Talento Digital para Chile:

Módulo 4
Lenguaje de Consultas a una Base de Datos

nodovirtual.awakelab.cl

JELOU futurejob by Adalid Chile

4.3.- Contenido 3: Construir sentencias utilizando el lenguaje de manipulación de datos DML para la modificación de los datos existentes en una base de datos a partir de un modelo de datos existente

Objetivo de la jornada

1. Construye sentencias de ingreso, actualización y borrado de registros en una tabla utilizando lenguaje DML de acuerdo a las condiciones solicitadas
2. Construye sentencias de ingreso, actualización y borrado de registros utilizando lenguaje DML para manipular la información de tablas con integridad referencial de acuerdo a un modelo de datos existente
3. Construye sentencias utilizando DDL para la modificación de los atributos de una tabla de acuerdo a los requerimientos planteados

4.3.1.- Sentencias para la manipulación de datos

4.3.1.1. Data Manipulation Languaje (DML)

El Lenguaje de manipulación (DML) se encarga del manejo de los datos de una base de datos como las que hemos visto hasta el momento. Algunas declaraciones DML se ven como sentencias comunes del lenguaje natural hablado o escrito y son fáciles de entender. Sin embargo, dado el alto grado de detalle de control que permite SQL sobre los datos, otras instrucciones pueden llegar a ser extremadamente complejas.

Si una instrucción DML incluye múltiples expresiones, cláusulas, predicados (Que explicaremos en este apartado) o subconsultas, entender qué es lo que ésta realmente hace puede ser una hazaña. Sin embargo, las sentencias SQL de mayor complejidad se pueden llegar a entender dividiendo sus componentes y analizando una parte a la vez.

Las instrucciones DML que encontraremos en SQL son INSERT, UPDATE, DELETE, MERGE y SELECT. Estas instrucciones pueden constar de una variedad de partes, incluidas varias cláusulas. Cada cláusula puede incorporar expresiones de valor, conectores lógicos, predicados, funciones de agregación y subconsultas. Utilizando estas cláusulas en las instrucciones SQL se puede realizar una discriminación fina de registros de la base de datos. Revisaremos aquí algunos de estos elementos:

EXPRESIONES DE VALOR

Puede utilizar expresiones de valor para combinar dos o más valores. Varias clases de valor existen expresiones, correspondientes a los diferentes tipos de datos:

```
Numérico, String, Datetime, Intervalo, Booleano,
Definido por el Usuario, Fila, Colección
```

Los tipos booleano, definido por el usuario, de fila y de colección se introdujeron con SQL: 1999. Es posible que algunas implementaciones aún no las admitan todas. Si deseamos utilizar estos datos tipos, debemos asegurarnos que la implementación que deseamos utilizar las incluya.

Expresiones de valor numérico

Para combinar valores numéricos, usamos la suma (+), resta (-), multiplicación (*), y operadores de división (/). Las siguientes líneas son ejemplos de expresiones de valor numérico:

```
12 - 7
15/3 - 4
6 * (8 + 2)
```

Los valores de estos ejemplos son literales numéricos. Estos valores también pueden ser de nombres de columna, parámetros, variables o subconsultas, siempre que estas se evalúen a un valor numérico. A continuación, se muestran algunos ejemplos:

- `SUBTOTAL + IMPUESTO + ENVÍO`
- `6 * KILOMETROS/HORAS`
- `:meses/12`

Los dos puntos en el último ejemplo indican que el siguiente término (meses) es un parámetro o una variable del lenguaje principal.

Expresiones de string

Las expresiones de valor de string pueden incluir el operador de concatenación (||). Podemos utilizar concatenación para unir dos strings, como se muestra a continuación:

Ejemplos de concatenación de strings

Expresión	Resultado
`'inteligencia ' || 'militar'`	'inteligencia militar'
`CIUDAD|| ' ' ||PROVINCIA|| ' '||REGION`	String con ciudad, provincia y región; todos separados por un espacio

Algunas implementaciones de SQL usan '+' como operador de concatenación en lugar de ||. Consulte la documentación del motor de base de datos utilizado, para ver qué operador utiliza. Algunas implementaciones pueden incluir operadores de string distintos de la concatenación, pero el estándar ISO SQL no admite dichos operadores. La concatenación se aplica a strings binarios, así como a strings de texto.

Expresiones de valor de Datetime e intervalos

Las **expresiones de valor Datetime** manejan fechas y tiempos. Datos de tipo DATE, TIME, TIMESTAMP y INTERVAL pueden aparecer en expresiones de valor Datetime. El resultado de su evaluación es siempre otro Datetime. Podemos sumar o restar un intervalo de un Datetime y especificar información de zona geográfica.

Un ejemplo de este tipo de expresiones es:

```
FechaVencimiento + INTERVAL '7' DAY
```

lo que podría ser utilizado por una aplicación, por ejemplo, para determinar que debe ejecutar cierta acción luego de pasados 7 días después de una fecha de vencimiento.

El siguiente ejemplo expresa una hora en relación a la zona horaria local:

```
TIME '18:55:48' AT LOCAL
```

Las **expresiones de valor de intervalo** manejan diferencias (Tiempo transcurrido) entre un Datetime y otro. Hay dos tipos de intervalos: año-mes y día-tiempo. Se pueden mezclar ambos en una expresión.

Como ejemplo de un intervalo, supongamos que alguien devuelve en una biblioteca un libro posteriormente a la fecha de vencimiento. Utilizando este tipo de expresiones de intervalo como se muestra a continuación podríamos calcular cuántos días de atraso posee la acción de devolución del libro y calcular la penalidad correspondiente.

```
(FechaDevolucion – FechaVencimiento) DAY
```

El intervalo debe ser año-mes o día-tiempo. Por lo tanto, es necesario especificar cuál de estos desea utilizarse. En el último ejemplo hemos indicado DAY.

Expresiones de valor booleano

Una expresión de valor booleano verifica la verdad de un predicado. Los siguientes son ejemplos de este tipo de expresiones para una columna **Clase** y un valor **SENIOR**:

```
(Clase = SENIOR) US TRUE
NOT (Clase = SENIOR) IS TRUE
(Clase = SENIOR) IS FALSE
(Clase = SENIOR) IS UNKNOWN
```

Expresiones de valores definidos por el usuario

Los tipos definidos por el usuario (UDT) representan otro ejemplo de características que llegaron a SQL: 1999 que provienen del mundo de la programación orientada a objetos. Como programadores SQL, no estamos restringidos sólo a los tipos de datos definidos en la especificación SQL. Podemos definir nuestros propios tipos de datos, utilizando los principios de tipos de datos abstractos (ADT) que se encuentran en lenguajes de programación orientados a objetos como C ++.

Las expresiones que incorporan elementos de datos de este tipo definido por el usuario deben evaluar a un elemento del mismo tipo.

Expresiones de valor de fila

Una expresión de valor de fila especifica el valor de una fila de datos. El valor de fila puede consistir de una, dos o más expresiones. Por ejemplo:

```
('Albert Einstein', 'Professor', 1918)
```

Donde esta fila podría pertenecer a una tabla Facultad, donde se muestre el nombre, el rango y el año de contratación de un colaborador.

Expresiones de valor de colección

Una expresión de valor de colección es evaluada a un arreglo.

Expresiones de valor de referencia

Este tipo de expresiones son evaluadas a un valor que referencia a otro componente de la base de datos, tal como una columna.

PREDICADOS

Los predicados son los equivalentes SQL de las proposiciones lógicas. Por ejemplo:

```
"El empleado es senior"
```

En una tabla que contiene información sobre los empleados, el dominio de la columna CLASE puede ser SENIOR, MEDIO, JUNIOR o NULL. Podemos usar el predicado CLASE = SENIOR para filtrar filas en las que el predicado es **False**, reteniendo solo aquellos para los que el predicado es **True**. A veces, el valor de un predicado en una fila es Desconocido (NULL). En esos casos, podemos optar por descartar la fila o retenerla.

CLASE = SENIOR es un ejemplo de predicado de comparación. SQL tiene seis operadores de comparación. Un predicado de comparación simple usa uno de estos operadores. A continuación, se muestran los predicados de comparación y algunos ejemplos.

