Electrónica
1	Lavadora	400	Electrodomésticos
2	Refrigerador	1000	Electrodomésticos
3	Licuadora	150	Cocina
3	Tostadora	100	Cocina

En esta versión, supongamos que hemos añadido `CategoriaProducto`, que es un atributo que depende de `Producto`, pero no de `VentaID`. Esto crea una dependencia transitoria, ya que `CategoriaProducto` depende indirectamente de la clave primaria (`VentaID`, `Producto`) a través de `Producto`. Este diseño viola 3NF porque `CategoriaProducto` no depende directamente de la clave primaria de la tabla.

Transformación a 3NF

Para corregir esto y llevar la tabla a 3NF, necesitamos eliminar las dependencias transitivas. Podemos hacerlo dividiendo la tabla en dos, de modo que `CategoriaProducto` esté en una tabla donde dependa directamente de una clave primaria.

Tabla Productos

Producto	CategoriaProducto

Televisor	Electrónica
Lavadora	Electrodomésticos
Refrigerador	Electrodomésticos
Licuadora	Cocina
Tostadora	Cocina

Tabla ProductosVendidos Actualizada

VentaID	Producto	PrecioProducto
1	Televisor	800
1	Lavadora	400
2	Refrigerador	1000
3	Licuadora	150
3	Tostadora	100

En este diseño:

- La **Tabla Productos** contiene la `CategoriaProducto` y tiene `Producto` como clave primaria. Ahora cada producto está asociado con su categoría directamente, sin depender de la venta.
- La **Tabla ProductosVendidos** se simplifica para contener solo la información específica de cada venta (precio y producto vendido), cumpliendo con 3NF. No hay dependencias transitivas porque todos los atributos no clave dependen únicamente de la clave primaria compuesta (`VentaID`, `Producto`).

Este enfoque mejora la estructura de la base de datos eliminando dependencias innecesarias y redundancias, lo que facilita el mantenimiento y la actualización de los datos.

Ejemplo de una tabla en la Forma Normal de Boyce-Codd (BCNF)

La Forma Normal de Boyce-Codd (BCNF) es una versión más estricta de la Tercera Forma Normal (3NF). Una tabla está en BCNF si, y solo si, para cada una de sus dependencias funcionales no triviales, el lado izquierdo es una superclave. Esto significa que la tabla no debe tener dependencias funcionales no triviales de un conjunto de atributos a otro, a menos que ese conjunto de atributos sea una superclave, es decir, un conjunto de atributos que puede identificar de manera única una fila en la tabla.

Un problema común que BCNF busca resolver es el de las dependencias funcionales que involucran claves candidatas que no son parte de la clave primaria, algo que 3NF permite siempre y cuando no haya dependencias transitivas. Vamos a presentar un ejemplo simple que muestra cómo una tabla puede cumplir con 3NF pero no con BCNF, y luego cómo transformarla a BCNF.

Ejemplo antes de BCNF

Supongamos que tenemos una tabla que registra las asignaciones de aulas para clases en una universidad:

Tabla AsignacionesAula (En 3NF, pero no en BCNF)

AulaID	Profesor	Materia
101	Ana Gómez	Matemáticas
102	Luis Pérez	Historia
103	Ana Gómez	Física

En este caso, supongamos que cada profesor solo puede enseñar una materia específica, lo que introduce una dependencia funcional de `Profesor` a `Materia`. Esto significa que conociendo el `Profesor`, podemos determinar la `Materia`. Sin embargo, `Profesor` no es una superclave, ya que no puede identificar de manera única cada fila de la tabla (Ana Gómez enseña dos materias en diferentes aulas).

Transformación a BCNF

Para resolver este problema y cumplir con BCNF, necesitamos dividir la tabla en dos de manera que todas las dependencias funcionales no triviales tengan una superclave como su lado izquierdo.

Tabla ProfesoresMaterias

Profesor	Materia
Ana Gómez	Matemáticas
Luis Pérez	Historia
Ana Gómez	Física

Tabla AsignacionesAula Actualizada

AulaID	Profesor
101	Ana Gómez
102	Luis Pérez
103	Ana Gómez

Ahora, en la **Tabla ProfesoresMaterias**, `Profesor` puede no ser una clave única por sí misma debido a que un profesor puede enseñar más de una materia, pero el par `(Profesor, Materia)` sí lo es. Esta tabla muestra claramente la relación entre profesores y materias, cumpliendo con la definición de BCNF porque cada atributo a la izquierda de una dependencia funcional es una superclave.

En la **Tabla AsignacionesAula Actualizada**, ya no mantenemos la `Materia` directamente, eliminando la dependencia funcional problemática. La relación entre un aula y quién la ocupa se mantiene, y si necesitamos encontrar qué materia se enseña en un aula, podemos hacer un JOIN con la **Tabla ProfesoresMaterias**.

