Módulo 2 Desarrollo de bases de datos-DML

Índice

Integridad de los datos en SQL SERVER 3

Tipos de restricciones de integridad 3

Restricciones Default: 3

Restricciones Check 5

Deshabilitar y habilitar restricciones check 6

Reglas de restricciones Check: 6

Restricciones usando expresiones regulares 7

Restricción Primary Key 8

Restricciones UNIQUE 9

Laboratorio de trabajo independiente para trabajar con restricciones 10

Índices 13

Índice agrupado: 14

índice no agrupado: 14

Índice Columnar: 14

Limitaciones: 16

Reglas de índices columnares 16

¿Como crear índices? 17

Crear un índice agrupado 17

Crear un índice NO agrupado 17

Crear un índice Columnar 17

Laboratorio de trabajo independiente para trabajar con índices 18

Transact -SQL 19

Categorías Transact -SQL 19

DML 20

Elementos del lenguaje T-SQL: Predicados y operadores 20

Teoría de Conjuntos y SQL Server 21

Teoría de conjuntos aplicada a las consultas de SQL Server 21

Lógica de predicados aplicada a consultas de SQL Server 22

Declaración SELECT 22

Elementos de una declaración SELECT 23
Procesamiento de las consultas Lógicas 23
Cláusula SELECT 23
DISTINCT 24
Cláusula WHERE 24
TOP 25
GROUP BY 25
HAVING 26
ALIAS de Columna(campo) y ALIAS de Tabla 27
ALIAS de columna: 27
ALIAS de Tabla 28
SUBCONSULTAS 28
Laboratorio de trabajo independiente sentencia SELECT SQL SERVER 30
Sentencia INSERT 34
Sentencia UPDATE 34
Sentencia DELETE (cuidado con esta sentencia) 35
Sentencia TRUNCATE 35
Laboratorio para practicar las sentencias INSERT, UPDATE, DELETE, TRUNCATE. 36

Integridad de los datos en SQL SERVER

Es importante, al diseñar una base de datos y las tablas que contiene, tener en cuenta la integridad de los datos, esto significa que la información almacenada en las tablas debe ser válida, coherente y exacta.

Hasta el momento, hemos controlado y restringido la entrada de valores a un campo mediante el tipo de dato que le definimos (cadena, numéricos, etc.), la aceptación o no de valores nulos, el valor por defecto. También hemos asegurado que cada registro de una tabla sea único definiendo una clave primaria y empleando la propiedad identity.

Las restricciones (constraints) son un método para mantener la integridad de los datos, asegurando que los valores ingresados sean válidos y que las relaciones entre las tablas se mantengan. Se establecen en los campos y las tablas.

Pueden definirse al crear la tabla (create table) o agregarse a una tabla existente (empleando alter table) y se pueden aplicar a un campo o a varios. Se aconseja crear las tablas y luego agregar las restricciones.

Se pueden crear, modificar y eliminar las restricciones sin eliminar la tabla y volver a crearla.

El procedimiento almacenado del sistema **sp_helpconstraint** junto al nombre de la tabla, nos muestra información acerca de las restricciones de dicha tabla.

sp_helpconstraint tabla

Tipos de restricciones de integridad

Restricciones Default:

especifica un valor por defecto para un campo cuando no se inserta explícitamente en un comando **insert** Para establecer un valor por defecto para un campo se emplea la cláusula **default** al crear la tabla, por ejemplo:

```
create table Alumno (

  nombre varchar(50) default 'Desconocido',

)
```

Cada vez que se establece un valor por defecto para un campo de una tabla, SQL Server crea automáticamente una restricción default para ese campo de esa tabla.

Dicha restricción, recibe un nombre dado por SQL Server que consiste DF (por default), seguido del nombre de la tabla, el nombre del campo y letras y números aleatorios.

Podemos agregar una restricción **default** a una tabla existente con la sintaxis básica siguiente:

```
alter table NOMBRETABLA
add constraint NOMBRECONSTRAINT
default VALORPREDETERMINADO
for CAMPO;
```

En la sentencia siguiente agregamos una restricción **default** al campo nombre de la tabla existente Alumno que almacena el valor Desconocido en dicho campo si no ingresamos un valor en un **insert**:

```
alter table Alumno
add constraint DF_Nombre
default 'Desconocido'
for Nombre;
```

En la siguiente sentencia se agrega una restricción para el campo documento de la tabla Alumno:

```
alter table Alumno
 add constraint DF_Alumno_documento
 default 0
 for documento;
```

Por convención, cuando demos el nombre a las restricciones **default** emplearemos un formato similar al que le da SQL Server: **DF_NOMBRETABLA_NOMBRECAMPO**

Solamente se permite una restricción **default** por campo y no se puede emplear junto con la propiedad **identity**. Una tabla puede tener varias restricciones **default** para sus distintos campos.

La restricción **default** acepta valores tomados de funciones del sistema, por ejemplo, podemos establecer que el valor por defecto de un campo de tipo **datetime** sea **getdate().**

Restricciones Check

La restricción check especifica los valores que acepta un campo, evitando que se ingresen valores inapropiados o basura.

La sintaxis básica es la siguiente:

alter table NOMBRETABLA

add constraint NOMBRECONSTRAINT

check CONDICION;

Trabajamos con la tabla **Alumno** que tiene los siguientes campos y una restricción **default** para el campo ciudad.

```
Documento bigint not null,
Carnet bigint not null,
Nombre varchar (150) not null,
Apellido varchar (150) not null,
Ciudad varchar(150) default('Ciudad'),
FechaNacimiento date
```

Crearemos una restricción **check** para no permitir la inserción o actualización del campo usando números negativos en el campo Documento:

alter table Alumno

add constraint CK_Alumno_Documento

check (Documento>0);

Este tipo de restricción verifica los datos cada vez que se ejecuta una sentencia **insert** o **update**, es decir, actúa en inserciones y actualizaciones. Sí la tabla ya contiene registros que no cumplen con la restricción que se va a crear, la restricción no se puede crear, hasta que todos los registros cumplan con dicha restricción.

La condición del check puede hacer referencia a otros campos de la misma tabla.

Ejemplo: En la siguiente sentencia se crea una restricción para que el número de documento sea diferente al número de Carnet.

alter table Alumno

add constraint CK_Alumno_DocumentoCarnet

check (Documento<>Carnet);

Deshabilitar y habilitar restricciones check

Sabemos que si agregamos una restricción a una tabla que contiene datos, SQL Server los controla para asegurarse que cumplen con la condición de la restricción, si algún registro no la cumple, la restricción no se establece.

Es posible deshabilitar esta comprobación en caso de restricciones **check**.

Podemos hacerlo cuando agregamos la restricción **check** a una tabla para que SQL Server acepte los valores ya almacenados que infringen la restricción. Para ello debemos incluir la opción **with nocheck** en la instrucción **alter table**

```
alter table TABLA
  with nocheck
  add constraint NOMBRERESTRICCION
  check (CONDICION);
```

Ejemplo:

```
alter table Alumno
with nocheck
 add constraint CK_Alumno_Documento
 check (Documento>0);
```

La restricción no se aplicará a los datos existentes de la tabla, pero si se trata de ingresar un nuevo valor que no cumpla la restricción, SQL Server no lo permite. También podemos deshabilitar las restricciones para agregar o actualizar datos sin comprobarla:

```
alter table Alumno
  nocheck constraint CK_Alumno_DocumentoCarnet
```

En este ejemplo deshabilitamos la restricción **CK_Alumno_Documento** para poder ingresar un valor negativo para **Documento.**

Reglas de restricciones Check:

- Para crear el nombre de las restricciones **check** se seguirá la siguiente estructura: comenzamos con **CK**, seguido del **nombre de la tabla**, del **campo** y alguna palabra con la cual podamos identificar fácilmente de qué se trata la restricción, en caso de que existan varias restricciones **check** para el mismo campo.
- Un campo puede tener varias restricciones **check** y una restricción **check** puede incluir varios campos.
- No se puede aplicar esta restricción junto con la propiedad **identity**
- Si un campo permite valores nulos, null es un valor aceptado, aunque no esté incluido en la condición de restricción.

