

Chapitre 4, le modèle de données avec SQL Server Management Studio.

Parfois, il est intéressant, même très utile de concevoir le modèle de la base de données (modèle relationnel) puis de générer le code SQL. C'est le cas de la plupart des SGBD. On se souvient par exemple du SQL Data Modeler du SGBD Oracle.

Pour MS SQL Server, c'est très simple de créer la base de données en utilisant un schéma relationnel.

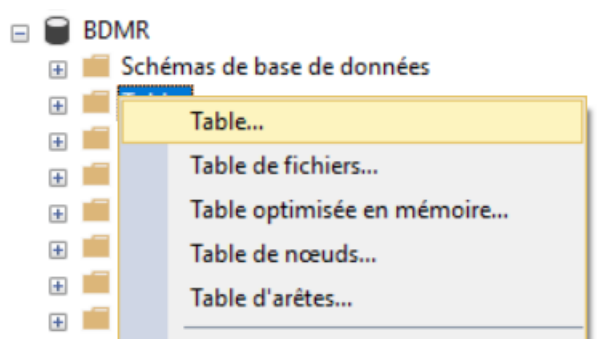
Vous pouvez soit obtenir une modèle relationnel d'une base de données déjà créée ou tout simplement créer un nouveau schéma.

Étape 0 : création de la base de données

Créer une nouvelle base de données avec un nom significatif (ou votre nom) Faites en sorte que vous en soyez le propriétaire.

Étape 2 : Création des tables :

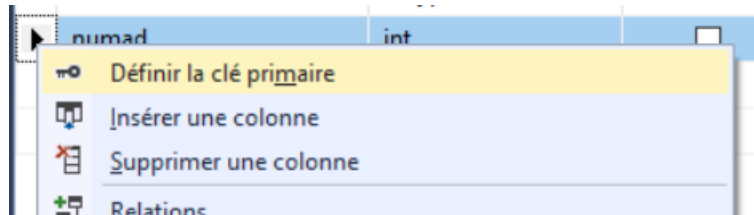
- 1- Faire bouton droit de sur l'onglet Tables de votre BD, puis table



- 2- Créer une table avec les colonnes souhaitées. Les types de données sont ceux que nous avons au chapitre 3

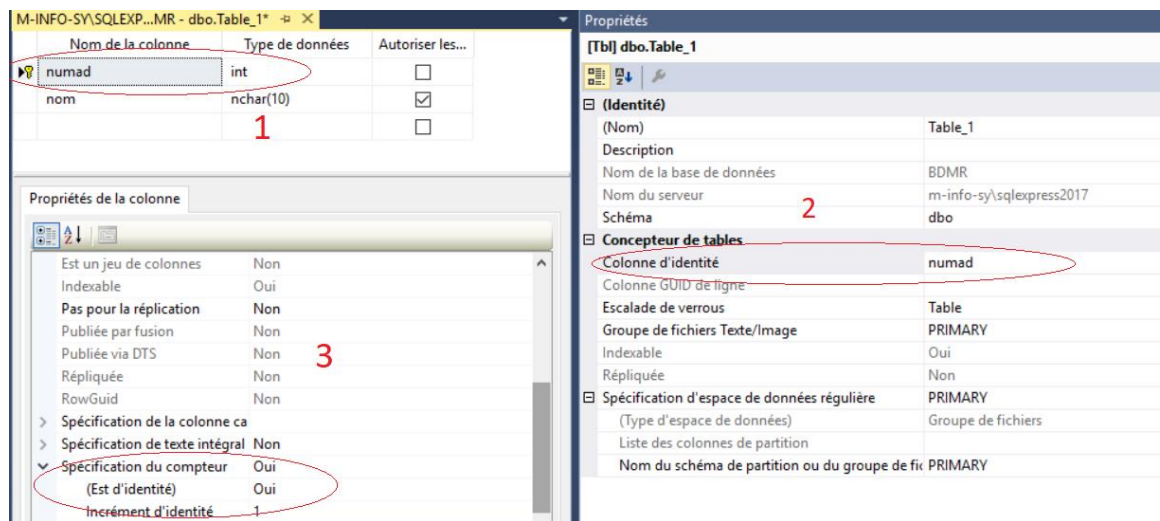
M-INFO-SY\SQLXP...dbo.EtudiantsInfo			
	Nom de la colonne	Type de données	Autoriser les...
?	numad	int	<input type="checkbox"/>
	nom	varchar(50)	<input checked="" type="checkbox"/>
	codep	char(3)	<input checked="" type="checkbox"/>
▶		<input type="text" value=""/>	<input type="checkbox"/>

- 3- Sélectionner la colonne de vous voulez qu'elle soit clé primaire, puis bouton droit et faire : définir la clé primaire : cette étape est obligatoire si vous voulez que la BD soit en 1FN.



Si vous voulez que votre clé primaire soit définie comme IDENTITY, alors :

- a. Positionnez-vous à la colonne de la clé primaire → 1
- b. Vérifiez que dans les propriétés de cette colonne, (à gauche → 2) la propriété « colonne d'identité » soit la clé primaire.
- c. Par la suite, vous allez remarquer que la propriété IDENTITY est bien définie sur la colonne → 3

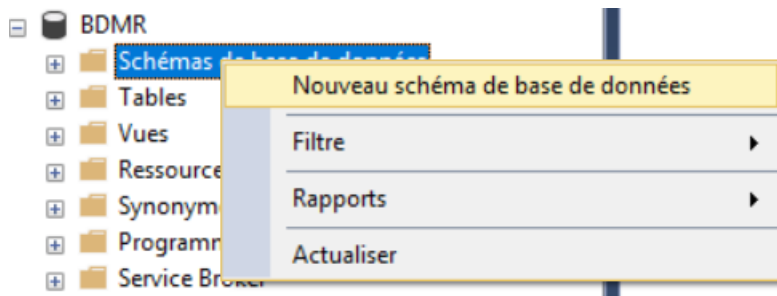


- 4- Enregistrer la table : cliquez sur le bouton enregistrez, puis donnez un nom à votre table. Notre table a pour nom : **EtudiantsInfo**

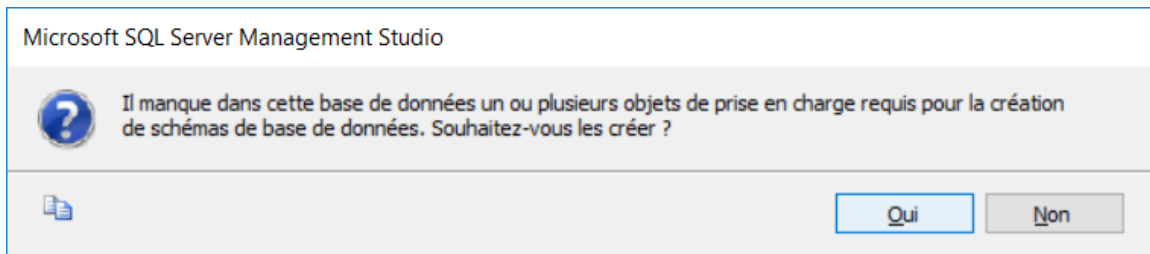
Étape 3, créer le schéma de la BD

Cette étape peut se faire après avoir créé l'ensemble des tables, ou après avoir créé la première table.

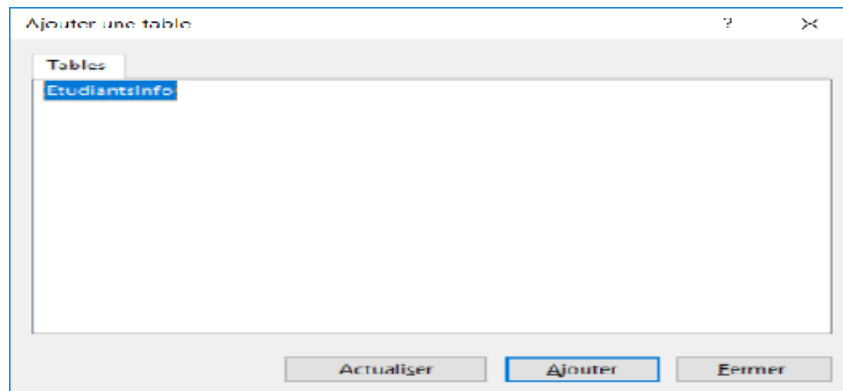
1. Sur le bouton droit de la BD, faire nouveau schéma :



