Victor Hugo Tapia Jacinto

BASE DE DATOS II

Experiencias Prácticas

ULADECH CATÓLICA
UNIVERSIDAD CATÓLICA LOS ÁNGELES CHIMBOTE

Utex

Víctor Hugo Tapia Jacinto

Estudió primaria y secundaria en colegios nacionales de la ciudad de Chimbote, así como la carrera técnica profesional de Computación e Informática en el IESTP Carlos Salazar Romero.

Es Ingeniero de Sistemas de profesión y Maestro en Ingeniería de Sistemas con mención en Tecnologías de Información y Comunicación.

Docente Universitario con más de 15 años de experiencia y asesor de diferentes empresas tecnológicas en el medio, solicitado para consultorías en gestión y administración de datos y TIC, habiendo laborado en empresas estatales y privadas. Actualmente participa en un proyecto del Instituto de Investigación de la ULADECH Católica.

Victor Hugo Tapia Jacinto

BASE DE DATOS II

Experiencias Prácticas

UNIVERSIDAD CATÓLICA LOS ÁNGELES
CHIMBOTE

BASE DE DATOS II

Victor Hugo Tapia Jacinto

Diseño y diagramación:
Ediciones Carolina (Trujillo).

Editado por:
Universidad Católica Los Ángeles de Chimbote
Jr. Tumbes 247 Casco Urbano Chimbote – Perú
RUC: 20319956043
Telf: (043)343444

Primera edición digital, febrero 2020.
ISBN: 978-612-4308-22-2

Libro digital disponible en:
http://repositorio.uladech.edu.pe/handle/123456789/

Dedico este libro al
irremplazable equipo de mi vida.
Mi esencia y motivación:
Adamaris Valeska y
Anahí Victoria

AGRADECIMIENTO

A los que viven todos los procesos de mi vida, mi familia: a Keyda, Adamaris y Anahí que siempre entienden y respetan pacientemente mi inmersión por horas entre ordenadores, servidores y trabajo, dejando otros placeres para más tarde. A ellos mis infinitas gracias por su comprensión y apoyo.

A mis padres Víctor y Luz por inculcarme desde muy pequeño la perseverancia, la planificación y las ganas de alcanzar mis metas.

A la ULADECH - Católica por brindarme la oportunidad de compartir mis conocimientos y experiencias en este mundo de gestión de datos.

A todos ustedes que invierten su tiempo en leer estas experiencias, mil gracias por estar ahí, y bienvenidos a disfrutar de las experiencias prácticas del SQL.

Contenido

CAPITULO II: USO DE CLÁUSULAS, OPERADORES Y FUNCIONES

Lista de Tablas

Lista de Ilustraciones

Presentación

Este libro contiene las experiencias adquiridas en el transcurso de brindar solución a las necesidades de gestión y manipulación de datos en algunos SGBD. También se exponen las aplicaciones de estas experiencias con la ayuda de los diferentes recursos que el SQL ofrece como son funciones del sistema y definidas por el usuario, operadores, cláusulas, controles de datos con lógica de usuario, procedimientos almacenados y triggers, que permiten que las soluciones propuestas al caso puedan gestionar y administrar los datos de una manera fluida y óptima, de forma autonómica o con mínima intervención del usuario.

Introducción

Base de datos II – Experiencias prácticas contiene un sinnúmero de atenciones que un caso práctico solicita de los usuarios operativos de la base de datos. Está dividido en 4 capítulos, cada uno con los ejemplos que sirven de guía para atender las necesidades de cualquier organización en su afán de gestionar la información que fluye de sus diferentes áreas.

En el primer capítulo, usted lector va a encontrar consideraciones básicas para modelar y utilizar herramientas complementarias y sentencias SQL para la implementación de una base de datos física, sentencias como CREATE, ALTER, DROP, y sus manejos forman parte de este capítulo.

En el segundo capítulo nos adentramos más en el uso y atención a través del lenguaje SQL, el apoyo de cláusulas, operadores y funciones que el sistema gestor ofrece para controlar y manipular diferentes tipos de datos, funciones que nos devuelven valores de texto, numéricos, fecha o de conversión de los mismos. Asimismo encontraremos ejemplos para el uso correcto de los diferentes operadores matemáticos y de comparación con las cláusulas en las sentencias SQL para insertar, actualizar, eliminar y mostrar datos.

En el tercer capítulo ya estamos preparados para atender las necesidades del caso con sentencias complejas, permitiendo consultar y gestionar datos en diferentes tablas a la vez con la ayuda de las sentencias complejas JOIN, INNER JOIN Y LEFT; las cuales nos

permiten validar los datos a nivel de Primary Key y Foreign Key. Asimismo podremos ver ejemplos para la importación y exportación de datos en diferentes formatos.

En el cuarto capítulo podremos controlar los datos a través de la lógica del usuario con la gestión de datos a través de los procedimientos almacenados, consistencia de datos con los Triggers (desencadenadores) y las funciones definidas por los usuarios, que nos permiten obtener datos de la misma base de datos con lógica que el usuario gestiona y es complemento de los procedimientos que controlan la parte operativa.

Capítulo I

INTRODUCCIÓN A UN SGBD Y AL SQL

I.1. Caso práctico

La empresa **miKioscoVirtual** es una organización que se dedica al rubro de las ventas de ejemplares (físicos y virtuales) a sus clientes vía web.

Estos clientes previamente registrados en su portal tienen acceso a los ejemplares que la empresa oferta catalogados por títulos del ejemplar, tipos de ejemplar (libros, periódicos, diccionarios, etc) y categorías de ejemplar (generalidades, acción, ficción, novelas, etc), los que pueden variar según la demanda diaria, semanal o anual; de la emisión de ejemplares, los autores según nacionalidad y editoriales y ordenados por año de publicación y edición, quedando desfasados –en el caso de ejemplares con el mismo título– por el de menor antigüedad.

Los clientes realizan sus pedidos vía el portal web, accediendo al catálogo a través de una ventana de seguridad que le solicita un *login* (validado a través del ID del cliente) y un *password* (que tendrá que ser validado por el administrador de la base de datos). Estos pedidos pueden ser de manera corporativa (empresa) o individual y no tienen límite de cantidad solicitada en el caso de que sea un pedido de un ejemplar virtual y según la disponibilidad del stock para el caso de que el ejemplar sea físico.

Los pedidos de los ejemplares serán entregados en el mismo día; en el caso de los ejemplares físicos dependerá de la ubicación física del cliente, variando siempre el estado del pedido que puede ser atendido, en proceso, enviado, cancelado, recepcionado.

En el caso de los ejemplares tipo periódicos, estos al finalizar el día automáticamente son considerados con stock 0, para iniciar el día con nuevo stock.

La empresa también requiere que al finalizar el día se emitan reportes con los movimientos realizados por los clientes (pedidos) respecto a los ejemplares, condiciones de sus ejemplares, así como los usuarios que han visitado o accedidos al sitio web a consultar, solicitar o hacer el seguimiento de algún ejemplar.

Las ofertas en el caso de los ejemplares se manejan en referencia a la antigüedad de los mismos que pueden variar según la política de la organización (por días, semanas, meses o años), considerando que cada ejemplar se actualiza según el tipo o a solicitud del autor.

Los clientes pueden darse de baja con respecto a su suscripción, pero no se puede eliminar su registro ya que ellos o algunos en algún momento pueden estar involucrados en alguna operación de pedido o de devolución de los pedidos.

Se requiere el uso de un lenguaje informático que nos permita estructurar, manipular, restringir, controlar y gestionar los datos de esta organización para la atención de sus necesidades y la interacción futura con aplicaciones en las diferentes plataformas para el uso de sus clientes.

I.2. Lenguaje SQL – DDL

DEFINICIÓN

Un Data Definition Language o Lenguaje de descripción de datos (DDL) es un lenguaje de programación para definir estructuras de datos. El término DDL fue introducido por primera vez en relación con el modelo de base de datos CODASYL (es el acrónimo para «Conference on Data Systems Languages»), donde el esquema de la base de datos ha sido escrito en un lenguaje de descripción de datos que describe los registros, los campos, y «conjuntos» que conforman el modelo de datos. Después fue usado para referirse a un subconjunto de SQL, pero ahora se utiliza en un sentido genérico para referirse a cualquier lenguaje formal para describir datos o estructuras de información, como los esquemas XML.[1]

1 SGBD. Sistemas de gestión de bases de datos (en inglés database management system, abreviado DBMS) o SGBD son un tipo de software muy específico, dedicado a servir de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan. (7)

SQL

A diferencia de muchos lenguajes de descripción de datos, SQL utiliza una acción de versos imperativo cuyo efecto es modificar el esquema de la base de datos para cualquier SGBD, añadiendo, cambiando o eliminando las declaraciones que se pueden mezclar libremente con otras sentencias SQL, por lo que el DDL no es realmente una lengua independiente. La declaración más común es CREATE TABLE. El lenguaje de consulta SQL, el más difundido entre los gestores de bases de datos, admite las siguientes sentencias de definición: CREATE, DROP y ALTER, cada una de las cuales se puede aplicar a las tablas, vistas, procedimientos almacenados y triggers de la base de datos.

SENTENCIA CREATE

Create Database

La sentencia CREATE DATABASE se utiliza para crear bases de datos.

Sintaxis CREATE DATABASE:

```
CREATE DATABASE nombreBaseDatos
```

Ejemplo CREATE DATABASE

```
CREATE DATABASE miKioscoVirtual
```

Create Table

La sentencia CREATE TABLE se utiliza para crear una tabla en una base de datos existente.

Sintaxis CREATE TABLE

```
CREATE TABLE nombretabla
(nombrecolumna1 tipodato1, nombrecolumna2
tipodato2, nombrecolumna3 tipodato3,... )
```

Ejemplo CREATE TABLE

```
CREATE TABLE Autores
(AUT_Id              int NOT NULL ,
AUT_Nombres          varchar(50) NULL ,
AUT_Nacionalidad     char(3) NULL)
```

Esta sentencia creará la tabla 'Autores' con 3 columnas.

La columna 'Aut_Nombres' es de tipo 'varchar'; es decir, acepta valores alfanuméricos hasta una longitud máxima de 50 caracteres.

La columna 'Aut_Id' es de tipo 'int'; es decir, acepta sólo números.

La columna 'Aut_Nacionalidad' es de tipo 'char'; es decir, acepta valores de cadena hasta una longitud máxima de 3 caracteres.

Existen diferentes tipos de datos, algunos son iguales en todas las bases de datos (MySQL, ORACLE, DB2, etc.) y otros pueden ser particulares para ser usados únicamente en alguna de estas bases de datos.

Create Index

Los índices se utilizan para recuperar datos de la base de datos muy rápidamente. Los usuarios no pueden ver los índices, solo se utilizan para acelerar las búsquedas / consultas.

Sintaxis

```
CREATE INDEX nombreindex ON nombretabla
(columna1, columna2, ...);
```

Ejemplo CREATE INDEX

```
CREATE INDEX idx_Nombres
ON Autores (AUT_Nombres);
```

Nota: La actualización de una tabla con índices lleva más tiempo que la actualización de una tabla (porque los índices también necesitan una actualización). Por lo tanto, solo cree índices en columnas en las que se buscará con frecuencia.

SENTENCIA DROP

La sentencia DROP se utiliza para borrar definitivamente un índice, tabla o base de datos.

DROP INDEX

Sintaxis DROP INDEX para MySQL

```
ALTER TABLE nombretabla
DROP INDEX nombreindice
```

Sintaxis DROP INDEX para DB2 y ORACLE

```
DROP INDEX nombreindice
```

Sintaxis DROP INDEX para ACCESS

```
DROP INDEX nombreindice
ON nombretabla
```

Sintaxis DROP INDEX para SQLSERVER

```
DROP INDEX nombretabla.nombreindice
```

DROP TABLE

Se utiliza DROP TABLE para borrar definitivamente una tabla

```
DROP TABLE nombretabla
```

DROP DATABASE

Se utiliza para borrar una base de datos definitivamente.

```
DROP DATABASE nombrebasededatos
```

SENTENCIA ALTER

La sentencia SQL ALTER se utiliza para añadir, eliminar o modificar columnas de una tabla.

Sintaxis SQL ALTER

Para añadir una nueva columna a una tabla

```
ALTER TABLE nombretabla
ADD nombrecolumna tipodatocolumna
```

Para borrar una columna de una tabla

```
ALTER TABLE nombretabla
DROP COLUMN nombrecolumna
```

Para modificar el tipo de dato de una columna de una tabla

```
ALTER TABLE nombretabla
ALTER COLUMN nombrecolumna tipodatocolumna
```

Ejemplos de SQL ALTER

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	ALE

Tabla 1: Ejemplo de SQL Alter - Tabla Inicial.

Dada la siguiente tabla de 'personas', queremos añadir una nueva columna, denominada 'ubigeo'

```
ALTER TABLE Autores ADD ubigeo varchar(6)
```

Aut_Id	Aut_Nombres	Aut_Nacionalidad	UBIGEO
111111	Adamaris Tapia Rosales	PER	
222222	Anahí Tapia Rosales	PER	
333333	Luz Jacinto Heredia	ALE	

Tabla 2: Ejemplo de SQL Alter - Campo Añadido.

Si queremos modificar el tipo de dato de la columna 'ubigeo' y ponerle tipo 'char' en lugar de tipo 'varchar'.

```
ALTER TABLE Autores ALTER COLUMN ubigeo char(6)
```

Si queremos borrar la columna 'ubigeo' y dejarlo igual que al principio

```
ALTER TABLE Autores DROP COLUMN ubigeo;
```

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	ALE

Tabla 3: Ejemplo de SQL Alter - Campo Eliminado.

Si queremos asignar una clave primaria (PK) a la tabla para SQLSERVER, MySQL, DB2, ORACLE y ACCESS

```
ALTER TABLE Autores ADD CONSTRAINT XPKAutores
PRIMARY KEY (AUT_Id ASC)
```

Si esta tabla es referenciada por otra (clave foránea FK) para SQLSERVER, MySQL, DB2, ORACLE y ACCESS

```
ALTER TABLE Ejemplares
ADD CONSTRAINT Ejemplares_Autor FOREIGN KEY
(AUT_Id) REFERENCES Autores (AUT_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION
```

SENTENCIA BACKUP Y RESTORE

En este tema se explica cómo crear una copia de seguridad completa de la base de datos en SQL.

Limitaciones y restricciones

- La instrucción BACKUP no se permite en una transacción explícita o implícita.
- Las copias de seguridad que se crean en una versión más reciente de SGBD no se pueden restaurar en versiones anteriores del mismo SGBD.

Recomendaciones

A medida que la base de datos aumenta de tamaño, las copias de seguridad completas requieren una mayor cantidad de tiempo para finalizar y espacio de almacenamiento. En el caso de una base de datos grande, considere la posibilidad de complementar una copia de seguridad completa con una serie de copias de seguridad diferenciales.

Para calcular el tamaño de una copia de seguridad completa de la base de datos en el SQL Server se utiliza el procedimiento almacenado del sistema sp_spaceused .

Crear una copia de seguridad completa de base de datos

Ejecute la instrucción BACKUP DATABASE para crear la copia de seguridad de base de datos completa, especificando:

- El nombre de la base de datos de la que se va a realizar una copia de seguridad.
- El dispositivo de copia de seguridad en el que se escribe la copia de seguridad de base de datos completa.
- La sintaxis básica de Transact-SQL para crear una copia de seguridad de base de datos completa es:

```
BACKUP DATABASE database
TO backup_device [ ,...n ]
[ WITH with_options [ ,...o ] ] ;
```

Donde:

database: Es la base de datos cuya copia de seguridad se desea hacer.

backup_device [,...n]: Especifica una lista de 1 a 64 dispositivos de copia de seguridad que se pueden utilizar en la operación de copia de seguridad. Puede especificar un dispositivo físico de copia de seguridad o puede especificar un dispositivo de copia de seguridad lógico correspondiente, si ya se definió. Para especificar un dispositivo de copia de seguridad físico, use la opción DISK o TAPE:

```
{ DISK | TAPE } =nombre_dispositivo_copia de seguridad_física
```

WITH with_options [,...o] : De forma opcional, puede especificar una o varias opciones.

Ejemplo:

Realizar la copia de seguridad en un dispositivo de disco.

En el ejemplo siguiente se realiza una copia de seguridad completa de la base de datos miKioskoVitual en el disco y se usa FORMAT para crear un conjunto de medios nuevo.

```
USE miKioskoVirtual;
GO
BACKUP DATABASE miKioskoVirtual
TO DISK = 'Z:\ Backups\ miKioskoVirtual.Bak'
 WITH FORMAT,
 MEDIANAME = 'Z_Backups',
 NAME = 'Backup Completo de miKioskoVirtual ';
GO
```

Usar PowerShell

Use el cmdlet Backup-SqlDatabase. Para indicar explícitamente que esta es una copia de seguridad completa de la base de datos, especifique el parámetro -BackupAction con su valor predeterminado, Database. Este parámetro es opcional para las copias de seguridad de base de datos completas.

Ejemplo:

Copia de seguridad local completa

En el ejemplo siguiente se crea una copia de seguridad completa de la base de datos **miKioskoVirtual** en la ubicación de copia de seguridad predeterminada de la instancia de servidor Servidor\Instance. Opcionalmente, en este ejemplo se especifica -BackupAction Database.

```
Backup-SqlDatabase -ServerInstance Servidor\Instance -
Database miKioskoVirtual -BackupAction Database
```

En el siguiente tema explicamos la forma de restaurar una copia de seguridad

El objetivo de una restauración completa de la base de datos es restaurar toda la base de datos. Durante el proceso de restauración, la base de datos completa se encuentra **sin conexión**. Antes de que ninguna parte de la base de datos esté en línea, se recuperan todos los datos a un punto coherente en el que todas las partes de la base de datos se encuentran en el mismo momento y en el que no existe ninguna transacción sin confirmar.

En el modelo de recuperación simple, no se puede restaurar la base de datos a un momento concreto de una copia de seguridad específica.

Importante:

Se recomienda no adjuntar ni restaurar bases de datos de orígenes desconocidos o que no sean de confianza. Estas bases de datos pueden contener código malintencionado que podría ejecutar código Transact-SQL inesperado o provocar errores debido a la modificación del esquema o de la estructura de la base de datos física.

Sintaxis :

```
RESTORE DATA16BASE database_name FROM
backup_device [ WITH NORECOVERY ]
```

Ejemplo:

```
USE master;
ALTER DATABASE miKioskoVirtual SET RECOVERY
SIMPLE;
GO
BACKUP DATABASE miKioskoVirtual
TO DISK = 'Z:\ Backups\miKioskoVirtual.bak'
 WITH FORMAT;
GO
BACKUP DATABASE miKioskoVirtual
TO DISK = 'Z:\Backups\miKioskoVirtual.bak'
 WITH DIFFERENTIAL;
GO
-- Restaure la copia de seguridad completa de la base de
datos (del conjunto de copia de seguridad 1)..
RESTORE DATABASE miKioskoVirtual
FROM DISK = 'Z:\Backups\miKioskoVirtual.bak'
 WITH FILE=1, NORECOVERY;
-- Restaurar la copia de seguridad diferencial (del conjunto
de copia de seguridad 2)...
RESTORE DATABASE miKioskoVirtual
FROM DISK = 'Z:\Backups\miKioskoVirtual.bak'
 WITH FILE=2, RECOVERY;
GO
```

OCURRENCIAS ENCONTRADAS LUEGO DE RESTAURAR BASES DE DATOS

Luego de haber restaurado nuestra copia de seguridad completa o diferencial, uno de los inconvenientes más comunes es que no se pueda visualizar el diagrama de la base de datos y al momento de abrir el asistente nos muestre el siguiente mensaje:

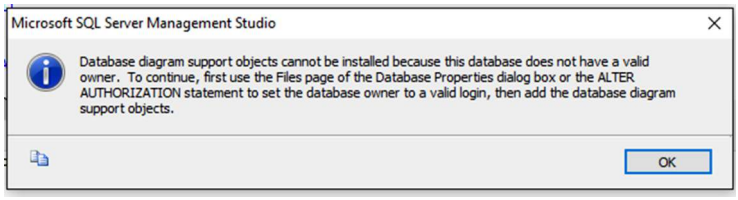


Figura 1: Mensaje que impide mostrar Diagrama de la Base de Datos.

Aquí debemos ejecutar la siguiente sentencia:

```
ALTER AUTHORIZATION
 ON [ <class_type>:: ] entity_name
 TO { principal_name | SCHEMA OWNER }
[;]
```

Ejemplo:

Autorizar a JPerez para crear, modifica o eliminar el diagrama de la Base de Datos miKioskoVirtual

```
ALTER AUTHORIZATION
 ON DATABASE::miKioskoVirtual
 TO JPerez;
```

I.3. Modelado de una Base de Datos - Modo Gráfico

Actualmente existe en el mercado una gran variedad de modeladores visuales de base de datos. Por mencionar algunos conocidos están Erwin Data Modeler, MySQL Workbench (bajo plataforma Windows), DBVisualizer, SQL Power Architect y otros ya están incluidos dentro de los gestores, los más utilizados como SQL Server y Oracle.

