Curso de SQL Avanzado - Sesión 1

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Contenido

```
Módulo 1. Optimización de Consultas

1. Estrategias para mejorar el rendimiento
2. Índices avanzados y su impacto
3. Optimización de subconsultas y joins
```

Temas

101. Repaso general a las consultas SQL
102. Tipos de datos principales
103. Subconsultas
104. Consultas anidadas
105. Uso de índices
106. Estrategias para mejorar el rendimiento

101. Repaso general a las consultas SQL

Este tema es esencial como punto de partida para cualquier curso avanzado de SQL, ya que establece una base sólida sobre la cual se pueden construir conceptos más complejos.

SQL: Lenguaje de Consulta Estructurada

SQL, que significa Lenguaje de Consulta Estructurada (Structured Query Language), es el lenguaje estándar utilizado para comunicarse con bases de datos. Permite a los usuarios realizar diversas tareas, como consultar datos, actualizar datos, insertar nuevos datos y borrar datos existentes, además de crear y modificar esquemas de base de datos y controlar el acceso a los datos.

Sintaxis Básica de SQL

La sintaxis básica de SQL implica la utilización de declaraciones o instrucciones para realizar operaciones en la base de datos. Algunas de las instrucciones más fundamentales incluyen:

- `SELECT`: Se utiliza para seleccionar datos de una base de datos. Permite especificar exactamente qué datos deseas obtener.
- `INSERT INTO`: Se usa para insertar nuevos registros en una tabla.
- `UPDATE`: Permite modificar los datos existentes en una tabla.
- `DELETE`: Se utiliza para borrar registros de una tabla.
- `CREATE DATABASE`: Para crear una nueva base de datos.
- `CREATE TABLE`: Para crear una nueva tabla en la base de datos.
- `DROP TABLE`: Para eliminar una tabla existente.

Estructura de una Consulta SQL

Una consulta SQL típica que utiliza `SELECT` puede tener varias partes, incluyendo:

- `SELECT`: Especifica las columnas que se van a retornar en el resultado.
- `FROM`: Indica la(s) tabla(s) de donde se van a extraer los datos.
- `WHERE`: Especifica las condiciones que deben cumplir los registros para ser seleccionados.
- `GROUP BY`: Agrupa filas que tienen los mismos valores en columnas especificadas.
- `HAVING`: Se utiliza después de `GROUP BY` para aplicar una condición a los grupos creados.
- `ORDER BY`: Ordena los registros resultantes según una o más columnas.

Funciones de Agregación

Las funciones de agregación realizan un cálculo en un conjunto de valores y devuelven un valor único. Algunas de las funciones de agregación más comunes incluyen:

- `COUNT()`: Devuelve el número de filas que coinciden con un criterio especificado.
- `SUM()`: Suma todos los valores de una columna.
- `AVG()`: Calcula el promedio de los valores en una columna.
- `MAX()`: Encuentra el valor máximo en una columna.
- `MIN()`: Encuentra el valor mínimo en una columna.

Joins

Los "joins" son utilizados para combinar filas de dos o más tablas, basados en una columna relacionada entre ellas. Los tipos de joins incluyen:

- **INNER JOIN**: Retorna filas cuando hay al menos una coincidencia en ambas tablas.
- **LEFT JOIN (o LEFT OUTER JOIN)**: Retorna todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha.
- **RIGHT JOIN (o RIGHT OUTER JOIN)**: Retorna todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda.
- **FULL JOIN (o FULL OUTER JOIN)**: Combina los resultados de los LEFT JOIN y RIGHT JOIN.

102. Tipos de datos principales

Entender los tipos de datos es fundamental para diseñar y trabajar con bases de datos de manera efectiva, ya que cada tipo de dato tiene un propósito específico y restricciones asociadas que ayudan a garantizar la integridad y el rendimiento óptimo de la base de datos.

Tipos de Datos en SQL

Los sistemas de gestión de bases de datos (DBMS, por sus siglas en inglés) soportan diversos tipos de datos para cubrir distintas necesidades de almacenamiento de información. Aunque los nombres y el soporte exacto pueden variar entre diferentes DBMS (como MySQL, PostgreSQL, SQL Server, Oracle, etc.), hay categorías comunes de tipos de datos en SQL:

1. Numéricos

- **Enteros**: Para números sin decimales. Pueden ser `INT` (o `INTEGER`), `SMALLINT`, `TINYINT`, y `BIGINT`, dependiendo del rango de valores que pueden almacenar.
- **Decimales/Fijos**: Para números con decimales donde se requiere precisión exacta, como `DECIMAL` o `NUMERIC`, donde puedes especificar la escala y precisión.
- **Flotantes/Reales**: Para números con decimales que no requieren una precisión fija, se utilizan `FLOAT`, `REAL`, o `DOUBLE`.

2. Caracteres y Cadenas de Texto

- **CHAR**: Para cadenas de caracteres de longitud fija. Si el valor insertado es más corto que la longitud definida, el DBMS puede rellenar los espacios restantes con espacios en blanco.
- **VARCHAR**: Para cadenas de caracteres de longitud variable. Almacena cadenas de texto que pueden variar en longitud hasta un máximo definido.
- **TEXT**: Para textos largos como artículos, posts de blogs, etc. La capacidad exacta puede variar entre sistemas.

3. Fecha y Hora

- **DATE**: Solo la fecha, sin la hora.
- **TIME**: Solo la hora, sin la fecha.
- **DATETIME**: La combinación de fecha y hora en un solo valor.
- **TIMESTAMP**: Similar a `DATETIME`, pero generalmente usado para registrar cuándo ocurren cambios en los registros.

4. Lógicos

- **BOOLEAN**: Para almacenar valores verdadero o falso (`TRUE` o `FALSE`).

5. Binarios

- **BINARY** y **VARBINARY**: Similar a `CHAR` y `VARCHAR` pero para datos binarios (como imágenes o archivos).
- **BLOB**: Para almacenar datos binarios grandes, como imágenes, archivos de audio, etc.

Elección del Tipo de Dato Adecuado

La elección del tipo de dato adecuado es crucial por varias razones:

- **Eficiencia en almacenamiento y rendimiento**: Usar el tipo de dato más ajustado al tipo de información que se desea almacenar minimiza el uso de almacenamiento y puede mejorar el rendimiento de las consultas.
- **Integridad de datos**: Asegura que los datos almacenados cumplan con ciertas condiciones y expectativas, evitando errores y problemas de calidad de datos.
- **Facilidad de uso y mantenimiento**: Facilita el desarrollo y mantenimiento de aplicaciones al proporcionar un esquema claro y comprensible.

Al diseñar una base de datos, es importante considerar cuidadosamente el tipo de información que cada columna almacenará y seleccionar el tipo de dato más apropiado para esa información. Esto no solo afecta cómo se almacenan y recuperan los datos, sino también cómo se pueden manipular y presentar esos datos a través de las consultas SQL.

Ejemplo de Creación de Tabla de Empleados

Este ejemplo refleja una variedad de tipos de datos para mostrar cómo se pueden aplicar en un contexto práctico.

```sql
CREATE TABLE Empleados (
    ID_Empleado INT PRIMARY KEY,
    Nombre VARCHAR(100),
    Apellido VARCHAR(100),
    Email VARCHAR(255),
    Salario DECIMAL(10, 2),
    Fecha_Nacimiento DATE,
    Fecha_Contratacion DATETIME,
    Tiempo_Parcial BOOLEAN,
    Foto_Perfil BLOB,
    Numero_Telefonico CHAR(10),
    Departamento VARCHAR(50),
    Posicion VARCHAR(50),
    Hora_Entrada TIME,
    Hora_Salida TIME
);
```

Descripción de Campos

- **ID_Empleado**: Un número entero (INT) que actúa como la clave primaria de la tabla. Cada empleado tiene un ID único.
- **Nombre y Apellido**: Cadenas de caracteres (VARCHAR) que almacenan los nombres y apellidos de los empleados. Se usa VARCHAR para permitir variabilidad en la longitud de estos campos.
- **Email**: Un campo VARCHAR más largo para almacenar direcciones de correo electrónico, que pueden variar en longitud.
- **Salario**: Tipo DECIMAL utilizado para almacenar el salario de los empleados con precisión de dos decimales, adecuado para valores monetarios.
- **Fecha_Nacimiento y Fecha_Contratacion**: DATE se utiliza para la fecha de nacimiento porque solo necesitamos la fecha, mientras que DATETIME se utiliza para la fecha de contratación para incluir tanto la fecha como la hora.
- **Tiempo_Parcial**: Un campo BOOLEAN para indicar si el empleado trabaja a tiempo parcial (TRUE) o a tiempo completo (FALSE).
- **Foto_Perfil**: Tipo BLOB para almacenar imágenes de perfil de los empleados, que son datos binarios.
- **Numero_Telefonico**: CHAR se utiliza para números telefónicos porque tienen una longitud fija.
- **Departamento y Posicion**: Campos VARCHAR para describir el departamento y la posición del empleado dentro de la empresa, respectivamente.
- **Hora_Entrada y Hora_Salida**: TIME se usa para registrar la hora de entrada y salida del trabajo.