- Si intentamos establecer una restricción check para un campo que entra en conflicto con otra restricción check establecida al mismo campo, SQL Server no lo permite.

Restricciones usando expresiones regulares

Las condiciones para restricciones **check** también pueden incluir un patrón o una lista de valores. Por ejemplo, establecer que cierto campo conste de 4 caracteres, 2 letras y 2 dígitos:

Para garantizar la integridad de los datos y la basura en los campos se pueden utilizar las restricciones check con expresiones regulares.

Si el tipo de datos de la columna es CHAR o VARCHAR es posible utilizar las siguientes expresiones regulares:

Ejemplos de restricciones con expresiones regulares:

- Esta restricción check solo permite insertar letras mayúsculas, minúsculas y dígitos en el campo Nombre, de la tabla Alumno.
  ALTER TABLE
  Alumno
  ADD CONSTRAINT
  CHK_Alumno
  CHECK (Nombre NOT like '%[^0-9-A-Z-az]%')

- Esta restricción check solo permite insertar letras mayúsculas, minúsculas y dígitos en el campo Nombre, de la tabla Alumno, los caracteres especiales no están permitidos.
  ALTER TABLE
  Alumno
  ADD CONSTRAINT
  CHK_Alumno
  CHECK (Nombre like '%[^@$%&/*-+?¿()!¡^.,;=<>º#]%')

- Esta restricción check solo permite insertar dígitos en el campo Nombre, de la tabla Alumno.
  ALTER TABLE
  Alumno
  ADD CONSTRAINT
  CHK_Alumno
  CHECK (Nombre NOT like '%[^0-9]%')

- Esta restricción check solo permite insertar letras mayúsculas y minúsculas en el campo Nombre, de la tabla Alumno.
  ALTER TABLE
  Alumno
  ADD CONSTRAINT
  CHK_Alumno
  CHECK (Nombre NOT like '%[^A-Z-az]%')

Restricción Primary Key

Cada vez que establecíamos la clave primaria para la tabla, SQL Server creaba automáticamente una restricción **primary key** para dicha tabla. Dicha restricción, a la cual no le dábamos un nombre, recibía un nombre dado por SQL Server que comienza con **PK(por primary key),** seguido del nombre de la tabla y una serie de letras y números aleatorios.

Podemos agregar una restricción primary key a una tabla existente con la sintaxis básica siguiente:

```
alter table NOMBRETABLA
add constraint NOMBRECONSTRAINT
primary key (CAMPO,...);
```

En el siguiente ejemplo definimos una restricción primary key para nuestra tabla Alumno para asegurarnos que cada Alumno tendrá un Documento diferente y único:

```
alter table Alumno
add constraint PK_Alumno_Documento
primary key(Documento);
```

Con esta restricción, si intentamos ingresar un registro con un valor para el campo **Documento** que ya existe o el valor **Null,** aparece un mensaje de error, porque no se permiten valores duplicados ni nulos. Igualmente, si actualizamos.

Por convención, cuando demos el nombre a las restricciones **primary key** seguiremos el formato **PK_NOMBRETABLA_NOMBRECAMPO.**

Cuando agregamos una restricción a una tabla que contiene información, SQL Server controla los datos existentes para confirmar que cumplen las exigencias de la restricción, si no los cumple, la restricción no se aplica y aparece un mensaje de error. Por ejemplo, si intentamos definir la restricción **primary key** para la tabla **Alumno** y hay registros con códigos repetidos o con un valor **null**, la restricción no se establece.

Cuando establecemos una clave primaria al definir la tabla, automáticamente SQL Server redefine el campo como **not null;** pero al agregar una restricción **primary key**, los campos que son clave primaria **DEBEN** haber sido definidos **not null** (o ser implícitamente **not null** si se definen como **identity**).

SQL Server permite definir solamente una restricción **primary key** por tabla, que asegura la unicidad de cada registro de una tabla. Si ejecutamos el procedimiento almacenado **sp_helpconstraint** junto al nombre de la tabla, podemos ver las restricciones **primary key** (y todos los tipos de restricciones) de dicha tabla.

Un campo con una restricción **primary key** puede tener una restricción **check**.

Un campo **primary key** también acepta una restricción **default** (excepto si es **identity**), pero no tiene sentido ya que el valor por defecto solamente podrá ingresarse una vez; si intenta ingresarse

cuando otro registro ya lo tiene almacenado, aparecerá un mensaje de error indicando que se intenta duplicar la clave.

Restricciones UNIQUE

Las restricciones aplicadas a tablas aseguran valores únicos para cada registro.

Anteriormente vimos la restricción **primary key**, otra restricción para las tablas es **unique**.

La restricción **unique** impide la duplicación de claves alternas (no primarias), es decir, especifica que dos registros no puedan tener el mismo valor en un campo. Se permiten valores nulos. Se pueden aplicar varias restricciones de este tipo a una misma tabla, y pueden aplicarse a uno o varios campos que no sean clave primaria.

Se emplea cuando ya se estableció una clave primaria, pero se necesita asegurar que otros datos también sean únicos y no se repitan (como número de documento).

La sintaxis general es la siguiente:

```
alter table NOMBRETABLA
 add constraint NOMBRERESTRICCION
 unique (CAMPO);
```

Ejemplo:

```
 alter table Alumno
  add constraint UQ_alumnos_Documento
  unique (Documento);
```

En el ejemplo anterior se agrega una restricción **unique** sobre el campo **Documento** de la tabla **Alumno**, esto asegura que no se pueda ingresar un Documento si ya existe. Esta restricción permite valores nulos, así que, si se ingresa el valor **null** para el campo Documento, se acepta.

Por convención, cuando se crea el nombre de las restricciones **unique** seguiremos la misma estructura: **UQ_NOMBRETABLA_NOMBRECAMPO**. Quizá parezca innecesario colocar el nombre de la tabla, pero cuando empleemos varias tablas verá que es útil identificar las restricciones por tipo, tabla y campo.

Recuerde que cuando agregamos una restricción a una tabla que contiene información, SQL Server controla los datos existentes para confirmar que cumplen la condición de la restricción, si no los cumple, la restricción no se aplica y aparece un mensaje de error. En el caso del ejemplo anterior, si la tabla contiene números de documento duplicados, la restricción no podrá establecerse; si podrá establecerse si tiene valores nulos. SQL Server controla la entrada de datos en inserciones y actualizaciones evitando que se ingresen valores duplicados.