Operadores y predicados de comparación

Operador	Comparación	Expresión
=	Igual que	Clase = SENIOR
<>	Distinto que	Clase <> SENIOR
<	Menor que	Clase < SENIOR
>	Mayor que	Clase > SENIOR
<=	Menor o igual que	Clase <= SENIOR
>=	Mayor o igual que	Clase >= SENIOR

En el ejemplo anterior, solo las dos primeras filas (Clase = SENIOR y Clase <> SENIOR) tienen sentido. Las demás categorías de valores no tendrán sentido en este caso no numérico pues serán ordenadas alfabéticamente en forma ascendente. Esta interpretación en casos de variables categóricas no necesariamente será lo deseado.

CONECTORES LÓGICOS

Los conectores lógicos le permiten construir predicados complejos a partir de otros simples. Por ejemplo, supongamos que deseamos identificar a los empleados más

exitosos de una base de datos de una compañía. Dos proposiciones que podrían identificar a estos empleados pueden leerse de la siguiente manera:

```
"El empleado es senior"
"Los años de experiencia del empleado son menos de 2 años".
```

Podemos utilizar el conector lógico AND para crear un predicado compuesto que identifica los registros de empleados que deseamos, como en el siguiente ejemplo:

```
Clase = SENIOR AND Experiencia < 2
```

Si usamos el conector **AND**, ambos predicados deben ser verdaderos para que el predicado compuesto sea verdadero. Podemos utilizar el conector **OR** cuando deseemos que el predicado compuesto se evalúe como verdadero si cualquiera de los componentes del predicado es verdadero. **NOT** es el tercer conector lógico. Estrictamente hablando, **NOT** no conecta dos predicados, sino que invierte el valor de verdad del predicado único al que se aplica. Por ejemplo, la siguiente expresión:

```
NOT (Clase = SENIOR)
```

Esta expresión es verdadera solo si **Clase** no es igual **a SENIOR**.

FUNCIONES DE CONJUNTO

A veces, la información que desea extraer de una tabla no se relaciona con filas individuales sino más bien a conjuntos de filas. SQL proporciona funciones de conjunto (o agregadas) para hacer frente a tales situaciones. Estas funciones son **COUNT**, **MAX**, **MIN**, **SUM** y **AVG**. Cada función realiza una acción que extrae datos de un conjunto de filas en lugar de una sola fila.

COUNT :

La función COUNT devuelve el número de filas en la tabla especificada. Para contar el número de empleados senior adelantados del ejemplo anterior, usaríamos la siguiente sentencia:

```
SELECT COUNT (*)
    FROM Empleado
    WHERE Clase = SENIOR AND Experiencia < 2;
```

MAX :

La función MAX devuelve el valor máximo que ocurre en una columna específica. Supongamos que deseamos encontrar el empleado de mayor edad de la compañía. La siguiente sentencia nos entregaría la fila apropiada:

```
SELECT Nombre, Apellido, Edad
    FROM Empleado
    WHERE Edad = (SELECT MAX(Edad) FROM Empleado);
```

Esta sentencia devuelve todos los empleados cuyas edades son iguales a la edad máxima. Es decir, si la edad del empleado mayor es 40, esta sentencia devuelve los nombres, apellidos y edades de todos los empleados que tengan 40 años.

Esta consulta utiliza una subconsulta. La subconsulta **SELECT MAX(Edad) FROM Empleado** es insertada dentro de la consulta principal.

MIN:

La función MIN funciona igual que MAX excepto que MIN busca el valor mínimo en la columna especificada en lugar del máximo. Para encontrar al empleado más joven inscrito, podemos utilizar la siguiente consulta:

```
SELECT Nombre, Apellido, Edad
    FROM Empleado
    WHERE Edad = (SELECT MIN(Edad) FROM Empleado);
```

Esta consulta devuelve todos los empleados cuya edad es igual a la edad del empleado más joven.

SUM:

La función SUM suma los valores en una columna especificada. La columna debe ser un dato de tipo numérico, y el valor de la suma debe pertenecer al rango de ese tipo. Por lo tanto, si la columna es de tipo SMALLINT, la suma no debe ser mayor que el límite superior del tipo de datos SMALLINT. Por ejemplo, si tuviéramos una base de datos con una tabla Factura que contuviera un registro de todas las ventas en la columna TotalVenta. Para encontrar el valor total de todas las ventas registradas, utilizaríamos la función SUM de la siguiente manera:

```
SELECT SUM(TotalVenta) FROM Factura;
```

AVG:

La función AVG devuelve el promedio de todos los valores en la columna especificada. Al igual que la función SUM, AVG se aplica solo a columnas con tipo de

datos numérico. Para encontrar el valor de la venta promedio, considerando todas las transacciones en la base de datos, usaríamos la función AVG de esta manera:

```
SELECT AVG(TotalVenta) FROM Factura;
```

Los valores nulos no tienen valor, por lo que si alguna de las filas de la columna TotalVenta contiene valores nulos, esas filas se ignorarán en el cálculo del valor de la venta promedio.

LISTAGG:

La función LISTAGG, nueva en SQL 2016, transforma valores de un grupo de filas en una tabla en una lista de valores, delimitada por un separador que nosotros especifiquemos. LISTAGG se utiliza a menudo para transformar los valores en las filas de la tabla en un string de valores separados por coma (CSV), o algún formato similar que sea más fácil de leer para los humanos.

LISTAGG no hace escape de los separadores, por lo que es imposible saber si un caracter encontrado es de hecho un caracter de escape, o simplemente una instancia de ese caracter que pasa a ser valor. LISTAGG se utiliza solo cuando estamos seguros de que el caracter separador elegido no aparece en ninguno de los datos que estamos agregando.
LISTAGG es una función de conjunto ordenado. El orden se logra mediante el uso de una cláusula WITHIN GROUP. La sintaxis básica es:

```
LISTAGG (<expresion>, <separador>) WITHIN GROUP (ORDER BY ...)
```

<expresion> no puede contener funciones de ventana, funciones agregadas o subconsultas. El **<separador>** debe ser un carácter.

LISTAGG elimina los valores NULL antes de agregar el resto de los valores. Si no quedan valores no nulos después de esa eliminación, el resultado de la operación LISTAGG es un valor nulo.

Si se utiliza la palabra clave DISTINCT, los valores duplicados y los valores NULL serán eliminados. La sintaxis es:

```
LISTAGG (DISTINCT <expresion>, <separador>) …
```

LISTAGG, introducido en SQL: 2016, es una característica opcional de SQL estándar y, hasta ahora, no está presente en todas las implementaciones. En aquellas

implementaciones donde está presente, es necesario consultar la documentación del sistema para ver si hay cláusulas adicionales que provean otras funcionalidades.

SUBQUERIES

Las subconsultas, como puede vimos en la sección **Funciones de Conjunto** más arriba, son consultas dentro de una consulta. En cualquier lugar donde pueda usarse una expresión en una sentencia SQL, también es posible usar una subconsulta (excepto en una función LISTAGG, como se mencionó anteriormente). Las subconsultas son herramientas poderosas para relacionar información en una tabla con información en otra tabla. Se puede anidar una consulta a una tabla, dentro de una consulta a otra tabla. Al anidar una subconsulta dentro de otra, permitimos el acceso a la información de dos o más tablas para generar un resultado final. Cuando se usan subconsultas correctamente, es posible obtener casi cualquier información que deseemos de una base de datos. No es necesario preocuparse por cuántos niveles de subconsultas soporta la base de datos. Lo más probable es que quedemos sin comprensión de lo que estamos haciendo mucho antes de que la base de datos se quede sin niveles de las subconsultas.

4.3.1.2. Obteniendo información de una tabla

Ya hemos recurrido a esta instrucción en múltiples oportunidades anteriormente, pero la revisamos aquí su uso básico nuevamente con la intención de mantener una estructura autocontenida de nuestro estudio de DML. La tarea de manipulación de datos que se realiza con más frecuencia es extraer información seleccionada de una base de datos. Es posible que deseemos obtener el contenido de una fila de entre miles desde una tabla. Es posible que deseemos obtener todas las filas que satisfagan una condición o una combinación de condiciones. Incluso podemos querer obtener todas las filas de la tabla. Una instrucción SQL, la instrucción SELECT, realiza todas estas tareas.