Este ejemplo ilustra cómo se pueden usar diferentes tipos de datos en una tabla para

representar de manera eficiente y efectiva la información sobre los empleados. Seleccionar el tipo de dato adecuado para cada campo es crucial para optimizar el almacenamiento, el rendimiento y la integridad de los datos.

Uso de Fechas con Timestamp

El manejo de fechas y horas en bases de datos puede volverse complejo, especialmente cuando se trata de zonas horarias y formatos de fecha específicos. SQL proporciona tipos de datos como `TIMESTAMP` y funciones como `TO_TIMESTAMP_TZ` para ayudar a manejar estos casos con mayor precisión y flexibilidad. Aquí explicaremos cómo se utiliza `TIMESTAMP` y cómo convertir un texto dado a un valor de fecha y hora con zona horaria usando `TO_TIMESTAMP_TZ`, aplicando un formato como ISO 8601.

Uso de `TIMESTAMP`

El tipo de datos `TIMESTAMP` se utiliza para almacenar un punto específico en el tiempo, incluyendo la fecha y la hora. A diferencia del tipo `DATE` que solo incluye la fecha, `TIMESTAMP` puede almacenar la fecha junto con la hora, minutos, segundos y fracciones de segundo. Esto lo hace especialmente útil para registrar eventos o transacciones hasta con precisión de milisegundos, dependiendo del sistema de gestión de bases de datos (DBMS).

Ejemplo de `TIMESTAMP`:

```
CREATE TABLE eventos (
    id_evento INT PRIMARY KEY,
    nombre_evento VARCHAR(100),
    momento TIMESTAMP
);
```

En este ejemplo, la columna `momento` podría almacenar valores como `2024-04-07 20:30:00.123456`, lo que indica el año, mes, día, hora, minuto, segundo y milisegundos del evento.

Uso de `TO_TIMESTAMP_TZ`

`TO_TIMESTAMP_TZ` es una función utilizada para convertir cadenas de texto (strings) en valores de fecha y hora con información de zona horaria. Es particularmente útil cuando se trabaja con fuentes de datos que incluyen la zona horaria en el texto de la fecha y hora, y se desea preservar esta información al almacenarla en la base de datos.

El formato ISO 8601 es un estándar internacional para la representación de fechas y horas, y tiene la forma `YYYY-MM-DDTHH:MM:SSZ` para UTC, o `YYYY-MM-DDTHH:MM:SS+HH:MM` para

otras zonas horarias.

Ejemplo de `TO_TIMESTAMP_TZ` con formato ISO 8601:

Para convertir una fecha y hora en formato ISO 8601 a un valor `TIMESTAMP WITH TIME ZONE`:

```
SELECT TO_TIMESTAMP_TZ('2024-04-07T20:30:00+01:00', 'YYYY-MM-DD"T"HH24
```

En este ejemplo, la cadena `'2024-04-07T20:30:00+01:00'` se convierte en un valor de fecha y hora teniendo en cuenta la zona horaria (`+01:00`). La parte `T` en el formato simplemente se usa para separar la fecha de la hora, siguiendo el estándar ISO 8601. `TZH` y `TZM` se utilizan para especificar la zona horaria, donde `TZH` representa las horas de la zona horaria y `TZM` los minutos.

Consideraciones

- La precisión y soporte para `TIMESTAMP` y `TO_TIMESTAMP_TZ` pueden variar entre diferentes DBMS. Es importante revisar la documentación específica del sistema que estás utilizando.
- Mantener la información de la zona horaria puede ser crucial para aplicaciones que operan a nivel global, para asegurar que los eventos o transacciones se registren con la hora local correcta.
- Al trabajar con zonas horarias, es importante considerar el efecto del horario de verano (DST) y cómo el DBMS maneja estos cambios.

Este manejo avanzado de fechas y horas permite una gran flexibilidad y precisión en el almacenamiento y consulta de datos relacionados con el tiempo, facilitando el desarrollo de aplicaciones robustas y precisas en el manejo del tiempo.

Uso de Fechas con Zona Horaria

Utilizar `TO_TIMESTAMP_TZ` con nombres de zona horaria como `America/Mexico_City` permite manejar de manera más precisa las fechas y horas, especialmente teniendo en cuenta las variaciones por cambios de horario de verano. Sin embargo, es importante mencionar que el soporte específico para nombres de zonas horarias (en lugar de desplazamientos de zona horaria numéricos) y la función `TO_TIMESTAMP_TZ` pueden variar según el sistema de gestión de bases de datos (DBMS) que estés utilizando. En sistemas como Oracle, por ejemplo, puedes hacerlo de la siguiente manera:

```
SELECT TO_TIMESTAMP_TZ('2024-04-07 20:30:00 America/Mexico_City',
                       'YYYY-MM-DD HH24:MI:SS TZR') AS fecha_hora_con_
FROM DUAL;
```

En este ejemplo:

- `'2024-04-07 20:30:00 America/Mexico_City'` es la cadena de texto que representa la fecha, hora y zona horaria.
- `'YYYY-MM-DD HH24:MI:SS TZR'` es el formato de la cadena. `TZR` representa la zona horaria basada en regiones, como `America/Mexico_City`.
- `TO_TIMESTAMP_TZ` convierte la cadena en un valor `TIMESTAMP WITH TIME ZONE`, teniendo en cuenta la zona horaria especificada.

Este método considera automáticamente los ajustes por horario de verano según las reglas de la zona horaria especificada. Esto significa que, si la fecha y hora dadas caen dentro de un período de horario de verano, el ajuste correspondiente se aplicará al resultado.

Es esencial asegurarse de que el DBMS tenga la información más reciente sobre las zonas horarias, ya que las reglas para el horario de verano pueden cambiar. En sistemas como Oracle, esto puede implicar actualizar periódicamente la base de datos de zonas horarias con los últimos parches proporcionados por el fabricante.

Recuerda que este ejemplo está basado en cómo se haría en Oracle, y la sintaxis exacta o el soporte para los nombres de zonas horarias pueden variar en otros sistemas como PostgreSQL, MySQL, SQL Server, etc. Por lo tanto, es crucial consultar la documentación específica de tu DBMS para el manejo de fechas y horas con zonas horarias.

Uso de la tabla DUAL

`FROM DUAL` es una sintaxis que se encuentra comúnmente en bases de datos Oracle. `DUAL` es una tabla especial en Oracle que se utiliza específicamente para realizar operaciones que no requieren una tabla real para trabajar. Aunque inicialmente se introdujo en Oracle, algunos otros sistemas de gestión de bases de datos (DBMS) han adoptado conceptos similares para proporcionar funcionalidad equivalente.

Propósitos y Uso de DUAL

- **Realizar cálculos o transformaciones de datos sin referenciar una tabla real**: Puedes utilizar `DUAL` para ejecutar funciones o realizar cálculos que no dependen de los datos de una tabla específica. Por ejemplo, obtener la fecha y hora actual, realizar cálculos matemáticos, o llamar a funciones que retornan un valor específico.
- **Pruebas de funciones y expresiones SQL**: `DUAL` es útil para probar funciones y

expresiones SQL rápidamente sin necesidad de crear y acceder a una tabla real, lo cual es especialmente útil durante el desarrollo y depuración de consultas SQL.

