EBook Gratis

APRENDIZAJE SQL

Free unaffiliated eBook created from
Stack Overflow contributors.

#sql

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sql

It is an unofficial and free SQL ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official SQL.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con SQL

Observaciones

SQL es un lenguaje de consulta estructurado que se utiliza para administrar datos en un sistema de base de datos relacional. Diferentes proveedores han mejorado el idioma y tienen una variedad de sabores para el idioma.

NB: esta etiqueta se refiere explícitamente al **estándar ISO / ANSI SQL** ; No a ninguna implementación específica de esa norma.

Versiones

Versión	Nombre corto	Estándar	Fecha de lanzamiento
1986	SQL-86	ANSI X3.135-1986, ISO 9075: 1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO / IEC 9075: 1989	1989-01-01
1992	SQL-92	ISO / IEC 9075: 1992	1992-01-01
1999	SQL: 1999	ISO / IEC 9075: 1999	1999-12-16
2003	SQL: 2003	ISO / IEC 9075: 2003	2003-12-15
2006	SQL: 2006	ISO / IEC 9075: 2006	2006-06-01
2008	SQL: 2008	ISO / IEC 9075: 2008	2008-07-15
2011	SQL: 2011	ISO / IEC 9075: 2011	2011-12-15
2016	SQL: 2016	ISO / IEC 9075: 2016	2016-12-01

Examples

Visión general

El lenguaje de consulta estructurado (SQL) es un lenguaje de programación de propósito especial diseñado para administrar datos almacenados en un sistema de administración de bases de datos relacionales (RDBMS). Los lenguajes similares a SQL también se pueden usar en sistemas de administración de flujo de datos relacionales (RDSMS) o en bases de datos "no solo de SQL" (NoSQL).

SQL se compone de 3 sub-lenguajes principales:

1. Lenguaje de definición de datos (DDL): para crear y modificar la estructura de la base de datos;
2. Lenguaje de manipulación de datos (DML): para realizar operaciones de lectura, inserción, actualización y eliminación en los datos de la base de datos;
3. Data Control Language (DCL): para controlar el acceso a los datos almacenados en la base de datos.

Artículo de SQL en Wikipedia

Las operaciones principales de DML son Crear, Leer, Actualizar y Eliminar (CRUD para abreviar), que se realizan mediante las instrucciones `INSERT` , `SELECT` , `UPDATE` y `DELETE` .
También hay una instrucción `MERGE` (recientemente agregada) que puede realizar las 3 operaciones de escritura (INSERTAR, ACTUALIZAR, BORRAR).

Artículo de CRUD en Wikipedia

Muchas bases de datos SQL se implementan como sistemas cliente / servidor; el término "servidor SQL" describe tal base de datos.
Al mismo tiempo, Microsoft crea una base de datos llamada "SQL Server". Si bien esa base de datos habla un dialecto de SQL, la información específica de esa base de datos no se encuentra en el tema de esta etiqueta, sino que pertenece a la documentación de SQL Server .

Lea Empezando con SQL en línea: https://riptutorial.com/es/sql/topic/184/empezando-con-sql

Capítulo 2: Actas

Observaciones

Una transacción es una unidad lógica de trabajo que contiene uno o más pasos, cada uno de los cuales debe completarse con éxito para que la transacción se comprometa con la base de datos. Si hay errores, entonces todas las modificaciones de datos se borran y la base de datos vuelve a su estado inicial al inicio de la transacción.

Examples

Transacción simple

```
BEGIN TRANSACTION
    INSERT INTO DeletedEmployees(EmployeeID, DateDeleted, User)
        (SELECT 123, GetDate(), CURRENT_USER);
    DELETE FROM Employees WHERE EmployeeID = 123;
COMMIT TRANSACTION
```

Transacción de reversión

Cuando algo falla en su código de transacción y desea deshacerlo, puede revertir su transacción:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
   COMMIT TRANSACTION
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
END CATCH
```

Lea Actas en línea: https://riptutorial.com/es/sql/topic/2424/actas

Capítulo 3: ACTUALIZAR

Sintaxis

- *Tabla de* actualización
  SET *nombre_columna = valor, COLUMN_NAME2 = valor_2, ..., column_name_n = Valor_N*
  DONDE *condición* (condición de *operador lógico*)

Examples

Actualizando todas las filas

Este ejemplo utiliza la tabla de coches de las bases de datos de ejemplo.

```
UPDATE Cars
SET Status = 'READY'
```

Esta declaración establecerá la columna 'estado' de todas las filas de la tabla 'Automóviles' en "LISTO" porque no tiene una cláusula `WHERE` para filtrar el conjunto de filas.

Actualización de filas especificadas

Este ejemplo utiliza la tabla de coches de las bases de datos de ejemplo.

```
UPDATE
    Cars
SET
    Status = 'READY'
WHERE
    Id = 4
```

Esta declaración establecerá el estado de la fila de 'Cars' con id 4 en "READY".

`WHERE` cláusula `WHERE` contiene una expresión lógica que se evalúa para cada fila. Si una fila cumple con los criterios, su valor se actualiza. De lo contrario, una fila se mantiene sin cambios.

Modificar valores existentes

Este ejemplo utiliza la tabla de coches de las bases de datos de ejemplo.

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

Las operaciones de actualización pueden incluir valores actuales en la fila actualizada. En este ejemplo simple, el `TotalCost` se incrementa en 100 para dos filas:

- El TotalCost del Auto # 3 se incrementa de 100 a 200.
- El TotalCost del Auto # 4 se incrementó de 1254 a 1354

El nuevo valor de una columna se puede derivar de su valor anterior o de cualquier otro valor de columna en la misma tabla o en una tabla combinada.

ACTUALIZAR con datos de otra tabla

Los ejemplos a continuación completan un `PhoneNumber` de `PhoneNumber` para cualquier empleado que también sea `Customer` y que actualmente no tenga un número de teléfono configurado en la Tabla de `Employees` .

(Estos ejemplos utilizan las tablas Empleados y Clientes de las Bases de datos de ejemplo).

SQL estándar

Actualización utilizando una subconsulta correlacionada:

```
UPDATE
    Employees
SET PhoneNumber =
    (SELECT
         c.PhoneNumber
     FROM
         Customers c
     WHERE
         c.FName = Employees.FName
         AND c.LName = Employees.LName)
WHERE Employees.PhoneNumber IS NULL
```

SQL: 2003

Actualizar utilizando `MERGE` :

```
MERGE INTO
    Employees e
USING
    Customers c
ON
    e.FName = c.Fname
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
   UPDATE
      SET PhoneNumber = c.PhoneNumber
```

servidor SQL

Actualizar usando `INNER JOIN` :

```
UPDATE
    Employees
SET
    PhoneNumber = c.PhoneNumber
FROM
    Employees e
INNER JOIN Customers c
        ON e.FName = c.FName
        AND e.LName = c.LName
WHERE
    PhoneNumber IS NULL
```

Capturando registros actualizados

A veces uno quiere capturar los registros que se acaban de actualizar.

```
CREATE TABLE #TempUpdated(ID INT)

Update TableName SET Col1 = 42
    OUTPUT inserted.ID INTO #TempUpdated
    WHERE Id > 50
```

Lea ACTUALIZAR en línea: https://riptutorial.com/es/sql/topic/321/actualizar

Capítulo 4: AGRUPAR POR

Introducción

Los resultados de una consulta SELECT pueden agruparse por una o más columnas usando la instrucción `GROUP BY` : todos los resultados con el mismo valor en las columnas agrupadas se agregan juntos. Esto genera una tabla de resultados parciales, en lugar de un resultado. GROUP BY se puede usar junto con las funciones de agregación que usan la instrucción `HAVING` para definir cómo se agregan las columnas no agrupadas.

Sintaxis

- AGRUPAR POR {
  expresión de columna
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBO (<group_by_expression> [, ... n])
  | CONJUNTOS DE AGRUPACIÓN ([, ... n])
  | () - calcula el total general
  } [, ... n]

- <group_by_expression> :: =
  expresión de columna
  | (columna-expresión [, ... n])

- <grouping_set> :: =
  () - calcula el total general
  | <grouping_set_item>
  | (<grouping_set_item> [, ... n])

- <grouping_set_item> :: =
  <group_by_expression>
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBO (<group_by_expression> [, ... n])

Examples

USE GROUP BY para CONTAR el número de filas para cada entrada única en una columna dada

Supongamos que desea generar recuentos o subtotales para un valor determinado en una columna.

Dada esta tabla, "westerosianos":

Nombre	GreatHouseAllegience
Arya	Rígido
Cercei	Lannister
Myrcella	Lannister
Yara	Greyjoy
Catelyn	Rígido
Sansa	Rígido

Sin GROUP BY, COUNT simplemente devolverá un número total de filas:

```
SELECT Count(*) Number_of_Westerosians
FROM Westerosians
```

devoluciones...

Número_de_Westerosianos
6

Pero al agregar GRUPO POR, podemos CONTAR a los usuarios para cada valor en una columna dada, para devolver el número de personas en una Gran Casa dada, por ejemplo:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

devoluciones...

Casa	Número_de_Westerosianos
Rígido	3
Greyjoy	1
Lannister	2

Es común combinar GROUP BY con ORDER BY para ordenar los resultados por categoría más grande o más pequeña:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
ORDER BY Number_of_Westerosians Desc
```

devoluciones...

Casa	Número_de_Westerosianos
Rígido	3
Lannister	2
Greyjoy	1

Filtrar los resultados de GROUP BY utilizando una cláusula HAVING.

Una cláusula HAVING filtra los resultados de una expresión GROUP BY. Nota: Los siguientes ejemplos utilizan la base de datos de ejemplo de la Biblioteca .

Ejemplos:

Devuelve todos los autores que escribieron más de un libro (ejemplo en vivo).

```
SELECT
  a.Id,
  a.Name,
  COUNT(*) BooksWritten
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
GROUP BY
  a.Id,
  a.Name
HAVING COUNT(*) > 1    -- equals to HAVING BooksWritten > 1
;
```

Devuelve todos los libros que tengan más de tres autores (ejemplo vivo).

```
SELECT
  b.Id,
  b.Title,
  COUNT(*) NumberOfAuthors
FROM BooksAuthors ba
  INNER JOIN Books b ON b.id = ba.bookid
GROUP BY
  b.Id,
  b.Title
HAVING COUNT(*) > 3    -- equals to HAVING NumberOfAuthors > 3
;
```

Ejemplo básico de GROUP BY

Podría ser más fácil si piensa en GROUP BY como "para cada uno" por el bien de la explicación. La consulta a continuación:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
```

```
GROUP BY EmpID
```

esta diciendo:

"Dame la suma de MonthlySalary's **para cada** EmpID"

Así que si tu mesa se veía así:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Resultado:

```
+-+---+
|1|200|
+-+---+
|2|300|
+-+---+
```

La suma no parece hacer nada porque la suma de un número es ese número. Por otro lado si se veía así:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|1    |300          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Resultado:

```
+-+---+
|1|500|
+-+---+
|2|300|
+-+---+
```

Entonces sería porque hay dos EmpID 1 para sumar.

Agregación ROLAP (Data Mining)

Descripción

El estándar SQL proporciona dos operadores agregados adicionales. Estos utilizan el valor

polimórfico "ALL" para denotar el conjunto de todos los valores que puede tomar un atributo. Los dos operadores son:

- `with data cube` que proporciona todas las combinaciones posibles que los atributos de argumento de la cláusula.
- `with roll up` que proporciona los agregados obtenidos al considerar los atributos en orden de izquierda a derecha en comparación con la forma en que se enumeran en el argumento de la cláusula.

Versiones estándar de SQL que admiten estas características: 1999, 200, 200, 200, 202011.

Ejemplos

Considere esta tabla:

Comida	Marca	Cantidad total
Pastas	Marca1	100
Pastas	Marca2	250
Pizza	Marca2	300

Con cubo

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with cube
```

Comida	Marca	Cantidad total
Pastas	Marca1	100
Pastas	Marca2	250
Pastas	TODOS	350
Pizza	Marca2	300
Pizza	TODOS	300
TODOS	Marca1	100
TODOS	Marca2	550
TODOS	TODOS	650

Con enrollar

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with roll up
```

Comida	Marca	Cantidad total
Pastas	Marca1	100
Pastas	Marca2	250
Pizza	Marca2	300
Pastas	TODOS	350
Pizza	TODOS	300
TODOS	TODOS	650

Lea AGRUPAR POR en línea: https://riptutorial.com/es/sql/topic/627/agrupar-por

Capítulo 5: Álgebra relacional

Examples

Visión general

El álgebra relacional no es un lenguaje *SQL* completo, sino una forma de obtener una comprensión teórica del procesamiento relacional. Como tal, no debería hacer referencias a entidades físicas como tablas, registros y campos; debe hacer referencias a construcciones abstractas tales como relaciones, tuplas y atributos. Dicho esto, no usaré los términos académicos en este documento y me atendré a los términos legos más conocidos: tablas, registros y campos.

Un par de reglas de álgebra relacional antes de comenzar:

- Los operadores utilizados en el álgebra relacional trabajan en tablas enteras en lugar de registros individuales.
- El resultado de una expresión relacional siempre será una tabla (esto se denomina *propiedad de cierre*)

A lo largo de este documento me referiré a las siguientes dos tablas:

Departments

ID	Dept
1	Production
2	Quality Control

People

ID	PersonName	StartYear	ManagerID	DepartmentID
1	Darren	2005		1
2	David	2006	1	1
3	Burt	2006	1	1
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

SELECCIONAR

El operador de **selección** devuelve un subconjunto de la tabla principal.
seleccione <*tabla*> ***donde*** <*condición*>

Por ejemplo, examine la expresión:

seleccione People **donde** DepartmentID = 2

Esto se puede escribir como:

$$\sigma_{\text{DepartmentID} = 2}(\text{People})$$

Esto resultará en una tabla cuyos registros comprenden todos los registros en la tabla de

Personas donde el valor de ID de *Departamento* es igual a 2:

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

Las condiciones también se pueden unir para restringir aún más la expresión:

seleccione People **donde** StartYear> 2005 **y** DepartmentID = 2

resultará en la siguiente tabla:

ID	PersonName	StartYear	ManagerID	DepartmentID
5	Fred	2008	4	2

PROYECTO

El operador del **proyecto** devolverá valores de campo distintos de una tabla.
proyecto *<tabla>* ***sobre*** *<lista de campos>*

Por ejemplo, examine la siguiente expresión:
proyecto People **over** StartYear

Esto se puede escribir como:

$$\Pi_{\text{StartYear}}\ \textbf{(People)}$$

Esto dará como resultado una tabla que comprende los valores distintos mantenidos dentro del campo *Inicio* de año de la tabla *Personas* .

StartYear
2005
2006
2004
2008

Los valores duplicados se eliminan de la tabla resultante debido a la *propiedad de cierre que* crea una tabla relacional: todos los registros en una tabla relacional deben ser distintos.

Si la *lista de campos* comprende más de un solo campo, la tabla resultante es una versión distinta de estos campos.
proyecto People **over** StartYear, DepartmentID devolverá:

StartYear	DepartmentID
2005	1
2006	1
2004	2
2008	2
2005	2

Se eliminó un registro debido a la duplicación de 2006 *StartYear* y 1 *DepartmentID* .

DAR

Las expresiones relacionales se pueden encadenar al nombrar las expresiones individuales usando la palabra clave **dar** o mediante la incorporación de una expresión dentro de otra.

<expresión de álgebra relacional> ***dando*** *<nombre de alias>*

Por ejemplo, considera las siguientes expresiones:
seleccione Personas **donde** DepartmentID = 2 **dando** A
proyecto A **sobre** PersonName **dando** B

Esto dará como resultado la tabla B a continuación, siendo la tabla A el resultado de la primera expresión.

A

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

B

PersonName
Sarah
Fred
Joanne

La primera expresión se evalúa y la tabla resultante recibe el alias A. Esta tabla se usa dentro de la segunda expresión para obtener la tabla final con un alias de B.

Otra forma de escribir esta expresión es reemplazar el nombre de alias de la tabla en la segunda expresión con el texto completo de la primera expresión entre paréntesis:
proyecto (**seleccione** People **donde** DepartmentID = 2) **sobre** PersonName **dando** B

Esto se llama una *expresión anidada* .

Unirse natural

Una combinación natural une dos tablas usando un campo común compartido entre las tablas.

unirse a *<tabla 1>* ***y*** *<tabla 2>* ***donde*** *<campo 1> = <campo 2>*
suponiendo que <field 1> está en <table 1> y <field 2> está en <table 2>.

Por ejemplo, la siguiente expresión de unión se unirá a *personas* y *departamentos* según las columnas *ID de departamento* e *ID* en las tablas respectivas:
unirse a las personas **y** departamentos **donde** DepartmentID = ID

ID	PersonName	StartYear	ManagerID	DepartmentID	Dept
1	Darren	2005		1	Production
2	David	2006	1	1	Production
3	Burt	2006	1	1	Production
4	Sarah	2004		2	Quality Control
5	Fred	2008	4	2	Quality Control
6	Joanne	2005	4	2	Quality Control

Tenga en cuenta que solo se muestra *DepartmentID* de la tabla *Personas* y no *ID* de la tabla *Departamento* . Solo se debe mostrar uno de los campos que se comparan, que generalmente es el nombre del campo de la primera tabla en la operación de unión.

Aunque no se muestra en este ejemplo, es posible que unir tablas puede dar como resultado que dos campos tengan el mismo encabezado. Por ejemplo, si hubiera utilizado el encabezamiento *de nombre* para identificar los campos *a PERSONNAME* y *Dept* (es decir, para identificar el nombre de la persona y el nombre del departamento). Cuando surge esta situación, usamos el nombre de la tabla para calificar los nombres de los campos usando la notación de puntos: *People.Name* y *Departments.Name*

la combinación combinada con la **selección** y el **proyecto** se pueden usar juntas para obtener información:

únete a las *personas* ***y*** *departamentos* ***donde*** *DepartmentID = ID* ***da*** *A*
seleccione *A* ***donde*** *StartYear = 2005* ***y*** *Dept = 'Producción'* ***dando*** *B*
proyecto *B* ***sobre*** *PersonName* ***dando*** *C*

o como una expresión combinada:

proyecto *(* ***seleccione*** *(* ***unirse a*** *Personas* ***y*** *Departamentos* ***donde*** *DepartmentID = ID)* ***donde*** *StartYear = 2005* ***y*** *Dept = 'Producción')* ***sobre*** *PersonName* ***dando*** *C*

Esto dará lugar a esta tabla:

PersonName
Darren

ALIAS

DIVIDIR

UNIÓN

INTERSECCIÓN

DIFERENCIA

ACTUALIZACIÓN (: =)

VECES

Lea Álgebra relacional en línea: https://riptutorial.com/es/sql/topic/7311/algebra-relacional

Capítulo 6: ALTERAR MESA

Introducción

El comando ALTER en SQL se usa para modificar la columna / restricción en una tabla

Sintaxis

- ALTER TABLE [table_name] ADD [column_name] [datatype]

Examples

Añadir columna (s)

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

La declaración anterior agregaría columnas denominadas `StartingDate` que no pueden ser NULL con el valor predeterminado como fecha actual y `DateOfBirth` que puede ser NULL en la tabla de empleados .

Colocar columna

```
ALTER TABLE Employees
DROP COLUMN salary;
```

Esto no solo eliminará la información de esa columna, sino que eliminará el salario de la columna de los empleados de la tabla (la columna ya no existirá).

Restricción de caída

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

Esto elimina una restricción llamada DefaultSalary de la definición de la tabla de empleados.

Nota: - Asegúrese de que las restricciones de la columna se eliminan antes de eliminar una columna.

Añadir restricción

```
ALTER TABLE Employees
ADD CONSTRAINT DefaultSalary DEFAULT ((100)) FOR [Salary]
```

Esto agrega una restricción llamada DefaultSalary que especifica un valor predeterminado de 100 para la columna Salario.

Se puede agregar una restricción a nivel de tabla.

Tipos de restricciones

- Clave principal: evita un registro duplicado en la tabla
- Clave externa: apunta a una clave principal de otra tabla
- No nulo: evita que se ingresen valores nulos en una columna
- Único: identifica de forma única cada registro en la tabla
- Predeterminado: especifica un valor predeterminado
- Verificar: limita los rangos de valores que se pueden colocar en una columna

Para obtener más información sobre las restricciones, consulte la documentación de Oracle .

Alterar columna

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

Esta consulta modificará el tipo de datos de la columna de `StartingDate` y lo cambiará de `date` simple a `datetime` y establecerá el valor predeterminado a la fecha actual.

Añadir clave principal

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

Esto agregará una clave principal a la tabla Empleados en el `ID` campo. Incluir más de un nombre de columna en los paréntesis junto con la ID creará una clave primaria compuesta. Al agregar más de una columna, los nombres de las columnas deben estar separados por comas.

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

Lea ALTERAR MESA en línea: https://riptutorial.com/es/sql/topic/356/alterar-mesa

Capítulo 7: aplicación cruzada, aplicación externa

Examples

Conceptos básicos sobre la aplicación y la aplicación externa

Aplicar se utilizará cuando la función con valor de tabla en la expresión correcta.

crear una tabla de departamento para contener información sobre los departamentos. Luego cree una tabla de empleados que contenga información sobre los empleados. Tenga en cuenta que cada empleado pertenece a un departamento, por lo que la tabla de Empleados tiene integridad referencial con la tabla de Departamentos.

La primera consulta selecciona los datos de la tabla del Departamento y usa CROSS APPLY para evaluar la tabla de Empleados para cada registro de la tabla del Departamento. La segunda consulta simplemente une la tabla de Departamento con la tabla de Empleado y se producen todos los registros coincidentes.

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
```

Si nos fijamos en los resultados que producen, es exactamente el mismo conjunto de resultados; ¿En qué se diferencia de un JOIN y cómo ayuda a escribir consultas más eficientes?

La primera consulta en el Script # 2 selecciona los datos de la tabla del Departamento y usa APLICACIÓN EXTERNA para evaluar la tabla de Empleados para cada registro de la tabla del Departamento. Para aquellas filas para las que no hay una coincidencia en la tabla Empleado, esas filas contienen valores NULOS como puede ver en el caso de las filas 5 y 6. La segunda consulta simplemente utiliza una JUNTA EXTERNA IZQUIERDA entre la tabla Departamento y la tabla Empleado. Como era de esperar, la consulta devuelve todas las filas de la tabla de Departamento; incluso para aquellas filas para las que no hay coincidencia en la tabla Empleado.

```
SELECT *
FROM Department D
OUTER APPLY (
    SELECT *
    FROM Employee E
```

```
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
GO
```

Aunque las dos consultas anteriores devuelven la misma información, el plan de ejecución será un poco diferente. Pero en cuanto al costo no habrá mucha diferencia.

Ahora llega el momento de ver dónde se requiere realmente el operador APLICAR. En el script # 3, estoy creando una función con valores de tabla que acepta el ID de departamento como parámetro y devuelve a todos los empleados que pertenecen a este departamento. La siguiente consulta selecciona datos de la tabla del Departamento y utiliza CROSS APPLY para unirse a la función que creamos. Pasa el ID de departamento para cada fila de la expresión de la tabla externa (en nuestra tabla de Departamento de casos) y evalúa la función para cada fila similar a una subconsulta correlacionada. La siguiente consulta utiliza la APLICACIÓN EXTERNA en lugar de la APLICACIÓN CRUZADA y, por lo tanto, a diferencia de la APLICACIÓN CRUZADA que solo devolvió datos correlacionados, la APLICACIÓN EXTERNA también devuelve datos no correlacionados, colocando NULL en las columnas que faltan.

```
CREATE FUNCTION dbo.fn_GetAllEmployeeOfADepartment (@DeptID AS int)
RETURNS TABLE
AS
  RETURN
  (
  SELECT
    *
  FROM Employee E
  WHERE E.DepartmentID = @DeptID
  )
GO
SELECT
  *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
  *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

Así que ahora, si se está preguntando, ¿podemos utilizar una combinación simple en lugar de las consultas anteriores? Entonces la respuesta es NO, si reemplaza CROSS / OUTER APPLY en las consultas anteriores con INNER JOIN / LEFT OUTER JOIN, especifique la cláusula ON (algo como 1 = 1) y ejecute la consulta, obtendrá "El identificador de varias partes" D.DepartmentID "no se pudo enlazar". error. Esto se debe a que, con JOINs, el contexto de ejecución de la consulta externa es diferente del contexto de ejecución de la función (o una tabla derivada), y no puede vincular un valor / variable de la consulta externa a la función como un parámetro. Por lo tanto, se requiere el operador APLICAR para tales consultas.

Lea aplicación cruzada, aplicación externa en línea:
https://riptutorial.com/es/sql/topic/2516/aplicacion-cruzada--aplicacion-externa

Capítulo 8: Bloques de ejecución

Examples

Usando BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
END
```

Lea Bloques de ejecución en línea: https://riptutorial.com/es/sql/topic/1632/bloques-de-ejecucion

Capítulo 9: BORRAR

Introducción

La instrucción DELETE se utiliza para eliminar registros de una tabla.

Sintaxis

1. ELIMINAR DE *TableName* [WHERE *Condition*] [LIMIT *count*]

Examples

ELIMINAR ciertas filas con DONDE

Esto eliminará todas las filas que coincidan con los criterios de `WHERE` .

```
DELETE FROM Employees
WHERE FName = 'John'
```

ELIMINAR todas las filas

La omisión de una cláusula `WHERE` eliminará todas las filas de una tabla.

```
DELETE FROM Employees
```

Consulte la documentación de TRUNCATE para obtener detalles sobre cómo el rendimiento de TRUNCATE puede ser mejor, ya que ignora los desencadenantes e índices y registros para simplemente eliminar los datos.

Cláusula TRUNCATE

Use esto para restablecer la tabla a la condición en la que se creó. Esto elimina todas las filas y restablece valores como el incremento automático. Tampoco registra cada eliminación de fila individual.

```
TRUNCATE TABLE Employees
```

BORRAR ciertas filas basadas en comparaciones con otras tablas

Es posible `DELETE` datos de una tabla si coincide (o no coincide) con ciertos datos en otras tablas.

Asumamos que queremos `DELETE` datos de la Fuente una vez que se carguen en Target.

```
DELETE FROM Source
```

```
WHERE  EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Las implementaciones de RDBMS más comunes (p. Ej., MySQL, Oracle, PostgresSQL, Teradata) permiten unir tablas durante `DELETE` permite una comparación más compleja en una sintaxis compacta.

Agregando complejidad al escenario original, asumamos que Aggregate se crea a partir de Target una vez al día y no contiene la misma ID, pero contiene la misma fecha. Supongamos también que queremos eliminar los datos de la Fuente *solo* después de que se haya completado el agregado del día.

En MySQL, Oracle y Teradata esto se puede hacer usando:

```
DELETE FROM Source
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

En el uso de PostgreSQL:

```
DELETE FROM Source
USING  TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Básicamente, esto da como resultado INNER JOINs entre Source, Target y Aggregate. La eliminación se realiza en el origen cuando existen los mismos ID en el objetivo Y la fecha presente en el destino para esos ID también existe en el agregado.

La misma consulta también se puede escribir (en MySQL, Oracle, Teradata) como:

```
DELETE Source
FROM   Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Las combinaciones explícitas pueden mencionarse en las declaraciones de `Delete` en algunas implementaciones de RDBMS (por ejemplo, Oracle, MySQL) pero no se admiten en todas las plataformas (por ejemplo, Teradata no las admite)

Las comparaciones pueden diseñarse para verificar los escenarios de desajuste en lugar de combinar los con todos los estilos de sintaxis (observe `NOT EXISTS` continuación)

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Lea BORRAR en línea: https://riptutorial.com/es/sql/topic/1105/borrar

Capítulo 10: BORRAR o BORRAR la base de datos

Sintaxis

- Sintaxis de MSSQL:
- DROP DATABASE [IF EXISTS] {database_name | database_snapshot_name} [, ... n] [;]
- Sintaxis de MySQL:
- DROP {BASE DE DATOS | SCHEMA} [IF EXISTS] db_name

Observaciones

`DROP DATABASE` se utiliza para eliminar una base de datos de SQL. Asegúrese de crear una copia de seguridad de su base de datos antes de eliminarla para evitar la pérdida accidental de información.

Examples

Base de datos DROP

Eliminar la base de datos es una simple declaración de una sola línea. Eliminar base de datos eliminará la base de datos, por lo tanto, siempre asegúrese de tener una copia de seguridad de la base de datos si es necesario.

A continuación se muestra el comando para eliminar la base de datos de empleados

```
DROP DATABASE [dbo].[Employees]
```

Lea BORRAR o BORRAR la base de datos en línea:
https://riptutorial.com/es/sql/topic/3974/borrar-o-borrar-la-base-de-datos

Capítulo 11: CASO

Introducción

La expresión CASE se utiliza para implementar la lógica if-then.

Sintaxis

- CASE input_expression
  CUÁNDO comparar1 ENTONCES resultado1
  [CUANDO compare2 ENTONCES result2] ...
  [ELSE resultx]
  FIN
- CASO
  CUANDO condiciona1 ENTONCES resultado1
  [CUANDO condicion2 LUEGO result2] ...
  [ELSE resultx]
  FIN

Observaciones

La *expresión CASE simple* devuelve el primer resultado cuyo valor `compareX` es igual a `input_expression` .

La *expresión CASE buscada* devuelve el primer resultado cuya `conditionX` X es verdadera.

Examples

CASO buscado en SELECCIONAR (coincide con una expresión booleana)

El CASE *buscado* devuelve resultados cuando una expresión *booleana* es VERDADERA.

