UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

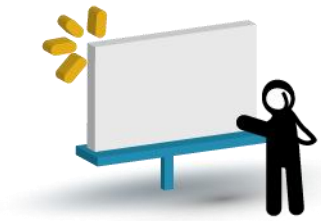
Centro de e-Learning

Diplomatura en Bases de Datos

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 1: Introducción y Fundamentos de Bases de Datos

Unidad 3: Objetos de las bases de datos



Presentación:

En esta Unidad exploramos los objetos que están contenidos en una base de datos relacional, vemos cómo crearlos, editarlos, usarlos y destruirlos mediante el DDL (Data Definition Language).

Objetivos:

Que los participantes:

- Comprendan la definición de los diferentes objetos que viven dentro de una base de datos.
- Entiendan las ventajas y desventajas de crear un índice.
- Entiendan como se optimizan las consultas y como se relaciona con la creación de índices.
- Puedan estimar las cargas administrativas relacionadas con el mantenimiento de índices.
- Incorporen las ventajas de definir funcionalidad dentro de las bases de datos
- Aprendan a programar en SQL.
- Tengan criterio para decidir cuando conviene utilizar un cursor y cuando no.

Bloques temáticos:

1. Tablas.
2. Índices.
3. Vistas.
4. Procedimientos almacenados.
5. Funciones.
6. Triggers.
7. Estructuras de control en SQL.
8. Cursores en SQL.

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

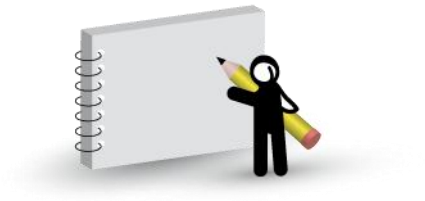
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

**** El MEC es el modelo de E-learning colaborativo de nuestro Centro.***



Tomen nota:

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Tablas

Ya hemos venido hablando de las tablas como los bloques fundacionales en los cuales guardamos la información que constituye la razón de ser de las bases de datos.

Para cada uno de los objetos que almacenará la base de datos vamos a ver también:

- Justificación de su existencia
- Estructura
- Características
- Sentencias para su creación, modificación y destrucción.

Respecto de la justificación de la existencia de las tablas no hace falta decir mucho. Las tablas guardan los datos que justifican la existencia de las bases de datos. El resto de los objetos se organizarán alrededor de ellas.

Ya hemos discutido que las tablas tienen filas y columnas. Todos los valores que almacenamos en una dada columna son del mismo tipo de datos, esto es, o son todos números, todos fechas o todos cadenas de caracteres.

Nota Conceptual:

Tipo de Dato es la propiedad de un valor que determina su dominio (esto es: que valores puede o no puede tomar) determina que operaciones se le pueden aplicar y también como será representado internamente.

Los distintos sistemas de bases de datos han ido incorporando en sus diversas versiones más y más tipos de datos para servir a diferentes necesidades.

Para que toda esta ensalada sea comprensible vamos a empezar por separar:

- Numéricos
- Cadenas de caracteres
- Fechas
- Lógicos
- Objetos grandes

Los tipos numéricos podemos a su vez clasificarlos en enteros y números con punto decimal.

Para los enteros nos quedan dos divisiones:

- Con signo/sin signo
- Con diferentes longitudes (1 byte, 2 bytes, 4 bytes y 8 bytes)

Los números con punto decimal pueden tener la posición del punto fija (por ejemplo siempre tienen dos decimales) o pueden tener el punto decimal flotante.

En el primer caso la longitud del número y la posición del punto decimal forman parte de la definición del tipo de dato.

Para el punto decimal flotante el número se almacena como dos partes. Una primera parte tiene que ver con el número que queremos representar como si tuviera una sola cifra significativa (mantisa). La segunda parte indica cuantos lugares debe correrse la coma (exponente).

Ejemplo:

Si tengo que guardar 1325.73 entonces:

- mantisa: 1.32573
- exponente 003

Y, de nuevo viene el tamaño del número que queremos representar. Pueden ser de simple precisión en cuyo caso consume 4 bytes o de doble precisión en cuyo caso tenemos 8 bytes.

Cuando vamos por doble precisión no sólo cambia la cantidad de cifras que guardamos de la mantisa sino que también aumenta la cantidad de bits asignada al exponente.

Cuando al realizar una operación válida un tipo no puede representarse entonces se produce un error. Ese error puede ser de dos tipos:

- overflow
- underflow

El primero es fácil de entender. Supongamos que estamos trabajando con enteros con signo de 1 byte. Esto significa que el número más bajo que podemos tener es -128 y el más alto es +127

Supongamos que tenemos guardados dos datos, uno es 100 y el otro es 50 y los queremos sumar. El resultado nos daría 150 que está fuera del dominio de los enteros de 1 byte con signo y que, por lo tanto, no se puede representar.

Cuando el motor de base de datos trata de realizar esa operación no puede y devuelve el error de overflow.

El underflow tiene que ver con un resultado que da un número tan cercano a cero que la mantisa no permite representarlo.

Normalmente el error por overflow aparece siempre que se produce. Por defecto la mayoría de las herramientas ignora la aparición del underflow y pone cero en el resultado.

Para muchas aplicaciones de cálculo esto puede ser una buena aproximación. Para otras puede ser fatal. Dentro de una base de datos las primeras son las más frecuentes.

Respecto de las cadenas de caracteres tenemos dos grandes tipos:

- Longitud fija
- Longitud variable

Las cadenas de caracteres de longitud fija consumen el mismo espacio dentro de cada registro.

Las cadenas de caracteres de longitud variable emplean espacio adicional para almacenar la longitud de cada valor. Son mucho más eficientes cuando las longitudes de los datos almacenados son variables.

Ya veremos que cuando las longitudes de las cadenas de caracteres son importantes estos tipos de datos se vuelven inconvenientes y debemos recurrir a otros tipos que todavía no consideramos.

Hay dos cosas más que debemos considerar respecto de los tipos de datos que guardan las cadenas de caracteres:

- Representación interna
- Sensibilidad a las mayúsculas y acentos

Al principio bastaba con registrar los caracteres alfabéticos, la puntuación, los números y algunos caracteres de control. En esta época 128 caracteres sobraban.

A medida que fuimos incorporando más símbolos relacionados con diferentes idiomas nos vimos en la necesidad de ampliar la cantidad de caracteres.

Un mapa de caracteres consiste en darle a cada símbolo un número. Como finalmente en la base de datos vamos a registrar sólo bits entonces será preciso saber cuál es el mapa de caracteres que corresponde para convertir un conjunto de bits en un símbolo.

El primer mapa de caracteres que yo recuerdo haber usado es ASCII y tenía 128 caracteres.

Luego, en los años 60 IBM generó EBCDIC y lo usó como base para el trabajo en los Mainframes del período.

En 1981 la propia IBM extendió ASCII a 256 caracteres incluyéndolo en la IBM PC. Consecuentemente MS-DOS también adoptó el ASCII de 256 caracteres.

Desde 1991 un consorcio formado por empresas e instituciones académicas mantiene Unicode como una forma única de mapear todos los caracteres de todas las lenguas y disciplinas en forma única. Un carácter unicode está compuesto por 4 bytes con lo cual habría, en principio habría del orden de 4.000.000.000 de caracteres posibles.

El otro punto que debemos discutir es la sensibilidad a mayúsculas y acentos. ¿Es lo mismo que una palabra diga Papa que papá que papa?

Cuando se inicia una base de datos hay que elegir los criterios con los que se harán esas comparaciones y como se considerarán los caracteres a los efectos del orden.

Fechas:

Seguramente todos recordamos el problema suscitado al final de los años 90 con el cambio de siglo. Existió el temor de que muchos sistemas no estuvieran debidamente preparados para tomar.

Cuando empezamos a discutir la historia de las bases de datos señalamos lo escaso que era el espacio de almacenamiento en esa época. En dicho contexto, pensar en usar cuatro dígitos para el año era inaceptable.

De hecho hoy todavía las fechas se registran internamente como números lo cual facilita comparaciones y diferencias. Ese número suele ser la cantidad de días transcurridos desde alguna fecha. (En general esa fecha correspondiente al día cero la elije el fabricante de la herramienta)

Si el campo no es del tipo fecha pura sino fecha - hora entonces las horas, minutos, segundos y fracciones se almacenan como las fracciones del día. Por ejemplo 12 hs equivalen a 0,5

Para realizar la manipulación de los datos tipo fecha y, sobre todo, para su paso de un sistema a otro suelen producirse abundantes rompederos de cabeza. Este es uno de los puntos que siempre conviene vigilar cada vez que se hace una importación o migración.

Para la manipulación de las fechas las bases de datos nos proveen variadas funciones:

- Para sumar un intervalo de tiempo a una fecha
- Para extraer una parte de una fecha (la hora o el mes)
- Para obtener la diferencia entre dos fechas (expresada en alguna unidad de tiempo)

Cuando vayamos viendo los distintos motores de bases de datos iremos señalando las sentencias que cada uno usa para la manipulación de fechas. Desgraciadamente en este sentido las cosas no están particularmente uniformizadas entre los productos de los distintos proveedores.

Otro tipo de dato es el lógico. Consiste en almacenar bits. Es la unidad más reducida con la que podemos contar.

Puede tratarse tanto de un valor único como de un grupo de bits.

A veces la cantidad de texto que queremos contener dentro de una variable del tipo cadena de caracteres es mayor a la cantidad máxima que este tipo admite (que, como se imaginan, depende de la implementación concreta)

En otras ocasiones queremos almacenar dentro de la base de datos documentos y objetos que no son texto puro y que están llenos de caracteres especiales (como las comillas simples que SQL usa para delimitar las cadenas de caracteres) por lo que tendríamos todo tipo de problemas al manipularlos como texto.

Para responder a ambas necesidades los proveedores de motores de bases de datos han introducido los tipos "Grandes objetos binarios" y "Texto" En ambos casos las implementaciones de los distintos fabricantes tienen el máximo nivel de divergencia. Su uso no está tan difundido como los otros datos.

En la mayoría de los casos los programadores han optado por dejar guardados en el sistema de archivos estos objetos binarios y guardar dentro de la base de datos el nombre de archivo de manera que le permita a una aplicación acceder al mismo. Por supuesto esto trae complicaciones de integridad y seguridad.

Al ir recorriendo los distintos motores veremos cómo los fabricantes enfrentaron y resolvieron este problema. También conviene señalar acá que esta dificultad incluye como hacer búsquedas eficientes en este tipo de campos y constituye el punto de partida de las bases de datos documentales que veremos también más adelante.

Sentencias para su creación, modificación y destrucción

Asi como ya vimos que para manejar los datos que guardamos en las tablas hay sentencias de INSERT, UPDATE y DELETE y ya empezamos a discutir el formato que se usa vamos a empezar a estudiar lo que pasa con las tablas.

Vamos a considerar:

- CREATE TABLE
- ALTER TABLE

- DROP TABLE
- TRUNCATE TABLE

Create Table es la sentencia que usamos para generar una tabla nueva.

La sintaxis general es:

CREATE TABLE [dueño de la tabla].[nombre de la tabla]

(

[nombre del campo] [tipo del campo](longitud) NULL,

...

)

Repasemos los elementos:

Las cosas que escribo entre [] es para indicar que se trata de un único concepto. En SQL Server usamos esos corchetes cuando dentro del nombre que estamos escribiendo hay caracteres en blanco y de esa forma nos aseguramos que el motor de base de datos los toma como una única cosa.

Dueño de la tabla: Es el usuario que tendrá primariamente todos los derechos sobre la tabla que vamos a crear.

Nombre de la tabla: Es el nombre por el que vamos a identificar a la tabla más adelante. Deberá ser único. (otra tabla deberá llamarse distinto)

Nombre del campo: Es el nombre que vamos a dar a un campo. Debe ser único dentro de la tabla pero puede repetirse en otras tablas. Mi sugerencia es que al mismo atributo lo llamen siempre igual en todas las tablas en las que aparezca.

Tipo del campo: Es el tipo de dato que vamos a guardar en ese campo. Distintos motores de bases de datos los organizan de diferentes maneras.

Veamos los más comunes:

Tipo de dato	Descripción
INT	para almacenar enteros con signo
FLOAT	para almacenar números con punto flotante
DATE	para almacenar fechas (año, mes y día)
DATETIME	Para almacenar fechas con horas, minutos y segundos
CHAR	Para almacenar una cadena de caracteres. Se debe indicar la longitud entre paréntesis y se almacena siempre esa cantidad de caracteres
VARCHAR	Para almacenar una cadena de caracteres. Se debe indicar la longitud máxima y se almacenan sólo los caracteres que correspondan.

NULL/ NOT NULL

El valor NULL representa los datos faltantes. Al crear una tabla podemos especificar para cada campo si vamos a permitir que se guarde un registro con este dato faltante o no.

Para indicar que vamos a permitir un valor NULL escribimos en la sentencia create table NULL y para indicar que no lo vamos a permitir ponemos NOT NULL

Los valores que conforman una clave primaria NUNCA PODRAN ser nulos.

Ejemplo:

Vamos a guardar los siguientes datos de un cliente:

Número de cliente (no podrá ser NULL)

Fecha de alta (no podrá ser NULL)

Razón social (no podrá ser NULL)

Vendedor (podrá ser NULL)

Entonces la sentencia para crear la tabla será:

```
CREATE TABLE dbo.clientes
(
        IdCliente INT NOT NULL,
        FechaAlta DATE NOT NULL,
        RazonSocial VARCHAR(50) NOT NULL,
        Vendedor VARCHAR(50) NULL
)
```

En este momento es oportuno realizar el ejercicio práctico 3.1

Alter Table es la sentencia que usamos para modificar una tabla.

Su sintaxis es idéntica a Create Table.

Si tratamos de crear una tabla que ya existe el motor de base de datos nos devolverá un error.

Si tratamos de alterar una tabla que no existe el motor de base de datos nos devolverá un error.

```
ALTER TABLE [dueño de la tabla].[nombre de la tabla]
(
    [nombre del campo] [tipo del campo](longitud) NULL,
     ...
)
```

Ejemplo:

Si en la tabla para los clientes queremos eliminar el nombre del vendedor y reemplazarlo por un IdVendedor que luego apuntará a una tabla de vendedores sería:

```
ALTER TABLE dbo.clientes
(
    IdCliente INT NOT NULL,
    FechaAlta DATE NOT NULL,
    RazonSocial VARCHAR(50) NOT NULL,
    IdVendedor INT NULL
)
```

Para borrar una tabla sólo necesitamos su dueño y su nombre:

```
DROP TABLE [dueño de la tabla].[nombre de la tabla]
```

Ejemplo:

DROP TABLE dbo.clientes

Al borrarla desaparecen tanto los datos como la estructura.

Cualquier nueva referencia (Select, Insert, Update, Delete) a la tabla que invoquemos dará error.

También dará error borrala de nuevo o hacer un alter table. Lo que sí es ahora posible es crearla!

Finalmente, si queremos borrar todo el contenido de una tabla pero sin tocar la estructura entonces podemos recurrir a la sentencia TRUNCATE TABLE

Ya vimos que para borrar registros usábamos la sentencia DELETE.

¿Cuál es la diferencia entre DELETE y TRUNCATE TABLE?

En primer lugar DELETE permite seleccionar que registros voy a borrar y que registros no. TRUNCATE TABLE es masivo. Pero hay una diferencia más profunda. DELETE sigue el camino normal del motor de base de datos para tratar los datos que es registrando transacciones.

El registro de transacciones va anotando cada cosa que hay que hacer sobre los datos de manera que, en principio, sería posible des-hacerla. Ya volveremos sobre este concepto más tarde.

Por ahora nos interesa subrayar que TRUNCATE TABLE no pasa por el log de transacciones y los cambios se operan directamente sobre los datos.

2. Índices

Como pasa con los libros los índices son ayudas para buscar.

Imaginemos que tenemos 1.000 números en una tabla y queremos encontrar uno en particular. (Por ejemplo queremos encontrar el cliente cuyo número es 537)

Una forma de encarar este problema es tomar el primer registro de la tabla y preguntarle ¿Sos el 537? si nos dice que si, bien, ya lo encontramos, si nos dice que no, pasamos al siguiente y le preguntamos también ¿Sos el 537? y así vamos siguiendo hasta que lo encontramos.

A veces recorreremos los 1000 números para encontrarlo cerca del final, a veces, con más suerte, lo encontramos al principio. Si hacemos muchas búsquedas no es difícil saber que, en promedio, tendremos que preguntar 500 veces en cada búsqueda.

¿Qué pasa si la tabla, en vez de 1000 números tiene 10000? Pues que, en promedio, vamos a tener que hacer 5000 preguntas antes de encontrar nuestro cliente.

Si tratamos de generalizar un poco más nos damos cuenta que si la tabla tiene N registros el número de preguntas va a estar cerca de N/2 (siempre en promedio)

Esto nos ayuda a pensar que el tiempo que va a demorar la búsqueda va a crecer más o menos linealmente cuando aumente la cantidad de datos. (Esto es, es proporcional a N)

Cuando las tablas van creciendo esto se vuelve un verdadero problema. ¿Lo podremos solucionar?

Imaginemos que podemos guardar los datos ordenados por el número de cliente.

Si tenemos 1000 clientes entonces consultamos a la mitad de la tabla y preguntamos es mayor, menor o igual a 537?

Si es igual ya lo encontré, si es mayor voy a buscar en la primera mitad de la tabla y si es menor buscaré en la segunda.

Igual que antes repito el paso buscando en la mitad de la parte de la tabla que me toca.

Notemos que mi búsqueda se va concentrando en un conjunto cada vez más pequeño.

Primer paso busco entre 1000

Segundo paso busco entre 500

Tercer paso busco entre 250

Cuarto paso busco entre 125

No es difícil ver que, más o menos rápidamente, la búsqueda llega a un único dato.

¿Cuantos pasos?

Si parto de 1000 datos con 10 pasos me alcanza.

Si parto de 2000 datos me alcanza con 11

Si parto de 4000 datos me alcanza con 12

¡Esto es muy distinto!

En vez de crecer linealmente con N cada vez que duplico N solo incremento la búsqueda en una sola pregunta.

Esto parece muy conveniente sobre todo para los grandes conjuntos de datos que están cada vez más de moda.

Pero, hay un problema, es preciso tener los datos ordenados...

Consideremos por un momento que estamos con una tabla que tiene facturas. Por un lado nos puede interesar buscar una factura por su número, pero también podría interesarnos ubicar todas las facturas de un cierto cliente.

¿Las tendremos ordenadas por número de factura o por número de cliente?

Los dos órdenes nos interesarían pero, es claramente imposible tener la tabla ordenada a la vez por dos criterios distintos.

La solución que se encontró para esto es tener un archivo independiente ordenado, por ejemplo con el número de cliente, que además tiene, en cada registro, un puntero que le indica donde encontrar la factura correspondiente.

Estos archivos auxiliares, ordenados por el criterio que queremos usar en las búsquedas son los índices.

¿Por qué, entonces, no hacer índices por cada criterio posible de búsqueda y así todas las búsquedas serán muy rápidas?

Hay dos contras que debemos considerar:

- Espacio consumido: Los índices no contienen información original pero igual ocupan espacio de almacenamiento. El espacio de almacenamiento es cada vez más barato pero NUNCA es gratis.
- Tiempo de actualización: cuando la información contenida en una tabla cambia entonces es necesario que se actualice el contenido de un índice. Por suerte el motor de base de datos se encarga de esto y no tendremos que hacerlo nosotros.

Sin embargo el tiempo necesario para modificar los índices se consumirá. Esto es algo aún peor cuando insertamos nuevos registros. En el caso de una tabla es fácil pensar que los agregamos "al final".

En el caso de un índice no pasará tal cosa. Probablemente al modificar o agregar un dato (UPDATE o INSERT) sea preciso incorporar registros dentro del índice en algún lugar en el medio de este "corriendo" los datos siguientes.

Para mitigar este problema al borrar un registro en la tabla (DELETE) se deja el espacio disponible por si después hay que insertar otro sin que haga falta correr a los demás.

Cuando se vuelve imprescindible correr a los demás entonces se deja, a propósito, espacio disponible para futuras inserciones. ¿Cuánto espacio se deja disponible? Depende.

Si dejamos mucho, las siguientes inserciones serán más rápidas (en promedio) pero el índice ocupará más espacio. Si se deja poco el índice será más compacto pero, en promedio, las inserciones serán más lentas.

Habrá que pensar que nos molesta más si la velocidad de respuesta o el espacio.

Tenemos dos tipos de índices, los "clusterizados" y los no "clusterizados"

Los índices no "clusterizados" son los que mejor responden a la descripción que hicimos más arriba. Reflejan exactamente lo que en un libro llamamos índice. Tenemos dentro del índice la lista de las cosas de las que habla el libro y el número de página donde ir a buscarlo.

Los índices "clusterizados" por otra parte consisten en mantener ordenada la misma tabla. Como la tabla sólo puede estar ordenada por un criterio sólo puede existir un índice "clusterizado" frente a varios "no clusterizados"

¿Cómo creamos los índices?

Los distintos motores de bases de datos nos presentan herramientas con interfaces gráficas que pueden hacer esta tarea. Por ahora vamos a ceñirnos a la creación mediante sentencias y repasaremos lo que ocurre con las interfaces gráficas cuando lleguemos a las herramientas específicas.

CREATE CLUSTERED INDEX [nombre del índice] ON [dueño de la tabla].[nombre de la tabla]

(

[lista de campos] [orden]

)

Nombre del índice: se trata de un nombre único que le daremos al índice. Sólo lo vamos a usar si fuera preciso invocar el uso del índice en forma explícita. Esto es muy raro por lo que, en general, poca gente se preocupa por el nombre que le dará al índice y deja que la base de datos use lo que propone por defecto que tiene la virtud de garantizar la unicidad.

Dueño de la tabla: ya lo hemos comentado más arriba cuando discutimos CREATE TABLE

Nombre de la tabla: es la tabla relacionada con el índice que estamos creando.

Lista de campos: son los nombres de los campos que se incluirán en el índice. Conviene notar que si ponemos dos campos entonces el orden se hará por la combinación de ambos campos. Por ejemplo si hay dos campos en un índice: A y B podremos buscar ágilmente por A y por ambos pero el índice no servirá para buscar por B.

Orden: indicamos si será ascendente o descendente con ASC o DESC.

En este punto se recomienda la realización de los ejercicios práctico 3.2 y 3.3

Para crear un índice no "clusterizado" la sintaxis es:

CREATE NONCLUSTERED INDEX [nombre del índice] ON [dueño de la tabla].[nombre de la tabla]

(

[lista de campos] [orden]

)

En este punto se recomienda realizar el ejercicio conceptual 3.1

3. Vistas

Una de las cosas en las que las bases de datos dieron un gran paso adelante sobre los aplicativos que guardaban los datos en los sistemas de archivos es que pudieron manejar con más niveles de detalle los permisos a diferentes usuarios.

A veces es necesario que algún tipo de usuarios puedan ver algunos campos pero no todos de una tabla.

A veces es preciso que la información que por razones de normalización está guardada en varias tablas sea vista como una sola por algún grupo de usuarios.

Para responder a estos dos tipos de necesidades recurrimos a las vistas.

Nota Conceptual:

Vista es una consulta que se presenta como una tabla virtual. Una vista se apoya en un conjunto de tablas. Las vistas tienen la misma estructura que una tabla: filas y columnas.

Ejemplo:

Supongamos que tenemos una tabla de clientes donde están las correspondientes direcciones. Sin embargo, por las razones de normalización que ya hemos discutido, las provincias, las localidades y las calles están en tablas separadas.

Para facilitar el acceso de un proveedor (por ejemplo del que nos cubre la logística) tengo que darle acceso a algunos datos de los clientes. Pero, por ningún motivo quiero que tengan un acceso a todos los atributos (por ejemplo no quiero compartir con ellos cual es el límite de crédito que tengo asignado a cada uno)

Una forma transparente de lograr esto es crear una vista que le permita a mi proveedor conectarse con un usuario que no tiene ningún permiso sobre las tablas subyacentes pero que tiene permiso de select sobre la vista que definimos para el.

Veamos ahora como se realiza la sintaxis para:

- Crear una vista
- Usar una vista
- Eliminar una vista

Para crear una vista usamos:

CREATE VIEW [nombre de la vista]

AS [sentencia de selección]

nombre de la vista: es el nombre único que asignamos a la vista

sentencia de selección: es cualquier sentencia select válida que querramos usar.

Ejemplo:

Consideremos la tabla clientes que hemos creado antes del Ejercicio Práctico 3.1 mediante:

```
CREATE TABLE dbo.clientes
(
        IdCliente INT NOT NULL,
        FechaAlta DATE NOT NULL,
        RazonSocial VARCHAR(50) NOT NULL,
        Vendedor VARCHAR(50) NULL
)
```

Vamos a crear una vista para ver los clientes que hemos dado de alta en el año 2018

```
CREATE VIEW Clientes2018
AS
SELECT * FROM dbo.clientes WHERE YEAR(FechaAlta) = 2018
```

En este momento se sugiere la realización de los ejercicios prácticos 3.4 y 3.5

Para consultar una vista lo hacemos con la misma sentencia SELECT que habríamos utilizado para consultar una tabla o varias tablas.

En este momento se sugiere la realización del ejercicio práctico 3.6

En este momento se sugiere realizar el ejercicio conceptual 3.2

En este punto conviene realizar el ejercicio práctico 3.7

Para eliminar una vista usamos (otra vez) la sentencia DROP:

DROP VIEW [nombre de la vista]

En este momento conviene ejecutar el ejercicio práctico 3.8

4. Procedimientos almacenados

Muchas veces queremos dejar almacenado y disponible un conjunto de instrucciones SQL que nos permitan obtener un resultado determinado. Para esto se incorporaron los procedimientos almacenados.

Una gran cosa que los procedimientos nos permiten es pasarles información que condicione su funcionamiento. A esos datos adicionales que condicionan su funcionamiento los llamamos parámetros.

Nota Conceptual:

Un procedimiento almacenado (en inglés: stored procedure) es un conjunto de sentencias con un orden (o procedimiento) que guardamos dentro de la misma base de datos.

Su implementación varía con los diferentes proveedores del motor de base de datos. Es capaz de recibir parámetros y de devolver un resultado.

Tendremos que ver ahora como:

- Crear un procedimiento almacenado
- Usar un procedimiento almacenado
- Eliminar un procedimiento almacenado

Para crear un procedimiento almacenado la sentencia es:

CREATE PROCEDURE [dueño].[nombre del procedimiento]

[parametro 1 tipo de dato del parámetro 1],

...

[parametro n tipo de datos del parámetro n]

AS

BEGIN

[paquete de sentencias que queremos que se ejecuten]

END

Donde:

- Dueño: es el usuario de base de datos que tendrá todos los permisos sobre el procedimiento almacenado.
- Nombre del procedimiento: es el nombre único por el que se identificará al procedimiento dentro de la base de datos.
- Parámetros: son las distintas piezas de información (variables) que le pasamos al procedimiento almacenado para condicionar su funcionamiento.

Los parámetros pueden ser varios por eso ponemos el parámetro 1, en el renglón siguiente ... para dar a entender que habrá varios parámetros y al final parámetro n.

Cada parámetro va separado por un espacio en blanco del tipo de dato que cargará ese parámetro.

Los distintos parámetros se separan por comas.

El último parámetro no tiene una coma tras de su tipo de dato sino que, a continuación, tras un espacio en blanco va la sentencia AS para indicar dónde comienza la parte que trae las sentencias.

Con las sentencias BEGIN y END delimitamos un bloque de código.

Entre BEGIN y END vienen las sentencias que indican lo que el procedimiento almacenado tiene que hacer.

Ejemplo:

Vamos a construir un procedimiento almacenado que cumpla la misma función que la vista que hemos definido en el Ejercicio Práctico 3.4.

Vamos a empezar por recordar cómo era la tabla donde se apoyaba la vista:

```
CREATE TABLE dbo.clientes
(
        IdCliente INT NOT NULL,
        FechaAlta DATE NOT NULL,
        RazonSocial VARCHAR(50) NOT NULL,
        Vendedor VARCHAR(50) NULL
)
```

El ejercicio práctico 3.4 nos pedía obtener todos los clientes del vendedor PEPE.

Para hacerlo con un procedimiento almacenado:

```
CREATE PROCEDURE [dbo].[ClientesPepe]
AS
BEGIN
```

```
select * from clientes where vendedor = 'PEPE'

END
```

En este ejemplo no estamos usando parámetros porque no hacen falta. Nos pidieron reproducir la vista de PEPE y eso hemos hecho.

Esto cumplirá la misma función que teníamos antes con la vista para PEPE. ¿Podemos ser más ambiciosos?

Nos gustaría que pudiéramos usar nuestro procedimiento almacenado para PEPE y para cualquier otro vendedor. ¿Por qué no usar un parámetro para pasar el nombre del vendedor cuyos clientes queremos ver?

En ese caso nuestro procedimiento almacenado quedaría:

```
CREATE PROCEDURE [dbo].[ClientesPepe]

@Vendedor varchar(50)

AS

BEGIN

select * from clientes where vendedor = @Vendedor

END
```

Notamos que aparece el parámetro @Vendedor al cual declaramos del tipo varchar(50).

En vez de igualar en nombre del vendedor guardado en la tabla de clientes a la constante 'PEPE' lo que hacemos es igualarlo al contenido del parámetro @Vendedor.

En este punto se sugiere realizar los ejercicios prácticos 3.9 y 3.10

¿Y cómo usamos un procedimiento almacenado?

Para decirle al motor de SQL que debe ejecutar un procedimiento almacenado la sentencia será:

EXEC [dueño].[nombre del procedimiento] parámetro1,...,parámetro n

En este punto se sugiere realizar el ejercicio práctico 3.11

Para eliminar un procedimiento almacenado la sentencia es:

DROP [dueño].[nombre del procedimiento]

Ejemplo:

Para eliminar el procedimiento almacenado ClientesPepe:

DROP PROCEDURE dbo.ClientesPepe

En este punto se sugiere realizar el ejercicio práctico 3.12

5. Funciones

Ya usamos subrepticiamente una función escalar cuando calculamos el año a partir de una fecha. Una función escalar es una máquina capaz de transformar un dato de acuerdo a una regla para producir un nuevo dato.

Por ejemplo transformamos una fecha para quedarnos solamente con el año.

Podemos usar las funciones escalares con libertad en cualquier lugar en el que SQL espera evaluar un resultado.

La forma más simple de probarlas es ir a la ventana de consultas (queries) del motor de base de datos y escribir, por ejemplo:

```
SELECT YEAR('2018-02-20')
```

y la respuesta que recibimos es:

```
2018
```

A medida que se van generando nuevas versiones de SQL se van agregando más y más funciones. Su cantidad es tal que nos conviene clasificarlas por tipo para ir recorriendo las más usadas.

Los tipos que vamos a cubrir son:

- Funciones matemáticas
- Funciones de manipulación de strings
- Funciones de manipulación de fechas
- Funciones de conversión
- Funciones no cubiertas por las clasificaciones anteriores
- Funciones definidas por el usuario

Funciones matemáticas:

ABS(x):

Devuelve el valor absoluto de “x”. Por ejemplo: ABS(1) es 1 y ABS(-1) es 1.

ROUND(x, n):

Redondea el número “x” con el n decimales indicado en “n”, si no se indica “n” asume cero decimales. Por ejemplo ROUND(1.51,1) es 1.5 y ROUND(1.51) es 1.

SQRT(x):

Da la raíz cuadrada de x. Debe tratarse de un número positivo. Por ejemplo SQRT(4) da 2.

POWER(x, n): Devuelve la potencia de “x” elevada el exponente “n”. Por ejemplo POWER(3,2) es 9

SIN(x): Devuelve el seno de "x". Conviene tener en cuenta que considera a "x" como expresada en radianes. Esto es, 1.57 corresponde aproximadamente a 90 grados, 3.14 corresponde aproximadamente a 180 grados.

COS(x): Devuelve el coseno de "x". Del mismo modo que el seno lo considera expresado en radianes.

TAN(x): Devuelve la tangente de "x". Del mismo modo que con el seno y el coseno lo considera expresado en radianes.

LOG(x): Devuelve el logaritmo de "x" en base e. Recordar que e es aproximadamente 2.7182856, x debe ser mayor que cero.

EXP(x): Devuelve e elevado a la "x". Se trata del mismo valor de e que mencionamos para el logaritmo.

En este momento es conveniente realizar el ejercicio práctico 3.13

Funciones de manipulación de strings:

LOWER(x): Devuelve la cadena “x” con cada letra convertida a minúscula.

UPPER(x): Es lo opuesto a LOWER. Devuelve la cadena “x” con todas las letras convertidas a mayúsculas.

LTRIM(x): Elimina los espacios desde la izquierda (al principio) de la cadena “x”.

RTRIM(x): Elimina los espacios desde la derecha (al final) de la cadena “x”.

REPLACE(x, v, n): Sirve para reemplazar dentro de una cadena "x" un trozo "v" (lo viejo) por otro trozo "n" (lo nuevo).

REPLICATE(x, n): Sirve para crear una cadena repitiendo varias veces otra. Devuelve el valor de la cadena “x” el número de veces “n” que hayamos indicado.

LEFT(x, n): Sirve para quedarnos con el principio de una cadena. Devuelve los “n” primeros caracteres (desde la izquierda) de la cadena “x”.

RIGHT(x, n): Devuelve los primeros “n” caracteres desde la derecha de la cadena “x”.

SUBSTRING(x, m, n): Devuelve una parte de la cadena de caracteres "x" tal que empieza en el caracter "m" y tiene una longitud de "n".

En este momento se recomienda realizar el ejercicio práctico 3.14

Funciones de manejo de fechas:

YEAR(f): Devuelve el año correspondiente de la fecha “f”. (Nuestra vieja conocida)

MONTH(f): Devuelve el mes de la fecha “f”.

DAY(f): Devuelve el día del mes de la fecha “f”.

DATEADD(f, n, d): Le suma a la fecha "f" n períodos del tipo "d". Por ejemplo DATEADD('2018-02-03',7,d) devuelve '2018-02-10'

Si n es negativo en vez de sumar, resta.

GETDATE(): Devuelve la fecha y hora actual que tiene configurado el sistema operativo.

En este momento se recomienda realizar el ejercicio práctico 3.15

Funciones de conversión

Esta funciones suelen ser propias de cada fabricante de motor de base de datos. Sirven para pasar un dato que está en un tipo a otro. Por ejemplo convertir un número en una cadena de caracteres o convertir una cadena de caracteres que contiene una fecha en un dato del tipo fecha.

Esta tabla nos da una idea de cuáles son esas funciones:

	SQL Server	Oracle	MySQL
	CAST, CONVERT, PARSE	TO_CHAR, TO_DATE, TO_NUMBER	CAST

Funciones definidas por el usuario:

Además de las funciones que ya trae el motor de base de datos tenemos la posibilidad de construir funciones propias.

Esto nos permite extender lo que el motor de base de datos sabe hacer y reutilizar nuestro código en diferentes consultas.

Vamos a necesitar como siempre:

- Crear funciones
- Modificar funciones
- Eliminar funciones

Creación de funciones:

```
CREATE FUNCTION [dueño de la función].[nombre de la función]
(
    -- Add the parameters for the function here
    [@nombre del parámetro 1 tipo de dato del parámetro 1],
    ...
    [@nombre del parámetro n tipo de dato del parámetro n]

)
RETURNS [tipo de dato que la función devuelve]
AS
BEGIN
    [código de la función]
```

RETURN [@valor que devuelve la función]

END

Veamos que es cada cosa:

- Dueño de la función: es el que podrá, por defecto, modificarla, eliminarla y usarla.
- Nombre de la función: es el nombre único por el que identificaremos e invocaremos la función que estamos definiendo.
- Nombre del parámetro: Podemos pasarle información a la función para que la procese. La forma de pasarle esa información es a través de parámetros. Cada parámetro opera como una variable dentro de la función. Como todo campo o variable cada parámetro tiene un tipo de datos, esto es, puede ser un número, una cadena de caracteres, una fecha, etc.
- Tipo de dato que devuelve la función: así como los parámetros tienen un tipo de dato asociado para introducir información en la función que estamos definiendo la salida que nos devuelva la función deberá ser de algún tipo de dato concreto y necesitamos declararlo.

Dentro del código de la función tenemos que incluir todas las sentencias necesarias para que la función haga su trabajo.

Repasemos algunas cosas que nos vamos a encontrar a menudo:

- Declarar variables para usar dentro de la función
- Hacer cálculos con las variables
- Almacenar el resultado de una consulta en una variable
- Ejecutar código si se cumple una condición
- Ejecutar código mientras se cumpla una condición
- Devolver un resultado

Declarar una variable

Para declarar una variable, en una efusión infinita de imaginación, usamos el comando DECLARE:

DECLARE @[nombre de variable] [tipo de variable]

Donde:

- nombre de variable es el identificador único que vamos a usar para la variable. Lo vamos a usar siempre precedido por un carácter @ (arroba).
- tipo de variable corresponde al tipo del dato que queremos guardar dentro de la variable que estamos declarando.

Una vez que la variable está declarada podemos usarla, pero, a menos que guardemos algo dentro de ella no deberíamos hacerlo.

Ejemplo:

Declare @a int

Hemos declarado una variable que llamaremos con @a y que guardará números enteros

Calculando con variables:

Si tenemos definidas dos variables enteras donde @a tiene guardado un 1 y @b tiene guardado un 2 la forma de hacer una cuenta con ellas es:

SELECT @a+@b

y nos devolverá 3

Si, por uno de esos caprichos de la naturaleza queremos almacenar en una variable @c el resultado de una cuenta entonces usamos:

SET @c = @a + @b

Por supuesto que @a, @b y @c tendrán que estar definidas primero y @a y @b tendrán que tener un valor cargado.

¿Qué hay dentro de una variable si nunca le cargamos un resultado?

Hagamos la prueba:

Primer paso:

Declaro la variable:

DECLARE @a int

Segundo paso:

Consulto lo que está adentro:

SELECT @a

Y me devuelve

NULL

¿Qué es eso?

NULL es la representación de un dato faltante. No es un cero, no es un espacio en blanco, no es una cadena vacía.

Ejecución condicional:

A veces necesitamos que si una condición se cumple nuestra función haga algún cálculo y, si no se cumple haga otro.

Un ejemplo trivial sería si estamos calculando la cantidad de días de Febrero:

Asumamos que a nuestra función le pasamos @Y como si fuera el año:

```
/* En principio sólo tiene 29 días los años bisiestos, o sea los divisibles por cuatro */
/* PERO: los múltiplos de 100 no son bisiestos */

IF FLOOR(@Y/4)*4 = @Y
BEGIN
    IF FLOOR(@Y/100)*100 = @Y
    BEGIN
        SET @D = 28
    END
    ELSE
    BEGIN
    SET @D = 29
```

```
END

END

ELSE

BEGIN

SET @D = 28

END
```

Explicación:

Me fijo si el año @Y es divisible por 4. Eso ocurre si y sólo si al tomar la parte entera (la función FLOOR) del año dividido por 4 y, luego de tomar la parte entera, volver a multiplicar por 4 obtenemos el mismo año.

Para mayor claridad pongamos dos ejemplos:

Supongamos que @Y vale 7.

Cuando hago 7/4 obtengo 1.75

Cuando tomo la parte entera de 1.75 obtengo 1

Cuando multiplico por 4 obtengo 4

Cuando comparo a 4 con 7 no me da igual!!!

Entonces 7 no es divisible por 4.

Ahora hago el intento con @Y=8

Cuando hago 8/4 obtengo 2

Cuando tomo la parte entera de 2 obtengo 2.

Cuando hago 2*4 obtengo 8.

Y 8 es, por supuesto, igual a 8

Luego 8 si es divisible por 4.

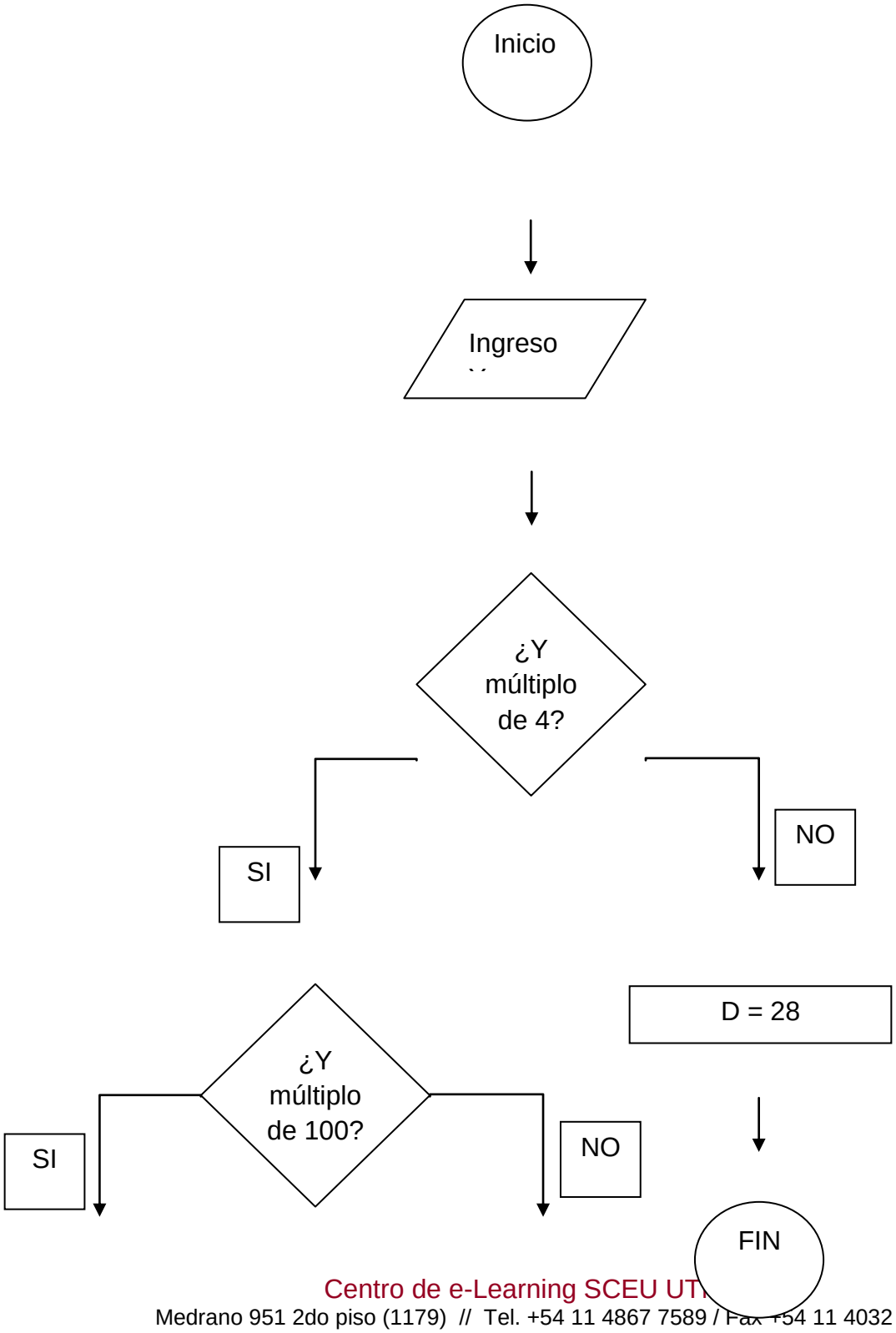
Una vez que establecí que @Y era divisible por 4 me pregunto usando el mismo truco si es divisible por 100. (Notar que al usarlo por segunda vez lo hemos convertido en un método. Un método es un truco que se usa varias veces!)

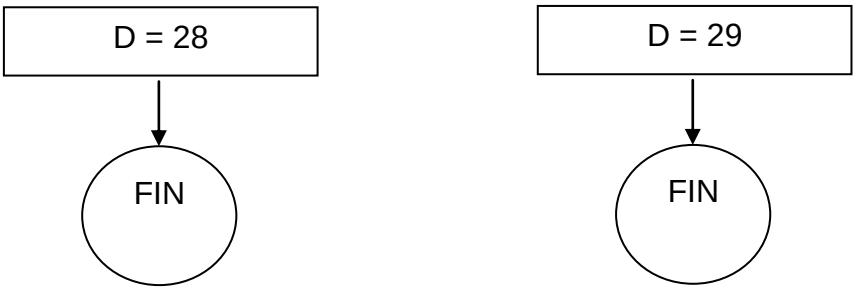
Si la respuesta a la pregunta sobre si @Y es múltiplo de 100 es afirmativa entonces el año no es bisiesto y, debemos entonces asumir que Febrero tiene 28 días. Por eso guardamos 28 dentro de @D.

Si la respuesta fuera negativa a la segunda pregunta (pero afirmativa a la primera) entonces estamos en presencia de un año bisiesto. Por eso le asignamos 29 días a Febrero.

Si, por otro lado, la respuesta a la primer pregunta hubiera sido negativa porque el número no era múltiplo de 4 entonces directamente hubiéramos pasado a asignar 28 a la variable @D

Como esto puede resultar confuso para los que no tengan experiencia previa en programación vamos a tratar de dibujarlo:


Inicio
Ingreso
¿Y múltiplo de 4?
SI
NO
¿Y múltiplo de 100?
SI
NO
D = 28
FIN



En este momento se recomienda realizar el ejercicio conceptual 3.3

En este punto se recomienda intentar el ejercicio práctico 3.16

Nota Conceptual:

De acuerdo a la Wikipedia: "El diagrama de flujo o flujograma o diagrama de actividades es la representación gráfica del algoritmo o proceso. Se utiliza en disciplinas como programación, economía, procesos industriales y psicología cognitiva."

En este punto se recomienda intentar el ejercicio práctico 3.17

Ejecución de ciclos:

Otro elemento frecuente en la programación dentro de las funciones pasa por el desarrollo de tareas repetitivas.

Tenemos instrucciones específicas para controlar una ejecución que se repite dentro de un ciclo.

WHILE [condición]

BEGIN

[conjunto de sentencias que se repiten]

END

Empecemos por armar un ejemplo sencillo.

Vamos a listar todos los números de 1 a 10.

Necesitaremos una variable donde almacenar un contador que llevará el control sobre si hemos terminado:

```
DECLARE @N INT

SET @N = 1


WHILE @N<=10

BEGIN

SELECT @N

SET @N = @N+1

END
```

En este punto se recomienda intentar los ejercicios prácticos 3.18 y 3.19

Funciones Agregadas

Las funciones agregadas responden a un problema que, los que están acostumbrados a Excel resuelven con las famosas tablas dinámicas.

Supongamos que tenemos una lista de personas con su nacionalidad y queremos saber cuántas personas vienen de cada país. Este es el caso típico de la función agregada COUNT

Si, para fijar ideas, la tabla Personas tiene dos campos, uno Nombre y el otro Nacionalidad entonces la sintaxis sería:

```
SELECT NACIONALIDAD,COUNT(*) FROM PERSONAS GROUP BY NACIONALIDAD
```

Y el resultado que obtendríamos sería:

Argentino 10

Peruano 8

Boliviano 7

Español 5

...

Nota Conceptual:

De acuerdo a la Wikipedia: "Una función agregada es la que agrupa para obtener un único valor más significativo las mediciones sobre un conjunto"

Ejemplos de funciones agregadas:

AVG: Promedio

COUNT: Cuenta

MAX: Nos devuelve el máximo

MIN: Nos devuelve el mínimo

SUM: Nos devuelve la suma

En este punto se recomienda realizar el ejercicio práctico 3.20

6. Triggers

Ya hemos visto que los cambios en las tablas los implementamos a través de las sentencias INSERT, DELETE y UPDATE.

A veces es conveniente que cuando se hace algún tipo de cambio en la información se actualicen otros datos a los efectos de mantener relaciones de integridad.

Supongamos que tenemos una tabla donde guardamos los socios activos y otra donde guardamos a sus familiares con las correspondientes fechas de cumpleaños de manera que podamos recordarle a quién tiene que saludar.

Cuando un socio se borra nos interesa que sus familiares también desaparezcan.

Una alternativa es dejar todo en manos del usuario final y pegar un recordatorio en su monitor para que se acuerde de borrar los familiares cada vez que borra un socio. El problema de hacer esto que pronto tendrá el monitor lleno de recordatorios y, a despecho de esas notitas, terminará olvidándose y tendremos en la tabla familiares que no dependen de ningún socio.

Otra alternativa (mucho mejor por cierto) es encargarse de que el programa que borra los socios se acuerde de borrar los familiares. Sin embargo esta opción también tiene debilidades. Si el programa anda mal o alguien hace un código aparte para correr desde la web o desde el celular se puede olvidar fácilmente de estas reglas.

Una alternativa es decirle al motor de base de datos que se encargue por sí mismo de realizar el borrado de los familiares cada vez que se borra un socio. El objeto que se encarga de este tipo de cosas es un Trigger o gatillo.

Vamos a tener triggers que se ejecutarán frente a un borrado y triggers que responderán a una inserción y otros que se invocarán frente a un update.

Como de costumbre vamos a ver:

- Creación de un trigger
- Borrado de un trigger
- Uso de un trigger

Creación de un trigger

La sentencia que nos permite crear un trigger tiene la siguiente sintaxis:

```
CREATE TRIGGER [nombre del trigger]
ON [nombre de la tabla]
AFTER [insert, delete o update]
AS
BEGIN
    [sentencias que se ejecutarán]
END
```

Durante la ejecución del trigger tendremos disponibles dentro de una tabla que se llamará inserted o deleted el conjunto de registros que han insertados o borrados.

En el caso de un update tendremos disponibles los registros viejos en la tabla deleted y los registros nuevos en la tabla inserted.

Ejemplo:

Volvamos al caso que usamos para introducir el tema. Tenemos dos tablas, la de socios y la de familiares:

Socios	
IdSocio	Entero
Nombre	Varchar

Familiares	
IdFamiliar	Entero
Nombre	Varchar
IdSocio	Entero
FechaCumpleaños	Fecha

Nos proponemos que cada vez que se borra uno o varios socios se borren sus familiares

La sentencia para la creación del trigger será:

```
CREATE TRIGGER BorraFamiliares
ON Socios
AFTER delete
AS
BEGIN
```

DELETE FROM Familiares WHERE IdSocio in (SELECT DISTINCT IdSocio FROM DELETED)

END

¿Qué vemos?

BorraFamiliares es el nombre del trigger

ON Socios nos indica que el trigger se disparará por las acciones sobre los socios.

AFTER delete significa que el trigger se disparará frente a borrados en la tabla Socios

Entre el BEGIN y el END tenemos las sentencias que se ejecutarán. En este caso se trata de una única sentencia de borrado que actua sobre la tabla Familiares y borra todos los registros cuyos IdSocio estén dentro del grupo de registros borrados que se encuentran (sólo para el trigger) en una "tabla" que invocamos con el nombre de DELETED.

En este punto se recomienda realizar el ejercicio práctico 3.21

Eliminación de un trigger:

Como con los otros objetos se usa la sentencia DROP:

DROP TRIGGER [nombre del trigger]

Uso del trigger:

Lo bueno es que no necesitamos usarlo, se usa solo. El problema es que si por algún motivo un trigger es borrado de la base de datos deja de hacer su tarea sin que recibamos mensaje de error alguno.

7. Estructuras de control en SQL

Con motivo de la creación de funciones escalares hemos visto ya las principales estructuras de control:

IF

WHILE

(generar vínculo a la sección correspondiente)

8. Cursores

A veces se hace necesario recorrer una tabla o una parte de la misma realizando operaciones registro por registro.

El objeto que nos ayuda a realizar esta tarea es un cursor.

En general su uso se encuentra des-recomendado y se considera más eficiente tratar de resolver las cosas sin cursores.

Vamos a aprender a:

Crear un cursor

Abrir el cursor

Recorrer el cursor

Cerrar el cursor

Liberar la memoria que ocupó

Creación de un cursor:

DECLARE [nombre del cursor] CURSOR FOR [select de los registros que el cursor va a recorrer]

Abrir el cursor:

Una vez creado el cursor debemos abrirlo. Esto nos posiciona ANTES del primer registro:

OPEN [nombre del cursor]

Recorrer el cursor:

Una vez abierto el cursor necesitamos pasar a variables de memoria los datos del registro que estamos parados. La forma de avanzar de registro y traerme los datos del nuevo es:

FETCH NEXT FROM [nombre del cursor] INTO [lista de variables que coinciden con los campos del select con que se definió el cursor]

Normalmente esa recorrida se hace dentro de un ciclo que pregunta si llegamos al final del cursor. Para eso utilizamos a nuestra buena amiga, la sentencia WHILE:

WHILE (@@FETCH_STATUS = 0)

BEGIN

[Sentencias que queremos que se repitan]

END

La variable del sistema @@FETCH_STATUS se mantendrá en cero mientras no se nos acabe el cursor. De esa manera el WHILE nos permitirá recorrerlo.

Normalmente vamos a precisar usar el FETCH NEXT FROM una vez afuera del ciclo para parar al cursor en el primer registro y luego también dentro del ciclo.

Cerrar el cursor

Para cerrar el cursor nos basta con hacer:

CLOSE [Nombre del cursor]

Para liberar la memoria:

DEALLOCATE [Nombre del cursor]

Ejemplo:

Quiero numerar todos los registros de una tabla en forma correlativa:

Los registros están dentro de la tabla llamada baseNueva.

Dentro de la variable @MaxFactId tengo guardado el último número a partir del cual quiero seguir contando.

Ya tengo declarada también la variable @k del mismo tipo de entero que @MaxFactId

```
declare c cursor for select ani from baseNueva

open c

set @k = @MaxFactId+1

fetch next from c into @ani

while @@FETCH_STATUS = 0
begin
	update baseNueva set factid = @k where ani = @ani
	set @k = @k+1
	fetch next from c into @ani
end

close c
deallocate c
```

En este punto se recomienda realizar el ejercicio práctico 3.22

Bibliografía utilizada y sugerida

Date, C. J. An Introduction to Database Systems, Pearson Educación, Mexico, 2001

Elmasri/Navathe. Sistemas de Base de Datos. Conceptos Fundamentales Pearson-Addison - Wesley 2007

Korth, Henry F. Fundamentos de Bases de Datos. Mac Graw Hill 1992.

Lo que vimos:

En esta unidad vimos cómo crear, editar, usar y destruir los principales objetos que guardamos en una base de datos.

Lo que viene:

En la próxima unidad nos sumergiremos en la estructura interna de las bases de datos, para entender el impacto que tienen distintas estrategias, en la velocidad con la que se resuelven las peticiones que enviamos al motor relacional.