Ejemplo de Uso

Supongamos que quieres obtener la fecha y hora actual en Oracle. Puedes hacerlo utilizando `DUAL` de la siguiente manera:

```
SELECT SYSDATE FROM DUAL;
```

`SYSDATE` es una función en Oracle que devuelve la fecha y hora actuales del sistema. Al seleccionar `SYSDATE FROM DUAL`, le estás diciendo a Oracle que ejecute esta función sin necesidad de consultar datos de una tabla específica.

En Otros Sistemas de Bases de Datos

En otros sistemas de gestión de bases de datos, como PostgreSQL o MySQL, la necesidad de `FROM DUAL` puede no ser tan directa porque estos sistemas permiten seleccionar valores sin referenciar una tabla. Por ejemplo, en PostgreSQL y MySQL, puedes simplemente hacer:

```
SELECT NOW(); -- PostgreSQL y MySQL para obtener la fecha y hora actu
```

En estos casos, no necesitas `FROM DUAL` para ejecutar la consulta. Sin embargo, Oracle requiere que especifiques `FROM` en tus consultas `SELECT`, por lo que `DUAL` se usa cuando no hay una tabla real de la que obtener los datos.

Conclusión

`FROM DUAL` es un artefacto útil en Oracle y en sistemas de bases de datos que adoptan una sintaxis similar, permitiendo a los desarrolladores y administradores de bases de datos ejecutar consultas que no requieren datos de una tabla. Facilita la ejecución de funciones, pruebas y cálculos rápidos directamente desde el SQL sin la necesidad de acceder a tablas con datos.

103. Subconsultas

Las subconsultas, también conocidas como consultas anidadas, son una herramienta poderosa en SQL que permite utilizar los resultados de una consulta como parte de otra. Son especialmente útiles para realizar comparaciones complejas, cálculos y para filtrar datos basados en los resultados de otra consulta. Vamos a desglosar el concepto y uso de las

subconsultas, enfocándonos en su aplicación sin el uso de `JOIN`.

Concepto de Subconsulta

Una subconsulta es, esencialmente, una consulta dentro de otra consulta. Puede aparecer en diversas partes de la consulta principal, incluyendo en la cláusula `SELECT`, `FROM`, o `WHERE`. Las subconsultas permiten realizar operaciones que serían difíciles o imposibles de lograr con una sola consulta o mediante el uso de joins.

Tipos de Subconsultas

- **Subconsultas en la cláusula `WHERE`**: Permiten filtrar los resultados de la consulta externa basándose en los resultados de la consulta interna. Pueden ser especialmente útiles para realizar comparaciones que implican valores máximos, mínimos, promedios, entre otros.
- **Subconsultas en la cláusula `FROM`**: Se utilizan para crear un conjunto de resultados temporal que luego se utiliza como una tabla desde la cual la consulta externa puede seleccionar. Es como si creases una tabla al vuelo basada en los resultados de otra consulta.
- **Subconsultas en la cláusula `SELECT`**: Estas subconsultas pueden usarse para incluir columnas calculadas en los resultados de la consulta principal que dependen de los valores obtenidos de otra consulta.

Ejemplos de Subconsulta

Subconsulta en `WHERE`

Supongamos que tienes una tabla `empleados` con los campos `id_empleado`, `nombre`, `departamento_id` y `salario`. Si quisieras encontrar los empleados que ganan más que el promedio de salario en su departamento, podrías usar una subconsulta en la cláusula `WHERE`:

```
SELECT nombre, salario
FROM empleados
WHERE salario > (
    SELECT AVG(salario)
    FROM empleados
    GROUP BY departamento_id
);
```

Subconsulta en `FROM`

Si necesitas listar el salario máximo de cada departamento, podrías hacer una subconsulta en la cláusula FROM para crear una vista temporal de los salarios máximos por departamento y luego seleccionar de esta vista:

```
SELECT d.nombre_departamento, m.salario_maximo
FROM (
    SELECT departamento_id, MAX(salario) as salario_maximo
    FROM empleados
    GROUP BY departamento_id
) AS m
JOIN departamentos d ON m.departamento_id = d.id_departamento;
```

Subconsulta en SELECT

Para incluir en tus resultados una columna calculada que indique, por ejemplo, el salario promedio de la empresa, puedes utilizar una subconsulta en la cláusula SELECT:

```
SELECT nombre, salario, (
    SELECT AVG(salario)
    FROM empleados
) AS salario_promedio
FROM empleados;
```

Consideraciones

- Las subconsultas deben estar encerradas entre paréntesis.
- Una subconsulta que aparece en la cláusula WHERE o SELECT debe retornar un único valor si se compara directamente con otro valor (a menos que se utilicen operadores como IN, ANY, ALL, etc.).
- Las subconsultas pueden impactar negativamente el rendimiento, especialmente si no están bien optimizadas o si operan sobre grandes conjuntos de datos. Es importante analizar y optimizar las subconsultas para asegurar un rendimiento adecuado.

Las subconsultas ofrecen una flexibilidad increíble en la manipulación y análisis de datos, permitiendo construir consultas que de otra manera requerirían múltiples pasos o incluso procedimientos almacenados. Sin embargo, su uso debe ser cuidadoso para evitar impactos negativos en el rendimiento.

104. Consultas anidadas

Las consultas anidadas usando JOIN son una técnica poderosa en SQL que permite combinar filas de dos o más tablas basadas en una columna relacionada entre ellas. A

diferencia de las subconsultas simples, que generalmente se utilizan para realizar operaciones de selección o condición dentro de una misma tabla, las consultas anidadas con JOIN amplían esta funcionalidad permitiendo realizar operaciones más complejas que involucran múltiples tablas. Esto es particularmente útil para extraer datos relacionados que están distribuidos en diferentes tablas de una base de datos.

Concepto de Consultas Anidadas con `JOIN`

Una consulta anidada con JOIN implica utilizar una subconsulta (una consulta dentro de otra consulta) como una de las tablas en una operación de JOIN. Esta técnica es útil cuando necesitas filtrar o procesar datos de una tabla antes de unirlos con otra tabla. Esencialmente, permite pre-seleccionar, filtrar o transformar datos de una tabla en una subconsulta antes de realizar el JOIN.

Ejemplos de Consultas Anidadas Usando `JOIN`

Imagina que tienes dos tablas: `empleados` (con campos `id_empleado`, `nombre`, y `departamento_id`) y `departamentos` (con campos `id_departamento` y `nombre_departamento`). Quieres obtener una lista de empleados junto con el nombre de su departamento, pero solo para aquellos departamentos que tienen más de 10 empleados.

Usando una Subconsulta en `JOIN`

```
SELECT e.nombre, d.nombre_departamento
FROM empleados e
JOIN (
    SELECT id_departamento
    FROM empleados
    GROUP BY departamento_id
    HAVING COUNT(id_empleado) > 10
) AS deptos_grandes ON e.departamento_id = deptos_grandes.id_departame
JOIN departamentos d ON e.departamento_id = d.id_departamento;
```

En este ejemplo, la subconsulta selecciona los `id_departamento` que tienen más de 10 empleados. Luego, esta subconsulta se une con la tabla `empleados` para filtrar solo aquellos empleados que pertenecen a estos departamentos. Finalmente, se realiza otro JOIN con la tabla `departamentos` para obtener el nombre del departamento.

Ventajas de las Consultas Anidadas con `JOIN`

- **Flexibilidad**: Permiten un alto grado de flexibilidad al permitirte pre-procesar los datos de una tabla antes de unirlos con otra tabla.

- **Eficiencia**: Pueden ser más eficientes que realizar múltiples consultas separadas o complejas condiciones en las cláusulas `WHERE` o `HAVING`, especialmente si las subconsultas reducen significativamente el tamaño del conjunto de datos antes del `JOIN`.
- **Organización**: Ayudan a organizar y simplificar consultas complejas, haciendo que el proceso de desarrollo y depuración de consultas sea más manejable.

Consideraciones

- **Rendimiento**: Aunque las consultas anidadas con `JOIN` pueden ser poderosas, también pueden afectar el rendimiento si no se utilizan cuidadosamente. Es crucial analizar y optimizar estas consultas, especialmente en bases de datos grandes.
- **Legibilidad**: A medida que las consultas se vuelven más complejas, también puede disminuir su legibilidad. Mantener una estructura clara y comentarios adecuados puede ayudar a mitigar este problema.