Este ejemplo ilustra cómo aplicar BCNF para resolver problemas de dependencias funcionales no triviales que involucran claves candidatas, mejorando la integridad y la eficiencia del diseño de la base de datos.

Ejemplo de una tabla en la Cuarta Forma Normal (4NF)

La Cuarta Forma Normal (4NF) aborda las relaciones multivaluadas independientes dentro de una base de datos. Una tabla se considera que está en 4NF si, y solo si, está en la Tercera Forma Normal (3NF) o en la Forma Normal de Boyce-Codd (BCNF) y no contiene ninguna forma de dependencias multivaluadas no triviales, a menos que sea respecto a una superclave. Esto significa que, para cada una de sus dependencias multivaluadas, cada conjunto de datos debe ser independiente de los otros conjuntos.

Para entender mejor, vamos a considerar un ejemplo que no cumple con 4NF y cómo transformarlo para que sí cumpla.

Ejemplo antes de 4NF

Supongamos que tenemos una tabla de una conferencia académica. Esta tabla registra qué conferenciantes participarán y los temas en los que son expertos. Además, registra en qué idiomas pueden presentar, dado que la conferencia es internacional y algunos conferenciantes pueden presentar en múltiples idiomas.

Tabla Conferenciantes (No en 4NF debido a dependencias multivaluadas)

Conferenciante	Tema	Idioma
Ana Gómez	Matemáticas	Español
Ana Gómez	Matemáticas	Inglés
Luis Pérez	Historia	Español
Juan Martínez	Inteligencia Artificial	Español
Juan Martínez	Inteligencia Artificial	Inglés
Juan Martínez	Inteligencia Artificial	Francés

En esta tabla, Ana Gómez puede presentar "Matemáticas" tanto en "Español" como en "Inglés", lo que representa una dependencia multivaluada entre `Conferenciante`, `Tema` y `Idioma`. Esto significa que los idiomas en los que puede presentar un conferenciante no dependen de los temas de los que habla, y viceversa, lo cual es una violación de 4NF.

Transformación a 4NF

Para corregir esta situación y llevar la tabla a 4NF, debemos dividir la tabla original en dos tablas, de manera que las dependencias multivaluadas se separen en tablas diferentes y cada una describa una relación independiente.

Tabla TemasConferenciantes

Conferenciante	Tema
Ana Gómez	Matemáticas
Luis Pérez	Historia
Juan Martínez	Inteligencia Artificial

Tabla IdiomasConferenciantes

Conferenciante	Idioma
Ana Gómez	Español
Ana Gómez	Inglés
Luis Pérez	Español
Juan Martínez	Español
Juan Martínez	Inglés
Juan Martínez	Francés

Con esta división:

- La **Tabla TemasConferenciantes** se enfoca exclusivamente en la relación entre conferenciantes y los temas de los que son expertos, sin considerar el idioma.
- La **Tabla IdiomasConferenciantes** se enfoca en los idiomas que cada conferenciante puede usar para presentar, independientemente del tema.

Esto asegura que cada tabla cumpla con 4NF al eliminar dependencias multivaluadas no triviales que no sean sobre una superclave. Ahora, la información está organizada de manera que cada relación multivaluada se maneja por separado, lo que simplifica la gestión de los datos y previene anomalías.

Ejercicios para determinar si se cumple o no la Forma Normal

Claro, aquí tienes una lista de ejercicios de opción múltiple enfocados en determinar si ciertas estructuras de tablas cumplen con las normas de normalización 1NF, 2NF, 3NF, BCNF, y 4NF. Cada pregunta incluye una breve descripción de una tabla y opciones para elegir qué forma normal cumple.

Ejercicio 1: 1NF

Tabla Estudiantes

ID_Estudiante	Nombre	Apellidos	Cursos
1	Ana	Gómez Ruiz	Matemáticas, Física
2	Luis	Pérez López	Literatura
3	Marta	Sánchez Molina	Biología, Química, Arte

¿Esta tabla cumple con la Primera Forma Normal (1NF)?

a) Sí

b) No

Ejercicio 2: 2NF

Tabla Asignaciones

ID_Asignación (PK)	ID_Profesor	NombreProfesor	ID_Curso	NombreCurso
1	P01	Ana Gómez	C01	Matemáticas
2	P02	Luis Pérez	C02	Física

Suponiendo que cada `ID_Profesor` siempre tiene el mismo `NombreProfesor`, y cada `ID_Curso` siempre tiene el mismo `NombreCurso`, ¿esta tabla cumple con la Segunda Forma Normal (2NF)?

a) Sí

b) No

Ejercicio 3: 3NF

Tabla Empleados

ID_Empleado (PK)	NombreEmpleado	ID_Departamento	NombreDepartamento	UbicaciónDepartamento
E01	Juan Martínez	D01	Contabilidad	Edificio A
E02	Ana Gómez	D02	Marketing	Edificio B