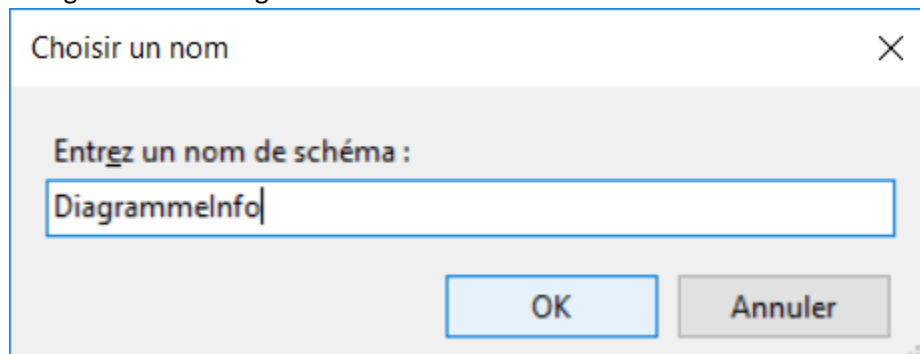
2. Si vous avez ce message, faites OK



3. Ajouter ensuite les tables à votre schéma. Pour l'instant la seule table que nous avons est EtudiantsInfo



4. Enregistrez votre diagramme.



Étape 4 : Définir les relations (la clé étrangère)

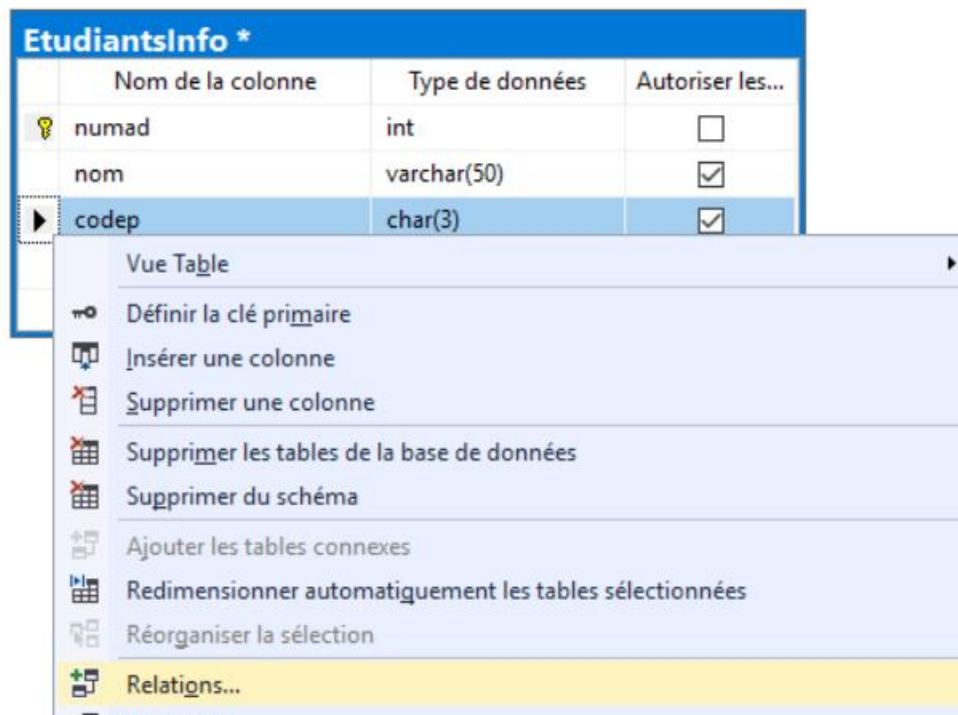
Une fois que vos tables sont créées, ou bien au fur et à mesure que vos tables vont se créer, il sera important de définir les liens entre les tables. Ces liens sont évidemment définis par le concept de Foreign Key ou clé étrangère.

Il est important de rappeler que les types de données et la taille des colonnes qui définissent la clé primaire et la clé étrangère soient les mêmes.

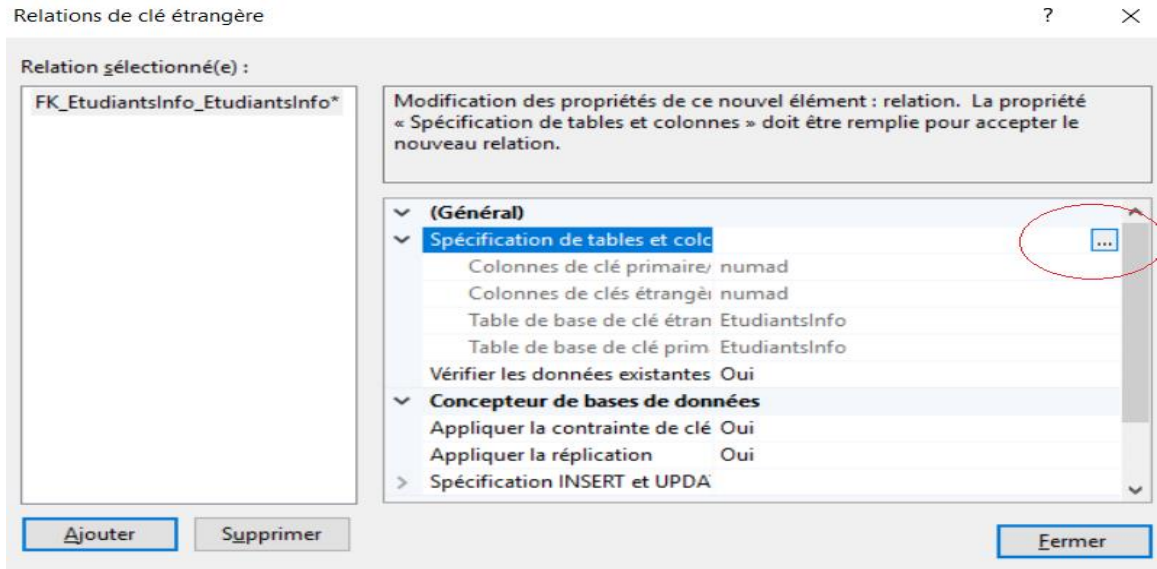
On suppose que la table ProgrammesInfo est créée.

ProgrammesInfo *			
	Nom de la colonne	Type de données	Autoriser les..
🔑	codep	char(3)	<input type="checkbox"/>
	nomprog	varchar(50)	<input type="checkbox"/>
			<input type="checkbox"/>

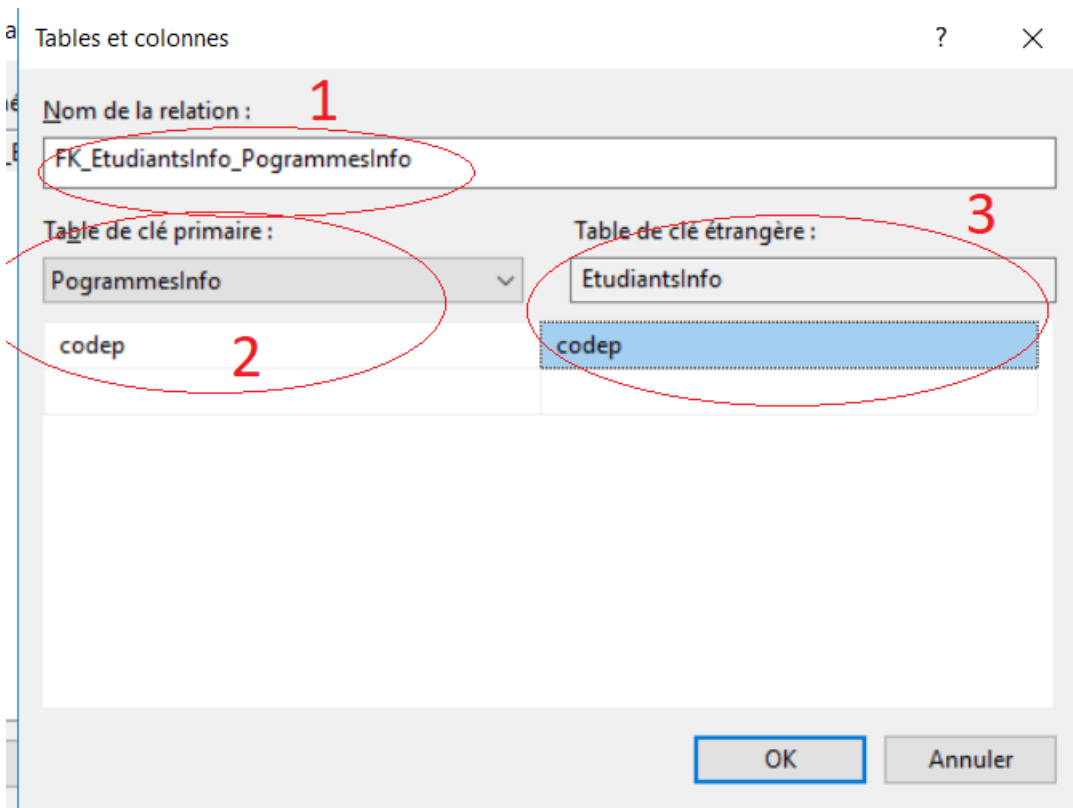
1. Sur la colonne Codep de EtudiantsInfo, (la colonne qui sera clé étrangère), faire Relation comme le montre la figure