Laboratorio de trabajo independiente para trabajar con restricciones

Usando el siguiente script, ejecutar las sentencias indicadas y verificar que se cumplan las distintas restricciones.

```
---Crear una base de datos ---
Create database Prematricula
---Usando contexto Prematricula
Use Prematricula
go

---Creando La tabla Alumno
Create table Alumno
(
IdAlumno bigint primary key identity(1,1),
Documento bigint not null,
Nombre varchar(150) not null,
Apellido varchar(150) not null,
Ciudad varchar(150) default('Ciudad'),
FechaNacimiento date,
Edad int
)
go

--- Este Check con expresión regular permite insertar letras y dígitos, no permite
la inserción de otro tipo de caracteres
ALTER TABLE
   Alumno
ADD CONSTRAINT
   CHK_Alumno
   CHECK(Nombre NOT like '%[^0-9-A-Z-az]%')

---Este Check con expresión regular, restringe la inserción de caracteres especiales
en el campo Nombre
ALTER TABLE
Alumno
ADD CONSTRAINT
CHK_Alumno
CHECK (Nombre like '%[^@$%&/*-+?¿()!¡^.,;=<>º#]%')

---Este Check con expresión regular, solo permite la inserción de dígitos en el
campo Nombre
ALTER TABLE
Alumno
ADD CONSTRAINT
CHK_Alumno
CHECK (Nombre NOT like '%[^0-9]%')

---Este Check con expresión regular, solo permite la inserción de letras mayúscula y
minúsculas en el campo Nombre
ALTER TABLE
Alumno
ADD CONSTRAINT
CHK_Alumno
CHECK (Nombre NOT like '%[^A-Z-az]%')
```

```
--Insertando unos datos
Use Prematricula
go
Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,
Ciudad)
values
(1, 3454355,'Miguel', 'Garcia', '01-01-2000', DEFAULT)
go
Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,Ciudad)
values
(2, 3454356,'Ana', 'Alvarez', '01-01-1988','Medellin')
go

Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,Ciudad)
values
(3,56566,'Juan', 'Castro', '08-09-1989','Envigado')
go

Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,Ciudad)
values
(4,56566,'Juan', 'Ordoñez', '02-03-2001','Itagui')
go

-- Insertar datos provenientes de otra tabla (tabla con la misma estructura)
USE [Prematricula]
GO

-- se crea una nueva tabla Alumno2 y se realiza una subconsulta para insertar datos
desde la tabla Alumno creada al inicio
Insert into Alumno2
SELECT * FROM Alumno WHERE Nombre='juan'
Go


---Restriccion Check: Permite insertar solo edades >= 18

Alter Table Alumno
with check
add constraint ck_Edad check (Edad>=18)
go


---Insertando un dato  con fecha que no cumple con la restriccion
Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento)
values
(3, 3454334,'Pedro', 'Galindo', '01-05-2002')
go
---Deshabilitar la restriccion check
Alter table Alumno
nocheck
constraint ck_FechaNacimiento2
go

---Borrar la restriccion check
Alter table Alumno
drop constraint ck_FechaNacimiento2
go
---Crear una restricción que NO compruebe los datos ya ingresados
```

```
Alter Table Alumno
with nocheck
add constraint ck_FechaNacimiento2 check
(datediff(year,FechaNacimiento,getdate())>=18)
go
---Crear la tabla AsignaturaClase
Create table AsignaturaClase
(
IdAsignatura bigint not null primary key,
IdAlumno bigint,
NombreClase varchar(150),
)

--Restriccion de llave foranea
Alter Table AsignaturaClase
add constraint FK_Alumno_clase foreign key (IdAlumno) references
Alumno (IdAlumno) on Update Cascade on Delete no Action
go
---Revisar constraint de una tabla
sp_helpconstraint Alumno

--Consultar los datos de la tabla Alumno y AsignaturaClase
SELECT * FROM Alumno
SELECT * FROM AsignaturaClase

--Consultar restricciones
sp_helpconstraint AsignaturaClase

--Deshabilitar la restriccion de llave foranea
Alter table AsignaturaClase
nocheck
constraint FK_Alumno_clase

--- eliminar llave foranea tabla

Alter table [tabla] drop constraint fk_constraint;


--Insertar datos a AsignaturaClase
Insert into AsignaturaClase(IdAsignatura,IdAlumno,NombreClase)
values
(1,1,'Matematica Avanzada')
go

Insert into AsignaturaClase(IdAsignatura,IdAlumno,NombreClase)
values
(2,2,'Lógica')
go

--Deshabilitar el identity----------------------------------------
---nueva tabla
Create table Clase
(IdClase int identity(1,1) not null primary key,
NombreClase varchar(100)
)
go
---Insertar un dato
Insert into Clase (NombreClase)
```

```
values ('Matematicas especiales')
--deshabilitar el identity_insert
Set identity_insert dbo.Clase ON
--Insertar el dato insertando la columna IdClase
Insert into Clase (IdClase,NombreClase)
values (2,'Bases de datos')
--Insertar datos a la nueva tabla
Insert into Clase (IdClase, NombreClase)
values (3,'Ingenieria de Software')

--deshabilitar el identity_insert
Set identity_insert dbo.Clase Off
---Consultando la tabla
SELECT * FROM Alumno
```

Índices

SQL Server accede a los datos de dos maneras:

- recorriendo las tablas; comenzando el principio y extrayendo los registros que cumplen las condiciones de la consulta.
- empleando índices; recorriendo la estructura de árbol del índice para localizar los registros y extrayendo los que cumplen las condiciones de la consulta.

Los índices se emplean para facilitar la obtención de información de una tabla. El índice de una tabla desempeña la misma función que el índice de un libro: permite encontrar datos rápidamente; en el caso de las tablas, localiza registros.

Una tabla se indexa por un campo (o varios). Un índice posibilita el acceso directo y rápido haciendo más eficiente las búsquedas. Sin índice, SQL Server debe recorrer secuencialmente toda la tabla para encontrar un registro.

El objetivo de un índice es acelerar la recuperación de información. La indexación es una técnica que optimiza el acceso a los datos, mejora el rendimiento acelerando las consultas y otras operaciones. Es útil cuando la tabla contiene miles de registros, cuando se realizan operaciones de ordenamiento y agrupamiento y cuando se combinan varias tablas (tema que veremos más adelante).

Los índices más adecuados son aquellos creados con campos que contienen valores únicos.

Es importante identificar el o los campos por los que sería útil crear un índice, aquellos campos por los cuales se realizan búsqueda con frecuencia: claves primarias, claves externas o campos que combinan tablas.

No se recomienda crear índices por campos que no se usan con frecuencia en consultas o no contienen valores únicos.

SQL Server permite crear tres tipos de índices:

1) agrupados(clustereados)

2) no agrupados (No clustereados)

3**) Columnares**

Índice agrupado: es similar a un directorio telefónico, los registros con el mismo valor de campo se agrupan juntos. Un índice agrupado determina la secuencia de almacenamiento de los registros en una tabla. Se utilizan para campos por los que se realizan búsquedas con frecuencia o se accede siguiendo un orden.

Una tabla sólo puede tener **UN** índice agrupado.

índice no agrupado: es como el índice de un libro, los datos se almacenan en un lugar diferente al del índice, los punteros indican el lugar de almacenamiento de los elementos indizados en la tabla. Un índice no agrupado se emplea cuando se realizan distintos tipos de búsquedas frecuentemente, con campos en los que los datos son únicos.

- Una tabla puede tener hasta **249 índices** no agrupados.
- Si no se especifica un tipo de índice, de modo predeterminado será no agrupado.
- Los campos de tipo text, ntext e image no se pueden indexar
- Es recomendable crear los índices agrupados antes que los no agrupados, porque los primeros modifican el orden físico de los registros, ordenándolos secuencialmente.
- La diferencia básica entre índices agrupados y no agrupados es que los registros de un índice agrupado están ordenados y almacenados de forma secuencial en función de su clave.
- SQL Server crea automáticamente índices cuando se crea una restricción primary key o unique en una tabla.

Índice Columnar:

Los índices, columnares, surgieron en un proyecto llamado Apollo, este proyecto incluía dos partes:

- Índices columnares – nuevo modo de almacenamiento.
- Un nuevo modo de procesamiento Batch.

Los índices columnares nos aportan la capacidad de resolver consultas, muy complejas, de una manera mucho más rápida.

Estos índices, no son ni más ni menos que un nuevo modelo de índices que se añaden al modelo relacional, con una peculiaridad en su modo de almacenamiento. Hasta ahora la información se almacenaba en páginas de datos, y estas páginas de datos contenían filas con los datos. Ahora ya no es así, ahora el modo de almacenamiento de estos índices es en columnas, más abajo entro en más en detalle cuando quiero decir en columnas.

Los índices columnares, traen compresión Vertipaq. Vertipaq surge en SQL Server 2008 R2, con la aparición de Powerpivot y los sistemas de carga en memoria, donde se desarrollaron nuevos algoritmos, los cuales comprimen alrededor de 10 veces los datos, más abajo tratamos esto más en detalle… La compresión nativa que teníamos en versiones anteriores a SQL Server no encaja en este nuevo tipo de índices, ya que se quiere manejar grandes volúmenes de datos eficientemente.

Funcionamiento de los índices columnares, principalmente tiene tres conceptos:

- La primera: el modo de almacenamiento, se cambió del modo de almacenamiento de filas por el almacenamiento en columnas. Cuando nosotros lanzamos consultas, normalmente seleccionamos las columnas que deseamos, y este modo de almacenamiento es propicio para ello. Luego hablo más en detalle de esto.
- La segunda: **Vertipaq**, algoritmos de compresión que comprimen los datos, más concretamente las columnas, de una manera bestial. Ganamos reducción de actividad de entrada y salida, es decir, cuando vamos a coger los datos del disco están comprimidos… Y ganamos con una maximización del uso de la memoria, donde estos datos comprimidos en memoria ocupan menos… Aquí tenemos la clave…
- La tercera: el nuevo modo de procesamiento en **Batch**, que viene incluido para este tipo de índices columnares

Ahora para el almacenamiento, ya no se habla de páginas de datos, se hace en segmentos. Segmentos que contienen los valores de las columnas. Un segmento por columna o parte de una columna. Altamente comprimidos con Vertipaq. Cada columna va por su cuenta.