El uso más simple de la instrucción SELECT es obtener todos los datos de todas las filas de una tabla especificada. Si volvemos al ejemplo de la base de datos **mibasededatos**, que utilizamos como ejemplo en apartados anteriores, usamos la siguiente sintaxis:

```
SELECT * FROM profesores;
```

Con lo que obtenemos:

profesor_id	nombre	apellido	escuela	fecha_de_contratacion	sueldo
1	Juanita	Perez	Gabriela Mistral	2011-10-30	234000
2	Bruce	Lee	Republica Popular China	1993-05-22	780945
3	Juan Alberto	Valdivieso	Sagrada Concepcion	2005-08-01	3400000
4	Pablo	Rojas	E-34	2011-10-30	300000
5	Nicolas	Echenique	Bendito Corazón de María	2005-08-30	8900000
6	Jericho	Jorquera	A-18 Abrazo de Maipu	2010-10-22	67500
7	Caupolicán	Catrileo	Santiago de la extremadura	2000-10-26	780000
9	Wong	Lee	Santiago de la extremadura	2000-10-26	780000

(8 rows)

Asterisco (*) es el caracter comodín que significa todo. En este contexto, asterisco es un sustituto abreviado de una lista de todos los nombres de columna de la tabla **profesor**. Después de ejecutar esta instrucción, todos los datos en todas las filas y columnas de la tabla **profesor** aparecen en pantalla.

Las instrucciones SELECT pueden ser mucho más complicadas que la del ejemplo anterior. De hecho, algunas sentencias SELECT pueden ser tan complicadas que son muy difíciles de entender. Esta complejidad potencial es el resultado del hecho de que pueden agregarse múltiples cláusulas de modificación a la instrucción principal.

Aunque las cláusulas de modificación son un tema que podría estudiarse en forma específica, en este apartado analizaremos brevemente la cláusula WHERE, que es el método más utilizado para restringir las filas que devuelve una instrucción SELECT.

Una instrucción SELECT con una cláusula WHERE tiene la siguiente forma general:

```
SELECT lista_de_columnas FROM nombre_tabla
    WHERE condicion;
```

La lista de columnas especifica qué columnas deseamos mostrar. La instrucción muestra solo las columnas que enumeramos en esta lista. La cláusula FROM especifica de qué tabla deseamos mostrar las columnas. La cláusula WHERE selecciona las filas que satisfacen una condición especifica. La condición puede ser simple (por ejemplo, WHERE sueldo > 500000), o puede ser compuesto (por ejemplo, WHERE sueldo > 500000 AND escuela = ' Santiago de la extremadura').

El siguiente ejemplo muestra una condición compuesta dentro de una instrucción SELECT:

```
SELECT profesor_id, apellido, escuela, sueldo FROM profesor
    WHERE sueldo > 500000
    AND escuela = 'Santiago de la extremadura';
```

Que en nuestro ejemplo nos entrega:

```
 profesor_id | apellido |          escuela           | sueldo
```

```
------------+----------+---------------------------+--------
          7 | Catrileo | Santiago de la extremadura | 780000
          9 | Lee      | Santiago de la extremadura | 780000
(2 rows)
```

La palabra clave AND (Operador booleano) significa que para que una fila califique para su obtención, esa fila debe cumplir ambas condiciones: sueldo > 500000 y escuela = ' Santiago de la extremadura'.

4.3.1.3. Ingresando información a una tabla

Cada tabla de la base de datos comienza vacía. Después de crear una tabla, ésta no es más que una estructura que no contiene datos. Para que la tabla sea útil, debemos ingresar algunos datos en ella. Los datos pueden estar en una de las siguientes formas:

- **Aún sin formato digital:** Si los datos aún no están en formato digital, alguien probablemente tendrá que ingresarlos manualmente, registro por registro.

- **En algún tipo de formato digital:** Si los datos ya están en formato digital, pero tal vez no en el formato de las tablas de la base de datos que estamos utilizando, tendremos que traducirlos apropiadamente y luego insertarlos en la base de datos.

- **En el formato digital correcto:** Si los datos ya están en formato digital y en el formato correcto, éstos están listos para ser transferidos a la nueva base de datos.

A continuación, abordamos la inserción de datos a una tabla para los tres casos mencionados. Dependiendo del formato en que se encuentren los datos, es posible que puedan ser transferidos a la base de datos en una sola operación, o puede que necesiten ser ingresados un registro a la vez. Cada registro de datos que ingresamos corresponde a una fila en una tabla de la base de datos.

Agregando datos fila por fila

La mayoría de los Sistemas de Gestion de Bases de Datos (DBMS – Database Management System), tales como PostgreSQL, MySQL, SQL Server, SQLite, Oracle; poseen herramientas relacionadas que facilitan la entrada de datos basada en una interfaz gráfica, para importación de datos desde una fuente externa, como archivos, o a través de ingreso manual tipo formulario. El operador puede ingresar datos a través de esta interfaz.

En términos de instrucciones SQL, el ingreso de una sola fila en una tabla de base de datos, el comando INSERT usa la siguiente sintaxis:

```
INSERT INTO nombre_table [(columna_1, columna_2, ..., columna_n)]
```

```
    VALUES (valor_1, valor_2, ..., valor_n) ;
```

Se indica con paréntesis cuadrados [ ] que los nombres de columnas son opcionales. El orden predeterminado de la lista de columnas es el orden de las columnas de la tabla. Si ponemos los VALORES en el mismo orden que las columnas de la tabla, estos elementos serán ingresados en las columnas correctas, ya sea que especifique esas columnas explícitamente o no. Si deseamos especificar los VALORES en un orden diferente al orden de las columnas en la tabla, debemos enumerar los nombres de las columnas en el mismo orden que la lista de valores que incluyamos en VALUES.

Para ingresar un registro en la tabla **profesor**, que hemos utilizado como ejemplo en secciones anteriores, usamos la siguiente sintaxis (Que corresponde a uno de los registros mostrados en la tabla **profesor** anteriormente):

```
INSERT INTO profesor ( nombre,
                       apellido,
                       escuela,
                       fecha_de_contratacion,
                       sueldo)
    VALUES (  'Caupolicán',
            'Catrileo',
            'Santiago de la extremadura',
            '2000-10-26',
            780000
            );
```

La columna **profesor_id** no se incluye en esta sintaxis pues al ser la llave primaria y única, es generada automáticamente por el motor de base de datos. El resto de los valores corresponden a los de las columnas que hemos indicado explícitamente.

Agregando datos a un subconjunto de columnas

A veces, se desea materializar la existencia de un registro incluso si no tenemos aún todos los valores para el mismo. En una tabla de una base de datos, podemos insertar una fila sin completar los datos de todas las columnas. Si queremos la tabla en la primera forma normal, debemos insertar suficientes datos para distinguir la nueva fila de todas las demás filas de la tabla). Insertar la llave primaria de la nueva fila es suficiente para este propósito. Es posible que la tabla posea restricciones para no aceptar valores nulos de algunas columnas y en ese caso deberemos incorporarlas. Las columnas no ingresadas, y que así lo permitan poseerán valores nulos.

El siguiente ejemplo muestra una entrada de fila parcial de este tipo:

```
INSERT INTO profesor (nombre, apellido, escuela)
    VALUES ('Romilio', 'Recabarren', 'Excelentísima Alma');
```

De manera que, si hacemos una consulta para obtener los registros que posean, por ejemplo, sueldo con valor NULL,

```
SELECT * FROM profesor WHERE sueldo IS NULL;
```

obtendremos lo siguiente:

```
 profesor_id | nombre  |  apellido  |      escuela       | fecha_de_contratacion | sueldo
-------------+---------+------------+--------------------+-----------------------+--------
          10 | Romilio | Recabarren | Excelentísima Alma |                       |
(1 row)
```

Donde los valores nulos (NULL) se muestran como valores vacíos.

Agregando bloques de filas a una tabla

La carga de una tabla de base de datos una fila a la vez mediante instrucciones INSERT puede ser tedioso. Claramente, si existe la forma de ingresar los datos automáticamente, esto será una mejor opción que tener a una persona sentada en un teclado ingresando datos. La entrada automática de datos es factible, por ejemplo, si los datos existen en forma electrónica, pues alguien ya ha introducido los datos manualmente o se han recopilado de alguna forma automatizada. Si es así, no hay razón para repetir el proceso. Transferir datos de un archivo de datos a otro es una tarea que una máquina puede realizar con una mínima participación humana. Si conocemos las características de los datos de origen y la forma deseada de la tabla de destino, una máquina puede realizar la transferencia de datos automáticamente.