Dada la relación de que un departamento siempre está en la misma ubicación, ¿esta tabla cumple con la Tercera Forma Normal (3NF)?

a) Sí

b) No

Ejercicio 4: BCNF

Tabla Clases

ID_Clase (PK)	ID_Profesor	ID_Materia	Horario
C01	P01	M01	Mañana
C02	P02	M02	Tarde

Si `ID_Profesor` y `ID_Materia` pueden determinar de manera única `ID_Clase`, pero `ID_Clase` es la única clave primaria, ¿esta tabla cumple con la Forma Normal de Boyce-Codd (BCNF)?

a) Sí

b) No

Ejercicio 5: 4NF

Tabla Investigaciones

ID_Investigador	CampoEstudio	Financiador
INV01	Biología	Gobierno
INV01	Biología	Industria Privada
INV02	Física Cuántica	Gobierno

Dado que un investigador puede trabajar en varios campos de estudio y puede ser financiado por diferentes entidades independientemente del campo de estudio, ¿esta tabla cumple con la Cuarta Forma Normal (4NF)?

a) Sí

b) No

Respuesta de los ejercicios

Ejercicio 1. Respuesta Correcta: b) No (El campo `Cursos` no es atómico.)

Ejercicio 2. Respuesta Correcta: b) No (Hay dependencias parciales; `NombreProfesor` depende solo de `ID_Profesor`, y `NombreCurso` depende solo de `ID_Curso`.)

Ejercicio 3. Respuesta Correcta: b) No (Hay una dependencia transitoria; `UbicaciónDepartamento` depende de `ID_Departamento`, que no es la clave primaria.)

Ejercicio 4. Respuesta Correcta: a) Sí (Todos los determinantes son superclaves.)

Ejercicio 5. Respuesta Correcta: b) No (Hay una dependencia multivaluada entre `CampoEstudio` y `Financiador` respecto a `ID_Investigador`.)

302. Diseño de esquemas avanzados

El diseño de esquemas avanzados en bases de datos es un tema complejo que abarca la creación de estructuras de datos sofisticadas para soportar aplicaciones empresariales, científicas, o de cualquier índole que requiera un manejo eficiente y seguro de grandes volúmenes de información. Este proceso implica no solo la normalización para garantizar la integridad de los datos, sino también consideraciones sobre el rendimiento, la escalabilidad, la seguridad, y la capacidad de adaptación a cambios en los requisitos. A continuación, se describen algunos aspectos clave del diseño de esquemas avanzados:

Diseño Conceptual

- **Modelado de Datos:** Utiliza modelos conceptuales, como el modelo entidad-relación (ER) o el modelo de clases en UML, para representar de manera abstracta los datos y sus interrelaciones. Este paso es crucial para entender el dominio del problema y planificar la estructura de la base de datos.
- **Identificación de Entidades y Relaciones:** Define claramente las entidades (objetos de datos) y las

relaciones entre estas entidades. Esto incluye especificar claves primarias y extranjeras, atributos, y las cardinalidades de las relaciones.

Diseño Lógico

- **Normalización:** Aplica técnicas de normalización hasta alcanzar, al menos, la Tercera Forma Normal (3NF) o la Forma Normal de Boyce-Codd (BCNF) para eliminar redundancias y dependencias innecesarias, garantizando así la integridad de los datos.
- **Esquemas de Indexación y Particionamiento:** Diseña índices para mejorar el rendimiento de las consultas y estrategias de particionamiento para distribuir los datos de manera eficiente entre varios discos o nodos en un sistema distribuido.

Diseño Físico

- **Selección de Tipos de Datos:** Escoge los tipos de datos más apropiados y eficientes para cada atributo, considerando el tipo de operaciones que se realizarán sobre estos datos.
- **Almacenamiento de Datos:** Define la estructura de almacenamiento, como la organización de archivos y métodos de acceso, optimizando el uso del espacio en disco y el tiempo de acceso.

Optimización de Consultas

- **Estrategias de Acceso:** Implementa técnicas para optimizar las consultas, como la selección de algoritmos de join eficientes, el uso de vistas materializadas, y el caching de consultas frecuentes.
- **Análisis de Rendimiento:** Utiliza herramientas de análisis de rendimiento y ajusta el diseño del esquema basándose en los resultados para mejorar la velocidad de las operaciones de lectura y escritura.

Consideraciones de Escalabilidad y Seguridad

- **Esquemas Distribuidos:** Planifica esquemas que soporten la distribución de datos en un entorno distribuido, facilitando la escalabilidad horizontal.
- **Control de Acceso:** Define políticas de seguridad y mecanismos de control de acceso para proteger los datos contra accesos no autorizados.