Las consultas anidadas con `JOIN` son una herramienta avanzada en el arsenal de SQL que, cuando se usan correctamente, pueden proporcionar soluciones eficientes y elegantes para problemas de datos complejos.

Uso de HAVING

La cláusula `HAVING` en SQL es una herramienta poderosa usada para filtrar los resultados de una consulta que utiliza la cláusula `GROUP BY`. Mientras que `WHERE` filtra filas antes de que se agrupen, `HAVING` filtra los grupos creados por `GROUP BY` basándose en una condición especificada. Esto es especialmente útil para trabajar con funciones de agregación, donde deseas aplicar condiciones a un conjunto de datos agregados, como la suma total, el promedio, el conteo, etc.

Uso Básico de **`HAVING`**

Supongamos que tienes una tabla llamada `ventas` con las siguientes columnas: `id_venta`, `fecha`, `monto` y `vendedor_id`. Si quisieras encontrar los vendedores que han generado ingresos por encima de un cierto umbral, podrías usar `GROUP BY` para agrupar las ventas por `vendedor_id` y luego `HAVING` para filtrar solo aquellos grupos que cumplan con tu criterio de ingresos.

```sql
SELECT vendedor_id, SUM(monto) AS total_ventas
FROM ventas
GROUP BY vendedor_id
HAVING SUM(monto) > 10000;
```

En este ejemplo, primero se agrupan las ventas por `vendedor_id`, luego se suma el `monto` de las ventas para cada vendedor, y finalmente, la cláusula `HAVING` filtra y muestra solo aquellos vendedores cuyo total de ventas supera los 10,000.

Diferencia Entre `WHERE` y `HAVING`

- `WHERE` se utiliza para filtrar filas antes de que sean agrupadas por `GROUP BY`.
- `HAVING` se utiliza para filtrar grupos después de que han sido formados por `GROUP BY`.

Es importante notar que puedes usar `WHERE` y `HAVING` en la misma consulta. `WHERE` filtrará los datos antes de que sean agrupados, y `HAVING` aplicará un filtro adicional a los grupos resultantes.

Ejemplo Combinando `WHERE` y `HAVING`

Siguiendo el ejemplo anterior, si solo te interesan las ventas realizadas en el año 2023 y además quieres aplicar el filtro de ingresos, podrías escribir la consulta de la siguiente manera:

```sql
SELECT vendedor_id, SUM(monto) AS total_ventas
FROM ventas
WHERE YEAR(fecha) = 2023
GROUP BY vendedor_id
HAVING SUM(monto) > 10000;
```

Aquí, la cláusula `WHERE` primero asegura que solo las ventas del año 2023 sean consideradas. Luego, `GROUP BY` agrupa estas ventas por `vendedor_id`, y `HAVING` filtra estos grupos para mostrar solo aquellos vendedores cuyo total de ventas excede los 10,000.

Consideraciones

- `HAVING` puede usarse sin `GROUP BY` para filtrar resultados agregados a nivel global. Sin embargo, su uso más común y poderoso es en combinación con `GROUP BY` para aplicar condiciones a grupos específicos.
- Al igual que con otras cláusulas de SQL, el rendimiento de las consultas que usan `HAVING` puede variar significativamente con el tamaño del conjunto de datos y la complejidad de las condiciones de filtro. Es importante considerar la optimización de consultas y posiblemente el uso de índices para mejorar el rendimiento.

La cláusula `HAVING` es una herramienta esencial en SQL para el análisis de datos agregados, permitiendo a los usuarios realizar consultas complejas y obtener insights significativos de grandes conjuntos de datos.

Uso de WITH

La cláusula `WITH`, también conocida como Common Table Expressions (CTE), es una característica muy útil en SQL que permite crear una o más tablas temporales que existen únicamente durante la ejecución de la consulta. Los CTE ofrecen una manera de organizar y simplificar consultas SQL complejas, haciéndolas más legibles y, en muchos casos, mejorando el rendimiento. Son especialmente útiles en situaciones que involucran consultas recursivas, consultas con múltiples pasos que requieren el uso de los mismos datos intermedios varias veces, o simplemente para hacer que la consulta sea más comprensible separando en pasos lógicos.

Estructura Básica de `WITH`

La estructura básica de un CTE es la siguiente:

```sql
WITH NombreCTE AS (
    -- Aquí va tu consulta SQL
)
SELECT * FROM NombreCTE;
```

Puedes definir varios CTE en una sola consulta, separándolos con comas:

```sql
WITH CTE1 AS (
    -- Consulta 1
), CTE2 AS (
    -- Consulta 2 que podría incluso referenciar a CTE1
)
SELECT * FROM CTE1 JOIN CTE2 ON CTE1.columna = CTE2.columna;
```

Ejemplo Práctico de Uso de `WITH`

Imagina que estás trabajando con una base de datos de una tienda en línea y quieres obtener los nombres de los productos más vendidos junto con la información de los proveedores. Supongamos que las ventas están registradas en una tabla `ventas` y la información del producto está en una tabla `productos`. Podrías usar `WITH` para simplificar la consulta de la siguiente manera:

```sql
WITH VentasTotales AS (
    SELECT producto_id, SUM(cantidad) AS total_vendido
    FROM ventas
    GROUP BY producto_id
), ProductosMasVendidos AS (
    SELECT p.nombre, p.proveedor_id, v.total_vendido
    FROM productos p
    JOIN VentasTotales v ON p.id = v.producto_id
    WHERE v.total_vendido > 100 -- Suponiendo que quieres productos co
)
SELECT pmv.nombre, proveedor.nombre
FROM ProductosMasVendidos pmv
JOIN proveedores ON pmv.proveedor_id = proveedores.id;
```

Ventajas del Uso de `WITH`

- **Legibilidad**: Al separar las consultas en bloques lógicos, `WITH` hace que las consultas complejas sean más fáciles de entender y mantener.
- **Reusabilidad**: Permite reutilizar los resultados de una subconsulta múltiples veces en una consulta más grande sin necesidad de duplicar código.
- **Facilita consultas recursivas**: Los CTE recursivos son una forma potente de tratar con datos jerárquicos, como estructuras de árboles o grafos, que serían difíciles o imposibles de manejar con consultas SQL estándar.

Consideraciones

- **Desempeño**: Aunque los CTE pueden mejorar la legibilidad de las consultas, su impacto en el rendimiento depende de cómo el sistema de gestión de base de datos (DBMS) optimiza la ejecución de la consulta. En algunos casos, un CTE puede ser menos eficiente que una subconsulta equivalente, mientras que en otros, puede ser igual o incluso más eficiente.
- **Soporte del DBMS**: La disponibilidad y el comportamiento específico de `WITH` y los CTE pueden variar entre diferentes sistemas de bases de datos. Es importante revisar la documentación del DBMS que estás utilizando.

En resumen, los CTE ofrecen una manera flexible y potente de estructurar consultas SQL, haciéndolas más claras y, en muchos casos, más eficientes. Su capacidad para desglosar consultas complejas en partes más manejables los convierte en una herramienta valiosa para cualquier desarrollador o analista de bases de datos.

105. Uso de Índices

El uso de índices en bases de datos es una técnica fundamental para mejorar la eficiencia de

las operaciones de búsqueda, consulta y actualización. Los índices son estructuras de datos especiales que el sistema de gestión de bases de datos (DBMS) mantiene para permitir accesos rápidos a las filas de una tabla. Funcionan de manera similar al índice de un libro: en lugar de leer todo el libro (o tabla) para encontrar un capítulo (o fila) específico, miras en el índice para encontrar rápidamente la página (o posición de fila) que necesitas.

Cómo Funcionan los Índices

Los índices son creados en columnas específicas de una tabla. Estas columnas son generalmente las que se usan con frecuencia en cláusulas `WHERE`, `JOIN`, o como parte de una operación de ordenamiento (`ORDER BY`). Al crear un índice en una columna, el DBMS construye una estructura de datos interna (como un árbol B, un árbol B+, o un hash) que mapea los valores de esa columna a las direcciones físicas o lógicas de las filas correspondientes en el disco.

Tipos de Índices

- **Índices únicos**: Aseguran que no haya dos filas con el mismo valor en la columna o columnas indexadas. Son útiles para mantener la integridad de los datos, como en el caso de un ID de usuario.
- **Índices no únicos**: Permiten valores duplicados y son útiles para mejorar el rendimiento de las consultas en columnas donde los datos tienden a repetirse.
- **Índices compuestos**: Se crean sobre más de una columna de una tabla. Son especialmente útiles cuando las consultas frecuentemente involucran operaciones sobre varias columnas.

