

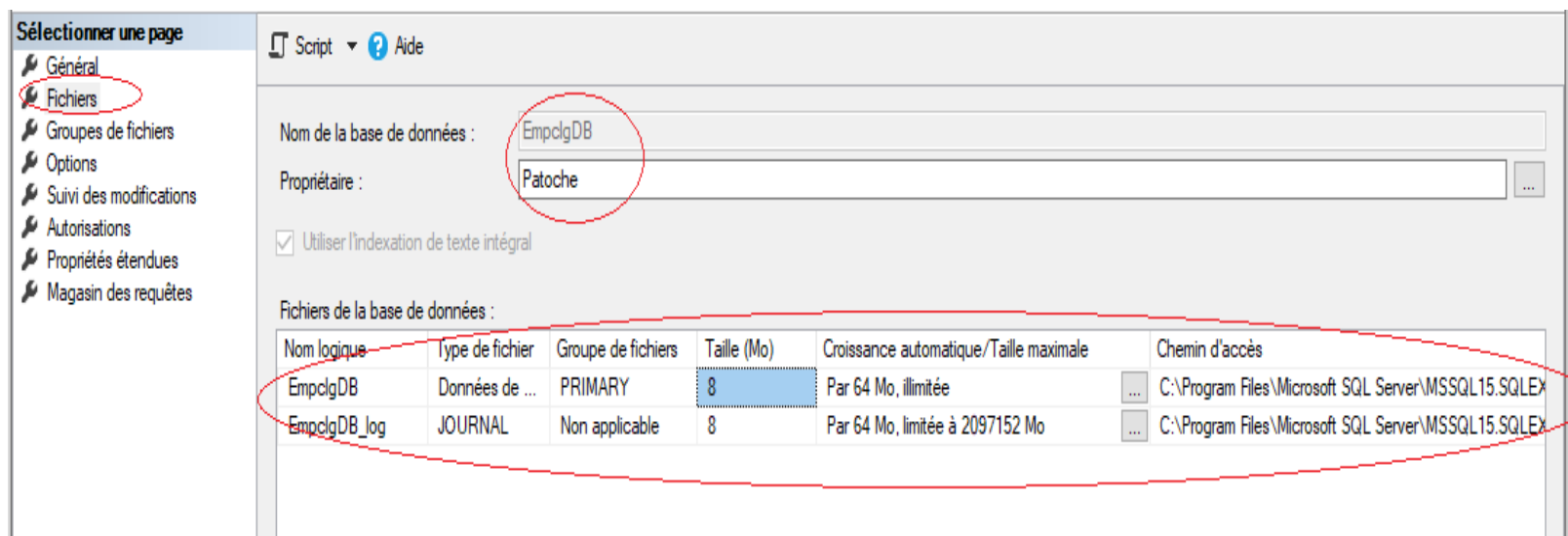
# Programmation de bases de données: Introduction

## Où sont stockés les fichiers de la bases de données ?

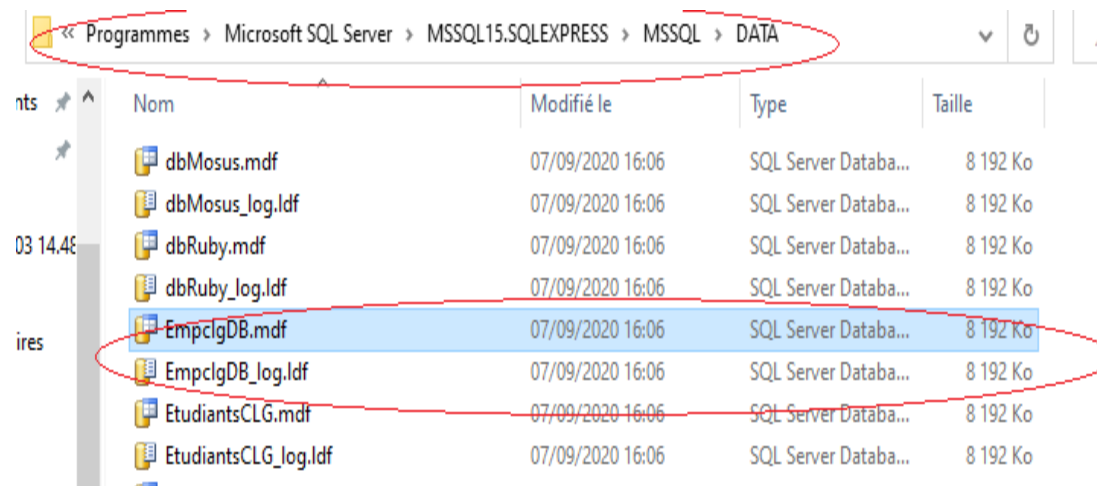
- A ne pas confondre: Fichiers de la base de données et fichiers SQL.
- Vos fichiers SQL, vous pouvez les stocker où vous voulez: sur votre disque, clé USB, etc..
- La commande CREATE DATABASE nomBD implique la création de deux fichiers sur votre disque dur
  - Un fichier nomBD.mdf
  - Un fichier nomBD.ldf
- Les données sont stockées dans un fichier MDF, toutes les transactions sont stockées dans un fichier LDF. (journal des logs nécessaires pour récupérer la base de données à partir d'un certain point).

# Où sont stockés les fichiers de la bases de données ?

- Bouton droit sur votre Base de données, puis propriétés.
- À l'onglet fichiers, remarquez les deux fichiers et leurs types.



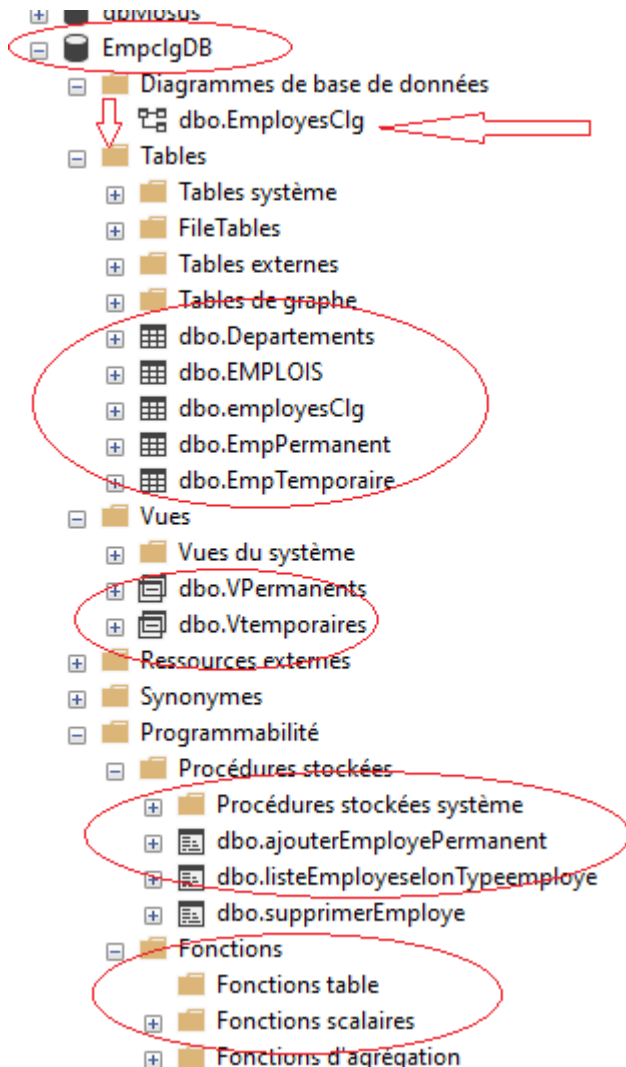
# Où sont stockés les fichiers de la bases de données ?



Nom	Modifié le	Type	Taille
dbMosus.mdf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
dbMosus_log.ldf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
dbRuby.mdf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
dbRuby_log.ldf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
EmpclgDB.mdf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
EmpclgDB_log.ldf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
EtudiantsCLG.mdf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko
EtudiantsCLG_log.ldf	07/09/2020 16:06	SQL Server Databa...	8 192 Ko

- Lorsque vous passez d'un ordinateur A à un ordinateur B , ce sera ces deux fichiers qu'il faudra copier.(de A vers B).
- Au niveau de l' ordinateur B, il faudra joindre le fichier MDF. Vous devez avoir les droits nécessaires. (on va le voir plus loin)

# Présentation de l'interface SSMS



Lorsque vous êtes connectés au serveur, l'ensemble de vos objets (tables, vues, diagrammes, procédures et fonctions stockées) se retrouvent dans la même base de données.