Anteriormente o en versiones anteriores, cuando hacemos una consulta, SQL Server, se va a disco y carga las páginas en memoria, se lee toda la tabla y carga las páginas que necesita en memora… Cada página ocupa 8 k… De esta forma carga información que no es útil en memoria, información que está dentro de estas páginas… Malgastando la memoria, un recurso que es tan preciado … En cambio, el almacenamiento por segmentos en columnas, donde tenemos almacenados los segmentos comprimidos en disco, cuando nos llega la consulta, simplemente va a cargar en memoria los segmentos necesarios de las columnas necesarias, cargando solo en memoria lo que realmente es útil y va a usar…

Modo de procesamiento en **Batch**:

El modo **Batch**, es un nuevo modo de procesamiento, que viene con los índices columnares, tenemos que cambiar la filosofía que tenemos hasta ahora. En versiones anteriores cuando nosotros procesábamos conjuntos de datos lo hacíamos fila por fila, ahora aparece un nuevo modo **Batch**, que nos proporciona un procesamiento vectorial, que en lugar de ir fila por fila, nos procesa conjuntos de filas, donde esos conjuntos de filas suelen ser de 1000 filas aproximadamente. Con esto se aprovecha el paralelismo mucho más, y al procesar por conjuntos de filas, reduce el consumo de CPU, produciendo mejor rendimiento a las consultas. Este modo de procesamiento solo funciona para índices columnares, siempre que usemos otro tipo de índices, procesará registro a registro, no por conjuntos de registros.

El modo **Batch**, no se utiliza siempre, aunque usemos índices columnares, para que use ente modo de procesamiento **Batch**, nuestra consulta tiene que tener algún JOIN o algún filtro o algún

operador de agregación, que lo que haga es reducir un gran conjunto de datos con la agregación, es cuando usará este tipo de procesamiento. También si el conjunto de datos para el cual hemos creado índices columnares, son muy pequeños, (tablas pequeñas), es muy probable que tampoco use este modo de procesamiento. Lo decide el Optimizador de consultas si utilizarlo o no…

Limitaciones:

Los datos no soportados para índices son:

- **Decimal y/o numéricos (precisión > 18)**
- **Datetimeoffset (precisión >2)**
- **Binary, Varbinary**
- **Image**
- **Text, ntext.**
- **Varchar(max), nvarchar(max)**
- **Hierarchyid**
- **Timestamp**
- **Uniqueidentifier**
- **Sqlvariant**
- **Xml**

Reglas de índices columnares

- Solo podemos tener un índice por tabla, se recomienda crear un índice clúster, por todas las tablas que tengan un tipo de datos compatible, porque el índice columnar aísla cada columna.
- No se puede aplicar a vistas indexadas.
- No admite compresión nativa de SQL Server, porque ya está comprimida con Vertipaq.
- El índice columnar no puede ser un índice filtrado
- El límite de columnas, no puede superar a 1024 columnas.
- Cuando creamos un índice columnar en una tabla, esta tabla pasa a convertirse en una tabla de solo lectura. No se podrán realizar inserciones, updates, deletes ni podrás usar operadores tipo Merge...

Aunque existen alternativas, pero a priori no funcionan las modificaciones… Alternativas como eliminar el índice, hacer las inserciones y volver a crear el índice. Esta operativa probablemente no encaje en la lógica de tu aplicación, por lo que en este caso tendríamos que olvidarnos de estos índices columnares.

Otra forma de hacerlo es manteniendo un histórico con la/s tabla/s con índice columnar y una/s tabla/s auxiliar/es sin índice columnar donde cargaremos los datos diarios, entonces sobre las tablas con índice c. y sin índice c. crearemos unas vistas con unión sobre las dos tablas, las consultas irán sobre la vista, y tendremos entonces para un mayor volumen de datos índices columnares y para los datos diarios no. Cuando crezca demasiado las tablas auxiliares hay que volcarlas en una ventana de mantenimiento sobre las tablas con índices columnares.

Y una última alternativa es utilizando particionado, donde partimos de nuestra tabla principal, con un índice columnar creado, y crearemos una tabla intermedia donde realizaremos las cargas y posteriormente crearemos un índice columnar, para nuestras consultas nos apoyaremos en las tablas intermedias y no en la tabla principal.

Resumiendo, los índices facilitan la recuperación de datos, permitiendo el acceso directo y acelerando las búsquedas, consultas y otras operaciones que optimizan el rendimiento general.

¿Como crear índices?

Para crear índices empleamos la instrucción CREATE INDEX.

La sintaxis básica es la siguiente:

CREATE TIPODEINDICE index NOMBREINDICE

on TABLA(CAMPO);

TIPODEINDICE indica si es agrupado (clustered) o no agrupado (nonclustered). Si no especificamos crea uno No agrupado. Independientemente de si es agrupado o no. Se utiliza el prefijo **CL_** para crear el nombre del índice.

Crear un índice agrupado:

```
CREATE CLUSTERED INDEX CL_NombreIndice ON Tabla(campo,…);
```

Crear un índice NO agrupado:

```
CREATE NONCLUSTERED INDEX CL_NombreIndice ON Tabla(campo1, campo2);
```

Crear un índice Columnar:

Para los índices de tipo columnar se agrega la palabra COLUMNSTORE entre CLUSTERED e INDEX

```
CREATE CLUSTERED COLUMNSTORE INDEX CL_NombreIndice ON tabla (campo…);
```

SQL Server crea automáticamente índices cuando se establece una restricción primary key o unique en una tabla. Al crear una restricción primary key, si no se especifica, el índice será agrupado (clustered) a menos que ya exista un índice agrupado para dicha tabla. Al crear una restricción unique, si no se especifica, el índice será no agrupado (non-clustered).

Puede especificarse que un índice sea agrupado o no agrupado al agregar estas restricciones.

Agregamos una restricción primary key al campo Documento de la tabla Alumno especificando que cree un índice NO agrupado:

```
alter table Alumno
 add constraint PK_Alumno_Documento
 primary key nonclustered (Documento);
```

Para ver los índices de una tabla:

exec sp_helpindex Alumno;

Muestra el nombre del índice, si es agrupado, no agrupado o columnar, primary (o unique) y el campo por el cual se indexa.

Todos los índices de la base de datos activan se almacenan en la tabla del sistema sysindexes, podemos consultar dicha tabla tipeando:

SELECT name FROM sysindexes;

Para ver todos los índices de la base de datos activa creados por nosotros podemos tipear la siguiente consulta:

SELECT name FROM sysindexes

WHERE name like 'CL_%';

Laboratorio de trabajo independiente para trabajar con índices

Para desarrollar el laboratorio es necesario restaurar el backup completo de la base de datos Ciudadanos, este .BAK fue compartido en la clase.