Ventajas del Uso de Índices

- **Mejora del rendimiento de las consultas**: Los índices pueden reducir significativamente el tiempo necesario para buscar y recuperar datos de una tabla.
- **Eficiencia en las operaciones de `JOIN`**: Los índices en las columnas utilizadas en `JOIN` pueden hacer que estas operaciones sean mucho más rápidas.
- **Mejora en la velocidad de ordenamiento de datos**: Los índices pueden acelerar las operaciones de ordenamiento especificadas en `ORDER BY`.

Consideraciones al Usar Índices

- **Costo de mantenimiento**: Cada vez que se inserta, actualiza o elimina una fila en la tabla, todos los índices en esa tabla deben ser actualizados. Esto puede llevar a un sobrecoste en operaciones de escritura.
- **Uso de espacio**: Los índices ocupan espacio en el disco, que puede ser significativo dependiendo del tamaño de la tabla y el tipo de índice.

- **Selección de columnas**: No todas las columnas son buenos candidatos para la indexación. Generalmente, se deben indexar las columnas que se usan frecuentemente en las cláusulas `WHERE`, `JOIN`, o `ORDER BY`.

Creación de un Índice

La sintaxis para crear un índice varía entre los sistemas de DBMS, pero el concepto es generalmente el mismo. Aquí tienes un ejemplo básico en SQL:

```sql
CREATE INDEX idx_columna ON tabla(columna);
```

Este comando crea un índice no único en la `columna` de la `tabla` especificada.

Conclusión

El uso adecuado de índices es crucial para el rendimiento de las bases de datos. Sin embargo, es importante usarlos de manera estratégica, ya que un exceso de índices o índices inapropiados pueden tener un impacto negativo en el rendimiento de la base de datos, especialmente en operaciones de escritura. La clave está en encontrar el equilibrio correcto entre la velocidad de lectura y la eficiencia de escritura, teniendo en cuenta las necesidades específicas de tu aplicación.

Ejemplo de uso de Índices

Vamos a ilustrar el uso de índices con un ejemplo práctico en un escenario de base de datos típico. Imaginemos que tenemos una base de datos de una librería en línea, con una tabla llamada `Pedidos` que registra todas las transacciones de los clientes. La tabla `Pedidos` tiene las siguientes columnas:

- `id_pedido` (clave primaria)
- `id_cliente`
- `fecha_pedido`
- `monto_total`
- `estado`

Supongamos que los usuarios de la base de datos a menudo necesitan buscar pedidos por `id_cliente` y `fecha_pedido`, y también filtrar por el `estado` del pedido. Sin embargo, la tabla `Pedidos` ha crecido significativamente y contiene millones de registros, haciendo que las consultas sean lentas, especialmente aquellas que buscan pedidos de un cliente específico dentro de un rango de fechas.

Creación de Índices

Para mejorar el rendimiento de estas consultas, podemos crear índices en las columnas `id_cliente`, `fecha_pedido` y `estado`.

Índice en `id_cliente`

```
CREATE INDEX idx_id_cliente ON Pedidos(id_cliente);
```

Este índice ayudará a mejorar el rendimiento de las consultas que buscan pedidos por el `id_cliente`. Por ejemplo:

```
SELECT * FROM Pedidos WHERE id_cliente = 12345;
```

Índice en `fecha_pedido`

```
CREATE INDEX idx_fecha_pedido ON Pedidos(fecha_pedido);
```

Este índice facilitará la búsqueda eficiente de pedidos por fecha. Es particularmente útil para consultas que involucran rangos de fechas:

```
SELECT * FROM Pedidos WHERE fecha_pedido BETWEEN '2024-01-01' AND '202
```

Índice Compuesto en `id_cliente` y `fecha_pedido`

Si las consultas comunes implican buscar por `id_cliente` y filtrar por un rango de `fecha_pedido`, podría ser beneficioso crear un índice compuesto que cubra ambas columnas:

```
CREATE INDEX idx_cliente_fecha ON Pedidos(id_cliente, fecha_pedido);
```

Este índice compuesto es óptimo para consultas como:

```
SELECT * FROM Pedidos WHERE id_cliente = 12345 AND fecha_pedido BETWEE
```

Índice en `estado`

```sql
CREATE INDEX idx_estado ON Pedidos(estado);
```

Este índice mejora las consultas que filtran por el `estado` del pedido, como buscar todos los pedidos que están "Enviados":

```sql
SELECT * FROM Pedidos WHERE estado = 'Enviado';
```

Consideraciones

Al decidir qué índices crear, es importante considerar cómo se usan los datos:

- **Análisis de consultas**: Identifica las consultas más frecuentes o críticas para el rendimiento y diseña índices que las optimicen.
- **Índices compuestos**: Siempre se utilizan de izquierda a derecha en su definición. Asegúrate de que el orden de las columnas en el índice compuesto refleje el uso común en las consultas.
- **Mantenimiento de índices**: Los índices incrementan el tiempo y los recursos necesarios para las operaciones de escritura (`INSERT`, `UPDATE`, `DELETE`) porque el DBMS debe actualizar los índices además de los datos. Evalúa el impacto en el rendimiento de escritura frente a las ganancias en rendimiento de lectura.

Este ejemplo ilustra cómo los índices pueden ser una herramienta poderosa para mejorar el rendimiento de las consultas en bases de datos grandes, permitiendo búsquedas y filtrados mucho más rápidos en columnas clave.

Índices únicos

Para asegurar que los valores en una columna (o un conjunto de columnas) sean únicos en una tabla, puedes crear un índice único. Un índice único no solo mejora el rendimiento de las consultas sobre esas columnas, sino que también previene la inserción de filas con valores duplicados en las columnas indexadas, lo cual es útil para mantener la integridad de los datos.

Ejemplo de Creación de un Índice Único

Supongamos que tienes una tabla llamada `Usuarios` con las siguientes columnas:

- `id_usuario` (clave primaria)
- `nombre_usuario`
- `email`
- `fecha_registro`

Quieres asegurarte de que el `email` de cada usuario sea único en la tabla `Usuarios`, porque cada dirección de correo electrónico debe estar asociada con un solo usuario. Para lograr esto, puedes crear un índice único en la columna `email`.

Crear un Índice Único en `email`

```
CREATE UNIQUE INDEX idx_email_unico ON Usuarios(email);
```

Este comando crea un índice único llamado `idx_email_unico` en la columna `email` de la tabla `Usuarios`. Si intentas insertar o actualizar filas de manera que dos filas tengan el mismo valor en la columna `email`, el sistema de gestión de bases de datos (DBMS) rechazará la operación y mostrará un error.

Uso Práctico

El uso de índices únicos es particularmente importante en campos que identifican de manera exclusiva a los registros, como pueden ser:

- Direcciones de correo electrónico en una tabla de usuarios.
- Números de identificación nacional o pasaporte en registros de personas.
- Nombres de usuario o IDs de inicio de sesión.

Intento de Inserción Duplicada

Aquí hay un ejemplo de lo que sucedería si intentas insertar una fila con un valor duplicado en la columna `email`:

```
INSERT INTO Usuarios (nombre_usuario, email, fecha_registro)
VALUES ('NuevoUsuario', 'usuarioexistente@email.com', '2024-01-01');
```

Si ya existe una fila en `Usuarios` con el `email` "usuarioexistente@email.com", esta operación de inserción fallará debido al índice único.

Beneficios de los Índices Únicos

- **Integridad de Datos**: Asegura que no se pueden insertar datos duplicados en las columnas clave, lo que es crucial para mantener la integridad de los datos.
- **Mejora del Rendimiento**: Optimiza las operaciones de búsqueda, actualización y eliminación de datos al proporcionar una vía rápida para localizar registros únicos.

Consideraciones

- Al diseñar tu base de datos, es importante considerar cuidadosamente qué campos deberían ser únicos para evitar errores y problemas de integridad de datos.
- Aunque los índices únicos son herramientas poderosas para la integridad de los datos, también añaden un sobrecoste a las operaciones de inserción y actualización, ya que el DBMS debe verificar la unicidad cada vez que se modifican los datos.