2. Une fenêtre s'ouvre, faire Ajouter. Une fenêtre s'ouvre.
3. Dérouler, spécification des tables et des colonnes



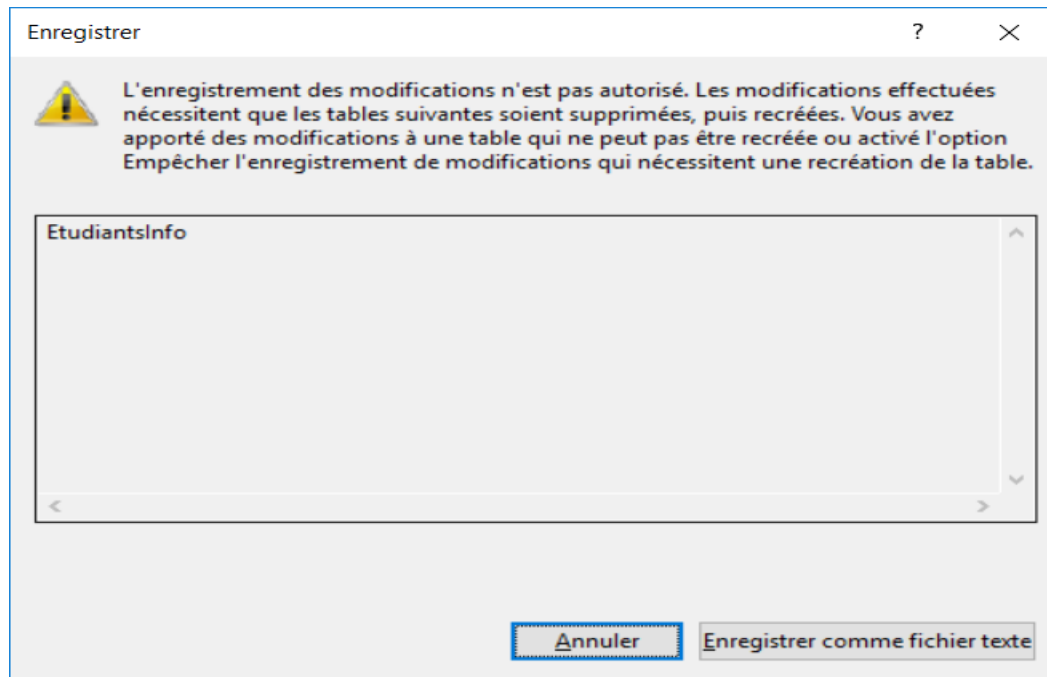
4. Vérifiez que vous avez bel et bien les bonnes colonnes avec les bonnes tables :
 - a. Vous pouvez changer le nom de la contrainte de FK →1
 - b. Vérifier la table et la colonne de la primary Key →2
 - c. Vérifier la table et la colonne de la FK→3



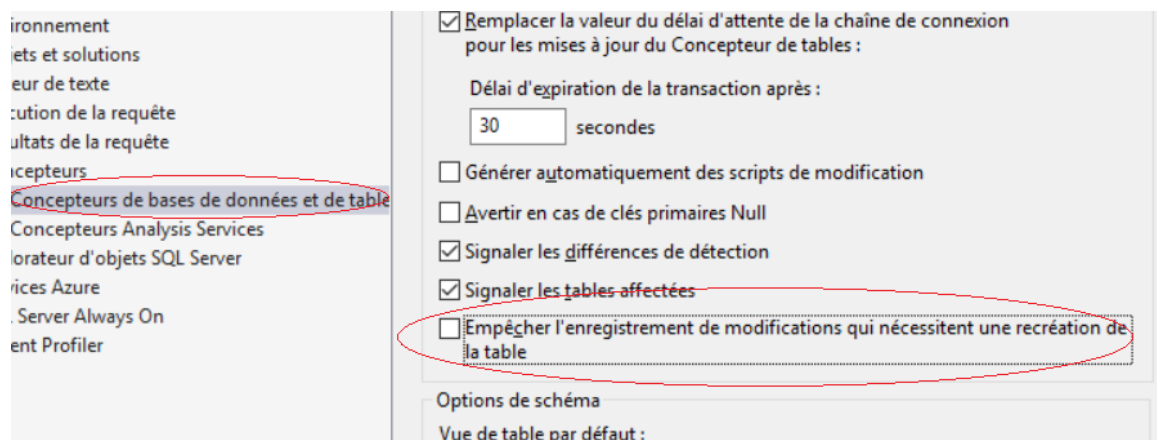
5. Faire OK, puis fermer pour terminer.
6. Enregistrez.

Attention : 

Si au moment d'enregistrer le diagramme vous avez cette fenêtre Alors



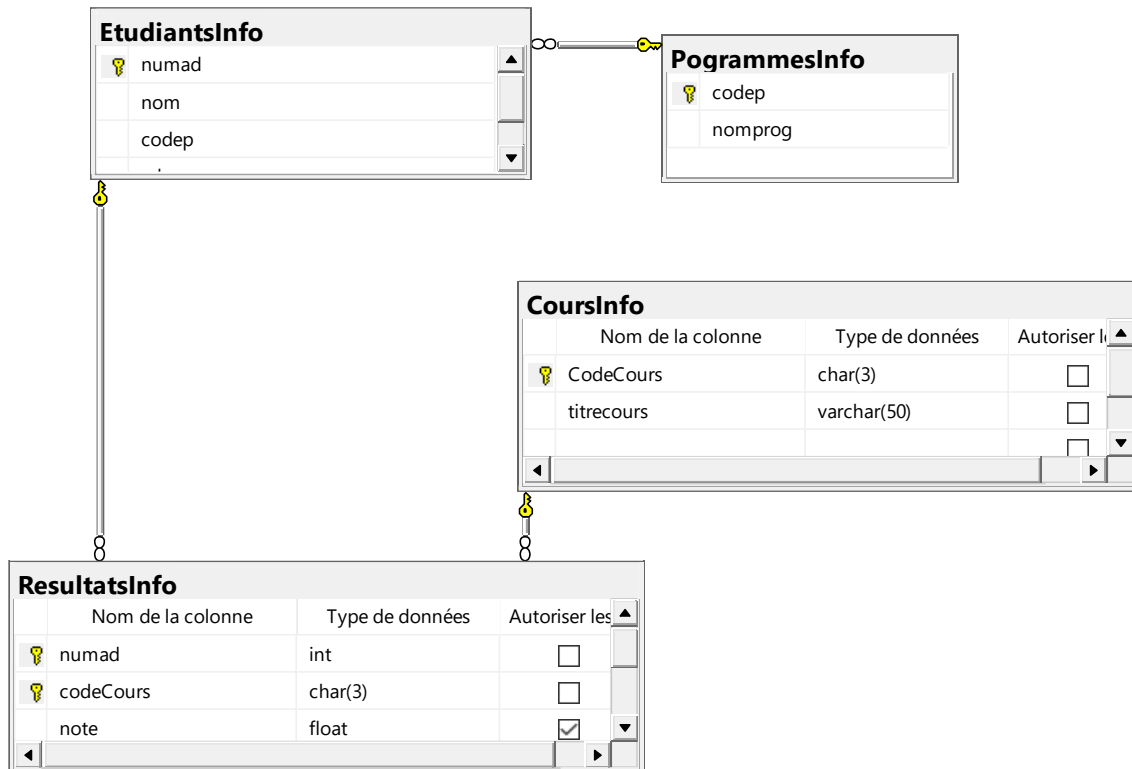
allez à Outils, puis Options, à concepteur de bases de données, décochez la case « Empêcher l'enregistrement » voir figure suivante



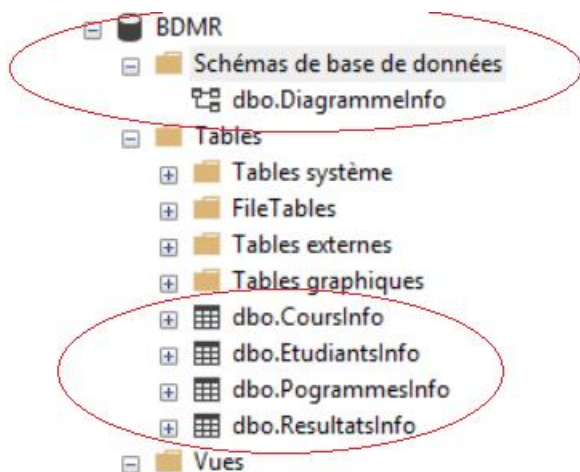
Définir la clé primaire composée

Pour définir une clé primaire composée sur une table, c'est très facile. Il suffit de sélectionner TOUTES les colonnes que l'on souhaite qu'elle soit clé primaire et d'ajouter une clé primaire comme au point 3 de l'étape 1. Il est probable que, les colonnes de

votre clé primaire composées soient des clés étrangères comme dans la plupart des cas. Il faudra alors les définir comme telle.