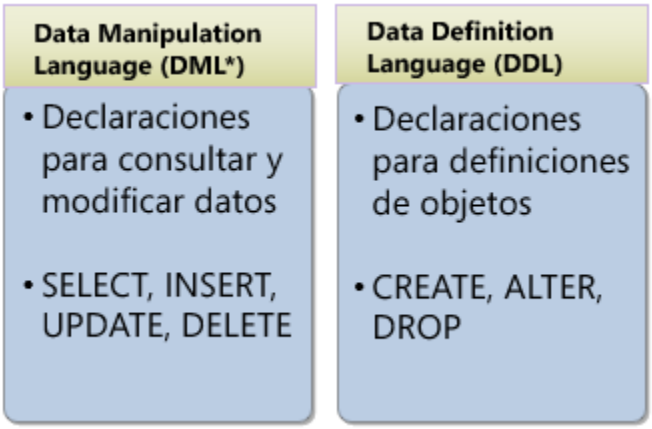
```
-- Usar el contexto de la base de datos Ciudadanos
Use Ciudadanos
--Borrar todos los planes de memoria de caché
DBCC FREEPROCCACHE WITH NO_INFOMSGS;

--vaciar la caché de datos
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

--consultar si una tabla tiene índices creados
sp_helpindex tabla


-- verificar si existen índices  en una tabla
sp_helpindex Ciudadano

--Borrar indice un índice
DROP INDEX CL_NOMBRE ON Ciudadano;

--Presionar Ctrl+L para ver el plan de ejecución estimado de la consulta antes y
después de crear un índice
SELECT Campos1, Campos2,Campo3 FROM Tabla WHERE Campos1='dato a buscar' and
Campos2='dato a buscar'

-- Crear un índice clustereado- o agrupado
CREATE CLUSTERED  INDEX CL_NombreIndice ON tabla(campo);
go


-- Crear un índice NO clustereado- o NO agrupado
CREATE NONCLUSTERED INDEX CL_NombreIndice on Tabla(campo1,campo2);
go
```

```
--Presionar Ctrl+L para ver el plan de ejecución estimado de la consulta antes y
después de crear un índice
SELECT NOM1,NOM2,APE1,APE2 FROM Ciudadano WHERE APE1='cardenas' and
APE2='valenzuela'

-- Crear índices columnares
CREATE CLUSTERED COLUMNSTORE INDEX Cl_Ciudadano ON Ciudadano


/*Adicionando  este comando a la creación de un íncide, se borra el índice si existe
previamente
e indica el máximo de procesadores de la máquina para usar en la ejecución.*/
WITH (DROP_EXISTING=ON, MAXDOP=4);
```

Transact -SQL

- Lenguaje de consulta estructurado (SQL)
- Desarrollado por IBM en 1970
- Adoptado como norma por los organismos de normas ANSI y ISO
- Ampliamente utilizado en la industria
- PL / SQL, SQL Procedural Language, Transact-SQL
- La implementación de Microsoft es Transact-SQL
- Referido como T-SQL
- Lenguaje de consulta para SQL Server
- SQL es declarativo, no procedural (Describa lo que desea, no especifique los pasos).

Categorías Transact -SQL



DML

Un lenguaje de manipulación de datos (Data Manipulation Language, o DML en inglés) es un lenguaje proporcionado por el sistema de gestión de base de datos que permite a los usuarios llevar a cabo las tareas de consulta o manipulación de los datos, organizados por el modelo de datos adecuado. El lenguaje de manipulación de datos más popular hoy día es SQL, usado para recuperar y manipular datos en una base de datos relacional.

Elementos del lenguaje T-SQL: Predicados y operadores

Elementos:	**Predicados y operadores:**
Predicados	**IN, BETWEEN, LIKE**
Operadores de Comparación	**=, >, <, >=, <=, <>, !=, !>, !<**
Operadores Lógicos	**AND, OR, NOT**
Operadores Aritméticos	**+, -, *, /, %**
Concatenación	**+**

Teoría de Conjuntos y SQL Server

La teoría de conjuntos es una base matemática para el modelo de base de datos relacional

¿Qué es un conjunto?

Una colección de objetos distintos considerados como un todo

Ej: Todos los clientes que viven en Medellín

Características del elemento de conjunto	**Ejemplo**
Elementos de un conjunto llamados miembros	Cliente como miembro de un conjunto de Clientes
Los elementos de un conjunto se describen por atributos	Nombre del cliente, apellido, fecha de nacimiento
Los elementos deben ser distintos o únicos	Customer ID

Teoría de conjuntos aplicada a las consultas de SQL Server

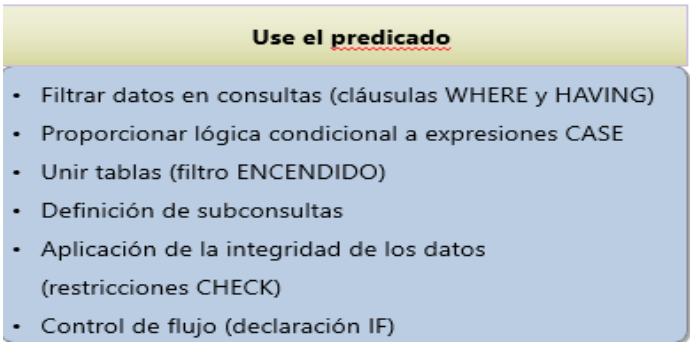
Aplicación de la teoría	**Comentario**
Actuar sobre todos los elementos de un conjunto a la vez	Consulta toda la tabla a la vez
Use un proceso declarativo basado en conjuntos	Dile al motor lo que quieres recuperar
Los elementos del conjunto deben ser únicos	Definir claves únicas en una tabla
Sin orden definido para resultado conjunto	Los artículos pueden ser devueltos en cualquier orden. Esto requiere instrucciones de clasificación explícitas si se desea una orden

Ejemplos:

- Empleados (conjunto de todos los empleados)
- Clientes (conjunto de todos los clientes)
- Pedidos (conjunto de todos los pedidos)
- Clientes ubicados en Medellín
- Empleados contratados entre 2019 y 2021
- Clientes que hicieron pedidos en 2018

Lógica de predicados aplicada a consultas de SQL Server

En SQL Server, un predicado es una propiedad o expresión que se evalúa como verdadera, falsa o desconocida (NULL).



Use el predicado
• Filtrar datos en consultas (cláusulas WHERE y HAVING) • Proporcionar lógica condicional a expresiones CASE • Unir tablas (filtro ENCENDIDO) • Definición de subconsultas • Aplicación de la integridad de los datos (restricciones CHECK) • Control de flujo (declaración IF)

Declaración SELECT

La sentencia SELECT nos permite consultar los datos almacenados en una tabla de una base de datos**.**

Se pueden combinar identificadores, valores y operadores evaluados para obtener un único resultado.

Elementos de una declaración SELECT

Elemento	Expresión	Rol
SELECT	<SELECT list>	Define qué columnas retornar
FROM	<table source>	Define tabla (s) para consultar
WHERE	<search condition>	Filtra filas usando un predicado
GROUP BY	<GROUP BY list>	Organiza filas por grupos
HAVING	<search condition>	Grupos de filtros que usan un predicado (conteos)
ORDER BY	<ORDER BY list>	Ordena la salida

Procesamiento de las consultas Lógicas

El orden en que se escribe una consulta no es el orden en que SQL Server lo evalúa.

```
5: SELECT    <lista de campos>
1: FROM    <table de origen>
2: WHERE    <condición del filtro>
3: GROUP BY <lista de campos por los que agrupar>
4: HAVING <condición del filtro>
6: ORDER BY <lista por la cual ordenar>
```

Cláusula SELECT

Puede ser una sola constante, función de un solo valor o variable

Se puede combinar si las expresiones tienen el mismo tipo de datos

`Sintaxis una sentencia SELECT:`

```
SELECT Campos FROM Tabla;
```

SELECT * FROM

Con el asterisco le indico a SLQ SERVER que retorno todas las columnas de la tabla y todos sus registros.

DISTINCT

Con la cláusula **DISTINCT** se especifica que los registros con ciertos datos duplicados sean ignorados en el resultado. Por ejemplo, queremos conocer todos los autores de los cuales tenemos libros, si utilizamos esta sentencia:

SELECT Nombre FROM Alumno;

Aparecen repetidos. Para obtener la lista de autores sin repetición usamos:

SELECT DISTINCT Nombre FROM Alumno;

SELECT Nombre FROM Alumno

GROUP BY Nombre r;

Cláusula WHERE

Existe una cláusula, **WHERE** con la cual podemos especificar condiciones para una consulta **SELECT** Es decir, podemos recuperar algunos registros, sólo los que cumplan con ciertas condiciones indicadas con la cláusula **WHERE** Por ejemplo, queremos ver el usuario cuyo nombre es Jairo, para ello utilizamos **WHERE** y luego de ella, la condición:

Para las condiciones se utilizan operadores relacionales, El signo igual (=) es un operador relacional.