En este material vamos a hacer uso del modelador de datos Erwin Data Modeler en su versión Educativa (Academic Edition)[2], conside-

2 Se ha utilizado en aulas de todo el mundo para enriquecer los estudios de gestión de datos. Complementa los cursos de TI y de negocios a nivel de pregrado y posgrado al proporcionar acceso práctico a modelos de datos conceptuales, lógicos y físicos. (8)

rando que solo el modelo resultado se puede gestionar cuando se implemente en cualquier gestor de Base de datos. En nuestro caso utilizaremos un modelo de base de datos que detallaremos más adelante en el caso práctico propuesto para los ejemplos.

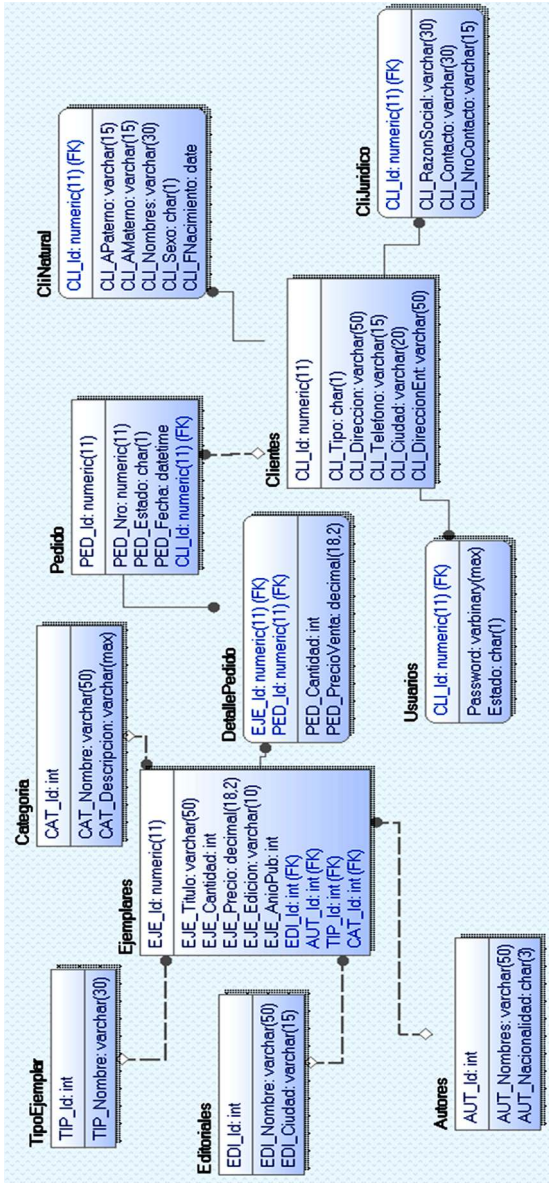


Figura 2: Modelo Relacional del Caso Práctico.

I.4. Implementación de una Base de Datos

La implementación del modelo de datos es considerada la estructura física de la base de datos dentro de un SGBD.

Este item muestra cómo utilizar la sintaxis de SQL para implementar el modelo de datos del caso práctico. Es decir, se muestra cómo crear una base de datos y las tablas.

CREACIÓN FÍSICA DE LA BASE DE DATOS

A continuación, se muestra parte del código SQL que nos permitirá crear nuestra Base de datos física partiendo del modelo de base de datos obtenido en el ejemplo anterior; considerando que se aplica el mismo procedimiento para la creación, asignación de claves primarias (PK), así como la relación (Referencias) entre las tablas mediante claves foráneas FK.

```
CREATE DATABASE miKioscoVirtual;

USE miKioscoVirtual;

CREATE TABLE Clientes
(CLI_Id numeric(11) NOT NULL , CLI_Tipo char(1) NULL,
CLI_Direccion varchar(50) NULL, CLI_Telefono varchar(15)
NULL, CLI_Ciudad varchar(20) NULL, CLI_DireccionEnt
varchar(50) NULL)

ALTER TABLE Clientes
ADD CONSTRAINT XPKClientes PRIMARY KEY (CLI_Id
ASC)

CREATE TABLE CliJuridico
(CLI_Id numeric(11) NOT NULL, CLI_RazonSocial varchar(30)
NULL, CLI_Contacto varchar(30) NULL, CLI_NroContacto
varchar(15) NULL )
```

```
ALTER TABLE CliJuridico
ADD CONSTRAINT XPKCliJuridico PRIMARY KEY ( CLI_Id
ASC)

CREATE TABLE CliNatural
(CLI_Id numeric(11) NOT NULL, CLI_APaterno varchar(15)
NULL , CLI_AMaterno varchar(15) NULL , CLI_Nombres
varchar(30) NULL , CLI_Sexo char(1) NULL, CLI_FNacimiento
date NULL )

ALTER TABLE CliNatural
ADD CONSTRAINT XPKCliNatural PRIMARY KEY ( CLI_Id
ASC)

CREATE TABLE Ejemplares
(EJE_Id numeric(11) NOT NULL, EDI_Id int NULL, EJE_Titulo
varchar(50) NULL, EJE_Cantidad int NULL, AUT_Id int NULL,
EJE_Edicion varchar(10) NULL, EJE_AnioPub int NULL,
TIP_Id int NULL, CAT_Id int NULL, EJE_Precio decimal(18,2)
NULL)

ALTER TABLE Ejemplares
ADD CONSTRAINT XPKEjemplares PRIMARY KEY (EJE_Id
ASC)

ALTER TABLE CliJuridico
ADD CONSTRAINT CliJuridico_Clientes FOREIGN KEY
(CLI_Id) REFERENCES Clientes (CLI_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION

ALTER TABLE CliNatural
ADD CONSTRAINT CliNatural_Clientes FOREIGN KEY
(CLI_Id) REFERENCES Clientes (CLI_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION
```

```
ALTER TABLE Ejemplares
ADD CONSTRAINT Ejemplares_Editoriales FOREIGN KEY
(EDI_Id) REFERENCES Editoriales (EDI_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION

ALTER TABLE Ejemplares
ADD CONSTRAINT Ejemplares_Autores FOREIGN KEY
(AUT_Id) REFERENCES Autores (AUT_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION

ALTER TABLE Ejemplares
ADD CONSTRAINT Ejemplares_ TipoEjemplar FOREIGN
KEY (TIP_Id) REFERENCES TipoEjemplar (TIP_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION

ALTER TABLE Ejemplares
ADD CONSTRAINT TipoEjemplar_Categoria FOREIGN KEY
(CAT_Id) REFERENCES Categoria (CAT_Id)
ON DELETE NO ACTION
ON UPDATE NO ACTION
```

Nota: también podemos utilizar el asistente de Erwin para generar físicamente el modelo anterior según el SGBD que utilicemos.

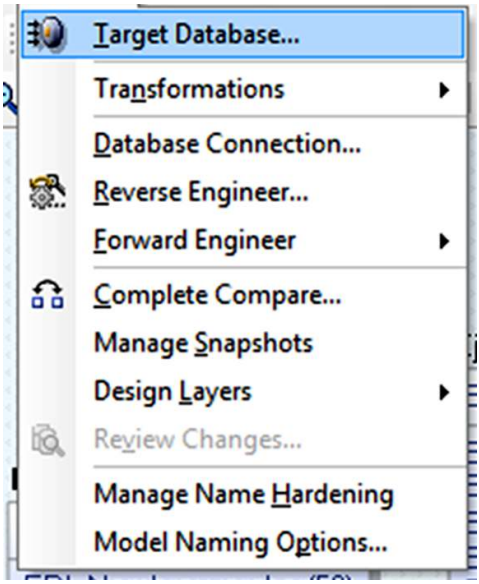


Figura 3: Acción para selección del Gestor de Base de Datos.

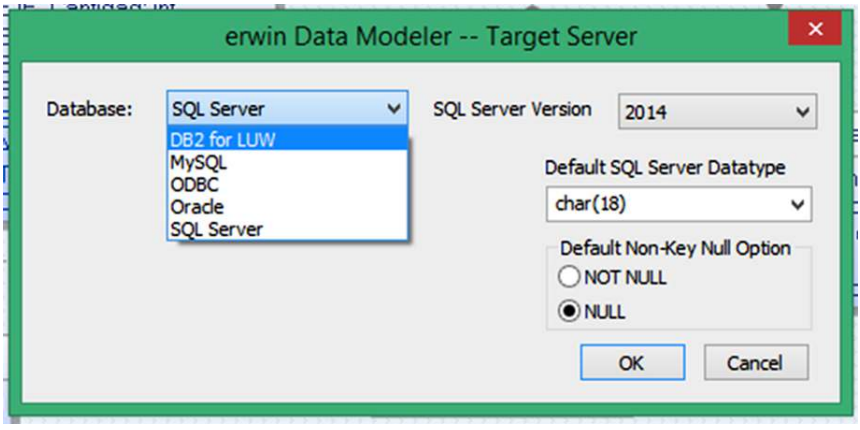


Figura 4: Selección del Gestor de Base de Datos para migrar el esquema.

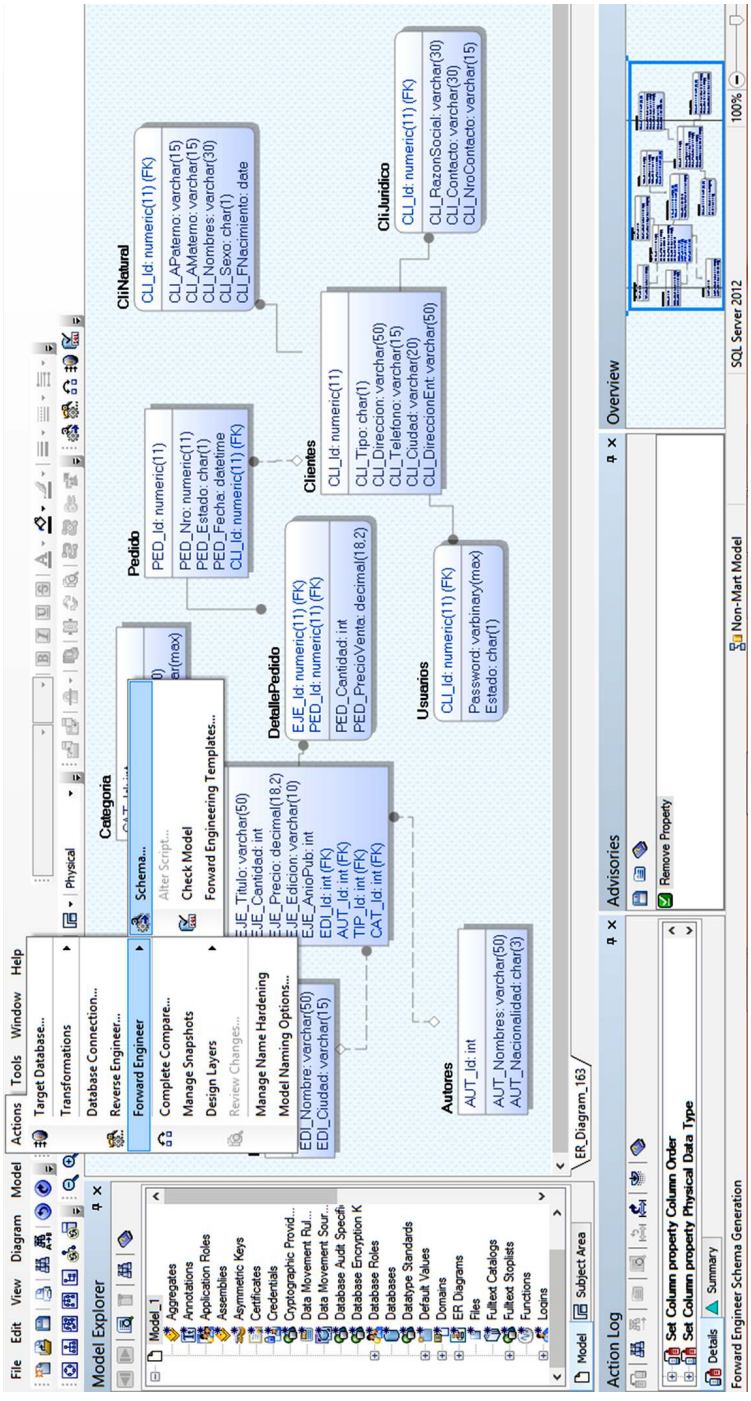


Figura 5: Selección de la acción desde Erwin Data Modeler

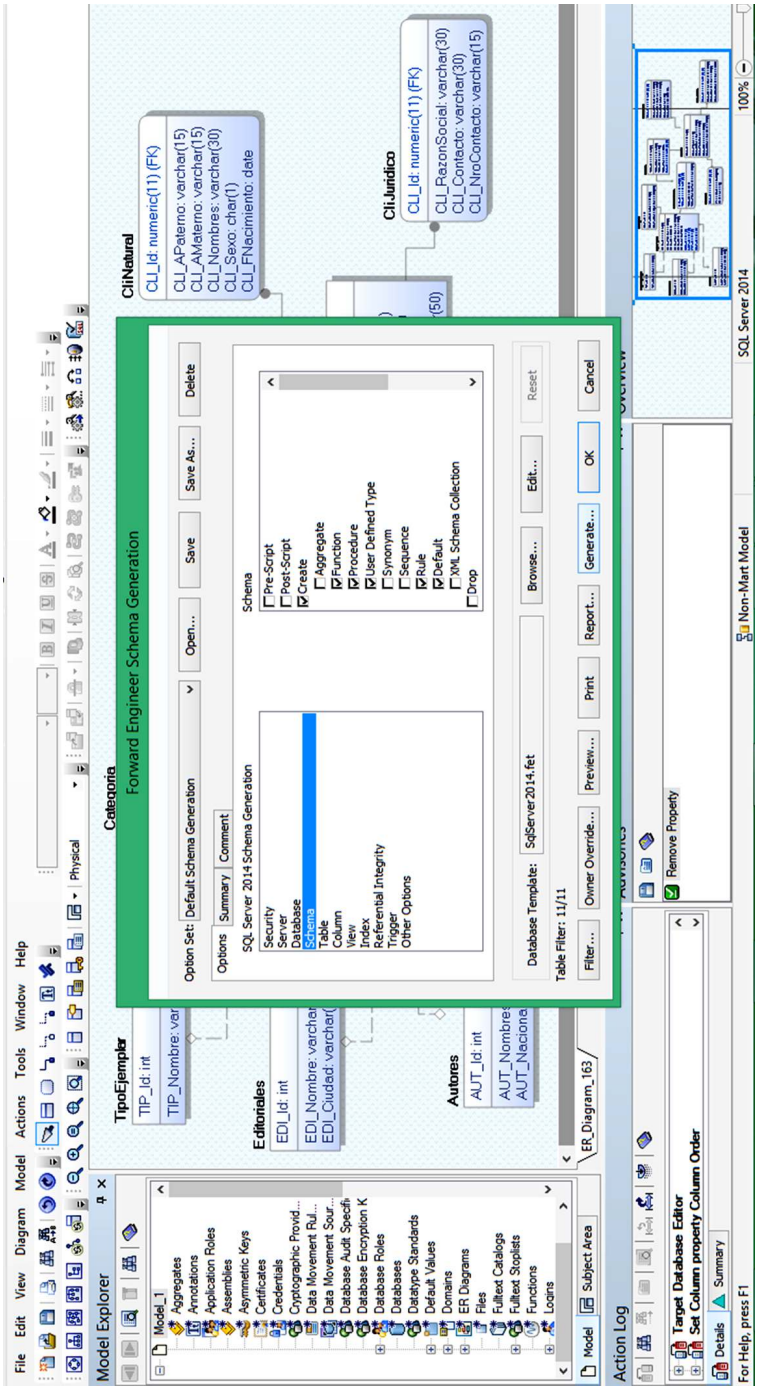


Figura 6: Migrando el Modelo Lógico desde Erwin al SGBD

DIAGRAMA DE BASE DE DATOS

Ahora, luego de haber migrado los datos desde el Erwin, podemos visualizar nuestro modelo de datos físico en el SGBD

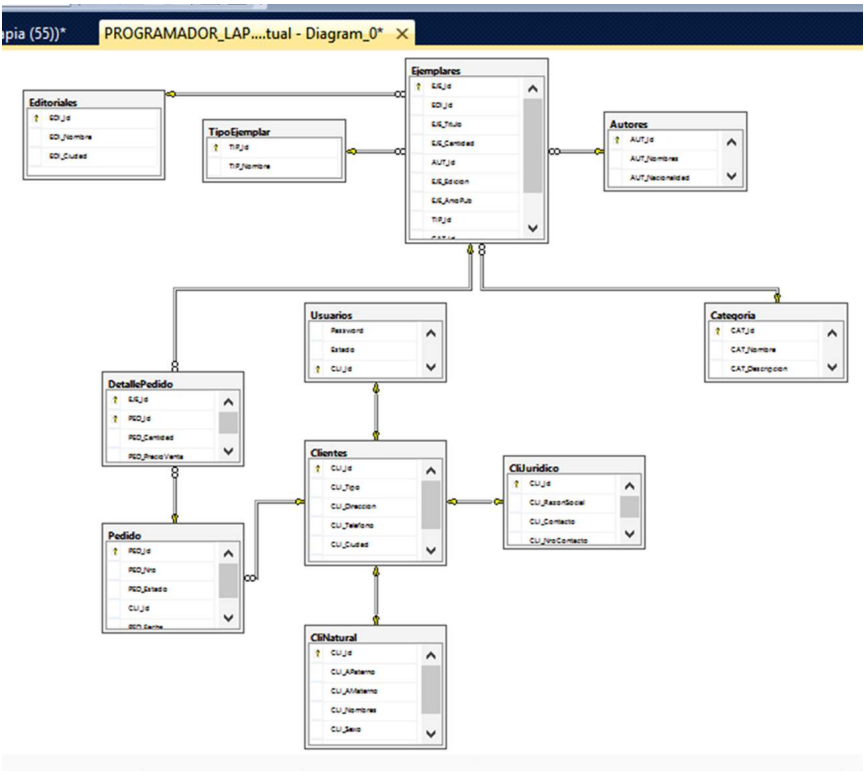


Figura 7: Modelo de Datos Físico en el SGBD SQL Server.

I.5. Lenguaje SQL – MDL

Lenguaje de Manipulación de Datos (Data Manipulation Language, DML) es un lenguaje proporcionado por los sistemas gestores de bases de datos que permite a los usuarios de la misma llevar a cabo las tareas de consulta o modificación de los datos contenidos en las Bases de Datos del Sistema Gestor de Bases de Datos.[2]

El lenguaje SQL se utiliza para acceder y manipular datos en cualquier base de datos del mercado, como, por ejemplo, para las bases de datos MySQL, Oracle, DB2, SQL Server, Access, etc.

El SQL es un lenguaje estructurado y un estándar ANSI para el acceso y manipulación de los datos de cualquier base de datos.

SECUENCIAS EN SQL

Se puede definir una secuencia como un conjunto de valores que parten de un valor inicial, tienen un incremento o decremento, lo que significa que la secuencia puede ser ascendente o descendente y pueden tener un valor final.[3]

SQL Server permite la creación de secuencias que pueden ser utilizadas para la generación de códigos en las tablas. Lo más importante de las secuencias es que no están ligadas a ningún campo en una tabla. Se recomienda usar la opción de Secuencias en lugar de usar la propiedad Identity, es necesario incidir en sugerir adicionalmente que no use la propiedad Identity.

Tipos de datos permitidos en secuencias

El tipo de dato de la secuencia es un dato Entero, los tipos de datos permitidos son:

3 ANSI. Viene de las siglas en inglés de American National Standards Institute (Instituto Nacional Estadounidense de Estándares), organización encargada de supervisar el desarrollo de normas para los servicios, productos, procesos y sistemas en los Estados Unidos. (9)

Tipo De Datos	Valores
Tinyint	Rango 0 to 255
smallint	Rango -32,768 to 32,767
int	Rango -2,147,483,648 to 2,147,483,647
bigint	Rango -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 Este es el tipo de dato por defecto.
decimal y numeric	con una escala de CERO.

Tabla 4: Tipos de Datos permitidos en secuencias.

La propiedad Identity

Identity es una propiedad que permite que un campo en una tabla incremente su valor de manera automática al insertar los registros en ella. Para el uso de la propiedad Identity el tipo de dato debe ser entero Int. Es necesario definir un valor inicial y un valor de incremento.

Es importante anotar que Identity no asegura la unicidad de valor, esta únicamente es posible con la restricciones Primary key, Unique o con el índice Unique. Solamente puede existir una columna por tabla con la propiedad Identidad.

Secuencia vs. Identity

En SQL Server se debe usar una secuencia en lugar de la propiedad Identity en los siguientes casos:

- La aplicación requiere obtener el valor antes de insertar el registro.
- La aplicación requiere compartir series de números entre múltiples tablas o múltiples columnas en las tablas.
- La aplicación requiere reiniciar el valor de la serie con un valor específico. Por ejemplo, reiniciar una secuencia que fue creada desde 1 hasta 100 con los mismos valores.