Il n'est pas nécessaire de générer le code SQL pour créer les tables, puis que celles-ci sont déjà créées



Chapitre 5, éléments du langage Transact-SQL

Définitions

La plupart des SGBDs relationnels offrent une extension du SQL, en y ajoutant des déclarations de variables, des structures de contrôles (alternatives et les répétitives) pour améliorer leurs performances

Transact-SQL ou T-SQL ou TSQL est l'extension du langage SQL pour Microsoft SQL Server et Sybase. Transact-SQL est un langage procédural permettant d'implémenter des fonctionnalités de bases de données que SQL seul ne peut implémenter.

Éléments du langage Transact-SQL :

Les variables et leurs déclarations

- Dans Transact SQL, on utilise le mot réservé DECLARE pour déclarer des variables.
- Les noms de variables sont précédés du symbole @
- Les types de variables, sont les types SQL
- Les variables peuvent être initialisées avec des valeurs en utilisant la fonction SET.

Exemple :

```
DECLARE  
@CHOIX int ;  
SET @CHOIX =1;
```

Les mots réservés : BEGIN ...END

Ces mots réservés permettent de définir un bloc ou un groupe d'instructions qui doivent être exécutées.

Les structures de contrôles

L'alternative :

L' Instruction IF

```
IF Boolean_expression  
    { sql_statement | statement_block }  
[ ELSE  
    { sql_statement | statement_block } ]
```


Ou encore

```
IF Boolean_expression
    { sql_statement | statement_block }
[ ELSE IF Boolean_expression
    { sql_statement | statement_block } ]
[ ELSE
    { sql_statement | statement_block } ]
```

Exemple: (ce bout de code est ce que nous appelons un bloc anonyme)

```
DECLARE
@sal money;
set @sal =40.00;
if @sal = (select salaire from etudiants where numad =8)
    (select * from etudiants where numad =8);
else if @sal=1.33 (select * from ETUDIANTS WHERE NUMAD=6);
else (select * from etudiants);
```

Attention: 

Lorsque vous avez un bloc d'instructions, celui-ci doit être placé entre BEGIN et END.

Exemple1:

```
DECLARE
@code char(3);
begin
set @code = 'tge';
if @code like '%' + (select codep from etudiants where numad =6) + '%'
    (select * from etudiants where numad =6);
else (select * from ETUDIANTS WHERE NUMAD=1);
end;
```

Dans les exemples précédents, remarquez:

- Le bloc d'instructions BEGIN .. END
- Le IF ..ELSE
- Le IF ..ELSE IF ..ELSE
- Comment est construit le LIKE

Exemple2

```
DECLARE
@code char(3);
begin
set @code = 'tge';
    if @code like '%' + (select codep from etudiants where numad =6) + '%'
        (select * from etudiants where numad =6);
    else
    BEGIN
        (select * from ETUDIANTS WHERE NUMAD=1);
        INSERT INTO ETUDIANTS (NOM,PRENOM,SALAIRE,CODEP)
            VALUES('Mosus','Chat',12,'sim');
        update etudiants set salaire = salaire + 5 where codep = 'inf';
    END;
end;
```

Remarquez:

Après le ELSE, nous avons trois instructions à exécuter. Un SELECT, un INSERT et un UPDATE. Ces instructions sont placées entre BEGIN et END :

L'instruction CASE

Syntaxe :

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Ou bien.

```
CASE
    WHEN Boolean_expression THEN result_expression [ ..n ]
    [ ELSE else_result_expression ]
END
```

Exemple 1, case avec un SELECT

```
SELECT  nom, prenom, codep =  
        CASE codep  
          WHEN 'inf' THEN 'Informatique'  
          WHEN 'tge' THEN 'Genie Ele'  
          WHEN 'ele' THEN 'Electronique'  
          ELSE 'Aucun Programme'  
        END, salaire  
FROM etudiants ;
```

Exemple 2, de CASE dans un UPDATE

```
UPDATE etudiants  
SET salaire=  
    ( CASE  
        WHEN (salaire < 5) THEN salaire + 40  
        ELSE (salaire + 20.00)  
      END  
    ) ;
```

La répétitive

La répétitive est implémentée à l'aide de la boucle WHILE.

Syntaxe :

```
WHILE Boolean_expression  
    { sql_statement | statement_block | BREAK | CONTINUE }
```

Exemple

Augmenter le salaire des étudiants, tant que la moyenne est inférieure à 80. Mais si le maximum des salaires dépasse 100 on arrête,

```
BEGIN  
    WHILE (select avg(salaire) from etudiants )<= 80  
    BEGIN update etudiants set salaire = salaire +10;  
    IF(select max(salaire) from etudiants) >100 BREAK;  
    ELSE CONTINUE;  
    END;  
END;
```

Les curseurs :

Les curseurs sont des zones mémoire (mémoire tampon) utilisées par les SGBDs pour récupérer un ensemble de résultats issu d'une requête SELECT.

Pour MS SQL Server, les curseurs sont explicites, ce qui veut dire qu'ils sont associés à une requête SELECT bien précise. Comme par exemple, le curseur CUR1 contiendra le résultat de la requête : SELECT ename, job from emp where deptn=30;

Pour utiliser un curseur, nous avons besoin de le déclarer.

```
DECLARE nomCurseur CURSOR FOR SELECT ... FROM
```

Exemple :

```
DECLARE  
cur1 CURSOR FOR  
SELECT idcircuit, coutcircuit FROM circuits;
```

Pour lire le contenu d'un curseur, on procède comme suit :

- 1- Ouvrir le curseur avec **OPEN**.
- 2- Lire le curseur avec **FETCHINTO** et une boucle WHILE: pour aller chercher chaque enregistrement dans l'ensemble actif, une ligne à la fois, nous utiliserons la commande FETCH. À chaque fois que le FETCH est utilisé, le curseur avance au prochain enregistrement dans l'ensemble actif
- 3- Fermer le curseur avec la commande avec **CLOSE**
- 4- Supprimer la référence au Curseur avec **DEALLOCATE**

La fonction : **@@FETCH_STATUS** : Renvoie l'état de la dernière instruction FETCH effectuée sur un curseur. Elle renvoie 0 si tout s'est bien passé, -1 s'il n'y a plus de lignes, -2 si la ligne est manquante et -9 le curseur ne fait aucune opération d'extraction.

La fonction **@@CURSOR_ROWS**, renvoie le nombre de lignes qualifiantes actuellement dans le dernier curseur ouvert sur la connexion.

Exemple1

```

DECLARE @id int, @cout int;
DECLARE cur1 CURSOR FOR SELECT idcircuit, coutcircuit
FROM circuits;
BEGIN
OPEN cur1 ;
print concat('numero', '---', 'cout');

-- on initialise les variable @id et @cout avec le premier
FETCH(la première ligne)
FETCH NEXT FROM cur1 INTO @id, @cout;
-- Tant que le FETCH se fait normalement
WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT concat(@id, '-----', @cout);
        FETCH NEXT FROM cur1 INTO @id, @cout;
    END;
CLOSE cur1;
DEALLOCATE cur1;
END

```

Exemple 2

```

DECLARE @cout int;
DECLARE Cout_cursor CURSOR FOR SELECT coutcircuit FROM circuits;

BEGIN
OPEN Cout_cursor ;
FETCH NEXT FROM Cout_cursor INTO @cout;

WHILE @@FETCH_STATUS = 0
    BEGIN
        IF @cout < 500 update circuits set coutcircuit = coutcircuit+
(coutcircuit*0.1) WHERE CURRENT OF Cout_cursor ;

        ELSE IF @cout BETWEEN 500 and 900 update circuits set coutcircuit
=coutcircuit+(coutcircuit*0.05) WHERE CURRENT OF Cout_cursor;

        ELSE update circuits set coutcircuit
=coutcircuit+(coutcircuit*0.01) WHERE CURRENT OF Cout_cursor;

        FETCH NEXT FROM Cout_cursor INTO @cout;
    END;
CLOSE Cout_cursor;
DEALLOCATE Cout_cursor;
END

```

Par défaut, les curseurs sont Forward ONLY : ils ne sont pas scrollables. Lorsqu'un curseur est déclaré avec l'attribut SCROLL alors on peut accéder au contenu du curseur par d'autres options de la fonction FETCH. Nous pouvons avoir accès à la première ligne, la dernière ligne, une position absolue, exemple la ligne 3. Position relative à partir d'une position prédéfinie.

```
DECLARE Curmonument SCROLL CURSOR FOR
SELECT nomMonument , nbEtoiles FROM Monuments
ORDER BY nbEtoiles desc;

declare @nom varchar(30), @nb int;
BEGIN
OPEN Curmonument;

print(' la premiere ligne');
FETCH FIRST FROM Curmonument into @nom,@nb;
print concat(@nom, '----', @nb)

print('la dernière ligne');
FETCH LAST FROM Curmonument into @nom,@nb;
print concat(@nom, '----', @nb)

print('la ligne numero 3');
FETCH ABSOLUTE 3 FROM Curmonument into @nom,@nb;
print concat(@nom, '----', @nb)

print('la deuxième ligne après la ligne 3');
FETCH RELATIVE 2 FROM Curmonument into @nom,@nb;
print concat(@nom, '----', @nb)

print('le num immédiatement avant la position courante');
FETCH PRIOR FROM Curmonument into @nom,@nb;
print concat(@nom, '----', @nb)

print('le num qui est deux lignes avant la ligne courante');
FETCH RELATIVE -2 FROM Curmonument into @nom,@nb;
print concat(@nom, '----', @nb)

CLOSE Curmonument;
DEALLOCATE Curmonument;
END
```

Remarque : Nous reviendrons sur les détails concernant les curseurs plus loin dans le cours.