Desde un archivo de datos externo:

Suponga que estamos creando una base de datos para una nueva aplicación. Algunos datos que necesitamos ya existen en un archivo de computador. El archivo puede ser un de texto plano o una tabla en una base de datos creado por un DBMS diferente al que estamos utilizando. Los datos pueden estar en ASCII o en alguna otra codificación determinada. ¿Qué debemos hacer?

Lo más favorables es que los datos que deseamos ingresar estén en un formato comúnmente utilizado. Si los datos están en un formato popular, tendremos muchas posibilidades de encontrar una aplicación de conversión de formato que pueda traducirlos a uno o más formatos que nos sean de utilidad. Nuestro entorno de desarrollo probablemente pueda importar al menos uno de estos formatos. Si tenemos mucha suerte, nuestro entorno de desarrollo podrá manejar el formato de datos original directamente. Los formaros de SQL Server, MySQL y Postgres son algunos de los más utilizados. Si el formato original de los datos es menos común, es posible que debamos someterlo a una conversión.

Si los datos están en un formato antiguo, propietario u obsoleto, como último recurso, podemos contratar un servicio profesional de traducción de datos. Estas empresas se especializan en la traducción y transferencia de datos informáticos de un formato a otro. Poseen experiencia en cientos de formatos, la mayoría de los

cuales nadie ha oído hablar y nos podrán entregar los datos en el formato que especifiquemos.

Desde otras tablas:
Podemos insertar nuevos registros en una nueva tabla desde otra usando una combinación de instrucciones INSERT y SELECT. Dado que una consulta SELECT nos devolverá algunos registros, combinándola con un INSERT, éste ingresará estos registros en la nueva tabla. Podemos incluso usar un condicional WHERE para limitar o filtrar los registros que deseamos ingresar en la nueva tabla.

Ahora crearemos una nueva tabla llamada **ex-profesor**, que tendrá solo dos campos: **apellido** y **fecha_de_contratacion**. En esta tabla insertaremos filas de la tabla profesor que tienen valor no nulo en el campo **sueldo**.

```
CREATE TABLE ex_profesor
    (
        apellido                VARCHAR(50),
        fecha_de_contratacion   DATE
    );

INSERT INTO ex_profesor
    SELECT apellido,
           fecha_de_contratacion
    FROM profesor
    WHERE sueldo IS NOT NULL;
```

Con lo que, al revisar la tabla con la siguiente consulta obtenemos el siguiente resultado:

```
SELECT * FROM ex_profesor;
```

```
  apellido   | fecha_de_contratacion
-------------+-----------------------
 Perez       | 2011-10-30
 Lee         | 1993-05-22
 Valdivieso  | 2005-08-01
 Rojas       | 2011-10-30
 Echenique   | 2005-08-30
 Jorquera    | 2010-10-22
 Catrileo    | 2000-10-26
 Lee         | 2000-10-26
(8 rows)
```

En este resultado vemos que se han transferido todos los valores de **apellido** y **fecha_de_contratacion** de registros de la tabla profesor, excepto el correspondiente a **profesor_id=10** que contiene un valor **NULL** en la columna **sueldo**.

Debemos mencionar que los datos insertados deben adherir a las restricciones que posee la nueva tabla desde su creación, por ejemplo, si un campo está especificado como único, debemos insertar datos que cumplan con esa condición para que nuestra operación sea exitosa.

4.3.1.4. Actualizando la información de una tabla

Para modificar algunos datos de un registro, usamos el comando UPDATE. Este comando no puede agregar o eliminar registros (Esas responsabilidades se delegan a otros comandos). Sin embargo, si existe un registro, puede modificar sus datos incluso afectando a múltiples campos de una vez y aplicando condiciones. La sintaxis general de una instrucción UPDATE es:

```
UPDATE nombre_tabla SET
<columna1> = <valor>,
<columna2> = <valor>,
<columna3> = <valor>
. . .
WHERE <condicion>;
```

Si volvemos a nuestro ejemplo de **mibasededatos** y la tabla **profesor**, podemos probar esta manera de actualizar datos. Intentemos darle un aumento de 200.000 en su sueldo a todos los profesores que ganen menos de 1.000.000. Utilizando la sintaxis anterior aplicada a este ejemplo tendremos:

```
UPDATE profesor SET
sueldo = sueldo + 200000
WHERE sueldo < 1000000;
```

Lo que dará origen a la siguiente actualización de la tabla profesores, con un aumento en su sueldo a todos los que recibían menos de 1.000.000:

```
 profesor_id |    nombre    |  apellido  |          escuela           | fecha_de_contratacion | sueldo
-------------+--------------+------------+----------------------------+-----------------------+---------
           1 | Juanita      | Perez      | Gabriela Mistral           | 2011-10-30            |  434000
           2 | Bruce        | Lee        | Republica Popular China    | 1993-05-22            |  980945
           3 | Juan Alberto | Valdivieso | Sagrada Concepcion         | 2005-08-01            | 3400000
           4 | Pablo        | Rojas      | E-34                       | 2011-10-30            |  500000
           5 | Nicolas      | Echenique  | Bendito Corazón de María   | 2005-08-30            | 8900000
           6 | Jericho      | Jorquera   | A-18 Abrazo de Maipu       | 2010-10-22            |  267500
           7 | Caupolicán   | Catrileo   | Santiago de la extremadura | 2000-10-26            |  980000
           9 | Wong         | Lee        | Santiago de la extremadura | 2000-10-26            |  980000
          10 | Romilio      | Recabarren | Excelentísima Alma         |                       |
(9 rows)
```

Para el comando UPDATE también es posible elegir múltiples columnas para hacer actualizaciones. Por ejemplo, si deseamos llenar los datos que se encuentran vacíos en el registro correspondiente al profesor Romilio Recabarren, podemos ejecutar la siguiente instrucción, con la que modificaremos los valores NULL de **fecha_de_contratacion** y **sueldo**:

```
UPDATE profesor SET
    fecha_de_contratacion='2011-10-22',
    sueldo=500000
WHERE profesor_id=10;
```

con lo que obtenemos la actualización deseada:

profesor_id	nombre	apellido	escuela	fecha_de_contratacion	sueldo
3	Juan Alberto	Valdivieso	Sagrada Concepcion	2005-08-01	3400000
5	Nicolas	Echenique	Bendito Corazón de María	2005-08-30	8900000
1	Juanita	Perez	Gabriela Mistral	2011-10-30	434000
2	Bruce	Lee	Republica Popular China	1993-05-22	980945
4	Pablo	Rojas	E-34	2011-10-30	500000
6	Jericho	Jorquera	A-18 Abrazo de Maipu	2010-10-22	267500
7	Caupolicán	Catrileo	Santiago de la extremadura	2000-10-26	980000
9	Wong	Lee	Santiago de la extremadura	2000-10-26	980000
10	Romilio	Recabarren	Excelentísima Alma	2011-10-22	500000

(9 rows)

4.3.1.5. Borrando información de una tabla

Podemos utilizar el comando DELETE para eliminar registros de una tabla. Esto significa que podemos elegir qué registros deseamos eliminar en función de una condición o eliminar todos los registros, pero no podemos eliminar ciertos campos de un grabar usando esta declaración. La sintaxis general de la instrucción DELETE es dado a continuación:

```
DELETE FROM nombre_tabla
WHERE <condicion>;
```

Si bien poner una cláusula condicional en DELETE es opcional, es casi siempre usado, simplemente porque no usarlo haría que todos los registros de una tabla se eliminen, lo que rara vez es una necesidad real.

Volviendo a nuestro ejemplo **mibasededatos** supongamos que los profesores de mayor y menor sueldo serán reasignados a otras regiones y deben ser retirados de esta base de datos. En este caso necesitaremos una instrucción DELETE con una condición que además nos permitirá aplicar algo de lo aprendido recientemente en relación a subconsultas y a funciones de conjuntos. Para lograr lo requerido tendremos que ejecutar:

```
DELETE FROM profesor
WHERE
  sueldo = (SELECT MAX(sueldo) FROM profesor)
  OR
  sueldo = (SELECT MIN(sueldo) FROM profesor);
```

Con esto se habrán borrado los siguientes registros de la base de datos

profesor_id	nombre	apellido	escuela	fecha_de_contratacion	sueldo
5	Nicolas	Echenique	Bendito Corazón de María	2005-08-30	8900000
6	Jericho	Jorquera	A-18 Abrazo de Maipu	2010-10-22	267500

4.3.1.6. Utilización de secuencias para asignar identificadores

Hay distintas formas de asignar un identificador numérico único a los registros de tablas perteneciente a una base de datos.