Diseño para Aplicaciones Específicas

- **Diseño para Big Data:** Adapta el esquema para manejar volúmenes masivos de datos, considerando tecnologías como NoSQL o sistemas de almacenamiento de datos distribuidos.
- **Diseño para Aplicaciones en Tiempo Real:** Considera los requisitos de latencia baja y procesamiento en tiempo real, utilizando bases de datos en memoria y optimizando el esquema para actualizaciones rápidas y accesos de lectura.

El diseño de esquemas avanzados es un proceso iterativo que requiere una comprensión profunda tanto del dominio de la aplicación como de las capacidades y limitaciones de la tecnología de bases de datos utilizada. La flexibilidad para adaptarse a cambios en los requisitos y el entorno tecnológico es crucial, al igual que la capacidad para prever y planificar el crecimiento futuro de la base de datos.

Caso de Aplicación

Vamos a diseñar un esquema avanzado para un sistema de gestión de bibliotecas que maneje libros, usuarios, préstamos y reservas. Este ejemplo ilustrará cómo se aplican algunos de los principios del diseño de esquemas avanzados para abordar requisitos complejos y garantizar la eficiencia, escalabilidad y seguridad.

Paso 1: Diseño Conceptual

Entidades Principales:

1. **Libros:** Cada libro tiene un identificador único, título, autores, ISBN, y categorías.
2. **Usuarios:** Los usuarios están identificados por un ID único, y tienen nombres, correos electrónicos y roles (por ejemplo, estudiante, profesor).
3. **Préstamos:** Representan la acción de prestar un libro a un usuario, con fechas de préstamo y devolución.
4. **Reservas:** Permiten a los usuarios reservar libros que están actualmente prestados.

Relaciones:

- Un usuario puede tener múltiples préstamos y reservas, pero cada préstamo o reserva se asocia a un solo usuario.
- Un libro puede estar en múltiples préstamos o reservas a lo largo del tiempo, pero en un momento dado, está asociado a lo más un préstamo o reserva.

Paso 2: Diseño Lógico

Tablas y sus relaciones:

1. **Tabla Libros**
    - LibroID (PK)
    - Título
    - ISBN
    - Autores (podría normalizarse más si los autores se repiten a través de libros)
    - Categorías (normalizada en una tabla relacionada si un libro puede tener múltiples categorías)
2. **Tabla Usuarios**
    - UsuarioID (PK)
    - Nombre
    - CorreoElectrónico
    - Rol
3. **Tabla Préstamos**
    - PréstamoID (PK)
    - LibroID (FK)
    - UsuarioID (FK)
    - FechaPréstamo
    - FechaDevoluciónPrevista
    - FechaDevoluciónReal

4. **Tabla Reservas**
    - ReservaID (PK)
    - LibroID (FK)
    - UsuarioID (FK)
    - FechaReserva
    - Estado (activa, completada)

Consideraciones:

- **Normalización:** Cada tabla está normalizada para reducir la redundancia. Los autores y categorías podrían requerir tablas separadas si se busca normalizar completamente.
- **Índices:** Crear índices en `LibroID` y `UsuarioID` en las tablas de préstamos y reservas para mejorar la eficiencia de las consultas.

Paso 3: Diseño Físico

- **Tipos de Datos:** Escoger tipos de datos adecuados (por ejemplo, `VARCHAR` para nombres, `INT` para identificadores, `DATE` para las fechas).
- **Métodos de Acceso:** Utilizar índices B-tree para los campos frecuentemente buscados como `LibroID`, `UsuarioID`, `ISBN`.

Paso 4: Optimización de Consultas

- **Vistas Materializadas:** Para consultas frecuentes, como la disponibilidad actual de libros o el historial de préstamos de un usuario, se pueden crear vistas materializadas.
- **Caching:** Implementar caching de las consultas más frecuentes a nivel de aplicación.

Paso 5: Consideraciones de Escalabilidad y Seguridad

- **Escalabilidad:** Diseñar la base de datos pensando en la escalabilidad horizontal, preparando las tablas para particionamiento si el volumen de datos crece significativamente.
- **Seguridad:** Implementar controles de acceso a nivel de base de datos, asegurando que solo los usuarios autorizados puedan modificar datos. Los roles de usuarios pueden ayudar a definir estos controles.

Paso 6: Diseño para Aplicaciones Específicas

- **Aplicación Web:** Asegurar que el esquema sea compatible con consultas desde una aplicación web, optimizando para operaciones CRUD (crear, leer, actualizar, borrar) comunes en este tipo de interfaces.

Este ejemplo muestra cómo el diseño de esquemas avanzados aborda los retos de manejar datos complejos y relacionados de manera eficiente y segura, preparando el sistema para un funcionamiento óptimo tanto en el presente como en el futuro.