Chapitre 6, les procédures stockées

Définition

Une procédure stockée est un ensemble d'instructions SQL précompilées stockées dans le serveur de bases de données

Avantages à utiliser les procédures stockées

Il existe plusieurs avantages à utiliser des procédures stockées à la place de simple requêtes SQL

- Rapidité d'exécution, puisque les procédures stockées sont déjà compilées.
- Clarté du code : dans un code C#, PHP ou autre, il vaut mieux utiliser l'appel d'une procédure que l'instruction SQL, en particulier lorsque l'instruction SQL est longue et complexe.
- Faciliter le débogage.
- Réutilisation de la procédure stockée.
- Possibilité d'exécuter un ensemble de requêtes SQL
- Prévention d'injections SQL
- Modularité. Facilite le travail d'équipe.

Syntaxe simplifiée de définition d'une procédure stockée avec Transact-SQL

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }  
    [schema_name.] procedure_name  
    [ { @parameter data_type }  
      [ OUT | OUTPUT ] ]  
AS  
{ [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }  
[;]
```

CREATE PROCEDURE : indique que l'on veut créer une procédure stockée.

OR ALTER est optionnel, indique que l'on veut modifier la procédure stockée si celle-ci existe déjà.

@parameter data_type : On doit fournir la liste des paramètres de la procédure avec le type de données correspondant à chacun des paramètres.

[OUT | OUTPUT] : Indique la direction en OUT ou en OUTPUT des paramètres de la procédure. Par défaut les paramètres sont en IN. Lorsque les paramètres sont en IN, il n'est pas nécessaire (c'est même une erreur) d'indiquer la direction.


Text

AS : mot réservé qui annonce le début du corps de la procédure et la fin de la déclaration des paramètres

BEGIN

Bloc SQL ou Transact-SQL

END;

Attention : 

Les paramètres sont précédés du symboles @

Le type de paramètre IN OUT est indiqué uniquement si le paramètre est en OUT ou INOUT (le type IN est par défaut) : La direction IN provoque une erreur si indiquée.

Exemple1 : Tous les paramètres sont en IN. (Insertion)

```
create procedure insertionEtudiants
(
  @pnom varchar(20), @pprenom varchar(30),@psal
  money,@pcodep char(3)
)
AS
begin
insert into etudiants(nom , prenom ,salaire ,codep )
values (@pnom , @pprenom ,@psal ,@pcodep)
end;
```

Exécution d'une procédure dans son SGBD natif (MS SQL Server)

Pour exécuter une procédure stockée, on utilise les commandes execute ou exec. Il faudra fournir la valeur des paramètres.

Exemple :

```
execute insertionEtudiants
@pnom = 'Lenouveau',
@pprenom = 'lenouveau',
@psal=22.5,
@pcodep = 'sim';
```


Même s'il est conseillé de passer les paramètres dans l'ordre de leur apparition dans la procédure, MS SQL Server peut accepter la passation des paramètres dans n'importe quel ordre. Par contre, les noms des paramètres sont très importants. En ce sens SQL Server est contraire d'ORACLE (pour ORACLE c'est l'ordre des paramètres qui est important et non le nom)

On aurait très bien pu faire ceci , le paramètre @nom est fourni en dernier.

```
execute insertionEtudiants  
@pprenom = 'aaaa',  
@psal=22.5,  
@pcodep = 'sim',  
@pnom = 'patate'
```

 Exemple 2 : Les paramètres en IN avec une sortie (SELECT)

```
create procedure lister  
(  
@pcodep char(3)  
)  
AS  
begin  
select nom,prenom from etudiants where @pcodep = codep;  
end;
```

Execution:

```
execute lister  
@pcodep='inf';
```

Exemple 3, utilisation de LIKE dans une procédure stockée

```
create procedure ChercherNom  
(  
@pnom varchar(20)  
)  
AS  
begin
```

```
select * from etudiants where nom Like '%' + @pnom + '%';  
end;
```

Execution

```
execute ChercherNom  
@pnom='Le';
```

Exemple 4 : Procédure avec un paramètre en OUTPUT

```
create procedure ChercherNom2  
(  
  @pnum int,  
  @pnom varchar(20) out  
)  
AS  
begin  
  
  select @pnom = nom  
  from etudiants where numad =@pnum;  
end;  
go
```

Execution

```
declare @pnum int =1;  
declare @pnom varchar(20);  
execute ChercherNom2  
@pnum ,  
@pnom output;  
print @pnom;
```

Les fonctions stockées : Syntaxe simplifiée.

Les fonctions stockées sont des procédures stockées qui retournent des valeurs. Leurs définitions sont légèrement différentes d'une procédure stockée mais le principe général de définition reste le même.

Cas d'une fonction qui ne retourne pas une table

```
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name parameter_data_type
    } ]
)
RETURNS return_data_type


[ AS ]
BEGIN
    function_body
    RETURN scalar_expression
END
[ ; ]
```

Exemple 1, fonction avec paramètres

```
create function compteretudiants(@pcode char(3)) returns int
as
begin
declare @total int;
select @total = count(*) from Etudiants where codep =@pcode;
return @total;
end
go
```

Exécution d'une fonction dans MS SQL Server

Pour exécuter une fonction qui ne retourne pas une table, il faudra utiliser la commande **SELECT**, suivie du nom de la fonction. Il faudra passer les valeurs des paramètres pour la fonction.

Attention : 

1. Pour l'appel des fonction (Exécution), nous avons besoin de préciser le schéma de la BD. Le schéma est toujours : nomUtilisateur.nomObjet .
2. Pour l'instant, tous les objets appartient à l'utilisateur **dbo**.
3. Pour une fonction qui ne retourne pas une table, pas besoin du FROM pour le select.

Remarque : le mappage des utilisateurs aux connexions, sera abordé plus loin.

Pour exécuter la fonction précédente :

```
select dbo.compteretudiants('inf');  
---Pas de clause FROM.
```

Exemple2 : fonction sans paramètres

```
create function compter() returns int  
as  
begin  
declare @total int;  
select @total = count(*) from etudiants;  
return @total;  
end;
```

```
select dbo.compter();  
--pas de clause FROM
```

Cas d'une fonction qui retourne une table.

```
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name  
( [ { @parameter_name parameter_data_type  
      } ]  
)  
RETURNS TABLE  
  
[ AS ]  
RETURN [ ( ) select_stmt [ ) ]  
[ ; ]
```

Exemple

```
Create FUNCTION Cherchertousetudiants  
(@pcodep char(3)) returns table  
AS  
return(  
SELECT nom, prenom  
FROM etudiants  
WHERE @pcodep = codep  
);  
GO
```

L'appel (Exécution) d'une fonction qui retourne une table est différent. Le SELECT dans ce cas, doit utiliser la clause FROM puis que ce qui est retourner est une table. De plus, si la fonction a des paramètres en IN (implicite) il faudra les déclarer et leur affecter des valeurs.