Este enfoque para garantizar valores únicos a través de índices es ampliamente soportado por la mayoría de los sistemas de gestión de bases de datos, incluyendo MySQL, PostgreSQL, SQL Server, Oracle, y muchos otros.

Índices compuestos

Un índice compuesto es un índice que abarca más de una columna en una tabla. Crear un índice compuesto puede ser muy útil cuando las consultas frecuentemente involucran operaciones de filtrado, ordenamiento o join que abarcan múltiples columnas. Al incluir varias columnas en un único índice, el sistema de gestión de bases de datos (DBMS) puede mejorar significativamente el rendimiento de estas consultas.

Ejemplo de Creación de un Índice Compuesto

Consideremos una tabla `Empleados` que tiene, entre otros, los campos `Nombre` y `Apellido`. Supongamos que comúnmente realizas consultas que buscan empleados usando tanto el nombre como el apellido. En este caso, un índice compuesto en ambas columnas puede mejorar el rendimiento de estas consultas.

Creación de la Tabla `Empleados`

Imaginemos que la tabla se define así:

```
CREATE TABLE Empleados (
    id_empleado INT PRIMARY KEY,
    Nombre VARCHAR(50),
    Apellido VARCHAR(50),
    Departamento VARCHAR(50),
    Fecha_Contratacion DATE
);
```

Crear un Índice Compuesto en `Nombre` y `Apellido`

Para crear un índice compuesto en las columnas `Nombre` y `Apellido`, podrías usar el siguiente comando SQL:

```
CREATE INDEX idx_nombre_apellido ON Empleados(Nombre, Apellido);
```

Este comando crea un índice compuesto llamado `idx_nombre_apellido` que abarca las columnas `Nombre` y `Apellido`. Ahora, cuando ejecutes consultas que busquen empleados por nombre y apellido, el DBMS podrá utilizar este índice para realizar la búsqueda de manera más eficiente.

Uso Práctico del Índice Compuesto

El índice compuesto `idx_nombre_apellido` es particularmente útil para consultas como:

```
SELECT * FROM Empleados WHERE Nombre = 'Juan' AND Apellido = 'Pérez';
```

O para consultas que ordenan los resultados por las columnas indexadas:

```
SELECT * FROM Empleados WHERE Departamento = 'Ventas' ORDER BY Nombre,
```

Aunque la cláusula `WHERE` solo filtre por `Departamento`, el índice compuesto puede mejorar el rendimiento del ordenamiento (`ORDER BY`) si el filtrado resulta en un subconjunto de filas suficientemente pequeño.

Beneficios de los Índices Compuestos

- **Eficiencia en Consultas**: Los índices compuestos pueden hacer que las consultas que filtran o ordenan basándose en múltiples columnas sean mucho más rápidas.
- **Optimización de `JOIN`**: También pueden optimizar las operaciones de `JOIN` que implican las columnas indexadas.
- **Mejora en el Ordenamiento de Datos**: Aceleran las operaciones de ordenamiento que usan las columnas indexadas.

Consideraciones

- **Orden de las Columnas**: El orden de las columnas en un índice compuesto es crucial. El índice será más efectivo para las consultas que utilicen las columnas en el mismo orden en que se definió el índice.
- **Uso de Espacio y Rendimiento de Escritura**: Al igual que con todos los índices, los índices compuestos ocupan espacio adicional en el disco y pueden hacer que las operaciones de inserción, actualización y eliminación sean más lentas debido a la necesidad de mantener el índice actualizado.

Los índices compuestos son herramientas potentes para la optimización de consultas en bases de datos relacionales, pero deben usarse con cuidado, teniendo en cuenta el impacto en el espacio y el rendimiento de escritura.

Índices compuestos y únicos

Puedes hacer que un índice compuesto sea único. Esto garantizará que la combinación de valores en las columnas indexadas sea única en toda la tabla, lo cual es útil cuando quieres prevenir registros duplicados basados en un conjunto específico de columnas.

Creación de un Índice Compuesto Único

Para seguir con el ejemplo anterior de la tabla `Empleados` y crear un índice compuesto único para las columnas `Nombre` y `Apellido`, usarías el siguiente comando SQL:

```
CREATE UNIQUE INDEX idx_nombre_apellido_unico ON Empleados(Nombre, Ape
```

Este comando crea un índice único llamado `idx_nombre_apellido_unico` que cubre las columnas `Nombre` y `Apellido`. Con este índice en su lugar, cualquier intento de insertar o actualizar filas en la tabla `Empleados` que resulte en una combinación de `Nombre` y `Apellido` duplicada será rechazado por el sistema de gestión de bases de datos.

Ejemplo Práctico

Consideremos que ya tienes un empleado llamado "Juan Pérez" en la tabla `Empleados`. Si intentas insertar otro empleado con el mismo nombre y apellido:

```
INSERT INTO Empleados (Nombre, Apellido, Departamento, Fecha_Contratac
VALUES ('Juan', 'Pérez', 'Desarrollo', '2024-01-01');
```

Este intento de inserción fallará con un error de violación de índice único, ya que ya existe un empleado con el `Nombre` "Juan" y el `Apellido` "Pérez".

Beneficios de los Índices Compuestos Únicos

- **Integridad de Datos**: Aseguran que no puedan existir dos filas con la misma combinación de valores en las columnas indexadas, lo cual es importante para evitar duplicados y mantener la integridad de los datos.
- **Rendimiento de Consulta**: Al igual que otros índices, un índice compuesto único también puede mejorar el rendimiento de las consultas que buscan o filtran datos basados en las columnas indexadas.

Consideraciones

- **Selección de Columnas**: Antes de crear un índice compuesto único, asegúrate de que la combinación de columnas realmente necesite ser única. Considera las reglas de negocio y los requisitos de la aplicación.
- **Impacto en Operaciones de Escritura**: Mientras que los índices únicos protegen contra datos duplicados, también incurren en un chequeo adicional durante las operaciones de inserción y actualización, lo que puede afectar el rendimiento de estas operaciones.
- **Uso de Espacio**: Los índices ocupan espacio adicional en el almacenamiento de la base de datos. Los índices compuestos únicos, al incluir múltiples columnas, pueden ocupar más espacio que los índices simples.

Crear índices compuestos únicos es una práctica poderosa para la gestión de bases de datos, especialmente cuando se necesita asegurar la unicidad a través de múltiples campos. Esto permite a los diseñadores de bases de datos imponer restricciones de integridad de datos a nivel de estructura de datos, complementando las reglas de negocio aplicadas en la capa de aplicación.

106. Estrategias para mejorar el rendimiento

Mejorar el rendimiento de las bases de datos es crucial para mantener aplicaciones ágiles y eficientes, especialmente a medida que las bases de datos crecen en tamaño y complejidad. Aquí se describen varias estrategias fundamentales que pueden ayudar a optimizar el rendimiento de las consultas SQL y la gestión general de las bases de datos.

1. Uso Efectivo de Índices

- **Crear Índices**: Los índices son esenciales para acelerar las operaciones de búsqueda y consulta. Identifica las columnas que se utilizan con frecuencia en las cláusulas `WHERE`, `JOIN`, o `ORDER BY` para indexarlas.
- **Índices Únicos y Compuestos**: Utiliza índices únicos para asegurar la unicidad de los datos y considera índices compuestos para consultas que involucran múltiples columnas.
- **Evitar Índices Innecesarios**: Cada índice adicional consume recursos y ralentiza las operaciones de escritura (`INSERT`, `UPDATE`, `DELETE`). Mantén solo los índices que benefician el rendimiento de las consultas.

2. Optimización de Consultas

- **Minimizar el Uso de `SELECT *`**: Especifica solo las columnas que necesitas en tus consultas `SELECT` para reducir la cantidad de datos transferidos y procesados.

- **Usar Cláusulas `WHERE` Efectivas**: Filtra los resultados lo más pronto posible en tus consultas para reducir el tamaño del conjunto de resultados.
- **Dividir Consultas Complejas**: Las consultas muy complejas pueden ser más lentas de ejecutar. Considera dividirlas en subconsultas más manejables si es posible.