SELECT Nombre

FROM Alumno

WHERE nombre='Jairo';

La sintaxis básica y general es la siguiente:

SELECT NOMBRECAMPO1, ..., NOMBRECAMPOn

FROM NOMBRETABLA

WHERE CONDICION;

- Si ningún registro cumple la condición establecida con el **WHERE** no aparecerá ningún registro.
- Entonces, con **WHERE** establecemos condiciones para recuperar algunos registros.

Si el retorne de la consulta arroja valores nulos, podemos modificar la sentencia para que ignore los valores nulos o desconocidos.

SELECT DISTINCT Nombre FROM Alumno

WHERE Nombre IS NOT NULL;

Para contar los distintos nombres, sin considerar el valor NULL usamos:

```
SELECT count(distinct Nombre)
 FROM Alumno;
```

Note que, si contamos los autores sin distinct, no incluirá los valores NULL pero si los repetidos:

```
SELECT count(nombre)
 FROM Alumno;
```

TOP

La palabra clave **TOP** se emplea para obtener sólo una cantidad limitada de registros, los primeros N registros de una consulta.

Con la siguiente consulta obtenemos todos los datos de los primeros 2 libros de la tabla:

```
SELECT TOP 2 * FROM Alumno;
```

Es decir, luego del SELECT se coloca TOP seguido de un número entero positivo y luego se continúa con la consulta.

Se puede combinar con **ORDER BY** para ordenar los resultados de forma ascendente o descendente por un campo de la tabla.

```
SELECT TOP 3 Nombre, Carnet
 FROM Alumno
 order by Nombre;
```

Otro argumento posible cuando utilizamos la cláusula **TOP** es **percent** indicando el porcentaje de registros a recuperar y no la cantidad, por ejemplo:

```
SELECT TOP 50 percent
 * FROM Autor
 order by Nombre;
```

Se recuperan la mitad de registros de la tabla Alumno.

- Si colocamos un valor para **TOP** que supera la cantidad de registros de la tabla, SQL Server muestra todos los registros.

GROUP BY

Podemos generar valores de resumen para un solo campo, combinando las funciones de agregado con la cláusula GROUP BY, que agrupa registros para consultas detalladas.

Con **GROUP BY** podemos agrupar los campos para mostrar los resultados sin repetir.

```
SELECT Carnet, count(distinct Nombre)
```

FROM Alumno

GROUP BY Carnet;

La instrucción anterior solicita que muestre el nombre del alumno y cuente la cantidad agrupando los registros por el campo Carnet. Como resultado aparecen los nombres de los alumnos y la cantidad de registros para cada valor del campo.

Entonces, para saber la cantidad de alumnos que tenemos con cada nombre, utilizamos la función count(), agregamos GROUP BY (que agrupa registros) y el campo por el que deseamos que se realice el agrupamiento, también colocamos el nombre del campo a recuperar; la sintaxis básica es la siguiente:

Sintaxis:

SELECT CAMPO, FUNCIONDEAGREGADO

FROM NOMBRETABLA

GROUP BY CAMPO;

También se puede agrupar por más de un campo, en tal caso, luego del GROUP BY se listan los campos, separados por comas. Todos los campos que se especifican en la cláusula GROUP BY deben estar en la lista de selección.

SELECT CAMPO1, CAMPO2, FUNCIONDEAGREGADO

FROM NOMBRETABLA

GROUP BY CAMPO1,CAMPO2;

HAVING

Así como la cláusula WHERE permite seleccionar (o rechazar) registros individuales; la cláusula having permite seleccionar (o rechazar) un grupo de registros.

Si queremos saber la cantidad de Alumno agrupados por Nombre usamos la siguiente instrucción:

SELECT editorial, count(*)

FROM libros

GROUP BY editorial;

Si queremos saber la cantidad de alumnos agrupados por nombre, pero considerando sólo algunos grupos, por ejemplo, los que devuelvan un valor mayor a 2, usamos la siguiente instrucción:

SELECT Nombre, count(*) FROM Alumno

GROUP BY Nombre

having count(*)>2;

Se utiliza having, seguido de la condición de búsqueda, para seleccionar ciertas filas retornadas por la cláusula GROUP BY.

Veamos otros ejemplos. Queremos el promedio de los precios de los productos agrupados por Categoría, pero solamente de aquellos grupos cuyo promedio supere los $ 50.000:

SELECT Categoria, avg(precio) FROM Productos

GROUP BY Categoria

having avg(precio)> 50.000;

En algunos casos es posible confundir las cláusulas **WHERE** y **HAVING**. Queremos contar los registros agrupados por editorial sin tener en cuenta a la editorial Planeta.

Analicemos las siguientes sentencias:

SELECT Categoria, count(*) FROM **Productos**

WHERE Categoria <>'Licores'

GROUP BY Categoria;

SELECT Categoria, count(*) FROM **Productos**

GROUP BY Categoria

HAVING Categoria <>'Licores';

Ambas devuelven el mismo resultado, pero son diferentes. La primera, selecciona todos los registros rechazando los de la categoría Licores y luego los agrupa para contarlos. La segunda, selecciona todos los registros, los agrupa para contarlos y finalmente rechaza filas con el conteo correspondiente a la categoría Licores.

- No debemos confundir la cláusula **WHERE** con la cláusula **HAVING**; la primera establece condiciones para la selección de registros de un **SELECT**; la segunda establece condiciones para la selección de registros de una salida **GROUP BY**.

ALIAS de Columna(campo) y ALIAS de Tabla

Una manera de hacer más comprensible el resultado de una consulta consiste en cambiar los encabezados de las columnas y asignar un predicado para identificar las tablas en las consultas. Por ejemplo, tenemos la tabla Alumno con un campo Nombre (entre otros) en el cual se almacena el nombre de un alumno; queremos que al mostrar la información de dicha tabla aparezca como encabezado del campo Nombre el texto Name, para ello colocamos un alias de la siguiente manera:

ALIAS de columna:

SELECT Nombre AS Name

FROM Alumno

Para reemplazar el nombre de un campo por otro, se coloca la palabra clave **AS** seguido del texto del encabezado.

Si el **ALIAS** consta de una sola cadena las comillas no son necesarias, pero si contiene más de una palabra, es necesario colocarla entre comillas simples:

Ejemplo:

SELECT Nombre as 'Nombres y apellidos,

FROM Alumno;

- Un alias puede contener hasta **128** caracteres.
- También se puede crear un alias para columnas calculadas.
- La palabra clave **AS** es opcional en algunos casos, pero es conveniente usarla.

Entonces, un **ALIAS** se usa como nombre de un campo o de una expresión. En estos casos, son opcionales, sirven para hacer más comprensible el resultado.

ALIAS de Tabla

El siguiente ejemplo describe la sentencia para usar un ALIAS de tabla, en donde se utiliza una letra que se usa como prefijo para identificar el nombre de una tabla dentro de la consulta.

```
SELECT P.ProductId, P.UnitPrice , P.QuantityPerUnit, P.Discontinued

 FROM Products P ;
```

SUBCONSULTAS

Una subconsulta (subquery) es una sentencia SELECT anidada en otra sentencia SELECT, INSERT UPDATE, DELETE (o en otra subconsulta).

Las subconsultas se emplean cuando una consulta es muy compleja, entonces se la divide en varios pasos lógicos y se obtiene el resultado con una única instrucción y cuando la consulta depende de los resultados de otra consulta.

- Generalmente, una subconsulta se puede reemplazar por combinaciones y estas últimas son más eficientes.
- Las subconsultas se DEBEN incluir entre paréntesis.
- Puede haber subconsultas dentro de subconsultas, se admiten hasta 32 niveles de anidación.

Se pueden emplear subconsultas:

- en lugar de una expresión, siempre que devuelvan un solo valor o una lista de valores.
- que retornen un conjunto de registros de varios campos en lugar de una tabla o para obtener el mismo resultado que una combinación (join).

Hay tres tipos básicos de subconsultas:

- las que retornan un solo valor escalar que se utiliza con un operador de comparación o en lugar de una expresión.

- las que retornan una lista de valores, se combinan con in, o los operadores ANY, SOME y ALL.
- los que testean la existencia con **EXIST**

Reglas a tener en cuenta al emplear subconsultas:

- la lista de selección de una subconsulta que va luego de un operador de comparación puede incluir sólo una expresión o campo (excepto si se emplea exists y in).