INDENTIFICADOR SERIAL

La forma más común de asignar un identificador para un registro en una tabla es definir al momento de creación de la tabla que el un campo, por ejemplo **id**, será una **PRIMARY KEY**.

Las como mencionamos en apartados anteriores, en una tabla sólo puede haber una PRIMARY KEY. Además, esta columna no puede tomar valores nulos (NULL) o faltantes. La PRIMARY KEY es el identificador único de un registro.

A modo de ejemplo creemos una tabla en nuestra base de datos **mibasededatos** y llamémosla **sala**. Nuestro deseo es que cada registro de esta tabla posea un identificador único llamado id, que sea una PRIMARY KEY y que se incremente automáticamente al momento de insertar nuevos registros. Para esto es usual utilizar el tipo de dato SERIAL, que es una de las opciones más comunes:

```
SMALLSERIAL --> Hasta 32767 registros.
SERIAL --> Hasta 2147483647 registros.
BIGSERIAL --> 9223372036854775807 registros.
```

De esta forma podemos crear nuestra tabla de la siguiente forma:

```
CREATE TABLE sala (
  id  SERIAL PRIMARY KEY,
  nombre TEXT,
  area NUMERIC,
  departamento VARCHAR(40));
```

Luego de esto, si solicitamos una descripción de la tabla **sala**, obtendremos lo siguiente:

```
mibasededatos=# \d sala
                                    Table "public.sala"
    Column    |         Type          | Collation | Nullable |             Default
--------------+-----------------------+-----------+----------+---------------------------------
 id           | integer               |           | not null | nextval('sala_id_seq'::regclass)
 nombre       | text                  |           |          |
 area         | numeric               |           |          |
 departamento | character varying(40) |           |          |
Indexes:
    "sala_pkey" PRIMARY KEY, btree (id)
```

Podemos insertar datos sin necesidad de especificar manualmente el identificador de registro (id) para esta tabla, el motor de la base de datos lo definirá automáticamente en forma incremental. Al insertar los siguientes registros vemos que se confirma lo descrito:

```
mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala A', 30, 'Depto B');
INSERT 0 1
mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala B', 30, 'Depto B');
INSERT 0 1
mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala C', 40, 'Depto B');
INSERT 0 1
mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala A', 40, 'Depto A');
INSERT 0 1
mibasededatos=# SELECT * FROM sala;
id | nombre | area | departamento
----+--------+------+--------------
  1 | Sala A |   30 | Depto B
  2 | Sala B |   30 | Depto B
  3 | Sala C |   40 | Depto B
  4 | Sala A |   40 | Depto A
(4 rows)
```

Vemos que los identificadores de registro (id) han sido generados secuencialmente desde el número 1 hasta el 4.

Es posible hacer un INSERT especificando un id que aún no se haya definido, de lo contrario se generará un error pues el campo id es único. En caso de ingresar un dato con, por ejemplo, un id manual de valor 6, obtendremos un INSERT exitoso:

```
mibasededatos=# INSERT INTO sala (id, nombre, area, departamento) values (6,
'Sala C', 40, 'Depto A');
INSERT 0 1
mibasededatos=# SELECT * FROM sala;
id | nombre | area | departamento
----+--------+------+--------------
  1 | Sala A |   30 | Depto B
  2 | Sala B |   30 | Depto B
  3 | Sala C |   40 | Depto B
  4 | Sala A |   40 | Depto A
  6 | Sala C |   40 | Depto A
(5 rows)
```

Sin embargo, al continuar insertando datos con asignación automática del id, ocurrirá lo siguiente:

```
mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala D', 45, 'Depto A');
INSERT 0 1
mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala A', 60, 'Depto D');
ERROR:  duplicate key value violates unique constraint "sala_pkey"
DETAIL:  Key (id)=(6) already exists.

mibasededatos=# INSERT INTO sala (nombre, area, departamento)
                    VALUES ('Sala A', 60, 'Depto D');
INSERT 0 1
```

Como puede apreciarse SERIAL realmente asigna una secuencia que se incrementa automáticamente desde el último valor generado. En nuestro ejemplo vemos que recibimos un error pues la asignación automática luego de id=5 asigna id=6 y éste genera un error en la inserción por la imposibilidad de ingresar un id ya existente dado que se trata de una columna PRIMARY KEY.

Por lo tanto, es recomendable utilizar la asignación automática en casos que los identificadores hayan sido declarado de esa forma en la creación de la tabla. Podemos considerar la excepción de los registros que hayan sido eliminados. Como el puntero de la secuencia SERIAL estará en números superiores, el ingreso de un registro con id manual no generará conflictos pues ocupará un espacio que ha sido eliminado.

IDENTIFICADOR POR SECUENCIA PERSONALIZADA

Pueden existir casos en los que deseemos asignar identificadores que no correspondan a una secuencia continua de números enteros, sino que el incremento sea un valor mayor a 1. De la misma forma podríamos desear que se inicie en un número mayor a 1, que es lo usual en el caso de SERIAL. Es así como podemos recurrir a la creación de una secuencia, que posteriormente asignamos a nuestro identificador en la tabla donde deseemos aplicarla para la asignación automática de id.

Por ejemplo, supongamos que queremos asignar como generador del identificador de registros, para una nueva tabla **gimnasio**, una secuencia que comience en 100 y que se incremente automáticamente de 5 en 5. Para esto, creamos una secuencia que llamaremos **gimnasio_id_seq**, de la siguiente forma:

```
mibasededatos=# CREATE SEQUENCE gimnasio_id_seq
                    START WITH 100
                    INCREMENT BY 5
                    ;
```

Luego utilizamos esta secuencia como la generadora de los identificadores de nuestra tabla. Para el id implementado utilizamos el tipo de datos INT4, que nos entrega una cantidad de identificadores del orden de SERIAL.

```
smallint (int2)  --->  -32768 a +32767
integer  (int4)  --->  -2147483648 a +2147483647
bigint   (int8)  --->  -9223372036854775808 a +9223372036854775807
```

Usamos la sentencia SQL que nos permite crear la tabla gimnasio en las condiciones descritas:

```
mibasededatos=# CREATE TABLE gimnasio (
      id  INT4 PRIMARY KEY DEFAULT NEXTVAL('gimnasio_id_seq'),
      nombre VARCHAR(50), area NUMERIC, zona VARCHAR(50))
      ;
CREATE TABLE
```

Luego de lo cual hacemos algunos INSERTs para verificar la asignación de identificadores:

```
mibasededatos=# INSERT INTO gimnasio (nombre, area, zona) VALUES
                  ('Central', 100, 'Estadio 1');
INSERT 0 1
mibasededatos=# INSERT INTO gimnasio (nombre, area, zona) VALUES
                  ('Lateral', 70, 'Estadio 1');
INSERT 0 1
mibasededatos=# INSERT INTO gimnasio (nombre, area, zona) VALUES
                  ('Posterior', 80, 'Estadio 1');
INSERT 0 1

mibasededatos=# SELECT * FROM gimnasio;

 id  |  nombre   | area |   zona
-----+-----------+------+-----------
 100 | Central   |  100 | Estadio 1
 105 | Lateral   |   70 | Estadio 1
 110 | Posterior |   80 | Estadio 1
(3 rows)
```

Vemos que la asignación de identificadores se ha realizado de manera automática en las condiciones que hemos especificado.

Existen muchas otras opciones para la generación de secuencias, y formas en que podemos definir identificadores para nuestros registros en una tabla. Sin embargo, por razones de espacio, dejamos al lector y nuevo desarrollador el desafío de

complementar estos conocimientos con estudio de la documentación oficial de la implementación de base de datos que utilice.

4.3.1.7. Restricciones en una tabla

Las restricciones son las reglas que se aplican a las columnas de datos de la tabla. Estos se utilizan para evitar que se ingresen datos no válidos en la base de datos. Esto asegura la precisión y confiabilidad de ésta.

Las restricciones pueden ser de nivel de columna o de tabla. Las restricciones a nivel de columna se aplican solo a una columna, mientras que las restricciones a nivel de tabla se aplican a toda la tabla. Definir un tipo de datos para una columna es una restricción en sí misma. Por ejemplo, una columna de tipo FECHA limita la columna a fechas válidas.

Las siguientes son restricciones de uso común:

- **NOT NULL:** Garantiza que una columna no pueda tener un valor NULL.
- **UNIQUE:** Garantiza que todos los valores de una columna sean diferentes.
- **PRIMARY KEY:** Identifica con un valor único cada registro en una tabla.
- **FOREIGN KEY:** Restringe los datos basados en columnas en otras tablas.
- **CHECK:** Garantiza que todos los valores de una columna satisfagan determinadas condiciones.
- **EXCLUSION:** Asegura que, si dos filas cualesquiera son comparadas en la(s) columna(s) o expresion(es) especificada(s) con un operador determinado, al menos una de estas comparaciones devolverá un valor Falso o Nulo.

NOT NULL

De forma predeterminada, una columna puede contener valores NULL. Si no desea que una columna tenga un valor NULL, entonces debe definir dicha restricción en la definición de la columna especificando que NULL ahora no está permitido para sus datos. Una restricción NOT NULL siempre se escribe como una restricción de columna.

Por ejemplo, la siguiente instrucción crea una base de datos determinada, un nueva tabla llamada EMPRESA y agrega cinco columnas; tres de las cuales, ID y NOMBRE y EDAD, especifican no aceptar valores NULL.

```
CREATE TABLE EMPRESA(
    ID            INT    PRIMARY KEY   NOT NULL,
    NOMBRE        TEXT   NOT NULL,
    EDAD          INT    NOT NULL,
    DIRECCION     CHAR(50),
    SUELDO        REAL
);
```

UNIQUE

La restricción UNIQUE evita que dos registros tengan valores idénticos en una columna en particular. En la tabla EMPRESA, por ejemplo, es posible definir que queremos evitar que dos o más personas tengan la misma edad.

Por ejemplo, la siguiente instrucción crea una nueva tabla llamada EMPRESA2 y agrega cinco columnas. La columna EDAD se establece en UNIQUE, por lo que no puede tener dos registros con la misma edad:

```
CREATE TABLE EMPRESA2(
    ID            INT    PRIMARY KEY   NOT NULL,
    NOMBRE        TEXT   NOT NULL,
    EDAD          INT    NOT NULL    UNIQUE,
    DIRECCION     CHAR(50),
    SUELDO        REAL   DEFAULT     50000.00
);
```

PRIMARY KEY

Como hemos visto anteriormente en otros ejemplos, la restricción PRIMARY KEY identifica de forma única cada registro en una tabla de base de datos. Puede haber más columnas UNIQUE, pero solo una PRIMARY KEY en una tabla. Las usamos para hacer referencia a las filas particulares de una tabla. Además, las PRIMARY KEYs se convierten en FOREIGN KEYS en otras tablas, al crear relaciones entre tablas.

Una PRIMARY KEY es un campo en una tabla, que identifica de forma única cada fila / registro de ésta. Las llaves primarias deben contener valores únicos. Una columna de llave primaria no puede contener valores NULL.

Una tabla solo puede tener una PRIMARY KEY, que puede constar de uno o varios campos. Cuando se utilizan varios campos como llave primaria, se denominan llave compuesta.

Un ejemplo de creación de una tabla EMPRESA3 con una llave primaria, similarmente a otras que hemos creado más arriba es:

```
CREATE TABLE EMPRESA3(
    ID            INT    PRIMARY KEY   NOT NULL,
    NOMBRE        TEXT   NOT NULL,
    EDAD          INT    NOT NULL,
    DIRECCION     CHAR(50),
    SUELDO        REAL
);
```

FOREIGN KEY

Una restricción de llave foránea (FOREIGN KEY) especifica que los valores de una columna (o un grupo de columnas) deben coincidir con los valores que aparecen en alguna fila de otra tabla. Decimos que esto mantiene la **Integridad Referencial** entre dos tablas relacionadas. Se llaman FOREIGN KEY porque las restricciones son externas; es decir, fuera de la tabla. Las FOREIGN KEYs a veces se denominan llave de referencia.

Ejemplo

Por ejemplo, consideremos dos tablas, EMPRESA4 y DEPARTAMENTO1, creadas a través de las siguientes sentencias:

```
CREATE TABLE EMPRESA4(
    ID            INT    PRIMARY KEY   NOT NULL,
    NOMBRE        TEXT   NOT NULL,
    EDAD          INT    NOT NULL,
    DIRECCION     CHAR(50),
    SUELDO        REAL
);

CREATE TABLE DEPARTAMENTO1(
   ID             INT    PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      REFERENCES EMPRESA4(ID)
);
```

CHECK

La restricción CHECK permite verificar una condición para el valor que se ingresa en un registro. Si la condición se evalúa como falsa, el registro viola la restricción y no se ingresa en la tabla.

Por ejemplo, la siguiente instrucción crea una nueva tabla llamada EMPRESA5 y agrega cinco columnas. Agregamos la restricción CHECK para la columna SUELDO, para que sólo permita valores mayores que cero.

```
CREATE TABLE EMPRESA5(
    ID            INT    PRIMARY KEY   NOT NULL,
    NOMBRE        TEXT   NOT NULL,
```

```
    EDAD            INT     NOT NULL,
    DIRECCION       CHAR(50),
    SUELDO          REAL    CHECK(SUELDO > 0)
);
```

EXCLUDE

Esta restricción especifica que si se comparan dos filas en las columnas especificadas o expresión (es) que utilizan los operadores especificados, al menos una de estas comparaciones de operadores devuelve falso o nulo. Veamos este ejemplo:

```
CREATE TABLE ejemplo(
    nombre varchar(20),
    edad integer,
    EXCLUDE USING gist
     (edad WITH <>));
```

Si hacemos:

```
INSERT INTO ejemplo VALUES ('Fito', 36);
INSERT 0 1
INSERT INTO ejemplo VALUES ('Juan', 36);
INSERT 0 1
INSERT INTO ejemplo VALUES ('Miguel', '26');
ERROR:  conflicting key value violates exclusion constraint
"ejemplo_edad_excl"
DETAIL:  Key (edad)=(26) conflicts with existing key (edad)=(36).
```

Mientras cumplamos con la restricción de ingresar datos con la misma edad nuestros INSERTs no serán rechazados. En caso contrario, obtendremos el error ocurrido arriba para una edad distinta a las existentes en la tabla.

ELIMINANDO RESTRICCIONES

Para eliminar una restricción, necesitamos referirnos a su nombre de la siguiente forma:

```
ALTER TABLE nombre_tabla DROP CONSTRAINT nombre_restriccion;
```

4.3.1.8. Restricciones de FOREIGN KEY e Integridad Referencial

Una llave foránea (FOREIGN KEY) es una columna o un grupo de columnas en una tabla que hacen referencia a la llave primaria (PRIMARY KEY) de otra tabla.

La tabla que contiene la FOREIGN KEY se denomina tabla referendo, hija o secundaria. Y la tabla a la que hace referencia la FOREIGN KEY se denomina tabla referenciada, madre o principal.

Una tabla puede tener varias llaves foráneas en función de sus relaciones con otras tablas.

La restricción de llave foránea ayuda a mantener la integridad referencial de los datos entre las tablas principal y secundaria.

Una restricción de llave foránea indica que los valores en una columna o un grupo de columnas en la tabla secundaria son iguales a los valores en una columna o un grupo de columnas de la tabla principal.

SINTAXIS DE RESTRICCION DE LLAVE FORÁNEA

```
[CONSTRAINT nombre_foreign_key]
   FOREIGN KEY(columnas_foreign_key)
   REFERENCES tabla_principal(columnas_tabla_principal)
   [ON DELETE accion_eliminar]
   [ON UPDATE accion_actualizar]
```

En esta sintaxis:

- Primero, se especifica el nombre de la restricción de llave foránea después de la palabra CONSTRAINT. La cláusula CONSTRAINT es opcional. Si lo omitimos, se asignará un nombre generado automáticamente.

- En segundo lugar, especificamos una o más columnas de clave foránea entre paréntesis después de las palabras clave de FOREIGN KEY.

- En tercer lugar, especificamos la tabla principal y las columnas de llave primaria a las que hacen referencia las columnas de llave foránea en la cláusula REFERENCES.

- Finalmente, especificamos las acciones de eliminación y actualización en las cláusulas ON DELETE y ON UPDATE.

Las acciones de eliminación y actualización determinan los comportamientos cuando se elimina y actualiza la llave primaria de la tabla principal. Dado que la llave primaria rara vez se actualiza, la acción ON UPDATE no se usa a menudo en la práctica. Nos centraremos en la acción ON DELETE.

Se admiten las siguientes acciones:

- SET NULL
- SET DEFAULT
- RESTRICT
- NO ACTION
- CASCADE

EJEMPLOS

Creemos una base de datos llamada **restricciones_foreign_key**. Crearemos dos tablas, una llamada **clientes** y otra llamada **contactos**:

```
CREATE TABLE clientes(
   id_cliente SERIAL PRIMARY KEY,
   Nombre_cliente VARCHAR(255) NOT NULL
);

CREATE TABLE contactos(
   id_contacto SERIAL PRIMARY KEY,
   Id_cliente INT,
   nombre_contacto VARCHAR(255) NOT NULL,
   telefono VARCHAR(15),
   email VARCHAR(100),
   CONSTRAINT fk_cliente
      FOREIGN KEY(id_cliente)
        REFERENCES clientes(id_cliente)
);
```

En este ejemplo, la tabla **clientes** es la tabla principal y la tabla **contactos** es la tabla secundaria.

Cada cliente tiene cero o muchos contactos y cada contacto pertenece a cero o un cliente.

La columna **id_cliente** en la tabla **contactos** es la columna de llave foránea que hace referencia a la columna de llave primaria con el mismo nombre en la tabla **clientes**.

Debido a que la restricción de llave foránea en este ejemplo no tiene definidas las acciones **ON DELETE** y **ON UPDATE**, ésta asume su valor por defecto, que es **NO ACTION**.

Ejemplo NO ACTION

Insertamos los siguientes datos en las tablas que recién hemos creado con el objetivo de explorar el funcionamiento de NO ACTION:

```
INSERT INTO clientes(nombre_cliente)
VALUES('Alfanetics SpA'),
      ('Gekonus SpA');

INSERT INTO contactos(id_cliente, nombre_contacto, telefono, email)
VALUES(1,'Flor Meza','56-99-111-1234','flor.meza@alfanetics.cl'),
      (1,'Juan Meza','56-99-111-1235','juan.meza@alfanetics.cl'),
      (2,'Jericho York','56-99-222-1234','jericho.york@gekonus.com');
```

La siguiente instrucción elimina el ID de cliente 1 de la tabla de clientes:

```
DELETE FROM clientes
WHERE id_cliente = 1;
```

Debido a ON DELETE NO ACTION, se obtiene un error por violación de restricción dado que las filas que hacen referencia a este **id_cliente** todavía existen en la tabla **contactos**:

```
ERROR:  update or delete on table "clientes" violates foreign key
constraint "fk_cliente" on table "contactos"
DETAIL:  Key (id_cliente)=(1) is still referenced from table
"contactos".
```

Ejemplo SET NULL

ON DELETE SET NULL actualiza a NULL los valores de en las columnas FOREIGN KEY de la tabla secundaria en casos en que se eliminen registros de la tabla principal que posean relación con éstas.

Las siguientes instrucciones eliminan las tablas clientes y contactos que creamos en el ejemplo anterior y las vuelven a crear con FOREIGN KEY configurada con ON DELETE SET NULL:

```
DROP TABLE IF EXISTS contactos;
DROP TABLE IF EXISTS clientes;

CREATE TABLE clientes(
   id_cliente SERIAL PRIMARY KEY,
   Nombre_cliente VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE contactos(
   id_contacto SERIAL PRIMARY KEY,
   Id_cliente INT,
   nombre_contacto VARCHAR(255) NOT NULL,
   telefono VARCHAR(15),
   email VARCHAR(100),
   CONSTRAINT fk_cliente
      FOREIGN KEY(id_cliente)
        REFERENCES clientes(id_cliente)
        ON DELETE SET NULL
);
```

Volvemos a insertar los mismos datos que en el ejemplo anterior y, de igual forma, damos la instrucción de borrar el registro de la tabla principal con **id_cliente=1**. Con lo que obtenemos el siguiente resultado:

```
restricciones_foreign_key=# DELETE FROM clientes
WHERE id_cliente = 1;
DELETE 1
```

En esta caso la instrucción se ejecuta exitósamente. Vemos que los valores de **id_cliente** en la tabla contactos, que dependían de **id_cliente=1** borrado en la tabla principal han sido actualizados a valor NULL.

```
 id_contacto | id_cliente | nombre_contacto |    telefono    |          email
-------------+------------+-----------------+----------------+-------------------------
           3 |          2 | Jericho York    | 56-99-222-1234 | jericho.york@gekonus.com
           1 |            | Flor Meza       | 56-99-111-1234 | flor.meza@alfanetics.cl
           2 |            | Juan Meza       | 56-99-111-1235 | juan.meza@alfanetics.cl
(3 rows)
```

Podemos verificar que efectivamente se trata de valores nulos con la siguiente consulta:

```
SELECT * FROM contactos WHERE id_cliente IS NULL;
```

Lo que nos entrega:

```
 id_contacto | id_cliente | nombre_contacto |    telefono    |          email
-------------+------------+-----------------+----------------+-------------------------
           1 |            | Flor Meza       | 56-99-111-1234 | flor.meza@alfanetics.cl
           2 |            | Juan Meza       | 56-99-111-1235 | juan.meza@alfanetics.cl
(2 rows)
```

CASCADE

ON DELETE CASCADE elimina automáticamente todas las filas de referencia en la tabla secundaria cuando se eliminan las filas a las que se hace referencia en la tabla principal. En la práctica, ON DELETE CASCADE es la opción más utilizada.

Las siguientes instrucciones recrean las tablas de muestra. Sin embargo, la acción de eliminación de **fk_cliente** cambia a CASCADE:

```
DROP TABLE IF EXISTS contactos;
DROP TABLE IF EXISTS clientes;

CREATE TABLE clientes(
   id_cliente SERIAL PRIMARY KEY,
   Nombre_cliente VARCHAR(255) NOT NULL
);

CREATE TABLE contactos(
   id_contacto SERIAL PRIMARY KEY,
   Id_cliente INT,
   nombre_contacto VARCHAR(255) NOT NULL,
   telefono VARCHAR(15),
   email VARCHAR(100),
   CONSTRAINT fk_cliente
      FOREIGN KEY(id_cliente)
        REFERENCES clientes(id_cliente)
        ON DELETE CASCADE
);
```

Volvemos a insertar los mismos datos que en el ejemplo anterior y, de igual forma, damos la instrucción de borrar el registro de la tabla principal con **id_cliente=1**. Con lo que obtenemos el siguiente resultado:

```
restricciones_foreign_key=# DELETE FROM clientes
WHERE id_cliente = 1;
DELETE 1
```

En este caso la instrucción se ejecuta exitosamente. Vemos que los registros completos que poseían **id_cliente=1** en la tabla contactos, han sido eliminados:

```
SELECT * FROM contactos;
```

```
 id_contacto | id_cliente | nombre_contacto |    telefono    |          email
-------------+------------+-----------------+----------------+-------------------------
           3 |          2 | Jericho York    | 56-99-222-1234 | jericho.york@gekonus.com
(1 row)
```

SET DEFAULT

ON DELETE SET DEFAULT establece el valor predeterminado en la columna de FOREIGN KEY de la tabla secundaria cuando se eliminan las filas referenciadas de la tabla principal.

RESTRICT

La acción RESTRICT es similar a NO ACTION. La diferencia solo surge cuando define la restricción de clave externa como DEFERRABLE con un modo INITIALLY DEFERRED o INITIALLY IMMEDIATE. Se deja esto al lector para consultar en la documentación de la base de datos.

AGREGANDO RESTRICCIONES DE FOREIGN KEY A UNA TABLA EXISTENTE

Para agregar una restricción de FOREIGN KEY a una tabla existente, debemos utilizar la siguiente forma de la instrucción ALTER TABLE:

```
ALTER TABLE table_secundaria
ADD CONSTRAINT nombre_restriccion
FOREIGN KEY (columnas_fk)
REFERENCES tabla_principal(columnas_llave_tabla_principal);
```

Cuando agregamos una restricción de FOREIGN KEY con la opción ON DELETE CASCADE a una tabla existente, debemos seguir los siguientes pasos:

1. Eliminar las restricciones de FOREIGN KEY existentes:

```
ALTER TABLE tabla_secundaria
DROP CONSTRAINT nombre_restriccion;
```

2. Agregar una nueva restricción de FOREIGN KEY, como por ejemplo con la acción ON DELETE CASCADE:

```
ALTER TABLE tabla_secundaria
ADD CONSTRAINT nombre_restriccion
FOREIGN KEY (columnas_fk)
REFERENCES tabla_principal(columnas_llave_tabla_principal)
ON DELETE CASCADE;
```

4.3.2.- Transaccionalidad en las operaciones

4.3.2.1. Qué es una transacción y por qué son importantes

Una transacción es una unidad de trabajo que se realiza en una base de datos. Las transacciones son unidades o secuencias de trabajo realizadas en un orden lógico, ya sea de forma manual por un usuario o automáticamente por algún tipo de programa de base de datos.

Una transacción es la propagación de uno o más cambios a la base de datos. Por ejemplo, si está creando, actualizando o eliminando un registro de una tabla, entonces se está realizando una transacción en la tabla. Es importante controlar las

transacciones para garantizar la integridad de los datos y manejar los errores de la base de datos.

En la práctica, se usa agrupar múltiples consultas y ejecutarlas todas juntas como parte de una transacción.

4.3.2.2. Propiedades de las transacciones: atomicidad, consistencia, aislamiento, durabilidad

Las transacciones tienen las siguientes cuatro propiedades estándar, a las que generalmente se hace referencia con el acrónimo **ACID** (Del inglés Atomicity, Coherence, Isolation, Durability):

- **Atomicidad:** Garantiza que todas las operaciones dentro de la unidad de trabajo se completen con éxito; de lo contrario, la transacción se aborta en el punto de falla y las operaciones anteriores se revierten a su estado anterior.

- **Coherencia:** Garantiza que la base de datos cambie correctamente de estado tras una transacción confirmada con éxito.

- **Aislamiento:** Permite que las transacciones funcionen de forma independiente y transparente entre sí.

- **Durabilidad:** Asegura que el resultado o efecto de una transacción comprometida persista en caso de falla del sistema.

4.3.2.3. Control de Transacciones

Los siguientes comandos se utilizan para controlar las transacciones:

- **BEGIN TRANSACTION:** Para iniciar una transacción.

- **COMMIT:** Para guardar los cambios, alternativamente puede usar el comando END TRANSACTION.

- **ROLLBACK: P**ara deshacer los cambios.

Los comandos de control transaccional se utilizan con los comandos DML INSERT, UPDATE y DELETE únicamente. No se pueden usar al crear tablas o descartarlas porque estas operaciones se confirman automáticamente en la base de datos.

INICIO DE UNA TRANSACCIÓN
Las transacciones se pueden iniciar usando BEGIN TRANSACTION o simplemente el comando BEGIN. Estas transacciones generalmente persisten hasta que se encuentra

el siguiente comando COMMIT o ROLLBACK. Una transacción también hará ROLLBACK si la base de datos se cierra o si ocurre un error.

La siguiente es la sintaxis simple para iniciar una transacción:

```
BEGIN;
```

o

```
BEGIN TRANSACTION;
```

CONFIRMACIÓN DE UNA TRANSACCIÓN

El comando COMMIT es el comando transaccional que se utiliza para guardar los cambios invocados por una transacción en la base de datos. Este comando guarda todas las transacciones en la base de datos desde el último comando COMMIT o ROLLBACK.

La sintaxis del comando COMMIT es la siguiente:

```
COMMIT;
```

o

```
END TRANSACTION;
```

DESHACER UNA TRANSACCIÓN

El comando ROLLBACK es el comando transaccional que se utiliza para deshacer transacciones que aún no se han guardado en la base de datos. El comando ROLLBACK solo se puede utilizar para deshacer transacciones desde que se emitió el último comando COMMIT o ROLLBACK.

La sintaxis del comando ROLLBACK es la siguiente:

```
ROLLBACK;
```

EJEMPLO DE APLICACIÓN

Tomaremos como ejemplo la tabla **gimnasio** de nuestra base de datos de demostración **mibasededatos**. Borraremos todos sus datos para que esté vacía con:

```
mibasededatos=# DELETE FROM gimnasio;
```

Para utilizar transacciones desactivaremos el modo de AUTOCOMMIT de la base de datos. De esta forma cada instrucción que ingresemos no será confirmada inmediatamente, sino que se esperará la instrucción explícita del comando COMMIT:

```
mibasededatos=# \set AUTOCOMMIT OFF
```

A continuación, iniciaremos una transacción e incluiremos dos INSERTs para agregar datos a nuestra tabla que está vacía:

```
mibasededatos=# BEGIN;
BEGIN

mibasededatos=# INSERT INTO gimnasio (nombre, area, zona) values
('Central', 100, 'Estadio 1');
INSERT 0 1

mibasededatos=# INSERT INTO gimnasio (nombre, area, zona) values
('Super', 120, 'Estadio 1');
INSERT 0 1
```

Al observar lo que tenemos en nuestra tabla gimnasio luego de estas acciones veremos lo siguiente:

```
mibasededatos=# SELECT * FROM gimnasio;
id  | nombre  | area |   zona
-----+---------+------+-----------
265 | Central |  100 | Estadio 1
270 | Super   |  120 | Estadio 1
(2 rows)
```

Vemos que los datos insertados se encuentran en la tabla. Sin embargo, dado que tenemos una transacción abierta estos datos no están confirmados. Por esto, si ejecutamos el comando ROLLBACK, lo que deberíamos esperar es que estos datos sean cancelados de la tabla en cuestión:

```
mibasededatos=# ROLLBACK;
ROLLBACK

mibasededatos=# SELECT * FROM gimnasio;
id  | nombre | area | zona
----+--------+------+------
(0 rows)
```

Vemos que luego del ROLLBACK, nuestra tabla gimnasio vuelve a su estado inicial y permanece vacía. Es decir, logramos deshacer las acciones que habíamos ejecutado.

A continuación ejecutaremos las mismas acciones, con la diferencia de que finalmente introduciremos el comando COMMIT y, luego de esto, intentaremos realizar ROLLBACK:

```
mibasededatos=# BEGIN;
BEGIN

INSERT INTO gimnasio (nombre, area, zona) values ('Central', 100,
'Estadio 1');
INSERT 0 1

mibasededatos=# INSERT INTO gimnasio (nombre, area, zona) values
('Super', 120, 'Estadio 1');
INSERT 0 1
```

```
mibasededatos=# SELECT * FROM gimnasio;

 id  | nombre  | area |   zona
-----+---------+------+-----------
 275 | Central |  100 | Estadio 1
 280 | Super   |  120 | Estadio 1
(2 rows)
```

A continuación, hacemos COMMIT de las instrucciones realizadas hasta el momento. Y luego de esto intentamos hacer ROLLBACK de estas acciones. Vemos a continuación que no es posible volver al estado inicial de nuestra tabla y por lo tanto el comando COMMIT ha confirmado los cambios que introdujimos en nuestra tabla.

```
mibasededatos=# COMMIT;
COMMIT
mibasededatos=# ROLLBACK;
WARNING:  there is no transaction in progress
ROLLBACK
mibasededatos=# SELECT * FROM gimnasio;
id  | nombre  | area |   zona
-----+---------+------+-----------
275 | Central |  100 | Estadio 1
280 | Super   |  120 | Estadio 1
(2 rows)

mibasededatos=# ROLLBACK;
ROLLBACK
mibasededatos=# SELECT * FROM gimnasio;
id  | nombre  | area |   zona
-----+---------+------+-----------
275 | Central |  100 | Estadio 1
280 | Super   |  120 | Estadio 1
(2 rows)
```

4.3.3.- Referencias

[1] Rahul Batra, SQL Primer An Accelerated Introduction to SQL Basics, 2018.

[2] PostgreSQL Tutorial
https://www.tutorialspoint.com/postgresql/

[3] Manpreet Kaur, PostgreSQL Development Essentials, 2016.

[4] PostgreSQL Foreign Key
https://www.postgresqltutorial.com/postgresql-foreign-key/

[5] PostgreSQL Documentation
https://www.postgresql.org/docs/12/index.html

[6] Allen G. Taylor, SQL, 2019.