303. Estrategias de particionamiento

Las estrategias de particionamiento son técnicas fundamentales en la administración de bases de datos que buscan mejorar el rendimiento, la gestión, y la escalabilidad de las bases de datos, especialmente en sistemas de

gran tamaño. El particionamiento implica dividir tablas grandes y sus índices en segmentos más pequeños y manejables, lo cual puede ayudar a optimizar las consultas, facilitar el mantenimiento y permitir una distribución más eficiente de los datos a través de múltiples discos o nodos en un sistema distribuido. A continuación, se describen varias estrategias de particionamiento comúnmente utilizadas:

Particionamiento Horizontal

El particionamiento horizontal divide una tabla en múltiples tablas más pequeñas, donde cada tabla contiene un subconjunto de filas de la tabla original. Cada partición puede ser tratada como una tabla separada por el sistema de gestión de bases de datos (DBMS), pero desde la perspectiva del usuario, sigue siendo parte de la tabla original.

Ejemplo: Particionar una tabla de transacciones por fecha, con cada partición conteniendo las transacciones de un año específico.

Particionamiento Vertical

El particionamiento vertical divide una tabla en varias tablas que contienen las mismas filas pero un subconjunto de columnas. Esto es útil para separar datos que son accedidos juntos frecuentemente de aquellos que no lo son, mejorando así el rendimiento de las lecturas.

Ejemplo: En una tabla de usuarios, separar los datos de login (que se acceden frecuentemente) de la información personal detallada (que se accede menos).

Particionamiento por Clave de Partición (Partition Key)

Esta estrategia utiliza una columna específica o un conjunto de columnas para determinar cómo se dividirán los datos entre las particiones. Es común en bases de datos distribuidas donde la clave de partición puede ayudar a distribuir los datos de manera uniforme a través de los nodos.

Ejemplo: Particionar una tabla de clientes usando el país como clave de partición, lo que coloca a todos los clientes del mismo país en la misma partición.

Particionamiento por Rango

El particionamiento por rango involucra dividir datos basados en rangos de valores para una columna específica. Esta estrategia es ideal para datos con un orden inherente.

Ejemplo: Particionar una tabla de registros financieros por rango de fechas, con cada partición almacenando los registros de un trimestre específico.

Particionamiento por Lista

Similar al particionamiento por rango, pero en lugar de rangos de valores, se utilizan listas de valores específicos para dividir los datos.

Ejemplo: Particionar una tabla de empleados basada en el departamento, donde cada partición corresponde a un departamento específico.

Particionamiento Compuesto

Combina dos o más estrategias de particionamiento para crear particiones más específicas y optimizadas. Por ejemplo, se podría primero aplicar un particionamiento horizontal y luego un particionamiento vertical sobre el resultado.

Ejemplo: Particionar una tabla de eventos primero por año (horizontal) y luego, dentro de cada año, particionar por tipo de evento (vertical).

Consideraciones

- **Rendimiento:** El particionamiento puede mejorar significativamente el rendimiento de las consultas si las particiones están diseñadas de acuerdo con los patrones de acceso a los datos.
- **Mantenimiento:** Las particiones facilitan tareas de mantenimiento como la reconstrucción de índices o las copias de seguridad, ya que estas operaciones pueden realizarse en particiones individuales.
- **Escalabilidad:** El particionamiento es clave para la escalabilidad, especialmente en sistemas distribuidos, ya que permite distribuir la carga de manera más eficiente.

Elegir la estrategia de particionamiento adecuada depende en gran medida del modelo de datos específico, los patrones de acceso a los datos y los objetivos de rendimiento.

Ejemplo de una Partición Horizontal

Vamos a profundizar con un ejemplo práctico de particionamiento horizontal utilizando una base de datos de un sistema de gestión de pedidos en línea. Imagina que la tabla `Pedidos` ha crecido considerablemente debido al aumento del volumen de transacciones. Para mejorar el rendimiento y la gestión de esta tabla, decidimos implementar un particionamiento horizontal.

Situación Inicial

Tabla `Pedidos` (Antes del Particionamiento):

PedidoID	ClienteID	FechaPedido	Total	Estado
1	101	2023-01-05	100.00	Enviado
2	102	2023-03-12	150.00	Pendiente
3	103	2023-04-15	200.00	Entregado
...	...	...	...	...
N	XYZ	2023-12-30	250.00	Cancelado

Con miles o millones de filas, las operaciones en esta tabla se vuelven lentas, especialmente las consultas, actualizaciones y el mantenimiento de índices.

Implementación del Particionamiento Horizontal

Decidimos particionar la tabla `Pedidos` por el `Estado` del pedido, ya que las consultas a la base de datos a

menudo requieren filtrar por este campo. Esto divide la tabla original en varias tablas más pequeñas, cada una conteniendo pedidos con el mismo estado.

Tablas Particionadas:

1. **Pedidos_Enviado**
2. **Pedidos_Pendiente**
3. **Pedidos_Entregado**
4. **Pedidos_Cancelado**

Cada tabla tiene la misma estructura que la tabla `Pedidos` original, pero contiene solo las filas correspondientes a su respectivo estado de pedido.