- si el **WHERE** de la consulta exterior incluye un campo, este debe ser compatible con el campo en la lista de selección de la subconsulta.

- no se pueden emplear subconsultas que recuperen campos de tipos text o image.

- las subconsultas luego de un operador de comparación (que no es seguido por any o all) no pueden incluir cláusulas **GROUP BY** ni **HAVING**

- **DISTINCT** no puede usarse con subconsultas que incluyan **GROUP BY**

- ORDER BY puede emplearse solamente si se especifica TOP también.

- una vista creada con una subconsulta no puede actualizarse.

- una subconsulta puede estar anidada dentro del **WHERE** o **HAVING** de una consulta externa o dentro de otra subconsulta.

- sí una tabla se nombra solamente en una subconsulta y no en la consulta externa, los campos no serán incluidos en la salida (en la lista de selección de la consulta externa).

las subconsultas que retornan una lista de valores reemplazan a una expresión en una cláusula **WHERE** que contiene la palabra clave **IN**.

El resultado de una subconsulta con **IN** (o **NOT IN**) es una lista. Luego que la subconsulta retorna resultados, la consulta exterior los usa.

La sintaxis básica es la siguiente:

WHERE EXPRESION in (SUBCONSULTA);

Ejemplos:

```
--subconsultas self contained(retorna más de un valor), no tienen dependencia de la
consulta externa IN: incluye únicamente los datos del IN()

SELECT * FROM CUSTOMERS

WHERE COUNTRY   IN ('BRAZIL','GERMANY','SPAIN')


--- subconsultas correlacionadas-(retorna más de un valor)-IN-NOT IN
SELECT * FROM [Order Details] AS OD
WHERE OD.ProductID IN(SELECT P.ProductID FROM Products AS P WHERE
P.UnitPrice=OD.UnitPrice )


--subconsultas self contained(retorna más de un valor),no tienen dependencia de la
consulta externa -NOT IN: EXCLUYE LOS DATOS CON NOT IN()

SELECT * FROM CUSTOMERS

WHERE COUNTRY  NOT IN ('BRAZIL','GERMANY','SPAIN')


--- subconsultas correlacionadas-(retorna más de un valor)-Exists

SELECT ContactName
FROM Suppliers
          WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =
          Suppliers.supplierID AND UnitPrice = 38.0000);
```

Laboratorio de trabajo independiente sentencia SELECT SQL SERVER

Para desarrollar el laboratorio es necesario tener creada la base de datos **NORTHWIND**, con el script siniestrado en clase.

```
--DECLARACIÓN SELECT

USE NORTHWIND

GO
-- sintaxis SELECT
--SELECT Campos FROM Tabla;

SELECT country
 FROM Customers;


-- Comentario de  una linea con  --
```

```sql
/*Comenario de un bloque */

/*  ejercicios usando la base de datos de
 Northwind */
 -- COUNT
 SELECT COUNT(*) FROM CUSTOMERS

 -- COUNT, CUENTA SOLO LOS REGISTROS LLENOS, LOS VALORES VACIOS NO

SELECT COUNT(FAX) FROM CUSTOMERS

--COUNT, CUENTA TODAS LAS FILAS

SELECT COUNT(*) FROM CUSTOMERS

 -- No es buena pràctica
  SELECT * FROM CUSTOMERS

-- SELECT TOP  ASC-DESC
SELECT TOP(20) COUNTRY,City  FROM CUSTOMERS ORDER BY COUNTRY ASC


--CLAUSULA PREDICADO WHERE- = <,>, <>, <=,>=

SELECT * FROM CUSTOMERS

WHERE COUNTRY=123

-- SELECT-WHERE-ORDER BY- AND-BETWEEN
SELECT ProductId, UnitPrice FROM Products WHERE UnitPrice between 18.0000 AND
30.0000 ORDER BY UnitPrice  asc

--SELECT-WHERE-OR
SELECT * FROM CUSTOMERS
WHERE City='Berlin' OR City='München';

 --Muestra todas las ciudades con duplicados
SELECT COUNTRY,City,PostalCode FROM CUSTOMERS

--Alias de Columna
SELECT ProductId, UnitPrice AS Precio
 FROM  Products;

 --Alias de tabla
SELECT P.ProductId, P.UnitPrice , P.QuantityPerUnit, p.Discontinued
 FROM  Products P ;

--DISTINCT MUESTRA SOLO VALORES UNICOS de lista de columna no de tabla fuente

SELECT DISTINCT COUNTRY FROM CUSTOMERS

-- SELECT-GROUP BY

SELECT COUNT(CustomerID) as ConteoCliente, Country, City
FROM CUSTOMERS
GROUP BY Country,City
ORDER BY COUNT(CustomerID) DESC;
```

```sql
-- SELECT-COUNT-GROUP BY-HAVING - >
SELECT COUNT(CustomerID) as ConteoCliente, Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 1
ORDER BY COUNT(CustomerID) DESC;



--Comparadores > < >= <= BETWEEN, LIKE

-- DATETIME AAAA-MM-D-  BETWEEN
SELECT * FROM ORDERS

WHERE ORDERDATE BETWEEN '1998-01-01' AND '1998-12-31' AND EMPLOYEEID=7


--Instruccion LIKE el simbolo _ sustituye una letra, % varias letras

SELECT * FROM CUSTOMERS

WHERE COMPANYNAME LIKE '_a%'




--Instrucci□n LIKE todos los que comienzan con A y con D

SELECT * FROM customers

WHERE companyname LIKE '[AD]%'


--Instruccion LIKE todos los que comienzan con A, B, C, D

SELECT * FROM customers

WHERE companyname like '[A-E]%'


--Instruccion LIKE todos los que NO comiencen con la letra A,  [^]no contiene

SELECT * FROM customers

WHERE companyname like '[^A]%'


SELECT * FROM customers

WHERE companyname like '%PARIS%'

--TODOS LOS QUE NO COMIENZAN CON B

SELECT * FROM CUSTOMERS
```

```sql
WHERE COUNTRY NOT LIKE 'B%'


--subconsultas self contained(retorna más de un valor), no tienen dependencia de la
consulta externa IN: incluye unicamente los datos del IN()

SELECT * FROM CUSTOMERS

WHERE COUNTRY   IN ( 'BRAZIL','GERMANY','SPAIN')


--- subconsultas correlacionadas-(retorna más de un valor)-IN-NOT IN
SELECT * FROM [Order Details] AS OD
WHERE OD.ProductID IN(SELECT P.ProductID FROM Products AS P WHERE
P.UnitPrice=OD.UnitPrice )



--subconsultas self contained(retorna más de un valor),no tienen dependencia de la
consulta externa -NOT IN: EXCLUYE LOS DATOS CON NOT IN()

SELECT * FROM CUSTOMERS

WHERE COUNTRY NOT IN ( 'BRAZIL','GERMANY','SPAIN')




--- subconsultas correlacionadas-(retorna más de un valor)-Exists

SELECT ContactName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =
Suppliers.supplierID AND UnitPrice = 38.0000);


--USO DEL NULL Y IS NOT NULL

--TODOS LOS REGISTROS DE FAX VACIOS

SELECT CUSTOMERID, COMPANYNAME, FAX, Region FROM CUSTOMERS

WHERE Region IS NULL


-- TODOS LOS REGISTROS DE FAX LLENOS (no nulos)

SELECT CUSTOMERID, COMPANYNAME, FAX, Region FROM CUSTOMERS

WHERE Fax IS NOT NULL

-- ORDER BY PARA ORDENAR POR LA COLUMNA O COLUMNAS QUE SE INDIQUEN
SELECT COUNTRY,CUSTOMERID,CONTACTNAME, CONTACTTITLE FROM CUSTOMERS

ORDER BY COUNTRY ASC , CUSTOMERID DESC
```

Sentencia INSERT

Una sentencia INSERT de SQL agrega uno o más registros a una (y sólo una) tabla en una base de datos relacional.