- La aplicación requiere valores que son ordenados por otro campo. La instrucción «NEXT VALUE FOR function» puede aplicarse la cláusula Over en la función de llamada.
- Una aplicación requiere múltiples valores asignados al mismo tiempo. Por ejemplo, una aplicación necesita obtener tres números seguidos al mismo tiempo.

Como crear una secuencia en SQL Server

Instrucción Create Sequence

Crea una secuencia en SQL Server.

```
CREATE SEQUENCE [Esquema. ] NombreDeSecuencia
[ AS [ TipoEntero | TipoEnteroDefinidoPorElUsuario ] ]
[ START WITH <constante> ]
[ INCREMENT BY <constante> ]
[ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
[ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
[ CYCLE | { NO CYCLE } ]
```

Donde:

NombreDeSecuencia: es el nombre de la secuencia a crear.

TipoEntero: Tipo de dato entero de SQL Server. La tabla está definida líneas arriba.

TipoEnteroDefinidoPorElUsuario: Tipo de dato definido por el usuario en base a los números enteros de SQL Server. (Ver tipos de datos definidos por el usuario)

Start With: define el valor inicial

Increment by: Define el incremento o decremento.

MinValue: Especifica el valor mínimo, por defecto es CERO para el tipo tinyint y un valor negativo para el resto de tipos.

MaxValue: Especifica el valor máximo. El valor por defecto está definido de acuerdo al valor máximo del tipo de dato entero. (Ver tabla arriba).

Cycle: Permite que la secuencia se reinicie cuando llega a su valor mínimo o máximo, dependiendo si es ascendente o descendente.

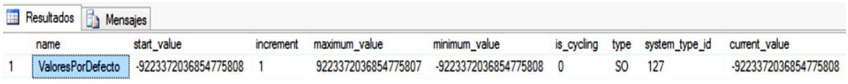
Ejemplo

Crear una secuencia con los valores por defecto

```
CREATE SEQUENCE ValoresPorDefecto
go
```

Para visualizar los datos de la secuencia

```
SELECT name, start_value, increment, maximum_value,
minimum_value,
is_cycling, type, system_type_id, current_value
FROM sys.sequences WHERE name =
'ValoresPorDefecto'
go
```



Resultados | Mensajes

	name	start_value	increment	maximum_value	minimum_value	is_cycling	type	system_type_id	current_value
1	ValoresPorDefecto	-9223372036854775808	1	9223372036854775807	-9223372036854775808	0	SO	127	-9223372036854775808

Figura 8: Resultado de sentencia con uso de Secuencia.

Note que el ID del tipo de dato es 127 para visualizar el tipo de dato

```
SELECT * FROM sys.types WHERE system_type_id = 127
go
```

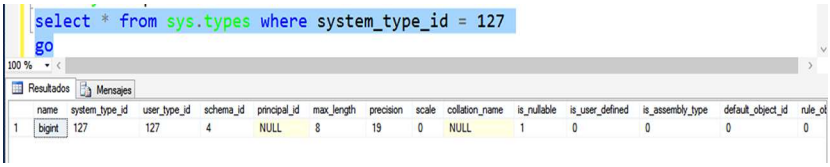


Figura 9: Resultado de sentencia con uso de Secuencia Específica.

Para obtener el valor inicial de acuerdo al tipo de dato bigint. Tenga en cuenta que al ejecutar la siguiente instrucción, el valor de la secuencia se va incrementando 1. En la tabla en la parte superior se puede visualizar el rango del tipo de dato binint.

```
SELECT NEXT VALUE FOR ValoresPorDefecto
go
```

Modificación de una Secuencia

Instrucción Alter Sequence

Modifica los argumentos de una secuencia existente.

Importante: para cambiar el tipo de dato numérico de la secuencia, esta se debe eliminar y luego volver a crear con el nuevo tipo.

Sintaxis:

```
ALTER SEQUENCE [Esquema. ] NombreDeSecuencia
[ RESTART [ WITH <constant> ] ]
[ INCREMENT BY <constant> ]
[ { MINVALUE <constant> } | { NO MINVALUE } ]
[ { MAXVALUE <constant> } | { NO MAXVALUE } ]
[ CYCLE | { NO CYCLE } ]
```

Donde:

NombreDeSecuencia: es el nombre de la secuencia a modificar.

Restart With: define el valor en el que reiniciará la secuencia.

Increment by: Define el incremento o decremento.

MinValue: Especifica el valor mínimo, por defecto es CERO para el tipo tinyint y un valor negativo para el resto de tipos.

MaxValue: Especifica el valor máximo. El valor por defecto está definido de acuerdo al valor máximo del tipo de dato entero. (Ver tabla arriba)

Cycle: Permite que la secuencia se reinice cuando llega a su valor mínimo o máximo, dependiendo si es ascendente o descendente.

Eliminar una secuencia

Instrucción Drop Sequence

Elimina una secuencia de la base de datos

Sintaxis:

```
DROP SEQUENCE [Esquema.]NombreSecuencia
```

Donde:

Esquema: es el nombre del esquema donde se encuentra la secuencia. (Ver esquemas)

NombreSecuencia: nombre de la secuencia a eliminar.

SQL INSERT

La sentencia INSERT INTO se utiliza para insertar nuevas filas en una tabla.

Es posible insertar una nueva fila en una tabla de dos formas distintas:

```
INSERT INTO nombre_tabla
VALUES (valor1, valor2, valor3, …)
```

```
INSERT INTO nombre_tabla (columna1, columna2, columna3,…) VALUES (valor1, valor2, valor3, …)
```

Ejemplo:

Dada la siguiente tabla Autores:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	ALE

Tabla 5: Tabla Autores - Datos Iniciales.

Si queremos insertar una nueva fila en la tabla Autores, lo podemos hacer con cualquiera de las dos sentencias siguientes:

```
INSERT INTO Autores
VALUES (444444, 'Juan Peres Pai', 'BRA')
INSERT INTO Autores (Aut_Id, Nombres, Nacionalidad)
VALUES (555555, 'Albert Rood McMillan', 'EUA')
```

Cualquiera de estas sentencias anteriores produce que se inserte una nueva fila en la tabla autores, quedando así dicha tabla:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	ALE
444444	Juan Peres Pai	BRA
555555	Albert Rood McMillan	EUA

Tabla 6: Tabla Autores - Resultado de Sentencia Insert.

Insertar varias filas de datos

En el ejemplo siguiente se usa el constructor de valores de tabla para insertar tres filas en la tabla Editoriales de la base de datos **miKioskoVirtual** en una sola instrucción INSERT. Dado que los

valores para todas las columnas se suministran e incluyen en el mismo orden que las columnas de la tabla, no es necesario especificar los nombres de columna en la lista de columnas.

```
INSERT INTO Editoriales VALUES
(1,'ULADECH Católica','Chimbote'),
(2,'Rio Santa Editores','Chimbote'),
(3,'Macro Editores','Lima');
```

La sentencia anterior produce que se inserte tres nuevas filas en la tabla Editoriales, quedando así dicha tabla:

Edi_Id	Edi_Nombre	Edi_Ciudad
1	ULADECH Católica	Chimbote
2	Rio Santa Editores	Chimbote
3	Macro Editores	Lima

Tabla 7: Resultado de Uso de Insert en una sola Línea.

Insertar datos solo en columnas especificadas

También es posible insertar solo datos en columnas específicas.

La siguiente declaración SQL insertará un nuevo registro, pero solo insertará datos en la columna «CAT_Nombre» (CAT_Id se actualizará automáticamente):

```
INSERT INTO Categorias (CAT_Nombre)
VALUES ('Ciencia Ficción');
```

Nota: en este caso en particular el Id fue declarado auto_increment (MySQL) o identity (SQLServer), asi mismo para el campo CAT_Descripción la propiedad de aceptar valores nulos está activa, en ambos casos se modifica o crea de la siguiente manera.

Para MySQL:

```
ALTER TABLE Categorias AUTO_INCREMENT=1;
ALTER TABLE Categorias MODIFY COLUMN CAT_Nombre
varchar(50) NOT NULL;
```

Para SQL Server:

```
CREATE TABLE Categorias (
 CAT_Id int IDENTITY(1,1) PRIMARY KEY,
 CAT_Nombres varchar(50) NOT NULL,
 CAT_Descripcion varchar(MAX));
```

Para Access:

```
CREATE TABLE Categorias (
 CAT_Id int PRIMARY KEY AUTOINCREMENT,
 CAT_Nombres varchar(50) NOT NULL,
 CAT_Descripcion varchar(255));
```

Para ORACLE:

```
CREATE SEQUENCE seq_categorias
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 10;
```

Para crear una restricción NOT NULL en la columna «CAT_Descripcion» cuando la tabla «Categorias» ya está creada, use el siguiente SQL:

```
ALTER TABLE Categorias
MODIFY CAT_Descripcion varchar(MAX) NOT NULL;
```

OCURRENCIAS ENCONTRADAS AL MODIFICAR ESTRUCTURA DE LAS TABLAS

Luego de haber creado la base de datos física en el SGBD SQL, podemos notar que el sistema no nos permite en algunos casos la modificación de las tablas.

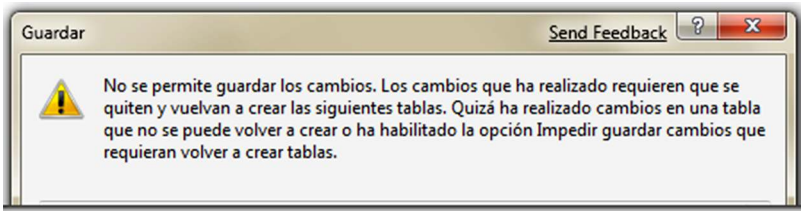


Figura 10: Mensaje que no permite guardar cambios.

Esto se debe a que las versiones del SQL Server a partir del 2008 ya trae activada esta opción. Como veremos aquí la manera de cómo desactivar esta política:

Ingresar al Menú Herramientas y luego Opciones

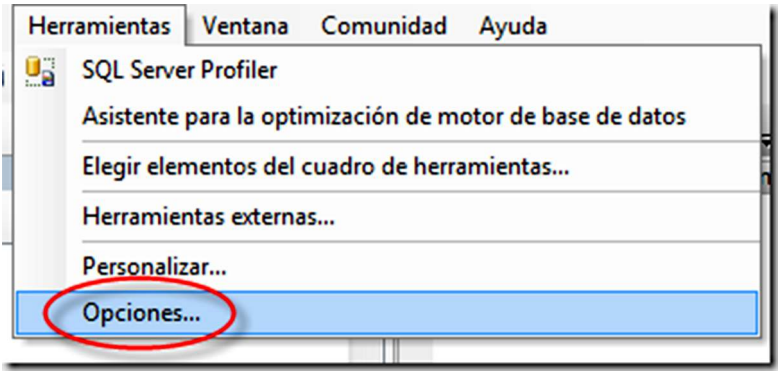


Figura 11: Secuencia para modificar «Guardar cambios en tablas».

Luego la opción Diseñadores (Designer) como se muestra en la siguiente imagen:

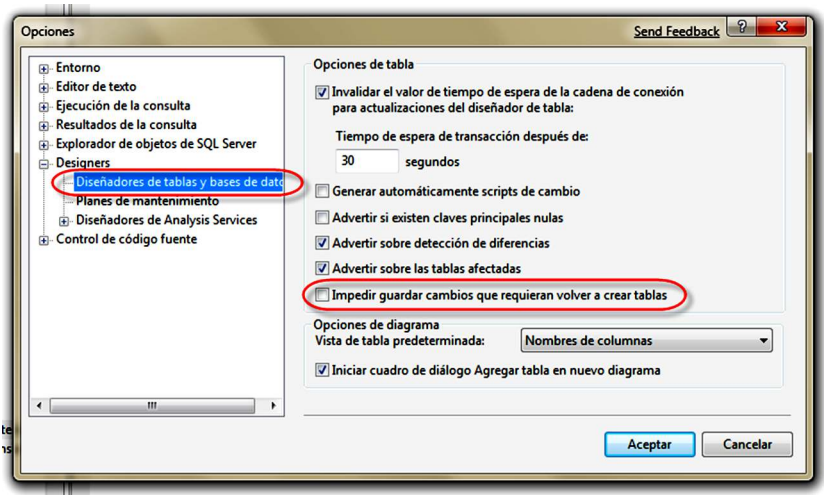


Figura 12: Secuencia final para modificar «Guardar cambios en tablas».

Finalmente Aceptar.

Riesgo de desactivar la opción «Impedir guardar cambios que requieran volver a crear tablas»

Aunque desactivar esta opción puede ayudarle a evitar volver a crear una tabla, también puede conducir a que los cambios se pierdan. Por ejemplo, supongamos que habilita la característica de seguimiento de cambios en SQL Server para realizar un seguimiento de los cambios a la tabla. Al realizar una operación que hace que la tabla vuelva a crearse, recibirá el mensaje de error que se menciona en la sección «Síntomas». Sin embargo, si desactiva esta opción, se elimina la información de seguimiento de cambios existentes cuando se vuelve a crear la tabla. Por lo tanto, recomendamos que solucione temporalmente este problema desactivando la opción.

SQL UPDATE

La sentencia UPDATE se utiliza para modificar valores en una tabla.

La sintaxis de SQL UPDATE es:

```
UPDATE nombre_tabla
SET columna1 = valor1, columna2 = valor2
WHERE columna3 = valor3
```

La cláusula **SET** establece los nuevos valores para las columnas indicadas.

La cláusula **WHERE** sirve para seleccionar las filas que queremos modificar.

Nota: Si se omite la cláusula WHERE, por defecto, modificará los valores en todas las filas de la tabla.

Su segundo tipo de sintaxis es:

```
UPDATE nombre_tabla SET columna1= (Sentencia
SELECT) [WHERE <condición>]
```

Este tipo de actualizaciones sólo son válidas si la Sentencia SELECT devuelve un único valor, que además debe de ser compatible con la columna que se actualiza.

Ejemplo del uso de SQL UPDATE

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	ALE
444444	Juan Peres Pai	BRA
555555	Albert Rood McMillan	EUA

Tabla 8: Tabla Autores Datos Iniciales - Uso UPDATE.

Si queremos cambiar del 'AUT_Id' = 333333 la AUT_Nacionalidad 'ALE' por 'GER' ejecutaremos:

```
UPDATE Autores
SET AUT_Nacionalidad = 'GER'
WHERE AUT_Id = '333333'
```

Ahora la tabla 'Autores' quedará así:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA
555555	Albert Rood McMillan	EUA

Tabla 9: Tabla Autores luego del Uso de UPDATE.

SQL DELETE

La sentencia DELETE sirve para borrar filas de una tabla.

La sintaxis de SQL DELETE es:

```
DELETE FROM nombre_tabla
WHERE nombre_columna = valor
```

La sentencia DELETE es de tipo DML mientras que la sentencia TRUNCATE es de tipo DDL; la diferencia está en dos aspectos:

- DELETE puede borrar 0, 1 o más registros de una tabla, ¿Si deseamos simplemente deshacernos de los datos pero no de la tabla en sí? Para esto podemos usar TRUNCATE TABLE.
- DELETE puede disparar un trigger de tipo DELETE asociado a la tabla con la que estemos trabajando, mientras que TRUNCATE no disparará ningún trigger.

La sintaxis para **TRUNCATE TABLE** es:

```
TRUNCATE TABLE nombre_tabla;
```

Entonces, si deseamos truncar una tabla denominada Clientes que creamos en **SQL CREATE TABLE**, simplemente ingresamos:

```
TRUNCATE TABLE Clientes;
```

O también podemos utilizar la siguiente sentencia para eliminar todos los registros, pero con la salvedad como ya dijimos que se disparará el trigger asociado a la tabla (ver capítulo IV):

```
DELETE * FROM nombre_tabla;
```

Ejemplo de SQL DELETE para borrar una fila de la tabla Autores

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA
555555	Albert Rood McMillan	EUA

Tabla 10: Tabla Autores Inicial - Uso DELETE.

Si queremos borrar a la persona Albert Rood McMillan, podemos ejecutar el comando:

```
DELETE FROM Autores WHERE Aut_Id = 555555;
```

La tabla 'Autores' resultante será:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA

Tabla 11: Resultado Tabla Autores luego de DELETE.

SQL SELECT

La sentencia SELECT recupera datos de una base de datos y los devuelve en forma de resultados de la consulta. Consta de seis cláusulas: las dos primeras (SELECT y FROM) obligatorias y las otras cuatro pueden ser opcionales.

Cláusula SELECT

Utilizado para consultar campos de una a varias tablas.

Sintaxis:

```
SELECT Nombredecampo1, Nombredecampo2, ...
```

Nótese que todos los campos van separados por comas, en el caso que se desee mostrar todos los campos de una tabla solo se debe utilizar el símbolo asterisco (*)

Cláusula FROM

Utilizado para consultar tablas de una base de datos.

Sintaxis:

```
FROM NombredeTabla1 alias_tabla1,
Nombredetabla2 alias_tabla2, …
```

alias_tabla es un nombre que se usa para referirse a la tabla en el resto de la sentencia SELECT para abreviar el nombre original y hacerlo más manejable, en el caso de existir más de una tabla en la consulta y, también para poder realizar consultas uniendo varias veces la misma tabla.

Ahora Veremos los siguientes ejemplos:

Ejemplo1. Seleccionar todas las columnas y registros de la tabla Autores.

```
SELECT * FROM Autores
```

El resultado que se muestra es el siguiente:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA

Tabla 12: Tabla Resultado - Uso Sentencia SELECT.

Ejemplo2. De la tabla Autores seleccionar AUT_Id y AUT_Nombres.

```
SELECT AUT_Id, AUT_Nombres FROM Autores
```

El resultado que se muestra es el siguiente:

Aut_Id	Aut_Nombres
111111	Adamaris Tapia Rosales
222222	Anahí Tapia Rosales
333333	Luz Jacinto Heredia
444444	Juan Peres Pai

Tabla 13: Tabla Resultado - Uso Sentencia SELECT algunos datos.

Ejemplo3. De la tabla Autores seleccionar primero AUT_NAcionalidad y luego AUT_Nombres en ese orden.

```
SELECT AUT_Nacionalidad, AUT_Nombres FROM Autores;
```

El resultado que se muestra es el siguiente:

Aut_Nacionalidad	Aut_Nombres
PER	Adamaris Tapia Rosales
PER	Anahí Tapia Rosales
GER	Luz Jacinto Heredia
BRA	Juan Peres Pai

Tabla 14: Tabla Resultado - Uso Sentencia SELECT, Orden del Usuario.

Y así hemos llegado al final de este capítulo, considerando solamente la parte esencial de la programación en SQL y sus diferentes cláusulas para crear, modificar estructuras de contenedores de datos (DDL), así como para gestionar datos (DML). En el siguiente capítulo mostraremos cómo podemos utilizar otras cláusulas, operadores y funciones que nos permitan atender las necesidades de gestión de datos de nuestro caso práctico, basado en los diferentes gestores de base de datos.

I.6. Ejercicios propuestos

1. Crear una base de datos llamada ComercialFuturo.
2. Hacer uso de la base de datos creada anteriormente y en ella crear la tabla Productos que contenga: código, descripción, precio, stock, fecha de vencimiento, origen y una imagen como foto, asignar la clave primaria al campo código.
3. Suponiendo una institución de enseñanza de nivel escolar, las tablas para niveles que pueden ser inicial (niños de 3 a 5 años), primaria y secundaria y grados, la tabla de niveles tiene en el diseño las restricciones de tipo Primary key, Default, Check y Unique. Grados, tiene su clave primaria y clave foránea a niveles.

   Importante:

   Los nombres de campos deberían tener el nombre de la tabla al inicio, se pueden usar hasta 128 caracteres para el nombre.

   El nombre de la clave primaria (Primary Key) es el nombre de la tabla y las letras PK. Para la tabla grados es GradosPK. Recuerde que se crea el índice agrupado y es importante poder saber el nombre de este.

   En la clave foránea (Foreign Key) se podría incluir los dos nombres de las tablas y luego las letras FK.

4. Crear una secuencia llamada EquipoBasket que inicia en 1 y termina en 12.

5. Crear una secuencia que permita especificar el código para los departamentos en una empresa.
6. Crear una secuencia con valores por defecto y luego modificarla para que su valor inicial sea 10 y se incremente de 5 en 5.
7. Crear una tabla llamada Empleados que tenga la siguiente estructura: EmpleadosCodigo nchar(3), EmpleadosPaterno nvarchar(20), EmpleadosMaterno nvarchar(20), EmpleadosNombre nvarchar(20), EmpleadosDireccion nvarchar(300), EmpleadosFechaNace Date.

   Y la clave primaria asignada a EmpleadosCodigo.
8. Aumentar el tamaño del campo código de 3 a 8 caracteres.