Ejemplo de Tabla Particionada: `Pedidos_Enviado`

PedidoID	ClienteID	FechaPedido	Total	Estado
1	101	2023-01-05	100.00	Enviado

Beneficios

- **Mejora del Rendimiento:** Las consultas que filtran por `Estado` se vuelven más rápidas, ya que cada consulta necesita buscar en una tabla más pequeña.
- **Mantenimiento Más Fácil:** Operaciones como la reconstrucción de índices, copias de seguridad, y purgas (eliminación de datos antiguos) se pueden realizar más eficientemente, ya que pueden enfocarse en tablas específicas basadas en criterios de negocio.
- **Escalabilidad:** El particionamiento facilita la distribución de las cargas de trabajo a través de múltiples discos o servidores, si cada partición se almacena separadamente.

Consideraciones

- **Diseño de Consultas:** Las consultas que no filtran por `Estado` podrían necesitar acceder a múltiples tablas particionadas y, por lo tanto, pueden requerir un diseño cuidadoso para mantener el rendimiento.
- **Gestión de Transacciones:** Las operaciones que involucran múltiples estados (por ejemplo, cambiar un pedido de `Pendiente` a `Enviado`) pueden necesitar manejo especial para asegurar la consistencia a través de las particiones.

Este ejemplo ilustra cómo el particionamiento horizontal puede ser utilizado para optimizar el rendimiento y la gestión de las bases de datos, especialmente en escenarios donde el volumen de datos es grande y hay patrones claros de acceso a los datos.

Ejemplo de una Tabla Particionada por Lista

El particionamiento de tablas es una funcionalidad avanzada que permite dividir tablas y sus índices en unidades más pequeñas y manejables, facilitando el manejo de grandes volúmenes de datos y mejorando el rendimiento de las consultas. Para el caso de la tabla `Pedidos` particionada por el campo `Estado`.

Oracle soporta varios tipos de particionamiento, pero para este ejemplo, utilizaremos el particionamiento por lista,

que es adecuado para dividir los datos en base a un conjunto discreto de valores, como los estados de un pedido.

Creación de la Tabla Particionada por Lista

```
CREATE TABLE Pedidos (
    PedidoID NUMBER PRIMARY KEY,
    ClienteID NUMBER,
    FechaPedido DATE,
    Total NUMBER(10,2),
    Estado VARCHAR2(50)
)
PARTITION BY LIST (Estado) (
    PARTITION Pedidos_Enviado VALUES ('Enviado'),
    PARTITION Pedidos_Pendiente VALUES ('Pendiente'),
    PARTITION Pedidos_Entregado VALUES ('Entregado'),
    PARTITION Pedidos_Cancelado VALUES ('Cancelado')
);
```

Explicación:

- `CREATE TABLE Pedidos (...)`: Define la estructura de la tabla `Pedidos`.
- `PARTITION BY LIST (Estado) (...)`: Especifica que el particionamiento se realizará por lista basándose en el campo `Estado`.
- Dentro de los paréntesis de `PARTITION BY LIST`, se definen las particiones individuales para cada valor del campo `Estado`. Cada partición se nombra (`Pedidos_Enviado`, `Pedidos_Pendiente`, etc.) y se asigna a un valor específico del campo `Estado`.

Consideraciones Adicionales:

- **Almacenamiento:** Oracle permite especificar opciones de almacenamiento para cada partición, lo que puede ser útil para optimizar el rendimiento según el acceso a los datos.
- **Índices Particionados:** También puedes crear índices particionados para mejorar el rendimiento de las consultas. Los índices pueden particionarse de manera similar a las tablas.
- **Mantenimiento de Particiones:** Oracle proporciona una amplia gama de operaciones de mantenimiento de particiones, como agregar nuevas particiones, fusionar particiones existentes, dividir particiones y eliminar particiones.

Este ejemplo muestra cómo crear una tabla particionada por lista, lo que puede mejorar significativamente el rendimiento de las consultas y la eficiencia del mantenimiento para tablas grandes al segmentar los datos en particiones más pequeñas y manejables.

Ejemplo de una Tabla Particionada por Rango de Fechas

Crear una tabla particionada por rango de fechas en Oracle es una práctica común para manejar datos históricos o datos que crecen con el tiempo, como registros de transacciones o logs de eventos. Este método facilita el manejo de grandes volúmenes de datos segmentándolos en particiones basadas en rangos de fechas específicos, lo cual puede mejorar el rendimiento de las consultas y simplificar las operaciones de mantenimiento. A

continuación, se muestra cómo crear una tabla particionada por rango de fechas en Oracle.