Sintaxis:

INSERT INTO NOMBRETABLA (NOMBRECAMPO1, ..., NOMBRECAMPOn)

VALUES (VALORCAMPO1, ..., VALORCAMPOn);

Usamos **INSERT INTO** luego el nombre de la tabla, detallamos los nombres de los campos entre paréntesis y separados por comas y luego de la cláusula **VALUES** colocamos los valores para cada campo, también entre paréntesis y separados por comas.

Las cantidades de columnas y valores deben ser iguales. Si una columna no se especifica, le será asignado el valor por default. Los valores especificados en la sentencia INSERT deberán satisfacer todas las restricciones aplicables. Si ocurre un error de sintaxis o si alguna de las restricciones es violada, no se agrega la fila y se devuelve un error.

```
-- PERMITE INSERTAR REGISTRO EN CAMPO PRIMARY KEY IdAlumno
SET IDENTITY_INSERT Alumno ON
```

Ejemplo:

INSERT INTO Alumno (IdAlumno, Documento, Nombre, Apellido, FechaNacimiento,

Ciudad) **VALUES** (1, 3454355,'Miguel', 'Garcia', '01-01-2000', DEFAULT)

Sentencia UPDATE

Decimos que actualizamos un registro cuando modificamos alguno de sus valores.

Para modificar uno o varios datos de uno o varios registros utilizamos **UPDATE** (actualizar).

Por ejemplo, en nuestra tabla Alumno queremos cambiar todas las ciudades pro ‘Medellín’, por

UPDATE Alumno **SET** Ciudad='Medellín';

Utilizamos **UPDATE** junto al nombre de la tabla y **SET** junto con el campo a modificar y su nuevo valor. El cambio afectará a todos los registros.

Podemos modificar algunos registros, para ello debemos establecer condiciones de selección con **WHERE.**

Por ejemplo, necesitamos cambiar la ciudad de residencia por Medellín para los alumnos que tengan como ciudad Envigado.

UPDATE Alumno **SET** Ciudad='Medellín’ **WHERE** Ciudad='Envigado';

Si Microsoft SQL Server no encuentra registros que cumplan con la condición del **WHERE** no se modifica ningún registro de la tabla **Alumno**.

Las condiciones no son obligatorias, pero si omitimos la cláusula **WHERE** la actualización afectará a todos los registros y puede causar un incidente.

También podemos actualizar varios campos en una sola instrucción:

UPDATE Alumno **SET** Ciudad='Medellín', Pais= 'Colombia' **WHERE** Cedula= '12345'

Para ello colocamos **UPDATE** el nombre de la tabla, **SET** junto al nombre del campo y el nuevo valor y separado por coma, el otro nombre del campo con su nuevo valor.

Sentencia DELETE (cuidado con esta sentencia)

Para eliminar los registros de una tabla usamos el comando **DELETE**

DELETE FROM Alumno;

Muestra un mensaje indicando la cantidad de registros que se han eliminado.

Si no queremos eliminar todos los registros, sino solamente algunos, debemos indicar cuál o cuáles, para ello utilizamos el comando **DELETE** junto con la cláusula **WHERE** con la cual establecemos la condición que deben cumplir los registros a borrar.

Por ejemplo, queremos eliminar un registro cuyo nombre de usuario es Jairo:

DELETE FROM Alumno **WHERE** Nombre='Jairo';

Si solicitamos el borrado de un registro que no existe, es decir, ningún registro cumple con la condición especificada, ningún registro será eliminado.

Tengan en cuenta que si no colocamos una condición (**WHERE**), se eliminan todos los registros de la tabla nombrada.

Sentencia TRUNCATE

Vimos que para borrar todos los registros de una tabla se usa **DELETE** sin condición **WHERE**

También podemos eliminar todos los registros de una tabla con **TRUNCATE TABLE**.

Por ejemplo, queremos vaciar la tabla Alumno usamos:

TRUNCATE TABLE Alumno;

- La sentencia **TRUNCATE TABLE** vacía la tabla (elimina todos los registros) y conserva la estructura de la tabla.
- La diferencia con **DROP TABLE** es que esta sentencia borra la tabla, **TRUNCATE TABLE** la vacía.

- La diferencia con **DELETE** es la velocidad, es más rápido **TRUNCATE TABLE** que **DELETE** (se nota cuando la cantidad de registros es muy grande) ya que éste borra los registros uno a uno.

- Otra diferencia es la siguiente: cuando la tabla tiene un campo **identity**, si borramos todos los registros con **DELETE** y luego ingresamos un registro, al cargarse el valor en el campo de identidad, continúa con la secuencia teniendo en cuenta el valor mayor que se había guardado; si usamos **TRUNCATE TABLE** para borrar todos los registros, al ingresar otra vez un registro, la secuencia del campo de identidad vuelve a iniciarse en 1.

Ejemplo:

Tenemos la tabla Alumno con el campo Documento definido como identity (1,1), el valor del consecutivo de la tabla va en 2, si borramos todos los registros con **DELETE** y luego insertamos un nuevo registro, el nuevo registro se guardará con el consecutivo 3, pero si vaciamos la tabla con **TRUNCATE TABLE,** al insertar un nuevo registro, el nuevo registro iniciará nuevamente en 1.

Laboratorio para practicar las sentencias INSERT, UPDATE, DELETE, TRUNCATE.

```
Use Master
go
---Crear una base de datos ---
Create database Prematricula
---Usando contexto Universidad
Use Prematricula
go

---Creando La tabla Alumno
Create table Alumno
(
IdAlumno bigint primary key identity(1,1),
Documento bigint not null,
Nombre varchar(150) not null,
Apellido varchar(150) not null,
Ciudad varchar(150) default('Ciudad'),
FechaNacimiento date,
Edad int
)
go



--Insertando unos datos
Use Prematricula
go
Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,
Ciudad)
values
(1, 3454355,'Miguel', 'Garcia', '01-01-2000', DEFAULT)
```

```
go
Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,Ciudad)
values
(2, 3454356,'Ana', 'Alvarez', '01-01-1988','Medellin')
go

Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,Ciudad)
values
(3,56566,'Juan', 'Castro', '08-09-1989','Envigado')
go

Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento,Ciudad)
values
(4,56566,'Juan', 'Ordoñez', '02-03-2001','Itagui')
go

-- Insertar datos provenientes de otra tabla (tabla con la misma estructura)
USE [Prematricula]
GO

-- se crea una nueva tabla Alumno2 y se realiza una subconsulta para insertar datos
desde la tabla Alumno creada al inicio
Insert into Alumno2
select * from Alumno where Nombre='juan'
Go


---Insertando un dato  con fecha que no cumple con la restriccion
Insert into Alumno (IdAlumno,Documento, Nombre, Apellido, FechaNacimiento)
values
(3, 3454334,'Pedro', 'Galindo', '01-05-2002')
go


---Crear la tabla AsignaturaClase
Create table AsignaturaClase
(
IdAsignatura bigint not null primary key,
IdAlumno bigint,
NombreClase varchar(150),
)



--Consultar los datos de la tabla Alumno y AsignaturaClase
Select * from Alumno
Select * from AsignaturaClase



--Insertar datos a AsignaturaClase
Insert into AsignaturaClase(IdAsignatura,IdAlumno,NombreClase)
values
(1,1,'Matematica Avanzada')
go

Insert into AsignaturaClase(IdAsignatura,IdAlumno,NombreClase)
values
```

```
(2,2,'Lógica')
go


---nueva tabla
Create table Clase
(IdClase int identity(1,1) not null primary key,
NombreClase varchar(100)
)
go
---Insertar un dato
Insert into Clase (NombreClase)
values ('Matematicas especiales')
--deshabilitar el identity_insert
Set identity_insert dbo.Clase ON
--Insertar el dato insertando la columna IdClase
Insert into Clase (IdClase,NombreClase)
values (2,'Bases de datos')
--Insertar datos a la nueva tabla
Insert into Clase (IdClase, NombreClase)
values (3,'Ingenieria de Software')

--deshabilitar el identity_insert
Set identity_insert dbo.Clase Off
---Consultando la tabla
Select * from Alumno
```

1. Diseñar varias sentencias para actualizar registros de las tablas creadas
2. Diseñar varias sentencias para eliminar registros de las tablas creadas
3. Crear sentencias para probar la sentencia TRUNCATE en la nueva tabla Alumno2 y Alumno