   Al ser la PK debe eliminar la restricción, modificar el campo y luego agregar la PK.
9. Agregar el campo correo.
10. Cambiar el nombre del campo correo por EmpleadosEmail

    /* Precaución: al cambiar cualquier parte del nombre de un objeto pueden dejar de ser scripts válidos y procedimientos almacenados. */
11. Ver la estructura de la tabla.
12. Aumentar los campos de los apellidos y el nombre a 100 caracteres de ancho.
13. Se pueden visualizar los campos de la tabla empleados, considerando que siempre al inicio del nombre de campo se incluya el nombre de la tabla.
14. Agregar los campos teléfono, página web y profesión.
15. Agregar restricción para email y página web que sean únicas.
16. Ver las restricciones de la tabla Empleados.
17. Agregar el campo sueldo.
18. Agregar registros.
19. Agregar restricción para que el sueldo no sea menor a 1200, los registros que no cumplen no se van a modificar.
20. Se puede hacer una consulta para saber quiénes son los registros con esos sueldos.

Capítulo II

USO DE CLÁUSULAS, OPERADORES Y FUNCIONES

II.1. Cláusula WHERE

La cláusula WHERE es usada para hacer filtros en las consultas, es decir, seleccionar solamente algunas filas de la tabla que cumplan una determinada condición haciendo uso para validar esta condición de un operador (ver operadores SQL).

El valor de la condición debe ir entre comillas simples (' ') en el caso de que sea una variable de cadena o fecha y para los demás tipos de datos se omite las comillas.

Ejemplo1. Seleccionar los autores cuya nacionalidad sea **PER** (dato tipo varchar)

```
SELECT * FROM Autores WHERE AUT_Nacionalidad='PER';
```

El resultado que se muestra es:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER

Tabla 15: Tabla Resultado - Uso Cláusula WHERE Ejemplo 1.

Ejemplo2. Seleccionar los datos del autor cuyo AUT_Id es 111111 (dato tipo int).

```
SELECT * FROM Autores WHERE AUT_Id=111111;
```

El resultado que se muestra es:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER

Tabla 16: Tabla Resultado - Uso Cláusula WHERE Ejemplo 2.

Mostrar datos a partir de una SubConsulta

También podemos utilizar una sentencia SELECT como resultado de comparación en la condición de la consulta.

Para este caso específico tenemos primero que obtener los datos resultantes (subconsulta) y luego hacer la comparación

```
SELECT * FROM nombre_tabla WHERE campo1 =
(SELECT campo_resultante FROM tblCondicion
FROM campo_condicion=condicion);
```

Ejemplo 3. Mostrar el Id, Titulo y cantidad física de la tabla Ejemplares cuyo autor es 'Adamaris Tapia Rosales'

```
SELECT Eje_Id, Eje_Titulo, Eje_Cantidad, Eje_Tipo
FROM Ejemplares WHERE AUT_Id = (SELECT AUT_Id
FROM Autores WHERE AUT_Nombres='Adamaris Tapia
Rosales');
```

El Resultado se muestra de la siguiente manera:

Eje_Id	Eje_Titulo	Eje_Cantidad	Tip_Id
121212	Base de datos Organizacionales	10	1
131313	Aplicaciones Web con JSF e Hibernate	15	2
141414	Minería de Datos – Esquemas Expertos		10

Tabla 17: Tabla Resultado - Uso Cláusula WHERE Ejemplo 3

II.2. Operadores SQL

El lenguaje SQL dispone de una multitud de operadores diferentes para cada uno de los tipos de columna.

Esos operadores son utilizados para construir expresiones que se usan en cláusulas ORDER BY y HAVING de la sentencia SELECT y en las cláusulas WHERE de las sentencias.

II.2.1. Operadores lógicos

Los operadores lógicos se usan para crear expresiones lógicas complejas. Permiten el uso de álgebra booleana, y nos ayudarán a crear condiciones mucho más precisas.

Operador lógico AND

Es el «y» lógico. Evalúa dos condiciones y devuelve un valor de verdad solo si ambas son ciertas.

Sintaxis

```
WHERE Condicion1 AND condicion2
```

Se pueden considerar varias condiciones con el operador AND.

La siguiente sentencia (ejemplo AND) dará el siguiente resultado:

Ejemplos

Mostrar los clientes que son de la Ciudad de Chimbote y Lima.

```
SELECT * FROM Clientes WHERE
Cli_Ciudad='Chimbote' AND Cli_Ciudad= 'Lima'
```

Operador lógico OR

Es el «o» lógico. Evalúa dos condiciones y devuelve un valor de verdad solo si una de las dos condiciones se cumple o ambas.

Sintaxis:

```
WHERE condicion1 OR condicion2
```

Se pueden considerar varias condiciones con el operador OR.

La siguiente sentencia (ejemplo OR) dará el siguiente resultado:

Mostrar los clientes que son de la Ciudad de Chimbote o Lima

```
SELECT * FROM Clientes WHERE
Cli_Ciudad='Chimbote' OR Cli_Ciudad= 'Lima'
```

También se pueden combinar AND y OR, como el siguiente ejemplo:

Mostrar los clientes Naturales que son de la Ciudad de Chimbote o Lima.

```
SELECT * FROM Clientes WHERE Cli_Tipo='N' AND
(Cli_Ciudad='Chimbote' OR Cli_Ciudad= 'Lima')
```

Operador lógico NOT

El operador «NOT» invierte el resultado de la condición a la cual antecede.

Los registros recuperados en una sentencia en la cual aparece el operador «NOT» no cumplen con la condición a la cual afecta el «NOT».

Sintaxis:

```
WHERE NOT condicion1
```

Se pueden considerar varias condiciones con el operador NOT.

La siguiente sentencia (ejemplo NOT) dará el siguiente resultado:

Mostrar aquellos clientes que no son de Lima.

```
SELECT * FROM Clientes WHERE NOT Cli_Ciudad= 'Lima'
```

También se pueden combinar con AND y OR, como el siguiente ejemplo:

Mostrar los clientes Naturales que son de la Ciudad de Chimbote o no son Lima

```
SELECT * FROM Clientes WHERE Cli_Tipo='N' AND
(Cli_Ciudad='Chimbote' OR NOT Cli_Ciudad= 'Lima')
```

El orden de prioridad de los operadores lógicos es el siguiente: «NOT» se aplica antes que «AND» y «AND» antes que «OR», si no se especifica un orden de evaluación mediante el uso de paréntesis.

El orden en el que se evalúan los operadores con igual nivel de precedencia es indefinido, por ello se recomienda usar los paréntesis.

II.2.2. Operadores de comparación

Para crear expresiones lógicas, a las que podremos aplicar el álgebra de Boole, disponemos de varios operadores de comparación. Estos operadores se aplican a cualquier tipo de columna: fechas, cadenas, números, etc, y devuelven valores lógicos: verdadero o falso (1/0).

Los operadores de comparación son los habituales en cualquier gestor de base de datos, lenguaje de programación o frameworks, pero además, SQL añade varios más que resultan de mucha utilidad, ya que son de uso muy frecuente.

Reglas para el uso de operadores de comparación

SQL sigue las siguientes reglas a la hora de comparar valores:

Si uno o los dos valores a comparar son NULL, el resultado es NULL, excepto con el operador <=>, de comparación con NULL segura.

Si los dos valores de la comparación son cadenas, se comparan como cadenas.

Si ambos valores son enteros, se comparan como enteros.

Los valores hexadecimales se tratan como cadenas binarias, si no se comparan con un número.

Si uno de los valores es del tipo TIMESTAMP, DATETIME o DATETIME2 y el otro es una constante, la constante se convierte a timestamp antes de que se lleve a cabo la comparación. Hay que tener en cuenta que esto no se hace para los argumentos de una expresión IN(). Para estar seguro, es mejor usar siempre cadenas completas datetime/date/time strings cuando se hacen comparaciones.

En el resto de los casos, los valores se comparan como números en coma flotante.

Operador de comparación de igualdad

El operador «=» compara dos expresiones, y da como resultado 1 si son iguales, o 0 si son diferentes. Ya lo hemos usado en ejemplos anteriormente.

Hay que mencionar que, al contrario que otros lenguajes, como C o C++, donde el control de tipos es muy estricto, en SQL se pueden comparar valores de tipos diferentes, y el resultado será el esperado.

Sintaxis

```
WHERE nombre_columna = valor
```

La siguiente sentencia (ejemplo =) dará el siguiente resultado:

Mostrar los datos del cliente con el id 32991246

```
SELECT * FROM Clientes WHERE cli_id=32991256;
```

Operador de Comparación de Desigualdad

SQL dispone de dos operadores equivalente para comprobar desigualdades, <> y !=. Si las expresiones comparadas son diferentes, el resultado es verdadero, y si son iguales, el resultado es falso:

Sintaxis

```
WHERE nombre_columna != valor
```

La siguiente sentencia (ejemplo !=) dará el siguiente resultado:

Mostrar aquellos clientes que no son de Lima

```
SELECT * FROM Clientes WHERE NOT Cli_Ciudad != 'Lima'
```

Operador de Comparación de Magnitud

Disponemos de los cuatro operadores corrientes.

Operador	Descripción
<=	Menor o igual
<	Menor
>	Mayor
>=	Mayor o igual

Tabla 18: Operadores de Comparación de Magnitud.

Estos operadores también permiten comparar cadenas, fechas y, por supuesto, números:

Cuando se comparan cadenas, se considera menor la cadena que aparezca antes por orden alfabético.

Si son fechas, se considera que es menor cuanto más antigua sea.

Las siguientes sentencias mostrarán los siguientes resultados:

Mostrar ejemplares cuya cantidad es menor a 10

```
SELECT * FROM Ejemplares where eje_cantidad < 10;
```

Mostrar ejemplares cuya cantidad es mayor igual a 50

```
SELECT * FROM Ejemplares where eje_cantidad >= 50;
```

Operador de comparación BETWEEN

Entre los operadores de SQL, hay uno para comprobar si una expresión está comprendida en un determinado rango de valores.

La sintaxis es:

```
WHERE nombre_columna BETWEEN ValorMinimo
AND ValorMaximo
```

En realidad es un operador prescindible, ya que se puede usar en su lugar dos expresiones de comparación y el operador AND.

La siguiente declaración SQL selecciona todos los ejemplares con un precio ENTRE 5 y 20:

```
SELECT * FROM Ejemplares WHERE EJE_Precio
BETWEEN 5 AND 20;
```

Para mostrar los productos fuera del rango del ejemplo anterior, use NOT BETWEEN

```
SELECT * FROM Ejemplares WHERE Eje_Precio NOT
BETWEEN 5 AND 20;
```

La siguiente declaración SQL selecciona todos los productos con un precio ENTRE 5 y 20. Además; No muestre productos con un Tipo_Id de 1,2 o 3:

```
SELECT * FROM Ejemplares
WHERE (Eje_Precio BETWEEN 5 AND 20)
AND NOT Tipo_Id IN (1,2,3);
```

La siguiente declaración SQL selecciona todos los Ejemplares con un EJE_Titulo ENTRE 'Aplicaciones Web con JSF e Hibernate' y 'Minería de Datos – Esquemas Expertos':

```
SELECT * FROM Ejemplares WHERE EJE_Titulo
BETWEEN 'Aplicaciones Web con JSF e Hibernate'
AND 'Minería de Datos – Esquemas Expertos'
ORDER BY EJE_Titulo;
```

Operador de Comparación IN y NOT IN

Los operadores IN y NOT IN sirven para averiguar si el valor de una expresión determinada está dentro de un conjunto indicado.

Sintaxis:

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE nombre_columna IN (valor1, valor2, ...);
```

O también puede ser la lista el resultado de una subconsulta:

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE nombre_columna IN (SELECT Sentencia);
```

La siguiente declaración SQL selecciona todos los Clientes que están ubicados en «Chimbote», «Lima» o «Sullana»:

```
SELECT * FROM Clientes
WHERE CLI_Ciudad IN ('Chimbote', 'Lima', 'Sullana');
```

La siguiente declaración SQL selecciona todos los Clientes que NO están ubicados en «Chimbote», «Lima» o «Sullana»:

```
SELECT * FROM Clientes
WHERE CLI_Ciudad NOT IN ('Chimbote', 'Lima', 'Sullana');
```

La siguiente declaración SQL selecciona los pedidos realizados por clientes de la ciudad de 'Chimbote'

```
SELECT * FROM Pedidos
WHERE CLI_Id IN (SELECT CLI_Id FROM Clientes
WHERE CLI_Ciudad ='Chimbote');
```

Operador de Comparación LIKE

El Operador LIKE es utilizado en la comparación de un modelo. Hay dos comodines utilizados junto con el operador LIKE:

- %: El signo de porcentaje representa cero, uno o varios caracteres
- _ - El subrayado representa un solo carácter

Nota: MS Access utiliza un signo de interrogación (?) En lugar del subrayado (_).

Sintaxis:

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE Campo LIKE Modelo
```

Consejo: También puede combinar cualquier número de condiciones utilizando los operadores AND u OR.

Aquí hay algunos ejemplos que muestran diferentes operadores LIKE con los comodines '%' y '_':

Operador LIKE	Descripción
SELECT * FROM Ejemplares **WHERE** Eje_Titulo **LIKE 'a%'**	Encuentra cualquier valor que comience con «a»
SELECT * FROM Ejemplares **WHERE** Eje_Titulo **LIKE '%a'**	Encuentra cualquier valor que termine con «a»
SELECT * FROM Ejemplares **WHERE** Eje_Titulo **LIKE '%or%'**	Encuentra cualquier valor que tenga «or» en cualquier posición
SELECT * FROM Ejemplares **WHERE** Eje_Titulo **LIKE '_r%'**	Encuentra cualquier valor que tenga «r» en la segunda posición
SELECT * FROM Ejemplares **WHERE** Eje_Titulo **LIKE 'a_%_%'**	Encuentra cualquier valor que comience con «a» y tenga al menos 3 caracteres de longitud
SELECT * FROM Ejemplares **WHERE** Eje_Titulo **LIKE 'a%o'**	Encuentra cualquier valor que comience con “a” y termine con “o”

Tabla 19: Ejemplos Varios de Uso Operador LIKE.

También lo podemos utilizar desde un arreglo de caracteres

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE Campo LIKE '[arreglo]comodin'
```

La siguiente declaración SQL muestra los ejemplares que empiecen con “C” y “D”

```
SELECT * FROM Ejemplares WHERE EJE_Titulo LIKE '[CD]%';
```

II.3. Funciones del sistema

Los SGBD como SQL Server, MySQL, Oracle, Access, etc. tienen muchas funciones integradas.

Esta referencia contiene cadenas, números, fechas, conversiones y algunas funciones avanzadas para cada SGBD.

II.3.1. Funciones de Agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Ahora que nos estamos familiarizando con las cláusulas básicas que se pueden utilizar con la instrucción SELECT, veamos algunas funciones para tener mayor flexibilidad en la creación de consultas.

Función AVG

Esta Función es utilizada para calcular el promedio de los valores de un campo determinado.

Sintaxis:

```
SELECT AVG(nombre_columna) FROM nombre_tabla
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar el promedio de cantidades vendidas del ejemplar 111111 en la tabla detalle de pedido

```
SELECT AVG(Ped_Cantidad) FROM DetallePedido
WHERE Eje_Id=111111
```

Función COUNT

Esta Función es utilizada para devolver el número de registros de la selección.

Sintaxis

```
SELECT COUNT (nombre_columna) FROM nombre_tabla
```

Nota: Utilizar esta función apuntando a una columna que sabemos que no acepta valores nulos o en algunos casos reemplazar por el comodín Asterisco (*)

La siguiente sentencia mostrará el siguiente resultado:

Mostrar cuántos ejemplares están registrados en la tabla ejemplares.

```
SELECT COUNT(Eje_Id) FROM Ejemplares
```

Función SUM

Utilizada para devolver la suma de todos los valores de un campo determinado

Sintaxis

```
SELECT SUM(nombre_columna) FROM nombre_tabla
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar el total de ejemplares vendidos del ejemplar con el id 111111 en la tabla detalle de pedido

```
SELECT SUM(Ped_Cantidad) FROM DetallePedido
WHERE Eje_Id=111111
```

Función MAX

Esta Función es utilizada para devolver el valor más alto de un campo especificado

Sintaxis:

```
SELECT MAX(nombre_columna) FROM nombre_tabla
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar la cantidad más alta del ejemplar 111111 vendido a una persona.

```
SELECT MAX(ped_cantidad) FROM detallepedido
WHERE eje_id=111111
```

Función MIN

Utilizada para devolver el valor más bajo de un campo especificado

Sintaxis:

```
SELECT MIN(nombre_columna) FROM nombre_tabla
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar la cantidad más baja del ejemplar 111111 vendido a una persona.

```
SELECT MIN(ped_cantidad) FROM detallepedido
WHERE eje_id=111111
```

Función STDDEV

Utilizada para devolver la desviación estándar de los valores de un campo especificado. Generalmente para obtener datos estadísticos

Sintaxis :

```
SELECT STDDEV(nombre_columna) FROM nombre_tabla
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar la desviación estándar de las ventas del ejemplar 111111

```
SELECT STDDEV(ped_cantidad * ped_precioVenta)
FROM detallepedido WHERE eje_id=111111
```

Función VAR

Utilizada para devolver la varianza de los valores de un campo especificado.

Sintaxis :

```
SELECT VAR(nombre_columna) FROM nombre_tabla
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar la varianza de las ventas del ejemplar 111111

```
SELECT VAR(ped_cantidad * ped_precioVenta) FROM
detallepedido WHERE eje_id=111111
```

Como hemos señalado, una función de agregado realiza un cálculo sobre un conjunto de valores y devuelve un único valor. Las funciones de agregado se pueden especificar en la lista de selección y se usan frecuentemente cuando la instrucción contiene una cláusula GROUP BY.

II.3.2. Funciones de Cadena

Estas funciones son utilizadas para manipular datos de tipo cadena (char, varchar, nvarchar, etc), las que nos permiten devolver o modificar un dato. Cada gestor gestiona sus propias funciones de cadena, pero en algunos casos estas funciones con la misma sintaxis tienen el soporte en la mayoría de ellos.

Función ASCII

Devuelve el valor de código ASCII del carácter más a la izquierda de la cadena String. Devuelve 0 si String es una cadena vacía.

Sintaxis:

```
SELECT ASCII(cadena)
```

La siguiente sentencia mostrará el siguiente resultado:

Mostrar el código ASCII asignada a la letra A

```
SELECT ASCII('A')
```

Función LEFT

Devuelve un determinado número de caracteres contando desde el lado izquierdo de la cadena:

Sintaxis:

```
LEFT(Cadena, numero_caracteres)
```

II.3.3. Funciones de Fecha

La parte más difícil cuando se trabaja con las fechas es estar seguro de que el formato de la fecha en la que está intentando insertar, coincide con el formato de la columna de la fecha en la base de datos.

Funciones de Fecha MySQL

La siguiente tabla enumera las más importantes funciones de fecha incorporadas en MySQL:

---

4 ASCII (acrónimo inglés de American Standard Code for Information Interchange —Código Estándar Estadounidense para el Intercambio de Información—), es un código de caracteres basado en el alfabeto. (10)

Función	Descripción
NOW()	Devuelve la fecha y la hora actual
CURDATE()	Devuelve la fecha actual
CURTIME()	Devuelve la hora actual
DATE()	Extrae la parte de fecha de una fecha o expresión de fecha / hora
EXTRACT()	Devuelve una sola parte de una fecha / hora
DATE_ADD()	Añade un intervalo de tiempo especificado en una fecha
DATE_SUB()	Resta un intervalo de tiempo especificado desde una fecha
DATEDIFF()	Devuelve el número de días entre dos fechas
DATE_FORMAT()	Muestra datos de fecha / hora en formatos diferentes

Tabla 20: Funciones de Fecha - SGBD MySQL.

Ejemplos:

La siguiente instrucción SELECT:

```
SELECT NOW(), CURDATE(), CURTIME()
```

dará lugar a algo como esto:

NOW()	CURDATE()	CURTIME()
2018-11-30 13:15:25	2018-11-30	13:15:25

Tabla 21: Resultado de Uso Funciones de Fech

Ahora queremos insertar un registro en la tabla "Pedido":

```
INSERT INTO Pedido VALUES (4, 42917459, NOW(),'P')
```

En la tabla “Pedido” ahora se verá algo como esto:

PED_ Id	PED_ Nro	CLI_Id	PED_Fecha	PED_ Estado
1	1	42991246	2018-11-25 10:12:15	A
2	2	30423493	2018-11-26 13:15:10	E
3	2	30423493	2018-11-26 13:30:05	A
4	3	42864186	2018-11-30 16:03:15	A
5	4	42917459	2018-11-30 18:23:35	P

Tabla 22: Resultado de Aplicación de función de fecha en un tabla.

La siguiente instrucción SELECT:

```
SELECT PED_Id, PED_Nro, CLI_Id, DATE(PED_Fecha)
AS Fecha_Pedido, PED_Estado FROM Pedido WHERE
PED_Id=5
```

dará lugar a esto:

PED_ Id	PED_ Nro	CLI_Id	Fecha_Pedido	PED_ Estado
5	4	42917459	2018-11-30	P

Tabla 23: Resultado de Uso de la Función DATE.

La siguiente instrucción SELECT:

```
SELECT EXTRACT(YEAR FROM PED_Fecha) AS AñoPedido,
EXTRACT(MONTH FROM PED_Fecha) AS MesPedido,
EXTRACT(DAY FROM PED_Fecha) AS DiaPedido FROM Pedido
WHERE PED_Id=5
```

dará lugar a esto:

AñoPedido	MesPedido	DiaPedido
2018	11	30

Tabla 24: Resultado de Uso de la Función EXTRACT.

Podemos utilizar la siguiente tabla para las partes de fecha en MySQL:

PARTES DE FECHA EN MYSQL
MICROSECOND
SECOND
MINUTE
HOUR
DAY
WEEK
MONTH
QUARTER
YEAR
SECOND_MICROSECOND
MINUTE_MICROSECOND
MINUTE_SECOND
HOUR_MICROSECOND
HOUR_SECOND
HOUR_MINUTE
DAY_MICROSECOND
DAY_SECOND
DAY_MINUTE
DAY_HOUR
YEAR_MONTH

Tabla 25: Partes de Fecha en MYSQL.

Ahora queremos añadir 4 días para el "PED_Fecha", para encontrar la fecha de Entrega. Nosotros usamos la siguiente instrucción SELECT:

```
SELECT PED_Id, DATE_ADD(PED_Fecha, INTERVAL 4
DAY) AS Fecha_Entrega FROM Pedido
```

La siguiente secuencia de comandos utiliza el DATE_FORMAT() función para visualizar diferentes formatos. Vamos a utilizar el NOW() función para obtener la fecha / hora actual:

```
DATE_FORMAT(NOW(),'%b %d %Y %h:%i %p')
DATE_FORMAT(NOW(),'%m-%d-%Y')
DATE_FORMAT(NOW(),'%d %b %y')
DATE_FORMAT(NOW(),'%d %b %Y %T:%f')
```

El resultado sería algo como esto:

```
Nov 30 2018 13:45 PM
30-11-2018
30 Nov 18
30 Nov 2018 13:45:34:243
```

Servidor SQL Funciones de fecha

La siguiente tabla enumera las más importantes funciones de fecha incorporadas en SQL Server:

Función	Descripción
GETDATE()	Devuelve la fecha y la hora actual.
DATEPART()	Devuelve una sola parte de una fecha / hora.
DATEADD()	Suma o resta un intervalo de tiempo especificado desde una fecha.
DATEDIFF()	Devuelve el tiempo entre dos fechas.
DATENAME()	Devuelve el nombre de la parte de una fecha / hora.

Tabla 26: Funciones de Fecha - SQL Server.

Mostraremos algunos ejemplos:

La siguiente instrucción SELECT:

```
SELECT GETDATE() AS Fecha_Hora_Actual
```

dará lugar a algo como esto:

Fecha_Hora_Actual
2018-11-30 17:54:14.243

Ahora queremos obtener el número de días entre dos fechas. Usaremos la siguiente instrucción SELECT para mostrar los días transcurridos desde el primer pedido efectuado:

```
SELECT TOP 1 DATEDIFF(day, PED_Fecha,
GETDATE()) AS Dias_Transcurridos FROM Pedido;
```

Resultado:

Dias_Trasncurridos
5

En el caso de SQL Server utilizaremos los siguientes partes de fecha mostrados en la siguiente tabla

Parte de fecha	Abreviatura
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw, w
hour	Hh
minute	mi, n
second	ss, s
millisecond	Ms
microsegundo	Mcs
nanosecond	Ns

Tabla 27: Partes de Fecha - SQL Server.

Fecha de tipos de datos SQL

MySQL incluye los siguientes tipos de datos para almacenar una fecha o un valor de fecha / hora en la base de datos:

- FECHA - formato AAAA-MM-DD
- DATETIME - formato: AAAA-MM-DD HH: MI: SS
- TIMESTAMP - formato: AAAA-MM-DD HH: MI: SS
- AÑO - AAAA formato o YY

SQL Server incluye los siguientes tipos de datos para almacenar una fecha o un valor de fecha / hora en la base de datos:

- FECHA - formato AAAA-MM-DD
- DATETIME - formato: AAAA-MM-DD HH: MI: SS
- SMALLDATETIME - formato: AAAA-MM-DD HH: MI: SS
- TIMESTAMP - formato: un número único

En la siguiente tabla Pedido con todos los campos seleccionados:

PED Id	PED_ Nro	CLI_Id	PED_Fecha	PED_ Estado
1	1	42991246	2018-11-25	A
2	2	30423493	2018-11-26	E
3	2	30423493	2018-11-26	A
4	3	42864186	2018-11-30	A

Tabla 28: Tabla Pedidos - Datos Iniciales.

Ahora queremos seleccionar los registros con un PED_Fecha de "2018-11-26" de la tabla anterior. Nosotros usamos la siguiente instrucción SELECT:

```
SELECT * FROM Pedido WHERE PED_Fecha='2018-11-26'
```

El conjunto de resultados se verá así:

PED Id	PED_ Nro	CLI_Id	PED_Fecha	PED_ Estado
2	2	30423493	2018-11-26	E
3	2	30423493	2018-11-26	A

Tabla 29: Resultado de Uso de Datos tipo fecha.

Ahora, supongamos que la tabla "Pedido" es el siguiente (nótese el componente de tiempo en la columna "PED_Fecha"):

PED Id	PED_ Nro	CLI_Id	PED_Fecha	PED_ Estado
1	1	42991246	2018-11-25 10:12:15	A
2	2	30423493	2018-11-26 13:15:10	E
3	2	30423493	2018-11-26 13:30:05	A
4	3	42864186	2018-11-30 16:03:15	A

Tabla 30: Tabla Pedido - Otros formatos de fecha

Si utilizamos la misma instrucción SELECT que el anterior:

```
SELECT * FROM Pedido WHERE PED_Fecha='2018-11-26'
```

obtendremos ningún resultado! Esto se debe a que la consulta está mirando solamente para citas con ninguna porción de tiempo.

II.3.4. Funciones CAST y CONVERT

Se puede realizar conversiones en tiempo de ejecución entre tipos de datos compatibles utilizando las funciones CAST y CONVERT.

Algunos tipos de datos requieren una conversión explícita a otros tipos de datos utilizando las funciones CAST o CONVERT. Otros tipos de datos se pueden convertir implícitamente, como parte de otro comando, sin utilizar las funciones CAST y CONVERT.

CAST

Puede usar dos formas equivalentes de sintaxis para convertir expresiones de un tipo de dato a otro:

```
CAST (expresión_columna AS nuevotipo)
```

Podemos ver el uso en el siguiente formato de MySQL

```
SELECT columna(s) FROM miTabla
WHERE CAST(dateColumn AS DATE) = NOW()
```

O lo podemos utilizar para mostrar datos de la tabla Pedido utilizando el gestor SQL Server

```
SELECT * FROM DocVenta WHERE CAST(Fecha AS
date) = CAST(GETDATE() AS date);
```

CONVERT

El CONVERT() la función es una función general que convierte una expresión de un tipo de datos a otro.

El CONVERT() función se puede utilizar para mostrar los datos de fecha / hora en formatos diferentes.

Sintaxis

```
CONVERT(data_type(length),expression,style)
```

Donde: Style es el formato que queremos darle a la fecha

La siguiente secuencia de comandos utiliza el CONVERT() función para visualizar diferentes formatos. Vamos a utilizar el GETDATE() función para obtener la fecha / hora actual:

```
CONVERT(VARCHAR(19),GETDATE())
CONVERT(VARCHAR(10),GETDATE(),10)
CONVERT(VARCHAR(10),GETDATE(),110)
CONVERT(VARCHAR(11),GETDATE(),6)
CONVERT(VARCHAR(11),GETDATE(),106)
CONVERT(VARCHAR(24),GETDATE(),113)
```

El resultado sería algo como esto:

```
Nov 30 2018 7:45 PM
11-30-18
11-30-2018
30 Nov 18
30 Nov 2018
30 Nov 2018 19:45:34:243
```

II.3.5. Funciones NULL

Si queremos cambiar un valor NULL por otro valor cualquiera, utilizaremos las siguientes funciones (ISNULL, IFNULL, NVL, COLACESCE) según el sistema de base de datos.

Para nuestros ejemplos, queremos que si el valor es NULL se cambie por el valor 0

Ejemplo para SQL SERVER se utiliza ISNULL:

```
SELECT Eje_Titulo, Eje_Precio * ISNULL(Eje_Cantidad,0)
as TotalPosibleVenta FROM ejemplares
```

Ejemplo para ORACLE se utiliza NVL:

```
SELECT Eje_Titulo, Eje_Precio * NVL(Eje_Cantidad, 0)
FROM Ejemplares
```

Ejemplo para MySQL, hay 2 funciones equivalentes (IFNULL, COALESCE):

```
SELECT Eje_Titulo, Eje_Precio * IFNULL(Eje_Cantidad, 0)
FROM Ejemplares
```

```
SELECT Eje_Titulo, Eje_Precio * COALESCE
(Eje_Cantidad, 0) FROM Ejemplares
```

II.3.6. Funciones Matemáticas

Las funciones matemáticas realizan cálculos basados en valores de entrada y reportan un valor numérico.

Las funciones matemáticas son:

Función	Descripción
ABS(Exp. Numérica)	Reporta el valor absoluto de una expresión numérica.
DEGREES(Exp. Numérica)	Reporta el valor del ángulo en grados de uno expresado en radianes.
RAND()	Reporta un número aleatorio entre 0 y 1.
ACOS(Exp. Numérica)	Reporta el ángulo en radianes llamado Arco Coseno.
EXP(Exp. Numérica)	Reporta el valor exponencial de la expresión numérica.
ROUND(Exp. Numérica, n)	Reporta una expresión numérica redondeada en n decimales.
ASIN(Exp. Numérica)	Reporta el ángulo en radianes llamado Arco seno.
FLOOR(Exp. Numérica)	Reporta el entero menor o igual que la expresión numérica especificada.
SIGN(Exp. Numérica)	Reporta el signo de la expresión numérica.
ATAN(Exp. Numérica)	Reporta el ángulo en radianes llamado Arco Tangente.
LOG(Exp. Numérica [,base])	Reporta el logaritmo natural de una expresión numérica.
SIN(Exp. Numérica)	Reporta el seno de un ángulo expresado en radianes.
ATN2(Exp. Numérica1, Exp. Numérica2)	Devuelve el ángulo, en radianes, entre el eje x positivo y el rayo desde el origen hasta el punto (y, x), donde x e y son los valores de las dos expresiones flotantes especificadas.
LOG10(Exp. Numérica)	Reporta el logaritmo en base 10 de la expresión numérica.
SQRT(Exp. Numérica)	Reporta la raíz cuadrada de la expresión numérica
CEILING(Exp. Numérica)	Reporta el entero más pequeño mayor o igual que la expresión numérica especificada.
PI()	Reporta el valor de Pi.
SQUARE(Exp. Numérica)	Reporta el cuadrado de la expresión numérica
COS(Exp. Numérica)	Reporta el ángulo en radianes llamado Coseno.
POWER(Exp. Numérica, n)	Reporta la expresión numérica elevada a la n potencia.
TAN(Exp. Numérica)	Reporta el ángulo en radianes llamado Tangente.
COT(Exp. Numérica)	Reporta el ángulo en radianes llamado Cotangente.
RADIANS(Exp. Numérica)	Reporta el valor en radianes de un ángulo especificado.

Tabla 31: Funciones Matemáticas.

Ejemplos:

Calcular el valor en radianes de 2 PI()

```
SELECT 'El valor de PI*2 en radianes es: ' +
CONVERT(varchar, DEGREES((PI()*2)))
Go
```

Calcular el valor del arco coseno de -1

```
SELECT 'El Arco Coseno de -1 es: ' +
CONVERT(varchar, ACOS(-1))
Go
```

II.3.7. Valores NULL

El valor NULL representa a un valor desconocido.

Este valor NULL puede ser asignado como valor a cualquier columna de una tabla.

Si el valor de una columna es opcional, quiere decir, que podemos insertar una fila en la tabla sin asignarle ningún valor a esa columna opcional, así que esa columna tomará el valor NULL.

El valor NULL es un valor especial, y por tanto, no se puede comparar con los operadores aritméticos normales (=, >, <, <>), y en su lugar debemos utilizar los operadores IS y IS NOT.

En la tabla Autores, tenemos la columna 'AUT_Nacionalidad' que es opcional y puede tener valores nulos:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	
333333	Luz Jacinto Heredia	
444444	Juan Peres Pai	BRA

Tabla 32: Tabla Autores con Nacionalidad NULA.

Ejemplo de uso de IS NULL

```
SELECT * FROM Autores WHERE
AUT_Nacionalidad IS NULL
```

Aut_Id	Aut_Nombres	Aut_Nacionalidad
222222	Anahí Tapia Rosales	
333333	Luz Jacinto Heredia	

Tabla 33: Ejemplo IS NULL.

Ejemplo de uso de IS NOT NULL

```
SELECT * FROM Autores WHERE
AUT_Nacionalidad IS NOT NULL
```

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
444444	Juan Peres Pai	BRA

Tabla 34: Ejemplo IS NOT NULL

II.4. Otras Cláusulas SQL

Las cláusulas o palabras claves adicionales que a continuación utilizaremos son opcionales a la sentencia SQL. Estas pueden complementar la información a mostrar en la atención del requerimiento de los usuarios de la Base de datos, ya agrupa, ordena, filtra, segmenta y valida datos en la misma sentencia. Para nuestro caso utilizaremos estas sentencias para atender los requerimientos de consulta y posterior emisión de reportes de nuestra base de datos.

II.4.1. SQL GROUP BY

La instrucción GROUP BY se usa a menudo con funciones agregadas (COUNT, MAX, MIN, SUM, AVG) para agrupar el conjunto de resultados por una o más columnas.

Una consulta con una cláusula GROUP BY se denomina consulta agrupada ya que agrupa los datos de la tabla origen y produce una única fila resumen por cada grupo formado. Recuerde que es opcional

Sintaxis del GROUP BY

```
SELECT nombre_columna(s)
FROM nombre_tabla
WHERE condicion
GROUP BY nombre_columna(s)
ORDER BY nombre_columna(s);
```

II.4.2. SQL ORDER BY

La palabra clave ORDER BY se utiliza para ordenar el conjunto de resultados en orden ascendente o descendente.

La palabra clave ORDER BY ordena los registros en orden ascendente por defecto y para ordenar los registros en orden descendente, use la palabra clave DESC. La tabla no se modifica, recuerde que es opcional.

Sintaxis de ORDER BY

```
SELECT columna1, columna2, ...
FROM nombre_tabla
ORDER BY columna1, columna2, ... ASC|DESC;
```

Mostrar los clientes naturales ordenados por apellido paterno

```
SELECT * FROM CliNatural ORDER BY APaterno
```

Mostrar los clientes naturales ordenados por fecha de nacimiento de manera descendente

```
SELECT * FROM CliNatural ORDER BY FNacimiento DESC
```

II.4.3. SQL HAVING

La cláusula HAVING se agregó a SQL porque la palabra clave WHERE no se pudo usar con funciones agregadas.

La cláusula HAVING nos permite seleccionar filas de la tabla resultante de una consulta de resumen.

Recuerde también es opcional

Sintaxis de HAVING

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE condicion
GROUP BY nombre_columna(s)
HAVING condicion
ORDER BY nombre_columna(s);
```

Mostrar los pedidos de los ejemplares cuyas cantidad de ejemplares hayan superado los 100 ejemplares

```
SELECT ped_id, eje_id FROM detallepedido HAVING
SUM(ped_cantidad)>100
```

II.4.4. SQL EXISTS

El operador EXISTS se utiliza para probar la existencia de cualquier registro en una subconsulta.

El operador EXISTS devuelve verdadero si la subconsulta devuelve uno o más registros.

Sintaxis del operador EXISTS

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE EXISTS
(SELECT nombre_columna FROM nombre_tabla
WHERE condicion);
```

Mostrar el detalle del pedido en el proceso de pedido del dia 12-12-2018

```
SELECT * FROM DetallePedido WHERE EXISTS
(SELECT ped_id FROM Pedido WHERE
ped_fecha='20818-12-12');
```

II.4.5. SQL ANY/ALL

Los operadores ANY y ALL se utilizan con una cláusula WHERE o HAVING.

El operador ANY devuelve true si alguno de los valores de la subconsulta cumple la condición.

El operador ALL devuelve verdadero si todos los valores de subconsulta cumplen la condición.

Sintaxis del operador ANY

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE nombre_columna operador ANY
(SELECT nombre_columna FROM nombre_tabla
WHERE condicion);
```

Sintaxis del operador ANY

```
SELECT nombre_columna(s) FROM nombre_tabla
WHERE nombre_columna operador ALL
(SELECT nombre_columna FROM nombre_tabla
WHERE condicion);
```

Nota: El operador debe ser un operador de comparación estándar (=, <>, !=, >, >=, <, or <=).

Mostrar los clientes si son de la ciudad de Chimbote, Lima o Arequipa.

```
SELECT * FROM Clientes Where Cli_Ciudad ANY
('Chimbote','Lima','Arequipa')
```

Mostrar los clientes si son de la ciudad de Chimbote, Lima y Arequipa.

```
SELECT * FROM Clientes Where Cli_Ciudad ALL
('Chimbote','Lima','Arequipa')
```

II.4.6. SQL DISTINCT

Al realizar una consulta puede ocurrir que existan valores repetidos para algunas columnas. Por ejemplo

```
SELECT Eje_Titulo FROM Ejemplares
```

Eje_Titulo
Base de datos Organizacionales
Base de datos Organizacionales
Aplicaciones WEB Empresariales
Aplicaciones Web con JSF e Hibernate
Minería de Datos – Esquemas Expertos

Tabla 35: Resultado Ejemplo sin DISTINCT.

Esto no es un problema, pero a veces queremos que no se repitan, por ejemplo, si queremos saber los Títulos diferentes que hay en la tabla Ejemplares", entonces utilizaremos DISTINCT.

```
SELECT DISTINCT Eje_Titulo FROM Ejemplares
```

Eje_Titulo
Base de datos Organizacionales
Aplicaciones WEB Empresariales
Aplicaciones Web con JSF e Hibernate
Minería de Datos – Esquemas Expertos

Tabla 36: Resultado Ejemplo con DISTINCT.

II.4.7. SQL TOP

La sentencia SQL TOP se utiliza para especificar el número de filas a mostrar en el resultado.

Esta cláusula SQL TOP es útil en tablas con muchos registros, para limitar el número de filas a mostrar en la consulta, y así sea más rápida la consulta, consumiendo también menos recursos en el sistema.

Esta cláusula se especifica de forma diferente según el sistema de bases de datos utilizado.

Cláusula SQL TOP para SQL SERVER

```
SELECT TOP número nombre_columna
FROM nombre_tabla
```

Cláusula SQL TOP para MySQL

```
SELECT columna(s) FROM tabla
LIMIT númerofilas
```

Cláusula SQL TOP para ORACLE

```
SELECT columna(s) FROM tabla
WHERE ROWNUM <= númerofilas
```

Ejemplo SQL TOP para SQL Server:

Dada la siguiente tabla 'Autores', quiero obtener los 2 primeros registros ingresados.

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA
555555	Albert Rood McMillan	EUA

Tabla 37: Resultado de Autores sin TOP.

```
SELECT TOP 2 * FROM Autores
```

Obtendríamos lo siguiente:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER

Tabla 38: Resultado de Autores con TOP.

II.5.Ejercicios propuestos

1. El promedio de Precios de venta de los Ejemplares.
2. El Precio más alto de los ejemplares.
3. El Precio más bajo de los ejemplares.
4. Ejemplares con Stock actualmente.
5. Valor total de los Ejemplares en Stock.
6. Monto total vendido en el año 2018 y la cantidad de Pedidos.
7. Cantidad de pedidos generados y monto total vendido al Cliente con código 32658790.
8. Modificar la tabla ejemplares incluyendo un campo de tipo entero para considerar si esta descontinuado o no.
9. Crear una tabla con los ejemplares descontinuados y luego insertar los registros. Los ejemplares descontinuados son: Eje_Id nchar(11), Eje_titulo nvarchar(50), Eje_Precio Numeric(9,2), Eje_Stock Numeric(8,2).
10. Utilizar un select con la misma cantidad de campos para inserta los registros.
11. Para visualizar los registros insertados.
12. Importante: note que el tipo de datos en la tabla Ejemplares-Descontinuados para el campo Código es nchar(11) y la tabla Ejemplares originalmente el tipo es numeric(11).

Capítulo III

SENTENCIAS SQL AVANZADAS

III.1. Consulta de Varias Tablas

JOIN

La sentencia SQL JOIN permite consultar datos de 2 o más tablas.

Dichas tablas estarán relacionadas entre ellas de alguna forma, a través de alguna de sus columnas.

Existen 3 tipos de JOINS: JOIN interno, JOIN externo y JOIN cruzado.

Una clave primaria es una columna con un valor único para cada registro de una tabla.

El propósito del JOIN es unir información de diferentes tablas, para no tener que repetir datos en diferentes tablas.

Ejemplo:

Si tenemos las siguientes tablas:

Tabla Ejemplares, con la clave primaria "EJE_Id".

Eje_Id	Edi _Id	Eje_Titulo	Eje_ Cantidad	Aut_Id	Tip_Id
121212	1	Base de datos Organizacionales	10	111111	1
131313	2	Aplicaciones Web con JSF e Hibernate	15	222222	2
141414	4	Minería de Datos – Esquemas Expertos		333333	10

Tabla 39: Uso JOIN - Tabla Ejemplares

Tabla Autores, con la clave primaria "AUT_Id".

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA

Tabla 40: Uso JOIN - Tabla Autores

Tabla Editoriales, con la clave primaria “EDI_Id”.

Edi_Id	Edi_Nombre	Edi_Ciudad
1	ULADECH Católica	Chimbote
2	Rio Santa Editores	Chimbote
3	Macro Editores	Lima

Tabla 41: Uso JOIN - Tabla Editoriales.

Si queremos saber los Títulos de los Ejemplares con sus respectivos nombres de autores, tendríamos que hacer un JOIN de las 2 tablas "Ejemplares" y "Autores", que se relacionarían por la columna "AUT_Id".

Es decir, que desde la tabla "Ejemplares" y mediante la columna "AUT_Id", podemos acceder a la información de la tabla "Autores".

INNER JOIN

La sentencia INNER JOIN es la sentencia JOIN por defecto, y consiste en combinar cada fila de una tabla con cada fila de la otra tabla, seleccionado aquellas filas que cumplan una determinada condición.

```
SELECT * FROM tabla1 AS t1 INNER JOIN tabla2
AS t2 WHERE t1.columna1 = t2.columna1
```

Ejemplo SQL INNER JOIN

```
SELECT Eje_Titulo, Eje_Cantidad, AUT_Nombres
FROM Ejemplares as E INNER JOIN Autores as A
WHERE A.AUT_Id = E.AUT_Id
```

Muestra el resultado siguiente:

Eje_Titulo	Eje_Cantidad	Aut_Nombres
Base de datos Organizacionales	10	Adamaris Tapia Rosales
Aplicaciones Web con JSF e Hibernate	15	Anahí Tapia Rosales
Minería de Datos – Esquemas Expertos		Luz Jacinto Heredia

Tabla 42: Resultado INNER JOIN

LEFT JOIN

La sentencia LEFT JOIN combina los valores de la primera tabla con los valores de la segunda tabla. Siempre devolverá las filas de la primera tabla, incluso aunque no cumplan la condición.

```
SELECT * FROM tabla1 as t1 LEFT JOIN tabla2 as t2
WHERE t1.columna1 = t2.columna1
```

Ejemplo de SQL LEFT JOIN

```
SELECT EJE_Titulo, EJE_Cantidad, EDI_Nombre
FROM Ejemplares As E LEFT JOIN Editoriales as ED
WHERE ED.EDI_Id = E.EDI_Id
```

Aunque la editorial '4' de 'Minería de Datos – Esquemas Expertos' no existe en la tabla de Editoriales, devolverá la fila con esa columna 'EDI_Nombre' en blanco.

III.2. Gestión de Datos con Clave Primaria y Clave Foránea

En muchas ocasiones, especialmente en procesos ETL o de carga de datos para un data warehouse, por ejemplo, interesa hacer en una sola sentencia o en un solo paso la comprobación de si un registro existe, y si existe actualizarlo, y si no insertarlo. A esta combinación se le ha apodado UPSERT, aunque en SQL existe una sentencia específica para hacerlo, que es MERGE.

INSTRUCCIÓN MERGE

Realiza instrucciones de inserción de registros, actualización o eliminación de registros en una tabla de destino en la misma base de datos o en otra base de datos, según los resultados de combinar los registros con una tabla de origen. (4)

Sintaxis

La forma de usar Merge es la siguiente:

```
MERGE
[ TOP ( n ) [ PERCENT ] ]
[ INTO ] <Tabla_Destino> [ [ AS ] AliasTablaDestino ]
USING <Tabla_Origen> [ [ As ] AliasTablaOrigen]
ON <CondiciónMergeComparación>
[ WHEN MATCHED [ AND <Condición> ]
THEN <Isntrucción Si encuentra> ] [ …n ]
[ WHEN NOT MATCHED [ BY TARGET ] [ AND <Condición> ]
THEN <Instrucción Si NO Encuentra en Destino> ]
[ WHEN NOT MATCHED BY SOURCE [ AND <Condición> ]
THEN <Instrucción Si NO Encuentra en Origen> ] [ …n ]
```

Ejemplo

En este ejemplo se tienen dos bases de datos cada una con una tabla de Productos.

La base de datos Antiguos, con la tabla ProductosAntiguos y la base de datos Nuevos con la tabla ProductosNuevos.

BASE DE DATOS ANTIGUOS

```
CREATE DATABASE Antiguos
go
USE Antiguos
go
CREATE TABLE ProductosAntiguos
(
Pro_Id nchar(4), Pro_Descripcion nvarchar(100),
Pro_PrecioUnitario Numeric(10,2),
Pro_StockActual Numeric(10,2),
constraint ProductosPk Primary key (Pro_Id)
)
go
```

Insertar registros en la tabla ProductosAntiguos

```
INSERT INTO ProductosAntiguos VALUES
('8856','Lámpara Personal',25.4,100)
INSERT INTO ProductosAntiguos VALUES
('8636','Auriculares Deluxe',98.4,20)
INSERT INTO ProductosAntiguos VALUES
('4685','Escritorio Gerencial',525,6)
INSERT INTO ProductosAntiguos VALUES
('5780','Marco Foto',20,80)
INSERT INTO ProductosAntiguos VALUES
('0665','Impresora HP',65,15)
go
```

BASE DE DATOS NUEVOS

```
CREATE DATABASE Nuevos
go
USE Nuevos
go
CREATE TABLE ProductosNuevos
(
Pro_Id nchar(4),
Pro_Descripcion nvarchar(100),
Pro_PrecioUnitario Numeric(10,2),
Pro_StockActual Numeric(10,2),
constraint ProductosPk Primary key (Pro_Id)
)
go
```

Insertar los registros en la tabla ProductosNuevos

Note los cambios, se han insertado registros y las coincidencias o diferencias se notan en la siguientes figuras.

- Primer registro “Lámpara Personal” con un valor del Stock de 80
- Segundo registro “Auriculares Deluxe” con precio de 115
- Tercer y cuarto registros nuevos
- Quinto registro “Impresora HP” tiene la descripción cambiada y nuevo Stock de 15 a 25

```
INSERT INTO ProductosNuevos VALUES
('8856','Lámpara Personal',25.4,80)
INSERT INTO ProductosNuevos VALUES
('8636','Auriculares Deluxe',115,20)
INSERT INTO ProductosNuevos VALUES ('9879','Switch
Ethernet 993',85,3)
INSERT INTO ProductosNuevos VALUES
('4567','Memoria USB 16 GB',50,10)
INSERT INTO ProductosNuevos VALUES
('0665','Impresora HP Multifuncional',65,25)
go
```

Antes del Merge

```
SELECT * FROM Antiguos.dbo.ProductosAntiguos
SELECT * FROM Nuevos.dbo.ProductosNuevos
go
```

Haciendo el Merge

La tabla Origen es Productos de la base de datos Nuevos y la tabla Destino en Productos en la base de datos Antiguos.

```
MERGE INTO Antiguos.dbo.ProductosAntiguos as Tabla-
Destino
USING Nuevos.dbo.ProductosNuevos as TablaOrigen
on (TablaDestino.Pro_Id = TablaOrigen.Pro_Id)
WHEN not matched THEN
INSERT VALUES (TablaOrigen.Pro_Id, TablaOrigen.-
Pro_Descripcion,
TablaOrigen.Pro_PrecioUnitario, TablaOrigen.Pro_S-
tockActual)
WHEN matched THEN
UPDATE SET Pro_Descripcion =
TablaOrigen.Pro_Descripcion,
Pro_PrecioUnitario = TablaOrigen.Pro_PrecioUnitario,
Pro_StockActual = TablaOrigen.Pro_StockActual;
go
```

Visualizar los resultados

En la tabla destino ProductosAntiguos se han insertados dos registros.

```
SELECT * FROM Antiguos.dbo.ProductosAntiguos
```

En la tabla origen de la base de datos Nuevos los registros son los mismos.

```
SELECT * FROM Nuevos.dbo.ProductosNuevos
```

III.3. Sentencias Complejas INSERT, UPDATE y DELETE

Insertando datos desde otras tablas

En el lenguaje SQL podemos disponer de múltiples opciones para insertar datos no solo la tradicional que comúnmente conocemos, aquí les muestro algunas de ellas.

Insertando datos en variables o tablas temporales

También se puede crear tablas temporales (MySQL) o variables tipo tabla (SQL Server, Oracle) para poder insertar datos y gestionarlos por la necesidad del usuario. Estas variables o tablas están visibles solo cuando se crea la conexión, eso significa que los datos no se comparten entre sesiones y se eliminan al final de la misma.

Ejemplo con MySQL

```
CREATE TEMPORARY TABLE miTabla (myId int,
miCampo varchar(100));
INSERT INTO miTabla SELECT tblid, tblCampo FROM
Tabla1;
SELECT * FROM miTabla;
```

Ejemplo con SQL Server

```
DECLARE @miTabla TABLE(myId int NOT NULL,
miCampo varchar(100) NOT NULL);
INSERT INTO @miTabla SELECT tblid, tblCampo FROM
Tabla1;
SELECT * FROM @miTabla;
```

Ejemplo con Oracle

```
CREATE GLOBAL TEMPORARY TABLE miTabla (myId
int NOT NULL, miCampo varchar(100)) {ON COMMIT
DELETE ROWS | ON COMMIT PRESERVE ROWS};
INSERT INTO miTabla SELECT tblid, tblCampo FROM
Tabla1;
SELECT * FROM miTabla;
```

Con la opción ON COMMIT DELETE ROWS se borran los datos cada vez que se hace COMMIT en la sesión.

Con la opcion ON PRESERVE DELETE ROWS los datos no se borran hasta el final de la sesión.

Actualizando Datos desde un Select

Se puede hacer de la siguiente manera. Fíjate que la consulta iguala las dos tablas por el campo clave que las dos tengan en común.

```
UPDATE Tabla_A SET Tabla_A.col1 = Tabla_B.col1,
Tabla_A.col2 = Tabla_B.col2
FROM alguna_tabla AS Tabla_A
 INNER JOIN otra_tabla AS Tabla_B
 ON Tabla_A.id = Tabla_B.id
WHERE
 Tabla_A.col3 = 'ok'
```

En el caso en que la tabla destino no tenga esos campos vacíos, y pueda haber datos que ya existan desde el origen, conviene utilizar una consulta que verifique antes de hacer el UPDATE, ya que un proceso de UPDATE hace primero un DELETE y luego un INSERT

Se podría hacer algo como lo siguiente:

```
UPDATE Tabla
SET Tabla.col1 = otra_tabla.col1,
 Tabla.col2 = otra_tabla.col2
FROM Tabla
INNER JOIN otra_tabla
ON Tabla.id = otra_tabla.id
WHERE EXISTS(SELECT Tabla.Col1, Tabla.Col2
EXCEPT SELECT otra_tabla.Col1, otra_tabla.Col2))
```

Eliminando datos de la tabla desde una sentencia SELECT usando JOIN en Múltiples Tablas

Ahora presta atención al siguiente diagrama. Aquí tenemos dos tablas Tabla 1 y Tabla 2. Nuestro requisito es que queremos eliminar esos dos registros de la Tabla 1 donde los valores de la Tabla 2 Col3 son "Dos-Tres" y "Dos-Cuatro" y la Col1 en ambas tablas son iguales.

He explicado la declaración anterior muy fácilmente en el siguiente diagrama.

	Col1	Col2	Col3
1	1	11	Primero
2	11	12	Segundo
3	21	13	Tercero
4	31	14	Cuarto

Eliminar

	Col1	Col2	Col3
1	1	21	Dos-uno
2	11	22	Dos-dos
3	21	23	Dos-tres
4	31	24	Dos-cuatro

Tabla 43: Resultado de Actualizar datos con SELECT.

Cuando nos topamos con esto, parece muy simple, pero cuando intentamos pensar en la solución he visto a los desarrolladores idear muchas soluciones diferentes. Por ejemplo, en algún momento escriben cursor, variables de tabla, variables locales, etc. Sin embargo, la más fácil y la más limpia. La forma es usar la cláusula JOIN en la sentencia DELETE y usar varias tablas en la sentencia DELETE y hacer la tarea.

```
DELETE Table1 FROM Table1 t1
INNER JOIN Table2 t2 ON t1.Col1 = t2.Col1
WHERE t2.Col3 IN ('Dos-tres','Dos-cuatro')
```

III.4. Importación y Exportación de Datos

En algunas oportunidades nos topamos con el problema de importar y exportar datos en diferentes formatos como txt, cvs, dbf o xls. Es por ello que algunos SGBD nos brindan la posibilidad de manejar archivos de diferentes formatos para poder recuperar o almacenar datos.

BULK INSERT

Carga datos de un archivo de datos a una tabla. Esta funcionalidad es parecida a la que ofrece la opción in del comando BCP, aunque el que lee el archivo de datos es el proceso de SQL Server.

Sintaxis

```
BULK INSERT tabla_insertar FROM 'Ruta_Archivo_nombre'
WITH (FIELDTERMINATOR= 'carácter_separador_columna',
ROWTERMINATOR = 'salto_linea');
```

Ejemplo

Vamos a importar datos desde un texto plano (TXT sin cabecera) a la tabla Autores conociendo que tiene la siguiente estructura:

Aut_Id	Aut_Nombres	Aut_Nacionalidad

El siguiente archivo txt con el siguiente formato:

```
555555, Juana de Arco, FRA
666666, Ozil Von Fosh, GER
777777, Marco VanDersart, NRL
888888, María La Madrid, ESP
```

Tenemos la clara intención de copiar la información desde el archivo TXT hacia la tabla en cuestión, vamos a utilizar entonces el siguiente comando

```
BULK INSERT Autores FROM 'D:\archivo.txt' WITH
(FIELDTERMINATOR= ',',ROWTERMINATOR = '\n');
```

Donde hemos especificado origen (el path al archivo TXT) destino (la tabla Autores) y los separadores de filas y columnas en los argumentos ROWTERMINATOR y FIELDTERMINATOR.

El resultado de la operación generará el siguiente resultado:

Aut_Id	Aut_Nombres	Aut_Nacionalidad
111111	Adamaris Tapia Rosales	PER
222222	Anahí Tapia Rosales	PER
333333	Luz Jacinto Heredia	GER
444444	Juan Peres Pai	BRA
555555	Juana de Arco	FRA
666666	Ozil Von Fosh	GER
777777	Marco VanDersart	NRL
888888	María La Madrid	ESP

Tabla 44: Resultado luego de Insertar datos con BULK INSERT.

Si nuestro SGBD es MYSQL podemos utilizar LOAD DATA pues sirve para tomar cualquier archivo “comma-separated” (separado por comas, aunque no necesariamente son comas) y cargarlo como datos en alguna tabla de mySQL; la sintaxis básica es:

```
LOAD DATA LOCAL INFILE 'D:/archivo.txt'
INTO TABLE Autores
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';
```

EXPORTAR UN QUERY A UN ARCHIVO .TXT

Ahora se necesitaba exportar el resultado de un query a un archivo de texto plano, para ello debes seguir la siguiente secuencia.

1. Debes ejecutar lo siguiente

```
EXEC master.dbo.sp_configure 'show advanced options', 1
RECONFIGURE
EXEC master.dbo.sp_configure 'xp_cmdshell', 1
RECONFIGURE
```

Esto es para asegurarnos que el xp_cmdshell está habilitado (también pueden habilitarlo desde la configuración de superficie).

2. Se debe ejecutar lo siguiente para que el comando haga lo que tenga que hacer

```
EXEC xp_cmdshell 'bcp "SELECT * FROM Autores"
queryout "D:\Autores.txt" -T -c -t,'
```

Aqui tenemos que le pasamos el query tal cual lo necesitamos, y le pasamos la ruta donde queremos que cree el archivo de texto (recuerden que la ruta es local, así que si están conectados a un servidor tendrán que ir por el archivo de texto a esa ruta en ese servidor)

El caso es que el archivo de texto es creado y tendrán un problema menos que resolver.

Los parámetros que se están utilizando son:
Queryout – es la que permite espeficar el query con el que se trabajará.
File name – donde se insertará el resultado (debe ser la ruta completa).
-T, que especifica que la utilidad bcp se conectara a SQL Server con una conexión segura, se pude usar -P (contraseña) y -U (usuario).
-c, especifica el tipo de caracteres que se usará para cada campo.

-t, permite especificar el delimitador de campo, el caracter que se especifique después del -t será el que separe cada campo.

-S, se pude usar para especificar el nombre del servidor. Si se tiene una Instancia nombrada, será forzoso usar este parámetro.

III.5. Ejercicios propuestos

1. Modificar la tabla usuarios añadiendo un campo para llevar el control de las veces que se accede a la base de datos a través de una aplicación este campo será de tipo INT e inicializa en 0.
2. Crear una tabla para los Usuarios Actualizados, con la misma estructura de la tabla Usuarios y como resultado nos va a dejar las dos tablas sincronizadas, tabla origen y tabla fuente. En un caso normal esto se tendría que hacer los 3 querys por separado, lo que significaría varias consultas y un select para verificar si el dato existe, otro para insertar otro para modificar y otro para borrar. Utilizar MERGE, que nos permite hacer todo esto en una sola consulta, lo que es mucho más eficiente y utiliza muchísimo menos recursos en el servidor, más aún cuando las tablas son muy grandes.
3. Exportar los datos de la tabla usuarios en un texto plano sin cabecera y manteniendo la estructura de la tabla migrada.
4. Cargar datos desde un archivo de texto plano con los datos de los usuarios actualizados modificando las cantidades de los accesos.
5. También puedes utilizar otra opción de insertar los datos a la tabla usuarios actualizados a través de una consulta SELECT.
6. Eliminar los datos de los usuarios que no han sido actualizados usando JOINs.

Capítulo IV

CONTROL DE DATOS CON LÓGICA DE USUARIO

IV.1. Procedimientos Almacenados

Un procedimiento almacenado (stored procedure en inglés) es un programa (o procedimiento) almacenado físicamente en una base de datos. Su implementación varía de un gestor de bases de datos a otro. Los procedimientos pueden ser ventajosos: cuando una base de datos es manipulada desde muchos programas externos. (5)

Características:

- Tienden a mejorar el rendimiento de los sistemas debido a que reducen el intercambio entre cliente y servidor.
- Los procedimientos almacenados son reutilizables, de manera que los usuarios mediante la aplicación cliente no necesitan relanzar los comandos individuales, sino que pueden llamar el procedimiento para ejecutarlo en el servidor tantas veces como sea necesario.

Utilidades:

- Por ejemplo, si deseamos obtener un reporte complejo que incluya instrucciones condicionales y cálculos complejos con datos obtenidos de varias tablas, un procedimiento almacenado es nuestro mejor aliado. También se pueden ejecutar complejos procesos que a veces tardan horas cuando son ejecutados desde el cliente, ya que en tales casos la información debe pasar del servidor al cliente y viceversa.

Elementos de los Procedimientos almacenados

Los procedimientos almacenados están compuestos por algunos de estos elementos:

- Parámetros de entrada (pueden esperar parámetros).
- Parámetros de salida (pueden devolver resultados).
- Declaración de variables (puede usarse variables en su cuerpo).
- Cuerpo del procedimiento (en su cuerpo se indican las acciones a realizar).

Sintaxis

Para crear un procedimiento almacenado debemos emplear la sentencia CREATE PROCEDURE.

```
CREATE PROCEDURE <nombre_procedimiento>
[@param1 <tipo>, ...]
 AS
 -- Sentencias del procedimiento
```

Para modificar un procedimiento almacenado debemos emplear la sentencia ALTER PROCEDURE.

```
ALTER PROCEDURE <nombre_procedimiento> [@pa-
ram1 <tipo>, ...]
 AS
 -- Sentencias del procedimiento
```

El siguiente ejemplo muestra un procedimiento almacenado, llamado **sp_LstAutores** que lista Autores, o sea, muestra todos los registros de la tabla 'Autores':

```
CREATE PROCEDURE sp_LstAutores
 AS
BEGIN
SET NOCOUNT ON;
SELECT * FROM Autores;
END
```

Nota: SET NOCOUNT Activar esta opción (el ON del final es lo que la activa) equivale a decirle al servidor de base de datos que no queremos que nos devuelva le número de filas afectadas por las instrucciones ejecutadas.

Luego para ejecutar el procedimiento debemos utilizar el comando EXECUTE o EXEC en el caso del SQL Server.

```
EXECUTE sp_LstAutores;
```

Si deseamos utilizar un parámetro de entrada en el procedimiento debemos realizar lo siguiente:

```
CREATE PROCEDURE sp_LstAutores_x_Nacionalidad
@Nacionalidad CHAR(3)
 AS
BEGIN
SET NOCOUNT ON;
SELECT * FROM Autores WHERE
Nacionalidad=@Nacionalidad;
END
```

En el caso que nuestro parámetro a mostrar fuese de salida debemos de definirlo con OUT

```
CREATE PROCEDURE
sp_LstAutores_x_Nacionalidad_NroAutores
@Nacionalidad CHAR(3),@NroAutores INT OUTPUT
 AS
BEGIN
SET NOCOUNT ON;
SELECT @NroAutores=COUNT(*) FROM Autores
WHERE Nacionalidad=@Nacionalidad
RETURN;
END
```

En este caso para mostrar el dato a devolver lo llamamos de la siguiente manera:

```
DECLARE @nro INT;
EXEC sp_LstAutores_x_Nacionalidad_NroAutores 'PER',
@nro OUTPUT
SELECT @nro;
```

Podemos apreciar que para utilizar una variable declarada dentro del SGBD SQL Server utilizaremos el comando DECLARE hacien-

do uso del carácter @ y luego el tipo de dato y para asignar el valor fijo a la variable utilizamos SET como mostramos a continuación:

```
DECLARE @Valor1 INT;
SET @Varlor1 = 3; //Valor Estático
```

Ahora bien el mismo ejemplo de procedimiento podemos utilizar para el SGBD MySQl y Oracle

MySQL

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
`sp_lstAutores`()
BEGIN
    SELECT * FROM Autores;
END
```

Oracle

```
CREATE OR REPLACE PROCEDURE
sp_LstAutores_x_Nacionalidad
(
P_Nacionalidad IN AUTORES%TYPE,
P_Autores_lst OUT SYS_REFCURSOR
)
 IS
BEGIN
OPEN P_Autores_lst for SELECT Aut_Id,
Aut_Nombres, Aut_Nacionalidad FROM Autores WHERE
Aut_Nacionalidad=P_Nacionalidad;
END;
```

IV.2. Mantenimiento de Tablas con Procedimientos Almacenados

Como hemos visto en el punto anterior de creación y uso de procedimientos almacenados, se puede utilizar las sentencias SQL aprendidas anteriormente. En esta oportunidad la utilizaremos para el mantenimiento a las tablas de nuestra base de datos.

Procedimiento para Insertar Datos

```
CREATE PROCEDURE sp_InsAutor @Id INT,
@Nombres VARCHAR(50), @Nacionalidad CHAR(3)
 AS
BEGIN
SET NOCOUNT ON;
INSERT INTO Autores VALUES (@Id, @Nombres,
@Nacionalidad);
END
```

Procedimiento para Actualizar Datos

```
CREATE PROCEDURE sp_UpdAutor @Id INT,
@Nombres VARCHAR(50), @Nacionalidad CHAR(3)
 AS
BEGIN
SET NOCOUNT ON;
UPDATE Autores SET Aut_Nombres = @Nombres,
Aut_Nacionalidad = @Nacionalidad WHERE Aut_Id = @Id;
END
```

Procedimiento para Eliminar Datos

```
CREATE PROCEDURE sp_DelAutor @Id INT
 AS
BEGIN
SET NOCOUNT ON;
DELETE FROM Autores WHERE Aut_Id = @Id;
END
```

Como se puede apreciar las sentencias que habíamos estudiado antes podemos utilizarlas bajo la misma sintaxis en nuestro procedimiento almacenado. Esto sumado a las ventajas que estos ofrecen aseguramos el óptimo rendimiento de nuestro gestor de base de datos.

Ahora como es costumbre también mostraré ejemplos de gestión de datos con el SGBD MySQL y Oracle y sus diferentes variantes:

MySQL

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
sp_InsAutor (_Id INT, _Nombres VARCHAR(50),
_Nacionalidad CHAR(3)
 AS
BEGIN
SET NOCOUNT ON;
INSERT INTO Autores VALUES (_Id, _Nombres,
_Nacionalidad);
END
```

Oracle

```
CREATE OR REPLACE PROCEDURE sp_InsAutor
(
P_Id AUTORES.Aut_Id%TYPE,
P_Nombres AUTORES.Aut_Nombres%TYPE,
P_Nacionalidad AUTORES.Aut_Nacionalidad%TYPE
)
 IS
BEGIN
INSERT INTO Autores (P_Id, P_Nombres, P_Nacionalidad);
COMMIT;
END;
```

IV.3. Constructores de control de flujo

Ningún lenguaje puede estar completo sin la habilidad de saltar a otras partes del código o realizar varias veces una tarea.

SQL Server permite el control del flujo mediante un pequeño conjunto de instrucciones:

IF… ELSE

Nos permite ejecutar instrucciones condicionales.

```
IF <Expresion_Logica>
 <Instruccion>
ELSE
 <Instruccion>
```

Podemos utilizar el siguiente ejemplo para utilizarlo

```
DECLARE @TotalEjemplares INT
SELECT @TotalEjemplares = COUNT(*) FROM Ejemplares
IF @TotalProductos > 50
 PRINT 'Existen Mas De 50 Ejemplares Registrados'
ELSE
 PRINT 'Existen Menos De 50 Ejemplares Registrados'
```

WHILE

La sentencia WHILE/DO es un comando que repite el comando que se encuentra en su interior mientras la condición del WHILE es cierta.

```
WHILE <Expresion_Logica>
BEGIN
 <Grupo_Sentencia>
END
```

En el siguiente ejemplo mostramos su funcionamiento

```
DECLARE @Contador INT
SET @Contador = 10
WHILE (@Contador > 0)
BEGIN
PRINT '@Contador = ' + CONVERT(NVARCHAR, @Contador)
SET @Contador = @Contador -1
END
```

CASE

La sentencia CASE compara un valor con una lista de valores y ejecuta una o más sentencias que corresponde al valor que se cumple. Y en caso de no cumplirse devolverá un valor por defecto.

```
CASE <expresion>
 WHEN <valor_expresion> THEN <valor_devuelto>
 WHEN <valor_expresion> THEN <valor_devuelto>
 ELSE <valor_devuelto>
END
```

Ejemplo

```
DECLARE @PAIS NVARCHAR(20)
SELECT @PAIS =
CASE Aut_Nacionalidad
 WHEN 'PER' THEN 'PERUANA'
 WHEN 'GER' THEN 'ALEMANA'
 WHEN 'BRA' THEN 'BRASILERA'
 ELSE 'No Existe Registro'
END FROM Autores;
PRINT @PAIS
```

RETURN

Es muy simple Le pone fin la instrucción que se ejecuta.

Ejemplo:

```
DECLARE @CONTADOR INT
SET @CONTADOR = 10
WHILE (@CONTADOR >0)
 BEGIN
 PRINT '@CONTADOR = ' + CON-
VERT(NVARCHAR,@CONTADOR)
 SET @CONTADOR = @CONTADOR -1
 IF (@CONTADOR = 5)
 RETURN
 END
PRINT 'FIN'
```

LOOP

Loop implementa un constructor de bucle simple que permite la ejecución repetida de comandos particulares.

Sintaxis

```
[begin_label:] LOOP
statment_list
END LOOP [end_label]
```

La ejecución del comando se repite hasta acabar el bucle, normalmente por el comando “Leave”.

Normalmente los comandos LOOP se etiquetan:

“end_label” no puede darse si no está presente “begin_label” , y si ambos lo están, deben ser el mismo.

Normalmente se usa conjuntamente con los constructores “Leave”, para abandonar el bucle en caso de cumplirse una condición, y “Iterate” para volver a hacer el bucle si cumple una condición.

LEAVE

Esta instrucción es utilizada para salir de alguna estructura de control. Puede ser usada dentro de un BEGIN ... END o dentro de algún ciclo (Bucles).

Sintaxis

```
label1: LOOP
SET p1 = p1 + 1;
IF p1 < 10 THEN
ITERATE label1;
END IF;
LEAVE label1; ---> Aquí rompemos un Bucle
END LOOP label1;
```

MANEJO DE EXCEPCIONES

En esta ocasión hablaremos acerca del manejo de excepciones en transacciones de SQL Server

Una mejora importante que tenemos en SQL Server es el manejo de errores que ahora es posible en T-SQL con los bloques TRY/CATCH sin olvidar la sintaxis que utilizamos para las transacciones.

Sintaxis:

```
BEGIN TRY
BEGIN TRANSACTION
— Bloque de código SQL a proteger
COMMIT TRANSACTION
END TRY
BEGIN CATCH
— Código para mostrar el mensaje de la excepción
ROLLBACK TRANSACTION
END CATCH
```

Si ocurre un error dentro de la transacción en un bloque TRY inmediatamente se dirige al bloque CATCH.

COMMIT

Es la finalización de nuestro bloque de código y todas las instrucciones dentro de él se llevarán a cabo siempre y cuando no se haya generado error alguno.

ROLLBACK

Sirve para deshacer todos los movimientos realizados dentro del bloque de la transacción del mismo nombre.

Ahora les mostraré un ejemplo, primero creamos una tabla de ejemplo en nuestro modelo:

```
IF OBJECT_ID( 'tblPruebas' ) IS NOT NULL
 DROP TABLE tblPruebas
CREATE TABLE tblPruebas (cve INT)
```

Ahora ejecutamos una serie de instrucciones con los bloques de código TRY, CATCH y COMMIT.

```
INSERT INTO tblPruebas VALUES( 1 )
BEGIN TRY
 INSERT INTO tblPruebas VALUES( 2 ) , ( 3 ) --b
 BEGIN TRAN nombreTransaccion
 INSERT INTO tblPruebas VALUES( 4 ) --c
 INSERT INTO tblPruebas VALUES( 5 ) , ( 'a' ) --d
 COMMIT TRAN nombreTransaccion
END TRY
BEGIN CATCH
 SELECT ERROR_NUMBER() AS errNumber
 , ERROR_SEVERITY() AS errSeverity
 , ERROR_STATE() AS errState
 , ERROR_PROCEDURE() AS errProcedure
 , ERROR_LINE() AS errLine
 , ERROR_MESSAGE() AS errMessage
 ROLLBACK TRAN nombreTransaccion
 SELECT * FROM tblPruebas
END CATCH
```

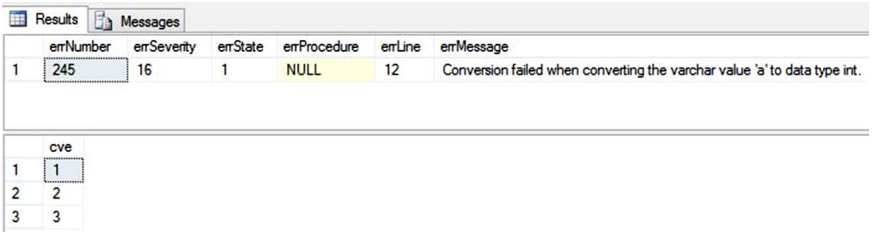


Figura 13: Resultado de Sentencia para manejo de Errores.

Como podemos observar, el INSERT (a) está fuera de ambos bloques, por lo tanto se ejecuta con normalidad; el (b) se encuentra dentro del bloque TRY pero fuera del bloque BEGIN TRAN… COMMIT TRAN, (c) y (d) se encuentran dentro de ambos bloques, solo que esté ultimo genera un error; entonces capturamos el error y se obtiene con las funciones de ERROR que solamente se pueden utilizar dentro del bloque CATCH y aplicamos un ROLLBACK TRAN, por lo tanto todo el bloque de instrucciones dentro de la transacción del mismo nombre, se revierten, es por ellos que el valor 4 no se inserta a la tabla.

Otro Ejemplo:

Usando la base de datos **miKioskoVirtual**, listar los registros de la tabla Ventas, tenga en cuenta que la tabla no existe.

El select se incluirá en un procedimiento almacenado.

Use miKioskoVirtual

El procedimiento almacenado

```
CREATE PROCEDURE spListaVentas
As
SELECT * FROM Ventas
```

Ahora ejecutamos el procedimiento almacenado.

```
BEGIN TRY
Execute spListaVentas
END TRY
BEGIN CATCH
SELECT ERROR_NUMBER() AS 'Nº Error',
ERROR_MESSAGE() AS 'Mensaje'
END CATCH
```

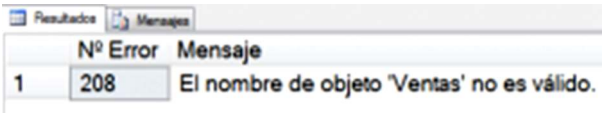


Figura 14: Mostrar mensaje con Manejo de Errores.

MANEJANDO ERRORES

Al usar el lenguage Transact-SQL debemos tener en cuenta, como en cualquier lenguaje de programación, que algunas instrucciones nos pueden dar errores debido a los valores de parámetros de entrada incorrectos o faltantes, ingresos de datos con tipos incorrectos, falta de datos en un procedimiento o función definida por el usuario o de manera general en una transacción no finalizada de manera correcta.

Podemos citar algunas funciones para el manejo de errores en el SQL Server, pues estas permiten conocer los parámetros que reporta un error.

Función	Descripción
ERROR_NUMBER()	Devuelve el número de error.
ERROR_SEVERITY()	Devuelve la severidad del error.
ERROR_STATE()	Devuelve el estado del error.
ERROR_PROCEDURE()	Devuelve el nombre del procedimiento almacenado que ha provocado el error
ERROR_LINE()	Devuelve el número de línea en el que se ha producido el error.
ERROR_MESSAGE()	Devuelve el mensaje de error.

Tabla 45: Funciones para el manejo de Errores

Ejemplo

El siguiente ejemplo muestra una división entre CERO, lo que arroja error, luego se muestran los valores de cada función. Se utiliza la estructura Try Catch cuya explicación está antes de este item.

```
BEGIN TRY
DECLARE @Valor1 Numeric(9,2),@Valor2 Numeric(9,2),
@Division Numeric(9,2)
SET @Valor1 = 100
SET @Valor2 = 0
SET @Division = @Valor1/@Valor2
PRINT 'La división no reporta error'
END TRY
BEGIN CATCH
SELECT ERROR_NUMBER() As 'Nº de Error',
ERROR_SEVERITY() As 'Severidad',
ERROR_STATE() As 'Estado',
ERROR_PROCEDURE() As 'Procedimiento',
ERROR_LINE() As 'Nº línea',
ERROR_MESSAGE() As 'Mensaje'
END CATCH
```

LA FUNCIÓN @@ERROR

La función @@ERROR almacena el número de error producido por la última sentencia Transact SQL ejecutada, si no se ha producido ningún error el valor de la función es CERO.

Se puede usar esta función para controlar los errores usando una estructura IF

Ejemplo:

El siguiente ejemplo muestra una división entre CERO, lo que arroja error, luego se da consistencia al error con una estructura IF

```
DECLARE @Valor1 Numeric(9,2),@Valor2 Numeric(9,2),
@Division Numeric(9,2)
SET @Valor1 = 100
SET @Valor2 = 0
SET @Division = @Valor1/@Valor2
IF @@ERROR = 0
Begin
Print 'El resultado es: ' + Str(@Division)
Print 'No hubo error'
End
ELSE
Begin
Print 'Error al dividir entre CERO'
End
```

Probamos el mismo código con el Valor2 igual a 2

```
DECLARE @Valor1 Numeric(9,2),@Valor2 Numeric(9,2),
@Division Numeric(9,2)
SET @Valor1 = 100
SET @Valor2 = 2
SET @Division = @Valor1/@Valor2
IF @@ERROR = 0
Begin
Print 'El resultado es: ' + Str(@Division)
Print 'No hubo error'
End
ELSE
Begin
Print 'Error al dividir entre CERO'
End
```

En la página de SQL podemos encontrar los diferentes niveles de severidad o gravedad del error que maneja ese gestor. Visitar https://docs.microsoft.com/es-es/sql/relational-databases/errors-events/database-engine-error-severities?view=sql-server-2017

PERSONALIZANDO MENSAJES DE ERROR EN EL GESTOR SQL SERVER

SQL Server tiene una vista de catálogo con los mensajes definidos por defecto, la vista es sys.messages, a la cual se le pueden añadir mensajes de error con sus parámetros respectivos usando el procedimiento almacenado sp_addmessage.

El procedimiento almacenado sp_addmessage permite agregar a la vista de catálogo sys.messages mensajes de error definidos por el usuario con niveles de gravedad de 1 a 25. Use Raiserror para utilizar los mensajes de error definidos por el usuario.

RAISERROR puede hacer referencia a un mensaje de error definidos por el usuario almacenados en la vista de catálogo sys.messages o puede generar un mensaje dinámicamente. Si se usa el mensaje de error definido por el usuario de sys.messages mientras se genera un error, la gravedad especificada por RAISERROR reemplazará a la gravedad especificada en sys.messages.

Para visualizar los mensajes de la vista de catálogo Sys.messages puede ejecutar la siguiente instrucción.

```
SELECT * FROM sys.messages
```

Resultados | Mensajes

	message_id	language_id	severity	is_event_logged	text
1	21	1033	20	0	Warning: Fatal error %d occurred at %S_DATE. Not...
2	101	1033	15	0	Query not allowed in Waitfor.
3	102	1033	15	0	Incorrect syntax near '%.*ls'.
4	103	1033	15	0	The %S_MSG that starts with '%.*ls' is too long. Max...
5	104	1033	15	0	ORDER BY items must appear in the select list if the...
6	105	1033	15	0	Unclosed quotation mark after the character string '...
7	106	1033	16	0	Too many table names in the query. The maximum a...
8	107	1033	15	0	The column prefix '%.*ls' does not match with a tabl...
9	108	1033	15	0	The ORDER BY position number %ld is out of range...
10	109	1033	15	0	There are more columns in the INSERT statement t...
11	110	1033	15	0	There are fewer columns in the INSERT statement t...

Figura 15: Lista de mensajes del Sistema.

El procedimiento sp_addmessage

Permite agregar un mensaje de error definido por el usuario a la vista de catálogo sys.messages.

Sintaxis:

```
sp_addmessage [ @msgnum= ] msg_id , [ @severity= ]
severidad , [ @msgtext= ] 'mensaje'
[ , [ @lang= ] 'languaje' ]
```

Donde:

[@msgnum=] msg_id Especifica el Id del mensaje, se pueden iniciar en 50001, el valor máximo es 2,147,483,647.

[@severity=] severidad Indica el nivel de gravedad del error, puede ser un valor entre 1 y 25.

[@msgtext=] 'mensaje' Especifica el mensaje definido por el usuario.

[@lang=] 'languaje' Especifica el lenguaje.

Ejemplos:

Agregar el mensaje para indicar que un porcentaje de descuento puede ser entre 0 y 25%.

Nota: Es necesario insertar el mensaje para el idioma inglés y así poder agregar el mensaje para español.

```
Use master
go
Execute sp_addmessage 50001, 16, 'The discount per-
centage should be between 0 and 25%', 'us_english',
false, replace;
Execute sp_addmessage 50001, 16, 'El porcentaje de
descuento debe ser entre 0 y 25%', 'Spanish', false,
replace;
```

Agregar el mensaje para indicar que un precio debe ser CERO o mayor.

```
Use master
go
Execute sp_addmessage 50002, 16,
'The price should be 0 or greater', 'us_english' , false,
replace
Execute sp_addmessage 50002, 16,
'El precio debe ser 0 o mayor', 'Spanish' , false, replace
go
```

Procedimientos almacenados complementarios para los mensajes de error definidos por el usuario

Procedimiento almacenado sp_altermessage

Modifica el estado de los mensajes definidos por el usuario o del sistema en una instancia del motor de base de datos de SQL Server.

Sintaxis:

```
sp_altermessage [ @message_id = ] message_number ,[
@parameter = ]'write_to_log'
,[ @parameter_value = ]'value'
```

Donde:

[@message_id =] message_number Especifica el Id del mensaje

[@parameter =] 'write_to_log' Especifica si se va a escribir en el log de Windows

[@parameter_value =] 'value' Se utiliza con @parameter para indicar que el error debe escribirse en el registro de aplicación de Microsoft Windows

Ejemplo

El siguiente ejemplo permite especificar que el mensaje de error creado se escriba en el log de Windows.

```
sp_altermessage 50001 , 'with_log' ,'true';
go
```

Procedimiento almacenado sp_dropmessage

Elimina un mensaje de error definido por el usuario

Sintaxis:

```
sp_dropmessage [ @msgnum = ] message_number [ , [
@lang = ] 'language' ]
```

Donde:

[@msgnum =] message_number Especifica el Id del mensaje

[@lang =] 'language' Especifica el lenguaje.

Ejemplo

El siguiente código elimina el mensaje creado con Id 50001

```
sp_dropmessage 50001, 'Spanish';
go
```

IV.4. Funciones definidas por el Usuario

Al igual que las funciones de los lenguajes de programación, las funciones definidas por el usuario (**FDU**) de SQL son rutinas que aceptan o no parámetros, realizan una acción, como un cálculo complejo, y devuelven el resultado de esa acción como un valor. Se pueden modificar y se pueden utilizar en cualquier lenguaje Transact-SQL; también estas pueden obtener resultados que las funciones propias del Sistema Gestor de Base de Datos no pueden mostrar.

Para crear una función al igual que un procedimiento tiene una sintaxis:

Las que retornan un valor:

```
CREATE FUNCTION NombreFuncion(Parámetros)
RETURNS TipoDato
AS
BEGIN
 Cuerpo
RETURN Expresión
END
```

Las que retornan una Tabla:

```
CREATE FUNCTION NombreFuncion (Parámetros)
RETURNS TABLE
AS
RETURN Instrucción Select
```

Ejemplos:

— Crear una función para retornar el total de Categorías

```
CREATE FUNCTION dbo.fCategoriasCuenta()
RETURNS INT
AS
BEGIN
DECLARE @CantidadCategorias INT
SELECT @CantidadCategorias=count(*) FROM categorías
RETURN @CantidadCategorias
END
```

Prueba:

```
SELECT 'Existen:' +
Ltrim(Str(dbo.fCategoriasCuenta()))+ ' categorías'
```

— Crear una función que reporte el total de Ejemplares en Stock de– cualquier categoría.

```
CREATE FUNCTION
dbo.fnStockxCategoria(@CodigoCategoria int)
RETURNS INT
AS
BEGIN
DECLARE @TotalStock INT

SELECT @TotalStock = SUM(Eje_Cantidad) FROM
Ejemplares WHERE Cat_Id = @CodigoCategoria
RETURN @TotalStock
END
```

Prueba:

— Categoria 1

```
Select 'Existe: '+Ltrim(Str(dbo.fnStockxCategoria(1))) + '
ejemplares'
```

— Implementar la función anterior, pero asignando el nombre de la categoría

— Cuantos ejemplares en stock hay de Libros

Debemos codificar lo siguiente:

```
DECLARE @CodigoCategoria INT
SELECT @CodigoCategoria=Cat_Id FROM
categorias WHERE Cat_Nombre='Libros'
SELECT 'Existe:
'+Ltrim(Str(dbo.fnStockxCategoria(@CodigoCategor
ia))) + ' Ejemplares'
```

— Cuantos Ejemplares hay de categoría Informes, podemos incluir la FDU en un Procedimiento (Ver procedimientos)

```
CREATE PROCEDURE spVerCantidadEjemplaresStock-
xCategoria ( @NombreCategoria nvarchar(50) )
AS
IF EXISTS (SELECT * FROM categorias WHERE
Cat_Nombre = @NombreCategoria)
BEGIN
DECLARE @IDCategoria INT
SELECT @IDCategoria = Cat_Id FROM Categorias
WHERE Cat_Nombre=@NombreCategoria
SELECT 'Existen : ' +
Ltrim(Str(dbo.fnStockxCategoria(@IDCategoria))) + '
Ejemplares'
END
ELSE
BEGIN
PRINT 'No existe la categoria ' + @NombreCategoria
END
```

—PruebaExec spVerCantidadArticulosStockxCategoria 'Mapas'

— Función que permita mostrar los clientes y la cantidad comprada de una determinada ciudad.

```
CREATE FUNCTION
dbo.fCantidadxClientexCiudad(@Ciudad nvarchar(20))
RETURNS Table
AS
RETURN SELECT C.Cli_Id As 'Código Cliente', CASE
C.Cli_Tipo WHEN 'N' THEN 'Natural' WHEN 'J' THEN
'Jurídico' END As 'Tipo Cliente', C.Cli_Ciudad,
SUM(DP.PED_Cantidad * DP.PED_PrecioVenta) As
Monto FROM Clientes As C
INNER JOIN Pedido As P
```

```
ON C.Cli_Id = P.Cli_Id
INNER JOIN DetallePedido As DP
ON P.Ped_Id=DP.Ped_Id
WHERE C.Cli_Ciudad= @Ciudad
GROUP BY C.Cli_Id, C.Cli_Tipo, C.Cli_Ciudad
```

— Ver los clientes de Lima

```
select * from dbo.fCantidadxClientexCiudad('Lima')
```

— Comprobando que el país exista

```
CREATE PROCEDURE spCantidadxClientesxCiudad
(@Ciudad nvarchar(20))
As
IF EXISTS(SELECT Cli_Ciudad FROM Clientes WHERE
Cli_Ciudad =@Ciudad)
BEGIN
 SELECT * FROM
dbo.fCantidadxClientexCiudad(@Ciudad)
END
ELSE
BEGIN
 SELECT 'No existe la ciudad especificada'
END
```

— Pruebas

```
Exec spCantidadxClientesxPais 'Lima'
```

IV.5. Desencadenadores (Triggers)

Un desencadenador o Trigger es una clase de procedimiento almacenado que se ejecuta automáticamente cuando se realiza una transacción en la base de datos. (6)

Tipos de Triggers

Existen los siguientes tipos de Triggers

- Los Triggers DML se ejecutan cuando se realizan operaciones de manipulación de datos (DML). Los eventos DML son instrucciones INSERT, UPDATE o DELETE realizados en una tabla o vista.
- Los Triggers DDL se ejecutan al realizar eventos de lenguaje de definición de datos (DDL). Estos eventos corresponden a instrucciones CREATE, ALTER y DROP.
- Los Triggers Logon, que se disparan al ejecutarse un inicio de sesión en SQL Server.

Consideraciones

- Una tabla puede tener un máximo de tres triggers: uno de actualización, uno de inserción y uno de eliminación.
- Cada trigger puede aplicarse a una sola tabla o vista. Por otro lado, un mismo trigger se puede aplicar a las tres acciones: UPDATE, INSERT y DELETE.
- No se puede crear un trigger en una vista ni en una tabla temporal, pero el trigger puede hacer referencia a estos objetos.
- Los trigger no se permiten en las tablas del sistema.

Las tablas Inserted y Deleted

Son tablas especiales que tienen la misma estructura de las tablas que han desencadenado la ejecución del trigger.

La tabla Inserted está disponible en las operaciones INSERT y UPDATE.

La tabla Deleted está disponible en las operaciones UPDATE y DELETE.

Note que para una operación de actualización, las dos tablas pueden ser utilizadas.

Para crear un Trigger DML se utiliza:

```
CREATE [ OR ALTER ] TRIGGER [Esquema].
NombreTrigger
ON { tabla | vista }
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
BEGIN
Instrucciones T-SQL
END
```

Para crear un Trigger DDL se utiliza:

```
CREATE [ OR ALTER ] TRIGGER NombreTrigger
ON { ALL SERVER | BaseDatos }
{ FOR | AFTER } {TipoEvento | GrupoEventos} [ ,…n ]
AS
BEGIN
Instrucciones T-SQL
END
```

Ejemplos

— **Usando miKioskoVirtual**

```
use miKioskoVirtual
```

1. Trigger que evita que se borren en la tabla categorías más de un registro, el Trigger detecta que se borra más de un registro y dispara un error. (Ver Manejo de errores)

```
CREATE TRIGGER trEliminaSoloUnaCategoria
ON Categorias FOR DELETE
AS
— Se cuentan cuantos registros se eliminaron
IF (Select COUNT(*) from Deleted)>1
```

```
BEGIN
 Raiserror('NO PUEDE ELIMINAR MÁS DE UN
REGISTRO',16,1)
 ROLLBACK Tran
END
```

Raiserror Raiserror('Mensaje',Severidad,Estado)

Como ya hemos visto el RaisError genera un mensaje de error e inicia el procesamiento de errores de la sesión. RAISERROR puede hacer referencia un mensaje definido por el usuario almacenado en la vista de catálogo sys.messages o puede generar un mensaje dinámicamente.

2. Visualizar los triggers de la base de datos

```
SELECT * FROM sys.triggers
go
```

3. Probar si el Trigger creado trEliminaSoloUnaCategoria funciona, se insertarán dos categorías con nombre Diccionarios y CDs

Insertar Categorías

```
INSERT INTO Categorias VALUES ('Diccionarios','Todo
tipo de Diccionarios'),
('CDs','CDs Iteractivos, de instalación, etc')
go
```

Eliminar una categoría

```
DELETE Categorias WHERE Cat_ID = 10
go
```

Se elimina sin inconvenientes porque es una sola categoría.

Al eliminar varias categorias el Trigger se dispara y no lo permite.

```
DELETE Categorias WHERE Cat_ID IN (12,13,15)
go
```

Resultado:

```
Mens. 50000, Nivel 16, Estado 1, Procedimiento
trEliminaSoloUnaCategoria, Línea 62
NO PUEDE ELIMINAR MÁS DE UN REGISTRO
Mens. 3609, Nivel 16, Estado 1, Línea 56
La transacción terminó en el desencadenador. Se anuló
el lote.
```

4. Borrar una sola categoría

```
delete Categories where CategoryID = 13
go
```

La categoría se borra con éxito

Trigger de Inserción

5. Crear un Trigger que permita comprobar que se inserta una categoría con nombre diferente. La tabla Inserted será utilizada para comprobar si ya hay una categoría con el mismo nombre que la insertada.

```
CREATE TRIGGER trCategoriaInsertaSinRepetidos ON
Categorias For Insert
AS
IF (SELECT COUNT (*) FROM Inserted, Categorias
WHERE Inserted.Cat_Nombre = Categorias.Cat_Nombre) >1
BEGIN
Rollback Transaction
PRINT 'El Nombre de la Categoría ya existe…'
END
ELSE
PRINT 'Categoría ingresada a la Base de datos'
go
```

6. Insertar categoría con nombre repetido: Diccionarios

```
INSERT INTO Categorias (Cat_Nombre,Cat_Descripcion)
VALUES ('Diccionarios','Todo tipo de Diccionarios')
go
```

Resultado:

```
El Nombre de la Categoría ya existe…
Mens. 3609, Nivel 16, Estado 1, Línea 87
La transacción terminó en el desencadenador. Se anuló el
lote.
```

Trigger Instead Of

Realizar acciones después de las instrucciones de un procedimiento o las escritas directamente por el usuario.

7. Copiar los datos a una vista de Clientes al insertar uno en la tabla Clientes

Primero se crea la vista con los clientes.

```
CREATE VIEW ClientesVista as
SELECT Cli_id,dbo. dbo.getNomCliente(cli_id),
cli_Direccion, Cli_Ciudad FROM Clientes;
go
```

Aquí estamos haciendo uso de una función que retorna el nombre del cliente según el Id enviado y tiene la siguiente estructura

```
CREATE FUNCTION [dbo].[getNomCliente]
(
@id numeric(11)
)
RETURNS varchar(60)
AS
BEGIN
```

```
        declare @tipo char(1);
        declare @clinom varchar(60);
        select @tipo=CLI_Tipo from Clientes where
 CLI_Id=@id;
        if @tipo='N'
            select @clinom=CLI_APaterno+'
'+CLI_AMaterno+' '+CLI_Nombres from CliNatural
where cli_id=@id;
        else
            select @clinom=CLI_RazonSocial from
CliJuridico where cli_id=@id;
 RETURN @clinom;

END
```

Al agregar un Cliente en Clientes se desea que se inserte en ClientesVista

Crear un Trigger Instead of para la tabla Clientes.

```
CREATE TRIGGER trClienteInsertaVista ON Clientes
Instead of Insert
AS
BEGIN
INSERT INTO ClientesVista
SELECT Cli_id, dbo.getNomCliente(cli_id),
cli_Direccion,Cli_Ciudad
FROM Inserted
Print 'Insertado correctamente en la vista'
END
go
```

Insertar un cliente en Clientes

Para insertar un cliente utilizaremos el siguiente procedimiento

```
CREATE PROCEDURE insCliente
@Id numeric(11),@tipo char(1),@direccion
varchar(50),@telefono varchar(15),
@ciudad varchar(20),@direccionent varchar(50),
@apaterno varchar(15),@amaterno
varchar(15),@nombres varchar(30),@sexo
char(1),@fnacimiento date,
@RazonSocial varchar(30),@contacto
varchar(30),@NroContacto varchar(15)
AS
BEGIN
      SET NOCOUNT ON;
      begin try
          INSERT INTO Clientes
values(@Id,@tipo,@direccion,@telefono,@ciudad,@
direccionent)
          BEGIN TRAN t1
          IF @tipo='N'
                  insert into CliNatural
values(@Id,@apaterno,@amaterno,@nombres,@sexo,
@fnacimiento);
          ELSE
                  insert into CliJuridico
values(@Id,@RazonSocial,@contacto,@NroContacto);
          COMMIT TRAN t1
      END TRY
      BEGIN CATCH
      SELECT ERROR_NUMBER() AS errNumber
 , ERROR_SEVERITY() AS errSeverity
 , ERROR_STATE() AS errState
 , ERROR_PROCEDURE() AS errProcedure
 , ERROR_LINE() AS errLine
 , ERROR_MESSAGE() AS errMessage
 ROLLBACK TRAN t1
       end catch
END
GO
```

Y lo ejecutamos

```
EXEC insCliente 12345678, 'N' , 'Av. Larco 8475',
'5467282', 'San Borja', ' Av. Larco 8475' , 'Perez', 'Perez',
'Juan', 'M', '02-02-1990', null, null, null;
go
```

Visualizar la vista para comprobar que el cliente ha sido insertado.

```
SELECT * FROM ClientesVista WHERE Cli_Id LIKE '123%'
go
```

Cómo crear triggers DDL en SQL Server

Ejemplos

Usando la base de datos miKioskoVirtual

1. Crear un Trigger para evitar que se creen, modifiquen o eliminen tablas

```
Create Trigger trNoCrearModificarBorrarTablas
ON DataBase FOR Create_Table, DROP_TABLE,
ALTER_TABLE
AS
BEGIN
RAISERROR ('Transacción anulada, no se permite crear,
editar o eliminar tablas', 16, 1)
Rollback transaction
END
go
```

Al intentar crear una tabla

```
CREATE TABLE Prueba
(id nchar(4), Descripcion nvarchar(100) )
go
```

Resultado:

```
Mens 50000, Nivel 16, Estado 1, Procedimiento trNoC-
rearModificarBorrarTablas, Línea 131
Transacción anulada, no se permite crear, editar o elimi-
nar tablas
Mens. 3609, Nivel 16, Estado 2, Línea 127
La transacción terminó en el desencadenador. Se anuló
el lote.
```

Para poder crear tablas, se debe eliminar el Trigger o solamente desactivar.

```
DISABLE TRIGGER trNoCrearModificarBorrarTablas on
Database
go
```

Probar ahora si se puede crear una tabla

```
CREATE TABLE Prueba
(id nchar(4), Descripcion nvarchar(100) )
go
```

Resultado:

```
Comandos completados correctamente.
```

Para activar nuevamente el Trigger se debe escribir la siguiente instrucción:

```
ENABLE TRIGGER trNoCrearModificarBorrarTablas on
Database
go
```

IV.6. Atención del caso práctico

1. Crear los procedimientos almacenados para la manipulación de datos de las tablas maestras Editorial, Categoría, Tipo Ejemplar y Autores (Insertar, Actualizar y Eliminar), incluyendo la validación de los datos a través de los triggers.
2. Crear procedimientos almacenados para la manipulación de datos de las tablas dependientes Pedido, Clientes (Natural y Jurídico), Detalle del Pedido, Usuarios y Ejemplares considerando la consistencia de los datos determinada por sus relaciones, validar la existencia previa del dato a registrar, así como considerar la manipulación de los mensajes de error.
3. Crear procedimientos almacenados para listar los datos haciendo uso de funciones para retornar datos como nombres de clientes, Título de ejemplares, nombres de autores, nombre de categoría, Nombre del tipo de ejemplar, asi como agrupaciones o datos ordenados según criterio. También se sugiere considerar el uso de funciones de agrupación para obtener total de ventas por parte de fecha, clientes top, ejemplares más vendidos, etc.
4. Crear triggers que controlen la cantidad en stock de los ejemplares cuando se realiza un pedido (variación de stock) o una devolución si fuese el caso, también considerar que los triggers no solo se ejecutan al disparar una acción específica (INSERT, UPDATE o DELETE), sino también permiten validar y dar consistencia a los datos que se están manipulado.
5. Crear funciones que permitan el incremento de algún Id de la tablas que no están habilitadas con la condición de autoincremento.

IV.7. Ejercicios propuestos

1. Usar Case para mostrar el nombre de la estación del año, las opciones posibles son Verano, Otoño, Invierno, Primavera.

2. Listado de los clientes y su estación en que nacieron. Para la estación se debe usar una función que devuelva ese dato.

3. Crear la tabla Matrícula con datos de los alumnos (IdAlumno), curso matriculado (IdCurso) y calificación, Fecha de Matricula, Fecha de Registro, Usuario de Registro, considerar que IdCurso así como IdAlumno son datos foráneos de la tabla Alumno y Cursos.

4. Ingresar algunos registros con la ayuda de un procedimiento almacenado, considerando que la fecha de registro así como el usuario que registra se deben capturar con las funciones del sistema getdate() current_user() a través de un trigger.

   Luego modificar la tabla añadiendo un campo para considerar la condición según la calificación registrada considerando el siguiente rango:

Rango calificación	Dato
Entre 0 y 10	Desaprobado
Entre 10 y 13	Aplazado
Entre 13 y 17	Aprobado
Entre 17 y 20	Aprobado con mérito

Tabla 46: Cuadro para Rango del Ejercicio Propuesto 4

   Este procedimiento deberá hacerla con la ayuda de la estructura case para el control de los rangos y la actualización de los datos en el campo de condición respectivo.

Luego crear un procedimiento para listar dichos datos con la siguiente estructura en respuesta al envío del parámetro idCurso:

Nro	Alumno	Calificación	Condición

Para el caso del Alumno deberá crear una función que devuelva el nombre del alumno.

Referencias Bibliográficas

1. Tierra de Lázaro. Tierra de Lázaro. [Online].; 2016 [cited 2018 12 27. Available from: http://www.tierradelazaro.com/wp-content/uploads/2016/12/DDL-DML-DCL-TCL.pdf

2. Universidad Autónoma del Estado de Hidalgo. Centro de Innovación para el Desarrollo y la Capacitación en Materiales Educativos. [Online].; 2017 [cited 2018 12 7. Available from: http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro14/.

3. Trainer SQL. Manuales SQL Server. [Online].; 2018 [cited 2018 12 27. Available from: http://www.manualsqlserver.com.

4. Microsoft Docs. MERGE (Transact-SQL) - SQL Server | Microsoft Docs. [Online].; 2018 [cited 2018 12 27. Available from: https://docs.microsoft.com/es-es/sql/t-sql/statements/merge-transact-sql.

5. Wikipedia. Wikipedia - Procedimiento almacenado. [Online].; 2018 [cited 2018 12 27. Available from: https://es.wikipedia.org/wiki/Procedimiento_almacenado.

6. Microsoft Docs. CREATE TRIGGER (Transact-SQL) - SQL Server | Microsoft Docs. [Online].; 2018 [cited 2018 12 27. Available from: https://docs.microsoft.com/es-es/sql/t-sql/statements/create-trigger-transact-sql.

7. EcuRED. SGBD. [Online].; 2018. Available from: https://www.ecured.cu/SGBD

8. Inc. E. erwin Academic Program. [Online].; 2018 [cited 2018 12 27. Available from: https://erwin.com/services/erwin-academic-program/.

9. Wikipedia. Wikipedia - Instituto Nacional Estadounidense de Estándares. [Online].; 2018 [cited 2018 12 27. Available from: https://es.wikipedia.org/wiki/Instituto_Nacional_Estadounidense_de_Est%C3%A1ndares.

10. Wikipedia. Wikipedia - ASCII. [Online].; 2018 [cited 2018 12 27. Available from: https://es.wikipedia.org/wiki/ASCII.

BASE DE DATOS II - Experiencias Prácticas
es una publicación del
Fondo Editorial de la Universidad Católica
Los Ángeles de Chimbote, Perú

Este libro recopila un sinnúmero de experiencias adquiridas durante la gestión de datos con el uso de diferentes SGBD, principalmente del SQL Server. Desde aquí se tratará de presentar y dar soluciones simples a la necesidad de administrar y gestionar datos con el lenguaje SQL dividido en sus 2 estructuras de gestión: Lenguaje de manipulación de datos (DML) y lenguaje de definición de datos (DDL).

Se parte de la atención de un caso práctico que hace evidente la necesidad de utilizar sentencias CREATE, ALTER, DROP basado en la estructura DDL y el uso de sentencias SELECT, INSERT, UPDATE Y DELETE para manipular datos, sin dejar de lados complementos de estas sentencias, así como operadores, funciones, controles de flujo y en el mejor de los casos manipulación y control a través de STORED PROCEDURE y TRIGGERS.

El autor cuenta con más de 13 años de experiencia en educación superior vinculada a la gestión de datos públicos y privados a través de diferentes gestores comerciales y no comerciales para brindar las soluciones en TIC de diferentes plataformas.

ISBN: 978-612-4308-22-2

FONDO EDITORIAL DE LA UNIVERSIDAD CATÓLICA
LOS ÁNGELES DE CHIMBOTE