```
declare @codep char(3);  
set @codep='inf';  
select * from Cherchertousetudiants(@codep);
```

Supprimer une fonction ou une procédure :

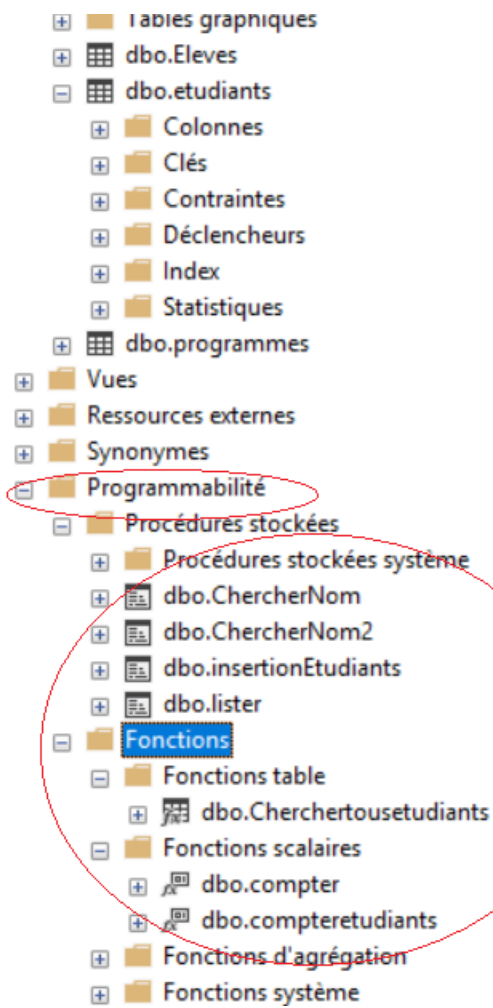
Les fonctions et les procédures sont des objets de la base de données. Ils se détruisent donc avec la commande DROP

```
drop procedure ChercherNom2;
```

```
drop function compteretudiants
```

En conclusion pour les procédures et les fonctions.

- Pour les procédures et les fonctions les paramètres sont précédés de @
- Le type IN est par défaut.
- Lorsque le paramètre est en OUT ou OUTPUT, il faudra l'indiquer clairement.
- Les procédures et fonctions sont terminées par GO. Il n'est cependant pas obligatoire.
- Le mot réservé DECLARE est obligatoire pour déclarer des variables.
- Les fonctions peuvent retourner des tables. Elles ne comportent pas les mots réservés BEGIN et END
- Pour exécuter une procédure il faut utiliser **execute** ou **exec**
- Pour exécuter une fonction il faut utiliser **select** nomuser.nomfonction (valeur paramètres)
- À l'exécution des procédures, l'affectation des valeurs aux paramètres se fait avec = pour les int et la fonction **set** pour les types text.
- Vos fonctions et procédures se trouvent à Programmabilité de la BD



Les procédures stockées et les fonctions : les Templates.

Voici le code généré par SQL Server lorsque vous essayer de créer une procédure ou une fonction

```
=====
-- Template generated from Template Explorer using:
-- Create Procedure (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- =====
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =====
-- Author:          <Author,,Name>
-- Create date:    <Create Date,,>
-- Description:    <Description,,>
-- =====
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
    -- Add the parameters for the stored procedure here
    <@Param1, sysname, @p1> <Datatype_For_Param1, , int> =
<Default_Value_For_Param1, , 0>,
    <@Param2, sysname, @p2> <Datatype_For_Param2, , int> =
<Default_Value_For_Param2, , 0>
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    -- Insert statements for procedure here
    SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

Template function TABLE

```
-- =====
-- Template generated from Template Explorer using:
-- Create Inline Function (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the function.
-- =====
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =====
-- Author:          <Author,,Name>
-- Create date: <Create Date,,>
-- Description:     <Description,,>
-- =====
CREATE FUNCTION <Inline_Function_Name, sysname, FunctionName>
(
    -- Add the parameters for the function here
    <@param1, sysname, @p1> <Data_Type_For_Param1, , int>,
    <@param2, sysname, @p2> <Data_Type_For_Param2, , char>
)
RETURNS TABLE
AS
RETURN
(
    -- Add the SELECT statement with parameter references here
    SELECT 0
)
GO
```


Chapitre 7, les Triggers ou déclencheurs

Définition :

Les triggers sont des procédures stockées qui s'exécutent automatiquement quand un événement se produit. En général cet événement représente une opération DML (Data Manipulation Language) sur une table. Les instructions DML doivent inclure INSERT, UPDATE ou DELETE

Rôle des triggers :

- Contrôler les accès à la base de données
- Assurer l'intégrité des données
- Garantir l'intégrité référentielle (DELETE, ou UPDATE CASCADE)
- Tenir un journal des logs.

Même si les triggers jouent un rôle important pour une base de données, il n'est pas conseillé d'en créer trop. Certains triggers peuvent rentrer en conflit, ce qui rend l'utilisation des tables impossible pour les mises à jour.

Syntaxe simplifiée pour créer un trigger avec une opération DML

Syntaxe simplifiée :

```
CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name
ON { table | view }
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS { sql_statement }
```

AFTER spécifie que le déclencheur DML est déclenché uniquement lorsque toutes les opérations spécifiées dans l'instruction SQL ont été exécutées avec succès.

Un trigger utilisant AFTER va effectuer l'opération DML même si celle-ci n'est pas valide, un message erreur est quand même envoyé.

Utilisez les triggers AFTER avec une ROLLBACK TRANSACTION

Le FOR fait la même chose que AFTER, donc il va quand même insérer ou mettre à jour. Par défaut on utilise AFTER.

INSTEAD OF indique un ensemble d'instructions SQL à exécuter à la place des instructions SQL qui déclenche le trigger.

Au maximum, un déclencheur INSTEAD OF par instruction INSERT, UPDATE ou DELETE peut être défini sur une table ou une vue. → Définir des vues pour des vues pour des INSTEAD OF.

PAS de INSTEAD OF sur des vues avec l'option with CHECK OPTION.

Pour INSTEAD OF pas d'instruction DELETE sur des tables ayant l'option ON DELETE CASCADE (idem pour UPDATE)

Principe de fonctionnement pour les triggers DML.

Lors de l'ajout d'un enregistrement pour un Trigger ...INSERT, le SGBD prévoit de récupérer l'information qui a été manipulée par l'utilisateur et qui a déclenché le trigger. Cette information (INSERT) est stockée dans une table temporaire appelée **INSERTED**.

Lors de la suppression d'un enregistrement, DELETE, le SGBD fait la même chose en stockant l'information qui a déclenché le trigger dans une table temporaire appelée **DELETED**.

Lors, d'une mise à jour, UPDATE l'ancienne valeur est stockée dans la table DELETED et la nouvelle valeur dans INSERTED.

Exemple 1, suppression en cascade

```
create trigger deletescascade on departements
instead of delete as
begin
declare
@code char(3);
    SELECT @code = deptno FROM deleted;
    delete from EmpPermanent where deptno =@code;
    delete from Departements where deptno=@code;
end;
```

Exemple 2

```
create TRIGGER ctrlSalairePermanent on EmpPermanent after update
as
declare
@ancienne money,
@nouvelle money;

BEGIN
select @ancienne = Salaire from deleted ;
select @nouvelle = Salaire from inserted;
IF (@ancienne > @nouvelle)
    rollback;
    RAISERROR (15600,-1,-1, 'pas bon salaire');

END;
```

Exemple 3

Le contenu de la table Emplois

typeEmplois	salaireMin	salaireMax
Analystes	75000,00	140000,00
Directeur	80000,00	200000,00
Finances	45000,00	120000,00
Programmeurs	55000,00	100000,00

Le contenu de la table EmployesBidon

	empno	nom	prenom	salaire	typeEmplois
1	1	Patoche	Alain	100000,00	Directeur
2	2	Leroy	Alain	55000,00	Finances
3	4	Lemieux	Thierry	80000,00	Programmeurs

Le trigger ci-dessous fait en sorte que les salaires des employés respectent la fourchette des salaires définie dans la table Emplois.

```

CREATE TRIGGER CTRLSALAIRES on employesBidon
after INSERT, UPDATE as

DECLARE
@minsalaire money,
@maxsalaire money,
@newsalaire money;
BEGIN
    SELECT @minsalaire = salaireMin from emplois WHERE typeemploi =
(select typeemploi from inserted);

    SELECT @maxsalaire =salaireMax from emplois WHERE typeemploi =
(select typeemploi from inserted);

    select @newsalaire = salaire from inserted;

        if (@newsalaire<@minsalaire or @newsalaire>@maxsalaire)

            rollback TRANSACTION;

        else commit transaction;
end;

```

RAISERROR:

Génère un message erreur défini par l'utilisateur. Le message n'arrête pas le trigger (ce n'est pas comme Raise_Application_error d'Oracle).

RAISERROR(id_message, sévérité ,État , 'Message');

id_message, indique le numéro du message. Ce numéro doit être >50000. Lorsqu'il n'est pas indiqué ce numéro vaut 5000.

Sévérité : indique le degré de gravité associé au trigger, ce niveau de gravité est défini par l'utilisateurs. Ce nombre se situe entre 0 et 25. Les utilisateurs ne peuvent donner que le nombre entre **0 et 18**. Les nombre entre 19 et 25 sont réservés aux membres du groupe sysadmin. Les nombre de 20 à 25 sont considérés comme fatals. Il est même possible que la connexion à la BD soit interrompue.