3. Normalización y Desnormalización

- **Normalización**: Diseña tu base de datos para eliminar la redundancia de datos y asegurar la integridad mediante la normalización. Esto puede mejorar el rendimiento de escritura y simplificar el mantenimiento de la base de datos.
- **Desnormalización**: En algunos casos, la desnormalización (agregar redundancia deliberadamente) puede mejorar el rendimiento de lectura al reducir el número de `JOIN` necesarios.

4. Particionamiento de Tablas

- **Particionamiento**: Divide una tabla grande en partes más manejables basadas en ciertos criterios (como fechas o regiones geográficas). Esto puede mejorar significativamente el rendimiento de las consultas que pueden operar en una partición individual.

5. Caching

- **Cache de Resultados**: Implementa caching en el nivel de aplicación para almacenar los resultados de consultas que se ejecutan con frecuencia, reduciendo así la carga en la base de datos.

6. Gestión de Conexiones

- **Pool de Conexiones**: Utiliza un pool de conexiones para reducir el overhead de establecer y cerrar conexiones a la base de datos para cada operación.

7. Actualizaciones de Hardware

- **Mejoras de Hardware**: Aumentar los recursos de hardware (como la memoria RAM, CPU más rápida, o almacenamiento SSD) puede ofrecer mejoras significativas en el rendimiento.

8. Análisis y Monitorización

- **Profiling de Consultas**: Utiliza herramientas de profiling para identificar cuellos de botella y consultas ineficientes.
- **Monitorización**: Implementa soluciones de monitorización para rastrear el rendimiento

de la base de datos y identificar problemas en tiempo real.

Conclusión

La optimización del rendimiento de las bases de datos es un proceso continuo que implica ajustes tanto en el diseño como en las consultas. Combinando múltiples estrategias, es posible manejar eficientemente grandes volúmenes de datos y mantener el rendimiento de las aplicaciones en niveles óptimos. Es crucial evaluar el impacto de cada cambio, probar exhaustivamente y monitorizar el rendimiento de manera regular.

Ejemplo de la Sesión

Se han diseñado dos tablas para almacenar la información de frutas y sus últimos precios conocidos o esperados.

```
create table frutas_info (
    id number,
    nombre varchar(255) not null,
    descripcion varchar(255),
    constraint pk_frutas_info primary key (id)
);

create table frutas_precio (
    id number,
    fruta_id number not null,
    precio float not null,
    fecha timestamp not null,
    constraint pk_frutas_precio primary key (id),
    constraint fk_fruta_id foreign key (fruta_id)
        references frutas_info (id)
);
```

La estructura proporcionada define dos tablas destinadas a almacenar información sobre frutas y sus precios a lo largo del tiempo. Vamos a desglosar cada una de estas tablas y su propósito, así como la relación entre ellas.

Tabla `frutas_info`

Esta tabla está diseñada para almacenar información básica sobre diferentes tipos de frutas. Cada fruta tiene un `id` único, un `nombre`, y una `descripcion` opcional.

- **id**: Es un campo numérico que actúa como clave primaria (`PRIMARY KEY`) de la tabla. Esto significa que cada valor en esta columna debe ser único y no nulo, identificando

de manera unívoca a cada registro de fruta en la tabla.
- **nombre**: Un campo de texto (VARCHAR) que almacena el nombre de la fruta. Este campo es obligatorio (NOT NULL), lo que significa que cada fruta debe tener un nombre.
- **descripcion**: Un campo de texto opcional (VARCHAR) que puede usarse para almacenar una descripción adicional de la fruta.

La restricción PRIMARY KEY asegura que el id de cada fruta sea único e identifique claramente cada fila.

Tabla frutas_precio

Esta tabla está destinada a almacenar información sobre los precios de las frutas a lo largo del tiempo, permitiendo registrar múltiples precios para la misma fruta en diferentes fechas.

- **id**: Un campo numérico que sirve como clave primaria de la tabla, asegurando que cada registro de precio sea único.
- **fruta_id**: Este campo numérico establece una relación con la tabla frutas_info, especificando la fruta a la que se refiere el precio. La restricción NOT NULL garantiza que cada precio registrado esté vinculado a una fruta.
- **precio**: Un campo de tipo FLOAT para almacenar el precio de la fruta. Es un campo obligatorio.
- **fecha**: Un campo de tipo TIMESTAMP que registra la fecha y hora exacta en que se marcó el precio. También es obligatorio.

Las restricciones definidas en la tabla son:

- **PRIMARY KEY (pk_frutas_precio)**: Asegura que el id de cada registro de precio sea único.
- **FOREIGN KEY (fk_fruta_id)**: Establece una relación de clave foránea con la columna id de la tabla frutas_info, asegurando que cada fruta_id en frutas_precio corresponda a un id válido en frutas_info. Esto mantiene la integridad referencial entre las tablas, lo que significa que no puedes tener un precio referenciado a una fruta que no exista en la tabla frutas_info.

Relación entre las Tablas

La relación establecida por la clave foránea (FOREIGN KEY) permite que la base de datos mantenga la integridad de los datos entre las dos tablas. Por ejemplo, no podrás insertar un registro en frutas_precio que haga referencia a una fruta_id que no exista en frutas_info, y si intentas eliminar una fruta de frutas_info que ya tiene precios asociados en frutas_precio, la operación será rechazada a menos que se manejen adecuadamente las dependencias (por ejemplo, borrando primero los registros de precios

relacionados o utilizando políticas de borrado en cascada, si el DBMS lo permite).

Este diseño permite un almacenamiento eficiente de los datos, donde la información fundamental de cada fruta se almacena una sola vez en `frutas_info`, mientras que `frutas_precio` puede contener múltiples registros de precios para cada fruta a lo largo del tiempo, capturando la evolución del precio de cada fruta.

Inserción de datos

Se insertaron datos mediante la consulta `INSERT` para proveer una lista inicial de frutas y sus precios conocidos:

```
insert into frutas_info (id, nombre, descripcion) values (1, 'Manzana
insert into frutas_info (id, nombre, descripcion) values (2, 'Pera Bar
insert into frutas_info (id, nombre, descripcion) values (3, 'Plátano
insert into frutas_info (id, nombre, descripcion) values (4, 'Naranja
insert into frutas_info (id, nombre, descripcion) values (5, 'Piña Mie
insert into frutas_info (id, nombre, descripcion) values (6, 'Uva Crim

insert into frutas_precio (id, fruta_id, precio, fecha) values (1, 1,
insert into frutas_precio (id, fruta_id, precio, fecha) values (2, 2,
insert into frutas_precio (id, fruta_id, precio, fecha) values (3, 2,
insert into frutas_precio (id, fruta_id, precio, fecha) values (4, 2,
insert into frutas_precio (id, fruta_id, precio, fecha) values (5, 3,
insert into frutas_precio (id, fruta_id, precio, fecha) values (6, 4,
insert into frutas_precio (id, fruta_id, precio, fecha) values (7, 4,
insert into frutas_precio (id, fruta_id, precio, fecha) values (8, 5,
insert into frutas_precio (id, fruta_id, precio, fecha) values (9, 5,
```

Las consultas proporcionadas son instrucciones `INSERT` que añaden registros a dos tablas relacionadas, `frutas_info` y `frutas_precio`. Vamos a desglosar el propósito y la función de estas consultas.

Consultas para `frutas_info`

Estas consultas insertan registros en la tabla `frutas_info`, que está diseñada para almacenar información básica sobre diferentes tipos de frutas. Cada registro incluye un `id` único, un `nombre` para la fruta y una `descripcion` detallada de la misma.

```
insert into frutas_info (id, nombre, descripcion) values (1, 'Manzana
...
insert into frutas_info (id, nombre, descripcion) values (6, 'Uva Crim
```

Cada fila representa un tipo de fruta distinto, con descripciones que ayudan a identificar sus características únicas. Por ejemplo, la `Manzana Golden` se describe como una manzana de "piel amarilla brillante, dulce y crujiente".