Ejemplo: Creación de una Tabla de Transacciones Particionada por Rango de Fechas

Supongamos que queremos crear una tabla `Transacciones` que contenga registros de transacciones financieras, y queremos particionar esta tabla por mes, para los años específicos que se esperan tener transacciones.

```
CREATE TABLE Transacciones (
    TransaccionID NUMBER PRIMARY KEY,
    FechaTransaccion DATE NOT NULL,
    ClienteID NUMBER,
    Monto NUMBER(10,2),
    Detalle VARCHAR2(255)
)
PARTITION BY RANGE (FechaTransaccion) (
    PARTITION transacciones_2023_q1 VALUES LESS THAN (TO_DATE('2023-04-01',
'YYYY-MM-DD')),
    PARTITION transacciones_2023_q2 VALUES LESS THAN (TO_DATE('2023-07-01',
'YYYY-MM-DD')),
    PARTITION transacciones_2023_q3 VALUES LESS THAN (TO_DATE('2023-10-01',
'YYYY-MM-DD')),
    PARTITION transacciones_2023_q4 VALUES LESS THAN (TO_DATE('2024-01-01',
'YYYY-MM-DD')),
    PARTITION transacciones_2024_q1 VALUES LESS THAN (TO_DATE('2024-04-01',
'YYYY-MM-DD'))
    -- Más particiones pueden ser añadidas según sea necesario.
);
```

Explicación:

- `CREATE TABLE Transacciones (...)`: Define la estructura de la tabla `Transacciones`.
- `PARTITION BY RANGE (FechaTransaccion) (...)`: Especifica que el particionamiento se realizará por rango, basado en la columna `FechaTransaccion`.
- `PARTITION transacciones_2023_q1 VALUES LESS THAN (...)`: Crea una partición para las transacciones del primer trimestre de 2023, indicando que esta partición contendrá todas las transacciones con una `FechaTransaccion` menor que el 1 de abril de 2023. Se siguen patrones similares para los trimestres restantes y años futuros.

Consideraciones Adicionales:

- **Flexibilidad:** Oracle permite modificar el esquema de particionamiento después de la creación, agregando, fusionando, dividiendo o eliminando particiones según cambien los requisitos de almacenamiento de datos.
- **Índices Particionados Localmente:** Los índices en tablas particionadas pueden ser localmente particionados, es decir, cada partición de la tabla tiene su propia partición correspondiente en el índice, lo que puede mejorar el rendimiento de las consultas.
- **Operaciones de Mantenimiento Eficientes:** Las particiones facilitan tareas de mantenimiento como copias

de seguridad, purga de datos antiguos y optimización de almacenamiento, ya que estas operaciones pueden enfocarse en particiones específicas sin afectar el resto de la tabla.

Este enfoque de particionamiento por rango de fechas es especialmente útil para gestionar datos que se acumulan con el tiempo, permitiendo un acceso más rápido a los segmentos de datos más relevantes y simplificando el mantenimiento de grandes volúmenes de información.

Las particiones deben ser excluyentes entre sí para evitar ambigüedades sobre dónde se almacenan los datos. Cada fila de datos solo puede pertenecer a una partición. En el contexto del particionamiento por rango, especialmente cuando se trata de fechas, esto significa que cada rango de fechas definido para una partición no debe solaparse con los rangos de las otras particiones. Oracle garantiza esto mediante el uso de la cláusula `VALUES LESS THAN`, asegurando que los límites superiores de una partición sean el límite inferior de la siguiente, sin superposiciones.

Por ejemplo, si tienes particiones definidas por trimestres para un año, se vería algo así:

```
PARTITION BY RANGE (FechaTransaccion) (
    PARTITION q1 VALUES LESS THAN (TO_DATE('2023-04-01', 'YYYY-MM-DD')),
    PARTITION q2 VALUES LESS THAN (TO_DATE('2023-07-01', 'YYYY-MM-DD')),
    PARTITION q3 VALUES LESS THAN (TO_DATE('2023-10-01', 'YYYY-MM-DD')),
    PARTITION q4 VALUES LESS THAN (TO_DATE('2024-01-01', 'YYYY-MM-DD'))
);
```

En este caso:

- La partición `q1` captura todas las fechas desde el inicio de la tabla hasta el 31 de marzo de 2023.
- La partición `q2` captura todas las fechas desde el 1 de abril de 2023 hasta el 30 de junio de 2023.
- Y así sucesivamente, con cada partición capturando un rango único y no solapado de fechas.

Estos rangos son excluyentes entre sí porque el límite superior de una partición es el límite inferior de la siguiente. Por lo tanto, cada transacción basada en su `FechaTransaccion` solo puede caer en una de estas particiones.

Si hay intentos de solapamiento en la definición de las particiones, Oracle rechazará la definición de la tabla particionada debido a errores de definición. La exclusividad de las particiones asegura la integridad y la lógica en la organización de los datos dentro de la base de datos.