Si ce nombre est négatif, il est ramené à 1.

Exemples :

Erreur :	Sévérité
Duplication de Clé primaire	14
Problème de FK	16
Problème insertion (valeurs non conformes)	16
Violation de contrainte Check	16
Trigger DML	15 ou 16

Si vous prêtez attention aux messages erreurs renvoyés par le SGBD, vous constaterez qu'ils se présentent sous la forme du RAISERROR vous pouvez vous baser sur ces messages pour fixer le degré de sévérité.

État : utilisé lorsque la même erreur définie par l'utilisateur se retrouve à plusieurs endroits, l'état qui est un numéro unique permet de retrouver la section du code ayant générée l'erreur. L'état est un nombre entre 0 et 255. Les valeurs >255 ne sont pas utilisées. Si négatifs alors ramenés à 0.

Exemples :

```
insert into EmpPermanent values(88,41111,12,'inf');
```

Ici, nous avons un problème de Foreign key puisque le 88 n'est pas un dans la table EmpClg. Pour la première fois, le niveau de sévérité est 16 est l'état est 0.

Vous pouvez également utiliser un try ---catch pour récupérer le message erreur proprement : Dans le cas de l'exemple 2

```

use EmpcIlgDB;
begin try
    begin transaction;
        update EmpPermanent set Salaire =1 where empno =12;
        commit transaction;
    end try

    begin catch
        select ERROR_MESSAGE() as message, ERROR_SEVERITY() as Gravité,
            ERROR_STATE() as etat, @@TRANCOUNT
            if @@TRANCOUNT>0 rollback;
    end catch;

```

	message	Gravité	etat
1	An invalid parameter or option was specified for procedure 'Le salaire ne doit pas être révisé à la baisse'.	15	1

Message : représente le message défini par l'utilisateur. Au maximum 2047 caractères.
 Vous pouvez également laisser le soin au SGBDR d'utiliser ses propres paramètres.

Attention : 

Les triggers sont définis sur une table , ce sont donc des objets de la table, tout comme une colonne, une contrainte...

Activer /désactiver un trigger

Utiliser la commande DISABLE pour désactiver temporairement un trigger

```

DISABLE TRIGGER { [ schema_name . ] trigger_name [ ,...n ] | ALL }
ON { object_name | DATABASE | ALL SERVER } [ ; ]

```

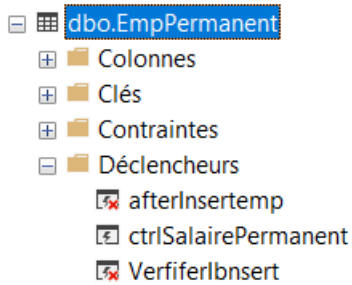
Exemples :

```

disable trigger [dbo].[afterInsertemp] ,[dbo].[VerfiferInsert]
on [dbo].[EmpPermanent];

```

Un trigger désactivé va toujours exister dans le système mais ne fait rien. (sans action).
 Dans Management Studio, il est marqué **en rouge**.



Pour réactiver votre trigger, utiliser la commande ENABLE. Cette commande a la même syntaxe que la commande DISABLE.

```
Enable trigger [dbo].[VerfiferIbnsert] on [dbo].[EmpPermanent];
```

Supprimer un trigger.

Un trigger est un objet de la base de données, il faudra utiliser la commande DROP pour le détruire.

```
DROP TRIGGER [ IF EXISTS ] [schema_name.]trigger_name [ ,.n ] [; ]
```

Exemple :

```
DROP TRIGGER [dbo].[VerfiferInsert];
```

Retour sur la commande CREATE TABLE : ON DELETE CASCADE

Les triggers sont un bon moyen de contrôler l'intégrité référentielle (→ la Foreign KEY) lors de la suppression d'un enregistrement référencé (ou des enregistrements référencés). Si lors de votre conception, vous avez déterminé que les enregistrements liés par la Foreign KEY doivent être supprimés car il s'agit d'un lien de composition, comme dans le cas d'un livre et ses chapitres, c'est-à-dire que lorsqu'un livre est supprimé alors tous les chapitres liés à ce livre doivent être également supprimé, ou encore lorsqu'il s'agit d'une relation de généralisation, alors vous pouvez le faire à la création de table.

Exemple


Voici la création de la table livres

```
create table livres
(
coteLivre char(5),
titre varchar(40) not null,
langue varchar(20) not null,
annee smallint not null,
nbPages smallint not null,
constraint pklivre primary key(coteLivre)
);
```

Voici la table Chapitres

```
create table Chapitres
(
idChapitre char(7) constraint pkChapitre primary key,
nomChapitre varchar(40) not null,
coteLivre char(5) not null,
constraint fkLivre foreign key (coteLivre)
references livres(coteLivre) ON DELETE CASCADE
)
```

Lorsqu'un livre (ou des livres) sont supprimés alors les chapitres de ce livre le sont aussi.

Attention : 

La suppression en cascade à la création des tables n'est pas toujours recommandée sauf si la conception l'exige....

Pour tester :

```
---insertion dans livres--
begin transaction trans1
insert into livres values('IF001', 'Introduction à C#','Français',2017,650);
insert into livres values('IF002', 'SQL pour Oracle 12C','Français',2015,500);
insert into livres values('IF003', 'Oracle pour Java et PHP','Français',2016,700);
insert into livres values('IF004', 'Windows Server 2016','Anglais',2016,1100);
insert into livres values('MA001', 'Algèbre Linéaire','Français',2013,400);
commit transaction trans1;

---insertion dans Chapitres
begin transaction trans2
insert into Chapitres values('IF00101','Pour bien commencer ','IF001');
insert into Chapitres values('IF00102','introduction à la POO ..','IF001');
```



```

insert into Chapitres values('IF00110','les tableaux ','IF001');
insert into Chapitres values('IF00201','Concepts de bases de données ','IF002');
insert into Chapitres values('IF00202','Create table ..','IF002');
insert into Chapitres values('IF00212','les indexs','IF002');
insert into Chapitres values('MA00101','introduction ','MA001');
insert into Chapitres values('MA00102','Les vecteurs','MA001');
insert into Chapitres values('MA0013','les matrices','MA001');
commit transaction trans2;

--pour tester
---en 1
begin transaction trans3;
delete from livres where coteLivre = 'MA001' or coteLivre = 'IF002';
---en 2
rollback transaction trans3;

```

Maintenant, si votre conception initiale, ne doit pas faire de suppression en cascade comme par exemple les employés et les départements, alors opter pour un trigger.

Exemple :

```

USE [Empc1gDB]
GO
/***** Object: Trigger [dbo].[deletescascadeDepartement] *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER trigger [dbo].[deletescascadeDepartement] on
[dbo].[Departements]
instead of delete as
begin
declare
@code char(3);
    SELECT @code = deptno FROM deleted;
    update EmpPermanent set deptno = null where deptno =@code;
    delete from Departements where deptno=@code;
end;

```

En conclusion :

1. Pour garantir l'intégrité de données en général et référentielle en particulier, faites-le par la base de données au CREATE TABLE. Comme les PK, le FK, Les Check...
2. Les triggers sont là pour renforcer l'intégrité des données. Leur avantage est qu'on peut les désactiver au besoin. De plus ils s'exécutent automatiquement (même s'ils sont oubliés).
3. Les procédures stockées sont un excellent moyen pour réduire les risques de briser l'intégrité des données, à condition qu'elles soient utilisées.
4. À moins que ce soit obligé, évitez le ON DELETE CASCADE.

Bonnes pratiques pour les procédures STOCKÉES :

1. Éviter les SELECT*
2. Utilisez des transactions explicites : BEGIN /COMMIT TRANSACTION et privilégiez des transactions courtes. Les transactions longues utilisent un verrouillage plus long.
3. À partir de MS SQL server 2016 vous avez la fonction TRY...CATCH, utilisez là si possible.

```
create procedure insertDept(@code char(3), @nom varchar(30)) as
begin
    begin try
        begin transaction;
        insert into departements values(@code,@nom);
        commit transaction;
    end try
    begin catch
        if(@@TRANCOUNT>0)
            rollback;
    end catch;
end;
```

4. Privilégiez des jointures à la place de sous-requêtes.
5. Restreindre les résultats le plus tôt possible. (WHERE). Éviter des fonctions qui retournent un trop gros nombre de données
6. Des procédures peuvent en appeler d'autres. Vous avez jusqu'à 32 niveaux d'imbrications. Mais faites attention

Chapitre 8, les transactions

Notions de Transactions :

Une transaction est un bloc d'instructions DML exécutés et qui laisse la base de données dans un état cohérent. Si une seule instruction dans le bloc n'est pas cohérente alors la transaction est annulée, toutes les opérations DML sont annulées. Le principe de transaction est implémenté dans tous les SGBDs.