Consultas para `frutas_precio`

Estas consultas insertan registros en la tabla `frutas_precio`, que almacena información sobre los precios de las frutas registradas en `frutas_info` en diferentes momentos. Cada registro incluye un `id` único, un `fruta_id` que vincula el precio a una fruta específica en `frutas_info`, un `precio` y una `fecha` que indica cuándo se registró ese precio.

```
insert into frutas_precio (id, fruta_id, precio, fecha) values (1, 1,
...
insert into frutas_precio (id, fruta_id, precio, fecha) values (9, 5,
```

Los precios se registran con timestamps precisos. En algunas consultas se utiliza `CURRENT_TIMESTAMP` para registrar el precio con el timestamp actual del sistema. En otros casos, se utiliza `TO_TIMESTAMP_TZ` para especificar manualmente la fecha y hora, incluyendo la zona horaria (por ejemplo, `'2024-05-01 12:00:00 America/Mexico_City'`), lo cual es útil para garantizar la precisión en ambientes distribuidos geográficamente o para registrar precios históricos.

Propósito y Relación

- **Propósito**: La inserción de datos en estas tablas permite a una aplicación rastrear no solo qué frutas están disponibles sino también cómo cambian sus precios con el tiempo. Esto puede ser fundamental para aplicaciones en negocios de venta al detalle, análisis de mercado, o sistemas de inventario.
- **Relación**: La clave `fruta_id` en `frutas_precio` establece una relación de clave foránea con la clave primaria `id` en `frutas_info`. Esto asegura que cada precio registrado en `frutas_precio` esté asociado con una fruta específica en `frutas_info`, manteniendo la integridad referencial entre las tablas.

Este diseño de base de datos permite consultas complejas, como determinar el último precio de una fruta específica, visualizar la historia de precios de una fruta, o comparar precios entre diferentes frutas en fechas específicas, todo mientras se mantiene organizada y accesible la información básica de las frutas.

Consulta de datos

Se diseñaron las siguientes consultas para reportar la relación entre la información de las frutas y sus precios:

```sql
-- Consulta de la Información de las Frutas

select id, nombre, descripcion from frutas_info;

-- Consulta de los Precios de las Frutas

select id, fruta_id, precio, fecha from frutas_precio;

-- Consulta de los últimos precios conocidos de cada Fruta (Subconsul

select
    fi.id as frutaId,
    fi.nombre,
    (
        select fp.precio
        from frutas_precio fp
        where fp.fruta_id = fi.id
        order by fp.fecha desc
        fetch first 1 row only
    ) as precio
from frutas_info fi;

-- Consulta de los últimos precios conocidos de cada fruta hasta la fe

select
    fi.id as frutaId,
    fi.nombre,
    (
        select fp.precio
        from frutas_precio fp
        where fp.fruta_id = fi.id and fp.fecha < CURRENT_TIMESTAMP
        order by fp.fecha desc
        fetch first 1 row only
    ) as precio
from frutas_info fi;

-- Consulta de los últimos precios conocidos de cada fruta hasta la fe

with fp as (
    select
        ROW_NUMBER() over (partition by fruta_id order by fecha desc)
        fruta_id,
        precio,
        fecha
    from frutas_precio
    where fecha < CURRENT_TIMESTAMP
)
select
    fi.id as frutaId,
    fi.nombre,
```

```
    fp.precio,
    fp.fecha
from frutas_info fi
left join fp on fp.fruta_id = fi.id and rn = 1;

-- Consulta de los precios conocidos para las frutas (Consulta anidad

select fp.id, fi.nombre, fp.precio, fp.fecha from frutas_precio fp
    join frutas_info fi on fp.fruta_id = fi.id;

-- Consulta de los precios conocidos para las frutas (Consulta anidad
-- con Subconsulta para el último precio más reciente a la fecha actu

select
    fr.id,
    fr.nombre,
    fr.precio,
    fr.fecha
from (
    select
        fp.id,
        fi.nombre,
        fp.precio,
        fp.fecha,
        ROW_NUMBER() over (partition by fi.id order by fp.fecha desc)
    from frutas_precio fp
    left join frutas_info fi on fp.fruta_id = fi.id
    where fp.fecha < CURRENT_TIMESTAMP
) fr
where fr.rn = 1;
```

Estas consultas SQL se enfocan en extraer y manipular datos de dos tablas relacionadas: `frutas_info`, que almacena información general sobre frutas, y `frutas_precio`, que registra los precios de estas frutas a lo largo del tiempo. Vamos a desglosar cada consulta para entender su propósito y cómo funciona.

Consulta de la Información de las Frutas

```
select id, nombre, descripcion from frutas_info;
```

Esta consulta selecciona todas las filas de la tabla `frutas_info`, devolviendo los `id`, `nombre` y `descripcion` de todas las frutas. Es una consulta básica que proporciona un resumen de todas las frutas disponibles.

Consulta de los Precios de las Frutas

```
select id, fruta_id, precio, fecha from frutas_precio;
```

Similar a la consulta anterior, pero para la tabla `frutas_precio`, recuperando todos los registros de precios de las frutas, incluyendo el `id` del registro de precio, `fruta_id` (que enlaza con `frutas_info`), `precio`, y la `fecha` en que se registró dicho precio.

Consulta de los Últimos Precios Conocidos de Cada Fruta

```
select
    fi.id as frutaId,
    fi.nombre,
    (
        select fp.precio
        from frutas_precio fp
        where fp.fruta_id = fi.id
        order by fp.fecha desc
        fetch first 1 row only
    ) as precio
from frutas_info fi;
```

Esta consulta muestra el último precio conocido de cada fruta. Usa una subconsulta en la lista de selección para encontrar el último precio de cada fruta, ordenando los resultados por fecha de manera descendente y limitando a la primera fila (`fetch first 1 row only`), lo cual asegura que solo se devuelva el precio más reciente.

Consulta de los Últimos Precios Conocidos Hasta la Fecha Actual

```
select
    fi.id as frutaId,
    fi.nombre,
    (
        select fp.precio
        from frutas_precio fp
        where fp.fruta_id = fi.id and fp.fecha < CURRENT_TIMESTAMP
        order by fp.fecha desc
        fetch first 1 row only
    ) as precio
from frutas_info fi;
```

Es similar a la consulta anterior pero con una condición adicional que asegura que solo se consideren los precios registrados hasta el momento actual (`CURRENT_TIMESTAMP`), excluyendo así cualquier precio futuro que pudiera haberse registrado anticipadamente.

Consulta de los Últimos Precios Conocidos con `WITH`

```
with fp as (
    select
        ROW_NUMBER() over (partition by fruta_id order by fecha desc)
        fruta_id,
        precio,
        fecha
    from frutas_precio
    where fecha < CURRENT_TIMESTAMP
)
select
    fi.id as frutaId,
    fi.nombre,
    fp.precio,
    fp.fecha
from frutas_info fi
left join fp on fp.fruta_id = fi.id and rn = 1;
```

Esta consulta usa una expresión de tabla común (`WITH`) para numerar los precios de cada fruta por su fecha, en orden descendente, y luego selecciona solo el precio más reciente (`rn = 1`). Es una forma más avanzada y posiblemente más eficiente de lograr el mismo resultado que las subconsultas anidadas anteriores, especialmente para conjuntos de datos grandes.

Consulta de los Precios Conocidos para las Frutas

```
select fp.id, fi.nombre, fp.precio, fp.fecha from frutas_precio fp
    join frutas_info fi on fp.fruta_id = fi.id;
```

Esta consulta une `frutas_precio` con `frutas_info` para devolver una lista de todos los precios conocidos para las frutas, incluyendo el `nombre` de la fruta de la tabla `frutas_info`.

Consulta de Precios con el Último Precio Más Reciente

```sql
select
    fr.id,
    fr.nombre,
    fr.precio,
    fr.fecha
from (
    select
        fp.id,
        fi.nombre,
        fp.precio,
        fp.fecha,
        ROW_NUMBER() over (partition by fi.id order by fp.fecha desc)
    from frutas_precio fp
    left join frutas_info fi on fp.fruta_id = fi.id
    where fp.fecha < CURRENT_TIMESTAMP
) fr
where fr.rn = 1

;
```

Esta consulta es una variante de las consultas anteriores que también utiliza la función de ventana `ROW_NUMBER()` para asignar un número de fila a cada precio de fruta, ordenado por fecha de manera descendente. Luego, selecciona solo la fila con el número 1 (`rn = 1`) para cada fruta, es decir, su último precio registrado hasta la fecha actual.

Todas estas consultas demuestran distintas técnicas para manipular y recuperar datos relacionados de manera eficiente en un sistema de base de datos, utilizando tanto métodos simples como avanzados para satisfacer requerimientos específicos de información.