Eliminar una Partición

Eliminar datos de una partición específica puede ser una operación eficiente para gestionar el tamaño de la base de datos y mantener solo los datos relevantes. Las operaciones de eliminación pueden ejecutarse directamente sobre particiones individuales sin afectar las otras, lo que puede ser mucho más rápido que ejecutar un comando `DELETE` basado en criterios de selección que requieren escanear toda la tabla. Aquí te muestro cómo podrías hacerlo en Oracle, considerando tanto la eliminación de filas específicas como la eliminación (y opcionalmente el truncamiento) de toda una partición.

Eliminación de Filas Específicas de una Partición

Para eliminar filas específicas de una partición basándote en ciertos criterios, usarías un comando `DELETE` estándar con una cláusula `WHERE` que especifique los criterios de las filas a eliminar. Oracle optimizará la operación al limitar la búsqueda a la partición relevante si los criterios de selección hacen uso de la columna de particionamiento. Por ejemplo:

```
DELETE FROM transacciones WHERE FechaTransaccion BETWEEN '2023-01-01' AND '2023-03-31';
```

Esta operación eliminaría filas de la partición correspondiente al primer trimestre de 2023, asumiendo que la tabla `transacciones` está particionada por la columna `FechaTransaccion` y que existe una partición que cubre este rango de fechas.

Truncamiento de una Partición

Si tu objetivo es eliminar todas las filas de una partición sin tocar las otras particiones, el truncamiento es una operación más eficiente que `DELETE`. Truncar una partición elimina todas sus filas de manera instantánea y libera el espacio ocupado, sin generar una gran cantidad de registros de redo (lo cual es beneficioso desde el punto de vista del rendimiento y la administración del espacio de almacenamiento).

```
ALTER TABLE transacciones TRUNCATE PARTITION transacciones_2023_q1;
```

Este comando elimina todas las filas de la partición `transacciones_2023_q1` sin afectar otras particiones de la tabla `transacciones`.

Consideraciones

- **Impacto en el Rendimiento:** El truncamiento de una partición es generalmente mucho más rápido que la eliminación de filas porque es una operación de metadatos. No tiene que escanear y marcar cada fila para eliminación.
- **Recuperación de Espacio:** El truncamiento libera el espacio ocupado por la partición, haciéndolo disponible para otros usos dentro de la base de datos.
- **Irreversible:** A diferencia de un `DELETE`, donde puedes usar transacciones para deshacer la operación antes de confirmar, el truncamiento es inmediato e irreversible una vez ejecutado.
- **Restricciones de Integridad y Llaves Foráneas:** Tanto para `DELETE` como para `TRUNCATE`, necesitas tener en cuenta las restricciones de integridad. El truncamiento de una partición puede fallar si hay restricciones de llave foránea que dependen de las filas de la partición.

El uso adecuado de estas operaciones puede ayudar a mantener el tamaño de la base de datos controlado y asegurar que el rendimiento no decaiga debido a la presencia de grandes cantidades de datos históricos innecesarios.

Uso de Particiones en las consultas

Oracle permite consultar particiones específicas utilizando la cláusula `PARTITION` en una consulta `SELECT`. Por

ejemplo, si tienes una tabla `transacciones` particionada por trimestre y quieres consultar solo los datos de la partición correspondiente al primer trimestre de 2023, podrías usar:

```
SELECT * FROM transacciones PARTITION (transacciones_2023_q1);
```

Esta consulta accedería directamente a los datos en la partición `transacciones_2023_q1`, lo cual puede ser más eficiente que una consulta a toda la tabla, especialmente si la partición es significativamente más pequeña que la tabla completa.

Consideraciones

- **Rendimiento:** Consultar una partición específica puede ser mucho más rápido que consultar toda la tabla, especialmente si la tabla es grande y la partición es relativamente pequeña. Sin embargo, es importante recordar que el optimizador de consultas de tu SGBD generalmente puede identificar cuándo una consulta solo afecta a una partición y optimizarla en consecuencia, incluso si consultas la tabla completa.
- **Mantenibilidad:** Aunque consultar particiones directamente puede ser útil, hacerlo regularmente en aplicaciones o scripts puede reducir la mantenibilidad de tu código. Si la estructura de particionamiento cambia (por ejemplo, si se agregan, fusionan o dividen particiones), cualquier consulta que acceda a particiones específicas por nombre necesitará actualizarse.
- **Usabilidad:** El acceso directo a las particiones puede no ser soportado o puede ser diferente en otros SGBD. Si tu aplicación necesita ser portátil entre diferentes sistemas de bases de datos, dependiendo de esta característica puede no ser ideal.

En resumen, aunque técnicamente es posible y a veces útil consultar particiones directamente, es una práctica que debe usarse con consideración. Para la mayoría de las operaciones, es mejor confiar en el optimizador de consultas del SGBD para gestionar el acceso a los datos particionados de manera eficiente, a menos que haya una razón específica para enfocarse en una partición.