(Esto difiere del caso simple, que solo puede verificar la equivalencia con una entrada).

```
SELECT Id, ItemId, Price,
  CASE WHEN Price < 10 THEN 'CHEAP'
       WHEN Price < 20 THEN 'AFFORDABLE'
       ELSE 'EXPENSIVE'
  END AS PriceRating
FROM ItemSales
```

Carné de identidad	Identificación del artículo	Precio	Precio
1	100	34.5	COSTOSO
2	145	2.3	BARATO

Carné de identidad	Identificación del artículo	Precio	Precio
3	100	34.5	COSTOSO
4	100	34.5	COSTOSO
5	145	10	ASEQUIBLE

Use CASO para CONTAR el número de filas en una columna que coincida con una condición.

Caso de uso

`CASE` se puede usar junto con `SUM` para devolver un recuento de solo aquellos elementos que coinciden con una condición predefinida. (Esto es similar a `COUNTIF` en Excel.)

El truco es devolver resultados binarios que indiquen coincidencias, por lo que los "1" devueltos para las entradas coincidentes se pueden sumar para un recuento del número total de coincidencias.

Dada esta tabla de `ItemSales` , supongamos que desea conocer el número total de artículos que se han categorizado como "Caros":

Carné de identidad	Identificación del artículo	Precio	Precio
1	100	34.5	COSTOSO
2	145	2.3	BARATO
3	100	34.5	COSTOSO
4	100	34.5	COSTOSO
5	145	10	ASEQUIBLE

Consulta

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM ( CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
          END
        ) AS ExpensiveItemsCount
FROM ItemSales
```

Resultados:

ArtículosCuenta	ExpensiveItemsCount
5	3

Alternativa:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN 1
            ELSE 0
        END
       ) AS ExpensiveItemsCount
FROM ItemSales
```

CASTILLO ABREVIADO en SELECCIONAR

La variante abreviada de `CASE` evalúa una expresión (generalmente una columna) contra una serie de valores. Esta variante es un poco más corta y guarda la repetición de la expresión evaluada una y otra vez. La cláusula `ELSE` todavía se puede utilizar, sin embargo:

```
SELECT Id, ItemId, Price,
  CASE Price WHEN 5  THEN 'CHEAP'
             WHEN 15 THEN 'AFFORDABLE'
             ELSE         'EXPENSIVE'
  END as PriceRating
FROM ItemSales
```

Una palabra de precaución. Es importante darse cuenta de que cuando se usa la variante corta, la declaración completa se evalúa en cada `WHEN` . Por lo tanto la siguiente declaración:

```
SELECT
    CASE ABS(CHECKSUM(NEWID())) % 4
        WHEN 0 THEN 'Dr'
        WHEN 1 THEN 'Master'
        WHEN 2 THEN 'Mr'
        WHEN 3 THEN 'Mrs'
    END
```

puede producir un resultado `NULL` . Esto se debe a que en cada `WHEN` `NEWID()` se llama de nuevo con un nuevo resultado. Equivalente a:

```
SELECT
    CASE
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
    END
```

Por lo tanto, puede pasar por alto todos los casos de `WHEN` y el resultado es `NULL`

CASO en una cláusula ORDENAR POR

Podemos usar 1,2,3 .. para determinar el tipo de orden:

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
      WHEN 'MARKETING' THEN  1
      WHEN 'SALES' THEN 2
      WHEN 'RESEARCH' THEN 3
      WHEN 'INNOVATION' THEN 4
      ELSE        5
      END,
      CITY
```

CARNÉ DE IDENTIDAD	REGIÓN	CIUDAD	DEPARTAMENTO	EMPLEADOS_NUMBER
12	Nueva Inglaterra	Bostón	MÁRKETING	9
15	Oeste	San Francisco	MÁRKETING	12
9	Medio oeste	Chicago	VENTAS	8
14	Atlántico medio	Nueva York	VENTAS	12
5	Oeste	los Angeles	INVESTIGACIÓN	11
10	Atlántico medio	Filadelfia	INVESTIGACIÓN	13
4	Medio oeste	Chicago	INNOVACIÓN	11
2	Medio oeste	Detroit	RECURSOS HUMANOS	9

Usando CASE en ACTUALIZAR

muestra sobre aumentos de precios:

```
UPDATE ItemPrice
SET Price = Price *
  CASE ItemId
    WHEN 1 THEN 1.05
```

```
    WHEN 2 THEN 1.10
    WHEN 3 THEN 1.15
    ELSE 1.00
  END
```

Uso de CASE para valores NULOS ordenados en último lugar

de esta manera, el '0' que representa los valores conocidos se clasifica primero, el '1' que representa los valores NULL se ordenan por el último:

```
SELECT ID
      ,REGION
      ,CITY
      ,DEPARTMENT
      ,EMPLOYEES_NUMBER
  FROM DEPT
  ORDER BY
  CASE WHEN REGION IS NULL THEN 1
  ELSE 0
  END,
  REGION
```

CARNÉ DE IDENTIDAD	REGIÓN	CIUDAD	DEPARTAMENTO	EMPLEADOS_NUMBER
10	Atlántico medio	Filadelfia	INVESTIGACIÓN	13
14	Atlántico medio	Nueva York	VENTAS	12
9	Medio oeste	Chicago	VENTAS	8
12	Nueva Inglaterra	Bostón	MÁRKETING	9
5	Oeste	los Angeles	INVESTIGACIÓN	11
15	NULO	San Francisco	MÁRKETING	12
4	NULO	Chicago	INNOVACIÓN	11
2	NULO	Detroit	RECURSOS HUMANOS	9

CASO en la cláusula ORDER BY para ordenar los registros por el valor más bajo de 2 columnas

Imagine que necesita ordenar los registros por el valor más bajo de una de las dos columnas. Algunas bases de datos podrían usar una función `MIN()` o `LEAST()` no agregada para esto (`... ORDER BY MIN(Date1, Date2)`), pero en SQL estándar, tiene que usar una expresión `CASE` .

La expresión `CASE` en la consulta siguiente examina las columnas `Date1` y `Date2` , verifica qué columna tiene el valor más bajo y ordena los registros según este valor.

Data de muestra

Carné de identidad	Fecha 1	Fecha 2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

Consulta

```
SELECT Id, Date1, Date2
FROM YourTable
ORDER BY CASE
           WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
           ELSE Date2
         END
```

Resultados

Carné de identidad	Fecha 1	Fecha 2
1	**2017-01-01**	2017-01-31
3	2017-01-31	**2017-01-02**
2	2017-01-31	**2017-01-03**
6	**2017-01-04**	2017-01-31
5	2017-01-31	**2017-01-05**

Carné de identidad	Fecha 1	Fecha 2
4	**2017-01-06**	2017-01-31

Explicación

Como ve, la fila con `Id = 1` es la primera, porque debido a que `Date1` tiene el registro más bajo de toda la tabla `2017-01-01` , la fila donde `Id = 3` es la segunda, porque `Date2` es igual a `2017-01-02` que es el segundo valor más bajo de la tabla y así.

Por lo tanto, hemos ordenado los registros de `2017-01-01` a `2017-01-06` ascendentes y no nos importa en qué columna `Date1` o `Date2` son esos valores.

Lea CASO en línea: https://riptutorial.com/es/sql/topic/456/caso

Capítulo 12: Cláusula IN

Examples

Cláusula de entrada simple

Para obtener registros que tengan **alguno** de los `id` dados

```
select *
from products
where id in (1,8,3)
```

La consulta anterior es igual a

```
select *
from products
where id = 1
   or id = 8
   or id = 3
```

Usando la cláusula IN con una subconsulta

```
SELECT *
FROM customers
WHERE id IN (
    SELECT DISTINCT customer_id
    FROM orders
);
```

Lo anterior le dará a usted todos los clientes que tienen pedidos en el sistema.

Lea Cláusula IN en línea: https://riptutorial.com/es/sql/topic/3169/clausula-in

Capítulo 13: Comentarios

Examples

Comentarios de una sola línea

Los comentarios de una sola línea están precedidos por `--` , y van hasta el final de la línea:

```
SELECT *
FROM Employees -- this is a comment
WHERE FName = 'John'
```

Comentarios multilínea

Los comentarios de código multilínea están envueltos en `/* ... */` :

```
/* This query
   returns all employees */
SELECT *
FROM Employees
```

También es posible insertar dicho comentario en la mitad de una línea:

```
SELECT /* all columns: */ *
FROM Employees
```

Lea Comentarios en línea: https://riptutorial.com/es/sql/topic/1597/comentarios

Capítulo 14: Como operador

Sintaxis

- **Comodín con%:** SELECCIONAR * DE [tabla] DONDE [nombre_columna] Me gusta '% Valor%'

  Comodín con _: SELECCIONAR * DE [tabla] DONDE [nombre_columna] como 'V_n%'

  Tarjeta comodín con [charlist]: SELECT * FROM [table] DONDE [column_name] Me gusta 'V [abc] n%'

Observaciones

La condición LIKE en la cláusula WHERE se usa para buscar valores de columna que coincidan con el patrón dado. Los patrones se forman usando los siguientes dos caracteres comodín

- % (Símbolo de porcentaje): se utiliza para representar cero o más caracteres
- _ (Subrayado): se utiliza para representar un solo carácter

Examples

Coincidir patrón abierto

El ٪ comodín añadido al principio o al final (o ambos) de una cadena permitirá que coincida con 0 o más de cualquier carácter antes del comienzo o después del final del patrón.

Usar '%' en el medio permitirá que coincidan 0 o más caracteres entre las dos partes del patrón.

Vamos a utilizar esta tabla de empleados:

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratación
1	Juan	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	Herrero	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

La siguiente declaración coincide con todos los registros que tienen FName que **contiene la** cadena 'on' de la Tabla de empleados.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratación
3	R **en** ny	Herrero	2462544026	2	1	600	06-08-2015
4	J **en**	Sanchez	2454124602	1	1	400	23-03-2005

La siguiente declaración coincide con todos los registros que tienen Número de teléfono que **comienza con la** cadena '246' de los Empleados.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratación
1	Juan	Johnson	**246** 8101214	1	1	400	23-03-2005
3	Ronny	Herrero	**246** 2544026	2	1	600	06-08-2015
5	Hilde	Knag	**246** 8021911	2	1	800	01-01-2000

La siguiente declaración coincide con todos los registros **cuyo** Número de teléfono **termina con la** cadena '11' de los Empleados.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratación
2	Sophie	Amudsen	24791002 **11**	1	1	400	11-01-2010
5	Hilde	Knag	24680219 **11**	2	1	800	01-01-2000

Todos los registros donde Fname **3er carácter** es 'n' de Empleados.

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

(dos guiones bajos se usan antes de 'n' para omitir los primeros 2 caracteres)

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratación
3	Ronny	Herrero	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Partido de un solo personaje

Para ampliar las selecciones de una declaración de lenguaje de consulta estructurada (SQL-SELECT), se pueden usar caracteres comodín, el signo de porcentaje (%) y el subrayado (_).

El carácter _ (guión bajo) se puede usar como comodín para cualquier carácter individual en una coincidencia de patrón.

Encuentre todos los empleados cuyo Fname comience con 'j' y termine con 'n' y tenga exactamente 3 caracteres en Fname.

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

_ (subrayado) también se puede usar más de una vez como comodín para hacer coincidir patrones.

Por ejemplo, este patrón coincidiría con "jon", "jan", "jen", etc.

Estos nombres no se mostrarán "jn", "john", "jordan", "justin", "jason", "julian", "jillian", "joann" porque en nuestra consulta se usa un guión bajo y se puede omitir exactamente Un carácter, por lo que el resultado debe ser de 3 caracteres Fname.

Por ejemplo, este patrón coincidiría con "LaSt", "LoSt", "HaLt", etc.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Coincidir por rango o conjunto

Haga coincidir cualquier carácter individual dentro del rango especificado (por ejemplo: `[af]`) o establezca (por ejemplo: `[abcdef]`).

Este patrón de rango coincidiría con "gary" pero no con "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

Este patrón establecido coincidiría con "mary" pero no con "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[lmnop]ary'
```

El rango o conjunto también se puede negar agregando ^ caret antes del rango o conjunto:

Este patrón de rango *no* coincidirá con "gary" pero sí coincidirá con "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

Este patrón establecido *no* coincidirá con "mary", pero coincidirá con "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[^lmnop]ary'
```

Empareja CUALQUIER versus TODOS

Emparejar cualquiera
Debe coincidir al menos una cadena. En este ejemplo, el tipo de producto debe ser 'electrónica', 'libros' o 'video'.

```
SELECT *
FROM   purchase_table
WHERE  product_type LIKE ANY ('electronics', 'books', 'video');
```

Coincidir todos (debe cumplir todos los requisitos).
En este ejemplo, tanto 'reino unido' *como* 'londres' *y* 'carretera del este' (incluidas las variaciones) deben coincidir.

```
SELECT *
FROM   customer_table
WHERE  full_address LIKE ALL ('%united kingdom%', '%london%', '%eastern road%');
```

Selección negativa:
Use ALL para excluir todos los artículos.
Este ejemplo produce todos los resultados donde el tipo de producto no es 'electrónica' y no 'libros' y no 'video'.

```
SELECT *
FROM   customer_table
WHERE  product_type NOT LIKE ALL ('electronics', 'books', 'video');
```

Buscar un rango de personajes

La siguiente declaración coincide con todos los registros que tienen un FName que comienza con una letra de la A a la F en la Tabla de empleados .

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

Sentencia ESCAPE en la consulta LIKE

Si implementas una búsqueda de texto como consulta `LIKE` , normalmente lo haces así:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

Sin embargo, (aparte del hecho de que no debe usar necesariamente `LIKE` cuando puede usar la búsqueda de texto completo), esto crea un problema cuando alguien ingresa texto como "50%" o "a_b".

Entonces (en lugar de cambiar a búsqueda de texto completo), puede resolver ese problema usando la sentencia `LIKE` -escape:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

Eso significa `\` ahora será tratado como carácter de escape. Esto significa que ahora puede simplemente añadir `\` a cada carácter en la cadena que busca, y los resultados comenzarán a ser correctos, incluso cuando el usuario ingrese un carácter especial como `%` o `_` .

p.ej

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
     newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

Nota: El algoritmo anterior es solo para fines de demostración. No funcionará en los casos en que 1 grafema consta de varios caracteres (utf-8). por ejemplo, `string stringToSearch = "Les Mise\u0301rables";` Tendrás que hacer esto para cada grafema, no para cada personaje. No debe usar el algoritmo anterior si está tratando con idiomas asiáticos / asiáticos / asiáticos del sur. O más bien, si quieres comenzar con el código correcto, debes hacer eso para cada graphemeCluster.

Vea también ReverseString, una entrevista de C #

Caracteres comodín

Los caracteres comodín se utilizan con el operador SQL LIKE. Los comodines SQL se utilizan para buscar datos dentro de una tabla.

Los comodines en SQL son:%, _, [charlist], [^ charlist]

% - Un sustituto para cero o más caracteres

```
   Eg:  //selects all customers with a City starting with "Lo"
        SELECT * FROM Customers
        WHERE City LIKE 'Lo%';
```

```
       //selects all customers with a City containing the pattern "es"
      SELECT * FROM Customers
       WHERE City LIKE '%es%';
```

_ - Un sustituto para un solo personaje.

```
Eg://selects all customers with a City starting with any character, followed by "erlin"
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

[charlist] - Conjuntos y rangos de caracteres para emparejar

```
Eg://selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[adl]%';

//selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^ charlist] : coincide solo con un carácter NO especificado entre paréntesis

```
Eg://selects all customers with a City starting with a character that is not "a", "p", or "l"
SELECT * FROM Customers
WHERE City LIKE '[^apl]%';

or

SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```

Lea Como operador en línea: https://riptutorial.com/es/sql/topic/860/como-operador

Capítulo 15: Crear base de datos

Sintaxis

- CREAR BASE DE DATOS dbname;

Examples

Crear base de datos

Se crea una base de datos con el siguiente comando SQL:

```
CREATE DATABASE myDatabase;
```

Esto crearía una base de datos vacía llamada myDatabase donde puede crear tablas.

Lea Crear base de datos en línea: https://riptutorial.com/es/sql/topic/2744/crear-base-de-datos

Capítulo 16: CREAR FUNCION

Sintaxis

- CREATE FUNCTION function_name ([list_of_paramenters]) RETURNS return_data_type AS BEGIN function_body RETURN scalar_expression END

Parámetros

Argumento	Descripción
nombre de la función	el nombre de la función
list_of_paramenters	parámetros que acepta la función
return_data_type	escriba que la función vuelve. Algunos tipos de datos SQL
function_body	el código de función
expresión escalar	valor escalar devuelto por la función

Observaciones

CREAR FUNCIÓN crea una función definida por el usuario que se puede usar al realizar una consulta SELECCIONAR, INSERTAR, ACTUALIZAR o BORRAR. Las funciones se pueden crear para devolver una sola variable o una sola tabla.

Examples

Crear una nueva función

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
END
```

Este ejemplo crea una función llamada **FirstWord** , que acepta un parámetro varchar y devuelve otro valor varchar.

Lea CREAR FUNCION en línea: https://riptutorial.com/es/sql/topic/2437/crear-funcion

Capítulo 17: CREAR MESA

Introducción

Se utiliza la sentencia CREATE TABLE para crear una nueva tabla en la base de datos. Una definición de tabla consiste en una lista de columnas, sus tipos y cualquier restricción de integridad.

Sintaxis

- CREATE TABLE tableName ([ColumnName1] [datatype1] [, [ColumnName2] [datatype2] ...])

Parámetros

Parámetro	Detalles
nombre de la tabla	El nombre de las mesa
columnas	Contiene una 'enumeración' de todas las columnas que tiene la tabla. Ver **Crear una nueva tabla** para más detalles.

Observaciones

Los nombres de las tablas deben ser únicos.

Examples

Crear una nueva tabla

Se puede crear una tabla básica de `Employees` , que contiene una identificación, y el nombre y apellido del empleado junto con su número de teléfono usando

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

Este ejemplo es específico de Transact-SQL.

`CREATE TABLE` crea una nueva tabla en la base de datos, seguida del nombre de la tabla, `Employees`

Esto es seguido por la lista de nombres de columna y sus propiedades, como la ID

```
Id int identity(1,1) not null
```

Valor	Sentido
`Id`	El nombre de la columna.
`int`	es el tipo de datos
`identity(1,1)`	indica que la columna tendrá valores generados automáticamente que comienzan en 1 y se incrementan en 1 para cada nueva fila.
`primary key`	establece que todos los valores en esta columna tendrán valores únicos
`not null`	Indica que esta columna no puede tener valores nulos.

Crear tabla desde seleccionar

Es posible que desee crear un duplicado de una tabla:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

Puede usar cualquiera de las otras características de una instrucción SELECT para modificar los datos antes de pasarlos a la nueva tabla. Las columnas de la nueva tabla se crean automáticamente de acuerdo con las filas seleccionadas.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName," ",LName) AS FullName FROM Employees
WHERE Id > 10;
```

Duplicar una tabla

Para duplicar una tabla, simplemente haga lo siguiente:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

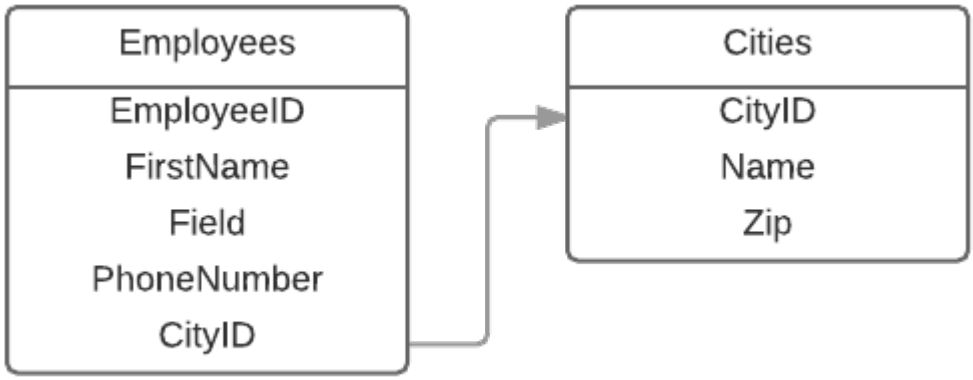
CREAR MESA CON CLAVE EXTRANJERA

A continuación puede encontrar la tabla `Employees` con una referencia a la tabla `Cities` .

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);

CREATE TABLE Employees(
```

```
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

Aquí podrías encontrar un diagrama de base de datos.



La columna `CityID` de la tabla `Employees` hará referencia a la columna `CityID` de la tabla `Cities` . A continuación puede encontrar la sintaxis para hacer esto.

```
CityID INT FOREIGN KEY REFERENCES Cities(CityID)
```

Valor	Sentido
`CityID`	Nombre de la columna
`int`	tipo de columna
`FOREIGN KEY`	Hace la clave externa *(opcional)*
`REFERENCES Cities(CityID)`	Hace la referencia a la tabla `Cities` columna `CityID`

Importante: No pudo hacer una referencia a una tabla que no existe en la base de datos. Sea fuente para hacer primero la tabla `Cities` y luego la mesa `Employees` . Si lo haces vise versa, te lanzará un error.

Crear una tabla temporal o en memoria

PostgreSQL y SQLite

Para crear una tabla temporal local para la sesión:

```
CREATE TEMP TABLE MyTable(...);
```

servidor SQL

Para crear una tabla temporal local para la sesión:

```
CREATE TABLE #TempPhysical(...);
```

Para crear una tabla temporal visible para todos:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

Para crear una tabla en memoria:

```
DECLARE @TempMemory TABLE(...);
```

Lea CREAR MESA en línea: https://riptutorial.com/es/sql/topic/348/crear-mesa

Capítulo 18: CURSOR SQL

Examples

Ejemplo de un cursor que consulta todas las filas por índice para cada base de datos

Aquí, un cursor se utiliza para recorrer todas las bases de datos.
Además, se utiliza un cursor de sql dinámico para consultar cada base de datos devuelta por el primer cursor.
Esto es para demostrar el alcance de conexión de un cursor.

```
DECLARE @db_name nvarchar(255)
DECLARE @sql nvarchar(MAX)

DECLARE @schema nvarchar(255)
DECLARE @table nvarchar(255)
DECLARE @column nvarchar(255)




DECLARE db_cursor CURSOR FOR
SELECT name FROM sys.databases


OPEN db_cursor
FETCH NEXT FROM db_cursor INTO @db_name

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @sql = 'SELECT * FROM ' + QUOTENAME(@db_name) + '.information_schema.columns'
    PRINT ''
    PRINT ''
    PRINT ''
    PRINT @sql
    -- EXECUTE(@sql)



    -- For each database

    DECLARE @sqlstatement nvarchar(4000)
    --move declare cursor into sql to be executed
    SET @sqlstatement = 'DECLARE  columns_cursor CURSOR FOR SELECT TABLE_SCHEMA, TABLE_NAME,
COLUMN_NAME FROM ' + QUOTENAME(@db_name) + '.information_schema.columns ORDER BY TABLE_SCHEMA,
TABLE_NAME, ORDINAL_POSITION'



    EXEC sp_executesql @sqlstatement


    OPEN columns_cursor
    FETCH NEXT FROM columns_cursor
```

```
    INTO @schema, @table, @column

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @schema + '.' + @table + '.' + @column
        --EXEC asp_DoSomethingStoredProc @UserId

    FETCH NEXT FROM columns_cursor --have to fetch again within loop
    INTO @schema, @table, @column

    END
    CLOSE columns_cursor
    DEALLOCATE columns_cursor

    -- End for each database




    FETCH NEXT FROM db_cursor INTO @db_name
END

CLOSE db_cursor
DEALLOCATE db_cursor
```

Lea CURSOR SQL en línea: https://riptutorial.com/es/sql/topic/8895/cursor-sql

Capítulo 19: Diseño de la mesa

Observaciones

The Open University (1999) Sistemas de bases de datos relacionales: Bloque 2 Teoría relacional, Milton Keynes, The Open University.

Examples

Propiedades de una mesa bien diseñada.

Una verdadera base de datos relacional debe ir más allá de lanzar datos en unas pocas tablas y escribir algunas declaraciones SQL para extraer esos datos.
En el mejor de los casos, una estructura de tabla mal diseñada ralentizará la ejecución de las consultas y podría hacer imposible que la base de datos funcione según lo previsto.

Una tabla de base de datos no debe considerarse simplemente como otra tabla; Tiene que seguir un conjunto de reglas para ser considerado verdaderamente relacional. Académicamente se le conoce como una 'relación' para hacer la distinción.

Las cinco reglas de una tabla relacional son:

1. Cada valor es *atómico* ; el valor en cada campo en cada fila debe ser un solo valor.
2. Cada campo contiene valores que son del mismo tipo de datos.
3. Cada encabezado de campo tiene un nombre único.
4. Cada fila de la tabla debe tener al menos un valor que la haga única entre los otros registros de la tabla.
5. El orden de las filas y columnas no tiene importancia.

Una tabla conforme a las cinco reglas:

Carné de identidad	Nombre	Fecha de nacimiento	Gerente
1	Fred	02/11/1971	3
2	Fred	02/11/1971	3
3	demandar	07/08/1975	2

- Regla 1: Cada valor es atómico. `Id` , `Name` , `DOB` y `Manager` solo contienen un único valor.
- Regla 2: la `Id` solo contiene números enteros, el `Name` contiene texto (podríamos agregar texto de cuatro caracteres o menos), el `DOB` contiene fechas de un tipo válido y el `Manager` contiene números enteros (podríamos agregar que corresponda a un campo de clave principal en un administrador mesa).
- Regla 3: `Id` , `Name` , `DOB` y `Manager` son nombres de encabezado únicos dentro de la tabla.

- Regla 4: la inclusión del campo `Id` asegura que cada registro sea distinto de cualquier otro registro dentro de la tabla.

Una mesa mal diseñada:

Carné de identidad	Nombre	Fecha de nacimiento	Nombre
1	Fred	02/11/1971	3
1	Fred	02/11/1971	3
3	demandar	Viernes 18 de julio de 1975.	2, 1

- Regla 1: El segundo campo de nombre contiene dos valores: 2 y 1.
- Regla 2: El campo DOB contiene fechas y texto.
- Regla 3: Hay dos campos llamados 'nombre'.
- Regla 4: El primer y segundo registro son exactamente iguales.
- Regla 5: Esta regla no está rota.

Lea Diseño de la mesa en línea: https://riptutorial.com/es/sql/topic/2515/diseno-de-la-mesa

Capítulo 20: Ejemplo de bases de datos y tablas

Examples

Base de Datos de Auto Shop

En el siguiente ejemplo, la base de datos para un negocio de taller de automóviles, tenemos una lista de departamentos, empleados, clientes y automóviles de clientes. Estamos utilizando claves externas para crear relaciones entre las distintas tablas.

Ejemplo en vivo: violín de SQL

Relaciones entre tablas

- Cada departamento puede tener 0 o más empleados
- Cada empleado puede tener 0 o 1 gerente
- Cada cliente puede tener 0 o más coches

Departamentos

Carné de identidad	Nombre
1	HORA
2	Ventas
3	Tecnología

Sentencias SQL para crear la tabla:

```
CREATE TABLE Departments (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(25) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Departments
    ([Id], [Name])
VALUES
    (1, 'HR'),
    (2, 'Sales'),
    (3, 'Tech')
;
```

Empleados

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratació
1	James	Herrero	1234567890	NULO	1	1000	01-01-2002
2	Juan	Johnson	2468101214	1	1	400	23-03-2005
3	Miguel	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Herrero	1212121212	2	1	500	24-07-2016

Sentencias SQL para crear la tabla:

```
CREATE TABLE Employees (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    PhoneNumber VARCHAR(11),
    ManagerId INT,
    DepartmentId INT NOT NULL,
    Salary INT NOT NULL,
    HireDate DATETIME NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
    FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO Employees
    ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
    (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
    (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
    (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
    (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

Clientes

Carné de identidad	FName	LName	Email	Número de teléfono	Contacto preferido
1	William	Jones	william.jones@example.com	3347927472	TELÉFONO
2	David	Molinero	dmiller@example.net	2137921892	CORREO ELECTRÓNICO
3	Ricardo	Davis	richard0123@example.com	NULO	CORREO ELECTRÓNICO

Sentencias SQL para crear la tabla:

```
CREATE TABLE Customers (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    Email varchar(100) NOT NULL,
    PhoneNumber VARCHAR(11),
    PreferredContact VARCHAR(5) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Customers
    ([Id], [FName], [LName], [Email], [PhoneNumber], [PreferredContact])
VALUES
    (1, 'William', 'Jones', 'william.jones@example.com', '3347927472', 'PHONE'),
    (2, 'David', 'Miller', 'dmiller@example.net', '2137921892', 'EMAIL'),
    (3, 'Richard', 'Davis', 'richard0123@example.com', NULL, 'EMAIL')
;
```

Coches

Carné de identidad	Identificación del cliente	ID de empleado	Modelo	Estado	Coste total
1	1	2	Ford F-150	LISTO	230
2	1	2	Ford F-150	LISTO	200
3	2	1	Ford Mustang	ESPERANDO	100
4	3	3	Toyota Prius	TRABAJANDO	1254

Sentencias SQL para crear la tabla:

```
CREATE TABLE Cars (
    Id INT NOT NULL AUTO_INCREMENT,
    CustomerId INT NOT NULL,
    EmployeeId INT NOT NULL,
    Model varchar(50) NOT NULL,
    Status varchar(25) NOT NULL,
    TotalCost INT NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
    FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
```

```
    ([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
    ('1', '1', '2', 'Ford F-150', 'READY', '230'),
    ('2', '1', '2', 'Ford F-150', 'READY', '200'),
    ('3', '2', '1', 'Ford Mustang', 'WAITING', '100'),
    ('4', '3', '3', 'Toyota Prius', 'WORKING', '1254')
;
```

Base de datos de la biblioteca

En esta base de datos de ejemplo para una biblioteca, tenemos tablas de *Autores* , *Libros* y *Libros de Autores* .

Ejemplo en vivo: violín de SQL

Los autores y los *libros* se conocen como **tablas base** , ya que contienen definición de columna y datos para las entidades reales en el modelo relacional. *BooksAuthors* se conoce como la **tabla de relaciones** , ya que esta tabla define la relación entre la tabla *Libros* y *Autores* .

Relaciones entre tablas

- Cada autor puede tener 1 o más libros.
- Cada libro puede tener 1 o más autores.

Autores

(*ver tabla*)

Carné de identidad	Nombre	País
1	JD Salinger	Estados Unidos
2	F. Scott. Fitzgerald	Estados Unidos
3	Jane Austen	Reino Unido
4	Scott Hanselman	Estados Unidos
5	Jason N. Gaylord	Estados Unidos
6	Pranav Rastogi	India
7	Todd miranda	Estados Unidos
8	Christian Wenz	Estados Unidos

SQL para crear la tabla:

```
CREATE TABLE Authors (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(70) NOT NULL,
    Country VARCHAR(100) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Authors
    (Name, Country)
VALUES
    ('J.D. Salinger', 'USA'),
    ('F. Scott. Fitzgerald', 'USA'),
    ('Jane Austen', 'UK'),
    ('Scott Hanselman', 'USA'),
    ('Jason N. Gaylord', 'USA'),
    ('Pranav Rastogi', 'India'),
    ('Todd Miranda', 'USA'),
    ('Christian Wenz', 'USA')
;
```

Libros

(*ver tabla*)

Carné de identidad	Título
1	El Guardian en el centeno
2	Nueve historias
3	Franny y Zooey
4	El gran Gatsby
5	Tierna id la noche
6	Orgullo y prejuicio
7	Profesional ASP.NET 4.5 en C # y VB

SQL para crear la tabla:

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
    (Id, Title)
VALUES
    (1, 'The Catcher in the Rye'),
    (2, 'Nine Stories'),
```

```
    (3, 'Franny and Zooey'),
    (4, 'The Great Gatsby'),
    (5, 'Tender id the Night'),
    (6, 'Pride and Prejudice'),
    (7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

LibrosAutoras

(*ver tabla*)

BookId	AuthorId
1	1
2	1
3	1
4	2
5	2
6	3
7	4
7	5
7	6
7	7
7	8

SQL para crear la tabla:

```
CREATE TABLE BooksAuthors (
    AuthorId INT NOT NULL,
    BookId  INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES Authors(Id),
    FOREIGN KEY (BookId) REFERENCES Books(Id)
);

INSERT INTO BooksAuthors
    (BookId, AuthorId)
VALUES
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 2),
```

```
    (5, 2),
    (6, 3),
    (7, 4),
    (7, 5),
    (7, 6),
    (7, 7),
    (7, 8)
;
```

Ejemplos

Ver todos los autores (ver ejemplo en vivo):

```
SELECT * FROM Authors;
```

Ver todos los títulos de libros (ver ejemplo en vivo):

```
SELECT * FROM Books;
```

Ver todos los libros y sus autores (ver ejemplo en vivo):

```
SELECT
  ba.AuthorId,
  a.Name AuthorName,
  ba.BookId,
  b.Title BookTitle
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
  INNER JOIN Books b ON b.id = ba.bookid
;
```

Tabla de países

En este ejemplo, tenemos una tabla de **países** . Una tabla para países tiene muchos usos, especialmente en aplicaciones financieras que involucran divisas y tipos de cambio.

Ejemplo en vivo: violín de SQL

Algunas aplicaciones de software de datos de mercado como Bloomberg y Reuters requieren que le asigne a su API un código de país de 2 o 3 caracteres junto con el código de moneda. Por lo tanto, esta tabla de ejemplo tiene la columna de código `ISO` 2 caracteres y las columnas de código `ISO3` 3 caracteres.

Países

(*ver tabla*)

Carné de identidad	YO ASI	ISO3	Isonumérico	Nombre del país	Capital	Código Continente	Código moneda
1	AU	AUS	36	Australia	Canberra	jefe	AUD
2	Delaware	DEU	276	Alemania	Berlina	UE	EUR
2	EN	INDIANA	356	India	Nueva Delhi	COMO	INR
3	LA	LAO	418	Laos	Vientiane	COMO	LAGO
4	NOSOTROS	Estados Unidos	840	Estados Unidos	Washington	N / A	Dólar estadou
5	ZW	ZWE	716	Zimbabue	Harare	AF	ZWL

SQL para crear la tabla:

```
CREATE TABLE Countries (
    Id INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    ISONumeric INT NOT NULL,
    CountryName VARCHAR(64) NOT NULL,
    Capital VARCHAR(64) NOT NULL,
    ContinentCode VARCHAR(2) NOT NULL,
    CurrencyCode VARCHAR(3) NOT NULL,
    PRIMARY KEY(Id)
)
;

INSERT INTO Countries
    (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, 'Australia', 'Canberra', 'OC', 'AUD'),
    ('DE', 'DEU', 276, 'Germany', 'Berlin', 'EU', 'EUR'),
    ('IN', 'IND', 356, 'India', 'New Delhi', 'AS', 'INR'),
    ('LA', 'LAO', 418, 'Laos', 'Vientiane', 'AS', 'LAK'),
    ('US', 'USA', 840, 'United States', 'Washington', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, 'Zimbabwe', 'Harare', 'AF', 'ZWL')
;
```

Lea Ejemplo de bases de datos y tablas en línea: https://riptutorial.com/es/sql/topic/280/ejemplo-de-bases-de-datos-y-tablas

Capítulo 21: Eliminar en cascada

Examples

En la eliminación de cascadas

Supongamos que tiene una aplicación que administra salas.
Supongamos además que su aplicación funciona por cliente (inquilino).
Tienes varios clientes.
Por lo tanto, su base de datos contendrá una tabla para clientes y una para salas.

Ahora, cada cliente tiene N habitaciones.

Esto debería significar que tiene una clave externa en la tabla de su sala, que hace referencia a la tabla del cliente.

```
ALTER TABLE dbo.T_Room  WITH CHECK ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

Suponiendo que un cliente pasa a algún otro software, tendrá que eliminar sus datos en su software. Pero si lo haces

```
DELETE FROM T_Client WHERE CLI_ID = x
```

Entonces obtendrás una violación de clave externa, porque no puedes eliminar al cliente cuando todavía tiene habitaciones.

Ahora tendría un código de escritura en su aplicación que elimina las habitaciones del cliente antes de que elimine al cliente. Supongamos además que, en el futuro, se agregarán muchas más dependencias de clave externa en su base de datos, ya que la funcionalidad de su aplicación se expande. Horrible. Para cada modificación en su base de datos, tendrá que adaptar el código de su aplicación en N lugares. Es posible que también tenga que adaptar el código en otras aplicaciones (por ejemplo, interfaces a otros sistemas).

Hay una mejor solución que hacerlo en tu código.
Solo puede agregar `ON DELETE CASCADE` a su clave externa.

```
ALTER TABLE dbo.T_Room  -- WITH CHECK -- SQL-Server can specify WITH CHECK/WITH NOCHECK
ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

Ahora puedes decir

```
DELETE FROM T_Client WHERE CLI_ID = x
```

y las habitaciones se eliminan automáticamente cuando se elimina el cliente.
Problema resuelto - sin cambios en el código de la aplicación.

Una palabra de advertencia: en Microsoft SQL-Server, esto no funcionará si tiene una tabla que hace referencia a sí misma. Entonces, si intentas definir una cascada de eliminación en una estructura de árbol recursiva, como esto:

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation]  WITH CHECK ADD  CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
GO

IF  EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

no funcionará, porque el servidor Microsoft-SQL no le permite establecer una clave externa con `ON DELETE CASCADE` en una estructura de árbol recursiva. Una de las razones de esto es que el árbol es posiblemente cíclico, y eso podría conducir a un punto muerto.

PostgreSQL por otro lado puede hacer esto;
El requisito es que el árbol no sea cíclico.
Si el árbol es cíclico, obtendrá un error de tiempo de ejecución.
En ese caso, solo tendrá que implementar la función de eliminación usted mismo.

Una palabra de precaución:
Esto significa que ya no puede simplemente eliminar y volver a insertar la tabla de clientes, porque si lo hace, eliminará todas las entradas en "T_Room" ... (ya no hay actualizaciones que no sean delta)

Lea Eliminar en cascada en línea: https://riptutorial.com/es/sql/topic/3518/eliminar-en-cascada

Capítulo 22: Encontrar duplicados en un subconjunto de columnas con detalles

Observaciones

- Para seleccionar filas sin duplicados, cambie la cláusula WHERE a "RowCnt = 1"
- Para seleccionar una fila de cada conjunto use Rank () en lugar de Sum () y cambie la cláusula WHERE externa para seleccionar filas con Rank () = 1

Examples

Alumnos con el mismo nombre y fecha de nacimiento.

```
WITH CTE (StudentId, Fname, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By
FirstName, LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

Este ejemplo utiliza una expresión de tabla común y una función de ventana para mostrar todas las filas duplicadas (en un subconjunto de columnas) lado a lado.

Lea Encontrar duplicados en un subconjunto de columnas con detalles en línea: https://riptutorial.com/es/sql/topic/1585/encontrar-duplicados-en-un-subconjunto-de-columnas-con-detalles

Capítulo 23: Esquema de información

Examples

Búsqueda básica de esquemas de información

Una de las consultas más útiles para usuarios finales de RDBMS grandes es la búsqueda de un esquema de información.

Dicha consulta permite a los usuarios encontrar rápidamente tablas de bases de datos que contienen columnas de interés, como cuando intentan relacionar datos de 2 tablas indirectamente a través de una tercera tabla, sin el conocimiento existente de qué tablas pueden contener claves u otras columnas útiles en común con las tablas de destino .

Al usar T-SQL para este ejemplo, se puede buscar el esquema de información de una base de datos de la siguiente manera:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

El resultado contiene una lista de columnas coincidentes, los nombres de sus tablas y otra información útil.

Lea Esquema de información en línea: https://riptutorial.com/es/sql/topic/3151/esquema-de-informacion

Capítulo 24: EXCEPTO

Observaciones

`EXCEPT` devuelve cualquier valor distinto del conjunto de datos a la izquierda del operador EXCEPT que no se devuelve también desde el conjunto de datos derecho.

Examples

Seleccione el conjunto de datos, excepto donde los valores están en este otro conjunto de datos

```
--dataset schemas must be identical
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column' UNION ALL
SELECT 'Data3' as 'Column' UNION ALL
SELECT 'Data4' as 'Column' UNION ALL
SELECT 'Data5' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
--Returns Data1, Data2, Data4, and Data5
```

Lea EXCEPTO en línea: https://riptutorial.com/es/sql/topic/4082/excepto

Capítulo 25: Explique y describa

Examples

DESCRIBIR nombre de tabla;

`DESCRIBE` y `EXPLAIN` son sinónimos. `DESCRIBE` en un nombre de tabla devuelve la definición de las columnas.

```
DESCRIBE tablename;
```

Ejemplo de resultado:

```
COLUMN_NAME     COLUMN_TYPE     IS_NULLABLE     COLUMN_KEY     COLUMN_DEFAULT    EXTRA
id              int(11)         NO              PRI            0
auto_increment
test            varchar(255)    YES                            (null)
```

Aquí se ven los nombres de las columnas, seguidos por el tipo de columnas. Muestra si se permite `null` en la columna y si la columna utiliza un índice. el valor predeterminado también se muestra y si la tabla contiene algún comportamiento especial como un `auto_increment` .

EXPLICAR Seleccionar consulta

Una `Explain` delante de una consulta de `select` muestra cómo se ejecutará la consulta. De esta manera, puede ver si la consulta utiliza un índice o si podría optimizar su consulta agregando un índice.

Consulta de ejemplo:

```
explain select * from user join data on user.test = data.fk_user;
```

Resultado de ejemplo:

```
id  select_type  table   type    possible_keys  key     key_len ref       rows  Extra
1   SIMPLE       user    index   test           test    5       (null)    1     Using where;
Using index
1   SIMPLE       data    ref     fk_user        fk_user 5       user.test 1     (null)
```

en el `type` se ve si se utilizó un índice. En la columna `possible_keys` verá si el plan de ejecución puede elegir entre diferentes índices o si no existe ninguno. `key` le indica el índice de uso actual. `key_len` muestra el tamaño en bytes para un elemento de índice. Cuanto más bajo sea este valor, más elementos de índice caben en el mismo tamaño de memoria y pueden procesarse más rápidamente. `rows` muestran el número esperado de filas que la consulta necesita escanear, cuanto más baja mejor.

Lea Explique y describa en línea: https://riptutorial.com/es/sql/topic/2928/explique-y-describa

Capítulo 26: Expresiones de mesa comunes

Sintaxis

- WITH QueryName [(ColumnName, ...)] AS (
  SELECCIONAR ...
)
  SELECT ... FROM QueryName ...;

- CON RECURSIVE QueryName [(ColumnName, ...)] AS (
  SELECCIONAR ...
  UNION [TODOS]
  SELECCIONAR ... DE QueryName ...
)
  SELECT ... FROM QueryName ...;

Observaciones

Documentación oficial: con cláusula.

Una expresión de tabla común es un conjunto de resultados temporal y puede ser el resultado de una sub consulta compleja. Se define utilizando la cláusula WITH. CTE mejora la legibilidad y se crea en la memoria en lugar de en la base de datos TempDB donde se crean la tabla Temp y la variable Table.

Conceptos clave de las expresiones de tabla comunes:

- Se puede utilizar para dividir consultas complejas, especialmente combinaciones complejas y subconsultas.
- Es una forma de encapsular una definición de consulta.
- Persistir solo hasta que se ejecute la siguiente consulta.
- El uso correcto puede llevar a mejoras tanto en la calidad / mantenibilidad del código como en la velocidad.
- Se puede usar para hacer referencia a la tabla resultante varias veces en la misma declaración (eliminar la duplicación en SQL).
- Puede ser un sustituto de una vista cuando no se requiere el uso general de una vista; es decir, no es necesario almacenar la definición en metadatos.
- Se ejecutará cuando se llame, no cuando esté definido. Si el CTE se usa varias veces en una consulta, se ejecutará varias veces (posiblemente con resultados diferentes).

Examples

Consulta temporal

Se comportan de la misma manera que las subconsultas anidadas pero con una sintaxis

diferente.

```
WITH ReadyCars AS (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
)
SELECT ID, Model, TotalCost
FROM ReadyCars
ORDER BY TotalCost;
```

CARNÉ DE IDENTIDAD	Modelo	Coste total
1	Ford F-150	200
2	Ford F-150	230

Sintaxis de subconsulta equivalente

```
SELECT ID, Model, TotalCost
FROM (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
) AS ReadyCars
ORDER BY TotalCost
```

subiendo recursivamente en un árbol

```
WITH RECURSIVE ManagersOfJonathon AS (
    -- start with this row
    SELECT *
    FROM Employees
    WHERE ID = 4

    UNION ALL

    -- get manager(s) of all previously selected rows
    SELECT Employees.*
    FROM Employees
    JOIN ManagersOfJonathon
        ON Employees.ID = ManagersOfJonathon.ManagerID
)
SELECT * FROM ManagersOfJonathon;
```

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId
4	Johnathon	Herrero	1212121212	**2**	1
2	Juan	Johnson	2468101214	**1**	1
1	James	Herrero	1234567890	NULO	1

generando valores

La mayoría de las bases de datos no tienen una forma nativa de generar una serie de números para uso ad-hoc; sin embargo, las expresiones de tabla comunes pueden usarse con recursión para emular ese tipo de función.

El siguiente ejemplo genera una expresión de tabla común llamada `Numbers` con una columna `i` que tiene una fila para los números del 1 al 5:

```
--Give a table name `Numbers" and a column `i` to hold the numbers
WITH Numbers(i) AS (
    --Starting number/index
    SELECT 1
    --Top-level UNION ALL operator required for recursion
    UNION ALL
    --Iteration expression:
    SELECT i + 1
    --Table expression we first declared used as source for recursion
    FROM Numbers
    --Clause to define the end of the recursion
    WHERE i < 5
)
--Use the generated table expression like a regular table
SELECT i FROM Numbers;
```

yo
1
2
3
4
5

Este método se puede utilizar con cualquier intervalo de números, así como con otros tipos de datos.

enumeración recursiva de un subárbol

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
    -- start with this row
    SELECT 1, ID, FName, LName
    FROM Employees
    WHERE ID = 1

    UNION ALL

    -- get employees that have any of the previously selected rows as manager
    SELECT ManagedByJames.Level + 1,
           Employees.ID,
```

```
          Employees.FName,
          Employees.LName
    FROM Employees
    JOIN ManagedByJames
        ON Employees.ManagerID = ManagedByJames.ID

    ORDER BY 1 DESC   -- depth-first search
)
SELECT * FROM ManagedByJames;
```

Nivel	CARNÉ DE IDENTIDAD	FName	LName
1	1	James	Herrero
2	2	Juan	Johnson
3	4	Johnathon	Herrero
2	3	Miguel	Williams

Funcionalidad Oracle CONNECT BY con CTEs recursivas

La funcionalidad CONNECT BY de Oracle proporciona muchas características útiles y no triviales que no están integradas cuando se usan CTE recursivos estándar de SQL. Este ejemplo replica estas características (con algunas adiciones para completar), utilizando la sintaxis de SQL Server. Es más útil para los desarrolladores de Oracle que encuentran muchas características que faltan en sus consultas jerárquicas en otras bases de datos, pero también sirve para mostrar lo que se puede hacer con una consulta jerárquica en general.

```
  WITH tbl AS (
       SELECT id, name, parent_id
         FROM mytable)
     , tbl_hierarchy AS (
       /* Anchor */
       SELECT 1 AS "LEVEL"
            --, 1 AS CONNECT_BY_ISROOT
            --, 0 AS CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , 0 AS CONNECT_BY_ISCYCLE
            , '/' + CAST(t.id   AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
            , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
            , t.id AS root_id
            , t.*
         FROM tbl t
        WHERE t.parent_id IS NULL                              -- START WITH parent_id IS NULL
       UNION ALL
       /* Recursive */
       SELECT th."LEVEL" + 1 AS "LEVEL"
            --, 0 AS CONNECT_BY_ISROOT
            --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) +
```

```
'/%' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
            , th.SYS_CONNECT_BY_PATH_id   + CAST(t.id   AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
            , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_name
            , th.root_id
            , t.*
         FROM tbl t
              JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id =
parent_id
        WHERE th.CONNECT_BY_ISCYCLE = 0)                     -- NOCYCLE
SELECT th.*
     --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
  FROM tbl_hierarchy th
       JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
 ORDER BY th.SYS_CONNECT_BY_PATH_name;                       -- ORDER SIBLINGS BY name
```

CONECTAR POR características demostradas anteriormente, con explicaciones:

- Cláusulas
  - CONECTAR POR: Especifica la relación que define la jerarquía.
  - COMENZAR CON: Especifica los nodos raíz.
  - ORDEN DE HABLAR POR: Ordena los resultados correctamente.
- Parámetros
  - NOCYCLE: detiene el procesamiento de una rama cuando se detecta un bucle. Las jerarquías válidas son Gráficos Acíclicos Dirigidos, y las referencias circulares violan esta construcción.
- Los operadores
  - ANTES: Obtiene datos del padre del nodo.
  - CONNECT_BY_ROOT: Obtiene datos de la raíz del nodo.
- Pseudocolumnas
  - NIVEL: indica la distancia del nodo desde su raíz.
  - CONNECT_BY_ISLEAF: indica un nodo sin hijos.
  - CONNECT_BY_ISCYCLE: indica un nodo con una referencia circular.
- Funciones
  - SYS_CONNECT_BY_PATH: devuelve una representación aplanada / concatenada de la ruta al nodo desde su raíz.

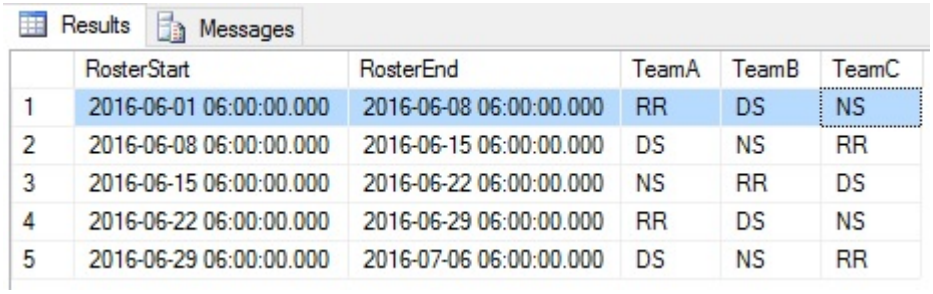
Generar fechas recursivamente, extendido para incluir la lista de equipos como ejemplo

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- Transition Sequence = Rest & Relax into Day Shift into Night Shift
-- RR (Rest & Relax) = 1
-- DS (Day Shift) = 2
-- NS (Night Shift) = 3

;WITH roster AS
(
   SELECT @DateFrom AS RosterStart, 1 AS TeamA, 2 AS TeamB, 3 AS TeamC
```

```
   UNION ALL
   SELECT DATEADD(d, @IntervalDays, RosterStart),
          CASE TeamA WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamA,
          CASE TeamB WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamB,
          CASE TeamC WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamC
   FROM roster WHERE RosterStart < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT RosterStart,
       ISNULL(LEAD(RosterStart) OVER (ORDER BY RosterStart), RosterStart + @IntervalDays) AS
RosterEnd,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

Resultado

Por ejemplo, para la semana 1, el equipo está en R&R, el equipo B está en el turno de día y el equipo C está en el turno de noche.


Results Messages

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

Refactorizar una consulta para usar expresiones de tabla comunes

Supongamos que queremos obtener todas las categorías de productos con ventas totales superiores a 20.

Aquí hay una consulta sin expresiones de tabla comunes:

```
SELECT category.description, sum(product.price) as total_sales
FROM sale
LEFT JOIN product on sale.product_id = product.id
LEFT JOIN category on product.category_id = category.id
GROUP BY category.id, category.description
HAVING sum(product.price) > 20
```

Y una consulta equivalente usando expresiones de tabla comunes:

```
WITH all_sales AS (
  SELECT product.price, category.id as category_id, category.description as
category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
, sales_by_category AS (
  SELECT category_description, sum(price) as total_sales
```

```
  FROM all_sales
  GROUP BY category_id, category_description
)
SELECT * from sales_by_category WHERE total_sales > 20
```

Ejemplo de un SQL complejo con expresión de tabla común

Supongamos que queremos consultar los "productos más baratos" de las "categorías principales".

Aquí hay un ejemplo de consulta usando expresiones de tabla comunes

```
-- all_sales: just a simple SELECT with all the needed JOINS
WITH all_sales AS (
  SELECT
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
-- Group by category
, sales_by_category AS (
  SELECT category_id, category_description,
  sum(product_price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
-- Filtering total_sales > 20
, top_categories AS (
  SELECT * from sales_by_category WHERE total_sales > 20
)
-- all_products: just a simple SELECT with all the needed JOINS
, all_products AS (
  SELECT
  product.id as product_id,
  product.description as product_description,
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM product
  LEFT JOIN category on product.category_id = category.id
)
-- Order by product price
, cheapest_products AS (
  SELECT * from all_products
  ORDER by product_price ASC
)
-- Simple inner join
, cheapest_products_from_top_categories AS (
  SELECT product_description, product_price
  FROM cheapest_products
  INNER JOIN top_categories ON cheapest_products.category_id = top_categories.category_id
)
--The main SELECT
SELECT * from cheapest_products_from_top_categories
```

Lea Expresiones de mesa comunes en línea: https://riptutorial.com/es/sql/topic/747/expresiones-

de-mesa-comunes

Capítulo 27: Filtrar los resultados usando WHERE y HAVING.

Sintaxis

- SELECCIONAR nombre_columna
  FROM nombre_tabla
  DÓNDE nombre del operador column_name
- SELECCIONAR column_name, aggregate_function (column_name)
  FROM nombre_tabla
  GRUPO POR nombre_columna
  TENIENDO el valor del operador aggregate_function (column_name)

Examples

La cláusula WHERE solo devuelve filas que coinciden con sus criterios

Steam tiene una sección de juegos por debajo de $ 10 en la página de su tienda. En algún lugar profundo del corazón de sus sistemas, probablemente haya una consulta que se parece a algo como:

```
SELECT *
FROM Items
WHERE Price < 10
```

Utilice IN para devolver filas con un valor contenido en una lista

Este ejemplo utiliza la tabla de coches de las bases de datos de ejemplo.

```
SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)
```

Esta consulta devolverá el Car # 2, que cuesta 200 y el Car # 3, que cuesta 100. Tenga en cuenta que esto es equivalente a usar varias cláusulas con `OR` , por ejemplo:

```
SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Usa LIKE para encontrar cadenas y subcadenas que coincidan

Ver la documentación completa en el operador LIKE .

Este ejemplo utiliza la tabla de empleados de las bases de datos de ejemplo.

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

Esta consulta solo devolverá al Empleado # 1 cuyo nombre coincide exactamente con 'John'.

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

Agregar `%` permite buscar una subcadena:

- `John%` : devolverá a cualquier Empleado cuyo nombre comience con 'John', seguido de cualquier cantidad de caracteres
- `%John` : devolverá a cualquier Empleado cuyo nombre termine con 'John', seguido por cualquier cantidad de caracteres
- `%John%` : devolverá a cualquier empleado cuyo nombre contenga "John" en cualquier lugar dentro del valor

En este caso, la consulta devolverá al Empleado # 2 cuyo nombre es 'John', así como al Empleado # 4 cuyo nombre es 'Johnathon'.

Cláusula WHERE con valores NULL / NOT NULL

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

Esta declaración devolverá todos los registros de Empleado donde el valor de la columna `ManagerId` es `NULL` .

El resultado será:

```
Id    FName    LName    PhoneNumber    ManagerId    DepartmentId
1     James    Smith    1234567890     NULL         1
```

```
SELECT *
FROM Employees
WHERE ManagerId IS NOT NULL
```

Esta declaración devolverá todos los registros de Empleado donde el valor de `ManagerId` *no sea* `NULL` .

El resultado será:

```
Id    FName       LName     PhoneNumber    ManagerId    DepartmentId
2     John        Johnson   2468101214     1            1
```

```
3     Michael     Williams  1357911131     1            2
4     Johnathon   Smith     1212121212     2            1
```

Nota: la misma consulta no devolverá resultados si cambia la cláusula `WHERE ManagerId = NULL` a `WHERE ManagerId = NULL` o `WHERE ManagerId <> NULL` .

Usar HAVING con funciones agregadas

A diferencia de la cláusula `WHERE` , `HAVING` se puede usar con funciones agregadas.

> Una función agregada es una función donde los valores de varias filas se agrupan como entrada en ciertos criterios para formar un valor único de significado o medida más significativo (Wikipedia).

Las funciones agregadas comunes incluyen `COUNT()` , `SUM()` , `MIN()` y `MAX()` .

Este ejemplo utiliza la tabla de coches de las bases de datos de ejemplo.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

Esta consulta devolverá el `Number of Cars` `CustomerId` y `Number of Cars` de cualquier cliente que tenga más de un auto. En este caso, el único cliente que tiene más de un automóvil es el Cliente # 1.

Los resultados se verán como:

Identificación del cliente	Numero de autos
1	2

Use ENTRE para filtrar los resultados

Los siguientes ejemplos utilizan las bases de datos de muestra Item Sales and Customers .

> Nota: El operador BETWEEN *es* inclusivo.

Usando el operador BETWEEN con números:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

Esta consulta devolverá todos los registros de `ItemSales` que tienen una cantidad mayor o igual a 10 y menor o igual a 17. Los resultados se verán como:

Carné de identidad	Fecha de venta	Identificación del artículo	Cantidad	Precio
1	2013-07-01	100	10	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

Usando el operador BETWEEN con valores de fecha:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

Esta consulta devolverá todos los registros de `ItemSales` con una fecha de `SaleDate` mayor o igual al 11 de julio de 2013 y menor o igual al 24 de mayo de 2013.

Carné de identidad	Fecha de venta	Identificación del artículo	Cantidad	Precio
3	2013-07-11	100	20	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

> Cuando compare valores de fecha y hora en lugar de fechas, es posible que deba convertir los valores de fecha y hora en valores de fecha, o sumar o restar 24 horas para obtener los resultados correctos.

Usando el operador BETWEEN con valores de texto:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L';
```

Ejemplo en vivo: violín de SQL

Esta consulta devolverá a todos los clientes cuyo nombre alfabéticamente se encuentre entre las letras 'D' y 'L'. En este caso, los clientes # 1 y # 3 serán devueltos. El cliente # 2, cuyo nombre comience con una 'M' no será incluido.

Carné de identidad	FName	LName
1	William	Jones
3	Ricardo	Davis

Igualdad

```
SELECT * FROM Employees
```

Esta declaración devolverá todas las filas de la tabla `Employees` .

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Usar un `WHERE` al final de su instrucción `SELECT` permite limitar las filas devueltas a una condición. En este caso, donde hay una coincidencia exacta usando el signo `=` :

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Solo devolverá las filas donde el `DepartmentId` es igual a `1` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Y y o

También puede combinar varios operadores para crear condiciones de `WHERE` más complejas. Los siguientes ejemplos utilizan la tabla `Employees` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Y

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Volverá

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
```

O

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Volverá

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Use HAVING para verificar múltiples condiciones en un grupo

Tabla de Pedidos

Identificación del cliente	Identificación de producto	Cantidad	Precio
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

Para verificar los clientes que han pedido ambos, ProductID 2 y 3, se puede usar HAVING

```
select customerId
from orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Valor de retorno:

Identificación del cliente
1

La consulta selecciona solo los registros con productIDs en las preguntas y con la cláusula

HAVING verifica los grupos que tienen 2 productIds y no solo uno.

Otra posibilidad seria

```
select customerId
from orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
   and sum(case when productID = 3 then 1 else 0 end) > 0
```

Esta consulta selecciona solo grupos que tienen al menos un registro con productID 2 y al menos uno con productID 3.

Donde exista

Seleccionará registros en `TableName` que tengan registros coincidentes en `TableName1` .

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Lea Filtrar los resultados usando WHERE y HAVING. en línea:
https://riptutorial.com/es/sql/topic/636/filtrar-los-resultados-usando-where-y-having-

Capítulo 28: Funciones (Agregado)

Sintaxis

- Función (expresión [*DISTINCT*]) -DISTINCT es un parámetro opcional
- AVG ([ALL | DISTINCT] expresión)
- COUNT ({[ALL | DISTINCT] expresión] | *})
- AGRUPACIÓN (<column_expression>)
- MAX (expresión [ALL | DISTINCT])
- MIN ([ALL | DISTINCT] expresión)
- SUM ([ALL | DISTINCT] expresión)
- VAR ([ALL | DISTINCT] expresión)
  OVER ([partition_by_clause] order_by_clause)
- VARP ([ALL | DISTINCT] expresión)
  OVER ([partition_by_clause] order_by_clause
- STDEV ([ALL | DISTINCT] expresión)
  OVER ([partition_by_clause] order_by_clause)
- STDEVP ([ALL | DISTINCT] expresión)
  OVER ([partition_by_clause] order_by_clause)

Observaciones

En la gestión de bases de datos, una función agregada es una función en la que los valores de varias filas se agrupan como entrada en ciertos criterios para formar un valor único de significado más significativo o medición, como un conjunto, una bolsa o una lista.

```
MIN         returns the smallest value in a given column
MAX         returns the largest value in a given column
SUM         returns the sum of the numeric values in a given column
AVG         returns the average value of a given column
COUNT       returns the total number of values in a given column
COUNT(*)    returns the number of rows in a table
GROUPING    Is a column or an expression that contains a column in a GROUP BY clause.
STDEV       returns the statistical standard deviation of all values in the specified
expression.
STDEVP      returns the statistical standard deviation for the population for all values in the
specified expression.
VAR         returns the statistical variance of all values in the specified expression. may be
followed by the OVER clause.
VARP        returns the statistical variance for the population for all values in the specified
expression.
```

> Las funciones agregadas se utilizan para calcular una "columna de datos numéricos devueltos" de su declaración `SELECT` . Básicamente, resumen los resultados de una columna particular de datos seleccionados. - SQLCourse2.com

Todas las funciones agregadas ignoran los valores NULL.

Examples

SUMA

`Sum` función suma del valor de todas las filas del grupo. Si se omite el grupo por cláusula, se suman todas las filas.

```
select sum(salary) TotalSalary
from employees;
```

Salario total
2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DepartmentId	Salario total
1	2000
2	500

Agregación condicional

Tabla de pagos

Cliente	Tipo de pago	Cantidad
Peter	Crédito	100
Peter	Crédito	300
Juan	Crédito	1000
Juan	Débito	500

```
select customer,
       sum(case when payment_type = 'credit' then amount else 0 end) as credit,
       sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
group by customer
```

Resultado:

Cliente	Crédito	Débito
Peter	400	0
Juan	1000	500

```
select customer,
       sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
       sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
group by customer
```

Resultado:

Cliente	credit_transaction_count	debit_transaction_count
Peter	2	0
Juan	1	1

AVG ()

La función agregada AVG () devuelve el promedio de una expresión dada, generalmente valores numéricos en una columna. Supongamos que tenemos una tabla que contiene el cálculo anual de la población en ciudades de todo el mundo. Los registros de la ciudad de Nueva York son similares a los de abajo:

Tabla de ejemplo

Nombre de la ciudad	población	año
Nueva York	8.550.405	2015
Nueva York	...	...
Nueva York	8,000,906	2005

Para seleccionar la población promedio de la ciudad de Nueva York, EE. UU. De una tabla que contiene nombres de ciudades, mediciones de población y años de medición de los últimos diez años:

CONSULTA

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Observe cómo el año de medición está ausente de la consulta, ya que la población se promedia a lo largo del tiempo.

RESULTADOS

Nombre de la ciudad	avg_population
Nueva York	8.250.754

> Nota: La función AVG () convertirá los valores a tipos numéricos. Esto es especialmente importante tenerlo en cuenta al trabajar con fechas.

Concatenación de listas

Crédito parcial a esta respuesta.

Enumerar concatenación agrega una columna o expresión combinando los valores en una sola cadena para cada grupo. Se puede especificar una cadena para delimitar cada valor (ya sea en blanco o una coma cuando se omite) y el orden de los valores en el resultado. Si bien no es parte del estándar de SQL, todos los principales proveedores de bases de datos relacionales lo admiten a su manera.

MySQL

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Oracle y DB2

```
SELECT ColumnA
     , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
```

```
 ORDER BY ColumnA;
```

servidor SQL

SQL Server 2016 y anteriores

(CTE incluido para alentar el principio DRY)

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName)
SELECT t0.ColumnA
     , STUFF((
       SELECT ',' + t1.ColumnB
         FROM CTE_TableName t1
        WHERE t1.ColumnA = t0.ColumnA
        ORDER BY t1.ColumnB
          FOR XML PATH('')), 1, 1, '') AS ColumnBs
  FROM CTE_TableName t0
 GROUP BY t0.ColumnA
 ORDER BY ColumnA;
```

SQL Server 2017 y SQL Azure

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

SQLite

sin pedir:

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

el pedido requiere una subconsulta o CTE:

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName
        ORDER BY ColumnA, ColumnB)
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM CTE_TableName
```

```
GROUP BY ColumnA
ORDER BY ColumnA;
```

Contar

Puedes contar el número de filas:

```
SELECT count(*) TotalRows
FROM employees;
```

TotalRows
4

O contar a los empleados por departamento:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DepartmentId	Numemployees
1	3
2	1

Puede contar sobre una columna / expresión con el efecto que no contará los valores `NULL` :

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

monseñor
3

(Hay una columna de ID de administrador de valor nulo)

También puede utilizar **DISTINCT** dentro de otra función, como **COUNT,** para buscar solo los miembros **DISTINCT** del conjunto para realizar la operación.

Por ejemplo:

```
SELECT COUNT(ContinentCode) AllCount
,      COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

Volverá valores diferentes. *SingleCount* solo contará los Continentes individuales una vez, mientras que *AllCount* incluirá duplicados.

Código Continente
jefe
UE
COMO
N / A
N / A
AF
AF

AllCount: 7 SingleCount: 5

Max

Encuentra el valor máximo de columna:

```
select max(age) from employee;
```

El ejemplo anterior devolverá el mayor valor para la `age` de la columna de `employee` tabla de `employee` .

Sintaxis:

```
SELECT MAX(column_name) FROM table_name;
```

Min

Encuentra el valor más pequeño de la columna:

```
 select min(age) from employee;
```

El ejemplo anterior devolverá el valor más pequeño para la `age` de la columna de `employee` tabla de `employee` .

Sintaxis:

```
 SELECT MIN(column_name) FROM table_name;
```

Lea Funciones (Agregado) en línea: https://riptutorial.com/es/sql/topic/1002/funciones--agregado-

Capítulo 29: Funciones (analíticas)

Introducción

Utiliza funciones analíticas para determinar valores basados en grupos de valores. Por ejemplo, puede usar este tipo de función para determinar totales acumulados, porcentajes o el resultado superior dentro de un grupo.

Sintaxis

1. FIRST_VALUE (scalar_expression) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
2. LAST_VALUE (scalar_expression) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
3. LAG (scalar_expression [, offset] [, predeterminado]) OVER ([partition_by_clause] order_by_clause)
4. LEAD (scalar_expression [, offset], [predeterminado]) OVER ([partition_by_clause] order_by_clause)
5. PERCENT_RANK () OVER ([partition_by_clause] order_by_clause)
6. CUME_DIST () OVER ([partition_by_clause] order_by_clause)
7. PERCENTILE_DISC (numeric_literal) WITHIN GROUP (ORDEN POR order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])
8. PERCENTILE_CONT (numeric_literal) WITHIN GROUP (ORDEN POR order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])

Examples

FIRST_VALUE

Utiliza la función `FIRST_VALUE` para determinar el primer valor en un conjunto de resultados ordenados, que identifica utilizando una expresión escalar.

```
SELECT StateProvinceID, Name, TaxRate,
       FIRST_VALUE(StateProvinceID)
        OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

En este ejemplo, la función `FIRST_VALUE` se usa para devolver el `ID` del estado o provincia con la tasa impositiva más baja. La cláusula `OVER` se utiliza para ordenar las tasas de impuestos para obtener la tasa más baja.

StateProvinceID	Nombre	Tasa de impuesto	FirstValue
74	Impuesto de ventas del estado de Utah	5.00	74

StateProvinceID	Nombre	Tasa de impuesto	FirstValue
36	Impuesto sobre las ventas del estado de Minnesota	6.75	74
30	Impuesto sobre las ventas del estado de Massachusetts	7.00	74
1	GST canadiense	7.00	74
57	GST canadiense	7.00	74
63	GST canadiense	7.00	74

LAST_VALUE

La función `LAST_VALUE` proporciona el último valor en un conjunto de resultados ordenados, que usted especifica usando una expresión escalar.

```
SELECT TerritoryID, StartDate, BusinessentityID,
       LAST_VALUE(BusinessentityID)
        OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

Este ejemplo utiliza la función `LAST_VALUE` para devolver el último valor para cada conjunto de filas en los valores ordenados.

ID de territorio	Fecha de inicio	BusinessentityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

LAG y LEAD

La función `LAG` proporciona datos en filas antes de la fila actual en el mismo conjunto de resultados. Por ejemplo, en una instrucción `SELECT` , puede comparar valores en la fila actual con valores en una fila anterior.

Utiliza una expresión escalar para especificar los valores que deben compararse. El parámetro de

desplazamiento es el número de filas antes de la fila actual que se utilizará en la comparación. Si no especifica el número de filas, se utiliza el valor predeterminado de una fila.

El parámetro predeterminado especifica el valor que debe devolverse cuando la expresión en el desplazamiento tiene un valor `NULL` . Si no especifica un valor, se devuelve un valor de `NULL` .

La función `LEAD` proporciona datos en filas después de la fila actual en el conjunto de filas. Por ejemplo, en una declaración `SELECT` , puede comparar valores en la fila actual con valores en la fila siguiente.

Usted especifica los valores que deben compararse usando una expresión escalar. El parámetro de desplazamiento es el número de filas después de la fila actual que se utilizará en la comparación.

Usted especifica el valor que debe devolverse cuando la expresión en el desplazamiento tiene un valor `NULL` usando el parámetro predeterminado. Si no especifica estos parámetros, se utiliza el valor predeterminado de una fila y se devuelve un valor de `NULL` .

```
SELECT BusinessEntityID, SalesYTD,
       LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
       LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

Este ejemplo utiliza las funciones LEAD y LAG para comparar los valores de ventas de cada empleado hasta la fecha con los de los empleados enumerados arriba y abajo, con registros ordenados según la columna BusinessEntityID.

BusinessEntityID	VentasYTD	Valor de plomo	Valor de retraso
274	559697.5639	3763178.1787	0.0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

PERCENT_RANK y CUME_DIST

La función `PERCENT_RANK` calcula la clasificación de una fila en relación con el conjunto de filas. El porcentaje se basa en el número de filas del grupo que tienen un valor inferior al de la fila actual.

El primer valor en el conjunto de resultados siempre tiene un rango de porcentaje de cero. El valor para el valor más alto (o último) en el conjunto es siempre uno.

La función `CUME_DIST` calcula la posición relativa de un valor especificado en un grupo de valores, determinando el porcentaje de valores menores o iguales a ese valor. Esto se llama la distribución acumulativa.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
PERCENT_RANK() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Percent Rank",
CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Cumulative Distribution"
FROM Employee;
```

En este ejemplo, usa una cláusula `ORDER` para particionar, o agrupar, las filas recuperadas por la declaración `SELECT` en función de los títulos de trabajo de los empleados, con los resultados en cada grupo ordenados según el número de horas de licencia por enfermedad que los empleados han utilizado.

BusinessEntityID	Título profesional	SickLeaveHours	Rango porcentual	Distribución acumulativa
267	Especialista en Aplicaciones	57	0	0.25
268	Especialista en Aplicaciones	56	0.333333333333333	0.75
269	Especialista en Aplicaciones	56	0.333333333333333	0.75
272	Especialista en Aplicaciones	55	1	1
262	Asistente del Oficial Financiero de Cheif	48	0	1
239	Especialista en Beneficios	45	0	1
252	Comprador	50	0	0.111111111111111
251	Comprador	49	0.125	0.333333333333333
256	Comprador	49	0.125	0.333333333333333

BusinessEntityID	Título profesional	SickLeaveHours	Rango porcentual	Distribución acumulativa
253	Comprador	48	0.375	0.555555555555555
254	Comprador	48	0.375	0.555555555555555

La función `PERCENT_RANK` clasifica las entradas dentro de cada grupo. Para cada entrada, devuelve el porcentaje de entradas en el mismo grupo que tienen valores más bajos.

La función `CUME_DIST` es similar, excepto que devuelve el porcentaje de valores menores o iguales al valor actual.

PERCENTILE_DISC y PERCENTILE_CONT

La función `PERCENTILE_DISC` enumera el valor de la primera entrada donde la distribución acumulada es más alta que el percentil que proporciona usando el parámetro `numeric_literal` .

Los valores se agrupan por conjunto de filas o partición, según lo especificado por la cláusula `WITHIN GROUP` .

La función `PERCENTILE_CONT` es similar a la función `PERCENTILE_DISC` , pero devuelve el promedio de la suma de la primera entrada coincidente y la entrada siguiente.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet"
FROM Employee;
```

Para encontrar el valor exacto de la fila que coincide o supera el percentil 0.5, se pasa el percentil como el literal numérico en la función `PERCENTILE_DISC` . La columna Percentil discreto en un conjunto de resultados enumera el valor de la fila en la cual la distribución acumulativa es más alta que el percentil especificado.

BusinessEntityID	Título profesional	SickLeaveHours	Distribución acumulativa	Percentil Discreto
272	Especialista en Aplicaciones	55	0.25	**56**
268	Especialista en Aplicaciones	56	0.75	**56**
269	Especialista en Aplicaciones	56	0.75	**56**
267	Especialista en	57	1	**56**

BusinessEntityID	Título profesional	SickLeaveHours	Distribución acumulativa	Percentil Discreto
	Aplicaciones			

Para basar el cálculo en un conjunto de valores, utilice la función `PERCENTILE_CONT` . La columna "Percentil Continuo" en los resultados enumera el valor promedio de la suma del valor del resultado y el siguiente valor coincidente más alto.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet",
       PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Continuous"
FROM Employee;
```

BusinessEntityID	Título profesional	SickLeaveHours	Distribución acumulativa	Percentil Discreto	Percentil Continuo
272	Especialista en Aplicaciones	55	0.25	56	**56**
268	Especialista en Aplicaciones	56	0.75	56	**56**
269	Especialista en Aplicaciones	56	0.75	56	**56**
267	Especialista en Aplicaciones	57	1	56	**56**

Lea Funciones (analíticas) en línea: https://riptutorial.com/es/sql/topic/8811/funciones--analiticas-

Capítulo 30: Funciones (escalar / fila única)

Introducción

SQL proporciona varias funciones escalares incorporadas. Cada función escalar toma un valor como entrada y devuelve un valor como salida para cada fila en un conjunto de resultados.

Utiliza funciones escalares siempre que se permita una expresión dentro de una instrucción T-SQL.

Sintaxis

- CAST (expresión AS tipo de datos [(longitud)])
- CONVERTIR (tipo_de_datos [(longitud)], expresión [, estilo])
- PARSE (string_value AS data_type [USING culture])
- NOMBRE DE DATOS (fecha parte, fecha)
- OBTENER LA FECHA ( )
- DATEDIFF (fecha, fecha de inicio, fecha de finalización)
- DATEADD (fecha, número, fecha)
- ELEGIR (índice, val_1, val_2 [, val_n])
- IIF (boolean_expression, true_value, false_value)
- SIGN (numeric_expression)
- PODER (float_expression, y)

Observaciones

Las funciones escalares o de una sola fila se utilizan para operar cada fila de datos en el conjunto de resultados, a diferencia de las funciones agregadas que operan en todo el conjunto de resultados.

Hay diez tipos de funciones escalares.

1. Las funciones de configuración proporcionan información sobre la configuración de la instancia de SQL actual.
2. Las funciones de conversión convierten los datos en el tipo de datos correcto para una operación determinada. Por ejemplo, estos tipos de funciones pueden reformatear la información al convertir una cadena en una fecha o número para permitir la comparación de dos tipos diferentes.
3. Las funciones de fecha y hora manipulan los campos que contienen valores de fecha y hora. Pueden devolver valores numéricos, de fecha o de cadena. Por ejemplo, puede usar una función para recuperar el día actual de la semana o el año o para recuperar solo el año de la fecha.

Los valores devueltos por las funciones de fecha y hora dependen de la fecha y la hora establecidas para el sistema operativo de la computadora que ejecuta la instancia de SQL.

4. Función lógica que realiza operaciones mediante operadores lógicos. Evalúa un conjunto de condiciones y devuelve un solo resultado.
5. Las funciones matemáticas realizan operaciones matemáticas, o cálculos, en expresiones numéricas. Este tipo de función devuelve un solo valor numérico.
6. Las funciones de metadatos recuperan información sobre una base de datos específica, como su nombre y los objetos de la base de datos.
7. Las funciones de seguridad proporcionan información que puede utilizar para administrar la seguridad de una base de datos, como información sobre los usuarios y roles de la base de datos.
8. Las funciones de cadena realizan operaciones en valores de cadena y devuelven valores numéricos o de cadena.

Mediante las funciones de cadena, puede, por ejemplo, combinar datos, extraer una subcadena, comparar cadenas o convertir una cadena a todos los caracteres en mayúscula o minúscula.

9. Las funciones del sistema realizan operaciones y devuelven información sobre valores, objetos y configuraciones para la instancia de SQL actual
10. Las funciones estadísticas del sistema proporcionan varias estadísticas sobre la instancia de SQL actual, por ejemplo, para que pueda monitorear los niveles de rendimiento actuales del sistema.

Examples

Modificaciones de caracteres

Las funciones de modificación de caracteres incluyen la conversión de caracteres a mayúsculas o minúsculas, la conversión de números a números formateados, la manipulación de caracteres, etc.

La función `lower(char)` convierte el parámetro de carácter dado en caracteres en minúsculas.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

devolvería el apellido del cliente cambiado de "SMITH" a "smith".

Fecha y hora

En SQL, utiliza los tipos de datos de fecha y hora para almacenar información de calendario. Estos tipos de datos incluyen la hora, la fecha, el tiempo pequeño, la fecha y la hora, la fecha y la hora y la fecha y hora. Cada tipo de datos tiene un formato específico.

Tipo de datos	Formato
hora	hh: mm: ss [.nnnnnnn]
fecha	YYYY-MM-DD

Tipo de datos	Formato
tiempo pequeño	YYYY-MM-DD hh: mm: ss
fecha y hora	YYYY-MM-DD hh: mm: ss [.nnn]
datetime2	YYYY-MM-DD hh: mm: ss [.nnnnnnn]
datetimeoffset	YYYY-MM-DD hh: mm: ss [.nnnnnnn] [+/-] hh: mm

La función `DATENAME` devuelve el nombre o valor de una parte específica de la fecha.

```
SELECT DATENAME (weekday,'2017-01-14') as Datename
```

Nombre de datos
sábado

Utiliza la función `GETDATE` para determinar la fecha y hora actuales de la computadora que ejecuta la instancia de SQL actual. Esta función no incluye la diferencia de zona horaria.

```
SELECT GETDATE() as Systemdate
```

Fecha del sistema
2017-01-14 11: 11: 47.7230728

La función `DATEDIFF` devuelve la diferencia entre dos fechas.

En la sintaxis, datepart es el parámetro que especifica qué parte de la fecha desea utilizar para calcular la diferencia. La fecha puede ser año, mes, semana, día, hora, minuto, segundo o milisegundo. A continuación, especifique la fecha de inicio en el parámetro de fecha de inicio y la fecha de finalización en el parámetro de fecha de finalización para la que desea encontrar la diferencia.

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
```

SalesOrderID	Tiempo de procesamiento
43659	7
43660	7
43661	7

SalesOrderID	Tiempo de procesamiento
43662	7

La función `DATEADD` permite agregar un intervalo a parte de una fecha específica.

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays
```

Added20MoreDays
2017-02-03 00: 00: 00.000

Configuración y función de conversión

Un ejemplo de una función de configuración en SQL es la función `@@SERVERNAME` . Esta función proporciona el nombre del servidor local que ejecuta SQL.

```
SELECT @@SERVERNAME AS 'Server'
```

Servidor
SQL064

En SQL, la mayoría de las conversiones de datos ocurren implícitamente, sin la intervención del usuario.

Para realizar cualquier conversión que no se pueda completar implícitamente, puede usar las funciones `CAST` o `CONVERT` .

La sintaxis de la función `CAST` es más simple que la sintaxis de la función `CONVERT` , pero está limitada en lo que puede hacer.

Aquí, usamos las funciones `CAST` y `CONVERT` para convertir el tipo de datos datetime al tipo de datos `varchar` .

La función `CAST` siempre utiliza la configuración de estilo predeterminada. Por ejemplo, representará fechas y horas utilizando el formato AAAA-MM-DD.

La función `CONVERT` utiliza el estilo de fecha y hora que especifique. En este caso, 3 especifica el formato de fecha dd / mm / aa.

```
USE AdventureWorks2012
GO
SELECT FirstName + ' ' + LastName + ' was hired on ' +
       CAST(HireDate AS varchar(20)) AS 'Cast',
       FirstName + ' ' + LastName + ' was hired on ' +
       CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM Person.Person AS p
```

```
JOIN HumanResources.Employee AS e
ON p.BusinessEntityID = e.BusinessEntityID
GO
```

Emitir	Convertir
David Hamiltion fue contratado el 2003-02-04	David Hamiltion fue contratado el 04/02/03

Otro ejemplo de una función de conversión es la función `PARSE` . Esta función convierte una cadena a un tipo de datos especificado.

En la sintaxis de la función, especifique la cadena que se debe convertir, la palabra clave `AS` y luego el tipo de datos requerido. Opcionalmente, también puede especificar la cultura en la que se debe formatear el valor de la cadena. Si no especifica esto, se usa el idioma de la sesión.

Si el valor de la cadena no se puede convertir a un formato numérico, de fecha o de hora, se producirá un error. A continuación, deberá utilizar `CAST` o `CONVERT` para la conversión.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'
```

Fecha en ingles
2012-08-13 00: 00: 00.0000000

Función lógica y matemática

SQL tiene dos funciones lógicas: `CHOOSE` y `IIF` .

La función `CHOOSE` devuelve un elemento de una lista de valores, según su posición en la lista. Esta posición está especificada por el índice.

En la sintaxis, el parámetro de índice especifica el elemento y es un número entero o entero. El parámetro val_1 ... val_n identifica la lista de valores.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;
```

Resultado
Ventas

En este ejemplo, utiliza la función `CHOOSE` para devolver la segunda entrada en una lista de departamentos.

La función `IIF` devuelve uno de los dos valores, en función de una condición particular. Si la condición es verdadera, devolverá el valor verdadero. De lo contrario, devolverá un valor falso.

En la sintaxis, el parámetro boolean_expression especifica la expresión booleana. El parámetro true_value especifica el valor que debe devolverse si la boolean_expression se evalúa como verdadera y el parámetro false_value especifica el valor que debe devolverse si la boolean_expression se evalúa como falsa.

```
SELECT BusinessEntityID, SalesYTD,
       IIF(SalesYTD > 200000, 'Bonus', 'No Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO
```

BusinessEntityID	VentasYTD	¿Prima?
274	559697.5639	Prima
275	3763178.1787	Prima
285	172524.4512	No Bonus

En este ejemplo, utiliza la función IIF para devolver uno de los dos valores. Si las ventas de un vendedor hasta la fecha son superiores a 200,000, esta persona será elegible para recibir un bono. Los valores inferiores a 200,000 significan que los empleados no califican para los bonos.

SQL incluye varias funciones matemáticas que puede utilizar para realizar cálculos en valores de entrada y devolver resultados numéricos.

Un ejemplo es la función `SIGN` , que devuelve un valor que indica el signo de una expresión. El valor de -1 indica una expresión negativa, el valor de +1 indica una expresión positiva y 0 indica cero.

```
SELECT SIGN(-20) AS 'Sign'
```

Firmar
-1

En el ejemplo, la entrada es un número negativo, por lo que el panel Resultados muestra el resultado -1.

Otra función matemática es la función de `POWER` . Esta función proporciona el valor de una expresión elevada a una potencia específica.

En la sintaxis, el parámetro float_expression especifica la expresión, y el parámetro y especifica la potencia a la que desea elevar la expresión.

```
SELECT POWER(50, 3) AS Result
```

Resultado
125000

Lea Funciones (escalar / fila única) en línea: https://riptutorial.com/es/sql/topic/6898/funciones--escalar---fila-unica-

Capítulo 31: Funciones de cadena

Introducción

Las funciones de cadena realizan operaciones en valores de cadena y devuelven valores numéricos o de cadena.

Mediante las funciones de cadena, puede, por ejemplo, combinar datos, extraer una subcadena, comparar cadenas o convertir una cadena a todos los caracteres en mayúscula o minúscula.

Sintaxis

- CONCAT (string_value1, string_value2 [, string_valueN])
- LTRIM (expresión_caracteres)
- RTRIM (expresión de caracteres)
- SUBSTRING (expresión, inicio, longitud)
- ASCII (expresión de caracteres)
- REPLICATE (string_expression, integer_expression)
- REVERSE (string_expression)
- SUPERIOR (expresión de caracteres)
- TRIM (cadena [caracteres de])
- STRING_SPLIT (cadena, separador)
- MATERIAL (expresión de caracteres, inicio, longitud, reemplazar con expresión)
- REPLACE (string_expression, string_pattern, string_replacement)

Observaciones

Referencia de funciones de cadena para Transact-SQL / Microsoft

Referencia de funciones de cadena para MySQL

Referencia de funciones de cadena para PostgreSQL

Examples

Recortar espacios vacíos

El recorte se utiliza para eliminar el espacio de escritura al principio o al final de la selección

En MSSQL no hay un solo `TRIM()`

```
SELECT LTRIM('  Hello  ') --returns 'Hello  '
SELECT RTRIM('  Hello  ') --returns '  Hello'
SELECT LTRIM(RTRIM('  Hello  ')) --returns 'Hello'
```

MySql y Oracle

```
SELECT TRIM('  Hello  ') --returns 'Hello'
```

Concatenar

En SQL (ANSI / ISO estándar), el operador para la concatenación de cadenas es `||` . Esta sintaxis es compatible con todas las bases de datos principales, excepto SQL Server:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

Muchas bases de datos admiten una función `CONCAT` para unir cadenas:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Algunas bases de datos admiten el uso de `CONCAT` para unir más de dos cadenas (Oracle no lo hace):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

En algunas bases de datos, los tipos sin cadena deben ser convertidos o convertidos:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Algunas bases de datos (por ejemplo, Oracle) realizan conversiones sin pérdida implícitas. Por ejemplo, un `CONCAT` en un `CLOB` y `NCLOB` produce un `NCLOB` . Un `CONCAT` en un número y un `varchar2` da como resultado un `varchar2` , etc .:

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Algunas bases de datos pueden usar el operador `+` no estándar (pero en la mayoría, `+` solo funciona con números):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

En SQL Server <2012, donde no se admite `CONCAT` , `+` es la única forma de unir cadenas.

Mayúsculas y minúsculas

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

Subcadena

La sintaxis es: `SUBSTRING ( string_expression, start, length )` . Tenga en cuenta que las cadenas SQL son 1-indexadas.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

Esto se usa a menudo junto con la función `LEN()` para obtener los últimos `n` caracteres de una cadena de longitud desconocida.

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --returns 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --returns 'Baz'
```

División

Divide una expresión de cadena usando un separador de caracteres. Tenga en cuenta que `STRING_SPLIT()` es una función con valores de tabla.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Resultado:

```
value
-----
Lorem
ipsum
dolor
sit
amet.
```

Cosas

Rellena una cadena en otra, reemplazando 0 o más caracteres en una posición determinada.

Nota: la posición `start` es 1-indexada (comienza la indexación en 1, no en 0).

Sintaxis:

```
STUFF ( character_expression , start , length , replaceWith_expression )
```

Ejemplo:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

Longitud

servidor SQL

El LEN no cuenta el espacio al final.

```
SELECT LEN('Hello') -- returns 5
```

```
SELECT LEN('Hello '); -- returns 5
```

El DATALENGTH cuenta el espacio al final.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

Sin embargo, se debe tener en cuenta que DATALENGTH devuelve la longitud de la representación de bytes subyacente de la cadena, que depende, entre otras cosas, del conjunto de caracteres utilizado para almacenar la cadena.

```
DECLARE @str varchar(100) = 'Hello ' --varchar is usually an ASCII string, occupying 1 byte
per char
SELECT DATALENGTH(@str) -- returns 6

DECLARE @nstr nvarchar(100) = 'Hello ' --nvarchar is a unicode string, occupying 2 bytes per
char
SELECT DATALENGTH(@nstr) -- returns 12
```

Oráculo

Sintaxis: Longitud (char)

Ejemplos:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

Ver también: LengthB, LengthC, Length2, Length4

Reemplazar

Sintaxis:

`REPLACE(` Cadena para buscar , Cadena para buscar y reemplazar , Cadena para colocar en la cadena original `)`

Ejemplo:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

IZQUIERDA DERECHA

La sintaxis es:
IZQUIERDA (expresión-cadena, entero)
DERECHA (string-expresión, entero)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL no tiene funciones IZQUIERDA y DERECHA. Se pueden emular con SUBSTR y LONGITUD.
SUBSTR (string-expresión, 1, entero)
SUBSTR (expresión-serie, longitud (expresión-serie) -integer + 1, entero)

```
SELECT SUBSTR('Hello',1,2)  --return He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --return lo
```

MARCHA ATRÁS

La sintaxis es: REVERSE (expresión-cadena)

```
SELECT REVERSE('Hello') --returns olleH
```

REPRODUCIR EXACTAMENTE

La función `REPLICATE` concatena una cadena consigo misma un número especificado de veces.

La sintaxis es: REPLICATE (expresión-cadena, entero)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

REGEXP

MySQL 3.19

Comprueba si una cadena coincide con una expresión regular (definida por otra cadena).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True

SELECT 'beam' REGEXP '[a-f]' -- returns False
```

Reemplazar la función en SQL Seleccionar y actualizar consulta

La función Reemplazar en SQL se usa para actualizar el contenido de una cadena. La llamada a la función es REPLACE () para MySQL, Oracle y SQL Server.

La sintaxis de la función Reemplazar es:

```
REPLACE (str, find, repl)
```

El siguiente ejemplo reemplaza las ocurrencias de `South` con `Southern` en la tabla Empleados:

Nombre de pila	Dirección
James	Sur de nueva york
Juan	Sur de boston
Miguel	Sur de san diego

Seleccione Declaración:

Si aplicamos la siguiente función Reemplazar:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Resultado:

Nombre de pila	Dirección
James	Sur de nueva york
Juan	Boston del sur
Miguel	Sur de san diego

Declaración de actualización:

Podemos usar una función de reemplazo para realizar cambios permanentes en nuestra tabla a través del siguiente enfoque.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

Un enfoque más común es usar esto junto con una cláusula WHERE como esta:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

Nombre de pila

BASE DE DATOS : SQL Server

La función **PARSENAME** devuelve la parte específica de la cadena dada (nombre del objeto). el nombre del objeto puede contener una cadena como el nombre del objeto, el nombre del propietario, el nombre de la base de datos y el nombre del servidor.

Más detalles MSDN: PARSENAME

Sintaxis

```
PARSENAME('NameOfStringToParse',PartIndex)
```

Ejemplo

Para obtener el nombre del objeto use el índice de parte 1

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1)  // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1)     // returns `Student`
```

Para obtener el nombre del esquema, use el índice de pieza 2

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2)  // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2)     // returns `school`
```

Para obtener el nombre de la base de datos utilice el índice de pieza 3

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3)    // returns `SchoolDatabase`
```

Para obtener el nombre del servidor use el índice de parte 4

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4)  // returns `ServerName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4)     // returns `[1012-1111]`
```

PARSENAME devolverá nulo si la parte especificada no está presente en la cadena de nombre de objeto dada

INSTR

Devuelve el índice de la primera aparición de una subcadena (cero si no se encuentra)

Sintaxis: INSTR (cadena, subcadena)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

Lea Funciones de cadena en línea: https://riptutorial.com/es/sql/topic/1120/funciones-de-cadena

Capítulo 32: Funciones de ventana

Examples

Sumando las filas totales seleccionadas a cada fila

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

carné de identidad	nombre	Ttl_Rows
1	ejemplo	5
2	foo	5
3	bar	5
4	baz	5
5	quux	5

En lugar de usar dos consultas para obtener un recuento y luego la línea, puede usar un agregado como función de ventana y usar el conjunto de resultados completo como ventana. Esto se puede usar como base para cálculos adicionales sin la complejidad de uniones adicionales.

Configuración de una bandera si otras filas tienen una propiedad común

Digamos que tengo estos datos:

Articulos de mesa

carné de identidad	nombre	etiqueta
1	ejemplo	unique_tag
2	foo	sencillo
42	bar	sencillo
3	baz	Hola
51	quux	mundo

Me gustaría obtener todas esas líneas y saber si una etiqueta es utilizada por otras líneas

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

El resultado será:

carné de identidad	nombre	etiqueta	bandera
1	ejemplo	unique_tag	falso
2	foo	sencillo	cierto
42	bar	sencillo	cierto
3	baz	Hola	falso
51	quux	mundo	falso

En caso de que su base de datos no tenga OVER y PARTITION, puede usar esto para producir el mismo resultado:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM
items A
```

Obtener un total acumulado

Teniendo en cuenta estos datos:

fecha	cantidad
2016-03-12	200
2016-03-11	-50
2016-03-14	100
2016-03-15	100
2016-03-10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

Te regalaré

fecha	cantidad	corriendo
2016-03-10	-250	-250
2016-03-11	-50	-300
2016-03-12	200	-100

fecha	cantidad	corriendo
2016-03-14	100	0
2016-03-15	100	-100

Obtención de las N filas más recientes sobre agrupación múltiple

Dados estos datos

ID_usuario	Fecha de Terminación
1	2016-07-20
1	2016-07-21
2	2016-07-20
2	2016-07-21
2	2016-07-22

```
;with CTE as
(SELECT *,
        ROW_NUMBER() OVER (PARTITION BY User_ID
                           ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FORM CTE WHERE Row_Num <= n
```

Usando n = 1, obtendrás la fila más reciente por `user_id` :

ID_usuario	Fecha de Terminación	Row_Num
1	2016-07-21	1
2	2016-07-22	1

Búsqueda de registros "fuera de secuencia" mediante la función LAG ()

Dados estos datos de muestra:

CARNÉ DE IDENTIDAD	ESTADO	STATUS_TIME	STATUS_BY
1	UNO	2016-09-28-19.47.52.501398	USER_1
3	UNO	2016-09-28-19.47.52.501511	USER_2
1	TRES	2016-09-28-19.47.52.501517	USER_3

CARNÉ DE IDENTIDAD	ESTADO	STATUS_TIME	STATUS_BY
3	DOS	2016-09-28-19.47.52.501521	USER_2
3	TRES	2016-09-28-19.47.52.501524	USER_4

Los elementos identificados por los valores de `ID` deben pasar de `STATUS` "UNO" a "DOS" a "TRES" en secuencia, sin saltarse estados. El problema es encontrar los valores de los usuarios (`STATUS_BY`) que infringen la regla y pasar de "UNO" inmediatamente a "TRES".

La función analítica `LAG()` ayuda a resolver el problema devolviendo a cada fila el valor en la fila anterior:

```
SELECT * FROM (
 SELECT
  t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
  FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

En caso de que su base de datos no tenga LAG (), puede usar esto para producir el mismo resultado:

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as
prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND   B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and
id = A.id)
AND   A.status = 'THREE' AND NOT B.status = 'TWO'
```

Lea Funciones de ventana en línea: https://riptutorial.com/es/sql/topic/647/funciones-de-ventana

Capítulo 33: Gatillos

Examples

Crear gatillo

Este ejemplo crea un activador que inserta un registro en una segunda tabla (MyAudit) después de insertar un registro en la tabla en la que se define el activador (MyTable). Aquí, la tabla "insertada" es una tabla especial que usa Microsoft SQL Server para almacenar las filas afectadas durante las instrucciones INSERT y UPDATE; también hay una tabla especial "eliminada" que realiza la misma función para las sentencias DELETE.

```
CREATE TRIGGER MyTrigger
    ON MyTable
    AFTER INSERT

AS

BEGIN
    -- insert audit record to MyAudit table
    INSERT INTO MyAudit(MyTableId, User)
    (SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Utilice Trigger para administrar una "Papelera de reciclaje" para los elementos eliminados

```
CREATE TRIGGER BooksDeleteTrigger
    ON MyBooksDB.Books
    AFTER DELETE
AS
  INSERT INTO BooksRecycleBin
    SELECT *
    FROM deleted;
GO
```

Lea Gatillos en línea: https://riptutorial.com/es/sql/topic/1432/gatillos

Capítulo 34: Grupo SQL por vs distinto

Examples

Diferencia entre GROUP BY y DISTINCT

`GROUP BY` se utiliza en combinación con funciones de agregación. Considere la siguiente tabla:

Solicitar ID	ID de usuario	nombre de la tienda	valor del pedido	fecha de orden
1	43	Tienda A	25	20-03-2016
2	57	Tienda B	50	22-03-2016
3	43	Tienda A	30	25-03-2016
4	82	Tienda c	10	26-03-2016
5	21	Tienda A	45	29-03-2016

La siguiente consulta utiliza `GROUP BY` para realizar cálculos agregados.

```
SELECT
    storeName,
    COUNT(*) AS total_nr_orders,
    COUNT(DISTINCT userId) AS nr_unique_customers,
    AVG(orderValue) AS average_order_value,
    MIN(orderDate) AS first_order,
    MAX(orderDate) AS lastOrder
FROM
    orders
GROUP BY
    storeName;
```

y devolverá la siguiente información

nombre de la tienda	total_nr_orders	nr_unique_customers	valor_ordenamiento_medio	primer orden	último pedido
Tienda A	3	2	33.3	20-03-2016	29-03-2016
Tienda B	1	1	50	22-03-2016	22-03-2016
Tienda c	1	1	10	26-03-2016	26-03-2016

Mientras que `DISTINCT` se usa para enumerar una combinación única de valores distintos para las columnas especificadas.

```
SELECT DISTINCT
    storeName,
    userId
FROM
    orders;
```

nombre de la tienda	ID de usuario
Tienda A	43
Tienda B	57
Tienda c	82
Tienda A	21

Lea Grupo SQL por vs distinto en línea: https://riptutorial.com/es/sql/topic/2499/grupo-sql-por-vs-distinto

Capítulo 35: Identificador

Introducción

Este tema trata sobre identificadores, es decir, reglas de sintaxis para nombres de tablas, columnas y otros objetos de base de datos.

Cuando sea apropiado, los ejemplos deben cubrir las variaciones utilizadas por diferentes implementaciones de SQL, o identificar la implementación de SQL del ejemplo.

Examples

Identificadores sin comillas

Los identificadores sin comillas pueden usar letras (`a` - `z`), dígitos (`0` - `9`) y guiones bajos (`_`), y deben comenzar con una letra.

Dependiendo de la implementación de SQL y / o la configuración de la base de datos, se pueden permitir otros caracteres, algunos incluso como el primer carácter, por ejemplo

- MS SQL: `@` , `$` , `#` y otras letras Unicode *(fuente)*
- MySQL: `$` *(fuente)*
- Oracle: `$` , `#` y otras letras del conjunto de caracteres de la base de datos *(fuente)*
- PostgreSQL: `$` , y otras letras Unicode *(fuente)*

Los identificadores no citados no distinguen entre mayúsculas y minúsculas. Cómo se maneja esto depende en gran medida de la implementación de SQL:

- MS SQL: preservación de mayúsculas y minúsculas, sensibilidad definida por el conjunto de caracteres de la base de datos, por lo que puede ser sensible a mayúsculas y minúsculas.
- MySQL: preservación de mayúsculas y minúsculas, la sensibilidad depende de la configuración de la base de datos y del sistema de archivos subyacente.
- Oracle: convertido a mayúsculas, luego manejado como identificador entre comillas.
- PostgreSQL: Convertido a minúsculas, luego manejado como identificador entre comillas.
- SQLite: Caso-preservación; Insensibilidad de mayúsculas y minúsculas solo para caracteres ASCII.

Lea Identificador en línea: https://riptutorial.com/es/sql/topic/9677/identificador

Capítulo 36: Índices

Introducción

Los índices son una estructura de datos que contiene punteros a los contenidos de una tabla organizada en un orden específico, para ayudar a la base de datos a optimizar las consultas. Son similares al índice del libro, donde las páginas (filas de la tabla) se indexan por su número de página.

Existen varios tipos de índices y se pueden crear en una tabla. Cuando existe un índice en las columnas utilizadas en la cláusula WHERE de una consulta, la cláusula JOIN o la cláusula ORDER BY, puede mejorar sustancialmente el rendimiento de la consulta.

Observaciones

Los índices son una forma de acelerar las consultas de lectura ordenando las filas de una tabla según una columna.

El efecto de un índice no es notable para bases de datos pequeñas como el ejemplo, pero si hay un gran número de filas, puede mejorar considerablemente el rendimiento. En lugar de revisar cada fila de la tabla, el servidor puede hacer una búsqueda binaria en el índice.

La compensación para crear un índice es la velocidad de escritura y el tamaño de la base de datos. El almacenamiento del índice ocupa espacio. Además, cada vez que se realiza un INSERT o se actualiza la columna, el índice debe actualizarse. Esta no es una operación tan costosa como escanear la tabla completa en una consulta SELECT, pero aún es algo a tener en cuenta.

Examples

Creando un índice

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

Esto creará un índice para la columna *EmployeeId* en la tabla *Cars* . Este índice mejorará la velocidad de las consultas que solicitan al servidor que ordene o seleccione los valores en *EmployeeId* , como los siguientes:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

El índice puede contener más de 1 columna, como en el siguiente;

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

En este caso, el índice sería útil para consultas que soliciten ordenar o seleccionar por todas las

columnas incluidas, si el conjunto de condiciones se ordena de la misma manera. Eso significa que al recuperar los datos, puede encontrar las filas para recuperar usando el índice, en lugar de mirar a través de la tabla completa.

Por ejemplo, el siguiente caso utilizaría el segundo índice;

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

Sin embargo, si el orden difiere, el índice no tiene las mismas ventajas, como en el siguiente;

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

El índice no es tan útil porque la base de datos debe recuperar el índice completo, en todos los valores de EmployeeId y CarID, para encontrar qué elementos tienen `OwnerId = 17` .

(El índice puede seguir utilizándose; puede darse el caso de que el optimizador de consultas encuentre que recuperar el índice y filtrar en el `OwnerId` , luego recuperar solo las filas necesarias es más rápido que recuperar la tabla completa, especialmente si la tabla es grande).

Índices agrupados, únicos y ordenados

Los índices pueden tener varias características que se pueden establecer en la creación o alterando los índices existentes.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

La declaración SQL anterior crea un nuevo índice agrupado en los empleados. Los índices agrupados son índices que dictan la estructura real de la tabla; La tabla en sí está ordenada para que coincida con la estructura del índice. Eso significa que puede haber como máximo un índice agrupado en una tabla. Si ya existe un índice agrupado en la tabla, la declaración anterior fallará. (Las tablas sin índices agrupados también se denominan montones).

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

Esto creará un índice único para la columna *Correo electrónico* en la tabla *Clientes* . Este índice, junto con la aceleración de consultas como un índice normal, también forzará que cada dirección de correo electrónico en esa columna sea única. Si se inserta o actualiza una fila con un valor de *correo electrónico* no único, la inserción o actualización fallará, de forma predeterminada.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

Esto crea un índice en los clientes que también crea una restricción de tabla que indica que el EmployeeID debe ser único. (Esto fallará si la columna no es actualmente única; en este caso, si hay empleados que comparten una ID).

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

Esto crea un índice que se ordena en orden descendente. De forma predeterminada, los índices (en el servidor MSSQL, al menos) son ascendentes, pero se pueden cambiar.

Inserción con un índice único

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

Esto fallará si se establece un índice único en la columna *Correo electrónico* de *Clientes* . Sin embargo, se puede definir un comportamiento alternativo para este caso:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

SAP ASE: índice de caída

Este comando caerá índice en la tabla. Funciona en servidor `SAP ASE` .

Sintaxis:

```
DROP INDEX [table name].[index name]
```

Ejemplo:

```
DROP INDEX Cars.index_1
```

Índice ordenado

Si usa un índice que está ordenado de la forma en que lo recuperaría, la instrucción `SELECT` no haría una clasificación adicional cuando estaba en recuperación.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

Cuando ejecutas la consulta

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

El sistema de base de datos no haría una clasificación adicional, ya que puede hacer una búsqueda de índice en ese orden.

Bajar un índice, o desactivarlo y reconstruirlo

```
DROP INDEX ix_cars_employee_id ON Cars;
```

Podemos usar el comando `DROP` para borrar nuestro índice. En este ejemplo vamos a `DROP` el índice denominado *ix_cars_employee_id* en los *coches* de mesa.

Esto elimina el índice por completo, y si el índice está agrupado, eliminará cualquier

agrupamiento. No se puede reconstruir sin volver a crear el índice, que puede ser lento y costoso computacionalmente. Como alternativa, el índice se puede desactivar:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

Esto permite que la tabla retenga la estructura, junto con los metadatos sobre el índice.

Críticamente, esto retiene las estadísticas del índice, de modo que es posible evaluar fácilmente el cambio. Si se justifica, el índice puede luego reconstruirse, en lugar de recrearse completamente;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Índice único que permite NULLS

```
CREATE UNIQUE INDEX idx_license_id
   ON Person(DrivingLicenseID) WHERE DrivingLicenseID IS NOT NULL
GO
```

Este esquema permite una relación 0..1: las personas pueden tener cero o un permiso de conducir y cada licencia solo puede pertenecer a una persona

Índice de reconstrucción

A lo largo del tiempo, los índices B-Tree pueden fragmentarse debido a la actualización / eliminación / inserción de datos. En la terminología de SQLServer podemos tener interno (página de índice que está medio vacía) y externo (el orden lógico de la página no corresponde al orden físico). La reconstrucción del índice es muy similar a soltarlo y recrearlo.

Podemos reconstruir un índice con

```
ALTER INDEX index_name REBUILD;
```

Por defecto, el índice de reconstrucción es una operación fuera de línea que bloquea la tabla y evita el uso de DML, pero muchos RDBMS permiten la reconstrucción en línea. Además, algunos proveedores de bases de datos ofrecen alternativas a la reconstrucción de índices como `REORGANIZE` (SQLServer) o `COALESCE` / `SHRINK SPACE` (Oracle).

Índice agrupado

Cuando se utiliza el índice agrupado, las filas de la tabla se ordenan por la columna a la que se aplica el índice agrupado. Por lo tanto, solo puede haber un índice agrupado en la tabla porque no puede ordenar la tabla por dos columnas diferentes.

En general, es mejor usar un índice agrupado cuando se realizan lecturas en tablas de big data. La parte negativa del índice agrupado es cuando se escribe en la tabla y los datos se deben reorganizar (reordenar).

Un ejemplo de creación de un índice agrupado en una tabla Empleados en la columna Empleado_ Apellido:

```
CREATE CLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Índice no agrupado

Los índices no agrupados se almacenan por separado de la tabla. Cada índice en esta estructura contiene un puntero a la fila en la tabla que representa.

Estos punteros se llaman localizadores de fila. La estructura del localizador de filas depende de si las páginas de datos se almacenan en un montón o en una tabla agrupada. Para un montón, un localizador de fila es un puntero a la fila. Para una tabla agrupada, el localizador de filas es la clave de índice agrupado.

Un ejemplo de creación de un índice no agrupado en la tabla Empleados y columna Empleado_ Apellido:

```
CREATE NONCLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Puede haber múltiples índices no agrupados en la tabla. Las operaciones de lectura son generalmente más lentas con índices no agrupados que con índices agrupados, ya que tiene que ir primero al índice y luego a la tabla. Sin embargo, no hay restricciones en las operaciones de escritura.

Indice parcial o filtrado

SQL Server y SQLite permiten crear índices que contienen no solo un subconjunto de columnas, sino también un subconjunto de filas.

Considere una cantidad creciente de pedidos con `order_state_id` igual a finalizado (2), y una cantidad estable de pedidos con `order_state_id equal` a iniciado (1).

Si su empresa utiliza consultas como esta:

```
SELECT id, comment
  FROM orders
 WHERE order_state_id =  1
   AND product_id = @some_value;
```

La indexación parcial le permite limitar el índice, incluidos solo los pedidos no finalizados:

```
CREATE INDEX Started_Orders
          ON orders(product_id)
       WHERE order_state_id = 1;
```

Este índice será más pequeño que un índice sin filtrar, lo que ahorra espacio y reduce el costo de actualización del índice.

Lea Índices en línea: https://riptutorial.com/es/sql/topic/344/indices

Capítulo 37: INSERTAR

Sintaxis

- INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
- INSERT INTO table_name (column1, column2 ...) SELECT value1, value2 ... from other_table

Examples

Insertar nueva fila

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

Esta declaración insertará una nueva fila en la tabla `Customers` . Tenga en cuenta que no se especificó un valor para la columna `Id` , ya que se agregará automáticamente. Sin embargo, todos los demás valores de columna deben ser especificados.

Insertar solo columnas especificadas

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

Esta declaración insertará una nueva fila en la tabla `Customers` . Los datos solo se insertarán en las columnas especificadas; tenga en cuenta que no se proporcionó ningún valor para la columna `PhoneNumber` . Tenga en cuenta, sin embargo, que todas las columnas marcadas como `not null` deben incluirse.

INSERTAR datos de otra tabla utilizando SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

Este ejemplo insertará a todos los empleados en la tabla Clientes . Dado que las dos tablas tienen diferentes campos y no desea mover todos los campos, debe establecer qué campos insertar y qué campos seleccionar. Los nombres de los campos correlacionados no necesitan llamarse la misma cosa, pero luego deben ser del mismo tipo de datos. Este ejemplo asume que el campo Id tiene un conjunto de especificaciones de identidad y se incrementará automáticamente.

Si tiene dos tablas que tienen exactamente los mismos nombres de campo y solo desea mover todos los registros, puede usar:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Insertar múltiples filas a la vez

Se pueden insertar varias filas con un solo comando de inserción:

```
INSERT INTO tbl_name (field1, field2, field3)

VALUES (1,2,3), (4,5,6), (7,8,9);
```

Para insertar grandes cantidades de datos (inserciones masivas) al mismo tiempo, existen recomendaciones y características específicas de DBMS.

MySQL - CARGAR DATOS DE CARGA

MSSQL - INSERTO A GRANEL

Lea INSERTAR en línea: https://riptutorial.com/es/sql/topic/465/insertar

Capítulo 38: Inyección SQL

Introducción

La inyección de SQL es un intento de acceder a las tablas de la base de datos de un sitio web mediante la inyección de SQL en un campo de formulario. Si un servidor web no protege contra los ataques de inyección de SQL, un pirata informático puede engañar a la base de datos para que ejecute el código SQL adicional. Al ejecutar su propio código SQL, los hackers pueden actualizar el acceso a su cuenta, ver la información privada de otra persona o realizar cualquier otra modificación en la base de datos.

Examples

Muestra de inyección SQL

Suponiendo que la llamada al manejador de inicio de sesión de su aplicación web se vea así:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Ahora en login.ashx, lees estos valores:

```
strUserName = getHttpsRequestParameterString("username");
strPassword = getHttpsRequestParameterString("password");
```

y consulte su base de datos para determinar si existe un usuario con esa contraseña.

Entonces construyes una cadena de consulta SQL:

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '"+
strPassword +"'";
```

Esto funcionará si el nombre de usuario y la contraseña no contienen una cotización.

Sin embargo, si uno de los parámetros contiene una cita, el SQL que se envía a la base de datos se verá así:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

Esto dará lugar a un error de sintaxis, porque la cita después de la `d` en `d'Alambert` termina la cadena SQL.

Puede corregir esto evitando comillas en el nombre de usuario y la contraseña, por ejemplo:

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

Sin embargo, es más apropiado usar parámetros:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

Si no usa parámetros y se olvida de reemplazar comillas incluso en uno de los valores, un usuario malintencionado (también conocido como hacker) puede usar esto para ejecutar comandos SQL en su base de datos.

Por ejemplo, si un atacante es malo, él / ella establecerá la contraseña en

```
lol'; DROP DATABASE master; --
```

y luego el SQL se verá así:

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --'";
```

Desafortunadamente para usted, este es un SQL válido, y el DB lo ejecutará!

Este tipo de exploit se llama inyección SQL.

Hay muchas otras cosas que un usuario malintencionado podría hacer, como robar la dirección de correo electrónico de cada usuario, robar la contraseña de todos, robar números de tarjetas de crédito, robar cualquier cantidad de datos en su base de datos, etc.

Por eso siempre necesitas escapar de tus cuerdas.
Y el hecho de que siempre se olvidará de hacerlo tarde o temprano es exactamente la razón por la que debe usar parámetros. Porque si usa parámetros, entonces el marco de su lenguaje de programación hará cualquier escape necesario para usted.

muestra de inyección simple

Si la sentencia de SQL se construye así:

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password ='" + pw + "'";
db.execute(SQL);
```

Luego, un pirata informático podría recuperar sus datos con una contraseña como `pw' or '1'='1` ;
La sentencia SQL resultante será:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

Este pasará la verificación de contraseña para todas las filas en la tabla de `Users` porque `'1'='1'` siempre es verdadero.

Para evitar esto, use los parámetros de SQL:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

Lea Inyección SQL en línea: https://riptutorial.com/es/sql/topic/3517/inyeccion-sql

Capítulo 39: LA CLÁUSULA EXISTA

Examples

LA CLÁUSULA EXISTA

Tabla de clientes

Carné de identidad	Nombre de pila	Apellido
1	Ozgur	Ozturk
2	Youssef	Medi
3	Enrique	Tai

Tabla de orden

Carné de identidad	Identificación del cliente	Cantidad
1	2	123.50
2	3	14.80

Obtener todos los clientes con al menos un pedido

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

Resultado

Carné de identidad	Nombre de pila	Apellido
2	Youssef	Medi
3	Enrique	Tai

Obtener todos los clientes sin orden

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Resultado

Carné de identidad	Nombre de pila	Apellido
1	Ozgur	Ozturk

Propósito

`EXISTS` , `IN` y `JOIN` podrían usarse en algún momento para el mismo resultado, sin embargo, no son iguales:

- `EXISTS` debe usarse para verificar si existe un valor en otra tabla
- `IN` debe utilizarse para la lista estática
- Se debe usar `JOIN` para recuperar datos de otra (s) tabla (s)

Lea LA CLÁUSULA EXISTA en línea: https://riptutorial.com/es/sql/topic/7933/la-clausula-exista

Capítulo 40: Limpiar código en SQL

Introducción

Cómo escribir consultas SQL buenas y legibles, y ejemplos de buenas prácticas.

Examples

Formato y ortografía de palabras clave y nombres

Nombres de tabla / columna

Dos formas comunes de formatear nombres de tablas / columnas son `CamelCase` y `snake_case` :

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;
```

```
SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

Los nombres deben describir lo que se almacena en su objeto. Esto implica que los nombres de las columnas usualmente deben ser singulares. Si los nombres de tablas deben usar singular o plural es una pregunta muy discutida , pero en la práctica, es más común usar nombres de tablas en plural.

Agregar prefijos o sufijos como `tbl` o `col` reduce la legibilidad, así que evítalos. Sin embargo, a veces se utilizan para evitar conflictos con las palabras clave de SQL y, a menudo, con desencadenantes e índices (cuyos nombres no suelen mencionarse en las consultas).

Palabras clave

Las palabras clave de SQL no distinguen entre mayúsculas y minúsculas. Sin embargo, es una práctica común escribirlos en mayúsculas.

SELECCIONAR *

`SELECT *` devuelve todas las columnas en el mismo orden en que están definidas en la tabla.

Cuando se utiliza `SELECT *` , los datos devueltos por una consulta pueden cambiar siempre que cambie la definición de la tabla. Esto aumenta el riesgo de que las diferentes versiones de su aplicación o su base de datos sean incompatibles entre sí.

Además, leer más columnas de las necesarias puede aumentar la cantidad de E / S de disco y de red.

Por lo tanto, siempre debe especificar explícitamente la (s) columna (s) que realmente desea recuperar:

```
--SELECT *                                  don't
  SELECT ID, FName, LName, PhoneNumber  -- do
  FROM Emplopees;
```

(Al hacer consultas interactivas, estas consideraciones no se aplican.)

Sin embargo, `SELECT *` no duele en la subconsulta de un operador EXISTS, porque EXISTS ignora los datos reales de todos modos (solo verifica si se ha encontrado al menos una fila). Por la misma razón, no tiene sentido enumerar ninguna columna específica para EXISTS, por lo que `SELECT *` realidad tiene más sentido:

```
-- list departments where nobody was hired recently
SELECT ID,
       Name
FROM Departments
WHERE NOT EXISTS (SELECT *
                  FROM Employees
                  WHERE DepartmentID = Departments.ID
                    AND HireDate >= '2015-01-01');
```

Sangría

No hay un estándar ampliamente aceptado. En lo que todos estamos de acuerdo es que apretar todo en una sola línea es malo:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID =
e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

Como mínimo, coloque cada cláusula en una nueva línea y divida las líneas si se alargarían demasiado, de lo contrario:

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM Departments AS d
JOIN Employees AS e ON d.ID = e.DepartmentID
WHERE d.Name != 'HR'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

A veces, todo después de la palabra clave SQL que introduce una cláusula está sangrado en la misma columna:

```
SELECT   d.Name,
         COUNT(*) AS Employees
```

```
FROM     Departments AS d
JOIN     Employees AS e ON d.ID = e.DepartmentID
WHERE    d.Name != 'HR'
HAVING   COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

(Esto también puede hacerse alineando las palabras clave de SQL a la derecha).

Otro estilo común es poner palabras clave importantes en sus propias líneas:

```
SELECT
    d.Name,
    COUNT(*) AS Employees
FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID
WHERE
    d.Name != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

La alineación vertical de múltiples expresiones similares mejora la legibilidad:

```
SELECT Model,
       EmployeeID
FROM Cars
WHERE CustomerID = 42
  AND Status     = 'READY';
```

El uso de varias líneas dificulta la incorporación de comandos SQL en otros lenguajes de programación. Sin embargo, muchos idiomas tienen un mecanismo para cadenas de varias líneas, por ejemplo, `@"..."` en C #, `"""..."""` en Python, o `R"(...)"` en C ++.

Se une

Siempre se deben utilizar combinaciones explícitas; Las uniones implícitas tienen varios problemas:

- La condición de unión está en algún lugar de la cláusula WHERE, mezclada con cualquier otra condición de filtro. Esto hace que sea más difícil ver qué tablas están unidas y cómo.
- Debido a lo anterior, existe un mayor riesgo de errores y es más probable que se detecten más adelante.
- En SQL estándar, las uniones explícitas son la única forma de usar uniones externas :

  ```
  SELECT d.Name,
  ```

```
       e.Fname || e.LName AS EmpName
FROM      Departments AS d
LEFT JOIN Employees   AS e ON d.ID = e.DepartmentID;
```

- Las combinaciones explícitas permiten el uso de la cláusula USING:

```
SELECT RecipeID,
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM      Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(Esto requiere que ambas tablas usen el mismo nombre de columna.
USAR elimina automáticamente la columna duplicada del resultado, por ejemplo, la unión en esta consulta devuelve una única columna de `RecipeID` .)

Lea Limpiar código en SQL en línea: https://riptutorial.com/es/sql/topic/9843/limpiar-codigo-en-sql

Capítulo 41: Llaves extranjeras

Examples

Creando una tabla con una clave externa

En este ejemplo tenemos una tabla existente, `SuperHeros` .

Esta tabla contiene un `ID` clave principal.

Agregaremos una nueva tabla para almacenar los poderes de cada superhéroe:

```
CREATE TABLE HeroPowers
(
    ID int NOT NULL PRIMARY KEY,
    Name nvarchar(MAX) NOT NULL,
    HeroId int REFERENCES SuperHeros(ID)
)
```

La columna `HeroId` es una **clave externa** a la tabla `SuperHeros` .

Claves foráneas explicadas

Las restricciones de claves externas garantizan la integridad de los datos, al imponer que los valores en una tabla deben coincidir con los valores en otra tabla.

Un ejemplo de dónde se requiere una clave extranjera es: En una universidad, un curso debe pertenecer a un departamento. El código para este escenario es:

```
CREATE TABLE Department (
    Dept_Code        CHAR (5)     PRIMARY KEY,
    Dept_Name        VARCHAR (20) UNIQUE
);
```

Insertar valores con la siguiente declaración:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

La siguiente tabla contendrá la información de los temas ofrecidos por la rama de informática:

```
CREATE TABLE Programming_Courses (
    Dept_Code       CHAR(5),
    Prg_Code        CHAR(9) PRIMARY KEY,
    Prg_Name        VARCHAR (50) UNIQUE,
    FOREIGN KEY (Dept_Code) References Department(Dept_Code)
);
```

(El tipo de datos de la clave externa debe coincidir con el tipo de datos de la clave de referencia).

La restricción de clave externa en la columna `Dept_Code` solo permite valores si ya existen en la tabla a la que se hace referencia, `Department` . Esto significa que si intenta insertar los siguientes valores:

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', 'Database Systems');
```

la base de datos generará un error de violación de clave externa, porque `CS300` no existe en la tabla de `Department` . Pero cuando intentas un valor clave que existe:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```

entonces la base de datos permite estos valores.

Algunos consejos para usar llaves extranjeras

- Una clave externa debe hacer referencia a una clave ÚNICA (o PRIMARIA) en la tabla principal.
- La introducción de un valor NULL en una columna de clave externa no genera un error.
- Las restricciones de clave externa pueden hacer referencia a tablas dentro de la misma base de datos.
- Las restricciones de clave externa pueden referirse a otra columna en la misma tabla (referencia propia).

Lea Llaves extranjeras en línea: https://riptutorial.com/es/sql/topic/1533/llaves-extranjeras

Capítulo 42: Llaves primarias

Sintaxis

- MySQL: CREATE TABLE Empleados (Id int NOT NULL, PRIMARY KEY (Id), ...);
- Otros: CREAR TABLA Empleados (Id int NO NULL PRIMARY KEY, ...);

Examples

Creación de una clave principal

```
CREATE TABLE Employees (
    Id int NOT NULL,
    PRIMARY KEY (Id),
    ...
);
```

Esto creará la tabla Empleados con 'Id' como su clave principal. La clave principal se puede utilizar para identificar de forma única las filas de una tabla. Solo se permite una clave principal por tabla.

Una clave también puede estar compuesta por uno o más campos, también llamados claves compuestas, con la siguiente sintaxis:

```
CREATE TABLE EMPLOYEE (
    e1_id INT,
    e2_id INT,
    PRIMARY KEY (e1_id, e2_id)
)
```

Usando Auto Incremento

Muchas bases de datos permiten que el valor de la clave principal se incremente automáticamente cuando se agrega una nueva clave. Esto asegura que cada clave sea diferente.

MySQL

```
CREATE TABLE Employees (
    Id int NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (Id)
);
```

PostgreSQL

```
CREATE TABLE Employees (
    Id SERIAL PRIMARY KEY
);
```

servidor SQL

```
CREATE TABLE Employees (
    Id int NOT NULL IDENTITY,
    PRIMARY KEY (Id)
);
```

SQLite

```
CREATE TABLE Employees (
    Id INTEGER PRIMARY KEY
);
```

Lea Llaves primarias en línea: https://riptutorial.com/es/sql/topic/505/llaves-primarias

Capítulo 43: Mesa plegable

Observaciones

DROP TABLE elimina la definición de la tabla del esquema junto con las filas, índices, permisos y activadores.

Examples

Gota simple

```
Drop Table MyTable;
```

Compruebe la existencia antes de dejar caer

MySQL 3.19

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL 8.x

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server 2005

```
If Exists(Select * From Information_Schema.Tables
          Where Table_Schema = 'dbo'
            And Table_Name = 'MyTable')
  Drop Table dbo.MyTable
```

SQLite 3.0

```
DROP TABLE IF EXISTS MyTable;
```

Lea Mesa plegable en línea: https://riptutorial.com/es/sql/topic/1832/mesa-plegable

Capítulo 44: NULO

Introducción

`NULL` en SQL, así como la programación en general, significa literalmente "nada". En SQL, es más fácil de entender como "la ausencia de cualquier valor".

Es importante distinguirlo de valores aparentemente vacíos, como la cadena vacía `''` o el número `0` , ninguno de los cuales es `NULL` .

También es importante tener cuidado de no incluir `NULL` entre comillas, como `'NULL'` , que se permite en las columnas que aceptan texto, pero no es `NULL` y puede causar errores y conjuntos de datos incorrectos.

Examples

Filtrado para NULL en consultas

La sintaxis para el filtrado de `NULL` (es decir, la ausencia de un valor) en los bloques `WHERE` es ligeramente diferente al filtrado de valores específicos.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

Tenga en cuenta que dado que `NULL` no es igual a nada, ni siquiera a sí mismo, usar operadores de igualdad `= NULL` o `<> NULL` (o `!= NULL`) siempre producirá el valor de verdad de `UNKNOWN` que será rechazado por `WHERE` .

`WHERE` filtra todas las filas en las que la condición es `FALSE` o `UKNOWN` y mantiene solo las filas en las que la condición es `TRUE` .

Columnas anulables en tablas

Al crear tablas, es posible declarar una columna como anulable o no anulable.

```
CREATE TABLE MyTable
(
    MyCol1 INT NOT NULL, -- non-nullable
    MyCol2 INT NULL      -- nullable
) ;
```

De forma predeterminada, todas las columnas (excepto aquellas en la restricción de clave principal) son anulables a menos que nosotros configuremos explícitamente la restricción `NOT NULL` .

Intentar asignar `NULL` a una columna no anulable resultará en un error.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ;  -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
        -- cannot insert
        -- the value NULL into column 'MyCol1', table 'MyTable';
        -- column does not allow nulls. INSERT fails.
```

Actualizando campos a NULL

Establecer un campo en `NULL` funciona exactamente igual que con cualquier otro valor:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Insertando filas con campos NULOS

Por ejemplo, insertar un empleado sin número de teléfono ni administrador en la tabla de ejemplo de Empleados :

```
INSERT INTO Employees
    (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

Lea NULO en línea: https://riptutorial.com/es/sql/topic/3421/nulo

Capítulo 45: Numero de fila

Sintaxis

- NUMERO DE FILA ( )
- OVER ([PARTITION BY value_expression, ... [n]] order_by_clause)

Examples

Números de fila sin particiones

Incluir un número de fila según el orden especificado.

```
SELECT
  ROW_NUMBER() OVER(ORDER BY Fname ASC) AS RowNumber,
  Fname,
  LName
FROM Employees
```

Números de fila con particiones

Utiliza un criterio de partición para agrupar la numeración de la fila de acuerdo con ella.

```
SELECT
  ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC) AS RowNumber,
  DepartmentId, Fname, LName
FROM Employees
```

Eliminar todo menos el último registro (1 a muchas tablas)

```
WITH cte AS (
  SELECT ProjectID,
         ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn
  FROM ProjectNotes
)
DELETE FROM cte WHERE rn > 1;
```

Lea Numero de fila en línea: https://riptutorial.com/es/sql/topic/1977/numero-de-fila

Capítulo 46: Operadores AND & OR

Sintaxis

1. SELECT * FROM table WHERE (condition1) AND (condition2);
2. SELECCIONAR * DESDE la tabla DONDE (condición1) O (condición2);

Examples

Y O Ejemplo

Tener una mesa

Nombre	Años	Ciudad
Mover	10	París
Estera	20	Berlina
María	24	Praga

```
select Name from table where Age>10 AND City='Prague'
```

Da

Nombre
María

```
select Name from table where Age=10 OR City='Prague'
```

Da

Nombre
Mover
María

Lea Operadores AND & OR en línea: https://riptutorial.com/es/sql/topic/1386/operadores-and--amp--or

Capítulo 47: Orden de Ejecución

Examples

Orden lógico de procesamiento de consultas en SQL

```
/*(8)*/  SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/  FROM
/*(3)*/         JOIN
/*(2)*/        ON
/*(4)*/  WHERE
/*(5)*/  GROUP BY
/*(6)*/  WITH {CUBE | ROLLUP}
/*(7)*/  HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

El orden en que se procesa una consulta y la descripción de cada sección.

VT significa "tabla virtual" y muestra cómo se producen diversos datos a medida que se procesa la consulta.

1. FROM: se realiza un producto cartesiano (combinación cruzada) entre las dos primeras tablas de la cláusula FROM y, como resultado, se genera la tabla virtual VT1.

2. ON: El filtro ON se aplica a VT1. Solo las filas para las cuales es VERDADERO se insertan en VT2.

3. OUTER (join): si se especifica OUTER JOIN (a diferencia de CROSS JOIN o INNER JOIN), las filas de la tabla o tablas conservadas para las que no se encontró una coincidencia se agregan a las filas de VT2 como filas externas, generando VT3. Si aparecen más de dos tablas en la cláusula FROM, los pasos 1 a 3 se aplican repetidamente entre el resultado de la última combinación y la siguiente tabla en la cláusula FROM hasta que se procesen todas las tablas.

4. DONDE: El filtro DONDE se aplica a VT3. Solo las filas para las cuales es VERDADERO se insertan en VT4.

5. GROUP BY: Las filas de VT4 se organizan en grupos según la lista de columnas especificada en la cláusula GROUP BY. Se genera VT5.

6. CUBO | ROLLUP: los supergrupos (grupos de grupos) se agregan a las filas de VT5, generando VT6.

7. HAVING: El filtro HAVING se aplica a VT6. Sólo los grupos para los cuales es TRUE se insertan en VT7.

8. SELECCIONAR: se procesa la lista SELECCIONAR, generando VT8.

9. DISTINTO: las filas duplicadas se eliminan de VT8. Se genera VT9.
10. ORDER BY: Las filas de VT9 se ordenan según la lista de columnas especificada en la cláusula ORDER BY. Se genera un cursor (VC10).
11. TOP: El número especificado o el porcentaje de filas se selecciona desde el principio de VC10. La tabla VT11 se genera y se devuelve a la persona que llama. LIMIT tiene la misma funcionalidad que TOP en algunos dialectos de SQL como Postgres y Netezza.

Lea Orden de Ejecución en línea: https://riptutorial.com/es/sql/topic/3671/orden-de-ejecucion

Capítulo 48: ORDEN POR

Examples

Utilice ORDER BY con TOP para devolver las filas superiores x basadas en el valor de una columna

En este ejemplo, podemos usar GROUP BY no solo para determinar el *tipo* de filas devueltas, sino también qué filas *se* devuelven, ya que estamos usando TOP para limitar el conjunto de resultados.

Digamos que queremos devolver a los 5 principales usuarios con mayor reputación de un sitio de preguntas y respuestas popular sin nombre.

Sin ORDENAR

Esta consulta devuelve las 5 filas principales ordenadas por el valor predeterminado, que en este caso es "Id", la primera columna de la tabla (aunque no es una columna que se muestra en los resultados).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

devoluciones...

Nombre para mostrar	Reputación
Comunidad	1
Geoff Dalgas	12567
Jarrod dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

Con ORDENAR

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
ORDER BY Reputation desc
```

devoluciones...

Nombre para mostrar	Reputación
JonSkeet	**865023**
Darin Dimitrov	**661741**
Balus	**650237**
Hans Passant	**625870**
Marc Gravell	**601636**

Observaciones

Algunas versiones de SQL (como MySQL) usan una cláusula `LIMIT` al final de `SELECT` , en lugar de `TOP` al principio, por ejemplo:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Clasificación por columnas múltiples

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY JoinDate, Reputation
```

Nombre para mostrar	Fecha de Ingreso	Reputación
Comunidad	**2008-09-15**	**1**
Jeff Atwood	**2008-09-16**	**25784**
Joel Spolsky	2008-09-16	**37628**
Jarrod dixon	**2008-10-03**	**11739**
Geoff Dalgas	2008-10-03	**12567**

Clasificación por número de columna (en lugar de nombre)

Puede usar el número de una columna (donde la columna de la izquierda es '1') para indicar en qué columna se basa la clasificación, en lugar de describir la columna por su nombre.

Pro: Si crees que es probable que cambies los nombres de las columnas más adelante, hacerlo no romperá este código.

Contras: esto generalmente reducirá la legibilidad de la consulta (se aclara instantáneamente lo

que significa 'ORDENAR POR Reputación', mientras que 'ORDENAR POR 14' requiere un recuento, probablemente con un dedo en la pantalla).

Esta consulta ordena el resultado por la información en la posición de columna relativa `3` de la declaración de selección en lugar del nombre de columna `Reputation` .

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY 3
```

Nombre para mostrar	Fecha de Ingreso	Reputación
Comunidad	2008-09-15	**1**
Jarrod dixon	2008-10-03	**11739**
Geoff Dalgas	2008-10-03	**12567**
Joel Spolsky	2008-09-16	**25784**
Jeff Atwood	2008-09-16	**37628**

Ordenar por Alias

Debido al orden de procesamiento de consultas lógicas, el alias se puede usar en orden por.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

Y puede usar el orden relativo de las columnas en la declaración de selección. Considere el mismo ejemplo que el anterior y en lugar de usar el alias use el orden relativo, ya que para el nombre de visualización es 1, para Jd es 2 y así sucesivamente

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3
```

Orden de clasificación personalizado

Para ordenar esta tabla `Employee` por departamento, usaría `ORDER BY Department` . Sin embargo, si desea un orden de clasificación diferente que no sea alfabético, debe asignar los valores del `Department` valores diferentes que se clasifiquen correctamente; Esto se puede hacer con una expresión CASE:

Nombre	Departamento
Hasan	ESO

Nombre	Departamento
Yusuf	HORA
Hillary	HORA
Joe	ESO
Alegre	HORA
Conocer	Contador

```
SELECT *
FROM Employee
ORDER BY CASE Department
         WHEN 'HR'         THEN 1
         WHEN 'Accountant' THEN 2
         ELSE                   3
         END;
```

Nombre	Departamento
Yusuf	**HORA**
Hillary	**HORA**
Alegre	**HORA**
Conocer	**Contador**
Hasan	**ESO**
Joe	**ESO**

Lea ORDEN POR en línea: https://riptutorial.com/es/sql/topic/620/orden-por

Capítulo 49: OTORGAR y REVERTIR

Sintaxis

- OTORGAR [privilegio1] [, [privilegio2] ...] EN [tabla] A [concesionario1] [, [concesionario2] ...] [CON OPCIÓN DE SUBVENCIÓN]
- REVOKE [privilege1] [, [privilege2] ...] ON [table] FROM [grantee1] [, [grantee2] ...]

Observaciones

Otorgar permisos a los usuarios. Si se especifica `WITH GRANT OPTION` , el concesionario obtiene además el privilegio de otorgar el permiso dado o revocar los permisos otorgados anteriormente.

Examples

Otorgar / revocar privilegios

```
GRANT SELECT, UPDATE
ON Employees
TO User1, User2;
```

Otorgue permiso a `User1` y `User2` para realizar operaciones de `SELECT` y `UPDATE` en la tabla `Employees` .

```
REVOKE SELECT, UPDATE
ON Employees
FROM User1, User2;
```

Revocar de `User1` y `User2` el permiso para realizar operaciones de `SELECT` y `UPDATE` en la tabla Empleados.

Lea OTORGAR y REVERTIR en línea: https://riptutorial.com/es/sql/topic/5574/otorgar-y-revertir

Capítulo 50: Procedimientos almacenados

Observaciones

Los procedimientos almacenados son sentencias SQL almacenadas en la base de datos que se pueden ejecutar o llamar en consultas. El uso de un procedimiento almacenado permite la encapsulación de lógica complicada o de uso frecuente, y mejora el rendimiento de las consultas mediante el uso de planes de consultas en caché. Pueden devolver cualquier valor que pueda devolver una consulta estándar.

Otros beneficios sobre las expresiones SQL dinámicas se enumeran en Wikipeida .

Examples

Crear y llamar a un procedimiento almacenado.

Los procedimientos almacenados se pueden crear a través de una GUI de administración de base de datos (ejemplo de SQL Server), o mediante una declaración SQL de la siguiente manera:

```
-- Define a name and parameters
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- Define the query to be run
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

Llamando al procedimiento:

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- Or
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- Or
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

Lea Procedimientos almacenados en línea:
https://riptutorial.com/es/sql/topic/1701/procedimientos-almacenados

Capítulo 51: Puntos de vista

Examples

Vistas simples

Una vista puede filtrar algunas filas de la tabla base o proyectar solo algunas columnas de ella:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

Si selecciona desde la vista:

```
select * from new_employees_details
```

Carné de identidad	FName	Salario	Fecha de contratación
4	Johnathon	500	24-07-2016

Vistas complejas

Una vista puede ser una consulta realmente compleja (agregaciones, uniones, subconsultas, etc.). Solo asegúrese de agregar nombres de columna para todo lo que seleccione:

```
Create VIEW dept_income AS
SELECT d.Name as DepartmentName, sum(e.salary) as TotalSalary
FROM Employees e
JOIN Departments d on e.DepartmentId = d.id
GROUP BY d.Name;
```

Ahora puedes seleccionarlo desde cualquier tabla:

```
SELECT *
FROM dept_income;
```

Nombre de Departamento	Salario total
HORA	1900
Ventas	600

Lea Puntos de vista en línea: https://riptutorial.com/es/sql/topic/766/puntos-de-vista

Capítulo 52: Secuencia

Examples

Crear secuencia

```
CREATE SEQUENCE orders_seq
START WITH     1000
INCREMENT BY   1;
```

Crea una secuencia con un valor inicial de 1000 que se incrementa en 1.

Usando Secuencias

se usa una referencia a *seq_name* .NEXTVAL para obtener el siguiente valor en una secuencia. Una sola instrucción solo puede generar un único valor de secuencia. Si hay varias referencias a NEXTVAL en una declaración, usarán el mismo número generado.

NEXTVAL puede ser utilizado para INSERTOS

```
INSERT INTO Orders (Order_UID, Customer)
        VALUES (orders_seq.NEXTVAL, 1032);
```

Puede ser utilizado para ACTUALIZACIONES

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

También se puede utilizar para selecciones.

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Lea Secuencia en línea: https://riptutorial.com/es/sql/topic/1586/secuencia

Capítulo 53: SELECCIONAR

Introducción

La instrucción SELECT está en el corazón de la mayoría de las consultas SQL. Define qué conjunto de resultados debe devolver la consulta, y casi siempre se usa junto con la cláusula FROM, que define qué parte (s) de la base de datos debe consultarse.

Sintaxis

- SELECCIONAR [DISTINTO] [columna1] [, [columna2] ...]
  DE [tabla]
  [DONDE condición]
  [GRUPO POR [columna1] [, [columna2] ...]

  [HAVING [column1] [, [column2] ...]

  [ORDENAR POR ASC | DESC]

Observaciones

SELECT determina los datos de las columnas que deben devolverse y en qué orden DE DESDE una tabla determinada (dado que coinciden con los otros requisitos en su consulta específicamente: dónde y con filtros y uniones).

```
SELECT Name, SerialNumber
FROM ArmyInfo
```

solo devolverá los resultados de las columnas `Name` y `Serial Number` , pero no de la columna denominada `Rank` , por ejemplo

```
SELECT *
FROM ArmyInfo
```

Indica que **todas las** columnas serán devueltas. Sin embargo, tenga en cuenta que es una mala práctica `SELECT *` ya que literalmente está devolviendo todas las columnas de una tabla.

Examples

Utilizando el carácter comodín para seleccionar todas las columnas en una consulta.

Considere una base de datos con las siguientes dos tablas.

Tabla de empleados:

Carné de identidad	FName	LName	DeptId
1	James	Herrero	3
2	Juan	Johnson	4

Mesa de departamentos:

Carné de identidad	Nombre
1	Ventas
2	Márketing
3	Financiar
4	ESO

Declaración de selección simple

`*` es el **carácter comodín** utilizado para seleccionar todas las columnas disponibles en una tabla.

Cuando se utiliza como sustituto de los nombres de columna explícitos, devuelve todas las columnas en todas las tablas en las que una consulta selecciona `FROM` . Este efecto se aplica a **todas las tablas a** las que accede la consulta a través de sus cláusulas `JOIN` .

Considere la siguiente consulta:

```
SELECT * FROM Employees
```

Devolverá todos los campos de todas las filas de la tabla `Employees` :

Carné de identidad	FName	LName	DeptId
1	James	Herrero	3
2	Juan	Johnson	4

Notación de puntos

Para seleccionar todos los valores de una tabla específica, el carácter comodín se puede aplicar a la tabla con *notación de puntos* .

Considere la siguiente consulta:

```
SELECT
    Employees.*,
    Departments.Name
FROM
    Employees
JOIN
    Departments
    ON Departments.Id = Employees.DeptId
```

Esto devolverá un conjunto de datos con todos los campos en la tabla `Employee` , seguido solo por el campo `Name` en la tabla `Departments` :

Carné de identidad	FName	LName	DeptId	Nombre
1	James	Herrero	3	Financiar
2	Juan	Johnson	4	ESO

Advertencias contra el uso

En general, se recomienda que se evite el uso de `*` en el código de producción siempre que sea posible, ya que puede causar una serie de problemas potenciales, entre ellos:

1. Exceso de E / S, carga de red, uso de memoria, etc., debido a que el motor de la base de datos lee datos que no son necesarios y los transmite al código frontal. Esto es particularmente preocupante cuando puede haber campos grandes como los que se usan para almacenar notas largas o archivos adjuntos.
2. Un exceso adicional de carga de IO si la base de datos necesita poner en cola los resultados internos en el disco como parte del procesamiento de una consulta más compleja que `SELECT <columns> FROM <table>` .
3. Procesamiento adicional (y / o incluso más IO) si algunas de las columnas innecesarias son:
    - Columnas computadas en bases de datos que las soportan.
    - en el caso de seleccionar de una vista, las columnas de una tabla / vista que el optimizador de consultas podría optimizar de otra manera
4. El potencial de errores inesperados si las columnas se agregan a las tablas y vistas más adelante resulta en nombres de columnas ambiguos. Por ejemplo, `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` : si se agrega una columna llamada nombre de `displayname` a la tabla de pedidos para permitir a los usuarios dar nombres significativos a sus pedidos para futuras referencias, entonces aparecerá el nombre de la columna dos veces en la salida, por lo que la cláusula `ORDER BY` será ambigua, lo que puede causar errores ("nombre de columna ambiguo" en las últimas versiones de MS SQL Server), y si no, en este ejemplo, el código de la aplicación puede comenzar a mostrar el nombre del orden donde se encuentra el nombre de la persona previsto porque la nueva columna es el primero de ese nombre devuelto, y así sucesivamente.

¿Cuándo se puede usar `*` , teniendo en cuenta la advertencia anterior?

Aunque es mejor evitarlo en el código de producción, usar `*` está bien como una abreviatura cuando se realizan consultas manuales en la base de datos para investigación o trabajo de

prototipo.

A veces, las decisiones de diseño en su aplicación lo hacen inevitable (en tales circunstancias, prefiera `tablealias.*` lugar de `*` solo cuando sea posible).

Cuando se utiliza `EXISTS` , como `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` , no `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` ningún dato de B. Por lo tanto, una combinación no es necesaria, y el motor sabe que no se deben devolver los valores de B, por lo que no hay un impacto de rendimiento para usar `*` . De manera similar, `COUNT(*)` está bien, ya que tampoco devuelve ninguna de las columnas, por lo que solo necesita leer y procesar aquellas que se utilizan para fines de filtrado.

Seleccionando con Condicion

La sintaxis básica de SELECT con la cláusula WHERE es:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

La *[condición]* puede ser cualquier expresión SQL, especificada mediante operadores de comparación u lógicos como>, <, =, <>,> =, <=, LIKE, NOT, IN, BETWEEN, etc.

La siguiente declaración devuelve todas las columnas de la tabla 'Coches' donde la columna de estado es 'LISTO':

```
SELECT * FROM Cars WHERE status = 'READY'
```

Vea DÓNDE y HABER para más ejemplos.

Seleccionar columnas individuales

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
FROM Customers
```

Esta declaración devolverá las columnas `PhoneNumber` , `Email` y `PreferredContact` de todas las filas de la tabla `Customers` . Además, las columnas se devolverán en la secuencia en la que aparecen en la cláusula `SELECT` .

El resultado será:

Número de teléfono	Email	Contacto preferido
3347927472	william.jones@example.com	TELÉFONO
2137921892	dmiller@example.net	CORREO ELECTRÓNICO

Número de teléfono	Email	Contacto preferido
NULO	richard0123@example.com	CORREO ELECTRÓNICO

Si se unen varias tablas, puede seleccionar columnas de tablas específicas especificando el nombre de la tabla antes del nombre de la columna: `[table_name].[column_name]`

```
SELECT
    Customers.PhoneNumber,
    Customers.Email,
    Customers.PreferredContact,
    Orders.Id AS OrderId
FROM
    Customers
LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

* `AS OrderId` significa que el campo `Id` de la tabla de `Orders` se devolverá como una columna denominada `OrderId` . Consulte la selección con alias de columna para obtener más información.

Para evitar usar nombres largos de tablas, puede usar alias de tablas. Esto mitiga el dolor de escribir nombres de tablas largas para cada campo que seleccione en las combinaciones. Si está realizando una unión automática (una unión entre dos instancias de la *misma* tabla), debe usar alias de tabla para distinguir sus tablas. Podemos escribir un alias de tabla como `Customers c` o `Customers AS c` . Aquí `c` funciona como un alias para los `Customers` y podemos seleccionar, digamos, `Email` como este: `c.Email` . `Email` .

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact,
    o.Id AS OrderId
FROM
    Customers c
LEFT JOIN
    Orders o ON o.CustomerId = c.Id
```

SELECCIONAR utilizando alias de columna

Los alias de columna se utilizan principalmente para acortar el código y hacer que los nombres de columna sean más legibles.

El código se acorta, ya que los nombres de tablas largas y la identificación innecesaria de columnas *(por ejemplo, puede haber 2 ID en la tabla, pero solo se usa una en la declaración)* se pueden evitar. Junto con los alias de tablas, esto le permite usar nombres descriptivos más largos en la estructura de su base de datos mientras mantiene las consultas sobre esa estructura concisa.

Además, a veces se *requieren* , por ejemplo, en vistas, para nombrar salidas computadas.

Todas las versiones de SQL

Los alias se pueden crear en todas las versiones de SQL usando comillas dobles (`"`).

```
SELECT
    FName AS "First Name",
    MName AS "Middle Name",
    LName AS "Last Name"
FROM Employees
```

Diferentes versiones de SQL

Puede usar comillas simples (`'`), comillas dobles (`"`) y corchetes (`[]`) para crear un alias en Microsoft SQL Server.

```
SELECT
    FName AS "First Name",
    MName AS 'Middle Name',
    LName AS [Last Name]
FROM Employees
```

Ambos resultarán en:

Nombre de pila	Segundo nombre	Apellido
James	Juan	Herrero
Juan	James	Johnson
Miguel	Marcus	Williams

Esta declaración volverá `FName` y `LName` columnas con un nombre dado (un alias). Esto se logra utilizando el operador `AS` seguido del alias, o simplemente escribiendo el alias directamente después del nombre de la columna. Esto significa que la siguiente consulta tiene el mismo resultado que el anterior.

```
SELECT
    FName "First Name",
    MName "Middle Name",
    LName "Last Name"
FROM Employees
```

Nombre de pila	Segundo nombre	Apellido
James	Juan	Herrero
Juan	James	Johnson
Miguel	Marcus	Williams

Sin embargo, la versión explícita (es decir, usar el operador `AS`) es más legible.

Si el alias tiene una sola palabra que no es una palabra reservada, podemos escribirla sin comillas simples, comillas dobles o corchetes:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

Nombre de pila	Apellido
James	Herrero
Juan	Johnson
Miguel	Williams

Una variación adicional disponible en MS SQL Server entre otras es `<alias> = <column-or-calculation>` , por ejemplo:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1    = FullStreetAddress,
       Addr2    = TownName
FROM CustomerDetails
```

que es equivalente a:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress          As Addr1,
       TownName                   As Addr2
FROM CustomerDetails
```

Ambos resultarán en:

Nombre completo	Addr1	Addr2
James Smith	123 AnyStreet	Ciudadville
John Johnson	668 MyRoad	Cualquier pueblo
Michael Williams	999 High End Dr	Williamsburgh

Algunos encuentran que usar `=` lugar de `As` más fácil de leer, aunque muchos recomiendan este formato, principalmente porque no es estándar, por lo que no todas las bases de datos lo admiten ampliamente. Puede causar confusión con otros usos del carácter `=` .

Todas las versiones de SQL

Además, si *necesita* usar palabras reservadas, puede usar corchetes o comillas para escapar:

```
SELECT
    FName as "SELECT",
    MName as "FROM",
    LName as "WHERE"
FROM Employees
```

Diferentes versiones de SQL

Del mismo modo, puede escapar de las palabras clave en MSSQL con todos los enfoques diferentes:

```
SELECT
    FName AS "SELECT",
    MName AS 'FROM',
    LName AS [WHERE]
FROM Employees
```

SELECCIONAR	DESDE	DÓNDE
James	Juan	Herrero
Juan	James	Johnson
Miguel	Marcus	Williams

Además, se puede usar un alias de columna en cualquiera de las cláusulas finales de la misma consulta, como `ORDER BY` :

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
    Employees
ORDER BY
    LastName DESC
```

Sin embargo, usted *no* puede usar

```
SELECT
    FName AS SELECT,
    LName AS FROM
FROM
    Employees
ORDER BY
    LastName DESC
```

Para crear un alias a partir de estas palabras reservadas (`SELECT` y `FROM`).

Esto causará numerosos errores en la ejecución.

Selección con resultados ordenados

```
SELECT * FROM Employees ORDER BY LName
```

Esta declaración devolverá todas las columnas de la tabla `Employees` .

Carné de identidad	FName	LName	Número de teléfono
2	Juan	Johnson	2468101214
1	James	Herrero	1234567890
3	Miguel	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

O

```
 SELECT * FROM Employees ORDER BY LName ASC
```

Esta declaración cambia la dirección de clasificación.

Uno también puede especificar múltiples columnas de clasificación. Por ejemplo:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

Este ejemplo ordenará los resultados primero por `LName` y luego, para los registros que tengan el mismo `LName` , ordene por `FName` . Esto le dará un resultado similar al que encontraría en una guía telefónica.

Para guardar volver a escribir el nombre de la columna en la cláusula `ORDER BY` , es posible usar el número de la columna. Tenga en cuenta que los números de columna comienzan desde 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

También puede incrustar una sentencia `CASE` en la cláusula `ORDER BY` .

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones` THEN 0
ELSE 1 END ASC
```

Esto ordenará sus resultados para tener todos los registros con el `LName` de "Jones" en la parte superior.

Seleccionar columnas que tengan nombres de palabras clave reservadas

Cuando un nombre de columna coincide con una palabra clave reservada, el estándar de SQL requiere que lo incluya entre comillas dobles:

```
SELECT
```

```
    "ORDER",
    ID
FROM ORDERS
```

Tenga en cuenta que hace que el nombre de la columna distinga entre mayúsculas y minúsculas.

Algunos DBMS tienen formas propietarias de citar nombres. Por ejemplo, SQL Server utiliza corchetes para este propósito:

```
SELECT
    [Order],
    ID
FROM ORDERS
```

mientras que MySQL (y MariaDB) por defecto usan backticks:

```
SELECT
    `Order`,
    id
FROM orders
```

Selección del número especificado de registros

El estándar SQL 2008 define la cláusula `FETCH FIRST` para limitar el número de registros devueltos.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

Este estándar solo se admite en versiones recientes de algunos RDMS. La sintaxis no estándar específica del proveedor se proporciona en otros sistemas. Progress OpenEdge 11.x también admite la sintaxis `FETCH FIRST <n> ROWS ONLY` .

Además, `OFFSET <m> ROWS` antes de `FETCH FIRST <n> ROWS ONLY` permite saltar filas antes de buscar filas.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

La siguiente consulta es compatible con SQL Server y MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
```

Para hacer lo mismo en MySQL o PostgreSQL, se debe usar la palabra clave `LIMIT` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

En Oracle se puede hacer lo mismo con `ROWNUM` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Resultados : **10** registros.

```
Id    ProductName               UnitPrice             Package
38    Côte de Blaye             263.50                12 - 75 cl bottles
29    Thüringer Rostbratwurst   123.79                50 bags x 30 sausgs.
9    Mishi Kobe Niku            97.00                 18 - 500 g pkgs.
20    Sir Rodney's Marmalade    81.00                 30 gift boxes
18    Carnarvon Tigers          62.50                 16 kg pkg.
59    Raclette Courdavault      55.00                 5 kg pkg.
51    Manjimup Dried Apples     53.00                 50 - 300 g pkgs.
62    Tarte au sucre            49.30                 48 pies
43    Ipoh Coffee               46.00                 16 - 500 g tins
28    Rössle Sauerkraut         45.60                 25 - 825 g cans
```

Matices de proveedores:

Es importante tener en cuenta que el `TOP` en Microsoft SQL funciona después de la cláusula `WHERE` y devolverá el número especificado de resultados si existen en cualquier lugar de la tabla, mientras que `ROWNUM` funciona como parte de la cláusula `WHERE` por lo que si no existen otras condiciones en el al especificar la cantidad de filas al comienzo de la tabla, obtendrá cero resultados cuando podría haber otras.

Seleccionando con alias de tabla

```
SELECT e.Fname, e.LName
FROM Employees e
```

La tabla Empleados recibe el alias 'e' directamente después del nombre de la tabla. Esto ayuda a eliminar la ambigüedad en escenarios en los que varias tablas tienen el mismo nombre de campo y debe ser específico en cuanto a la tabla de la que desea devolver los datos.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
    JOIN Managers m ON e.ManagerId = m.Id
```

Tenga en cuenta que una vez que define un alias, ya no puede usar el nombre de la tabla canónica. es decir,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
```

```
FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

lanzaría un error.

Vale la pena señalar que los alias de tablas, más formalmente "variables de rango", se introdujeron en el lenguaje SQL para resolver el problema de las columnas duplicadas causadas por `INNER JOIN` . El estándar SQL de 1992 corrigió este defecto de diseño anterior al introducir `NATURAL JOIN` (implementado en mySQL, PostgreSQL y Oracle, pero aún no en SQL Server), cuyo resultado nunca tiene nombres de columna duplicados. El ejemplo anterior es interesante ya que las tablas se unen en columnas con diferentes nombres (`Id` y `ManagerId`), pero no se supone que deben estar unidos en las columnas con el mismo nombre (`LName` , `FName`), lo que requiere el cambio de nombre de las columnas para llevar a cabo *antes de* la unión:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
     NATURAL JOIN
     ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
       FROM Managers ) m;
```

Tenga en cuenta que aunque se debe declarar una variable de alias / rango para la tabla dividida (de lo contrario, SQL generará un error), nunca tiene sentido utilizarla en la consulta.

Seleccionar filas de tablas múltiples

```
SELECT *
FROM
    table1,
    table2
```

```
SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1,
    table2
```

Esto se llama producto cruzado en SQL, es igual que producto cruzado en conjuntos

Estas declaraciones devuelven las columnas seleccionadas de varias tablas en una consulta.

No hay una relación específica entre las columnas devueltas de cada tabla.

Seleccionando con funciones agregadas

Promedio

La función agregada `AVG()` devolverá el promedio de los valores seleccionados.

```
SELECT AVG(Salary) FROM Employees
```

Las funciones agregadas también se pueden combinar con la cláusula where.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Las funciones agregadas también se pueden combinar con una cláusula por grupo.

Si el empleado está categorizado con varios departamentos y queremos encontrar un salario promedio para cada departamento, podemos usar la siguiente consulta.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Mínimo

La función agregada `MIN()` devolverá el mínimo de valores seleccionados.

```
SELECT MIN(Salary) FROM Employees
```

Máximo

La función agregada `MAX()` devolverá el máximo de los valores seleccionados.

```
SELECT MAX(Salary) FROM Employees
```

Contar

La función agregada `COUNT()` devolverá el conteo de los valores seleccionados.

```
SELECT Count(*) FROM Employees
```

También se puede combinar con las condiciones donde se obtiene el recuento de filas que satisfacen condiciones específicas.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

También se pueden especificar columnas específicas para obtener el número de valores en la columna. Tenga en cuenta que los valores `NULL` no se cuentan.

```
Select Count(ManagerId) from Employees
```

El recuento también se puede combinar con la palabra clave distinta para un recuento distinto.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Suma

La función agregada `SUM()` devuelve la suma de los valores seleccionados para todas las filas.

```
SELECT SUM(Salary) FROM Employees
```

Seleccionando con nulo

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

La selección con nulos toma una sintaxis diferente. No use `=` , use `IS NULL` o `IS NOT NULL` lugar.

Seleccionando con CASO

Cuando los resultados deben tener alguna lógica aplicada "al vuelo", se puede usar la declaración CASE para implementarla.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

también puede ser encadenado

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         WHEN Col1 > 50 AND Col1 <100 THEN 'between'
         ELSE 'over'
    END threshold
FROM TableName
```

También se puede tener `CASE` dentro de otra sentencia `CASE` .

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         ELSE
            CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
            ELSE 'over' END
    END threshold
FROM TableName
```

Seleccionando sin bloquear la mesa

A veces, cuando las tablas se usan principalmente (o solo) para lecturas, la indexación ya no ayuda y cada bit cuenta, uno puede usar selecciones sin BLOQUEO para mejorar el rendimiento.

---

servidor SQL

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oráculo

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

donde `UR` significa "lectura no comprometida".

Si se utiliza en la tabla que tiene modificaciones de registro en curso, puede tener resultados impredecibles.

Seleccione distinto (solo valores únicos)

```
SELECT DISTINCT ContinentCode
FROM Countries;
```

Esta consulta devolverá todos los valores `DISTINCT` (únicos, diferentes) de `ContinentCode` columna `ContinentCode` de la tabla de `Countries`

Código Continente
jefe
UE
COMO
N / A
AF

Demostración de SQLFiddle

Seleccione con la condición de múltiples valores de la columna

```
SELECT * FROM Cars  WHERE status IN ( 'Waiting', 'Working' )
```

Esto es semánticamente equivalente a

```
SELECT * FROM Cars  WHERE ( status = 'Waiting' OR status = 'Working' )
```

es decir, el `value IN ( <value list> )` es una abreviatura de disyunción (`OR` lógico).

Obtener resultado agregado para grupos de filas

Recuento de filas basadas en un valor de columna específico:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Obtención de ingresos medios por departamento:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

Lo importante es seleccionar solo las columnas especificadas en la cláusula `GROUP BY` o utilizadas con funciones agregadas .

La cláusula `WHERE` también se puede utilizar con `GROUP BY` , pero `WHERE` filtra los registros *antes de que* se realice la agrupación:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

Si necesita filtrar los resultados después de que se haya realizado la agrupación, por ejemplo, para ver solo los departamentos cuyo ingreso promedio es mayor a 1000, debe usar la cláusula `HAVING` :

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

Seleccionando con más de 1 condición.

La palabra clave `AND` se utiliza para agregar más condiciones a la consulta.

Nombre	Años	Género
Sam	18	METRO

Nombre	Años	Género
Juan	21	METRO
Mover	22	METRO
María	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

Esto volverá:

Nombre
Juan
Mover

usando la palabra clave `OR`

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

Esto volverá:

nombre
Sam
Juan
Mover

Estas palabras clave se pueden combinar para permitir combinaciones de criterios más complejas:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
   OR (gender = 'F' AND age > 20);
```

Esto volverá:

nombre
Sam
María

Lea SELECCIONAR en línea: https://riptutorial.com/es/sql/topic/222/seleccionar

Capítulo 54: Sinónimos

Examples

Crear sinónimo

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Lea Sinónimos en línea: https://riptutorial.com/es/sql/topic/2518/sinonimos

Capítulo 55: SKIP TAKE (Paginación)

Examples

Saltando algunas filas del resultado

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- skips 20 for take use very large number that is more than rows in table
```

Oráculo:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL:

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

Limitar la cantidad de resultados

ISO / ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Oráculo:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

Servidor SQL:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

Saltando luego tomando algunos resultados (Paginación)

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Oráculo; Servidor SQL:

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

Lea SKIP TAKE (Paginación) en línea: https://riptutorial.com/es/sql/topic/2927/skip-take--paginacion-

Capítulo 56: Subconsultas

Observaciones

Las subconsultas pueden aparecer en diferentes cláusulas de una consulta externa, o en la operación establecida.

Deben estar entre paréntesis `()` . Si el resultado de la subconsulta se compara con otra cosa, el número de columnas debe coincidir. Los alias de tabla son necesarios para las subconsultas en la cláusula FROM para nombrar la tabla temporal.

Examples

Subconsulta en la cláusula WHERE

Utilice una subconsulta para filtrar el conjunto de resultados. Por ejemplo, esto devolverá a todos los empleados con un salario igual al empleado mejor pagado.

```
SELECT *
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Subconsulta en la cláusula FROM

Una subconsulta en una cláusula `FROM` actúa de manera similar a una tabla temporal que se genera durante la ejecución de una consulta y se pierde después.

```
SELECT Managers.Id, Employees.Salary
FROM (
  SELECT Id
  FROM Employees
  WHERE ManagerId IS NULL
) AS Managers
JOIN Employees ON Managers.Id = Employees.Id
```

Subconsulta en cláusula SELECT

```
SELECT
  Id,
  FName,
  LName,
  (SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id) AS NumberOfCars
FROM Customers
```

Subconsultas en la cláusula FROM

Puede usar subconsultas para definir una tabla temporal y usarla en la cláusula FROM de una

consulta "externa".

```
SELECT * FROM (SELECT city, temp_hi - temp_lo AS temp_var FROM weather) AS w
WHERE temp_var > 20;
```

Lo anterior encuentra ciudades de la tabla meteorológica cuya variación de temperatura diaria es superior a 20. El resultado es:

ciudad	temp_var
SAN LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

.

Subconsultas en la cláusula WHERE

El siguiente ejemplo encuentra ciudades (del ejemplo de ciudades) cuya población está por debajo de la temperatura promedio (obtenida a través de una sub-consulta):

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000)  FROM cities);
```

Aquí: la subconsulta (SELECT avg (pop2000) FROM cities) se usa para especificar condiciones en la cláusula WHERE. El resultado es:

nombre	pop2000
San Francisco	776733
SAN LOUIS	348189
ciudad de Kansas	146866

Subconsultas en cláusula SELECT

Las subconsultas también se pueden utilizar en la parte `SELECT` de la consulta externa. La siguiente consulta muestra todas las columnas de la tabla de clima con los estados correspondientes de la tabla de ciudades .

```
SELECT w.*,  (SELECT c.state FROM cities AS c WHERE c.name = w.city ) AS state
FROM weather AS w;
```

Filtrar resultados de consultas usando consultas en diferentes tablas.

Esta consulta selecciona a todos los empleados que no están en la tabla de Supervisores.

```
SELECT *
FROM Employees
WHERE EmployeeID not in (SELECT EmployeeID
                            FROM Supervisors)
```

Se pueden lograr los mismos resultados utilizando un IZQUIERDA.

```
SELECT *
FROM Employees AS e
LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID is NULL
```

Subconsultas correlacionadas

Las subconsultas correlacionadas (también conocidas como sincronizadas o coordinadas) son consultas anidadas que hacen referencias a la fila actual de su consulta externa:

```
SELECT EmployeeId
    FROM Employee AS eOuter
    WHERE Salary > (
       SELECT AVG(Salary)
       FROM Employee eInner
       WHERE eInner.DepartmentId = eOuter.DepartmentId
    )
```

La subconsulta `SELECT AVG(Salary) ...` está *correlacionada* porque se refiere a la fila de `Employee eOuter` de su consulta externa.

Lea Subconsultas en línea: https://riptutorial.com/es/sql/topic/1606/subconsultas

Capítulo 57: Tipos de datos

Examples

DECIMAL Y NUMÉRICO

Números decimales fijos de precisión y escala. `DECIMAL` y `NUMERIC` son funcionalmente equivalentes.

Sintaxis:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Ejemplos:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

FLOTANTE Y REAL

Tipos de datos de números aproximados para usar con datos numéricos de punto flotante.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Enteros

Tipos de datos de números exactos que utilizan datos enteros.

Tipo de datos	Distancia	Almacenamiento
Bigint	-2 ^ 63 (-9,223,372,036,854,775,808) a 2 ^ 63-1 (9,223,372,036,854,775,807)	8 bytes
En t	-2 ^ 31 (-2,147,483,648) a 2 ^ 31-1 (2,147,483,647)	4 bytes
smallint	-2 ^ 15 (-32,768) a 2 ^ 15-1 (32,767)	2 bytes
tinyint	0 a 255	1 byte

DINERO Y PEQUEÑAS

Tipos de datos que representan valores monetarios o monetarios.

Tipo de datos	Distancia	Almacenamiento
dinero	-922,337,203,685,477.5808 a 922,337,203,685,477.5807	8 bytes
poco dinero	-214,748.3648 a 214,748.3647	4 bytes

BINARIO Y VARBINARIO

Tipos de datos binarios de longitud fija o variable.

Sintaxis:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

`n_bytes` puede ser cualquier número de 1 a 8000 bytes. `max` indica que el espacio de almacenamiento máximo es 2 ^ 31-1.

Ejemplos:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x0000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

CHAR y VARCHAR

Tipos de datos de cadena de longitud fija o variable.

Sintaxis:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Ejemplos:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC       ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10))  -- 'ABCDEFGHIJ' (truncated to 10
characters)
```

NCHAR y NVARCHAR

Tipos de datos de cadena UNICODE de longitud fija o variable.

Sintaxis:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Use `MAX` para cadenas muy largas que pueden exceder los 8000 caracteres.

IDENTIFICADOR ÚNICO

Un GUID / UUID de 16 bytes.

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string,   -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```

Lea Tipos de datos en línea: https://riptutorial.com/es/sql/topic/1166/tipos-de-datos

Capítulo 58: TRATA DE ATRAPARLO

Observaciones

TRY / CATCH es una construcción de lenguaje específica para T-SQL de MS SQL Server.

Permite el manejo de errores dentro de T-SQL, similar al visto en el código .NET.

Examples

Transacción en un TRY / CATCH

Esto hará retroceder ambas inserciones debido a una fecha y hora no válida:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Esto comprometerá ambas inserciones:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Lea TRATA DE ATRAPARLO en línea: https://riptutorial.com/es/sql/topic/4420/trata-de-atraparlo

Capítulo 59: TRUNCAR

Introducción

La instrucción TRUNCATE elimina todos los datos de una tabla. Esto es similar a BORRAR sin filtro, pero, dependiendo del software de la base de datos, tiene ciertas restricciones y optimizaciones.

Sintaxis

- TRUNCATE TABLE table_name;

Observaciones

TRUNCATE es un comando DDL (lenguaje de definición de datos), y como tal, hay diferencias significativas entre él y DELETE (un lenguaje de manipulación de datos, DML, comando). Si bien TRUNCATE puede ser un medio para eliminar rápidamente grandes volúmenes de registros de una base de datos, estas diferencias deben entenderse para decidir si el uso de un comando TRUNCATE es adecuado para su situación particular.

- TRUNCATE es una operación de página de datos. Por lo tanto, los activadores de DML (ON DELETE) asociados con la tabla no se activarán cuando realice una operación TRUNCATE. Si bien esto ahorrará una gran cantidad de tiempo para operaciones masivas de eliminación, sin embargo, es posible que deba eliminar manualmente los datos relacionados.
- TRUNCATE liberará el espacio en disco utilizado por las filas eliminadas, DELETE liberará espacio
- Si la tabla a truncar utiliza columnas de identidad (MS SQL Server), entonces el comando TRUNCATE restablece la semilla. Esto puede resultar en problemas de integridad referencial.
- Dependiendo de los roles de seguridad existentes y la variante de SQL en uso, es posible que no tenga los permisos necesarios para ejecutar un comando TRUNCATE

Examples

Eliminar todas las filas de la tabla Empleado

```
TRUNCATE TABLE Employee;
```

El uso de la tabla truncada suele ser mejor que el uso de BORRAR TABLA ya que ignora todos los índices y activadores y solo elimina todo.

Eliminar tabla es una operación basada en filas, esto significa que cada fila se elimina. La tabla truncada es una operación de página de datos, se reasigna toda la página de datos. Si tiene una

tabla con un millón de filas, será mucho más rápido truncar la tabla que utilizar una declaración de eliminación de tabla.

Si bien podemos eliminar filas específicas con BORRAR, no podemos TRUNCAR filas específicas, solo podemos TRUNCAR todos los registros a la vez. Eliminar todas las filas y luego insertar un nuevo registro continuará agregando el valor de la Clave primaria incrementada automáticamente del valor insertado anteriormente, donde, como Truncar, el valor de la clave primaria Incremental automática también se restablecerá y comenzará desde 1.

Tenga en cuenta que al truncar la tabla, **no debe haber claves externas** , de lo contrario obtendrá un error.

Lea TRUNCAR en línea: https://riptutorial.com/es/sql/topic/1466/truncar

Capítulo 60: UNION / UNION ALL

Introducción

La palabra clave ***UNION*** en SQL se usa para combinar los resultados de la instrucción ***SELECT*** sin duplicar. Para utilizar UNION y combinar los resultados, ambas sentencias SELECT deben tener el mismo número de columnas con el mismo tipo de datos en el mismo orden, pero la longitud de la columna puede ser diferente.

Sintaxis

- SELECCIONE column_1 [, column_2] FROM table_1 [, table_2] [WHERE condicion]
  UNION | UNION TODO
  SELECCIONE column_1 [, column_2] FROM table_1 [, table_2] [WHERE condicion]

Observaciones

`UNION` cláusulas `UNION` y `UNION ALL` combinan el conjunto de resultados de dos o más sentencias SELECT idénticamente estructuradas en un solo resultado / tabla.

Tanto el recuento de columnas como los tipos de columnas para cada consulta deben coincidir para que funcione `UNION` / `UNION ALL` .

La diferencia entre una consulta `UNION` y `UNION ALL` es que la cláusula `UNION` eliminará cualquier fila duplicada en el resultado donde la `UNION ALL` no lo hará.

Esta eliminación distinta de registros puede ralentizar significativamente las consultas, incluso si no hay filas distintas para eliminar debido a esto, si sabe que no habrá duplicados (o que no le importe), el valor predeterminado es `UNION ALL` para una consulta más optimizada.

Examples

Consulta básica de UNION ALL

```
CREATE TABLE HR_EMPLOYEES
(
    PersonID int,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    Position VARCHAR(30)
);

CREATE TABLE FINANCE_EMPLOYEES
(
    PersonID INT,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
```

```
    Position VARCHAR(30)
);
```

Digamos que queremos extraer los nombres de todos los `managers` de nuestros departamentos.

Usando un `UNION` podemos obtener todos los empleados de los departamentos de recursos humanos y finanzas, que ocupan el `position` de `manager`

```
SELECT
    FirstName, LastName
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

La instrucción `UNION` elimina filas duplicadas de los resultados de la consulta. Dado que es posible tener personas con el mismo nombre y posición en ambos departamentos, estamos utilizando `UNION ALL` para no eliminar duplicados.

Si desea usar un alias para cada columna de salida, solo puede ponerlos en la primera instrucción de selección, de la siguiente manera:

```
SELECT
    FirstName as 'First Name', LastName as 'Last Name'
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Explicación simple y ejemplo.

En lenguaje sencillo:

- `UNION` une 2 conjuntos de resultados mientras elimina duplicados del conjunto de resultados
- `UNION ALL` une 2 conjuntos de resultados sin intentar eliminar duplicados

  Un error que cometen muchas personas es usar `UNION` cuando no necesitan que se eliminen los duplicados. El costo de rendimiento adicional frente a grandes conjuntos de resultados puede ser muy significativo.

Cuando necesites `UNION`

Supongamos que necesita filtrar una tabla por 2 atributos diferentes, y ha creado índices no agrupados separados para cada columna. `UNION` permite aprovechar ambos índices mientras evita los duplicados.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

Esto simplifica su ajuste de rendimiento, ya que solo se necesitan índices simples para realizar estas consultas de manera óptima. Es posible que incluso pueda arreglárselas con un poco menos de índices no agrupados, lo que también mejora el rendimiento general de escritura en la tabla de origen.

Cuando necesites `UNION ALL`

Supongamos que aún necesita filtrar una tabla contra 2 atributos, pero no necesita filtrar registros duplicados (ya sea porque no importa o porque sus datos no producirían duplicados durante la unión debido al diseño de su modelo de datos).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

Esto es especialmente útil cuando se crean vistas que unen datos que están diseñados para particionarse físicamente en varias tablas (tal vez por razones de rendimiento, pero que aún así quieran acumular registros). Dado que los datos ya están divididos, hacer que el motor de la base de datos elimine los duplicados no agrega ningún valor y solo agrega tiempo de procesamiento adicional a las consultas.

Lea UNION / UNION ALL en línea: https://riptutorial.com/es/sql/topic/349/union---union-all

Capítulo 61: UNIR

Introducción

MERGE (a menudo también llamado UPSERT para "actualizar o insertar") permite insertar nuevas filas o, si ya existe una fila, actualizar la fila existente. El punto es realizar todo el conjunto de operaciones de forma atómica (para garantizar que los datos permanezcan consistentes), y para evitar la sobrecarga de comunicación para varias declaraciones SQL en un sistema cliente / servidor.

Examples

MERGE para hacer Target Match Source

```
MERGE INTO targetTable t
    USING sourceTable s
        ON t.PKID = s.PKID
    WHEN MATCHED AND NOT EXISTS (
            SELECT s.ColumnA, s.ColumnB, s.ColumnC
            INTERSECT
            SELECT t.ColumnA, t.ColumnB, s.ColumnC
            )
        THEN UPDATE SET
            t.ColumnA = s.ColumnA
            ,t.ColumnB = s.ColumnB
            ,t.ColumnC = s.ColumnC
    WHEN NOT MATCHED BY TARGET
        THEN INSERT (PKID, ColumnA, ColumnB, ColumnC)
        VALUES (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
    WHEN NOT MATCHED BY SOURCE
        THEN DELETE
    ;
```

Nota: la parte `AND NOT EXISTS` evita la actualización de registros que no han cambiado. El uso de la construcción `INTERSECT` permite que las columnas anulables se comparen sin un manejo especial.

MySQL: contando usuarios por nombre

Supongamos que queremos saber cuántos usuarios tienen el mismo nombre. Vamos a crear `users` tablas de la siguiente manera:

```
create table users(
    id int primary key auto_increment,
    name varchar(8),
    count int,
    unique key name(name)
);
```

Ahora, acabamos de descubrir un nuevo usuario llamado Joe y nos gustaría tenerlo en cuenta.

Para lograrlo, debemos determinar si existe una fila con su nombre y, de ser así, actualizarla para incrementar el conteo; por otro lado, si no hay una fila existente, deberíamos crearla.

MySQL usa la siguiente sintaxis: insertar ... en actualización de clave duplicada ... En este caso:

```
insert into users(name, count)
       values ('Joe', 1)
       on duplicate key update count=count+1;
```

PostgreSQL: contando usuarios por nombre

Supongamos que queremos saber cuántos usuarios tienen el mismo nombre. Vamos a crear `users` tablas de la siguiente manera:

```
create table users(
    id serial,
    name varchar(8) unique,
    count int
);
```

Ahora, acabamos de descubrir un nuevo usuario llamado Joe y nos gustaría tenerlo en cuenta. Para lograrlo, debemos determinar si existe una fila con su nombre y, de ser así, actualizarla para incrementar el conteo; por otro lado, si no hay una fila existente, deberíamos crearla.

PostgreSQL usa la siguiente sintaxis: insertar ... en conflicto ... actualizar ... En este caso:

```
insert into users(name, count)
    values('Joe', 1)
    on conflict (name) do update set count = users.count + 1;
```

Lea UNIR en línea: https://riptutorial.com/es/sql/topic/1470/unir

Capítulo 62: UNIRSE

Introducción

JOIN es un método para combinar (unir) información de dos tablas. El resultado es un conjunto de columnas unidas de ambas tablas, definidas por el tipo de combinación (INNER / OUTER / CROSS y LEFT / RIGHT / FULL, explicadas a continuación) y criterios de combinación (cómo se relacionan las filas de ambas tablas).

Una tabla se puede unir a sí misma o a cualquier otra tabla. Si es necesario acceder a la información de más de dos tablas, se pueden especificar varias combinaciones en una cláusula FROM.

Sintaxis

- `[ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } ] JOIN`

Observaciones

Las combinaciones, como sugiere su nombre, son una forma de consultar datos de varias tablas de manera conjunta, con las filas que muestran columnas tomadas de más de una tabla.

Examples

Unión interna explícita básica

Una unión básica (también llamada "unión interna") consulta datos de dos tablas, con su relación definida en una cláusula de `join` .

El siguiente ejemplo seleccionará los nombres de los empleados (FName) de la tabla Empleados y el nombre del departamento para el que trabajan (Nombre) de la tabla Departamentos:

```
SELECT Employees.FName, Departments.Name
FROM   Employees
JOIN   Departments
ON Employees.DepartmentId = Departments.Id
```

Esto devolvería lo siguiente de la base de datos de ejemplo :

Empleados. Nombre	Departamentos.Nombre
James	HORA
Juan	HORA
Ricardo	Ventas

Ingreso implícito

Las combinaciones también se pueden realizar teniendo varias tablas en la cláusula `from` , separadas por comas `,` y definiendo la relación entre ellas en la cláusula `where` . Esta técnica se denomina `join` implícita (ya que en realidad no contiene una cláusula de `join`).

Todos los RDBMS lo admiten, pero la sintaxis suele desaconsejarse. Las razones por las que es una mala idea usar esta sintaxis son:

- Es posible obtener combinaciones cruzadas accidentales que luego devuelven resultados incorrectos, especialmente si tiene muchas combinaciones en la consulta.
- Si pretendía una unión cruzada, no está claro en la sintaxis (escriba CROSS JOIN en su lugar), y es probable que alguien la cambie durante el mantenimiento.

El siguiente ejemplo seleccionará los nombres de los empleados y el nombre de los departamentos para los que trabajan:

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DeptartmentId = d.Id
```

Esto devolvería lo siguiente de la base de datos de ejemplo :

e.FName	d.Nombre
James	HORA
Juan	HORA
Ricardo	Ventas

Izquierda combinación externa

Una combinación externa izquierda (también conocida como combinación izquierda o externa) es una combinación que garantiza que todas las filas de la tabla izquierda estén representadas; si no existe una fila coincidente de la tabla derecha, sus campos correspondientes son `NULL` .

El siguiente ejemplo seleccionará todos los departamentos y el primer nombre de los empleados que trabajan en ese departamento. Los departamentos sin empleados aún se devuelven en los resultados, pero tendrán NULL para el nombre del empleado:

```
SELECT          Departments.Name, Employees.FName
FROM            Departments
LEFT OUTER JOIN Employees
ON              Departments.Id = Employees.DepartmentId
```

Esto devolvería lo siguiente de la base de datos de ejemplo :

Departamentos.Nombre	Empleados. Nombre
HORA	James
HORA	Juan
HORA	Johnathon
Ventas	Miguel
Tecnología	NULO

Entonces, ¿cómo funciona esto?

Hay dos tablas en la cláusula FROM:

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratació
1	James	Herrero	1234567890	NULO	1	1000	01-01-2002
2	Juan	Johnson	2468101214	1	1	400	23-03-2005
3	Miguel	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Herrero	1212121212	2	1	500	24-07-2016

y

Carné de identidad	Nombre
1	HORA
2	Ventas
3	Tecnología

Primero se crea un producto *cartesiano a* partir de las dos tablas que dan una tabla intermedia. Los registros que cumplen con los criterios de combinación (*Departments.Id* = *Employees.DepartmentId*) están resaltados en negrita; estos se pasan a la siguiente etapa de la consulta.

Como se trata de una JUNTA EXTERNA IZQUIERDA, todos los registros se devuelven del lado IZQUIERDO de la unión (Departamentos), mientras que a los registros en el lado DERECHO se les asigna un marcador NULO si no coinciden con los criterios de la unión. En la tabla de abajo esto devolverá **Tech** con `NULL`

Carné de identidad	Nombre	Carné de identidad	FName	LName	Número de teléfono	ManagerId	Departmer
1	**HORA**	**1**	**James**	**Herrero**	**1234567890**	**NULO**	**1**
1	**HORA**	**2**	**Juan**	**Johnson**	**2468101214**	**1**	**1**
1	HORA	3	Miguel	Williams	1357911131	1	2
1	**HORA**	**4**	**Johnathon**	**Herrero**	**1212121212**	**2**	**1**
2	Ventas	1	James	Herrero	1234567890	NULO	1
2	Ventas	2	Juan	Johnson	2468101214	1	1
2	**Ventas**	**3**	**Miguel**	**Williams**	**1357911131**	**1**	**2**
2	Ventas	4	Johnathon	Herrero	1212121212	2	1
3	Tecnología	1	James	Herrero	1234567890	NULO	1
3	Tecnología	2	Juan	Johnson	2468101214	1	1
3	Tecnología	3	Miguel	Williams	1357911131	1	2
3	Tecnología	4	Johnathon	Herrero	1212121212	2	1

Finalmente, cada expresión utilizada en la cláusula **SELECT** se evalúa para devolver nuestra tabla final:

Departamentos.Nombre	Empleados. Nombre
HORA	James
HORA	Juan
Ventas	Ricardo
Tecnología	NULO

Auto unirse

Una tabla se puede unir a sí misma, con diferentes filas que coinciden entre sí por alguna condición. En este caso de uso, se deben usar alias para distinguir las dos apariciones de la tabla.

En el ejemplo a continuación, para cada empleado en la tabla de empleados de la base de datos de ejemplo , se devuelve un registro que contiene el nombre del empleado junto con el nombre correspondiente del gerente del empleado. Como los gerentes también son empleados, la tabla

se une consigo misma:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
    Employees e
JOIN
    Employees m
    ON e.ManagerId = m.Id
```

Esta consulta devolverá los siguientes datos:

Empleado	Gerente
Juan	James
Miguel	James
Johnathon	Juan

Entonces, ¿cómo funciona esto?

La tabla original contiene estos registros:

Carné de identidad	FName	LName	Número de teléfono	ManagerId	DepartmentId	Salario	Fecha de contratació
1	James	Herrero	1234567890	NULO	1	1000	01-01-2002
2	Juan	Johnson	2468101214	1	1	400	23-03-2005
3	Miguel	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Herrero	1212121212	2	1	500	24-07-2016

La primera acción es crear un producto *cartesiano* de todos los registros en las tablas utilizadas en la cláusula **FROM** . En este caso, es la tabla Empleados dos veces, por lo que la tabla intermedia se verá así (eliminé todos los campos que no se usaron en este ejemplo):

e.Id	e.FName	e.ManagerId	medio	m.FName	m.ManagerId
1	James	NULO	1	James	NULO
1	James	NULO	2	Juan	1
1	James	NULO	3	Miguel	1

e.Id	e.FName	e.ManagerId	medio	m.FName	m.ManagerId
1	James	NULO	4	Johnathon	2
2	Juan	1	1	James	NULO
2	Juan	1	2	Juan	1
2	Juan	1	3	Miguel	1
2	Juan	1	4	Johnathon	2
3	Miguel	1	1	James	NULO
3	Miguel	1	2	Juan	1
3	Miguel	1	3	Miguel	1
3	Miguel	1	4	Johnathon	2
4	Johnathon	2	1	James	NULO
4	Johnathon	2	2	Juan	1
4	Johnathon	2	3	Miguel	1
4	Johnathon	2	4	Johnathon	2

La siguiente acción es sólo para mantener los registros que cumplen los criterios de **unión,** por lo que todos los registros donde el alias `e` mesa `ManagerId` es igual al alias `m` tabla `Id` :

e.Id	e.FName	e.ManagerId	medio	m.FName	m.ManagerId
2	Juan	1	1	James	NULO
3	Miguel	1	1	James	NULO
4	Johnathon	2	2	Juan	1

Luego, cada expresión utilizada en la cláusula **SELECT** se evalúa para devolver esta tabla:

e.FName	m.FName
Juan	James
Miguel	James
Johnathon	Juan

Finalmente, los nombres de columna `e.FName` y `m.FName` se reemplazan por sus nombres de

columna de alias, asignados con el operador AS :

Empleado	Gerente
Juan	James
Miguel	James
Johnathon	Juan

Unirse a la cruz

La unión cruzada hace un producto cartesiano de los dos miembros. Un producto cartesiano significa que cada fila de una tabla se combina con cada fila de la segunda tabla en la unión. Por ejemplo, si `TABLEA` tiene 20 filas y `TABLEB` tiene 20 filas, el resultado sería `20*20 = 400` filas de salida.

Usando la base de datos de ejemplo

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

Que devuelve:

d.Nombre	e.FName
HORA	James
HORA	Juan
HORA	Miguel
HORA	Johnathon
Ventas	James
Ventas	Juan
Ventas	Miguel
Ventas	Johnathon
Tecnología	James
Tecnología	Juan
Tecnología	Miguel
Tecnología	Johnathon

Se recomienda escribir una UNIÓN CRUZADA explícita si desea realizar una unión cartesiana, para resaltar que esto es lo que desea.

Uniéndose a una subconsulta

La unión a una subconsulta se usa a menudo cuando se desean obtener datos agregados de una tabla secundaria / de detalles y mostrarlos junto con los registros de la tabla principal o de encabezado. Por ejemplo, es posible que desee obtener un recuento de registros secundarios, un promedio de alguna columna numérica en los registros secundarios o la fila superior o inferior basada en una fecha o campo numérico. Este ejemplo usa alias, lo que se puede argumentar hace que las consultas sean más fáciles de leer cuando tiene varias tablas involucradas. Así es como se ve una unión de subconsulta bastante típica. En este caso, estamos recuperando todas las filas de las órdenes de compra de la tabla principal y recuperando solo la primera fila de cada registro principal de la tabla secundaria PurchaseOrderLineItems.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
       FROM PurchaseOrderLineItems l
       GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

APLICACIÓN CRUZADA Y UNIÓN LATERAL

Un tipo muy interesante de JOIN es el LATERAL JOIN (nuevo en PostgreSQL 9.3+), que también se conoce como CROSS APPLY / OUTER APPLY en SQL-Server & Oracle.

La idea básica es que una función con valores de tabla (o una subconsulta en línea) se aplique a cada fila en la que se una.

Esto hace posible, por ejemplo, solo unir la primera entrada coincidente en otra tabla.
La diferencia entre una unión normal y una lateral reside en el hecho de que puede usar una columna que previamente se unió **en la subconsulta** que "CRUZAS APLICA".

Sintaxis:

PostgreSQL 9.3+

izquierda | derecha | interior ÚNETE **LATERAL**

Servidor SQL:

CRUZ | OUTER **APPLY**

`INNER JOIN LATERAL` es lo mismo que `CROSS APPLY`
y `LEFT JOIN LATERAL` es lo mismo que `OUTER APPLY`

Ejemplo de uso (PostgreSQL 9.3+):

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989


LEFT JOIN LATERAL
(
    SELECT
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
   LIMIT 1
) AS FirstOE
```

Y para SQL Server.

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = INNER JOIN
OUTER APPLY    -- = LEFT JOIN
(
    SELECT TOP 1
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
```

```
        (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

ÚNETE COMPLETO

Un tipo de JOIN que es menos conocido, es el FULL JOIN.
(Nota: MySQL no admite FULL JOIN a partir de 2016)

Un FULL OUTER JOIN devuelve todas las filas de la tabla izquierda y todas las filas de la tabla derecha.

Si hay filas en la tabla de la izquierda que no tienen coincidencias en la tabla de la derecha, o si hay filas en la tabla de la derecha que no tienen coincidencias en la tabla de la izquierda, esas filas también aparecerán en la lista.

Ejemplo 1 :

```
SELECT * FROM Table1

FULL JOIN Table2
     ON 1 = 2
```

Ejemplo 2:

```
SELECT
     COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
FROM T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
      ON tYear.Year = T_Budget.Year
```

Tenga en cuenta que si está utilizando eliminaciones de software, tendrá que comprobar el estado de eliminación de software de nuevo en la cláusula WHERE (porque FULL JOIN se comporta como una UNION);
Es fácil pasar por alto este pequeño hecho, ya que se coloca AP_SoftDeleteStatus = 1 en la cláusula de unión.

Además, si está realizando una UNIÓN COMPLETA, normalmente deberá permitir NULL en la cláusula WHERE; olvidarse de permitir NULL en un valor tendrá los mismos efectos que una combinación INNER, que es algo que no desea si está realizando una UNIÓN COMPLETA.

Ejemplo:

```
SELECT
     T_AccountPlan.AP_UID
    ,T_AccountPlan.AP_Code
    ,T_AccountPlan.AP_Lang_EN
```

```
    ,T_BudgetPositions.BUP_Budget
    ,T_BudgetPositions.BUP_UID
    ,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
    ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
    AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS
NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

Uniones recursivas

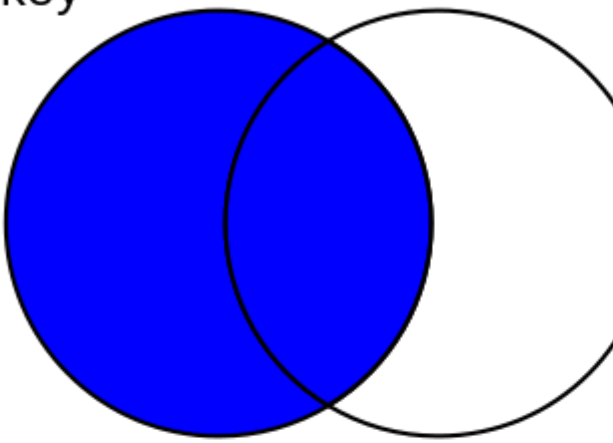
Las combinaciones recursivas se utilizan a menudo para obtener datos padre-hijo. En SQL, se implementan con expresiones de tabla comunes recursivas, por ejemplo:

```
WITH RECURSIVE MyDescendants AS (
    SELECT Name
    FROM People
    WHERE Name = 'John Doe'

    UNION ALL

    SELECT People.Name
    FROM People
    JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

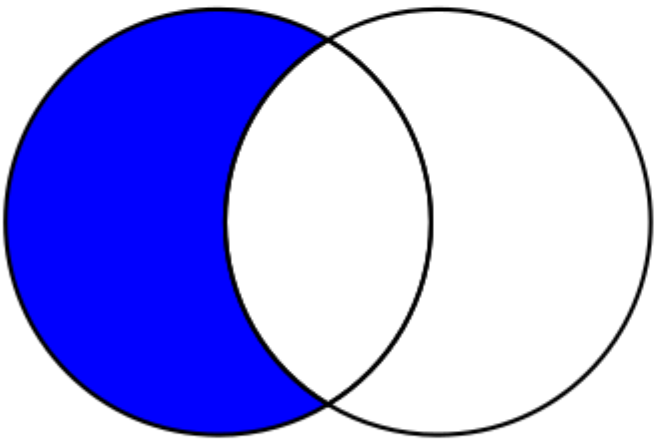
Diferencias entre uniones internas / externas

SQL tiene varios tipos de unión para especificar si se incluyen (no) filas coincidentes en el resultado: `INNER JOIN`, `LEFT OUTER JOIN`, `LEFT OUTER JOIN RIGHT OUTER JOIN`, `RIGHT OUTER JOIN FULL OUTER JOIN` (las palabras clave `INNER` y `OUTER` son opcionales). La siguiente figura subraya las diferencias entre estos tipos de uniones: el área azul representa los resultados devueltos por la unión y el área blanca representa los resultados que la unión no devolverá.

```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
```



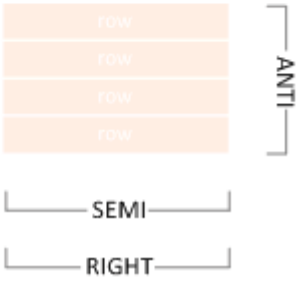
```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
WHERE B.key IS NULL
```



Tenga cuidado si está utilizando NOT IN en una columna NULABLE! Más detalles aquí .

Right Anti Semi Join

Incluye las filas de la derecha que **no** coinciden con las filas de la izquierda.




```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);

Y
-------
Tim
Vincent
```

Como puede ver, no hay una sintaxis NOT IN dedicada para la combinación semi izquierda / derecha: logramos el efecto simplemente cambiando las posiciones de la tabla dentro del texto SQL.

Cruzar

Un producto cartesiano de todas las izquierdas con todas las filas derechas.

```
SELECT * FROM A CROSS JOIN B;

X      Y
```

```
-----  -------
Amy    Lisa
John   Lisa
Lisa   Lisa
Marco  Lisa
Phil   Lisa
Amy    Marco
John   Marco
Lisa   Marco
Marco  Marco
Phil   Marco
Amy    Phil
John   Phil
Lisa   Phil
Marco  Phil
Phil   Phil
Amy    Tim
John   Tim
Lisa   Tim
Marco  Tim
Phil   Tim
Amy    Vincent
John   Vincent
Lisa   Vincent
Marco  Vincent
Phil   Vincent
```

La unión cruzada es equivalente a una unión interna con condición de unión que siempre coincide, por lo que la siguiente consulta habría dado el mismo resultado:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Auto-unirse

Esto simplemente denota una tabla que se une consigo misma. Una auto-unión puede ser cualquiera de los tipos de unión mencionados anteriormente. Por ejemplo, esta es una auto-unión interna:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);

X     X
----  -----
Amy   John
Amy   Lisa
Amy   Marco
John  Marco
Lisa  Marco
Phil  Marco
Amy   Phil
```

Lea UNIRSE en línea: https://riptutorial.com/es/sql/topic/261/unirse

Capítulo 63: Vistas materializadas

Introducción

Una vista materializada es una vista cuyos resultados se almacenan físicamente y se deben actualizar periódicamente para mantenerse actualizados. Por lo tanto, son útiles para almacenar los resultados de consultas complejas y de larga ejecución cuando no se requieren resultados en tiempo real. Las vistas materializadas se pueden crear en Oracle y PostgreSQL. Otros sistemas de bases de datos ofrecen una funcionalidad similar, como las vistas indizadas de SQL Server o las tablas de consultas materializadas de DB2.

Examples

Ejemplo de PostgreSQL

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
 number
--------
      1
(1 row)

INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
 number
--------
      1
(1 row)

REFRESH MATERIALIZED VIEW myview;

SELECT * FROM myview;
 number
--------
      1
      2
(2 rows)
```

Lea Vistas materializadas en línea: https://riptutorial.com/es/sql/topic/8367/vistas-materializadas

Capítulo 64: XML

Examples

Consulta desde tipo de datos XML

```
DECLARE @xmlIN XML = '<TableData>
<aaa Main="First">
  <row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value',  'VARCHAR(25)') [Value]
FROM    @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

Resultados

```
Header    name    Value
First      a        1
First      b        2
First      c        3
Second     a        3
Second     b        4
Second     c        5
Third      a        10
Third      b        20
Third      c        30
```

Lea XML en línea: https://riptutorial.com/es/sql/topic/4421/xml

Creditos

S. No	Capítulos	Contributors
1	Empezando con SQL	Arjan Einbu, brichins, Burkhard, cale_b, CL., Community, Devmati Wadikar, Epodax, geeksal, H. Pauwelyn, Hari, Joey, JohnLBevan, Jon Ericson, Lankymart, Laurel, Mureinik, Nathan, omini data, PeterRing, Phrancis, Prateek, RamenChef, Ray, Simone Carletti, SZenC, t1gor, ypercube
2	Actas	Amir Pourmand, CL., Daryl, John Odom
3	ACTUALIZAR	Akshay Anand, CL., Daniel Vérité, Dariusz, Dipesh Poudel, FlyingPiMonster, Gidil, H. Pauwelyn, Jon Chan, KIRAN KUMAR MATAM, Matas Vaitkevicius, Matt, Phrancis, Sanjay Bharwani, sunkuet02, Tot Zam, TriskalJM, vmaroli, WesleyJohnson
4	AGRUPAR POR	3N1GM4, Abe Miessler, Bostjan, Devmati Wadikar, Filipe Manuel, Frank, Gidil, Jaydles, juergen d, Nathaniel Ford, Peter Gordon, Simone - Ali One, WesleyJohnson, Zahiro Mor, Zoyd
5	Álgebra relacional	CL., Darren Bartrup-Cook, Martin Smith
6	ALTERAR MESA	Aidan, blackbishop, bluefeet, CL., Florin Ghita, Francis Lord, guiguiblitz, Joe W, KIRAN KUMAR MATAM, Lexi, mithra chintha , Ozair Kafray, Simon Foster, Siva Rama Krishna
7	aplicación cruzada, aplicación externa	Karthikeyan, RamenChef
8	Bloques de ejecución	Phrancis
9	BORRAR	Batsu, Chip, CL., Dylan Vander Berg, fredden, Joel, KIRAN KUMAR MATAM, Phrancis, Umesh, xenodevil, Zoyd
10	BORRAR o BORRAR la base de datos	Abhilash R Vankayala, John Odom
11	CASO	ɐlǝx, Christos, CL., Dariusz, Fenton, Infinity, Jaydles, Matt, MotKohn, Mureinik, Peter Lang, Stanislovas Kalašnikovas
12	Cláusula IN	CL., juergen d, walid, Zaga
13	Comentarios	CL., Phrancis

14	Como operador	Abhilash R Vankayala, Aidan, ashja99, Bart Schuijt, CL., Cristian Abelleira, guiguiblitz, Harish Gyanani, hellyale, Jenism, Lohitha Palagiri, Mark Perera, Mr. Developer, Ojen, Phrancis, RamenChef, Redithion, Stefan Steiger, Tot Zam, Vikrant, vmaroli
15	Crear base de datos	Emil Rowland
16	CREAR FUNCION	John Odom, Ricardo Pontual
17	CREAR MESA	Aidan, alex9311, Almir Vuk, Ares, CL., drunken_monkey, Dylan Vander Berg, Franck Dernoncourt, H. Pauwelyn, Jojodmo, KIRAN KUMAR MATAM, Matas Vaitkevicius, Prateek
18	CURSOR SQL	Stefan Steiger
19	Diseño de la mesa	Darren Bartrup-Cook
20	Ejemplo de bases de datos y tablas	Abhilash R Vankayala, Arulkumar, Athafoud, bignose, Bostjan, Brad Larson, Christian, CL., Dariusz, Dr. J. Testington, enrico.bacis, Florin Ghita, FlyingPiMonster, forsvarir, Franck Dernoncourt, hairboat, JavaHopper, Jaydles, Jon Ericson, Magisch, Matt, Mureinik, Mzzzzzz, Prateek, rdans, Shiva, tinlyx, Tot Zam, WesleyJohnson
21	Eliminar en cascada	Stefan Steiger
22	Encontrar duplicados en un subconjunto de columnas con detalles	Darrel Lee, mnoronha
23	Esquema de información	Hack-R
24	EXCEPTO	LCIII
25	Explique y describa	Simulant
26	Expresiones de mesa comunes	CL., Daniel, dd4711, fuzzy_logic, Gidil, Luis Lema, ninesided, Peter K, Phrancis, Sibeesh Venu
27	Filtrar los resultados usando WHERE y HAVING.	Arulkumar, Bostjan, CL., Community, Franck Dernoncourt, H. Pauwelyn, Jon Chan, Jon Ericson, juergen d, Matas Vaitkevicius, Mureinik, Phrancis, Tot Zam
28	Funciones (Agregado)	ashja99, CL., Florin Ghita, Ian Kenney, Imran Ali Khan, Jon Chan, juergen d, KIRAN KUMAR MATAM, Mark Stewart, Maverick, Nathan, omini data, Peter K, Reboot, Tot Zam, William Ledbetter, winseybash, Алексей Неудачин

29	Funciones (analíticas)	CL., omini data
30	Funciones (escalar / fila única)	CL., Kewin Björk Nielsen, Mark Stewart
31	Funciones de cadena	ɐlǝx, Allan S. Hansen, Arthur D, Arulkumar, Batsu, Chris, CL., Damon Smithies, Franck Dernoncourt, Golden Gate, hatchet, Imran Ali Khan, IncrediApp, Jaydip Jadhav, Jones Joseph, Kewin Björk Nielsen, Leigh Riffel, Matas Vaitkevicius, Mateusz Piotrowski, Neria Nachum, Phrancis, RamenChef, Robert Columbia, vmaroli, ypercube
32	Funciones de ventana	Arkh, beercohol, bhs, Gidil, Jerry Jeremiah, Mureinik, mustaccio
33	Gatillos	Daryl, IncrediApp
34	Grupo SQL por vs distinto	carlosb
35	Identificador	Andreas, CL.
36	Índices	a1ex07, Almir Vuk, carlosb, CL., David Manheim, FlyingPiMonster, forsvarir, Franck Dernoncourt, Horaciux, Jenism, KIRAN KUMAR MATAM, mauris, Parado, Paulo Freitas , Ryan
37	INSERTAR	Ameya Deshpande, CL., Daniel Langemann, Dipesh Poudel, inquisitive_mind, KIRAN KUMAR MATAM, rajarshig, Tot Zam, zplizzi
38	Inyección SQL	120196, CL., Clomp, Community, Epodax, Knickerless-Noggins, Stefan Steiger
39	LA CLÁUSULA EXISTA	Blag, Özgür Öztürk
40	Limpiar código en SQL	CL., Stivan
41	Llaves extranjeras	CL., Harjot, Yehuda Shapira
42	Llaves primarias	Andrea Montanari, CL., FlyingPiMonster, KjetilNordin
43	Mesa plegable	CL., Joel, KIRAN KUMAR MATAM, Stu
44	NULO	Bart Schuijt, CL., dd4711, Devmati Wadikar, Phrancis, Saroj Sasmal, StanislavL, walid, ypercube

45	Numero de fila	CL., Phrancis, user1221533
46	Operadores AND & OR	guiguiblitz
47	Orden de Ejecución	a1ex07, Gallus, Ryan Rockey, ypercube
48	ORDEN POR	Andi Mohr, CL., Cristian Abelleira, Jaydles, mithra chintha, nazark, Özgür Öztürk, Parado, Phrancis, Wolfgang
49	OTORGAR y REVERTIR	RamenChef, user2314737
50	Procedimientos almacenados	brichins, John Odom, Lamak, Ryan
51	Puntos de vista	Amir978, CL., Florin Ghita
52	Secuencia	John Smith
53	SELECCIONAR	Abhilash R Vankayala, aholmes, Alok Singh, Amnon, Andrii Abramov, apomene, Arpit Solanki, Arulkumar, AstraSerg, Brent Oliver, Charlie West, Chris, Christian Sagmüller, Christos, CL., controller, dariru, Daryl, David Pine, David Spillett, day_dreamer , Dean Parker, DeepSpace, Dipesh Poudel, Dror, Durgpal Singh , Epodax, Eric VB, FH-Inway, Florin Ghita, FlyingPiMonster, Franck Dernoncourt, geeksal, George Bailey, Hari K M, HoangHieu, iliketocode, Imran Ali Khan, Inca, Jared Hooper, Jaydles, John Odom, John Slegers, Jojodmo, JonH, Kapep, KartikKannapur, Lankymart, Mark Iannucci, Mark Perera, Mark Wojciechowicz, Matas Vaitkevicius, Matt, Matt S, Mattew Whitt, Matthew Moisen, MegaTom, Mihai-Daniel Virna, Mureinik, mustaccio, mxmissile, Oded, Ojen, onedaywhen, Paul Bambury, penderi, Peter Gordon, Prateek, Praveen Tiwari, Přemysl Šťastný, Preuk, Racil Hilan, Robert Columbia, Ronnie Wang, Ryan, Saroj Sasmal, Shiva, SommerEngineering, sqluser, stark, sunkuet02, ThisIsImpossible, Timothy, user1336087, user1605665, waqasahmed, wintersolider, WMios, xQbert, Yury Fedorov, Zahiro Mor, zedfoxus
54	Sinónimos	Daryl
55	SKIP TAKE (Paginación)	CL., Karl Blacquiere, Matas Vaitkevicius, RamenChef
56	Subconsultas	CL., dasblinkenlight, KIRAN KUMAR MATAM, Nunie123, Phrancis, RamenChef, tinlyx
57	Tipos de datos	bluefeet, Jared Hooper, John Odom, Jon Chan, JonMark Perry,

		Phrancis
58	TRATA DE ATRAPARLO	Uberzen1
59	TRUNCAR	Abhilash R Vankayala, CL., Cristian Abelleira, DaImTo, Hynek Bernard, inquisitive_mind, KIRAN KUMAR MATAM, Paul Bambury, ss005
60	UNION / UNION ALL	Andrea, Athafoud, Daniel Langemann, Jason W, Jim, Joe Taras , KIRAN KUMAR MATAM, Lankymart, Mihai-Daniel Virna, sunkuet02
61	UNIR	Abhilash R Vankayala, CL., Kyle Hale, SQLFox, Zoyd
62	UNIRSE	A_Arnold, Akshay Anand, Andy G, bignose, Branko Dimitrijevic, Casper Spruit, CL., Daniel Langemann, Darren Bartrup-Cook, Dipesh Poudel, enrico.bacis, Florin Ghita, forsvarir, Franck Dernoncourt, hairboat, Hari K M, HK1, HLGEM, inquisitive_mind , John C, John Odom, John Slegers, Mark Iannucci, Marvin, Mureinik, Phrancis, raholling, Raidri, Saroj Sasmal, Stefan Steiger, sunkuet02, Tot Zam, xenodevil, ypercube, Рахул Маквана
63	Vistas materializadas	dmfay
64	XML	Steven