Exemple :

```
begin transaction trans1;  
insert into Departements values('dept', 'resources humaines');  
update EmpPermanent set deptno = 'inf' where empno=1;  
update EmpPermanent set Salaire = 45000 where empno =1;  
insert into Departements values('dept', 'resources humaines');  
commit transaction trans1;
```

Le bloc d'instruction précédent ne va s'exécuter puisque nous avons un problème avec le INSERT, la clé primaire est dupliquée.

Propriétés d'une transaction

Les transactions ont la propriété **ACID**

A : pour Atomicité :

Une transaction doit être une unité de travail indivisible ; soit toutes les modifications de données sont effectuées, soit aucune ne l'est.

C : pour la Cohérence

Lorsqu'elle est terminée, une transaction doit laisser les données dans un état cohérent. Dans une base de données relationnelle, toutes les règles doivent être appliquées aux modifications apportées par la transaction, afin de conserver l'intégrité de toutes les données.

Des fonctionnalités de gestion des transactions qui assurent l'atomicité et la cohérence des transactions. Lorsqu'une transaction a débuté, elle doit se dérouler correctement jusqu'à la fin (validée), sans quoi l'instance du Moteur de base de données annule toutes les modifications effectuées sur les données depuis le début de la transaction. Cette opération est appelée restauration d'une transaction, car elle retourne les données telles qu'elles étaient avant ces modifications.

I : pour Isolement

Les modifications effectuées par des transactions concurrentes doivent être isolées transaction par transaction. Une transaction reconnaît les données dans l'état où elles se trouvaient avant d'être modifiées par une transaction simultanée, ou les reconnaît une fois que la deuxième transaction est terminée, mais ne reconnaît jamais un état intermédiaire.

Des fonctionnalités de verrouillage (verrou ou LOCK) permettant d'assurer l'isolement des transactions.

D : Durabilité

Lorsqu'une transaction durable est terminée, ses effets sur le système sont permanents. Les modifications sont conservées même en cas de défaillance du système

Des fonctionnalités de consignation assurent la durabilité des transactions. Pour les transactions durables, l'enregistrement du journal est renforcé sur le disque avant les validations des transactions. Ainsi, en cas de défaillance du matériel serveur, du système d'exploitation ou de l'instance du Moteur de base de données lui-même, l'instance utilise au redémarrage les journaux des transactions pour restaurer automatiquement toutes les transactions incomplètes jusqu'au moment de la défaillance du système

Pour SQL SERVER certaines transactions sont atomique et donc auto-commit, instruction individuelle qui n'ont pas de BEGIN Transaction.

D'autres transaction sont explicites, dans ce cas elle commence par un : BEGIN TRANSACTION et se termine par un COMMIT Transaction ou un ROLLBACK.

BEGIN TRANSACTION : est comme un point, ou un état où les données référencées par une connexion sont **cohérentes logiquement et physiquement**. En cas d'erreur, toutes les modifications de données effectuées après BEGIN TRANSACTION peuvent être annulées pour ramener les données à cet état de cohérence connu. Chaque transaction dure jusqu'à ce qu'elle soit terminée proprement par un COMMIT ou par un ROLLBACK ;

À chaque BEGIN TRANSACTION, le système incrémente la variable @@TRANCOUNT de 1. Cette variable système retourne le nombre de BEGIN Transaction exécuté pendant la

connexion en cours. Lorsqu'une transaction est comitée (COMMIT) alors

@@TRANCOUNT décrémente de 1. Le ROLLBACK TRANSACTION décrémente la variable **@@TRANCOUNT** jusqu'à 0. (La base de données est dans un état cohérent)

Récupération d'une transaction

Une transaction débute par un **begin transaction** et termine par un **commit** ou un **rollback**.

L'opération **commit** détermine le point où la base de données est de nouveau **cohérente**.

L'opération **rollback** annule toutes les opérations et retourne la base de données dans l'état où elle était au moment du **begin transaction**, donc du dernier **commit**.

Une transaction n'est pas uniquement une unité logique de traitement des données, c'est aussi une **unité de récupération**.

Après qu'une transaction ait terminé avec succès (commit) le SGBDR garantit que les changements seront permanents dans la BD. Cela ne veut pas dire, cependant, que les changements ont été écrits sur le disque dans le fichier physique de la BD. Ils peuvent être encore seulement dans la mémoire de l'ordinateur.

Supposons que 1 seconde après le commit et avant que les changements soient écrits sur le disque, une panne électrique vient tout effacer le contenu de la mémoire et en même temps les changements tout juste 'comités'.

Dans une telle situation, le SGBDR sera quand même capable, au redémarrage, de poursuivre la mise à jour en récupérant la transaction des journaux. Cela est possible à cause d'une règle qui stipule que les journaux sont physiquement sauvegardés sur le disque avant que le commit complète.

Cette double sauvegarde ou redondance des données permet de récupérer non seulement une transaction, mais une BD complète advenant une panne du disque.

Récupération complète de la base de données

Au moment d'une panne électrique ou d'une panne d'ordinateur, le contenu de la mémoire est perdu. L'état des transactions en cours est perdu. Les transactions complétées, mais non écrites sont disponibles dans les journaux.

Au moment du redémarrage du SGBDR, toutes les transactions qui n'ont pas complété seront annulées. Celles qui n'ont pas été sauvegardées dans la BD seront rejouées à partir des journaux.

À intervalle régulier le SGBDR sauvegarde le contenu de ses structures de données en mémoire dans le fichier physique de la BD. Au même moment un enregistrement 'CheckPoint' est ajouté au journal indiquant que toutes les transactions complétées avant le CP sont contenues dans la BD sur le disque.

Pour déterminer quelles transactions seront annulées et quelles transactions seront rejouées, le SGBDR utilise cet enregistrement CP dans le journal.

Supposons la situation suivante







Temps →	tcp		tp	
T1				
T2				
T3				
T4				
T5				
		CheckPoint	Panne	

Figure 1: États de 5 transactions au moment de la panne dans le journal des transactions

- Le SGBDR tombe en panne au temps **tp**;
- Le 'CheckPoint' le plus récent avant la panne est au temps **tcp**;
- T1 a complété avant tcp; donc sauvegardé dans le fichier;
- T2 a débuté avant tcp et a complété après, mais avant la panne à tp; donc pas écrit dans le fichier;
- T3 a débuté avant tcp mais n'a pas complété avant la panne à tp;
- T4 a débuté et complété après tcp; pas écrit dans le fichier;
- T5 a débuté après tcp mais n'a pas complété avant la panne à tp;

Transactions concurrentes

Un SGBDR permet à plusieurs transactions d'accéder la même information en même temps. Pour éviter que les transactions interfèrent l'une avec l'autre, des mécanismes sont nécessaires pour contrôler l'accès aux données.

Transaction A	temps	Transaction B
A(a = 40)		A(a = 40)
Update A set A.a = A.a - 20 where ...;	t1	
	t2	Update A set A.a = A.a - 5 where
Commit A(a = 20)		
		Commit A(a = 15)

Perte de mise à jour

Transaction A	temps	Transaction B
A(a = 40)		A(a = 40)
select @v = A.a from A where ...	t1	
@v = @v - 20 (@v == 20)		
	t2	select @v = A.a from A where
		@v = @v - 10 (@v = 30)
If @v <= 15 rollback		If @v <= 15 rollback
Update A set A.a = @v where ...;	t3	
	t4	Update A set A.a = @v where
Commit A(a = 20)	t5	
	t6	Commit A(a = 30) → A(a = 10)

Les verrous

Le verrouillage est un mécanisme utilisé par le Moteur de base de données SQL Server pour synchroniser l'accès simultané de plusieurs utilisateurs à la même donnée.

Avant qu'une transaction acquière une dépendance sur l'état actuel d'un élément de données, par exemple par sa lecture ou la modification d'une donnée, elle doit se protéger des effets d'une autre transaction qui modifie la même donnée. Pour ce faire, la transaction demande un verrou sur l'élément de données. Le verrou possède plusieurs modes, par exemple partagé ou exclusif. Le mode de verrouillage définit le niveau de dépendance de la transaction sur les données

Le tableau suivant illustre les modes de verrouillage des ressources utilisés par le Moteur de base de données.

Mode de verrouillage	Description
Partagé (S)	Utilisé pour les opérations de lecture qui n'effectuent aucune modification ou mise à jour des données, par exemple une instruction SELECT
Mise à jour (U)	Utilisé pour les ressources pouvant être mises à jour. Empêche une forme de blocage courante qui se produit lorsque plusieurs sessions lisent, verrouillent et mettent à jour des ressources ultérieurement.
Exclusif(X)	Utilisé par les opérations de modification de données, telles que INSERT, UPDATE ou DELETE. Empêche des mises à jour multiples sur la même ressource au même moment.