Remarquez :

- La BD EmpclgDB est sélectionnée.
  - Pour cette BD nous avons un diagramme de nom EmployesClg
  - Dans cette BD nous avons 5 tables: Departements, EMPLOIS, ...EmpTemporaire
  - Dans cette BD nous avons deux vues: Vpermanents et Vtemporaires
  - Dans cette BD nous avons 3 procédures stockées: ajouterEmployePermanent .....
  - Nous n'avons pas de fonctions pour le moment.
- Pour écrire des requêtes SQL, vous devez faire: Fichier, nouveau , puis requête avec la connexion actuelle.

# Rappels SQL

- Toutes les commandes que vous avez apprises lors de votre cours de la session 2 sont valables.
- Les types de données de MS SQL server sont différents. **Voir les détails sur le site du cours**

Types	Explications
Varchar(n)	Les types caractères de longueur variable. Lorsque n'est pas précisé il vaut 1
Char(n)	Les types caractères de longueur fixe. Lorsque n'est pas précisé il vaut 1
Date	Définit une date dans SQL Server
Bigint, int, smallint, tinyint	représentent des entiers signés stockés respectivement dans 8,4,2 et un octet.
<b>decimal</b> [ ( p[,s] ) ]	p, Nombre total maximal (incluant le s)de chiffres décimaux à stocker. s est le nombre de chiffres après la virgule
Money	Données monétaires

# La propriété IDENTITY d'une colonne

- Vous pouvez mettre en œuvre des colonnes d'identification à l'aide de la propriété IDENTITY. Ce qui permet de réaliser un auto-incrément sur une colonne.
- En général, la propriété IDENTITY se définit sur la clé primaire.
- Si la propriété IDENTITY est définie sur une colonne, alors c'est le système qui insère des données dans cette colonne (pas l'utilisateur).
- La propriété IDENTITY se fait à la création de table.
- L'avantage d'avoir IDENTITY est que la clé primaire ne sera jamais dupliquée. Ce qui est utile lorsque les insertions se font par une interface C#, JAVA, PHP ...
- Exemples (au tableau ou voir le cours théorique)

## La propriété IDENTITY d'une colonnes

Parfois, nous souhaitons récupérer le dernier numéro inséré, dans ce cas on utilise:

- `SELECT @@IDENTITY`: Permet de retourner la dernière valeur insérée **pour l'ensemble** de la base de données durant la session en cours.
- `SELECT IDENT_CURRENT ('nomTable')` Permet de retourner le dernier numéro inséré d'une table.
- Exemples (au tableau ou voir le cours théorique)



## La propriété IDENTITY d'une colonnes

- SET IDENTITY\_INSERT nomTable ON: permet de faire une insertion manuelle de la colonne dans la table. Dans ce cas vous **perdez la séquence**.
- SET IDENTITY\_INSERT nomTable OFF permet de revenir à l'insertion automatique dans la colonne. On reprend la séquence après IDENTITY\_INSERT ON
- Exemples

# Rappels SQL

- À des exceptions près, Les commandes ALTER Table sont presque identiques avec ORACLE
- Les commandes DML sont identiques avec ORACLE. Pour les insertions, SQL SERVER offre une insertion multiples

**INSERT INTO Departements VALUES**

**('inf','Informatique'),**

**('rsh','Ressources humaines'),**

**('ges','Gestion'),**

**('rec','Recherche et developpement');**

- Le COMMIT et le ROLLBACK ont le même rôle, mais n'ont pas la même syntaxe. Pour faire un COMMIT il faut d'abord faire un: **BEGIN TRANSACTION**
- Pour le SELECT, SQL Server offre: SELECT TOP N pour chercher les N premières lignes. Avec ORACLE il fallait faire une sous-requête avec un ROWNUM

# Points clés

- › Les commandes DDL (Data Definition Language) sont les même qu'avec Oracle que vous avez vu en session2. Seul les types de données changent. Dans la commande DROP TABLE vous n'avez pas CASCADE CONSTRAINTS
- › MS SQL Server permet de faire une auto incrémentation de la clé primaire avec la propriété IDENTITY.
- › Les commandes DML (Data Manipulation Language ) sont les mêmes qu'avec Oracle.
- › Les transactions : Pour faire un COMMIT vous avez besoin de d'avoir un BEGIN TRANSACTION. Le ROLLBACK se fait jusqu'au dernier BEGIN TRANSACTION.
- › Il y a un chapitre sur les transactions plus loin.
- › La Commande SELECT se fait de la même façon qu'avec ORACLE. Les jointures se font avec INNER JOIN (ou RIGHT /LEFT OUTER JOIN)
- › Pour afficher les N première lignes utiliser SELECT **TOP N**

# Programmation de bases de données: T-SQL, Introduction

## Plan de la séance:

- Introduction
- Les variables
- Les mots BEGIN - END
- Les structures de contrôle
- Exemples
- Points clés

# Introduction

- 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.
- Nous présenterons rapidement les blocs « anonymes ». Les éléments du langage seront utilisés dans les procédures stockées.

# Les variables, définition

- Dans Transact SQL, on utilise le mot réservé **DECLARE** pour déclarer des variables.
- Un seul DECLARE est suffisant si vos déclarations sont suivies par une virgule. La dernière variable déclarée se termine par point virgule.
- 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** .

# Les variables, déclaration

```
DECLARE
```

```
@CHOIX int ;
```

```
SET @CHOIX =1;
```

```
DECLARE
```

```
@variable INT =12,
```

```
@nom VARCHAR(30) ='Patoche',
```

```
@date DATE ='2020-08-21';
```

```
PRINT @variable; PRINT @nom; PRINT @date;
```

# Les variables: Affectation de valeurs

Par le mot réservé SET

```
DECLARE
```

```
@variable INT =12,
```

```
@nom VARCHAR(30),
```

```
@date DATE ;
```

```
SET @nom ='Patoche';
```

```
SET @date ='2020-09-30';
```

```
PRINT @variable; PRINT @nom; PRINT @date;
```



# Les variables: Affectation de valeurs

Par des données provenant de la base de données: **SELECT**

```
USE bdEmployesClg
```

```
DECLARE
```

```
@salaire_min MONEY;
```

```
BEGIN
```

```
SELECT @salaire_min =MIN(SALAIRE) FROM Employes;
```

```
PRINT @salaire_min;
```

```
END;
```

# BEGIN....END

## Les mots réservés : BEGIN ...END

Ces mots réservés permettent de définir un bloc ou un groupe d'instructions qui doivent être exécutées. Un peu comme { } en C#.

# L'instruction IF..ELSE

IF Boolean\_expression

{ sql\_statement | statement\_block }

[ ELSE

{ sql\_statement | statement\_block } ]

# L'instruction IF.. ELSE IF

```
DECLARE
```

```
@vsalaire MONEY;
```

```
BEGIN
```

```
SELECT @vsalaire =AVG(SALAIRE) FROM Employes WHERE codeDep ='inf';
```

```
    IF (@vsalaire>60000)
```

```
        UPDATE Employes SET Salaire = Salaire + 0.01*salaire WHERE codeDep ='inf';
```

```
    ELSE
```

```
        UPDATE Employes SET Salaire = Salaire + 0.02*salaire WHERE codeDep ='inf';
```

```
END;
```

# L'instruction IF.. ELSE IF

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

# L'instruction CASE

CASE input\_expression

    WHEN when\_expression THEN result\_expression [ ...n ]

    [ ELSE else\_result\_expression ]

END

Ou

CASE

    WHEN Boolean\_expression THEN result\_expression [ ..n ]

    [ ELSE else\_result\_expression ]

END

# L'instruction CASE

```
SELECT  nom,prenom, codeDep =  
        CASE  codeDep  
            WHEN 'inf' THEN 'Informatique'  
            WHEN 'rsh' THEN 'Ressources humaines'  
            WHEN 'ges' THEN 'Gestion'  
            WHEN 'rec' THEN 'Recherche et developpement'  
            ELSE 'aucun département connu'  
        END, salaire  
FROM employees
```

**Important:** Une jointure est beaucoup moins couteuse qu'un CASE pour ce cas.

# L'instruction CASE

```
UPDATE employes SET salaire =  
(  
  CASE  
    WHEN (salaire < 25000 ) THEN salaire + 0.05*salaire  
    ELSE (salaire + 0.01*salaire)  
  END  
);
```



# L'instruction WHILE

Syntaxe:

WHILE Boolean\_expression

{ sql\_statement | statement\_block | BREAK | CONTINUE }

Exemple:

BEGIN

WHILE (SELECT AVG (salaire) FROM employes ) <= 150000

BEGIN UPDATE employes SET salaire = salaire +1000;

IF (SELECT MAX (salaire) FROM employes) > 500000  
BREAK;

ELSE CONTINUE;

END;

END;

# L'instruction WHILE

Pour un livre donné, insérer plusieurs exemplaires:

Table Livres:

```
create table Livres (  
  idLivre char(3) constraint pk_livres primary key,  
  anneeEdition int ,  
  maisonEdition varchar(50),  
  auteur varchar(50),  
);
```

Table Exemplaires

```
create table Exemplaires (  
  numEx varchar(20) CONSTRAINT PK_EXEMPLAIRES primary key,  
  idLivre char(3) not null,  
  constraint fk_exemplaires_livres foreign key(idLivre) references Livres(idLivre)  
);
```

# L'instruction WHILE

Pour un livre donné, insérer plusieurs exemplaires:

On obtient: Dans la table Livres:

idLivre	anneeEdition	maisonEdition	auteur
PHP	2020	CLG	Patoche

Dans la table Exemplaires:

numEx	idLivre
EXPHP-1	PHP
EXPHP-2	PHP
EXPHP-3	PHP
EXPHP-4	PHP
EXPHP-5	PHP

# L'instruction WHILE

Le code est: (**attention, ici nous n'avons pas mis en œuvre les Transactions**)

```
begin
declare @idLivre char(3)='PHP',
        @cnt int = 1,
        @anneeEdition int =2020,
        @nbExemplaires smallint =5,
        @maisonEdition varchar(50) ='CLG',
        @auteur varchar(30)='Patoche';

insert into Livres(idLivre, anneeEdition,maisonEdition,auteur)
values(@idLivre,@anneeEdition,@maisonEdition,@auteur);

while @cnt <= @nbExemplaires
begin
insert into Exemplaires(numEx,idLivre) values(concat('EX',@idLivre,'-',@cnt), @idLivre);
set @cnt = @cnt + 1;
end;

end;
```

# Le mot réservé GO

## Le mot réservé : GO

GO ne fait pas partie du SQL. C'est un mot réservé optionnel qui permet de finir un lot d'instructions.

Exemple:

Les procédures et les fonctions stockées sont comme vos fonctions C#. En principe, à moins que ce soit dans un programme principal, vous ne pouvez pas mettre les fonctions et les procédures dans un même fichier.

Si vous finissez votre procédure ou votre fonction par GO dans ce cas, vous pouvez mettre vos procédures et ou fonction dans un même fichier sans qu'il y soit des erreurs. Évidemment ces procédure s'exécutent séparément.

# Points clés:

- Dans Transact-SQL la déclaration de variable se fait par le mot DECLARE
- Les variables sont précédées du symbole @
- Les types de variable sont les type SQL de SQL Server
- L'affectation de valeurs aux variables se fait par = ou le SET
- L'affectation de valeurs issues de la base de données à une variable se fait par un SELECT : SELECT @variable = donnée issue de la BD
- Chaque bloc d'instructions commence par BEGIN et fini par END
- Le IF a le même rôle de que ce que vous connaissez en C#, cependant il n' a pas la même syntaxe.
- Le WHILE a le même rôle de que ce que vous connaissez en C#, cependant il n' a pas la même syntaxe.
- Le GO est optionnel (n'est pas du SQL) et permet de terminer un lot d'instructions.
- Surtout, évitez de faire des IF imbriqués à l'infini. Utiliser la fonction RETURN pour sortir d'un IF.

# Programmation de bases de données: Transactions

## Plan :

- Introduction
- La propriété ACID d'une transaction
- COMMIT et ROLLBACK
- TRY ...CATCH

# Retour sur la dernière séance : Point de vue des enseignants:

- › Bon point pour les étudiants: continuez d'être attentifs.
- › Transact-SQL est l'extension du SQL pour SQL Server. il permet d'améliorer la performance du SGBD
- › Transact-SQL c'est essentiellement du SQL auquel on a ajouté des déclarations et manipulation de variables, des structures de contrôles.
- › Les variables sont précédées du symbole @
- › BEGIN –END permettent de délimiter un BLOC.



# Introduction, Transactions

- **Notions de Transactions :**
- Une transaction est un bloc d'instructions DML exécuté 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.
- Pour tous les SGBD certaines transactions sont atomiques et donc auto-commit, instruction individuelle qui n'ont pas de BEGIN Transaction (pour SQL Server)
- Une transaction a la propriété ACID

# La propriété ACID

## A: Atomicité

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

## C: Cohérent

Lorsqu'elle est terminée, une transaction doit laisser les données dans un état cohérent

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

# La propriété ACID

## I: Isolement

Les modifications effectuées par des transactions concurrentes doivent être isolées transaction par transaction → système de verrous.

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.

## 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

# Transactions, principe

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**.

# Transactions, principe

Définition: la variable @@TRANCOUNT contient le nombre de BEGIN TRANSACTION de la connexion actuelle.

BEGIN TRANSACTION: permet de débiter une transaction. Cette instruction incrémente la variable @@TRANCOUNT de 1

COMMIT, permet d'officialiser la transaction. Ce qui permet de diminuer la variable @@TRANCOUNT de 1 ;

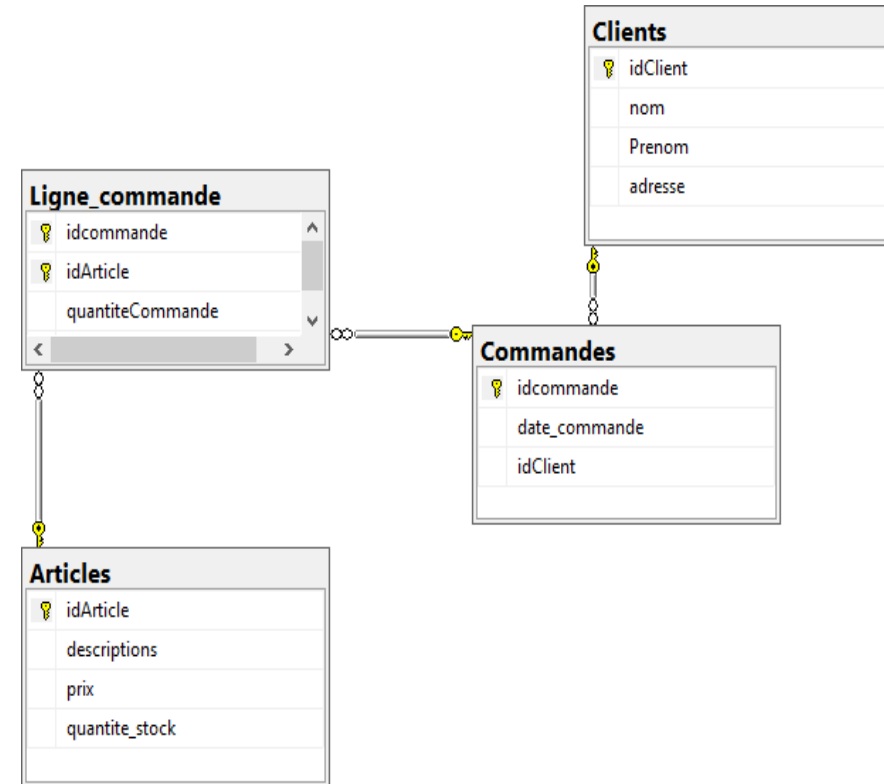
ROLLBACK permet d'annuler une transaction qui n'a pas été COMMITÉ et ramène la variable @@TRANCOUNT à zéro.

Lorsque @@TRANCOUNT >0, cela veut dire qu'il y a encore des transaction en suspend. Il est fort possible que la table sur laquelle porte la transaction soit verrouillée.

Il est très important de ne pas imbriquer des transactions.

# Transactions, Exemple 1

1. Écrire le code Transact-SQL qui permet de faire les opérations suivantes : (comme un TOUT, doit être dans un bloc BEGIN et END)
  - a. On crée une nouvelle commande pour le client numéro 6. (ou un client de votre choix. (Donc on insère dans la table Commandes)
  - b. On insère dans la table Ligne\_commande la commande que l'on vient de créer pour l'article numéro 7 (ou un article de votre choix)
  - c. Si la quantité dans la table Articles n'est pas suffisante : ce qui veut dire que la quantité commandée par le client est plus grande ou égale à la quantité en stock, alors on annule la transaction.
  - d. Sinon( Si la quantité dans la table Articles est suffisante) alors
    - i. On met à jour la table Articles par la nouvelle quantité en stocke. La nouvelle quantité est égale à la quantité initiale moins la quantité commandée.
    - ii. On met à jour le montant de la commande pour cet article dans la table Ligne\_commande.
    - iii. On officialise la transaction



# Transactions, Exemple 1

begin

```
declare @idcommande int,@quantite int,@quaCde int =20,@prix money,@idclient smallint =6,@idArticle int =7;
```

```
select @prix = prix from Articles where idArticle=6;
```

begin transaction

```
INSERT INTO Commandes(datecommande,idClient) values('2024-08-29',@idclient);
```

```
select @idcommande=@@IDENTITY;
```

```
insert into Lignecommande (idcommande,idArticle, quantiteCommande) values (@idcommande,@idArticle,@quaCde);
```

```
select @quantite = quantitestock from Articles where idArticle =@idArticle;
```

```
    if (@quantite <= @quaCde) rollback;
```

```
    else
```

Begin

```
    update Articles set quantitestock =quantitestock-@quaCde where idArticle =@idArticle;
```

```
    update Lignecommande set montant =@quaCde* @prix
```

```
    where idArticle =@idArticle and idcommande =@idcommande;
```

```
    commit; end;
```

end;

## Transactions, Exemple 2

Exemple2:

```
begin transaction
```

```
insert into CategorieTrivia values('y', 'physique', 'mauve');
```

Remarquez que la transaction ne se termine pas correctement. Il n'y a ni COMMIT ni ROLLBACK.

Si vous essayez de faire un `SELECT * FROM CategorieTrivia` vous n'aurez pas de résultat. La table est verrouillée. La BD n'est pas dans un état cohérent.

@@TRANCOUNT =1



# Transactions, principe

## TRY ... CATCH

Il arrive que nous voulons que toutes les opérations à l'intérieur d'une transaction soit TOUTES exécutées si toutes les opérations DML sont correctes, dans ce cas, il faudra les inclure à l'intérieur d'un block TRY CATCH.

Exemple:

# Transactions, Exemple 3

BEGIN TRY

BEGIN TRANSACTION

INSERT INTO Questions(enonce, difficulte, flag) VALUES

('Qui a écrit: les neiges de kilimandjaro ?', 'M', 'N');

INSERT INTO Reponses values (11, ' John Steinbeck', 'N', 2);

INSERT INTO Reponses values (12, 'Ernest Hemingway', 'O', 2);

INSERT INTO Reponses values (12, 'Victor Hugo', 'N', 2);

INSERT INTO Reponses values (14, 'Edgar Frank Codd', 'N', 2);

COMMIT;

END TRY

BEGIN CATCH

ROLLBACK;

END CATCH;

# Programmation de bases de données: Procédures stockées

## Plan de la séance:

- Procédures stockées
  - Définition
  - Avantages
  - Syntaxe
  - Exemples
    - Cas de paramètres en IN
    - Procédure SELECT
    - Cas des paramètres en OUT
- Les fonctions stockées
  - Les fonctions scalaire
  - Les fonctions TABLE

# Retour sur la séance précédente

- Point de vue des étudiants
- Point de vue des enseignants.

# Procédures stockées, définition

## Définition:

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

## Avantages:

- Rapidité d'exécution, puisque les procédures stockées sont déjà compilées.
- Réutilisation de la procédure stockée.
- Possibilité d'exécuter un ensemble de requêtes SQL
- Modularité. Facilite le travail d'équipe.
- Clarté du code
- Prévention d'injection SQL

# Exemple

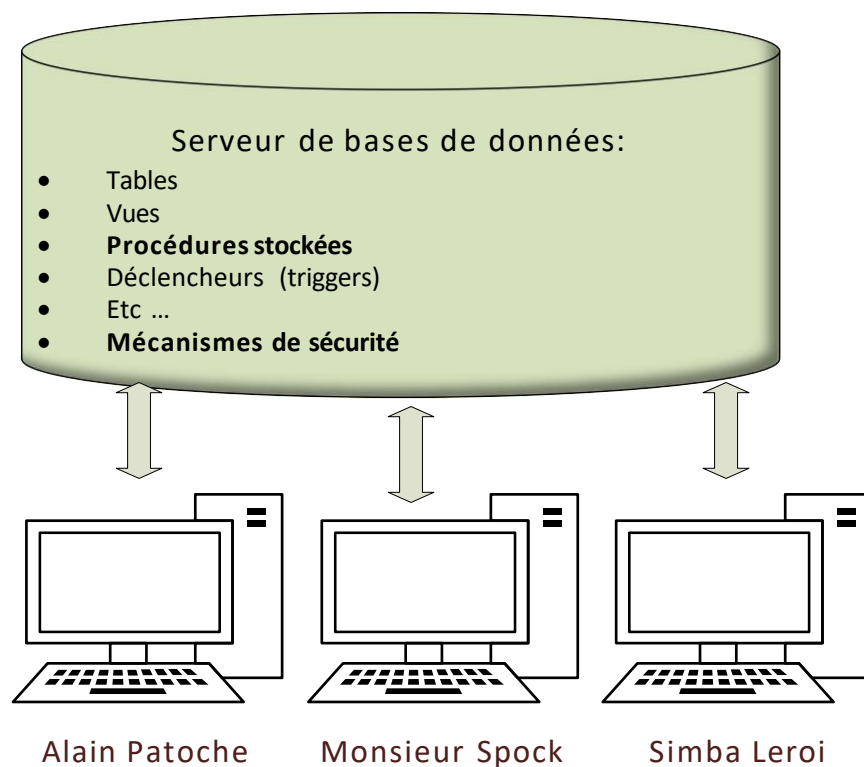
Exemple: Quel code est plus claire ?

```
SqlCommand objCommand = new SqlCommand("select nom, prenom,salaire, nomequipe from joueurs " +  
    "inner join equipes on joueurs.CODEEQUIPE = equipes.CODEEQUIPE ", nomConnection);
```

Ou le code suivant:

```
SqlCommand objCommand1 = new SqlCommand("afficherJoueurs", nomConnection);
```

# Exemple



- Les procédures stockées sont des objets de la BD comme les tables, ou les view.
- Une procédure stockée se crée avec la commande CREATE

# Procédures stockées, syntaxe simplifiée

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



# Procédures stockées, syntaxe simplifiée

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.

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;

# Procédures stockées: Les paramètres sont en IN

Exemple1: 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;
```

# Procédures stockées: Les paramètres sont en IN

## 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 1:

**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

## Procédures stockées: Les paramètres sont en IN

On aurait pu faire ceci (les paramètres ne sont pas passés dans l'ordre de la procédure).

```
EXECUTE insertionEtudiants
```

```
@pprenom='Alain',
```

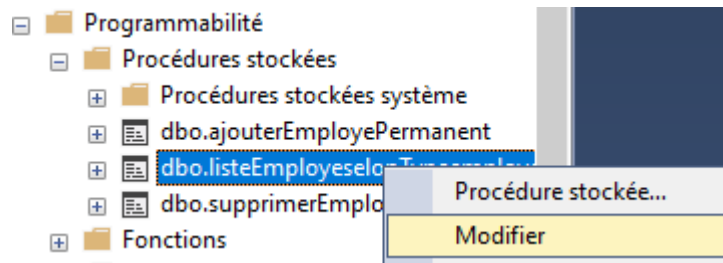
```
@psal=22.5,
```

```
@pcodep='sim',
```

```
@pnom='Patoche';
```

Mais n'oubliez pas, que c'est vous qui allez déboguer une procédure qui ne marche pas. Transmettre les paramètres dans l'ordre de la procédure est ce qui est recommandé.

# Procédure stockée, par l'interface graphique



Après Modifier

```
9 |
10 | ALTER procedure [dbo].[listeEmployeselonTypeemploye]
11 | begin
```

- > Vos procédures stockées se trouvent à l'onglet Programmabilité
- > Pour modifier une procédure stockée soit:
  - On fait un DROP et on la recrée avec CREATE
  - On fait ALTER Procedure le nom de la procédure ..
- Le plus simple est de faire bouton droit sur la procédure puis modifier. Vous avez le code de la procédure en ALTER

# Procédures stockées: Les paramètres sont en IN

Exemple2: Les paramètres sont en IN, **SELECTION**

```
CREATE PROCEDURE lister (@pcodep CHAR(3))
```

```
AS
```

```
BEGIN
```

```
SELECT nom,prenom from etudiants WHERE codep =@pcodep ;
```

```
END;
```

**Execution:**

```
EXECUTE lister
```

```
@pcodep='inf'
```

# Procédures stockées: Les paramètres sont en IN

Exemple3: Les paramètres sont en IN, **SELECTION**

```
CREATE PROCEDURE ChercherNom (@pnom VARCHAR (20))  
AS  
BEGIN  
SELECT * FROM  etudiants WHERE nom Like '%' + @pnom + '%';  
END;
```

Execution

```
EXECUTE ChercherNom  
@pnom='Le'
```

# Procédures stockées: Les paramètres sont en OUT

Exemple4: Un paramètre en OUT

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

## Exécution:

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



# Programmation de bases de données: Procédures stockées (suite)

## Plan de la séance:

- Rappels
- Fonctions stockées
  - Fonction scalaire
  - Fonction table
- Exemples
- Détruire une procédure stockée
- Points clés

# Procédures stockées– Les fonctions

## Les fonctions stockées:

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.

Il existe deux types de fonctions:

- Celles qui retournent un type scalaire (associé à une valeur unique: int, char, varchar..)
- Celles qui retournent une TABLE

La syntaxe pour l'écriture de ces deux types de fonctions n'est pas la même, et ne s'exécutent pas de la même façon.

# Procédures stockées, fonctions scalaires

## 1. Fonction scalaire

```
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
```

```
[ ; ]
```

# Procédures stockées: fonctions scalaires

Exemple

```
CREATE FUNCTION compteretudiants(@pcode CHAR(3)) RETURNS INT  
AS  
  
BEGIN  
  
DECLARE @total INT;  
  
SELECT @total = COUNT(*) FROM Etudiants WHERE codep =@pcode;  
  
RETURN @total;  
  
END;
```

# Procédures stockées: fonctions scalaires

Les fonctions retournent un résultat, la manière d'obtenir le résultat est d'utiliser une requête SELECT.

SELECT nomFonction (liste de valeurs pour les paramètres).

Important: pour l'exécution des fonction, nous avons besoin de préciser le schéma (nom schema • nom de l'objet). Pour l'instant, le nom du schéma est dbo(Data Base Owner)

Donc:

Pour afficher les résultat de la fonction précédente :

```
SELECT dbo.compteretudiants('inf');
```

- Remarquez le dbo
- Il n' y a pas de from (pour Oracle from DUAL)

# Procédures stockées, les fonctions TABLE

2- Fonction TABLE.

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

Il n'y a pas de BEGIN --END

# Procédures stockées, les fonctions TABLE

Exemple:

```
CREATE FUNCTION ChercherEMPLOYE (@code CHAR(3)) RETURNS TABLE
AS
    RETURN
    (
        SELECT nom,prenom,salaire
        FROM employes
        WHERE @code =CodeDep
    );
```

Exécution:

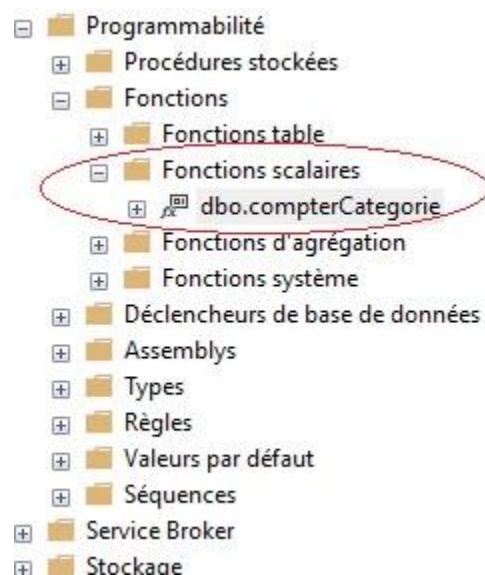
```
SELECT * FROM ChercherEMPLOYE('inf');
```

La fonction TABLE se comporte comme une table.

# Procédures stockées, les fonctions

Les fonctions stockées sont aussi dans l'onglet programmabilité. Vous remarquerez que vous avez les deux types de fonctions: TABLE et SCALAIRE.

Vous pouvez également modifier une fonction par le bouton droit, puis modifier → vous obtiendrez le code de la fonction en ALTER FUNCTION



```
9  
10  
11  
12 ALTER function [dbo].[compterCategorie]()  
13 begin  
14 declare  
15 @totalCategorie int  
16 select @totalCategorie = count(*) from Cat  
17 return @totalCategorie;  
18 end;
```



# Procédures stockées, destruction

Pour détruire une procédure ou une fonction :

```
DROP PROCEDURE nomProcedure
```

```
DROP FUNCTION nomFonction
```

Pour modifier une procédure ou une fonction.

```
ALTER PROCEDURE nomProcedure
```

```
ALTER FUNCTION nomFonction
```

# Points clés:

- › Dans la littérature des BD, le terme « procédures stockées » englobe les procédures et les fonctions.
- › Pour les procédures et les fonctions les paramètres sont précédés de @
- › Le type IN est par défaut. L'indiquer provoque une erreur
- › 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 scalaire il faut utiliser select dbo.nomfonction (valeurs paramètres)
- › Pour exécuter une fonction table c'est SELECT .... FROM nomFonction (valeurs paramètres)
- › Vos fonctions et procédures se trouvent à Programmabilité de la BD
- › Vous pouvez modifier les procédures/fonctions par le bouton modifier de votre SSMS

# Programmation de bases de données: Les triggers ou déclencheurs

## Plan de la séance:

- Les triggers ou déclencheurs
  - Définition et rôle
  - Syntaxe
  - Les tables INSERTED et DELETED
  - Exemples
  - Points clés

# Objectifs de cette séance:

- Retour sur la dernière séance
  - Point de vue des enseignants
  - Point de vue des étudiants
- Rappel, procédures stockées
- Les triggers ou déclencheurs
  - Définition et rôle
  - Syntaxe
  - Les tables INSERTED et DELETED
  - Exemples

# Définition et Rôle

- 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

- 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.

$\pi$

# Syntaxe

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

# Syntaxe

- 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.  
Utiliser ROLLBACK pour annuler une opération invalide
- FOR fait la même chose que AFTER. 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 définie sur une table ou une vue. Définir des vues pour des vues pour des INSTEAD OF.

## Fonctionnement: Les tables INSERTED et DELETED

- 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. On contrôle le nombre de bonnes réponses

```
create or alter trigger CTRLnbBonnerep on ReponseTrivia after insert, update as
begin
declare @idQuestion int,
@totalBonneRep smallint;
select @idQuestion = idQuestion from inserted;

select @totalBonneRep = count(idQuestion) from ReponseTrivia
    where idQuestion =@idQuestion and estBonne ='o';

    if (@totalBonneRep > 1)
        begin
            rollback;
            RAISERROR (15600,-1,-1, 'Limite de une bonne reponse par question');
        end;
end;
```

# Exécution

```
begin transaction
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kba','o',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbo','o',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbi','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kby','n',116);
```

```
commit;
```

```
(1 ligne affectée)
```

```
Msg 15600, Niveau 15, État 1, Procédure CTRLnbBonnerep, Ligne 16 [Ligne de départ du lot 6]
```

```
Paramètre ou option non valide pour la procédure 'Limite d'une bonne reponse par question'.
```

```
Msg 3609, Niveau 16, État 1, Ligne 9
```

```
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.
```

## Exemple2, Trigger qui contrôle le nombre total de réponses par question.

```
create trigger CTRLnbTotalReponses on ReponseTrivia after insert, update as
begin
declare @idQuestion int,
@nbReponse smallint;

select @idQuestion = idQuestion from inserted;
select @nbReponse = count(*) from ReponseTrivia
where idQuestion =@idQuestion

    if (@nbReponse > 4)
    begin
        rollback;
        RAISERROR (15600,-1,-1, 'Limite de 4 reponses par question.');
```

end;

# Exécution

```
begin transaction
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kba','o',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbo','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbi','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kby','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbu','n',116);
```

```
commit;
```

```
(1 ligne affectee)
```

```
Msg 15600, Niveau 15, État 1, Procédure CTRLnbTotalReponses, Ligne 13 [Ligne de départ du lot 6]
```

```
Paramètre ou option non valide pour la procédure 'Limite de 4 reponses par question.'.
```

```
Msg 3609, Niveau 16, État 1, Ligne 12
```

```
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.
```

$\pi$ 

### Exemple 3, on supprime dans la table categorieTrivia et on met à null le idCategorie dans la table questionTrivia (pour les categories supprimées)

```
create trigger updateQuestion on CategorieTrivia instead of delete as
begin
    declare @iCategorie char(1);
    select @iCategorie= idCategorie from deleted;
    update QuestionTrivia set idCategorie= null
    where idCategorie =@iCategorie;

    delete from CategorieTrivia where idCategorie =@iCategorie
end;
```

Pour tester, il faut s'assurer que la colonne idCategorie dans la table QuestionTrivia accepte les valeurs NULL. Sinon on exécute cette instruction en premier:

```
alter table questionTrivia alter column idCategorie char(1) null;
```

Puis, pour tester le trigger:

```
delete from CategorieTrivia where idCategorie = 'h';
```

## Exemple 4

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

    select @empno = empno from deleted
        select @ancienne = Salaire from deleted where empno = @empno;
    select @nouvelle = Salaire from inserted where empno = @empno;
        IF (@ancienne > @nouvelle)
            begin
                rollback;

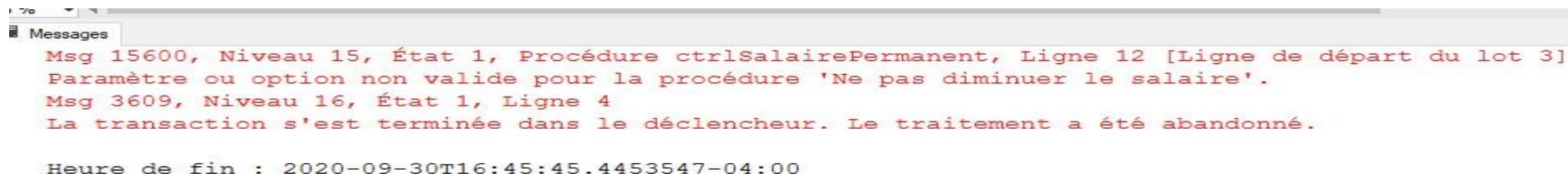
                RAISERROR (15600,-1,-1, 'Ne pas diminuer le salaire');

            end;
END;
```

# Execution

Pour tester :

update EmpPermanent set Salaire =salaire -1 where empno =9;



The screenshot shows a 'Messages' window with the following text:

```
Msg 15600, Niveau 15, État 1, Procédure ctrlSalairePermanent, Ligne 12 [Ligne de départ du lot 3]
Paramètre ou option non valide pour la procédure 'Ne pas diminuer le salaire'.
Msg 3609, Niveau 16, État 1, Ligne 4
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.

Heure de fin : 2020-09-30T16:45:45.4453547-04:00
```

- Dans la table DELETED on retrouve l'ancien salaire de l'employé numéro 9. Le salaire est de 50000
- Dans la table INSERTED on retrouve le nouveau salaire de l'employé numéro 9. Le salaire est 49999
- Ce que nous demandons au trigger c'est de faire un ROLLBACK si le nouveau salaire est plus bas que l'ancien salaire.
- Remarquez:
  - Dans l'instruction UPDATE, nous n'avons pas de BEGIN TRANSACTION, ce qui est normal puis que c'est une transaction autonome.
  - Nous avons un ROLLBACK dans le trigger. (Sans BEGIN Transaction)

# Execution

Ce qui qu'il faut faire c'est: (commiter la transaction à l'extérieur du trigger)

**begin transaction**

**update EmpPermanent set Salaire =salaire -1 where empno =9;**

**commit;**

Ce qui qu'il ne faut absolument pas faire est «ELSE dans le trigger pour un COMMIT »

A ne pas faire dans un trigger. (ce qui est en rouge)

IF (@ancienne > @nouvelle) ROLLBACK **ELSE COMMIT**





# Activer et désactiver un trigger

Parfois, il est nécessaire de désactiver un triggers pour pouvoir effectuer certaines opérations sur la table. Certains triggers pourrait également être en conflit. Au lieu de les détruire, on pourrait simplement les désactiver.

Syntaxe:

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

Exemples:

```
disable trigger afterInsertemp ,VerifierInsert on EmpPermanent;
```

```
disable trigger all on EmpPermanent
```

**Pour Activer un trigger, utilisez: ENABLE.** (c'est la même syntaxe:

```
enable trigger ctrlInsertionPermanent on EmpPermanent;
```

```
enable trigger all on EmpPermanent
```

# RAISERROR

- › RAISERROR est une fonction qui 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).
- › Elle permet à un programmeur de mieux gérer ses messages.
- › RAISERROR(id\_message, sévérité ,État ,'Message');

# RAISERROR

- *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 à l'erreur (ici le trigger), ce niveau de gravité est défini par l'utilisateur.

- 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 négatif ramené à 1
  - Pour les triggers la sévérité est 15 ou 16
  - Violation de contrainte CHECK et FK, sévérité =16
  - Duplication de clé primaire : sévérité 14
- 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. Si négatifs alors ramené à 0.

***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.

# Exécution, avec les messages erreurs pour le trigger de l'exemple 1

```
begin try

    begin transaction

        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('moi','o',114);
        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('toi','o',114);
        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('vous','n',114);
        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('nous','n',114);

    commit;

end try

begin catch

    select ERROR_MESSAGE() as message, ERROR_SEVERITY() as Gravité,
ERROR_STATE() as etat,@@TRANCOUNT as nb_transaction

end catch;
```

message	Gravité	etat	nb_transaction
Paramètre ou option non valide pour la procédure 'Limite d'une bonne reponse par question'.	15	1	0

# Exécution, avec les messages erreurs pour le trigger de l'exemple 2

```
begin try

    begin transaction

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('moi','o',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('toi','n',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('vous','n',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('nous','n',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('nous','n',115);

    commit;

end try

begin catch

select ERROR_MESSAGE() as message, ERROR_SEVERITY() as Gravité,
ERROR_STATE() as etat,@@TRANCOUNT as nb_transaction

end catch;
```

Resultats Messages			
message	Gravité	etat	nb_transaction
Paramètre ou option non valide pour la procédure 'Limite de 4 reponses par question.'	15	1	0

# Points clés, triggers

- Les triggers sont un bon moyen pour garantir l'intégrité des données. Il ne faut pas en abuser.
- MS SQL Server manipule deux types de triggers: AFTER(FOR) et INSTEAD OF
- Vous avez un trigger INSTEAD OF par table.
- Les triggers sont définis sur une table. Lorsque la table est détruite (DROP), le trigger l'est aussi. (ce qui n'est pas le cas d'une procédure ou d'une fonction)
- Il n'y a pas de commande EXECUTE pour déclencher un trigger. C'est l'opération DML qui le déclenche.
- Avec MS SQL Server, lorsque l'opération DML n'est pas valide, le trigger ne l'arrête pas. Il faut faire un ROLLBACK de manière explicite. Dans ce cas, même s'il n'y a pas de BEGIN TRANSACTION, le rollback va se faire.
- Les transactions sont commitées à l'extérieur du trigger
- Les triggers ne sont pas des procédures stockées dans lesquelles vous allez faire vos transactions.

# Programmation de bases de données:

Conclusion: Gestion des  
contraintes d'intégrité et  
automatisation des tâches

## **Plan de la séance:**

- Retour sur la commande CREATE TABLE, l'option ON DELETE CASCADE.
- Conclusion, contraintes d'intégrité
- Travailler sur le laboratoire 3.



## Retour sur la commande CREATE TABLE

- Lors de votre conception, si 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 supprimés, alors vous pouvez le faire à la création de table.
- L'option ON DELETE CASCADE de la commande CREATE TABLE permet de faire cette suppression.

# Exemple

› Syntaxe et exemple:

```
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 pk_livre primary key(coteLivre)  
);
```

# Exemple

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

Attention !! L'option ON DELETE CASCADE est irréversible. Ce n'est pas comme un trigger que vous pouvez **désactiver**.

# Conclusion et bonnes pratiques

- Pour garantir l'intégrité , faites-le par la base de données au CREATE TABLE. Comme les PK, le FK, Les CHECK...C'est la meilleure façon. Pour la PK, lorsque c'est possible, utilisez IDENTITY.
- Donnez un nom significatif à vos contraintes d'intégrité.
- 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).
- Les procédures stockées sont un excellent moyen pour réduire les risques de briser l'intégrité des données, à condition qu'elles soient utilisées.
- À moins que ce soit obligé, évitez tout le temps le ON DELETE CASCADE.
- Dans ce cours, si vous devez utiliser ON DELETE CASCADE, nous vous le demanderons de manière EXPLICITE.

# Conclusion et bonnes pratiques

- Éviter les SELECT \*, c'est lourd
- Au lieu de faire des sous-requêtes, optez pour les jointures si c'est possible. Les SGBDs sont conçus pour les jointures.
- Utilisez le WHERE avant le GROUP BY. C'est plus rapide de grouper un ensemble d'enregistrements restreint.
- Utilisez des transactions explicites : BEGIN TRANSACTION /COMMIT TRANSACTION et privilégiez des transactions courtes. Les transactions longues utilisent un verrouillage plus long. (rappel: une transaction est un ensemble d'instructions **DML** qui doivent-être exécutées comme un tout)
- Utilisez des procédures stockées . Elles sont :
  - Plus rapide à l'exécution,
  - Préviennent les injections SQL
  - Facilite le travail d'équipe
  - Peuvent être réutilisée
  - Facilitent le débogage.
- À partir de MS SQL server 2016 vous avez la fonction TRY...CATCH, utilisez là si possible.
- Ne pas abuser de l'utilisation de triggers. Ils peuvent rentrer en conflit les uns, les autres.

# Programmation de bases de données: Les triggers ou déclencheurs

## Plan de la séance:

- Les triggers ou déclencheurs
  - Définition et rôle
  - Syntaxe
  - Les tables INSERTED et DELETED
  - Exemples
  - Points clés

# Objectifs de cette séance:

- Retour sur la dernière séance
  - Point de vue des enseignants
  - Point de vue des étudiants
- Rappel, procédures stockées
- Les triggers ou déclencheurs
  - Définition et rôle
  - Syntaxe
  - Les tables INSERTED et DELETED
  - Exemples

# Définition et Rôle

- 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

- 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.



$\pi$

# Syntaxe

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

# Syntaxe

- 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.  
Utiliser ROLLBACK pour annuler une opération invalide
- FOR fait la même chose que AFTER. 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 définie sur une table ou une vue. Définir des vues pour des vues pour des INSTEAD OF.

## Fonctionnement: Les tables INSERTED et DELETED

- 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. On contrôle le nombre de bonnes réponses

```
create or alter trigger CTRLnbBonnerep on ReponseTrivia after insert, update as
begin
declare @idQuestion int,
@totalBonneRep smallint;
select @idQuestion = idQuestion from inserted;

select @totalBonneRep = count(idQuestion) from ReponseTrivia
    where idQuestion =@idQuestion and estBonne ='o';

    if (@totalBonneRep > 1)
        begin
            rollback;
            RAISERROR (15600,-1,-1, 'Limite de une bonne reponse par question');
        end;
end;
```

# Exécution

```
begin transaction
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kba','o',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbo','o',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbi','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kby','n',116);
```

```
commit;
```

```
(1 ligne affectée)
```

```
Msg 15600, Niveau 15, État 1, Procédure CTRLnbBonnerep, Ligne 16 [Ligne de départ du lot 6]
```

```
Paramètre ou option non valide pour la procédure 'Limite d'une bonne reponse par question'.
```

```
Msg 3609, Niveau 16, État 1, Ligne 9
```

```
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.
```

## Exemple2, Trigger qui contrôle le nombre total de réponses par question.

```
create trigger CTRLnbTotalReponses on ReponseTrivia after insert, update as
begin
declare @idQuestion int,
@nbReponse smallint;

select @idQuestion = idQuestion from inserted;
select @nbReponse = count(*) from ReponseTrivia
where idQuestion =@idQuestion

    if (@nbReponse > 4)
    begin
        rollback;
        RAISERROR (15600,-1,-1, 'Limite de 4 reponses par question.');
```

end;

# Exécution

```
begin transaction
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kba','o',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbo','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbi','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kby','n',116);
```

```
insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('kbu','n',116);
```

```
commit;
```

```
(1 ligne affectee)
```

```
Msg 15600, Niveau 15, État 1, Procédure CTRLnbTotalReponses, Ligne 13 [Ligne de départ du lot 6]
```

```
Paramètre ou option non valide pour la procédure 'Limite de 4 reponses par question.'.
```

```
Msg 3609, Niveau 16, État 1, Ligne 12
```

```
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.
```

$\pi$ 

### Exemple 3, on supprime dans la table categorieTrivia et on met à null le idCategorie dans la table questionTrivia (pour les categories supprimées)

```
create trigger updateQuestion on CategorieTrivia instead of delete as
begin
    declare @iCategorie char(1);
    select @iCategorie= idCategorie from deleted;
    update QuestionTrivia set idCategorie= null
    where idCategorie =@iCategorie;

    delete from CategorieTrivia where idCategorie =@iCategorie
end;
```

Pour tester, il faut s'assurer que la colonne idCategorie dans la table QuestionTrivia accepte les valeurs NULL. Sinon on exécute cette instruction en premier:

```
alter table questionTrivia alter column idCategorie char(1) null;
```

Puis, pour tester le trigger:

```
delete from CategorieTrivia where idCategorie = 'h';
```



## Exemple 4

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

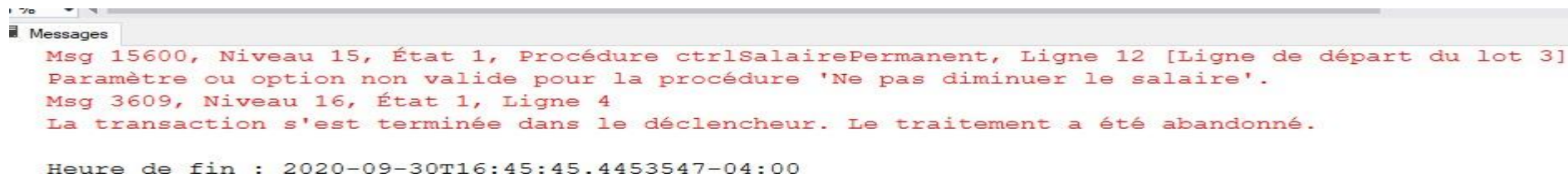
    select @empno = empno from deleted
        select @ancienne = Salaire from deleted where empno = @empno;
    select @nouvelle = Salaire from inserted where empno = @empno;
        IF (@ancienne > @nouvelle)
            begin
                rollback;

                RAISERROR (15600,-1,-1, 'Ne pas diminuer le salaire');
            end;
END;
```

# Execution

Pour tester :

update EmpPermanent set Salaire =salaire -1 where empno =9;



The screenshot shows a 'Messages' window with the following text:

```
Msg 15600, Niveau 15, État 1, Procédure ctrlSalairePermanent, Ligne 12 [Ligne de départ du lot 3]
Paramètre ou option non valide pour la procédure 'Ne pas diminuer le salaire'.
Msg 3609, Niveau 16, État 1, Ligne 4
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.

Heure de fin : 2020-09-30T16:45:45.4453547-04:00
```

- Dans la table DELETED on retrouve l'ancien salaire de l'employé numéro 9. Le salaire est de 50000
- Dans la table INSERTED on retrouve le nouveau salaire de l'employé numéro 9. Le salaire est 49999
- Ce que nous demandons au trigger c'est de faire un ROLLBACK si le nouveau salaire est plus bas que l'ancien salaire.
- Remarquez:
  - Dans l'instruction UPDATE, nous n'avons pas de BEGIN TRANSACTION, ce qui est normal puis que c'est une transaction autonome.
  - Nous avons un ROLLBACK dans le trigger. (Sans BEGIN Transaction)

# Execution

Ce qui qu'il faut faire c'est: (commiter la transaction à l'extérieur du trigger)

**begin transaction**

**update EmpPermanent set Salaire =salaire -1 where empno =9;**

**commit;**

Ce qui qu'il ne faut absolument pas faire est «ELSE dans le trigger pour un COMMIT »

A ne pas faire dans un trigger. (ce qui est en rouge)

IF (@ancienne > @nouvelle) ROLLBACK **ELSE COMMIT**



# Activer et désactiver un trigger

Parfois, il est nécessaire de désactiver un triggers pour pouvoir effectuer certaines opérations sur la table. Certains triggers pourrait également être en conflit. Au lieu de les détruire, on pourrait simplement les désactiver.

Syntaxe:

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

Exemples:

```
disable trigger afterInsertemp ,VerifierInsert on EmpPermanent;
```

```
disable trigger all on EmpPermanent
```

**Pour Activer un trigger, utilisez: ENABLE.** (c'est la même syntaxe:

```
enable trigger ctrlInsertionPermanent on EmpPermanent;
```

```
enable trigger all on EmpPermanent
```

# RAISERROR

- › RAISERROR est une fonction qui 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).
- › Elle permet à un programmeur de mieux gérer ses messages.
- › RAISERROR(id\_message, sévérité ,État , 'Message');

# RAISERROR

- *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 à l'erreur (ici le trigger), ce niveau de gravité est défini par l'utilisateur.

- 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 négatif ramené à 1
  - Pour les triggers la sévérité est 15 ou 16
  - Violation de contrainte CHECK et FK, sévérité =16
  - Duplication de clé primaire : sévérité 14
- 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. Si négatifs alors ramené à 0.

***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.

# Exécution, avec les messages erreurs pour le trigger de l'exemple 1

```
begin try

    begin transaction

        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('moi','o',114);
        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('toi','o',114);
        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('vous','n',114);
        insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('nous','n',114);

    commit;

end try

begin catch

    select ERROR_MESSAGE() as message, ERROR_SEVERITY() as Gravité,
ERROR_STATE() as etat,@@TRANCOUNT as nb_transaction

end catch;
```

message	Gravité	etat	nb_transaction
Paramètre ou option non valide pour la procédure 'Limite d'une bonne reponse par question'.	15	1	0



# Exécution, avec les messages erreurs pour le trigger de l'exemple 2

```
begin try

    begin transaction

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('moi','o',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('toi','n',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('vous','n',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('nous','n',115);

    insert into ReponseTrivia(laReponse,estBonne,idQuestion) values('nous','n',115);

    commit;

end try

begin catch

select ERROR_MESSAGE() as message, ERROR_SEVERITY() as Gravité,
ERROR_STATE() as etat,@@TRANCOUNT as nb_transaction

end catch;
```

Resultats Messages			
message	Gravité	etat	nb_transaction
Paramètre ou option non valide pour la procédure 'Limite de 4 reponses par question.'	15	1	0

# Points clés, triggers

- Les triggers sont un bon moyen pour garantir l'intégrité des données. Il ne faut pas en abuser.
- MS SQL Server manipule deux types de triggers: AFTER(FOR) et INSTEAD OF
- Vous avez un trigger INSTEAD OF par table.
- Les triggers sont définis sur une table. Lorsque la table est détruite (DROP), le trigger l'est aussi. (ce qui n'est pas le cas d'une procédure ou d'une fonction)
- Il n'y a pas de commande EXECUTE pour déclencher un trigger. C'est l'opération DML qui le déclenche.
- Avec MS SQL Server, lorsque l'opération DML n'est pas valide, le trigger ne l'arrête pas. Il faut faire un ROLLBACK de manière explicite. Dans ce cas, même s'il n'y a pas de BEGIN TRANSACTION, le rollback va se faire.
- Les transactions sont commitées à l'extérieur du trigger
- Les triggers ne sont pas des procédures stockées dans lesquelles vous allez faire vos transactions.

# Programmation de bases de données:

Conclusion: Gestion des  
contraintes d'intégrité et  
automatisation des tâches

## **Plan de la séance:**

- Retour sur la commande CREATE TABLE, l'option ON DELETE CASCADE.
- Conclusion, contraintes d'intégrité
- Travailler sur le laboratoire 3.

## Retour sur la commande CREATE TABLE

- Lors de votre conception, si 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 supprimés, alors vous pouvez le faire à la création de table.
- L'option ON DELETE CASCADE de la commande CREATE TABLE permet de faire cette suppression.

# Exemple

› Syntaxe et exemple:

```
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 pk_livre primary key(coteLivre)  
);
```

# Exemple

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

Attention !! L'option ON DELETE CASCADE est irréversible. Ce n'est pas comme un trigger que vous pouvez **désactiver**.

# Conclusion et bonnes pratiques

- Pour garantir l'intégrité , faites-le par la base de données au CREATE TABLE. Comme les PK, le FK, Les CHECK...C'est la meilleure façon. Pour la PK, lorsque c'est possible, utilisez IDENTITY.
- Donnez un nom significatif à vos contraintes d'intégrité.
- 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).
- Les procédures stockées sont un excellent moyen pour réduire les risques de briser l'intégrité des données, à condition qu'elles soient utilisées.
- À moins que ce soit obligé, évitez tout le temps le ON DELETE CASCADE.
- Dans ce cours, si vous devez utiliser ON DELETE CASCADE, nous vous le demanderons de manière EXPLICITE.

# Conclusion et bonnes pratiques

- Éviter les SELECT \*, c'est lourd
- Au lieu de faire des sous-requêtes, optez pour les jointures si c'est possible. Les SGBDs sont conçus pour les jointures.
- Utilisez le WHERE avant le GROUP BY. C'est plus rapide de grouper un ensemble d'enregistrements restreint.
- Utilisez des transactions explicites : BEGIN TRANSACTION /COMMIT TRANSACTION et privilégiez des transactions courtes. Les transactions longues utilisent un verrouillage plus long. (rappel: une transaction est un ensemble d'instructions **DML** qui doivent-être exécutées comme un tout)
- Utilisez des procédures stockées . Elles sont :
  - Plus rapide à l'exécution,
  - Préviennent les injections SQL
  - Facilite le travail d'équipe
  - Peuvent être réutilisée
  - Facilitent le débogage.
- À partir de MS SQL server 2016 vous avez la fonction TRY...CATCH, utilisez là si possible.
- Ne pas abuser de l'utilisation de triggers. Ils peuvent rentrer en conflit les uns, les autres.



# programmation de bases de données

# Les curseurs

- Retour sur les dernières séances:
  - Point de vue des enseignants:
    - Lisez les consignes
    - Lisez les notes de cours (pas uniquement le Power Point)
    - Lisez les exemples et les solutionnaires lorsqu'ils sont disponibles
    - Trop d'absences , surtout dans le groupe 1;
    - