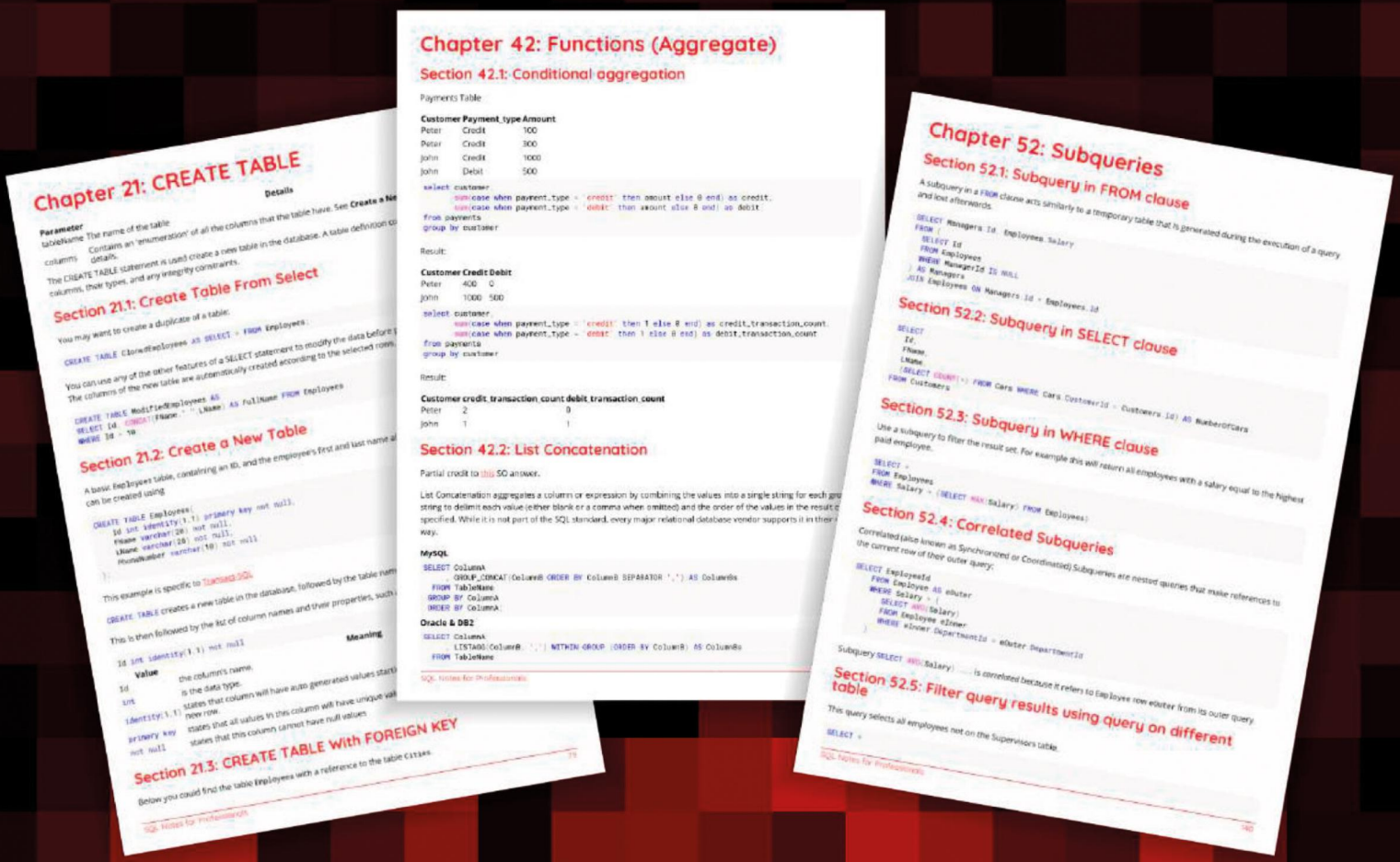
SQL

Apuntes para Profesionales



Traducido por:

rortegag

100+ páginas

de consejos y trucos profesionales

GoalKicker.com
Free Programming Books

Contenidos

Acerca de

Este libro ha sido traducido por rortegag.com

Si desea descargar el libro original, puede descargarlo desde:

https://goalkicker.com/SQLBook/

Si desea contribuir con una donación, hazlo desde:

https://www.buymeacoffee.com/GoalKickerBooks

Por favor, siéntase libre de compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

https://goalkicker.com/SQLBook/

Envíe sus comentarios y correcciones a web@petercv.com

Capítulo 1: Introducción a SQL

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

Sección 1.1: Visión general

El lenguaje de consulta estructurado (SQL) es un lenguaje de programación específico diseñado para gestionar datos almacenados en un sistema de gestión de bases de datos relacionales (RDBMS). También pueden utilizarse lenguajes similares a SQL en sistemas de gestión de flujo de datos relacionales (RDSMS) o en bases de datos "no solo SQL" (NoSQL).

SQL se compone de 3 grandes sublenguajes:

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 de los datos de la base de datos;
3. Lenguaje de control de datos (DCL): para controlar el acceso a los datos almacenados en la base de datos.

Artículo sobre SQL en Wikipedia

Las principales operaciones DML son Crear, Leer, Actualizar y Eliminar (abreviado CRUD), que se realizan mediante las sentencias `INSERT`, `SELECT`, `UPDATE` y `DELETE`.

También existe una sentencia `MERGE` (añadida recientemente) que puede realizar las 3 operaciones de escritura (`INSERT`, `UPDATE`, `DELETE`).

Artículo CRUD en Wikipedia

Muchas bases de datos SQL se implementan como sistemas cliente/servidor; el término "servidor SQL" describe una base de datos de este tipo. Al mismo tiempo, Microsoft fabrica una base de datos que recibe el nombre de "SQL Server". Aunque esa base de datos habla un dialecto de SQL, la información específica de esa base de datos no es el tema de esta etiqueta, sino que pertenece a la documentación de SQL Server.

Capítulo 2: Identificadores

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

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

Sección 2.1: Identificadores sin citar

Los identificadores no entrecomillados pueden utilizar letras (`a`-`z`), dígitos (`0`-`9`) y guiones bajos (`_`), y deben empezar por una letra.

Dependiendo de la implementación de SQL, y/o de la configuración de la base de datos, se pueden permitir otros caracteres, algunos incluso como 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 entrecomillados no distinguen entre mayúsculas y minúsculas. La forma en que se gestionan 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 distinguir entre 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: Se convierte a mayúsculas y se trata como un identificador entre comillas.
- PostgreSQL: Se convierte a minúsculas y se trata como un identificador entre comillas.
- SQLite: Preservación de mayúsculas y minúsculas; insensibilidad a mayúsculas y minúsculas sólo para caracteres ASCII.

Capítulo 3: Tipos de datos

Sección 3.1: DECIMAL y NUMERIC

Precisión fija y escala de números decimales. `DECIMAL` y `NUMERIC` son funcionalmente equivalentes.

Sintaxis:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Ejemplos:

```
SELECT CAST(123 AS DECIMAL(5,2)) --devuelve 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --devuelve 12345.12000
```

Sección 3.2: FLOAT y REAL

Tipos de datos numéricos aproximados para utilizar con datos numéricos en coma flotante.

```
SELECT CAST( PI() AS FLOAT) --devuelve 3.14159265358979
SELECT CAST( PI() AS REAL) --devuelve 3.141593
```

Sección 3.3: Enteros

Tipos de datos de número exacto que utilizan datos enteros.

Tipo de datos	Rango	Almacenamiento
`BIGINT`	-2^63 (-9,223,372,036,854,775,808) a 2^63-1(9,223,372,036,854,775,807)	8 Bytes
`INT`	-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

Sección 3.4: MONEY y SMALLMONEY

Tipos de datos que representan valores monetarios o de divisas.

Tipo de datos	Rango	Almacenamiento
`MONEY`	-922,337,203,685,477.5808 a 922,337,203,685,477.5807	8 bytes
`SMALLMONEY`	-214,748.3648 a 214,748.3647	4 bytes

Sección 3.5: BINARY y VARBINARY

Tipos de datos binarios de longitud fija o variable.

Sintaxis:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

`n_bytes` puede ser cualquier número entre 1 y 8000 bytes. `max` indica que el espacio máximo de almacenamiento es 2^31-1.

Ejemplos:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x0000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

Sección 3.6: 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       ' (con espacios a la derecha)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (sin relleno debido al carácter variable)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10)) -- 'ABCDEFGHIJ' (truncado a 10 caracteres)
```

Sección 3.7: NCHAR y NVARCHAR

Tipos de datos de cadena de caracteres UNICODE de longitud fija o variable.

Sintaxis:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Utilice `MAX` para cadenas de caracteres muy largas que pueden superar los 8000 caracteres.

Sección 3.8: UNIQUEIDENTIFIER

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'
```

Capítulo 4: NULL

`NULL` en SQL, así como en programación en general, significa literalmente “nada”. En SQL, es más fácil entenderlo como “la ausencia de cualquier valor”.

Es importante distinguirlo de los valores aparentemente vacíos, como la cadena de caracteres vacía `''` o el número `0`, ninguno de los cuales es realmente `NULL`.

También es importante tener cuidado de no encerrar `NULL` entre comillas, como `'NULL'`, que está permitido en columnas que aceptan texto, pero no es `NULL` y puede provocar errores y conjuntos de datos incorrectos.

Sección 4.1: Filtrado de NULL en las consultas

La sintaxis para filtrar por `NULL` (es decir, la ausencia de un valor) en los bloques `WHERE` es ligeramente diferente a la de filtrar por 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, el uso de los operadores de igualdad `= NULL` o `<> NULL` (o `!= NULL`) siempre producirá el valor verdadero de `UNKNOWN`, que será rechazado por `WHERE`.

`WHERE` filtra todas las filas en las que la condición es `FALSE` o `UKNOWN` y mantiene sólo las filas en las que la condición es `TRUE`.

Sección 4.2: Columnas anulables en tablas

Al crear tablas es posible declarar una columna como anulable o no anulable.

```
CREATE TABLE MyTable
(
     MyCol1 INT NOT NULL, -- no anulable
     MyCol2 INT NULL -- anulable
) ;
```

Por defecto, todas las columnas (excepto las de la restricción de clave primaria) son anulables a menos que establezcamos explícitamente la restricción `NOT NULL`.

Si se intenta asignar `NULL` a una columna no anulable, se producirá un error.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL); -- funciona bien

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2);
-- no se puede insertar
-- el valor NULL en la columna 'MyCol1', tabla 'MyTable';
-- no admite nulos. INSERT falla.
```

Sección 4.3: Actualización de campos a NULL

Establecer un campo como `NULL` funciona exactamente igual que con cualquier otro valor:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Sección 4.4: Inserción de filas con campos NULL

Por ejemplo, insertar un empleado sin número de teléfono y sin jefe en la tabla de ejemplo `Employees`:

```
INSERT INTO Employees (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22');
```

Capítulo 5: Ejemplos de bases de datos y tablas

Sección 5.1: Base de datos de talleres

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

Un ejemplo vivo: SQL fiddle

Relaciones entre tablas

- Cada Department puede tener 0 o más Employees
- Cada Employee puede tener 0 o 1 Manager
- Cada Customer puede tener 0 o más Cars

Departments

Id	Name
1	HR
2	Sales
3	Tech

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');
```

Employees

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
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

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');
```

Customers

Id	FName	LName	Email	PhoneNumber	PreferredContact
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

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');
```

Cars

Id	CustomerId	EmployeeId	Model	Status	Total Cost
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

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');
```

Sección 5.2: Base de datos de la biblioteca

En este ejemplo de base de datos para una biblioteca, tenemos las tablas *Authors*, *Books* y *BooksAuthors*.

Un ejemplo vivo: SQL fiddle

Authors y *Books* se conocen como **tablas base**, ya que contienen la definición de las columnas y los datos de las entidades reales del modelo relacional. *BooksAuthors* se conoce como **tabla de relación**, ya que esta tabla define la relación entre las tablas *Books* y *Authors*.

Relaciones entre tablas

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

Authors

(*ver tabla*)

Id	Name	Country
1	J.D. Salinger	USA
2	F. Scott. Fitzgerald	USA
3	Jane Austen	UK
4	Scott Hanselman	USA
5	Jason N. Gaylord	USA
6	Pranav Rastogi	India
7	Todd Miranda	USA
8	Christian Wenz	USA

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');
```

Books

(ver tabla)

Id	Title
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

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');
```

BooksAuthors

(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 directo):

```
SELECT * FROM Authors;
```

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

```
SELECT * FROM Books;
```

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

```
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;
```

Sección 5.3: Tabla de países

En este ejemplo, tenemos una tabla de **Countries**. Una tabla de países tiene muchos usos, especialmente en aplicaciones financieras relacionadas con divisas y tipos de cambio.

Un ejemplo vivo: SQL fiddle

Algunas aplicaciones de software de datos de mercado, como Bloomberg y Reuters, le exigen que proporcione a su API un código de país de 2 o 3 caracteres junto con el código de moneda. De ahí que esta tabla de ejemplo contenga tanto la columna del código ISO de 2 caracteres como la columna del código ISO3 de 3 caracteres.

Countries

(ver tabla)

Id	ISO	ISO3	ISONumeric	CountryName	Capital	ContinentCode	CurrencyCode
1	AU	AUS	36	Australia	Canberra	OC	AUD
2	DE	DEU	276	Germany	Berlin	EU	EUR
2	IN	IND	356	India	New Delhi	AS	INR
3	LA	LAO	418	Laos	Vientiane	AS	LAK
4	US	USA	840	United States	Washington	NA	USD
5	ZW	ZWE	716	Zimbabwe	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');
```

Capítulo 6: SELECT

La sentencia `SELECT` es el núcleo de la mayoría de las consultas SQL. Define el conjunto de resultados que debe devolver la consulta y casi siempre se utiliza junto con la cláusula `FROM`, que define qué parte o partes de la base de datos deben consultarse.

Sección 6.1: Uso del carácter comodín para seleccionar todas las columnas de una consulta

Considere una base de datos con las dos tablas siguientes.

Tabla de Employees:

Id	FName	LName	DeptId
1	James	Smith	3
2	John	Johnson	4

Tabla de Departments:

Id	Name
1	Sales
2	Marketing
3	Finance
4	IT

Sentencia `SELECT` simple

`*` es el **carácter comodín** utilizado para seleccionar todas las columnas disponibles en una tabla.

Cuando se utiliza como sustituto de nombres de columna explícitos, devuelve todas las columnas de todas las tablas de las que una consulta está seleccionando. Este efecto se aplica a **todas las tablas** a las que la consulta accede 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`:

Id	FName	LName	DeptId
1	James	Smith	3
2	John	Johnson	4

Notación del punto

Para seleccionar todos los valores de una tabla específica, se puede aplicar el carácter comodín a la tabla con *notación del punto*.

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 de la tabla `Employee`, seguido sólo por el campo `Name` de la tabla `Departments`:

Id	FName	LName	DeptId	Name
1	James	Smith	3	Finance
2	John	Johnson	4	IT

Advertencias contra el uso

En general, se aconseja evitar el uso de `*` en el código de producción siempre que sea posible, ya que puede causar una serie de problemas potenciales, incluyendo:

1. Exceso de IO, carga de red, uso de memoria, etc., debido a que el motor de la base de datos lee datos que no se necesitan y los transmite al código del front-end. Esto es especialmente preocupante cuando puede haber campos de gran tamaño, como los utilizados para almacenar notas largas o archivos adjuntos.
2. Más carga de E/S si la base de datos necesita enviar los resultados internos al disco como parte del procesamiento de una consulta más compleja que `SELECT <columns> FROM <table>`.
3. Procesamiento extra (y/o incluso más IO) si algunas de las columnas no son necesarias:
    - columnas computadas en las bases de datos que las admiten
    - en el caso de la selección a partir de una vista, columnas de una tabla/vista que el optimizador de consultas podría optimizar de otro modo
4. La posibilidad de que se produzcan errores inesperados si posteriormente se añaden columnas a tablas y vistas que den lugar a nombres de columna ambiguos. Por ejemplo `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` - si se añade una columna llamada `displayname` a la tabla `orders` para permitir a los usuarios dar a sus pedidos nombres significativos para futuras referencias entonces el nombre de la columna aparecerá dos veces en la salida por lo que la cláusula `ORDER BY` será ambigua lo que puede causar errores («nombre de columna ambiguo» en versiones recientes de MS SQL Server), y si no en este ejemplo su código de aplicación podría empezar a mostrar el nombre del pedido donde se pretende el nombre de la persona porque la nueva columna es la primera de ese nombre devuelta, y así sucesivamente.

¿Cuándo se puede utilizar `*` teniendo en cuenta la advertencia anterior?

Aunque es mejor evitarlo en el código de producción, el uso de `*` está bien como abreviatura cuando se realizan consultas manuales contra la base de datos para la investigación o el trabajo de prototipo.

A veces, las decisiones de diseño de su aplicación lo hacen inevitable (en tales circunstancias, prefiera `tablealias.*` a sólo `*` siempre que 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 estamos devolviendo ningún dato de B. Por lo tanto, no es necesaria una unión y el motor sabe que no se van a devolver valores de B, por lo que el uso de `*` no afecta al rendimiento. Del mismo modo, `COUNT(*)` está bien, ya que tampoco devuelve ninguna de las columnas, por lo que sólo necesita leer y procesar las que se utilizan para filtrar.

Sección 6.2: SELECT con alias de columna

Los alias de columna se utilizan principalmente para acortar el código y hacer más legibles los nombres de las columnas.

El código se acorta, ya que se pueden evitar los nombres de tabla largos y la identificación innecesaria de columnas (*por ejemplo, puede haber 2 ID en la tabla, pero en la sentencia sólo se utiliza uno*). Junto con los alias de tabla, esto permite utilizar nombres descriptivos más largos en la estructura de la base de datos y mantener concisas las consultas sobre dicha estructura.

Además, a veces son *necesarios*, por ejemplo, en las vistas, para dar nombre a los resultados calculados.

Todas las versiones de SQL

Los alias pueden crearse en todas las versiones de SQL utilizando comillas dobles (`"`).

```
SELECT FName AS "First Name", MName AS "Middle Name", LName AS "Last Name" FROM Employees
```

Diferentes versiones de SQL

Puede utilizar 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 tendrán como resultado:

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

Esta sentencia devolverá las columnas `FName` y `LName` con un nombre dado (un alias). Esto se consigue utilizando el operador `AS` seguido del alias, o simplemente escribiendo alias directamente después del nombre de la columna. Esto significa que la siguiente consulta tiene el mismo resultado que la anterior.

```
SELECT FName "First Name", MName "Middle Name", LName "Last Name" FROM Employees
```

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

Sin embargo, la versión explícita (es decir, utilizando el operador `AS`) es más legible.

Si el alias tiene una sola palabra que no es una palabra reservada, podemos escribirla sin comillas simples, dobles o corchetes:

```
SELECT FName AS FirstName, LName AS LastName FROM Employees
```

FirstName	LastName
James	Smith
John	Johnson
Michael	Williams

Otra variante disponible en MS SQL Server, entre otros, 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 tendrán como resultado:

FullName	Addr1	Addr2
James Smith	123 AnyStreet	TownVille
John Johnson	668 MyRoad	Anytown
Michael Williams	999 High End	Dr Williamsburgh

Para algunos, utilizar `=` en lugar de `As` es más fácil de leer, aunque muchos desaconsejan este formato, principalmente porque no es estándar y, por tanto, no todas las bases de datos lo admiten. Puede causar confusión con otros usos del carácter `=`.

Todas las versiones de SQL

Además, si necesitas utilizar palabras reservadas, puedes 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 palabras clave en MSSQL con todos los enfoques diferentes:

```
SELECT FName AS "SELECT", MName AS 'FROM', LName AS [WHERE] FROM Employees
```

SELECT	FROM	WHERE
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

Además, un alias de columna puede utilizarse en cualquiera de las cláusulas finales de la misma consulta, como un `ORDER BY`:

```
SELECT FName AS FirstName, LName AS LastName FROM Employees ORDER BY LastName DESC
```

Sin embargo, *no podrá utilizar*

```
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 provocará numerosos errores en la ejecución.

Sección 6.3: Seleccionar columnas individuales

```
SELECT PhoneNumber, Email, PreferredContact FROM Customers
```

Esta sentencia 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:

PhoneNumber	Email	PreferredContact
3347927472	william.jones@example.com	PHONE
2137921892	dmiller@example.net	EMAIL
NULL	richard0123@example.com	EMAIL

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 `Orders` será devuelto como una columna llamada `OrderId`. Consulte Seleccionar con alias de columna para obtener más información.

Para evitar el uso de nombres de tabla largos, puede utilizar alias de tabla. De este modo se evita tener que escribir nombres de tabla largos para cada campo que se selecciona en las uniones. Si realiza una autounión (una unión entre dos instancias de la *misma* tabla), debe utilizar alias de tabla para distinguir las tablas. Podemos escribir un alias de tabla como `Customers c` o `Customers AS c`. Aquí `c` funciona como un alias para `Customers` y podemos seleccionar digamos `Email` así: `c.Email`.

```
SELECT c.PhoneNumber, c.Email, c.PreferredContact, o.Id AS OrderId FROM Customers c
     LEFT JOIN Orders o ON o.CustomerId = c.Id
```

Sección 6.4: Selección de un número determinado 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 sólo está soportado en versiones recientes de algunos RDMS. En otros sistemas se proporciona una sintaxis no estándar específica del proveedor. 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 omitir filas antes de obtener 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 debe utilizarse 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 récords.

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 del vendedor:

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 parte de la tabla, mientras que `ROWNUM` funciona como parte de la cláusula `WHERE` por lo que, si no existen otras condiciones en el número especificado de filas al principio de la tabla, obtendrá cero resultados cuando podría haber otros por encontrar.

Sección 6.5: Seleccionar con condición

La sintaxis básica de `SELECT` con cláusula `WHERE` es:

```
SELECT column1, column2, column FROM table_name WHERE [condition]
```

La *[condición]* puede ser cualquier expresión SQL, especificada mediante operadores de comparación o lógicos como `>`, `<`, `=`, `<>`, `>=`, `<=`, `LIKE`, `NOT`, `IN`, `BETWEEN`, etc.

La siguiente sentencia devuelve todas las columnas de la tabla '`Cars`' donde la columna de estado es `'READY'`:

```
SELECT * FROM Cars WHERE status = 'READY'
```

Véase `WHERE` y `HAVING` para más ejemplos.

Sección 6.6: Seleccionar con CASE

Cuando los resultados necesitan que se aplique cierta lógica 'sobre la marcha' se puede utilizar la sentencia `CASE` para implementarla.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold FROM TableName
```

también pueden encadenarse

```
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
```

Sección 6.7: Seleccionar columnas con nombres de palabras clave reservadas

Cuando un nombre de columna coincide con una palabra clave reservada, SQL estándar exige que se encierre entre comillas dobles:

```
SELECT "ORDER", ID FROM ORDERS
```

Tenga en cuenta que el nombre de la columna distingue entre mayúsculas y minúsculas.

Algunos SGBD tienen formas propias de entrecomillar nombres. Por ejemplo, SQL Server utiliza corchetes para este propósito:

```
SELECT [Order], ID FROM ORDERS
```

mientras que MySQL (y MariaDB) utilizan por defecto backticks:

```
SELECT `Order`, id FROM orders
```

Sección 6.8: Selección 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 situaciones en las que varias tablas tienen el mismo nombre de campo y es necesario especificar de qué tabla se quieren obtener 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 defina un alias, ya no podrá utilizar el nombre canónico de la tabla, es decir,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName FROM Employees e
      JOIN Managers m ON e.ManagerId = m.Id
```

arrojaría un error.

Cabe señalar que los alias de tabla -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 introduciendo `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 en el sentido de que las tablas se unen en columnas con nombres diferentes (`Id` y `ManagerId`), pero se supone que no deben unirse en las columnas con el mismo nombre (`LName`, `FName`), lo que requiere que se cambie el nombre de las columnas 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 debe declararse una variable de alias/rango para la tabla desviada (de lo contrario, SQL arrojará un error), nunca tiene sentido utilizarla realmente en la consulta.

Sección 6.9: Seleccionar con más de 1 condición

La palabra clave `AND` se utiliza para añadir más condiciones a la consulta.

Name	Age	Gender
Sam	18	M
John	21	M
Bob	22	M
Mary	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

Esto devolverá:

Name
John
Bob

utilizando la palabra clave `OR`

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

Esto volverá:

Name
Sam
John
Bob

Estas palabras clave pueden combinarse 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:

Name
Sam
Mary

Sección 6.10: Seleccionar sin bloquear la tabla

A veces, cuando las tablas se utilizan principalmente (o sólo) para lecturas, la indexación ya no ayuda y cada pequeño bit cuenta, uno podría utilizar `SELECT`s sin `LOCK` para mejorar el rendimiento.

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracle

```
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 una mesa en la que se están modificando los registros, los resultados pueden ser impredecibles.

Sección 6.11: Seleccionar con funciones agregadas

Media

La función de agregado `AVG()` devolverá la media de los valores seleccionados.

```
SELECT AVG(Salary) FROM Employees
```

Las funciones agregadas también pueden combinarse con la cláusula `WHERE`.

```
SELECT AVG(Salary) FROM Employees WHERE DepartmentId = 1
```

Las funciones agregadas también pueden combinarse con la cláusula agrupar por.

Si el empleado esta categorizado con múltiples departamentos y queremos encontrar el salario promedio para cada departamento entonces podemos usar la siguiente consulta.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Mínimo

La función agregada `MIN()` devolverá el mínimo de los valores seleccionados.

```
SELECT MIN(Salary) FROM Employees
```

Máximo

La función agregada `MAX()` devolverá el máximo de los valores seleccionados.

```
SELECT MAX(Salary) FROM Employees
```

Cuenta

La función agregada `COUNT()` devolverá el recuento de valores seleccionados.

```
SELECT Count(*) FROM Employees
```

También puede combinarse con condiciones `WHERE` para obtener el recuento de filas que satisfacen condiciones específicas.

```
SELECT Count(*) FROM Employees WHERE ManagerId IS NOT NULL
```

También se pueden especificar columnas concretas para obtener el número de valores de la columna. Tenga en cuenta que los valores `NULL` no se cuentan.

```
SELECT Count(ManagerId) FROM Employees
```

`Count` también se puede combinar con la palabra clave `DISTINCT` para obtener un recuento `DISTINCT`.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Suma

La función de agregado `SUM()` devuelve la suma de los valores seleccionados para todas las filas.

```
SELECT SUM(Salary) FROM Employees
```

Sección 6.12: Seleccionar con 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, `value IN ( <value list> )` es una abreviatura de disyunción (`OR` lógico).

Sección 6.13: Obtener el resultado agregado de los grupos de filas

Contar filas basándose en el valor de una columna específica:

```
SELECT category, COUNT(*) AS item_count FROM item GROUP BY category;
```

Obtener ingresos medios por departamento:

```
SELECT department, AVG(income) FROM employees GROUP BY department;
```

Lo importante es seleccionar sólo las columnas especificadas en la cláusula `GROUP BY` o utilizadas con funciones de agregación.

La cláusula `WHERE` también se puede utilizar con `GROUP BY`, pero `WHERE` filtra los registros antes de agruparlos:

```
SELECT department, AVG(income) FROM employees WHERE department <> 'ACCOUNTING'
     GROUP BY department;
```

Si necesita filtrar los resultados una vez realizada la agrupación, por ejemplo, para ver sólo los departamentos cuyos ingresos medios sean superiores a 1000, deberá utilizar la cláusula `HAVING`:

```
SELECT department, AVG(income) FROM employees WHERE department <> 'ACCOUNTING'
     GROUP BY department HAVING avg(income) > 1000;
```

Sección 6.14: Selección con resultados ordenados

```
SELECT * FROM Employees ORDER BY LName
```

Esta sentencia devolverá todas las columnas de la tabla `Employees`.

Id	FName	LName	PhoneNumber
2	John	Johnson	2468101214
1	James	Smith	1234567890
3	Michael	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

O

```
SELECT * FROM Employees ORDER BY LName ASC
```

Esta declaración cambia la dirección de la clasificación.

También se pueden especificar varias columnas de clasificación. Por ejemplo:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

Este ejemplo ordenará los resultados primero por `LName` y después, para los registros que tengan el mismo `LName`, los ordenará por `FName`. Así obtendrá un resultado similar al que encontraría en una guía telefónica.

Para no tener que volver a escribir el nombre de la columna en la cláusula `ORDER BY`, es posible utilizar en su lugar el número de la columna. Tenga en cuenta que los números de columna empiezan por 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

También puede incluir 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 los resultados para que todos los registros con el `LName` de "Jones" aparezcan al principio.

Sección 6.15: Seleccionar con null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

La selección con nulos tiene una sintaxis diferente. No utilice `=`, utilice `IS NULL` o `IS NOT NULL` en su lugar.

Sección 6.16: Seleccionar distinto (sólo valores únicos)

```
SELECT DISTINCT ContinentCode FROM Countries;
```

Esta consulta devolverá todos los valores `DISTINCT` (únicos, diferentes) de la columna `ContinentCode` de la tabla `Countries`.

ContinentCode
OC
EU
AS
NA
AF

Demostración de SQLFiddle

Sección 6.17: Seleccionar filas de varias tablas

```
SELECT * FROM table1, table2
```

```
SELECT table1.column1, table1.column2, table2.column1 FROM table1, table2
```

Esto se llama producto cruzado en SQL y es lo mismo que el producto cruzado en conjuntos.

Estas sentencias devuelven las columnas seleccionadas de varias tablas en una sola consulta.

No existe ninguna relación específica entre las columnas devueltas por cada tabla.

Capítulo 7: GROUP BY

Los resultados de una consulta `SELECT` pueden agruparse por una o varias columnas mediante la sentencia `GROUP BY`: todos los resultados con el mismo valor en las columnas agrupadas se agregan. Esto genera una tabla de resultados parciales, en lugar de un único resultado. `GROUP BY` puede utilizarse junto con funciones de agregación mediante la sentencia `HAVING` para definir cómo se agregan las columnas no agrupadas.

Sección 7.1: Ejemplo básico de GROUP BY

Puede ser más fácil si usted piensa en `GROUP BY` como "para cada uno" en aras de la explicación. La siguiente consulta:

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

está diciendo:

"Dame la suma de `MonthlySalary` **para cada** `EmpID`"

Así que si tu mesa tuviera este aspecto:

EmpID	MonthlySalary
1	200
2	300

Resultado:

1	200
2	300

Suma no parecería hacer nada porque la suma de un número es ese número. Por otro lado, si se viera así:

EmpID	MonthlySalary
1	200
1	300
2	300

Resultado:

1	500
2	300

Entonces sí, porque hay dos `EmpID 1` que sumar.

Sección 7.2: Filtrar resultados GROUP BY mediante 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 `Library`.

Ejemplos:

Devuelve todos los autores que escribieron más de un libro (ejemplo 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 -- es igual a HAVING LibrosEscritos > 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 -- es igual a HAVING NúmeroDeAutores > 3 ;
```

Sección 7.3: USE GROUP BY para COUNT el número de filas por cada entrada única en una columna dada

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

Dada esta tabla, "`Westerosians`":

Name	GreatHouseAllegience
Arya	Stark
Cercei	Lannister
Myrcella	Lannister
Yara	Greyjoy
Catelyn	Stark
Sansa	Stark

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

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

vuelve...

Number_of_Westerosians
6

Pero añadiendo `GROUP BY`, podemos `CONTAR` los usuarios para cada valor de una columna determinada, para devolver el número de personas de una Casa Grande determinada, digamos:

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

vuelve...

House	Number_of_Westerosians
Stark	3
Lannister	1
Greyjoy	2

Es habitual combinar `GROUP BY` con `ORDER BY` para ordenar los resultados por categoría mayor o menor:

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

vuelve...

House	Number_of_Westerosians
Stark	3
Lannister	2
Greyjoy	1

Sección 7.4: Agregación ROLAP (minería de datos)

Descripción

El estándar SQL proporciona dos operadores de agregación adicionales. Éstos 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 argumento de la cláusula.
- `with roll up` que proporciona los agregados obtenidos al considerar los atributos en orden de izquierda a derecha frente a como aparecen en el argumento de la cláusula.

Versiones estándar de SQL que admiten estas funciones: 1999, 2003, 2006, 2008, 2011.

Ejemplos

Considera esta tabla:

Food	Brand	Total_amount
Pasta	Brand1	100
Pasta	Brand2	250
Pizza	Brand2	300

`with cube`

```
SELECT Food,Brand,Total_amount FROM Table GROUP BY Food,Brand,Total_amount with cube
```

Food	Brand	Total_amount
Pasta	Brand1	100
Pasta	Brand2	250
Pasta	ALL	350
Pizza	Brand2	300
Pizza	ALL	300
ALL	Brand1	100
ALL	Brand2	550
ALL	ALL	650

`with roll up`

```
SELECT Food,Brand,Total_amount FROM Table GROUP BY Food,Brand,Total_amount with roll up
```

Food	Brand	Total_amount
Pasta	Brand1	100
Pasta	Brand2	250
Pizza	Brand2	300
Pasta	ALL	350
Pizza	ALL	300
ALL	ALL	650

Capítulo 8: ORDER BY

Sección 8.1: Ordenación por número de columna (en lugar de por nombre)

Puede utilizar el número de una columna (donde la columna más a la izquierda es '1') para indicar en qué columna basar la ordenación, 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.

Contra: En general, esto reducirá la legibilidad de la consulta (queda claro al instante lo que significa "`ORDER BY Reputation`", mientras que "`ORDER BY 14`" requiere contar, probablemente con un dedo en la pantalla).

Esta consulta ordena el resultado por la información en la posición relativa de la columna `3` de la sentencia `SELECT` en lugar de por el nombre de la columna `Reputation`.

```
SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY 3
```

DisplayName	JoinDate	Reputation
Community	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**

Sección 8.2: Utilice ORDER BY con TOP para devolver las x filas superiores en función del valor de una columna

En este ejemplo, podemos utilizar `GROUP BY` no sólo para determinar el orden de las filas devueltas, sino también qué filas se devuelven, ya que estamos utilizando `TOP` para limitar el conjunto de resultados.

Supongamos que queremos obtener los 5 usuarios con mayor reputación de un sitio de preguntas y respuestas popular sin nombre.

Sin `ORDER BY`

Esta consulta devuelve las 5 primeras filas ordenadas por defecto, que en este caso es "`Id`", la primera columna de la tabla (aunque no sea una columna que se muestre en los resultados).

```
SELECT TOP 5 DisplayName, Reputation FROM Users
```

vuelve...

DisplayName	Reputation
Community	1
Geoff Dalgas	12567
Jarrod Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

Con `ORDER BY`

```
SELECT TOP 5 DisplayName, Reputation FROM Users ORDER BY Reputation desc
```

vuelve...

DisplayName	Reputation
JonSkeet	**865023**
Darin Dimitrov	**661741**
BalusC	**650237**
Hans Passant	**625870**
Marc Gravell	**601636**

Observaciones

Algunas versiones de SQL (como MySQL) utilizan una cláusula `LIMIT` al final de un `SELECT`, en lugar de `TOP` al principio, por ejemplo:

```
SELECT DisplayName, Reputation FROM Users ORDER BY Reputation DESC LIMIT 5
```

Sección 8.3: Orden de clasificación personalizado

Para ordenar esta tabla `Employee` por departamento, utilizaría `ORDER BY Department`. Sin embargo, si desea un orden diferente que no sea alfabético, tiene que asignar los valores de `Department` a valores diferentes que ordenen correctamente; esto se puede hacer con una expresión `CASE`:

Name	Department
Hasan	IT
Yusuf	HR
Hillary	HR
Joe	IT
Merry	HR
Ken	Accountant

```
SELECT * FROM Employee ORDER BY CASE Department
     WHEN 'HR' THEN 1 WHEN 'Accountant' THEN 2 ELSE 3 END;
```

Name	Department
Yusuf	**HR**
Hillary	**HR**
Merry	**HR**
Ken	**Accountant**
Hasan	**IT**
Joe	**IT**

Sección 8.4: Ordenar por alias

Debido al orden lógico de procesamiento de las consultas, los alias pueden utilizarse ordenados por.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep FROM Users ORDER BY jd, rep
```

Y puede utilizar el orden relativo de las columnas en la sentencia select. Considere el mismo ejemplo anterior y en lugar de utilizar alias utilizar el orden relativo como para el nombre de pantalla es `1`, para `jd` es `2` y así sucesivamente

```
SELECT DisplayName, JoinDate as jd, Reputation as rep FROM Users ORDER BY 2, 3
```

Sección 8.5: Ordenar por varias columnas

```
SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY JoinDate, Reputation
```

DisplayName	JoinDate	Reputation
Community	**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**

Capítulo 9: Operadores AND & OR

Sección 9.1: Ejemplo AND OR

Tener una tabla

Name	Age	City
Bob	10	Paris
Mat	20	Berlin
MAry	24	Prague

```
SELECT Name FROM TABLE WHERE Age>10 AND City='Prague'
```

Da

Name
Mary

```
SELECT Name FROM TABLE WHERE Age=10 OR City='Prague'
```

Da

Name
Bob
Mary

Capítulo 10: CASE

La expresión `CASE` se utiliza para aplicar la lógica si-entonces.

Sección 10.1: Utilice CASE para COUNT el número de filas de una columna que coinciden con una condición

Caso práctico

`CASE` puede utilizarse junto con `SUM` para obtener un recuento de sólo los elementos que cumplan una condición predefinida. (Esto es similar a `COUNTIF` en Excel).

El truco consiste en devolver resultados binarios que indiquen las coincidencias, de modo que los "`1`" devueltos por las entradas coincidentes puedan sumarse para obtener un recuento del número total de coincidencias.

Dada esta tabla `ItemSales`, digamos que desea conocer el número total de artículos que han sido categorizados como "Expensive":

Id	ItemId	Price	Rating
1	100	34.5	EXPENSIVE
2	145	2.3	CHEAP
3	100	34.5	EXPENSIVE
4	100	34.5	EXPENSIVE
5	145	10	AFFORDABLE

Consulta

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

Resultados:

ItemsCount	ExpensiveItemsCount
5	3

Alternativa:

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

Sección 10.2: Búsqueda CASE en SELECT (Coincide con una expresión booleana)

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

(Esto difiere del caso simple, que sólo puede comprobar 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
```

Id	ItemId	Price	PriceRating
1	100	34.5	EXPENSIVE
2	145	2.3	CHEAP
3	100	34.5	EXPENSIVE
4	100	34.5	EXPENSIVE
5	145	10	AFFORDABLE

Sección 10.3: CASE en una cláusula ORDER BY

Podemos utilizar `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
```

ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
12	New England	Boston	MARKETING	9
15	West	San Francisco	MARKETING	12
9	Midwest	Chicago	SALES	8
14	Mid-Atlantic	New York	SALES	12
5	West	Los Angeles	RESEARCH	11
10	Mid-Atlantic	Philadelphia	RESEARCH	13
4	Midwest	Chicago	INNOVATION	11
2	Midwest	Detroit	HUMAN RESOURCES	9

Sección 10.4: Abreviatura CASE en SELECT

La variante abreviada de `CASE` evalúa una expresión (normalmente una columna) contra una serie de valores. Esta variante es un poco más corta y ahorra repetir la expresión evaluada una y otra vez. No obstante, puede seguir utilizándose la cláusula `ELSE`:

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

Una advertencia. Es importante tener en cuenta que, cuando se utiliza la variante corta, la sentencia completa se evalúa en cada `WHEN`. Por lo tanto, la siguiente sentencia:

```
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()` está siendo llamada 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 omitir todos los casos `WHEN` y dar como resultado `NULL`.

Sección 10.5: Uso de CASE en UPDATE

muestra sobre subidas 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
```

Sección 10.6: Uso de CASE para valores NULL ordenados en último lugar

de este modo, los '0' que representan los valores conocidos se clasifican en primer lugar, los '1' que representan los valores NULL se clasifican en último lugar:

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

ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
10	Mid-Atlantic	Philadelphia	RESEARCH	13
14	Mid-Atlantic	New York	SALES	12
9	Midwest	Chicago	SALES	8
12	New England	Boston	MARKETING	9
5	West	Los Angeles	RESEARCH	11
15	NULL	San Francisco	MARKETING	12
4	NULL	Chicago	INNOVATION	11
2	NULL	Detroit	HUMAN RESOURCES	9

Sección 10.7: CASE 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 utilizar una función `MIN()` o `LEAST()` no agregada para esto (`... ORDER BY MIN(Date1, Date2)`), pero en SQL estándar, tiene que utilizar una expresión `CASE`.

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

Datos de la muestra

Id	Date1	Date2
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

Id	Date1	Date2
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**
4	**2017-01-06**	2017-01-31

Explicación

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

Así que hemos ordenado los registros de `2017-01-01` a `2017-01-06` ascendente y no importa en que una columna `Date1` o `Date2` son esos valores.

Capítulo 11: Operador LIKE

Sección 11.1: Emparejar patrón abierto

El comodín `%` añadido al principio o al final (o a ambos) de una cadena de caracteres permitirá que coincidan `0` o más caracteres antes del principio o después del final del patrón.

El uso de "`%`" en el medio permitirá que coincidan `0` o más caracteres entre las dos partes del patrón.

Vamos a utilizar esta Tabla de `Employees`:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	Smith	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 sentencia busca todos los registros de la tabla `Employees` cuyo `FName` **contenga** la cadena de caracteres 'on'.

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

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

La siguiente sentencia busca todos los registros de `Employees` cuyo `PhoneNumber` **empiece por** la cadena de caracteres '246'.

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

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
1	John	Johnson	2468101214	1	1	400	23-03-2005
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

La siguiente sentencia busca todos los registros de `Employees` cuyo `PhoneNumber` **termine en** '11'.

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

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

Todos los registros donde `FName` **3er carácter** es 'n' de `Employees`.

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

(se utilizan dos guiones bajos antes de "n" para omitir los 2 primeros caracteres)

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Sección 11.2: Coincidencia de un solo carácter

Para ampliar las selecciones de una sentencia en lenguaje de consulta estructurado (SQL-SELECT), pueden utilizarse los caracteres comodines, el signo de porcentaje (`%`) y el guión bajo (`_`).

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

Buscar todos los empleados cuyo `FName` empiece por 'j' y termine por 'n' y tenga exactamente 3 caracteres en `FName`.

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

El carácter `_` (guión bajo) también puede utilizarse más de una vez como comodín para coincidir con 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 utiliza un guión bajo y 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'
```

Sección 11.3: Declaración ESCAPE en la consulta LIKE

Si implementa una búsqueda de texto como consulta `LIKE`, normalmente lo hace así:

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

Sin embargo, (aparte del hecho de que no debería utilizar necesariamente `LIKE` cuando puede utilizar la búsqueda de texto completo) esto crea un problema cuando alguien introduce un texto como “50%” o “a_b”.

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

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

Eso significa que `\` ahora será tratado como carácter ESCAPE. Esto significa que ahora puede anteponer `\` a cada carácter de la cadena que busque, y los resultados empezarán a ser correctos, incluso cuando el usuario introduzca un carácter especial como `%` o `_`.

Por ejemplo

```
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 sólo para fines de demostración. No funcionará en los casos en que un grafema esté formado por varios caracteres (utf-8). Por ejemplo, `string stringToSearch = "Les Mise\u0301rables";` Tendrá que hacer esto para cada grafema, no para cada carácter. No debería utilizar el algoritmo anterior si trabaja con lenguas asiáticas/del este/del sur de Asia. O más bien, si quieres un código correcto para empezar, deberías hacerlo para cada graphemeCluster.

Véase también ReverseString, una entrevista-pregunta de C#.

Sección 11.4: Buscar una serie de caracteres

La siguiente sentencia empareja todos los registros que tienen `FName` que empieza con una letra de la A a la F de la Tabla de `Employees`.

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

Sección 11.5: Coincidir por rango o conjunto

Coincide con cualquier carácter individual dentro del rango especificado (por ejemplo: `[a-f]`) o conjunto (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 coincidiría con "mary" pero no con "gary":

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

El intervalo o conjunto también puede negarse añadiendo el signo `^` antes del intervalo o conjunto:

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

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

Este patrón *no* coincidiría con "mary" pero sí con "gary":

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

Sección 11.6: Caracteres comodín

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

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

`%` - Sustituye a cero o más caracteres

```
Eg: // selecciona todos los clientes cuya ciudad empiece por "Lo".
SELECT * FROM Customers WHERE City LIKE 'Lo%';

// selecciona todos los clientes cuya ciudad contenga el patrón "es".
SELECT * FROM Customers WHERE City LIKE '%es%';
```

`_` - Sustituye a un único carácter

```
Eg:
// selecciona todos los clientes cuya ciudad empiece por cualquier carácter seguido de "erlin".
SELECT * FROM Customers WHERE City LIKE '_erlin';
```

`[charlist]` - Conjuntos y rangos de caracteres a comparar

```
Eg:// selecciona todos los clientes cuya ciudad empiece por "a", "d" o "l".
SELECT * FROM Customers WHERE City LIKE '[adl]%';

// selecciona todos los clientes cuya ciudad empiece por "a", "d" o "l".
SELECT * FROM Customers WHERE City LIKE '[a-c]%';
```

`[^charlist]` - Coincide sólo con un carácter NO especificado entre los corchetes

```
Eg:
// selecciona todos los clientes cuya ciudad empiece por un carácter que no sea "a", "p" o "l".
SELECT * FROM Customers WHERE City LIKE '[^apl]%';

Or

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

Capítulo 12: Cláusula IN

Sección 12.1: Cláusula IN simple

Para obtener registros que tengan **cualquiera** de los `id`s 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
```

Sección 12.2: Uso de la cláusula IN con una subconsulta

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

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

Capítulo 13: Filtrar resultados mediante WHERE y HAVING

Sección 13.1: Utilizar BETWEEN para filtrar los resultados

Los siguientes ejemplos utilizan las bases de datos de ejemplo `Sales` de artículos y `Customers`.

> Nota: El operador `BETWEEN` *es* inclusivo.

Uso del operador `BETWEEN` con números:

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

Esta consulta devolverá todos los registros de `ItemSales` que tengan una cantidad mayor o igual a 10 y menor o igual a 17. Los resultados serán los siguientes:

Id	SaleDate	ItemId	Quantity	Price
1	2013-07-01	100	10	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

Utilización del 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 `SaleDate` mayor o igual que el 11 de julio de 2013 y menor o igual que el 24 de mayo de 2013.

Id	SaleDate	ItemId	Quantity	Price
3	2013-07-11	100	20	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

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

Uso del operador `BETWEEN` con valores de texto:

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

Un ejemplo vivo: SQL fiddle

Esta consulta devolverá todos los clientes cuyo nombre esté alfabéticamente comprendido entre las letras “D” y “L”. En este caso, se mostrarán los clientes nº 1 y nº 3. El cliente nº 2, cuyo nombre empieza por “M”, no se incluirá. El cliente nº 2, cuyo nombre empieza por “M”, no se incluirá.

Id	FName	LName
1	William	Jones
3	Richard	Davis

Sección 13.2: Utilizar HAVING con funciones agregadas

A diferencia de la cláusula `WHERE`, `HAVING` puede utilizarse con funciones de agregación.

> Una función agregada es una función en la que los valores de varias filas se agrupan en función de determinados criterios para formar un único valor con un significado o medida más significativos (Wikipedia).

Las funciones de agregación más comunes son `COUNT()`, `SUM()`, `MIN()` y `MAX()`.

Este ejemplo utiliza la Tabla de `Car` de las Bases de Datos de `Example`.

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

Esta consulta devolverá la cuenta `CustomerId` y `Number of Cars` de cualquier cliente que tenga más de un coche. En este caso, el único cliente que tiene más de un coche es el cliente nº 1.

Los resultados serán los siguientes:

CustomerId	Number of Cars
1	2

Sección 13.3: Cláusula WHERE con valores NULL/NOT NULL

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

Esta sentencia devolverá todos los registros de `Employees` en los que el valor de la columna `ManagerId` sea `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 sentencia devolverá todos los registros de `Employees` en los que 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` a `WHERE ManagerId = NULL` o `WHERE ManagerId <> NULL`.

Sección 13.4: Igualdad

```
SELECT * FROM Employees
```

Esta sentencia 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

El uso de un `WHERE` al final de la sentencia `SELECT` permite limitar las filas devueltas a una condición. En este caso, donde hay una coincidencia exacta utilizando el signo `=`:

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

Sólo devolverá las filas en las que `DepartmentId` sea 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

Sección 13.5: La cláusula WHERE sólo devuelve las filas que coinciden con sus criterios

Steam tiene una sección de juegos por menos de 10 dólares en la página de su tienda. En algún lugar en el corazón de sus sistemas, probablemente hay una consulta que se parece a algo:

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

Sección 13.6: AND y OR

También puede combinar varios operadores para crear condiciones `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

AND

```
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

OR

```
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

Sección 13.7: Utilizar IN para devolver filas con un valor contenido en una lista

Este ejemplo utiliza la Tabla de `Car` de las Bases de Datos de `Example`.

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

Esta consulta devolverá el `Car` nº 2, que cuesta `200`, y el `Car` nº 3, que cuesta `100`. Tenga en cuenta que esto equivale a utilizar varias cláusulas con `OR`, por ejemplo

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

Sección 13.8: Utilizar LIKE para encontrar cadenas y subcadenas coincidentes

Consulte la documentación completa sobre el operador `LIKE`.

Este ejemplo utiliza la Tabla de `Employees` de las Bases de Datos de `Example`.

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

Esta consulta sólo devolverá el empleado nº 1 cuyo nombre coincida exactamente con "Juan".

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

Añadir `%` permite buscar una subcadena de caracteres:

- `John%` - devolverá cualquier `Employee` cuyo nombre empiece por 'John', seguido de cualquier cantidad de caracteres
- `%John` - devolverá cualquier `Employee` cuyo nombre termine en "John", precedido por cualquier cantidad de caracteres
- `%John%` - devolverá cualquier `Employee` cuyo nombre contenga "John" en cualquier parte del valor

En este caso, la consulta devolverá el `Employee` nº 2, cuyo nombre es "John", y el empleado nº 4, cuyo nombre es "Johnathon".

Sección 13.9: WHERE EXISTS

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

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

Sección 13.10: Utilizar HAVING para comprobar varias condiciones en un grupo

Tabla de `Orders`

CustomerId	ProductId	Quantity	Price
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

Para comprobar si hay clientes que hayan pedido tanto el `ProductId` 2 como el 3, se puede utilizar `HAVING`

```
SELECT customerId FROM orders WHERE productID IN (2,3) GROUP BY customerId
     HAVING COUNT(DISTINCT productID) = 2
```

Valor de retorno:

CustomerId
1

La consulta selecciona sólo los registros con los `productIds` en las preguntas y con la cláusula `HAVING` comprueba los grupos que tienen 2 `productIds` y no sólo uno.

Otra posibilidad sería

```
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 sólo los grupos que tienen al menos un registro con `productID` 2 y al menos uno con `productID` 3.

Capítulo 14: SKIP TAKE (Paginación)

Sección 14.1: Cantidad limitada de resultados

ISO/ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Oracle:

```
SELECT Id, Col1 FROM (SELECT Id, Col1, row_number() OVER (ORDER BY Id) RowNumber FROM TableName)
     WHERE RowNumber <= 20
```

SQL Server:

```
SELECT TOP 20 * FROM dbo.[Sale]
```

Sección 14.2: Saltar y luego tomar 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
```

Oracle; SQL Server:

```
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;
```

Sección 14.3: Saltar 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;
-- omite 20 para tomar uso número muy grande que es más que filas en la tabla
```

Oracle:

```
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;
```

Capítulo 15: EXCEPT

Sección 15.1: Seleccionar conjunto de datos excepto cuando los valores están en este otro conjunto de datos

```
-- los esquemas de los conjuntos de datos deben ser idénticos
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'
-- Devuelve Data1, Data2, Data4 y Data5
```

Capítulo 16: EXPLAIN y DESCRIBE

Sección 16.1: Consultar EXPLAIN SELECT

Una explicación delante de una consulta select muestra cómo se ejecutará la consulta. De esta forma puedes ver si la consulta utiliza un índice o si podrías optimizar tu consulta añadiendo un índice.

Ejemplo de consulta:

```
EXPLAIN SELECT * FROM user JOIN data ON user.test = data.fk_user;
```

Ejemplo de resultado:

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

Sección 16.2: DESCRIBE nombretabla;

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

```
DESCRIBE nombretabla;
```

Resultado ejemplar:

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 muestran los nombres de las columnas, seguidos del tipo de columna. Muestra si se permite `null` en la columna y si la columna utiliza un Índice. También se muestra el valor por defecto y si la tabla contiene algún comportamiento especial como un `auto_increment`.

Capítulo 17: Cláusula EXISTS

Sección 17.1: Cláusula EXISTS

Tabla `Customer`

Id	FirstName	LastName
1	Ozgur	Ozturk
2	Youssef	Medi
3	Henry	Tai

Tabla `Order`

Id	CustomerId	Amount
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

Id	FirstName	LastName
2	Youssef	Medi
3	Henry	Tai

Obtener todos los clientes sin pedido

```
SELECT * FROM Customer WHERE NOT EXISTS
     ( SELECT * FROM Order WHERE Order.CustomerId = Customer.Id )
```

Resultado

Id	FirstName	LastName
1	Ozgur	Ozturk

Propósito

`EXISTS`, `IN` y `JOIN` pueden utilizarse a veces para obtener el mismo resultado, pero no son iguales:

- `EXISTS` debe utilizarse para comprobar si un valor existe en otra tabla.
- `IN` debe utilizarse para listas estáticas.
- `JOIN` debe utilizarse para recuperar datos de otra tabla o tablas.

Capítulo 18: JOIN

JOIN es un método para combinar (unir) información de dos tablas. El resultado es un conjunto unido de columnas de ambas tablas, definido por el tipo de unión (INNER/OUTER/CROSS y LEFT/RIGHT/FULL, explicados más adelante) y los criterios de unión (cómo se relacionan las filas de ambas tablas).

Una tabla puede unirse a sí misma o a cualquier otra tabla. Si es necesario acceder a la información de más de dos tablas, se pueden especificar múltiples uniones en una cláusula FROM.

Sección 18.1: SELF JOIN

Una tabla puede estar unida a sí misma, con filas diferentes que coincidan entre sí por alguna condición. En este caso, los alias para distinguir las dos apariciones de la tabla.

En el siguiente ejemplo, para cada empleado de la tabla `Employees` de la base de datos de ejemplo, se devuelve un registro que contiene el nombre de pila del empleado junto con el correspondiente nombre de pila del responsable del empleado. Puesto que los jefes 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:

Employee	Manager
John	James
Michael	James
Johnathon	John

¿Cómo funciona?

La tabla original contiene estos registros:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
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

La primera acción consiste en crear un producto *cartesiano* de todos los registros de las tablas utilizadas en la cláusula `FROM`. En este caso es la tabla `Employees` dos veces, por lo que la tabla intermedia tendrá este aspecto (he eliminado todos los campos que no se utilizan en este ejemplo):

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
1	James	NULL	1	James	NULL
1	James	NULL	2	John	1
1	James	NULL	3	Michael	1
1	James	NULL	4	Johnathon	2
2	John	1	1	James	NULL
2	John	1	2	John	1
2	John	1	3	Michael	1
2	John	1	4	Johnathon	2
3	Michael	1	1	James	NULL
3	Michael	1	2	John	1
3	Michael	1	3	Michael	1
3	Michael	1	4	Johnathon	2
4	Johnathon	2	1	James	NULL
4	Johnathon	2	2	John	1
4	Johnathon	2	3	Michael	1
4	Johnathon	2	4	Johnathon	2

La siguiente acción consiste en conservar únicamente los registros que cumplan los criterios de `JOIN`, es decir, todos los registros en los que el `ManagerId` de tabla `e` alias sea igual al `Id` de tabla `m` alias:

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	John	1	1	James	NULL
3	Michael	1	1	James	NULL
4	Johnathon	2	2	John	1

A continuación, se evalúa cada expresión utilizada dentro de la cláusula **`SELECT`** para devolver esta tabla:

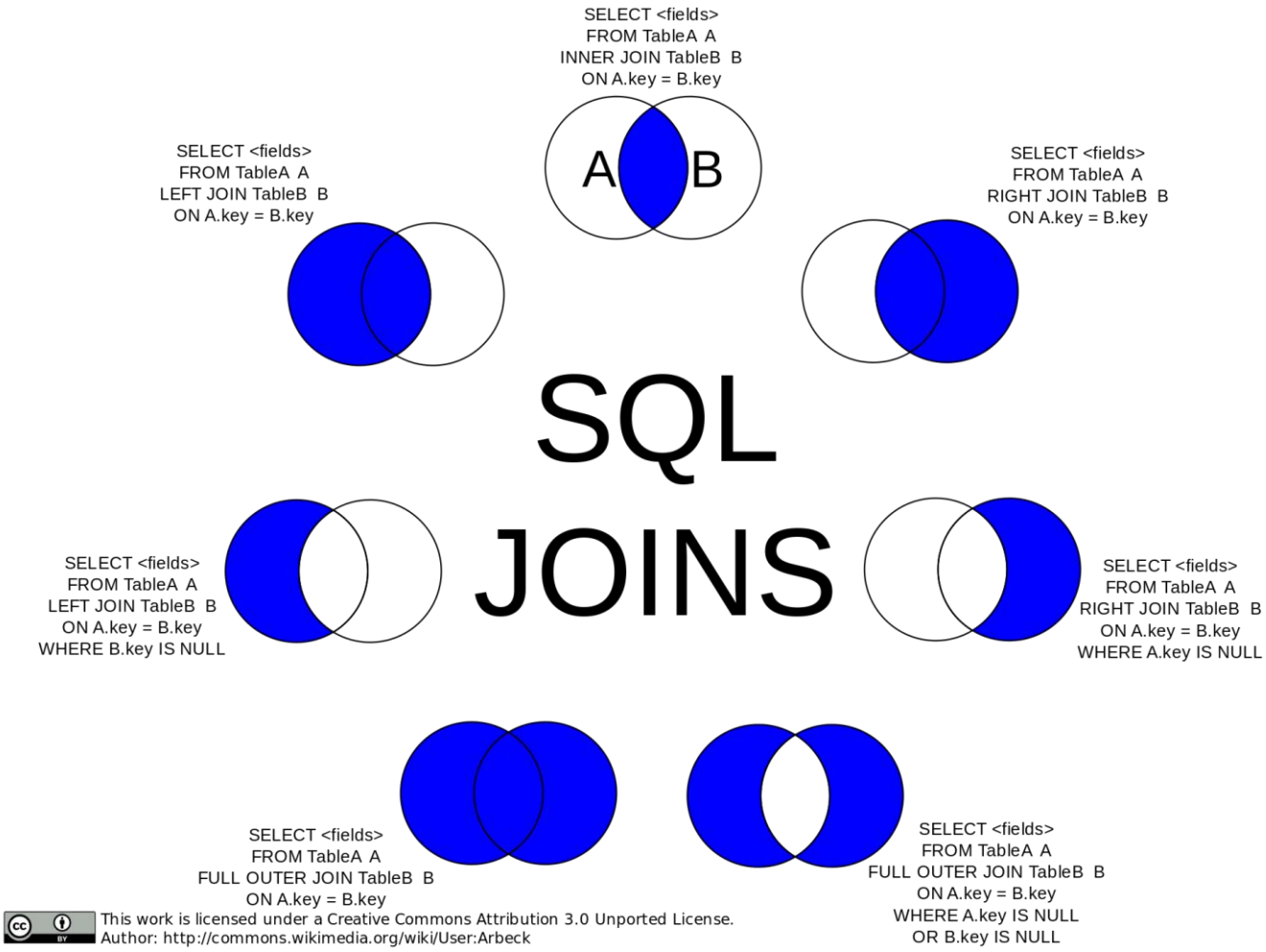
e.FName	m.FName
John	James
Michael	James
Johnathon	John

Por último, los nombres de columna `e.FName` y `m.FName` se sustituyen por sus nombres de columna alias, asignados con el operador `AS`:

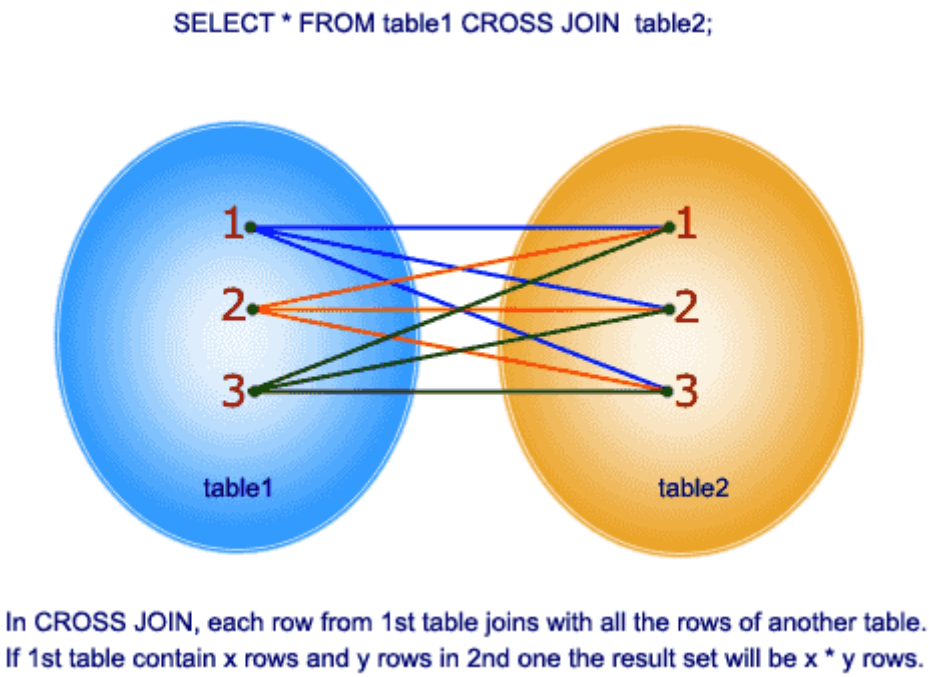
Employee	Manager
John	James
Michael	James
Johnathon	John

Sección 18.2: Diferencias entre INNER JOIN y OUTER JOINS

SQL dispone de varios tipos de unión para especificar si las filas (no) coincidentes se incluyen en el resultado: `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN` y `FULL OUTER JOIN` (las palabras clave `INNER` y `OUTER` son opcionales). La siguiente figura subraya las diferencias entre estos tipos de `JOINs`: el área azul representa los resultados devueltos por el `JOIN`, y el área blanca representa los resultados que el `JOIN` no devolverá.



Presentación gráfica de `CROSS JOIN` SQL (referencia):



A continuación, se ofrecen ejemplos de esta respuesta.

Por ejemplo, hay dos tablas:

A	B
1	3
2	4
3	5
4	6

Observe que (1,2) son exclusivas de A, (3,4) son comunes y (5,6) son exclusivas de B.

`INNER JOIN`

Un `INNER JOIN` utilizando cualquiera de las consultas equivalentes proporciona la intersección de las dos tablas, es decir, las dos filas que tienen en común:

```
SELECT * FROM a INNER JOIN b ON a.a = b.b;
SELECT a.*,b.* FROM a,b WHERE a.a = b.b;
```

A	B
3	3
4	4

`LEFT OUTER JOIN`

Un `LEFT OUTER JOIN` dará todas las filas de A, más las filas comunes de B:

```
SELECT * FROM a LEFT OUTER JOIN b ON a.a = b.b;
```

A	B
1	NULL
2	NULL
3	3
4	4

`RIGHT OUTER JOIN`

Del mismo modo, un `RIGHT OUTER JOIN` dará todas las filas de B, más las filas comunes de A:

```
SELECT * FROM a RIGHT OUTER JOIN b ON a.a = b.b;
```

A	B
3	3
4	4
NULL	5
NULL	6

`FULL OUTER JOIN`

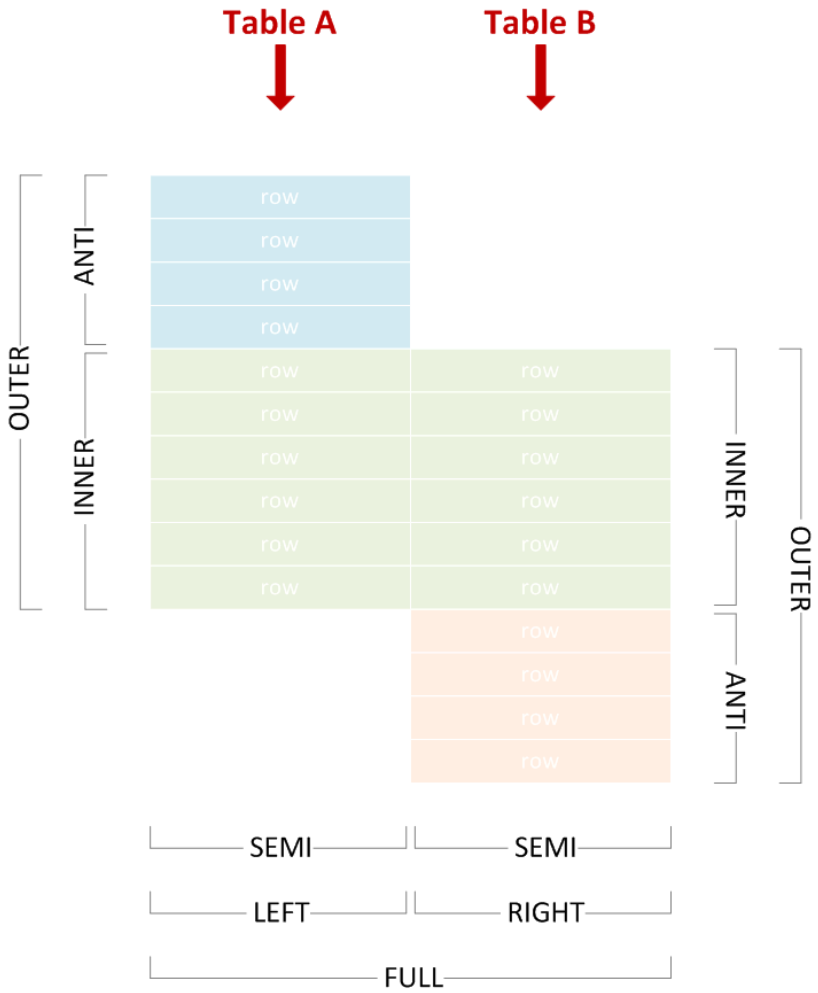
Un `FULL OUTER JOIN` le dará la unión de A y B, es decir, todas las filas de A y todas las filas de B. Si algo en A no tiene un dato correspondiente en B, entonces la parte de B es nula, y viceversa.

```
SELECT * FROM a FULL OUTER JOIN b ON a.a = b.b;
```

A	B
1	NULL
2	NULL
3	3
4	4
NULL	5
NULL	6

Sección 18.3: Terminología JOIN: INNER, OUTER, SEMI, ANTI...

Supongamos que tenemos dos tablas (A y B) y algunas de sus filas coinciden (en relación con la condición `JOIN` dada, sea cual sea en el caso concreto):



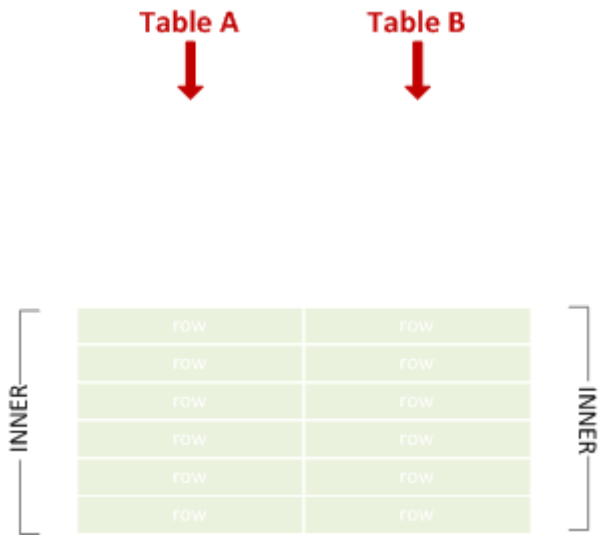
Podemos utilizar varios tipos de unión para incluir o excluir filas coincidentes o no coincidentes de cada lado, y nombrar correctamente la unión eligiendo los términos correspondientes del diagrama anterior.

Los ejemplos siguientes utilizan los siguientes datos de prueba:

```
CREATE TABLE A (X varchar(255) PRIMARY KEY);

CREATE TABLE B (Y varchar(255) PRIMARY KEY);

INSERT INTO A VALUES ('Amy'), ('John'), ('Lisa'), ('Marco'), ('Phil');

INSERT INTO B VALUES ('Lisa'), ('Marco'), ('Phil'), ('Tim'), ('Vincent');
```

INNER JOIN

Combina las filas izquierda y derecha que coinciden.

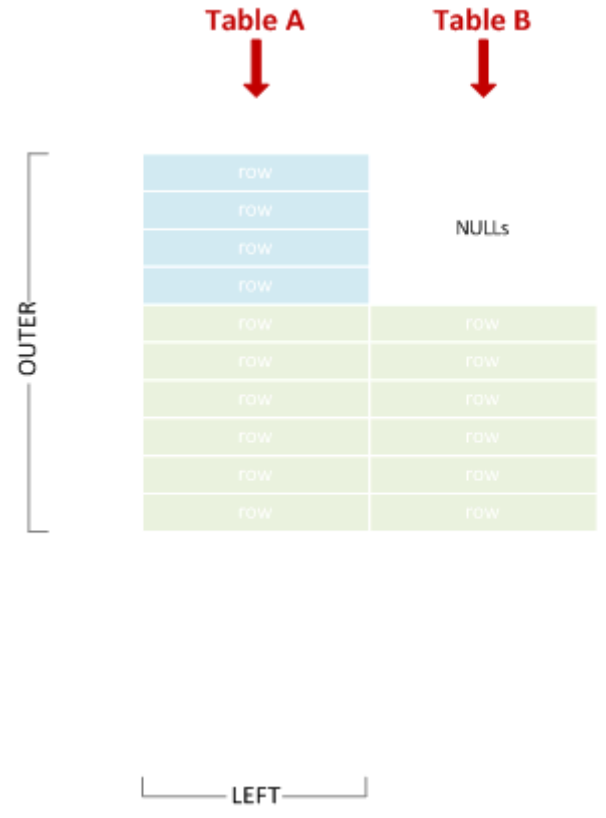



```
SELECT * FROM A JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil

`LEFT OUTER JOIN`

A veces se abrevia como "`LEFT JOIN`". Combina las filas de la izquierda y la derecha que coinciden, e incluye las filas de la izquierda que no coinciden.

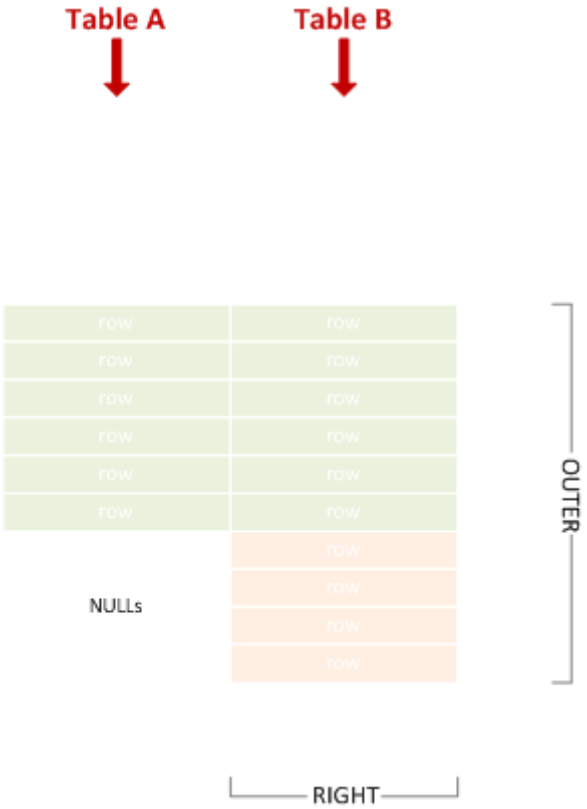



```
SELECT * FROM A LEFT JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil

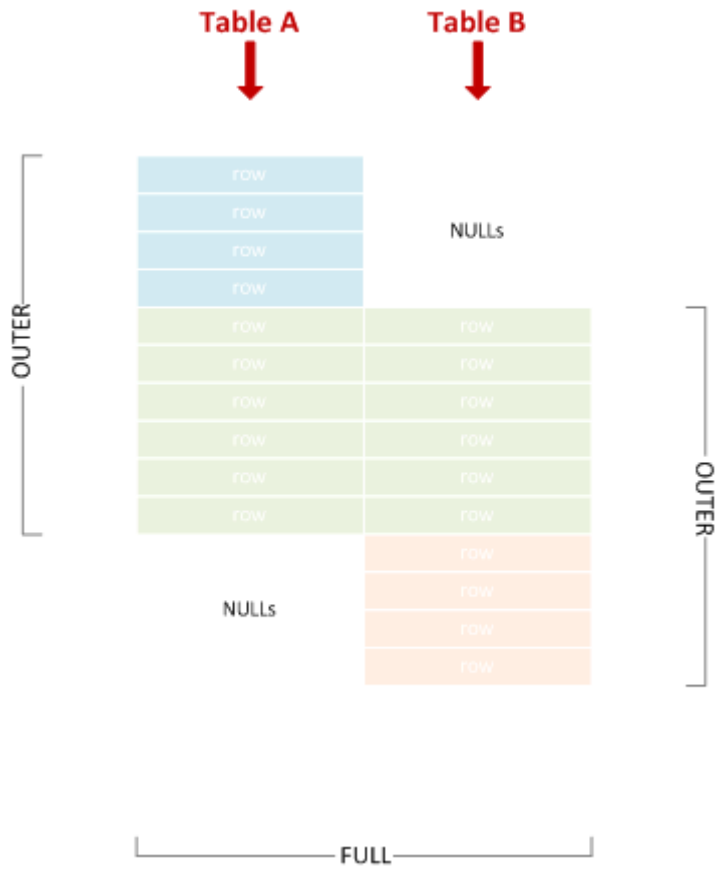
`RIGHT OUTER JOIN`

A veces se abrevia como "`RIGHT JOIN`". Combina las filas izquierdas y derechas que coinciden, e incluye las filas derechas que no coinciden.




```
SELECT * FROM A RIGHT JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim
NULL	Vincent



FULL OUTER JOIN

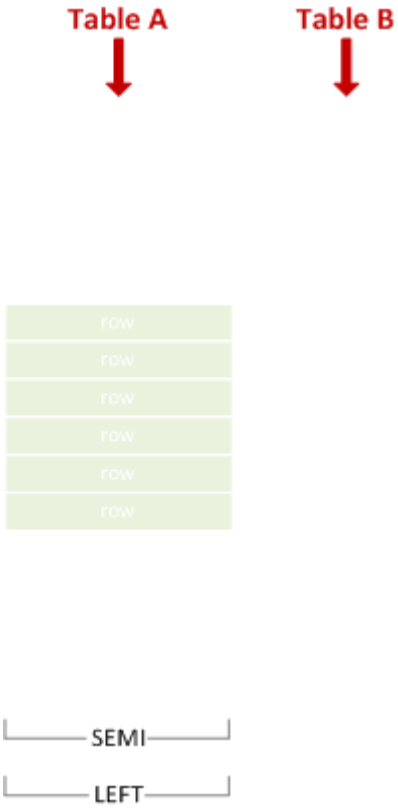
A veces se abrevia como "**FULL JOIN**". Unión de la unión exterior izquierda y derecha.

```
SELECT * FROM A FULL JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim
NULL	Vincent

LEFT SEMI JOIN

Incluye las filas de la izquierda que coinciden con las de la derecha.

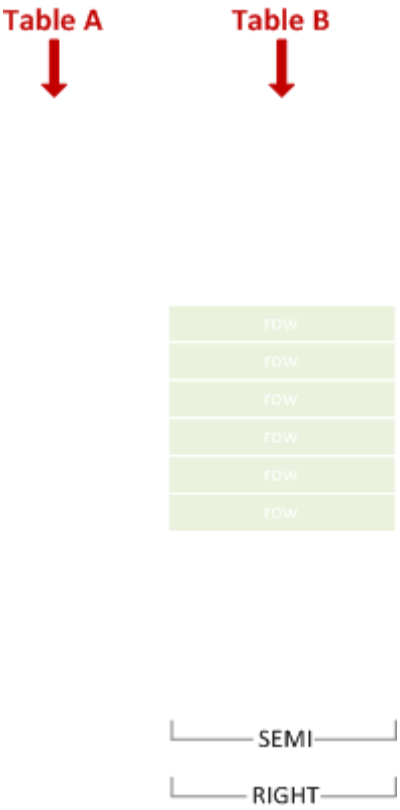



```
SELECT * FROM A WHERE X IN (SELECT Y FROM B);
```

X
Lisa
Marco
Phil

RIGHT SEMI JOIN

Incluye las filas de la derecha que coinciden con las de la izquierda.



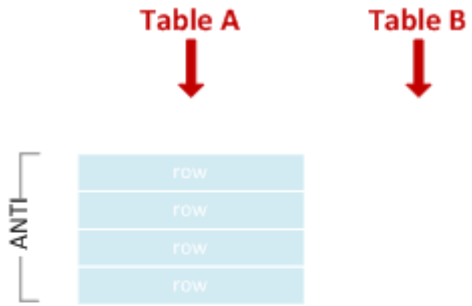

```
SELECT * FROM B WHERE Y IN (SELECT X FROM A);
```

Y
Lisa
Marco
Phil

Como puede ver, no existe una sintaxis `IN` específica para el `LEFT SEMI JOIN` frente al `RIGHT`: conseguimos el efecto simplemente cambiando las posiciones de las tablas dentro del texto SQL.

`LEFT ANTI SEMI JOIN`

Incluye las filas de la izquierda que no coinciden con las de la derecha.



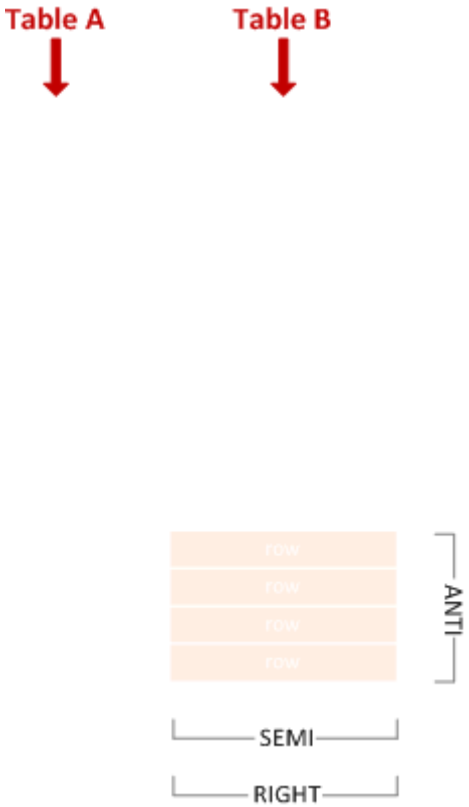

```
SELECT * FROM A WHERE X NOT IN (SELECT Y FROM B);
```

X
Amy
John

ADVERTENCIA: ¡Tenga cuidado si utiliza `NOT IN` en una columna NULL! Más detalles aquí.

`RIGHT ANTI SEMI JOIN`

Incluye las filas de la derecha que no coinciden con las de la izquierda.




```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);
```

Y
Tim
Vincent

Como puede ver, no existe una sintaxis `NOT IN` específica para el `LEFT ANTI SEMI JOIN` frente al `RIGHT`; el efecto se consigue simplemente cambiando las posiciones de las tablas en el texto SQL.

`CROSS JOIN`

Un producto cartesiano de todas las filas de la izquierda con todas las filas de la derecha.

```
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 `CROSS JOIN` es equivalente a una `INNER JOIN` con una condición de `JOIN` que siempre coincide, por lo que la siguiente consulta habría devuelto el mismo resultado:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

SELF-JOIN

Indica simplemente que una tabla se une consigo misma. Un `SELF-JOIN` puede ser cualquiera de los tipos de `JOINs` mencionados anteriormente. Por ejemplo, ésta es una `INNER SELF-JOIN`:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);
```

X	Y
Amy	John
Amy	Lisa
Amy	Marco
John	Marco
Lisa	Marco
Phil	Marco
Amy	Phil

Sección 18.4: LEFT OUTER JOIN

Un `LEFT OUTER JOIN` (también conocido como `LEFT JOIN` u `OUTER JOIN`) es un `JOIN` que asegura que todas las filas de la tabla izquierda están representadas; si no existe ninguna fila coincidente de la tabla derecha, sus campos correspondientes son `NULL`.

El siguiente ejemplo seleccionará todos los departamentos y el nombre de los empleados que trabajan en ese departamento. Los departamentos que no tengan empleados seguirán apareciendo 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:

Departments.Name	Employees.FName
HR	James
HR	John
HR	Johnathon
Sales	Michael
Tech	NULL

¿Cómo funciona?

Hay dos tablas en la cláusula `FROM`:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	Johnson	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

y

Id	Name
1	HR
2	Sales
3	Tech

En primer lugar, se crea un producto cartesiano a partir de las dos tablas y se obtiene una tabla intermedia. Los registros que cumplen los criterios de `JOIN` (*Departments.Id = Employees.IdDepartment*) se resaltan en negrita; éstos se pasan a la siguiente etapa de la consulta.

Como se trata de un `LEFT OUTER JOIN`, se devuelven todos los registros del lado `LEFT` del `JOIN` (Departments), mientras que los registros del lado `RIGHT` reciben un marcador `NULL` si no coinciden con los criterios del `JOIN`. En la tabla siguiente, **Tech** aparecerá con `NULL`

Id	Name	Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	**HR**	**1**	**James**	**Smith**	**1234567890**	**NULL**	**1**	**1000**	**01-01-2002**
1	**HR**	**2**	**John**	**Johnson**	**2468101214**	**1**	**1**	**400**	**23-03-2005**
1	HR	3	Michael	Williams	1357911131	1	2	600	12-05-2009
1	**HR**	**4**	**Johnathon**	**Smith**	**1212121212**	**2**	**1**	**500**	**24-07-2016**
2	Sales	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	Sales	2	John	Johnson	2468101214	1	1	400	23-03-2005
2	**Sales**	**3**	**Michael**	**Williams**	**1357911131**	**1**	**2**	**600**	**12-05-2009**
2	Sales	4	Johnathon	Smith	1212121212	2	1	500	24-07-2016
3	Tech	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
3	Tech	2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Tech	3	Michael	Williams	1357911131	1	2	600	12-05-2009
3	Tech	4	Johnathon	Smith	1212121212	2	1	500	24-07-2016

Por último, se evalúa cada expresión utilizada en la cláusula `SELECT` para obtener la tabla final:

Departments.Name	Employees.FName
HR	James
HR	John
Sales	Richard
Tech	NULL

Sección 18.5: JOIN implícito

Las uniones 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 unión implícita (ya que en realidad no contiene una cláusula de `JOIN`).

Todos los RDBMS la soportan, pero esta sintaxis suele desaconsejarse. Las razones por las que es una mala idea utilizar esta sintaxis son:

- Es posible que se produzcan uniones cruzadas accidentales que devuelvan resultados incorrectos, especialmente si hay muchas uniones en la consulta.
- Si pretendía una unión cruzada, entonces no está claro en la sintaxis (escriba `CROSS JOIN` en su lugar), y es probable que alguien lo cambie durante el mantenimiento.

El siguiente ejemplo seleccionará los nombres de pila 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.Name
James	HR
John	HR
Richard	Sales

Sección 18.6: CROSS JOIN

`CROSS JOIN` hace de 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 `TABLAA` tiene 20 filas y `TABLAB` tiene 20 filas, el resultado sería `20*20 = 400` filas de salida.

Utilización de una base de datos de ejemplo

```
SELECT d.Name, e.FName FROM Departments d CROSS JOIN Employees e;
```

Que vuelve:

d.Name	e.FName
HR	James
HR	John
HR	Michael
HR	Johnathon
Sales	James
Sales	John
Sales	Michael
Sales	Johnathon
Tech	James
Tech	John
Tech	Michael
Tech	Johnathon

Se recomienda escribir un `CROSS JOIN` explícito si desea realizar una unión cartesiana, para resaltar que esto es lo que desea.

Sección 18.7: CROSS APPLY y LATERAL JOIN

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 y Oracle.

La idea básica es que se aplica una función con valor de tabla (o subconsulta en línea) a cada fila que se une.

Esto permite, por ejemplo, unir sólo la primera entrada coincidente de otra tabla. La diferencia entre una unión normal y una lateral radica en el hecho de que puede utilizar una columna a la que se haya unido previamente en la subconsulta que "`CROSS APPLY`".

Sintaxis:

PostgreSQL 9.3+

```
LEFT | RIGHT | INNER JOIN LATERAL
```

SQL-Server:

```
CROSS | 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
```

Sección 18.8: FULL JOIN

Un tipo de JOIN menos conocido es el FULL JOIN. (Nota: FULL JOIN no está soportado por MySQL en 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 izquierda que no tienen coincidencias en la tabla derecha, o si hay filas en la tabla derecha que no tienen coincidencias en la tabla 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 utiliza eliminaciones suaves, tendrá que comprobar el estado de la eliminación suave de nuevo en la cláusula `WHERE` (porque `FULL JOIN` se comporta como `UNION`); es fácil pasar por alto este pequeño hecho, ya que pone `AP_SoftDeleteStatus = 1` en la cláusula `JOIN`.

Además, si está haciendo un `FULL JOIN`, normalmente tendrá que permitir `NULL` en la cláusula `WHERE`; olvidarse de permitir `NULL` en un valor tendrá los mismos efectos que un `INNER JOIN`, que es algo que no quiere si está haciendo un `FULL JOIN`.

Por 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)
```

Sección 18.9: JOINs recursivos

Las uniones recursivas se utilizan a menudo para obtener datos padre-hijo. En SQL, se implementan con expresiones recursivas de tabla común, 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;
```

Sección 18.10: INNER JOIN explícito básico

Un `JOIN` básico (también llamada "`INNER JON`") consulta datos de dos tablas, con su relación definida en una cláusula de `JOIN`.

El siguiente ejemplo seleccionará los nombres de pila de los empleados (`FName`) de la tabla `Employees` y el nombre del departamento para el que trabajan (`Name`) de la tabla `Departments`:

```
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:

Employees.FName	Departments.Name
James	HR
John	HR
Richard	Sales

Sección 18.11: Unir en una subconsulta

La unión de una subconsulta se utiliza a menudo cuando se desea obtener datos agregados de una tabla hija/detalles y mostrarlos junto con los registros de la tabla padre/cabecera. Por ejemplo, es posible que desee obtener un recuento de los registros secundarios, una media de alguna columna numérica de los registros secundarios o la fila superior o inferior en función de una fecha o un campo numérico. Este ejemplo utiliza alias, lo que facilita la lectura de las consultas cuando hay varias tablas implicadas. Este es el aspecto de una subconsulta `JOIN` bastante típica. En este caso estamos recuperando todas las filas de la tabla padre `Purchase Orders` y recuperando sólo la primera fila para cada registro padre de la tabla hija `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
```

Capítulo 19: UPDATE

Sección 19.1: UPDATE con datos de otra tabla

Los siguientes ejemplos rellenan un `PhoneNumber` para cualquier `Employee` que también sea `Customer` y que actualmente no tenga un número de teléfono establecido en la Tabla de `Employees`.

(Estos ejemplos utilizan las tablas `Employees` y `Customers` de las bases de datos de ejemplo).

Estándar SQL

Actualización mediante 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

Actualización mediante `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
```

SQL Server

Actualización mediante `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
```

Sección 19.2: Modificación de los valores existentes

Este ejemplo utiliza la Tabla de `Cars` 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 sencillo ejemplo, el `TotalCost` se incrementa en 100 para dos filas:

- El CosteTotal del Coche nº 3 pasa de 100 a 200
- El CosteTotal del Coche nº 4 pasa de 1254 a 1354

El nuevo valor de una columna puede derivarse de su valor anterior o del valor de cualquier otra columna de la misma tabla o de una tabla unida.

Sección 19.3: Actualización de filas especificadas

Este ejemplo utiliza la Tabla de `Cars` de las Bases de Datos de Ejemplo.

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

Esta sentencia establecerá el estado de la fila de '`Cars`' con `Id` 4 a `'READY'`.

La cláusula `WHERE` contiene una expresión lógica que se evalúa para cada fila. Si una fila cumple los criterios, se actualiza su valor. En caso contrario, la fila permanece sin cambios.

Sección 19.4: Actualizar todas las filas

Este ejemplo utiliza la Tabla de `Cars` de las Bases de Datos de Ejemplo.

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

Esta sentencia pondrá la columna "`Status`" de todas las filas de la tabla "`Cars`" a `'READY'` porque no tiene una cláusula `WHERE` para filtrar el conjunto de filas.

Sección 19.5: Captura de registros actualizados

A veces se desea capturar los registros que acaban de actualizarse.

```
CREATE TABLE #TempUpdated(ID INT)

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

Capítulo 20: CREATE DATABASE

Sección 20.1: CREATE DATABASE

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

```
CREATE DATABASE myDatabase;
```

Esto crearía una base de datos vacía llamada `myDatabase` donde se pueden crear tablas.

Capítulo 21: CREATE TABLE

Parametro	Detalles
nombreTabla	El nombre de la tabla
columnas	Contiene una "enumeración" de todas las columnas que tiene la tabla. Consulte **Crear una nueva tabla** para obtener más detalles.

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

Sección 21.1: Crear tabla a partir de SELECT

Es posible que desee crear un duplicado de una tabla:

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

Puede utilizar cualquiera de las otras funciones de una sentencia `SELECT` para modificar los datos antes de pasarlos a la nueva tabla. Las columnas de la nueva tabla se crean automáticamente en función de las filas seleccionadas.

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

Sección 21.2: Crear una nueva tabla

Se puede crear una tabla básica de `Employees`, que contenga un ID y el nombre y apellidos del empleado junto con su número de teléfono, utilizando

```
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`

A continuación, aparece la lista de nombres de columnas y sus propiedades, como el `Id`

```
Id INT IDENTITY(1,1) NOT NULL
```

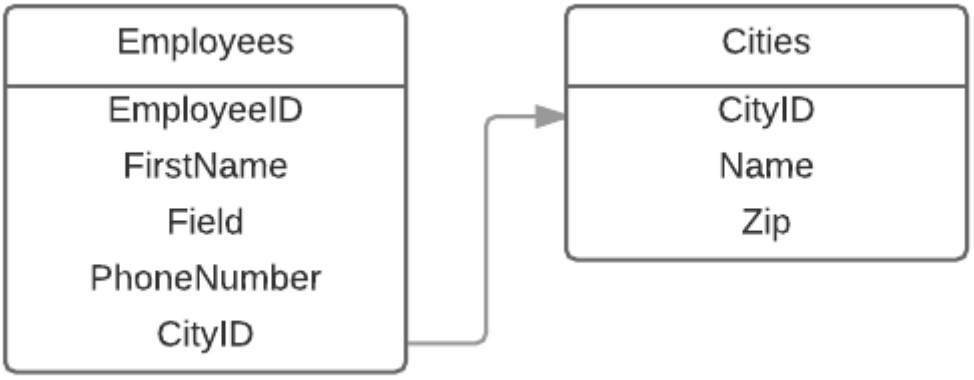
Valor	Significado
`Id`	el nombre de la columna.
`INT`	es el tipo de datos.
`IDENTITY(1,1)`	establece que la columna tendrá valores autogenerados comenzando en 1 e incrementándose en 1 por cada nueva fila.
`PRIMARY KEY`	establece que todos los valores de esta columna tendrán valores únicos
`NOT NULL`	establece que esta columna no puede tener valores nulos

Sección 21.3: CREATE TABLE con FOREIGN KEY

A continuación, encontrará la tabla Empleados con una referencia a la tabla Ciudades.

```
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 puede encontrar un diagrama de la 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	Significado
`CityID`	Nombre de la columna
`int`	tipo de columna
`FOREIGN KEY`	Hace la clave foránea *(opcional)*
`REFERENCES Cities(CityID)`	Hace referencia a la tabla `Cities` columna `CityID`

Importante: No se puede hacer referencia a una tabla que no existe en la base de datos. Sea fuente de hacer primero la tabla `Cities` y segundo la tabla `Employees`. Si lo haces al revés, dará error.

Sección 21.4: Duplicar una tabla

Para duplicar una tabla, basta con hacer lo siguiente:

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

Sección 21.5: Crear una tabla temporal o en memoria

PostgreSQL y SQLite

Para crear una tabla temporal local a la sesión:

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

SQL Server

Para crear una tabla temporal local a 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(...);
```

Capítulo 22: CREATE FUNCTION

Argumento	Descripcion
function_name	el nombre de la función
list_of_paramenters	parámetros que acepta la función
return_data_type	tipo que devuelve la función. Algunos tipos de datos SQL
function_body	el código de función
scalar_expression	valor escalar devuelto por la función

Sección 22.1: 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`.

Capítulo 23: TRY/CATCH

Sección 23.1: TRANSACTION en un TRY/CATCH

Esto revertirá ambas inserciones debido a una fecha inválida:

```
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 confirmará 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
```

Capítulo 24: UNION / UNION ALL

La palabra clave ***UNION*** en SQL se utiliza para combinar resultados de sentencias ***SELECT*** sin duplicados. Para utilizar `UNION` y combinar 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.

Sección 24.1: Consulta básica 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)
);
```

Supongamos que queremos extraer los nombres de todos los `managers` de nuestros departamentos.

Mediante `UNION` podemos obtener todos los empleados de los departamentos de RRHH y Finanzas que ocupan el `position` del `manager`

```
SELECT FirstName, LastName FROM HR_EMPLOYEES WHERE Position = 'manager' UNION ALL
     SELECT FirstName, LastName FROM FINANCE_EMPLOYEES WHERE Position = 'manager'
```

La sentencia `UNION` elimina las filas duplicadas de los resultados de la consulta. Dado que es posible que haya personas con el mismo nombre y cargo en ambos departamentos, utilizamos `UNION ALL` para no eliminar duplicados.

Si desea utilizar un alias para cada columna de salida, sólo tiene que ponerlos en la primera sentencia select, como se indica a continuación:

```
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'
```

Sección 24.2: Explicación sencilla y ejemplo

En términos sencillos:

- `UNION` une 2 conjuntos de resultados eliminando los duplicados del conjunto de resultados
- `UNION ALL` une 2 conjuntos de resultados sin intentar eliminar los duplicados

Un error que cometen muchas personas es utilizar `UNION` cuando no necesitan eliminar los duplicados. El coste de rendimiento adicional frente a grandes conjuntos de resultados puede ser muy significativo.

Cuando puede necesitar `UNION`

Supongamos que necesita filtrar una tabla en función de 2 atributos diferentes y ha creado índices no agrupados independientes para cada columna. Una `UNION` le permite aprovechar ambos índices sin que se produzcan duplicados.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

Esto simplifica el ajuste del rendimiento, ya que sólo se necesitan índices simples para realizar estas consultas de forma óptima. Incluso es posible que pueda arreglárselas con un número bastante menor de índices no agrupados, lo que también mejorará el rendimiento general de escritura contra la tabla de origen.

Cuando puede necesitar `UNION ALL`

Suponga 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 ser particionados físicamente a través de múltiples tablas (tal vez por razones de rendimiento, pero todavía quiere enrollar registros). Dado que los datos ya están divididos, hacer que el motor de base de datos elimine los duplicados no añade ningún valor y sólo añade tiempo de procesamiento adicional a las consultas.

Capítulo 25: ALTER TABLE

El comando `ALTER` en SQL se utiliza para modificar una columna/limitación en una tabla

Sección 25.1: Añadir columna(s)

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

La sentencia anterior añadiría columnas denominadas StartingDate que no puede ser NULL con valor por defecto como fecha actual y DateOfBirth que puede ser NULL en la tabla Employees.

Sección 25.2: Borrar columna

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

Esto no sólo borrará la información de esa columna, sino que eliminará la columna salario de la tabla empleados (la columna dejará de existir).

Sección 25.3: Añadir clave primaria

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

Esto añadirá una clave primaria a la tabla `Employees` en el campo `ID`. Si se incluye más de un nombre de columna entre paréntesis junto con `ID`, se creará una clave primaria compuesta. Al añadir más de una columna, los nombres de columna deben estar separados por comas.

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

Sección 25.4: Alterar columna

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

Esta consulta alterará el tipo de dato de la columna `StartingDate` y lo cambiará de un simple `DATE` a `DATETIME` y establecerá por defecto la fecha actual.

Sección 25.5: Borrar restricción

```
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.

Capítulo 26: INSERT

Sección 26.1: INSERT datos de otra tabla utilizando SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

Este ejemplo insertará todos los `Employees` en la tabla `Customers`. Como las dos tablas tienen campos diferentes y no quiere mover todos los campos, necesita establecer en qué campos insertar y qué campos seleccionar. No es necesario que los nombres de los campos correlativos se llamen igual, pero sí que tengan el mismo tipo de datos. En este ejemplo se asume que el campo Id tiene una especificación de identidad y se incrementará automáticamente.

Si tienes dos tablas que tienen exactamente los mismos nombres de campo y sólo quieres mover todos los registros puedes utilizar:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Sección 26.2: Insertar nueva fila

```
INSERT INTO Customers VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

Esta sentencia insertará una nueva fila en la tabla `Customers`. Observe que no se ha especificado un valor para la columna Id, ya que se añadirá automáticamente. Sin embargo, todos los demás valores de columna deben ser especificados.

Sección 26.3: Insertar sólo columnas especificadas

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
     VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

Esta sentencia insertará una nueva fila en la tabla `Customers`. Los datos sólo se insertarán en las columnas especificadas - observe que no se ha proporcionado ningún valor para la columna `PhoneNumber`. Tenga en cuenta, sin embargo, que todas las columnas marcadas como `NOT NULL` deben ser incluidas.

Sección 26.4: Insertar varias 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 (inserción masiva) al mismo tiempo, existen funciones y recomendaciones específicas del SGBD.

MySQL - LOAD DATA INFILE

MSSQL - BULK INSERT

Capítulo 27: MERGE

`MERGE` (a menudo también llamado `UPSERT` para "actualizar o insertar") permite insertar nuevas filas o, si una fila ya existe, actualizar la fila existente. El objetivo es realizar todo el conjunto de operaciones de forma atómica (para garantizar la coherencia de los datos) y evitar la sobrecarga de comunicación de varias sentencias SQL en un sistema cliente/servidor.

Sección 27.1: MERGE para que el destino coincida con el origen

```
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` impide actualizar registros que no han cambiado. El uso de la construcción `INTERSECT` permite comparar columnas anulables sin un tratamiento especial.

Sección 27.2: MySQL: recuento de usuarios por nombre

Supongamos que queremos saber cuántos usuarios tienen el mismo nombre. Creemos la tabla `users` como sigue:

```
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 ello, debemos determinar si existe una fila con su nombre y, en caso afirmativo, actualizarla para incrementar el recuento; por el contrario, si no existe ninguna fila, debemos crearla.

MySQL utiliza la siguiente sintaxis: insert ... on duplicate key update ... . En este caso

```
INSERT INTO users(name, COUNT) VALUES ('Joe', 1) ON DUPLICATE KEY UPDATE COUNT=COUNT+1;
```

Sección 27.3: PostgreSQL: recuento de usuarios por nombre

Supongamos que queremos saber cuántos usuarios tienen el mismo nombre. Creemos la tabla `users` como sigue:

```
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 ello, debemos determinar si existe una fila con su nombre y, en caso afirmativo, actualizarla para incrementar el recuento; por el contrario, si no existe ninguna fila, debemos crearla.

PostgreSQL utiliza la siguiente sintaxis: insert ... on conflict ... do update ...;. En este caso

```
INSERT INTO users(name, COUNT) VALUES('Joe', 1)
      ON CONFLICT (name) DO UPDATE SET COUNT = users.COUNT + 1;
```

Capítulo 28: CROSS APPLY, OUTER APPLY

Sección 28.1: Conceptos básicos de CROSS APPLY y OUTER APPLY

Aplicar se utilizará cuando la función valorada tabla en la expresión correcta.

Crear una tabla `Department` que contenga información sobre los departamentos. A continuación, cree una tabla `Employee` que contenga información sobre los empleados. Tenga en cuenta que cada empleado pertenece a un departamento, por lo que la tabla `Employee` tiene integridad referencial con la tabla `Department`.

La primera consulta selecciona los datos de la tabla `Department` y utiliza `CROSS APPLY` para evaluar la tabla `Employee` para cada registro de la tabla `Department`. La segunda consulta simplemente une la tabla `Department` con la tabla `Employee` 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 observa los resultados que han producido, se trata exactamente del mismo conjunto de resultados. ¿En qué se diferencia de un `JOIN` y cómo ayuda a escribir consultas más eficientes?

La primera consulta del Script #2 selecciona los datos de la tabla `Department` y utiliza `OUTER APPLY` para evaluar la tabla `Employee` para cada registro de la tabla `Department`. Para aquellas filas para las que no hay coincidencia en la tabla `Employee`, esas filas contienen valores `NULL` como puede ver en el caso de las filas 5 y 6. La segunda consulta utiliza simplemente un `LEFT OUTER JOIN` entre la tabla `Department` y la tabla `Employee`. Como era de esperar, la consulta devuelve todas las filas de la tabla `Department`, incluso aquellas filas que no coinciden con las de la tabla `Employee`.

```
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 términos de costes no habrá mucha diferencia.

Ahora llega el momento de ver dónde es realmente necesario el operador `APPLY`. En el Script #3, estoy creando una función con valores de tabla que acepta `DepartmentID` como parámetro y devuelve todos los empleados que pertenecen a este departamento. La siguiente consulta selecciona los datos de la tabla `Department` y utiliza `CROSS APPLY` para unirse a la función que hemos creado. Pasa el `DepartmentID` de cada fila de la expresión de la tabla externa (en nuestro caso la tabla `Department`) y evalúa la función para cada fila de forma similar a una subconsulta correlacionada. La siguiente consulta utiliza `OUTER APPLY` en lugar de `CROSS APPLY` y, por tanto, a diferencia de `CROSS APPLY`, que sólo devuelve datos correlacionados, `OUTER APPLY` devuelve también 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
```

Si ahora se pregunta, ¿podemos utilizar una simple unión en lugar de las consultas anteriores? La respuesta es NO, si sustituye `CROSS`/`OUTER APPLY` en las consultas anteriores por `INNER JOIN`/`LEFT OUTER JOIN`, especifique la cláusula `ON` (algo como `1=1`) y ejecute la consulta, obtendrá el error "The multi-part identifier "`D.DepartmentID`" could not be bound". Esto se debe a que con los `JOIN`s el contexto de ejecución de la consulta externa es diferente del contexto de ejecución de la función (o de una tabla derivada), y no se puede vincular un valor/variable de la consulta externa a la función como parámetro. Por lo tanto, el operador `APPLY` es necesario para este tipo de consultas.

Capítulo 29: DELETE

La sentencia `DELETE` se utiliza para eliminar registros de una tabla.

Sección 29.1: DELETE todas las filas

Si se omite una cláusula `WHERE`, se eliminarán todas las filas de una tabla.

```
DELETE FROM Employees
```

Consulte la documentación de `TRUNCATE` para obtener más detalles sobre cómo el rendimiento de `TRUNCATE` puede ser mejor porque ignora los desencadenadores y los índices y registros para simplemente eliminar los datos.

Sección 29.2: DELETE determinadas filas con WHERE

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

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

Sección 29.3: Cláusula TRUNCATE

Utilice esta opción para restablecer la tabla al estado en que se creó. Esto borra todas las filas y restablece valores como el autoincremento. Tampoco registra la eliminación de cada fila individual.

```
TRUNCATE TABLE Employees
```

Sección 29.4: DELETE determinadas filas basándose en comparaciones con otras tablas

Es posible que `DELETE` datos de una tabla sí coinciden (o no) con determinados datos de otras tablas.

Supongamos que queremos ELIMINAR los datos de la fuente una vez cargados en el destino.

```
DELETE FROM Source WHERE EXISTS
     (SELECT 1 -- el valor específico en SELECT no importa FROM Target
          WHERE Source.ID = Target.ID)
```

La mayoría de las implementaciones RDBMS comunes (por ejemplo, MySQL, Oracle, PostgresSQL, Teradata) permiten unir tablas durante `DELETE` permitiendo una comparación más compleja en una sintaxis compacta.

Añadiendo complejidad al escenario original, supongamos que el Agregado se construye desde el `Target` una vez al día y no contiene el mismo `ID`, pero sí la misma fecha. Supongamos también que deseamos eliminar los datos de `Source` sólo después de que el agregado se haya completado para el día.

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

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

En PostgreSQL utilizar:

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

Esto resulta esencialmente en `INNER JOIN`s entre `Source`, `Target` y `Aggregate`. La eliminación se realiza en `Source` cuando existen los mismos `ID` en `Target` `AND` la fecha presente en `Target` para esos `ID` también existe en `Aggregate`.

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 uniones explícitas pueden mencionarse en las sentencias `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 comprobar las situaciones de no coincidencia en lugar de las de coincidencia con todos los estilos sintácticos (observe `NOT EXISTS` a continuación)

```
DELETE FROM Source WHERE NOT EXISTS
     (SELECT 1 -- el valor específico en SELECT no importa FROM Target
          WHERE Source.ID = Target.ID)
```

Capítulo 30: TRUNCATE

La sentencia `TRUNCATE` borra todos los datos de una tabla. Es similar a `DELETE` sin filtro, pero, dependiendo del software de base de datos, tiene ciertas restricciones y optimizaciones.

Sección 30.1: Eliminar todas las filas de la tabla Empleados

```
TRUNCATE TABLE Employee;
```

Usar `TRUNCATE TABLE` es a menudo mejor que usar `DELETE TABLE` ya que ignora todos los índices y triggers y simplemente elimina todo.

`DELETE TABLE` es una operación basada en filas, lo que significa que se borra cada fila. `TRUNCATE TABLE` es una operación de página de datos en la que 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 sentencia `DELETE TABLE`.

Aunque podemos eliminar filas específicas con `DELETE`, no podemos `TRUNCATE` filas específicas, sólo podemos `TRUNCATE` todos los registros a la vez. Al eliminar todas las filas y luego insertar un nuevo registro, se seguirá añadiendo el valor del clave principal autoincrementado del valor insertado anteriormente, mientras que, al truncar, el valor del clave principal autoincrementado también se restablecerá y comenzará a partir de `1`.

Tenga en cuenta que, al truncar la tabla, **no debe haber claves externas**, de lo contrario obtendrá un error.

Capítulo 31: DROP TABLE

Sección 31.1: Comprobar la existencia antes de soltar

MySQL Version ≥ 3.19

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL Version ≥ 8.x

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server Version ≥ 2005

```
IF EXISTS (SELECT * FROM Information_Schema.Tables WHERE Table_Schema = 'dbo'
     AND Table_Name = 'MyTable')
DROP TABLE dbo.MyTable
```

SQLite Version ≥ 3.0

```
DROP TABLE IF EXISTS MyTable;
```

Sección 31.2: Borrado simple

```
DROP TABLE MyTable;
```

Capítulo 32: DROP o DELETE DATABASE

Sección 32.1: DROP DATABASE

Soltar la base de datos es una simple sentencia de una línea. `DROP DATABASE` borrará la base de datos, por lo tanto, asegúrese siempre 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 `Employees`

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

Capítulo 33: Eliminar en cascada

Sección 33.1: ON DELETE CASCADE

Suponga que tiene una aplicación que administra habitaciones.
Suponga además que su aplicación funciona por cliente (inquilino).
Tiene varios clientes.
Así que su base de datos contendrá una tabla para clientes y otra para habitaciones.

Ahora, cada cliente tiene N habitaciones.

Esto debería significar que tiene una clave foránea en su tabla de habitaciones, que hace referencia a la tabla de clientes.

```
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
```

Si un cliente se pasa a otro programa, tendrás que borrar sus datos en tu programa. Pero si lo hace

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

Entonces tendrás una violación de clave foránea, porque no puedes borrar al cliente cuando todavía tiene habitaciones.

Ahora tendrías que escribir código en tu aplicación que borre las habitaciones del cliente antes de borrar el cliente. Asume además que, en el futuro, muchas más dependencias de claves foráneas serán añadidas en tu base de datos, porque la funcionalidad de tu aplicación se expande. Horrible. Por cada modificación en su base de datos, tendrá que adaptar el código de su aplicación en N lugares. Posiblemente también tendrás que adaptar el código de otras aplicaciones (por ejemplo, las interfaces con otros sistemas).

Hay una solución mejor que hacerlo en tu código.

Puede añadir `ON DELETE CASCADE` a su clave externa.

```
ALTER TABLE dbo.T_Room -- WITH CHECK -- SQL-Server puede especificar 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 salas se borran automáticamente cuando se borra el cliente.

Problema resuelto - sin cambios en el código de la aplicación.

Una advertencia: En Microsoft SQL-Server, esto no funcionará si tiene una tabla que hace referencia a sí misma. Por lo tanto, si intenta definir una cascada de eliminación en una estructura de árbol recursiva, como esta:

```
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 Microsoft-SQL-server no permite establecer una clave ajena con `ON DELETE CASCADE` en una estructura de árbol recursiva. Una razón para esto es, que el árbol es posiblemente cíclico, y que posiblemente conduciría 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 en tiempo de ejecución.
En ese caso, tendrás que implementar la función de borrado tú mismo.

Una palabra de precaución:

Esto significa que ya no puedes simplemente borrar y volver a insertar la tabla de clientes, porque si haces esto, se borrarán todas las entradas en "`T_Room`"... (ya no hay actualizaciones no delta).

Capítulo 34: GRANT y REVOKE

Sección 34.1: Conceder/revocar privilegios

```
GRANT SELECT, UPDATE ON Employees TO User1, User2;
```

Conceda permiso a `User1` y `User2` para realizar operaciones `SELECT` y `UPDATE` en la tabla `Employees`.

```
REVOKE SELECT, UPDATE ON Employees FROM User1, User2;
```

Revocar a `User1` y `User2` el permiso para realizar operaciones `SELECT` y `UPDATE` en la tabla `Employees`.

Capítulo 35: XML

Sección 35.1: Consulta a partir de un 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
```

Capítulo 36: Claves primarias

Sección 36.1: Creación de una clave primaria

```
CREATE TABLE Employees (
     Id int NOT NULL,
     PRIMARY KEY (Id),
     ...
);
```

Esto creará la tabla `Employees` con "`Id`" como clave primaria. La clave primaria se puede utilizar para identificar de forma única las filas de una tabla. Sólo se permite una clave primaria por tabla.

Una clave también puede estar compuesta por uno o más campos, lo que se denomina clave compuesta, con la siguiente sintaxis:

```
CREATE TABLE EMPLOYEE (
     e1_id INT,
     e2_id INT,
     PRIMARY KEY (e1_id, e2_id)
)
```

Sección 36.2: Uso del incremento automático

Muchas bases de datos permiten hacer que el valor de la clave primaria se incremente automáticamente cuando se añade una nueva clave. Esto garantiza 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
);
```

SQL Server

```
CREATE TABLE Employees (
Id int NOT NULL IDENTITY,
PRIMARY KEY (Id)
);
```

SQLite

```
CREATE TABLE Employees (
Id INTEGER PRIMARY KEY
);
```

Capítulo 37: Índices

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

Existen varios tipos de índices que pueden crearse en una tabla. Cuando existe un índice en las columnas utilizadas en la cláusula `WHERE`, `JOIN` u `ORDER BY` de una consulta, puede mejorar sustancialmente el rendimiento de la consulta.

Sección 37.1: Índice ordenado

Si utiliza un índice que está ordenado de la forma en que lo recuperaría, la sentencia `SELECT` no realizaría una ordenación adicional al recuperarlo.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

Al ejecutar la consulta

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

El sistema de base de datos no realizaría una ordenación adicional, ya que puede realizar una búsqueda de índices en ese orden.

Sección 37.2: Índice parcial o filtrado

SQL Server y SQLite permiten crear índices que contengan no sólo un subconjunto de columnas, sino también un subconjunto de filas.

Considere una cantidad creciente constante de pedidos con `order_state_id` igual a `finished(2)`, y una cantidad estable de pedidos con `order_state_id` igual a `started(1)`.

Si su empresa hace uso de consultas de este tipo:

```
SELECT id, comment FROM orders WHERE order_state_id = 1 AND product_id = @some_value;
```

La indexación parcial permite limitar el índice, incluyendo sólo las órdenes no finalizadas:

```
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 coste de actualización del índice.

Sección 37.3: Creación de 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 piden al servidor que ordene o seleccione por valores en *`EmployeeId`*, como la siguiente:

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

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

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

En este caso, el índice sería útil para las consultas que piden ordenar o seleccionar por todas las columnas incluidas, si el conjunto de condiciones está ordenado de la misma manera. Esto significa que, al recuperar los datos, puede encontrar las filas que desea recuperar utilizando el índice, en lugar de buscar en toda la tabla.

Por ejemplo, en el siguiente caso se 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 caso 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, a través de 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 descubra que recuperar el índice y filtrar por el `OwnerId`, y luego recuperar sólo las filas necesarias es más rápido que recuperar la tabla completa, especialmente si la tabla es grande).

Sección 37.4: Dar de baja un índice o desactivarlo y reconstruirlo

```
DROP INDEX ix_cars_employee_id ON Cars;
```

Podemos utilizar el comando `DROP` para eliminar nuestro índice. En este ejemplo vamos a eliminar el índice llamado *`ix_cars_employee_id`* en la tabla `Cars`.

Esto elimina el índice por completo, y si el índice está agrupado, eliminará cualquier agrupación. No se puede reconstruir sin volver a crear el índice, lo que puede ser lento y costoso desde el punto de vista computacional. Como alternativa, se puede desactivar el índice:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

Esto permite que la tabla conserve la estructura, junto con los metadatos sobre el índice.

Además, se conservan las estadísticas del índice, lo que permite evaluar fácilmente el cambio. Si se justifica, el índice puede reconstruirse posteriormente, en lugar de volver a crearse por completo;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Sección 37.5: Índices agrupados, únicos y ordenados

Los índices pueden tener varias características que pueden establecerse en el momento de su creación o modificando los índices existentes.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

La sentencia SQL anterior crea un nuevo índice agrupado en *`Employees`*. Los índices agrupados son índices que dictan la estructura real de la tabla; la propia tabla se ordena para que coincida con la estructura del índice. Esto significa que sólo puede haber un índice agrupado en una tabla. Si ya existe un índice agrupado en la tabla, la sentencia anterior fallará. (Las tablas sin índices agrupados también se llaman heaps).

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

Esto creará un índice único para la columna *`Email`* en la tabla *`Customers`*. Este índice, además de acelerar las consultas como un índice normal, también obligará a que cada dirección de correo electrónico de esa columna sea única. Si se inserta o actualiza una fila con un valor de *`Email`* no único, la inserción o actualización fallará por defecto.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

Esto crea un índice en *`Customers`* que también crea una restricción en la tabla de que el *`EmployeeID`* debe ser único. (Esto fallará si la columna no es actualmente única - en este caso, si hay empleados que comparten un ID).

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

Esto crea un índice que se ordena en orden descendente. Por defecto, los índices (en MSSQL server, al menos) son ascendentes, pero eso se puede cambiar.

Sección 37.6: Índice de reconstrucción

Con el paso 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 las páginas no se corresponde con el orden físico). Reconstruir un índice es muy similar a eliminarlo y volver a crearlo.

Podemos reconstruir un índice con

```
ALTER INDEX index_name REBUILD;
```

Por defecto, la reconstrucción del índice es una operación offline que bloquea la tabla e impide realizar DML sobre ella, pero muchos RDBMS permiten la reconstrucción online. Además, algunos proveedores de bases de datos ofrecen alternativas a la reconstrucción de índices como `REORGANIZE` (SQLServer) o `COALESCE`/`SHRINK SPACE` (Oracle).

Sección 37.7: 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 *`Email`* de *`Customers`*. Sin embargo, se puede definir un comportamiento alternativo para este caso:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

Capítulo 38: Número de fila

Sección 38.1: Borrar todos los registros excepto el último (tabla de 1 a muchos)

```
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;
```

Sección 38.2: 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
```

Sección 38.3: Números de fila con particiones

Utiliza un criterio de partición para agrupar la numeración de las filas en función del mismo.

```
SELECT ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC)
     AS RowNumber, DepartmentId, Fname, LName FROM Employees
```

Capítulo 39: SQL GROUP BY vs DISTINCT

Sección 39.1: Diferencia entre GROUP BY y DISTINCT

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

orderId	userId	storeName	orderValue	orderDate
1	43	Store A	25	20-03-2016
2	57	Store B	50	22-03-2016
3	43	Store A	30	25-03-2016
4	82	Store C	10	26-03-2016
5	21	Store 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

storeName	total_nr_orders	nr_unique_customers	average_order_value	first_order	lastOrder
Store A	3	2	33.3	20-03-2016	29-03-2016
Store B	1	1	50	22-03-2016	22-03-2016
Store C	1	1	10	26-03-2016	26-03-2016

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

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

storeName	userId
Store A	43
Store B	57
Store C	82
Store A	21

Capítulo 40: Búsqueda de duplicados en un subconjunto de columnas con detalle

Sección 40.1: Estudiantes 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) una al lado de la otra.

Capítulo 41: Funciones de cadena de caracteres

Las funciones de cadena de caracteres (strings) realizan operaciones con valores de cadena de caracteres y devuelven valores numéricos o de cadena de caracteres.

Las funciones de cadena de caracteres permiten, por ejemplo, combinar datos, extraer una subcadena de caracteres, comparar cadenas de caracteres o convertir una cadena de caracteres a mayúsculas o minúsculas.

Sección 41.1: Concatenar

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

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

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

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

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

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

En algunas bases de datos, los tipos que no son cadenas de caracteres deben ser fundidos o convertidos:

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

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

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

Algunas bases de datos pueden utilizar el operador no estándar `+` (pero en la mayoría, `+` sólo funciona para números):

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

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

Sección 41.2: Longitud

SQL Server

El `LEN` no cuenta el espacio final.

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

`DATALENGTH` cuenta el espacio final.

```
SELECT DATALENGTH('Hello') -- devuelve 5
SELECT DATALENGTH('Hello '); -- devuelve 6
```

No obstante, debe tenerse en cuenta que `DATALENGTH` devuelve la longitud de la representación en bytes subyacente de la cadena de caracteres, que depende, entre otras cosas, del conjunto de caracteres utilizado para almacenar la cadena de caracteres.

```
DECLARE @str varchar(100) = 'Hello '
-- varchar suele ser una cadena de caractere ASCII, que ocupa 1 byte por char
SELECT DATALENGTH(@str) -- devuelve 6
DECLARE @nstr nvarchar(100) = 'Hello '
-- nvarchar es una cadena de caracteres unicode que ocupa 2 bytes por carácter
SELECT DATALENGTH(@nstr) -- deveulve 12
```

Oracle

Sintaxis: `Length ( char )`

Ejemplos:

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

Véase también: `LengthB`, `LengthC`, `Length2`, `Length4`

Sección 41.3: Recortar espacios vacíos

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

En MSSQL no existe un único `TRIM()`

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

MySql y Oracle

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

Sección 41.4: Mayúsculas y minúsculas

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

Sección 41.5: Dividir

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

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

Resultado:

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

Sección 41.6: Sustituir

Sintaxis:

`REPLACE(` Cadena de caracteres a buscar , Cadena de caracteres a buscar y reemplazar , Cadena de caracteres a colocar en la cadena de caracteres original `)`

Ejemplo:

```
SELECT REPLACE('Peter Steve Tom', 'Steve', 'Billy') -- Valores de retorno: Peter Billy Tom
```

Sección 41.7: REGEXP

MySQL `Version ≥ 3.19`

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

```
SELECT 'bedded' REGEXP '[a-f]' -- devuelve True
SELECT 'beam' REGEXP '[a-f]' -- devuelve False
```

Sección 41.8: SUBSTRING

La sintaxis es: `SUBSTRING ( string_expression, start, length )`. Tenga en cuenta que las cadenas de caracteres SQL tienen un índice 1.

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

Suele utilizarse junto con la función `LEN()` para obtener los últimos n caracteres de una cadena de caracteres de longitud desconocida.

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

Sección 41.9: STUFF

Introduce una cadena de caracteres en otra, sustituyendo 0 o más caracteres en una posición determinada.

Nota: la posición de `start` está indexada en 1 (se empieza a indexar en 1, no en 0).

Sintaxis:

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

Por ejemplo:

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

Sección 41.10: LEFT - RIGHT

La sintaxis es:

```
LEFT( string-expression , integer )
RIGHT( string-expression , integer )
```

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

Oracle SQL no tiene funciones `LEFT` y `RIGHT`. Pueden emularse con `SUBSTR` y `LENGTH`.

```
SUBSTR( string-expression, 1, integer )
SUBSTR( string-expression, length(string-expression)-integer+1, integer)
```

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

Sección 41.11: REVERSE

La sintaxis es: `REVERSE( string-expression )`

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

Sección 41.12: REPLICATE

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

La sintaxis es: `REPLICATE ( string-expression , integer )`

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

Sección 41.13: Función REPLACE en consulta SQL SELECT y UPDATE

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

La sintaxis de la función `REPLACE` es:

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

El siguiente ejemplo sustituye las apariciones de `South` por `Southern` en la tabla `Employees`:

FirstName	Address
James	South New York
John	South Boston
Michael	South San Diego

Declaración de `SELECT`:

Si aplicamos la siguiente función `REPLACE`:

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

Resultado:

FirstName	Address
James	Southern New York
John	Southern Boston
Michael	Southern San Diego

Declaración de `UPDATE`:

Podemos utilizar una función de `REPLACE` para hacer cambios permanentes en nuestra tabla a través del siguiente enfoque.

```
UPDATE Employees SET city = (Address, 'South', 'Southern');
```

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

```
UPDATE Employees SET Address = (Address, 'South', 'Southern') WHERE Address LIKE 'South%';
```

Sección 41.14: INSTR

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

Sintaxis: `INSTR ( string, substring )`

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

Sección 41.15: PARSENAME

BASE DE DATOS: SQL Server

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

Más detalles MSDN:PARSENAME

Sintaxis

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

Ejemplo

Para obtener el nombre del objeto, utilice el índice de pieza `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, utilice el índice parcial `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 `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, utilice el índice `4`

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

`PARSENAME` devolverá null si la parte especificada no está presente en la cadena de caracteres de nombre de objeto dada

Capítulo 42: Funciones (Agregar)

Sección 42.1: Agregación condicional

Tabla de `Payments`

Customer	Payment_type	Amount
Peter	Credit	100
Peter	Credit	300
John	Credit	1000
John	Devit	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:

Customer	Credit	Debit
Peter	400	0
John	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:

Customer	credit_transaction_count	debit_transaction_count
Peter	2	0
John	1	1

Sección 42.2: Concatenación de listas

Crédito parcial para esta respuesta SO.

La concatenación de listas agrega una columna o expresión combinando los valores en una sola cadena para cada grupo. Se puede especificar una cadena para delimitar cada valor (en blanco o una coma cuando se omite) y el orden de los valores en el resultado. Aunque no forma parte del estándar SQL, cada uno de los principales proveedores de bases de datos relacionales lo admite 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;
```

SQL Server

SQL Server 2016 y anteriores

(CTE incluido para fomentar 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 hacer el pedido:

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

ordenar 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;
```

Sección 42.3: SUM

La función `SUM` suma el valor de todas las filas del grupo. Si se omite la cláusula agrupar por, suma todas las filas.

```
SELECT SUM(Salary) TotalSalary FROM Employees;
```

TotalSalary
2500

```
SELECT DepartmentId, SUM(Salary) TotalSalary FROM Employees GROUP BY DepartmentId;
```

DepartmentId	TotalSalary
1	2000
2	500

Sección 42.4: AVG()

La función de agregado `AVG()` devuelve la media de una expresión dada, normalmente valores numéricos de 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 que se muestran a continuación:

TABLA DE EJEMPLOS

city_name	population	year
New York City	8,550,405	2015
New York City	...	...
New York City	8,000,906	2005

Seleccionar la población media de la ciudad de Nueva York, EE.UU. a partir de una tabla que contiene los nombres de las ciudades, las medidas de población y los 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';
```

Obsérvese que el año de medición no aparece en la consulta, ya que la población se promedia a lo largo del tiempo.

RESULTADOS

city_name	avg_population
New York City	8,250,754

Nota: La función `AVG()` convertirá los valores a tipos numéricos. Esto es especialmente importante cuando se trabaja con fechas.

Sección 42.5: COUNT

Puede contar el número de filas:

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

TotalRows
4

O contar 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 de que no contará los valores `NULL`:

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

mgr
3

(Hay una columna `ManagerID` con valor nulo)

También puede utilizar **`DISTINCT`** dentro de otra función como **`COUNT`** para encontrar sólo los miembros **`DISTINCT`** del conjunto sobre los que realizar la operación.

Por ejemplo:

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

Devolverán valores diferentes. *`SingleCount`* sólo contará los continentes una vez, mientras que *`AllCount`* incluirá los duplicados.

ContinetCode
OC
EU
AS
NA
NA
AF
AF

`AllCount`: 7 `SingleCount`: 5

Sección 42.6: MIN

Encuentra el valor más pequeño de la columna:

```
SELECT MIN(age) FROM Employee;
```

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

Sintaxis:

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

Sección 42.7: MAX

Encuentra el valor máximo de la columna:

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

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

Sintaxis:

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

Capítulo 43: Funciones (escalar/una fila)

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

Las funciones escalares se utilizan siempre que se permita una expresión dentro de una sentencia T-SQL.

Sección 43.1: DATE y TIME

En SQL, se utilizan tipos de datos de fecha y hora para almacenar información de calendario. Estos tipos de datos incluyen `TIME`, `DATE`, `SMALLDATETIME`, `DATETIME`, `DATETIME2` y `DATETIMEOFFSET`. Cada tipo de dato tiene un formato específico.

Tipo de datos	Formato
TIME	hh:mm:ss[.nnnnnnn]
DATE	YYYY-MM-DD
SMALLDATETIME	YYYY-MM-DD hh:mm:ss
DATETIME	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
```

DATENAME
Saturday

Se utiliza la función `GETDATE` para determinar la fecha y hora actuales del ordenador que ejecuta la instancia SQL actual. Esta función no incluye la diferencia de zona horaria.

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

SYSTEMDATE
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 parte de la fecha puede ser año, mes, semana, día, hora, minuto, segundo o milisegundo. A continuación, especifique la fecha inicial en el parámetro startdate y la fecha final en el parámetro enddate para las que desea encontrar la diferencia.

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

SalesOrderID	Processing time
43659	7
43660	7
43661	7
43662	7

La función `DATEADD` permite añadir un intervalo a parte de una fecha concreta.

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

Added20MoreDays
2017-02-03 00:00:00.000

Sección 43.2: Modificaciones de carácter

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

La función `LOWER(char)` convierte el parámetro de caracteres dado en caracteres minúsculos.

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

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

Sección 43.3: Función de configuración y conversión

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

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

Server
SQL064

En SQL, la mayoría de las conversiones de datos se producen de forma implícita, sin intervención del usuario.

Para realizar cualquier conversión que no pueda completarse implícitamente, puede utilizar las funciones `CAST` o `CONVERT`.

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

En este caso, utilizamos las funciones `CAST` y `CONVERT` para convertir el tipo de datos `DATETIME` en el tipo de datos `VARCHAR`.

La función `CAST` utiliza siempre el estilo por defecto. Por ejemplo, representará fechas y horas utilizando el formato YYYY-MM-DD.

La función `CONVERT` utiliza el estilo de fecha y hora que usted 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
```

Cast	Convert
David Hamiltion was hired on 2003-02-04	David Hamiltion was hired on 04/02/03

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

En la sintaxis de la función, se especifica la cadena que debe convertirse, la palabra clave AS y, a continuación, el tipo de datos requerido. Opcionalmente, también puede especificar la cultura en la que debe formatearse el valor de la cadena. Si no se especifica, se utiliza el idioma de la sesión.

Si el valor de la cadena no puede convertirse a un formato numérico, de fecha o de hora, se producirá un error. En ese caso, deberá utilizar `CAST` o `CONVERT` para la conversión.

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

Date in English
2012-08-13 00:00:00.0000000

Sección 43.4: Función lógica y matemática

SQL dispone de dos funciones lógicas: `CHOOSE` e `IIF`.

La función `CHOOSE` devuelve un elemento de una lista de valores, en función de su posición en la lista. Esta posición se especifica mediante el índice.

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

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

Result
Sales

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

La función `IIF` devuelve uno de dos valores, basándose en una condición determinada. Si la condición es verdadera, devolverá un valor verdadero. En caso 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 expresión booleana se evalúa como `true` y el parámetro `false_value` especifica el valor que debe devolverse si la expresión booleana se evalúa como `false`.

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

BusinessEntityID	SalesYTD	Bonus?
274	559697.5639	Bonus
275	3763178.1787	Bonus
285	172524.4512	No Bonus

En este ejemplo, se utiliza la función `IIF` para devolver uno de dos valores. Si las ventas de un vendedor en lo que va de año son superiores a 200.000, esta persona podrá optar a una bonificación. Los valores inferiores a 200.000 significan que los empleados no tienen derecho a bonificaciones.

SQL incluye varias funciones matemáticas que puede utilizar para realizar cálculos sobre 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 `-1` indica una expresión negativa, el valor `+1` indica una expresión positiva y `0` indica cero.

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

Sign
-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 `POWER`. Esta función proporciona el valor de una expresión elevado a una potencia especificada.

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

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

Result
125000

Capítulo 44: Funciones (analíticas)

Las funciones analíticas se utilizan para determinar valores basados en grupos de valores. Por ejemplo, puede utilizar este tipo de función para determinar totales, porcentajes o el resultado principal dentro de un grupo.

Sección 44.1: LAG y LEAD

La función `LAG` proporciona datos sobre las filas anteriores a la fila actual en el mismo conjunto de resultados. Por ejemplo, en una sentencia `SELECT`, puede comparar valores de la fila actual con valores de una fila anterior.

Se utiliza una expresión escalar para especificar los valores que deben compararse. El parámetro offset es el número de filas anteriores a la fila actual que se utilizarán en la comparación. Si no se especifica el número de filas, se utiliza el valor por defecto de una fila.

El parámetro por defecto especifica el valor que debe devolverse cuando la expresión en offset tiene un valor NULL. Si no se especifica un valor, se devuelve un valor `NULL`.

La función `LEAD` proporciona datos sobre las filas posteriores a la fila actual en el conjunto de filas. Por ejemplo, en una sentencia `SELECT`, puede comparar valores de la fila actual con valores de la fila siguiente.

Los valores que deben compararse se especifican mediante una expresión escalar. El parámetro offset es el número de filas después de la fila actual que se utilizarán en la comparación.

Se especifica el valor que debe devolverse cuando la expresión en offset tiene un valor `NULL` utilizando el parámetro por defecto. Si no especifica estos parámetros, se utiliza el valor predeterminado de una fila y se devuelve un valor `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 listados arriba y abajo, con los registros ordenados en base a la columna `BusinessEntityID`.

BusinessEntityID	SalesYTD	Lead value	Lag value
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

Sección 44.2: PERCENTILE_DISC y PERCENTILE_CONT

La función `PERCENTILE_DISC` lista el valor de la primera entrada en la que la distribución acumulativa es mayor que el percentil que usted proporciona utilizando el parámetro `numeric_literal`.

Los valores se agrupan por conjuntos de filas o particiones, según se especifique en la cláusula `WITHIN GROUP`.

La función `PERCENTILE_CONT` es similar a la función `PERCENTILE_DISC`, pero devuelve la media de la suma de la primera entrada coincidente y la 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 literal numérico en la función `PERCENTILE_DISC`. La columna Percentil Discreto de un conjunto de resultados enumera el valor de la fila en el que la distribución acumulativa es superior al percentil especificado.

BusinessEntityID	JobTitle	SickLeaveHours	Cumulative Distribution	Percentile Discreet
272	Application Specialist	55	0.25	**56**
268	Application Specialist	56	0.75	**56**
269	Application Specialist	56	0.75	**56**
267	Application Specialist	57	1	**56**

Para basar el cálculo en un conjunto de valores, se utiliza la función `PERCENTILE_CONT`. La columna «Percentil continuo» de los resultados enumera el valor medio de la suma del valor resultante 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	JobTitle	SickLeaveHours	Cumulative Distribution	Percentile Discreet	Percentile Continuous
272	Application Specialist	55	0.25	**56**	**56**
268	Application Specialist	56	0.75	**56**	**56**
269	Application Specialist	56	0.75	**56**	**56**
267	Application Specialist	57	1	**56**	**56**

Sección 44.3: FIRST_VALUE

La función `FIRST_VALUE` se utiliza para determinar el primer valor de un conjunto de resultados ordenado, que se identifica mediante 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 utiliza para devolver el `ID` del estado o provincia con el tipo impositivo más bajo. La cláusula `OVER` se utiliza para ordenar los tipos impositivos y obtener el tipo más bajo.

StateProvinceID	Name	TaxRate	FirstValue
74	Utah State Sales Tax	5.00	74
36	Minnesota State Sales Tax	6.75	74
30	Massachusetts State Sales Tax	7.00	74
1	Canadian GST	7.00	74
57	Canadian GST	7.00	74
63	Canadian GST	7.00	74

Sección 44.4: LAST_VALUE

La función `LAST_VALUE` proporciona el último valor de un conjunto de resultados ordenado, que se especifica mediante 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 de cada conjunto de filas en los valores ordenados.

TerritoryID	StartDate	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

Sección 44.5: 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 del conjunto de resultados siempre tiene un rango porcentual de cero. El valor del valor más alto -o último- del 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 denomina 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, se utiliza una cláusula `ORDER` para dividir - o agrupar - las filas recuperadas por la sentencia `SELECT` en función de los cargos de los empleados, con los resultados de cada grupo ordenados en función del número de horas de baja por enfermedad que han utilizado los empleados.

BusinessEntityID	JobTitle	SickLeaveHours	Percent Rank	Cumulative Distribution
267	Application Specialist	57	0	0.25
268	Application Specialist	56	0.333333333333333	0.75
269	Application Specialist	56	0.333333333333333	0.75
272	Application Specialist	55	1	1
262	Assitant to the Cheif Financial Officer	48	0	1
239	Benefits Specialist	45	0	1
252	Buyer	50	0	0.111111111111111
251	Buyer	49	0.125	0.333333333333333
256	Buyer	49	0.125	0.333333333333333
253	Buyer	48	0.375	0.555555555555555
254	Buyer	48	0.375	0.555555555555555

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

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

Capítulo 45: Funciones de ventana

Sección 45.1: Establecer un indicador si otras filas tienen una propiedad común

Digamos que tengo estos datos:

Artículos de la tabla

id	name	tag
1	example	unique_tag
2	foo	simple
42	bar	simple
3	baz	hello
51	quux	world

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 ítems
```

El resultado será:

id	name	tag	flag
1	example	unique_tag	false
2	foo	simple	true
42	bar	simple	true
3	baz	hello	false
51	quux	world	false

En caso de que su base de datos no tenga `OVER` y `PARTITION` puede utilizar 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
```

Sección 45.2: Búsqueda de registros “fuera de secuencia” mediante la función LAG()

Dados estos datos de muestra:

ID	STATUS	STATUS_TIME	STATUS_BY
1	ONE	2016-09-28-19.47.52.501398	USER_1
2	ONE	2016-09-28-19.47.52.501511	USER_2
1	THREE	2016-09-28-19.47.52.501517	USER_3
3	TWO	2016-09-28-19.47.52.501521	USER_2
3	THREE	2016-09-28-19.47.52.501524	USER_4

Los elementos identificados por valores `ID` deben pasar de `STATUS` “ONE” a “TWO” a “THREE” en secuencia, sin saltarse estados. El problema es encontrar usuarios (valores `STATUS_BY`) que violen la regla y pasen de “ONE” inmediatamente a “THREE”.

La función analítica `LAG()` ayuda a resolver el problema devolviendo para cada fila el valor de 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 disponga de `LAG()`, puede utilizarla para obtener 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'
```

Sección 45.3: Obtener un total actualizado

Teniendo en cuenta estos datos:

date	amount
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
```

le dará

date	amount	running
2016-03-10	-250	-250
2016-03-11	-50	-300
2016-03-12	200	-100
2016-03-14	100	0
2016-03-15	100	-100

Sección 45.4: Suma del total de filas seleccionadas a cada fila

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

id	name	Ttl_Rows
1	example	5
2	foo	5
3	bar	5
4	baz	5
5	quux	5

En lugar de utilizar dos consultas para obtener un recuento y luego la línea, puede utilizar un agregado como función de ventana y utilizar el conjunto de resultados completo como ventana.

Esto puede utilizarse como base para cálculos posteriores sin la complejidad de autouniones adicionales.

Sección 45.5: Obtención de las N filas más recientes en agrupaciones múltiples

Teniendo en cuenta estos datos

User_ID	Completion_Date
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`:

User_ID	Completion_Date	Row_Num
1	2016-07-21	1
2	2016-07-22	1

Capítulo 46: Expresiones comunes de tabla

Sección 46.1: Generar 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 utilizarse con recursividad 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 1-5:

```
-- Dé un nombre de tabla `Numbers" y una columna `i` para contener los números
WITH Numbers(i) AS (
     -- Número inicial/índice
     SELECT 1
     -- Operador UNION ALL de nivel superior necesario para la recursividad
     UNION ALL
     -- Expresión de iteración:
     SELECT i + 1
     -- Expresión de tabla que declaramos en primer lugar utilizada como fuente para la recursión
     FROM Numbers
     -- Cláusula para definir el final de la recursion
     WHERE i < 5
)
-- Utilice la expresión de tabla generada como una tabla regular
SELECT i FROM Numbers;
```

i
1
2
3
4
5

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

Sección 46.2: Enumerar recursivamente un subárbol

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
     -- empezar con esta fila
     SELECT 1, ID, FName, LName FROM Employees WHERE ID = 1

     UNION ALL

     -- obtener los empleados que tienen alguna de las filas previamente seleccionadas como
     manager
     SELECT ManagedByJames.Level + 1, Employees.ID, Employees.FName, Employees.LName
          FROM Employees JOIN ManagedByJames ON Employees.ManagerID = ManagedByJames.ID

     ORDER BY 1 DESC -- búsqueda en profundidad
)
SELECT * FROM ManagedByJames;
```

Level	ID	FName	LName
1	1	James	Smith
2	2	John	Johnson
3	4	Johnathon	Smith
2	3	Michael	Williams

Sección 46.3: 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;
```

ID	Model	TotalCost
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
```

Sección 46.4: Subir recursivamente en un árbol

```
WITH RECURSIVE ManagersOfJonathon AS (
     -- empezar con esta fila
     SELECT * FROM Employees WHERE ID = 4

     UNION ALL

     -- obtener los gestores de todas las filas seleccionadas anteriormente
     SELECT Employees.* FROM Employees JOIN ManagersOfJonathon
          ON Employees.ID = ManagersOfJonathon.ManagerID
)
SELECT * FROM ManagersOfJonathon;
```

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId
4	Johnathon	Smith	1212121212	2	1
2	John	Johnson	2468101214	1	1
1	James	Smith	1234567890	NULL	1

Sección 46.5: Generar fechas de forma recursiva, ampliado para incluir la elaboración de listas de equipo como ejemplo

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

-- Secuencia de transición = De descanso y relajación a turno diurno y a turno nocturno
-- RR (Descanso y relajación) = 1
-- DS (Turno de día) = 2
-- NS (Turno de noche) = 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 A 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

Sección 46.6: Funcionalidad Oracle CONNECT BY con CTEs recursivos

La funcionalidad `CONNECT BY` de Oracle proporciona muchas características útiles y no triviales que no están incorporadas cuando se utilizan CTEs recursivas estándar de SQL. Este ejemplo replica estas características (con algunas adiciones en aras de la exhaustividad), utilizando la sintaxis de SQL Server. Es muy útil para los desarrolladores de Oracle que encuentren que faltan muchas características 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 (
     /* Ancla */
     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
     /* Recursivo */
     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
```

`CONNECT BY` características demostradas anteriormente, con explicaciones:

- Cláusulas
  - `CONNECT BY`: Especifica la relación que define la jerarquía.
  - `START WITH`: Especifica los nodos raíz.
  - `ORDER SIBLINGS BY`: Ordena los resultados adecuadamente.
- Parámetros
  - `NOCYCLE`: Detiene el procesamiento de una rama cuando se detecta un bucle. Las jerarquías válidas son grafos acíclicos dirigidos, y las referencias circulares violan esta construcción.
- Operadores
  - `PRIOR`: Obtiene datos del padre del nodo.
  - `CONNECT_BY_ROOT`: Obtiene datos de la raíz del nodo.
- Pseudocolumnas
  - `LEVEL`: Indica la distancia del nodo a su raíz.
  - `CONNECT_BY_ISLEAF`: Indica un nodo sin hijos.
  - `CONNECT_BY_ISCYCLE`: Indica un nodo con referencia circular.
- Funciones
  - `SYS_CONNECT_BY_PATH`: Devuelve una representación aplanada/concatenada de la ruta al nodo desde su raíz.

Capítulo 47: Vistas

Sección 47.1: Vistas sencillas

Una vista puede filtrar algunas filas de la tabla base o proyectar sólo algunas columnas de la misma:

```
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 formar la vista:

```
SELECT * FROM new_employees_details
```

Id	FName	Salary	Hire_date
4	Johnathon	500	24-07-2016

Sección 47.2: Vistas complejas

Una vista puede ser una consulta realmente compleja (agregaciones, uniones, subconsultas, etc.). Sólo asegúrate de añadir nombres de columna para todo lo que selecciones:

```
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 puede seleccionar en ella como en cualquier tabla:

```
SELECT * FROM dept_income;
```

DepartmentName	TotalSalary
HR	1900
Sales	600

Capítulo 48: Vistas materializadas

Una vista materializada es una vista cuyos resultados se almacenan físicamente y deben actualizarse periódicamente para mantenerse al día. Por lo tanto, son útiles para almacenar los resultados de consultas complejas y de larga duración cuando no se requieren resultados en tiempo real. Las vistas materializadas pueden crearse en Oracle y PostgreSQL. Otros sistemas de bases de datos ofrecen funciones similares, como las vistas indexadas de SQL Server o las tablas de consultas materializadas de DB2.

Sección 48.1: 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)
```

Capítulo 49: Comentarios

Sección 49.1: Comentarios de una línea

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

```
SELECT * FROM Employees -- este es un comentario WHERE FName = 'John'
```

Sección 49.2: Comentarios multilínea

Los comentarios de código de varias líneas se envuelven en `/* ... */`:

```
/* Esta consulta
devuelve todos los empleados */
SELECT * FROM Employees
```

También es posible insertar un comentario de este tipo en medio de una línea:

```
SELECT /* todas las columnas: */ * FROM Employees
```

Capítulo 50: Claves externas

Sección 50.1: Explicación de las claves externas

Las restricciones de clave foránea garantizan la integridad de los datos, ya que obligan a que los valores de una tabla coincidan con los de otra.

Un ejemplo de dónde se requiere una clave foránea 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
);
```

Inserte los valores con la siguiente sentencia:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

La siguiente tabla contiene la información de las asignaturas ofrecidas 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 ajena debe coincidir con el tipo de datos de la clave referenciada).

La restricción de clave externa de la columna `Dept_Code` sólo permite valores si ya existen en la tabla de 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 emitirá un error de violación de clave foránea, porque `CS300` no existe en la tabla Departamento. Pero cuando intente 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 utilizar claves foráneas

- Una clave externa debe hacer referencia a una clave `UNIQUE` (o `PRIMARY`) en la tabla principal.
- La introducción de un valor `NULL` en una columna de clave externa no genera ningún error.
- Las restricciones de clave externa pueden hacer referencia a tablas de la misma base de datos.
- Las restricciones de clave externa pueden hacer referencia a otra columna de la misma tabla (autorreferencia).

Sección 50.2: Creación de una tabla con una clave externa

En este ejemplo tenemos una tabla existente, `SuperHeros`.

Esta tabla contiene una clave primaria `ID`.

Añadiremos 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`.

Capítulo 51: SEQUENCE

Sección 51.1: CREATE SEQUENCE

```
CREATE SEQUENCE orders_seq START WITH 1000 INCREMENT BY 1;
```

Crea una secuencia con un valor inicial de `1000` que se incrementa en `1`.

Sección 51.2: Utilizar secuencias

Se utiliza una referencia a *`seq_name`*`.NEXTVAL` para obtener el siguiente valor de una secuencia. Una sentencia sólo puede generar un único valor de secuencia. Si hay varias referencias a `NEXTVAL` en una sentencia, utilizarán el mismo número generado.

`NEXTVAL` puede utilizarse para `INSERTS`

```
INSERT INTO Orders (Order_UID, Customer) VALUES (orders_seq.NEXTVAL, 1032);
```

Puede utilizarse para `UPDATES`

```
UPDATE Orders SET Order_UID = orders_seq.NEXTVAL WHERE Customer = 581;
```

También puede utilizarse para `SELECTS`

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Capítulo 52: Subconsultas

Sección 52.1: Subconsulta en la cláusula FROM

Una subconsulta en una cláusula `FROM` actúa de forma 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
```

Sección 52.2: Subconsulta en la cláusula SELECT

```
SELECT Id, FName, LName, (SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id)
     AS NumberOfCars FROM Customers
```

Sección 52.3: Subconsulta en la cláusula WHERE

Utilice una subconsulta para filtrar el conjunto de resultados. Por ejemplo, esto devolverá todos los empleados con un salario igual al empleado mejor pagado.

```
SELECT * FROM Employees WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Sección 52.4: Subconsultas correlacionadas

Las subconsultas correlacionadas (también conocidas como sincronizadas o coordinadas) son consultas anidadas que hacen referencia 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 hace referencia a la fila `Employee eOuter` de su consulta externa.

Sección 52.5: Filtrar los resultados de la consulta utilizando la consulta en una tabla diferente

Esta consulta selecciona todos los empleados que no están en la tabla `Supervisors`.

```
SELECT * FROM Employees WHERE EmployeeID not in (SELECT EmployeeID FROM Supervisors)
```

Se pueden obtener los mismos resultados utilizando una `LEFT JOIN`.

```
SELECT * FROM Employees AS e LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
     WHERE s.EmployeeID is NULL
```

Sección 52.6: Subconsultas en la cláusula FROM

Puede utilizar subconsultas para definir una tabla temporal y utilizarla 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 las ciudades de la tabla meteorológica cuya variación diaria de temperatura es superior a 20. El resultado es:

city	temp_var
ST LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

Sección 52.7: Subconsultas en la cláusula WHERE

El siguiente ejemplo encuentra las ciudades (del ejemplo de ciudades) cuya población está por debajo de la temperatura media (obtenida mediante una subconsulta):

```
SELECT name, pop2000 FROM cities WHERE pop2000 < (SELECT avg(pop2000) FROM cities);
```

Aquí: la subconsulta `(SELECT avg(pop2000) FROM cities)` se utiliza para especificar condiciones en la cláusula `WHERE`. El resultado es:

name	pop2000
San Francisco	776733
ST LOUIS	348189
Kansas City	146866

Capítulo 53: Bloques de ejecución

Sección 53.1: Usando BEGIN ... FIN

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

Capítulo 54: Procedimientos almacenados

Sección 54.1: Crear y llamar a un procedimiento almacenado

Los procedimientos almacenados pueden crearse a través de una GUI de gestión de bases de datos (ejemplo SQL Server), o a través de una sentencia SQL, como se indica a continuación:

```
-- Definir un nombre y parámetros
CREATE PROCEDURE Northwind.getEmployee
     @LastName nvarchar(50),
     @FirstName nvarchar(50)
AS

-- Definir la consulta a ejecutar
SELECT FirstName, LastName, Department FROM Northwind.vEmployeeDepartment
     WHERE FirstName = @FirstName AND LastName = @LastName AND EndDate IS NULL;
```

Llamada 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
```

Capítulo 55: Disparadores

Sección 55.1: CREATE TRIGGER

Este ejemplo crea un disparador que inserta un registro en una segunda tabla (`MyAudit`) después de que se inserta un registro en la tabla en la que está definido el disparador (`MyTable`). Aquí la tabla "insertada" es una tabla especial utilizada por Microsoft SQL Server para almacenar las filas afectadas durante las sentencias `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
     -- insertar registro de auditoría en la tabla MiAuditoría
     INSERT INTO MyAudit(MyTableId, User)
     (SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Sección 55.2: Utilice Trigger para gestionar una "Papelera de reciclaje" para los elementos eliminados

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

Capítulo 56: Transacciones

Sección 56.1: Transacción simple

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

Sección 56.2: Transacción de reversión

Cuando algo falla en el código de tu transacción y quieres deshacerlo, puedes revertir tu 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
```

Capítulo 57: Diseño de la tabla

Sección 57.1: Propiedades de una tabla bien diseñada

Una verdadera base de datos relacional debe ir más allá de introducir datos en unas cuantas tablas y escribir algunas sentencias SQL para extraer esos datos.

En el mejor de los casos, una estructura de tablas mal diseñada ralentizará la ejecución de las consultas y podría imposibilitar que la base de datos funcione según lo previsto.

Una tabla de una base de datos no debe considerarse una tabla más; tiene que seguir una serie de reglas para ser considerada verdaderamente relacional. Académicamente se denomina «relación» para hacer la distinción.

Las cinco reglas de una tabla relacional son:

1. Cada valor es *atómico*; el valor de cada campo de cada fila debe ser un único 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 demás registros de la tabla.
5. El orden de las filas y columnas no tiene importancia.

Una tabla conforme a las cinco reglas:

Id	Name	DOB	Manager
1	Fred	11/02/1971	3
2	Fred	11/02/1971	3
3	Sue	08/07/1975	2

Regla 1: Cada valor es atómico. `Id`, `Name`, `DOB` y `Manager` sólo contienen un único valor.

Regla 2: Id contiene sólo enteros, `Name` contiene texto (podríamos añadir que es texto de cuatro caracteres o menos), `DOB` contiene fechas de un tipo válido y `Manager` contiene enteros (podríamos añadir que corresponde a un campo `Primary Key` en una tabla de managers).

Regla 3: `Id`, `Name`, `DOB` y `Manager` son nombres de encabezado únicos dentro de la tabla.

Regla 4: La inclusión del campo `Id` garantiza que cada registro es distinto de cualquier otro registro de la tabla.

Una mesa mal diseñada:

Id	Name	DOB	Name
1	Fred	11/02/1971	3
1	Fred	11/02/1971	3
3	Sue	Friday the 18th July 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 el segundo registro son exactamente iguales.
- Regla 5: Esta regla no está rota.

Capítulo 58: Sinónimos

Sección 58.1: Crear sinónimo

```
CREATE SYNONYM EmployeeData FOR MyDatabase.dbo.Employees
```

Capítulo 59: Sistema de información

Sección 59.1: Información básica Búsqueda por esquema

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

Una consulta de este tipo permite a los usuarios encontrar rápidamente tablas de bases de datos que contengan columnas de interés, como cuando se intenta relacionar datos de 2 tablas indirectamente a través de una tercera tabla, sin tener conocimiento de qué tablas pueden contener claves u otras columnas útiles en común con las tablas de destino.

Utilizando T-SQL para este ejemplo, se puede buscar en 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.

Capítulo 60: Orden de ejecución

Sección 60.1: 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
```

Orden en que se procesa una consulta y descripción de cada sección.

VT son las siglas de "Virtual Table" (tabla virtual) y muestra cómo se producen los distintos datos a medida que se procesa la consulta.

1. `FROM`: Se realiza un producto cartesiano (`CROSS JOIN`) 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. Sólo las filas para las que él es `TRUE` se insertan en VT2.
3. `OUTER` (`JOIN`): Si se especifica un `OUTER JOIN` (a diferencia de un `CROSS JOIN` o un `INNER JOIN`), las filas de la tabla o tablas conservadas para las que no se ha encontrado una coincidencia se añaden 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 unión y la siguiente tabla de la cláusula `FROM` hasta que se procesan todas las tablas.
4. `WHERE`: El filtro `WHERE` se aplica a VT3. Sólo las filas para las que él es `TRUE` se insertan en VT4.
5. `GROUP BY`: Las filas de VT4 se organizan en grupos en función de la lista de columnas especificada en la cláusula `GROUP BY`. Se genera VT5.
6. `CUBE | ROLLUP`: Los supergrupos (grupos de grupos) se añaden a las filas de VT5, generando VT6.
7. `HAVING`: El filtro `HAVING` se aplica a VT6. Sólo los grupos para los que es `TRUE` se insertan en VT7.
8. `SELECT`: Se procesa la lista `SELECT`, generando VT8.
9. `DISTINCT`: Se eliminan las filas duplicadas 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 o porcentaje de filas especificado se selecciona desde el principio de VC10. Se genera la tabla VT11 y se devuelve al llamante. `LIMIT` tiene la misma funcionalidad que `TOP` en algunos dialectos SQL como Postgres y Netezza.

Capítulo 61: Código limpio en SQL

Cómo escribir consultas SQL buenas y legibles, y ejemplos de buenas prácticas.

Sección 61.1: Formato y ortografía de palabras clave y nombres

Nombres de tabla/columna

Dos formas comunes de dar formato a los nombres de tabla/columna 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 deben ser generalmente singulares. Si los nombres de las tablas deben ser singulares o plurales es una cuestión muy discutida, pero en la práctica es más habitual utilizar nombres de tablas plurales.

Añadir prefijos o sufijos como `tbl` o `col` reduce la legibilidad, así que evítelos. Sin embargo, a veces se utilizan para evitar conflictos con palabras clave de SQL, y con frecuencia se emplean con desencadenadores e índices (cuyos nombres no suelen mencionarse en las consultas).

Palabras clave

Las palabras clave SQL no distinguen entre mayúsculas y minúsculas. Sin embargo, es práctica común escribirlas en mayúsculas.

Sección 61.2: Sangría

No existe una norma ampliamente aceptada. En lo que todo el mundo está de acuerdo es en que comprimir 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 línea nueva y divida las líneas si, de lo contrario, serían demasiado largas:

```
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 lo que aparece 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 SQL a la derecha).

Otro estilo habitual es colocar las 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;
```

Alinear verticalmente varias expresiones similares mejora la legibilidad:

```
SELECT Model, EmployeeID FROM Cars WHERE CustomerID = 42 AND Status = 'READY';
```

El uso de varias líneas dificulta la incrustación de comandos SQL en otros lenguajes de programación. Sin embargo, muchos lenguajes disponen de un mecanismo para cadenas de varias líneas, por ejemplo, `@"..."` en C#, `"""..."""` en Python, o `R"(...)"` en C++.

Sección 61.3: SELECT *

`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 distintas versiones de tu aplicación o de tu 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.

Así que siempre debe especificar explícitamente la(s) columna(s) que realmente desea recuperar:

```
--SELECT * <- no
SELECT ID, FName, LName, PhoneNumber -- <- si
FROM Emplopees;
```

(Cuando se realizan consultas interactivas, estas consideraciones no son aplicables).

Sin embargo, `SELECT *` no hace daño en la subconsulta de un operador `EXISTS`, porque `EXISTS` ignora los datos reales de todos modos (sólo comprueba si se ha encontrado al menos una fila). Por la misma razón, no tiene sentido listar ninguna(s) columna(s) específica(s) para `EXISTS`, por lo que `SELECT *` en realidad tiene más sentido:

```
-- lista de departamentos en los que no se ha contratado a nadie recientemente
SELECT ID, Name FROM Departments WHERE NOT EXISTS (SELECT * FROM Employees
     WHERE DepartmentID = Departments.ID AND HireDate >= '2015-01-01');
```

Sección 61.4: JOINs

Las uniones explícitas deben utilizarse siempre; las implícitas presentan varios problemas:

- La condición de unión se encuentra en algún lugar de la cláusula `WHERE`, mezclada con cualquier otra condición de filtro. Esto hace más difícil ver qué tablas están unidas y cómo.
- Debido a lo anterior, existe un mayor riesgo de cometer errores, y es más probable que se descubran más tarde.
- En SQL estándar, las uniones explícitas son la única forma de utilizar 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 uniones explícitas permiten utilizar la cláusula `USING`:

```
SELECT RecipeID, Recipes.Name, COUNT(*) AS NumberOfIngredients FROM Recipes
     LEFT JOIN Ingredients USING (RecipeID);
```

  (Esto requiere que ambas tablas utilicen el mismo nombre de columna. `USING` elimina automáticamente la columna duplicada del resultado, por ejemplo, la unión en esta consulta devuelve una única columna `RecipeID`).

Capítulo 62: Inyección SQL

La inyección SQL es un intento de acceder a las tablas de la base de datos de un sitio web inyectando SQL en un campo de formulario. Si un servidor web no está protegido contra los ataques de inyección 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 mejorar el acceso a su cuenta, ver la información privada de otra persona o realizar cualquier otra modificación en la base de datos.

Sección 62.1: Ejemplo de inyección SQL

Suponiendo que la llamada al gestor de inicio de sesión de tu aplicación web tenga este aspecto:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Ahora en login.ashx, se leen estos valores:

```
strUserName = getHttpsRequestParameterString("username");
strPassword = getHttpsRequestParameterString("password");
```

y consulta tu base de datos para determinar si existe un usuario con esa contraseña.

Así que construye 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 comillas.

Sin embargo, si uno de los parámetros contiene una comilla, el SQL que se envía a la base de datos tendrá el siguiente aspecto:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

Esto dará lugar a un error de sintaxis, porque la comilla después de la `d` en `d'Alambert` termina la cadena SQL.

Puedes corregir esto escapando las comillas en nombre de usuario y contraseña, por ejemplo:

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

Sin embargo, es más apropiado utilizar parámetros:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";
cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

Si no utiliza parámetros y se olvida de reemplazar las comillas, aunque sea en uno de los valores, un usuario malintencionado (también conocido como hacker) puede utilizarlo para ejecutar comandos SQL en su base de datos.

Por ejemplo, si un atacante es malvado, establecerá la contraseña en

```
lol'; DROP DATABASE master; --
```

y entonces el SQL se verá así:

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master;
--'";
```

Desafortunadamente para ti, esto es SQL válido, ¡y la BD lo ejecutará!

Este tipo de exploit se denomina inyección SQL.

Hay muchas otras cosas que un usuario malintencionado podría hacer, como robar la dirección de correo electrónico de todos los usuarios, robar la contraseña de todos, robar números de tarjetas de crédito, robar cualquier cantidad de datos de su base de datos, etc.

Por eso siempre hay que escapar las cadenas de caracteres.
Y el hecho de que invariablemente te olvides de hacerlo tarde o temprano es exactamente la razón por la que deberías usar parámetros. Porque si usas parámetros, entonces el framework de tu lenguaje de programación hará cualquier escape necesario por ti.

Sección 62.2: Ejemplo de inyección simple

Si la sentencia SQL se construye así

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password ='" + pw + "'";
db.execute(SQL);
```

Entonces un hacker podría recuperar tus datos dando una contraseña como `pw' or '1'='1;` la sentencia SQL resultante será:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

Ésta pasará la comprobación de contraseña para todas las filas de la tabla Usuarios porque `'1'='1'` es siempre verdadero.

Para evitarlo, utilice parámetros SQL:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, más cambios pueden ser enviados a web@petercv.com para que el nuevo contenido sea publicado o actualizado.

Traductor al español

rortegag

Özgür Öztürk	Capítulos 8 y 17
3N1GM4	Capítulo 7
a1ex07	Capítulo 37
Abe Miessler	Capítulo 7
Abhilash R Vankayala	Capítulos 5, 6, 11, 27, 30 y 32
aholmes	Capítulo 6
Aidan	Capítulos 21 y 25
alex9311	Capítulo 21
Almir Vuk	Capítulos 21 y 37
Alok Singh	Capítulo 6
Ameya Deshpande	Capítulo 26
Amir Pourmand	Capítulo 56
Amnon	Capítulo 6
Andrea	Capítulo 24
Andrea Montanari	Capítulo 36
Andreas	Capítulo 2
Andy G	Capítulo 18
apomene	Capítulo 6
Ares	Capítulo 21
Arkh	Capítulo 45
Arpit Solanki	Capítulo 6
Arthur D	Capítulo 41
Arulkumar	Capítulos 13 y 41
ashja99	Capítulos 11 y 42
Athafoud	Capítulo 24
Ayaz Shah	Capítulo 11
A_Arnold	Capítulo 18
Bart Schuijt	Capítulo 11
Batsu	Capítulo 41
bhs	Capítulo 45
bignose	Capítulo 5
blackbishop	Capítulo 25
Blag	Capítulo 17
Bostjan	Capítulos 5, 7 y 13
Branko Dimitrijevic	Capítulo 18
Brent Oliver	Capítulo 6
brichins	Capítulo 54
carlosb	Capítulos 37 y 39
Chris	Capítulo 6
Christian	Capítulo 5
Christian Sagmüller	Capítulo 6
Christos	Capítulo 6
CL.	Capítulos 1, 2, 6, 8, 10, 14, 18, 19, 21, 31, 36, 37, 41, 42, 46, 49, 61 y 62
Cristian Abelleira	Capítulo 30
DaImTo	Capítulo 30

Daniel	Capítulo 46
Daniel Langemann	Capítulos 18 y 24
dariru	Capítulo 6
Dariusz	Capítulos 10 y 19
Darrel Lee	Capítulo 40
Darren Bartrup	Capítulos 18 y 57
Daryl	Capítulos 6, 55, 56 y 58
dasblinkenlight	Capítulo 52
David Manheim	Capítulo 37
David Pine	Capítulo 6
David Spillett	Capítulo 6
day_dreamer	Capítulo 6
dd4711	Capítulo 46
dmfay	Capítulo 48
Durgpal Singh	Capítulo 6
Dylan Vander Berg	Capítulos 21 y 29
Emil Rowland	Capítulo 20
Eric VB	Capítulo 6
Florin Ghita	Capítulos 5, 18, 25, 42 y 47
FlyingPiMonster	Capítulos 5, 6, 19, 36 y 37
forsvarir	Capítulos 5 y 18
Franck Dernoncourt	Capítulos 6, 18 y 41
Frank	Capítulo 7
fuzzy_logic	Capítulo 46
Gallus	Capítulo 60
geeksal	Capítulo 1
Gidil	Capítulo 45
Golden Gate	Capítulo 41
guiguiblitz	Capítulo 9
H. Pauwelyn	Capítulo 21
Hack	Capítulo 59
Harish Gyanani	Capítulo 11
Harjot	Capítulo 50
hatchet	Capítulo 41
hellyale	Capítulo 11
HK1	Capítulo 18
HLGEM	Capítulo 18
HoangHieu	Capítulo 6
Horaciux	Capítulo 37
Hynek Bernard	Capítulo 30
Ian Kenney	Capítulo 42
iliketocode	Capítulo 6
Imran Ali Khan	Capítulos 6, 41 y 42
Inca	Capítulo 6
IncrediApp	Capítulo 55
Jared Hooper	Capítulo 6
Jason W	Capítulo 24
JavaHopper	Capítulo 5
Jaydip Jadhav	Capítulo 41
Jaydles	Capítulos 6, 7, 8 y 10
Jenism	Capítulo 37
Jerry Jeremiah	Capítulo 45
Jim	Capítulo 24
Joe Taras	Capítulo 24
Joel	Capítulos 29 y 31
John Odom	Capítulos 6, 18, 22, 32 y 56

John Slegers	Capítulos 6 y 18
John Smith	Capítulo 51
JohnLBevan	Capítulo 1
Jojodmo	Capítulo 21
Jon Chan	Capítulo 13
Jon Ericson	Capítulos 1 and 13
JonH	Capítulo 6
juergen d	Capítulos 12, 13 y 42
Karthikeyan	Capítulo 28
Kewin Björk Nielsen	Capítulos 41 y 43
KIRAN KUMAR MATAM	Capítulo 21
KjetilNordin	Capítulo 36
Knickerless	Capítulo 62
Lankymart	Capítulo 6
LCIII	Capítulo 15
Leigh Riffel	Capítulo 41
Lexi	Capítulo 25
Lohitha Palagiri	Capítulo 11
Mark Iannucci	Capítulos 6 y 18
Mark Perera	Capítulos 6 y 11
Mark Stewart	Capítulo 43
Matas Vaitkevicius	Capítulos 6, 13, 14, 19, 21 y 41
Mateusz Piotrowski	Capítulo 41
Matt	Capítulos 5, 6 y 10
Matt S	Capítulo 6
Mattew Whitt	Capítulo 6
mauris	Capítulo 37
Mihai	Capítulos 6 y 24
mithra chintha	Capítulos 8 y 25
MotKohn	Capítulo 10
Mureinik	Capítulos 10, 18 y 45
mustaccio	Capítulos 6 y 45
Mzzzzzz	Capítulo 5
Nathan	Capítulo 42
nazark	Capítulo 8
Neria Nachum	Capítulo 41
Nunie123	Capítulo 52
Oded	Capítulo 6
Ojen	Capítulos 6 y 11
omini data	Capítulos 42 y 44
onedaywhen	Capítulo 6
Ozair Kafray	Capítulo 25
Parado	Capítulos 8 y 37
Paul Bambury	Capítulo 30
Paulo Freitas	Capítulo 37
Peter K	Capítulos 42 y 46
Phrancis	Capítulos 1, 3, 4, 8, 11, 13, 18, 19, 29, 38, 41, 46, 49, 52 y 53
Prateek	Capítulos 1, 6 y 21
Preuk	Capítulo 6
Racil Hilan	Capítulo 6
raholling	Capítulo 18
rajarshig	Capítulo 26
RamenChef	Capítulo 41
Reboot	Capítulo 42
Redithion	Capítulo 11
Ricardo Pontual	Capítulo 22

Robert Columbia	Capítulos 6 y 41
Ryan	Capítulo 37
Ryan Rockey	Capítulo 60
Saroj Sasmal	Capítulos 4 y 6
Shiva	Capítulo 5
Sibeesh Venu	Capítulo 46
Simon Foster	Capítulo 25
Simone	Capítulo 7
Simulant	Capítulo 16
SommerEngineering	Capítulo 6
SQLFox	Capítulo 27
sqluser	Capítulo 6
Stanislovas Kalašnikovas	Capítulo 10
Stefan Steiger	Capítulos 11, 18, 33 y 62
Steven	Capítulo 35
Stivan	Capítulo 61
Stu	Capítulo 31
Timothy	Capítulo 6
tinlyx	Capítulo 52
Tot Zam	Capítulos 5, 13, 18, 19, 26 y 42
Uberzen1	Capítulo 23
Umesh	Capítulo 29
user1221533	Capítulo 38
user1336087	Capítulo 6
user2314737	Capítulo 34
user5389107	Capítulo 5
Vikrant	Capítulo 11
vmaroli	Capítulos 11, 19 y 41
walid	Capítulos 4 y 12
WesleyJohnson	Capítulo 5
William Ledbetter	Capítulo 42
wintersolider	Capítulo 6
Wolfgang	Capítulo 8
xenodevil	Capítulos 18 y 29
xQbert	Capítulo 6
Yehuda Shapira	Capítulo 50
ypercube	Capítulos 1 y 4
Yury Fedorov	Capítulo 6
Zaga	Capítulo 12
Zahiro Mor	Capítulos 6 y 7
zedfoxus	Capítulo 6
Zoyd	Capítulo 27
zplizzi	Capítulo 26
ɐlǝx	Capítulos 10 y 41
Алексей Неудачин	Capítulo 42
Рахул Маквана	Capítulo 18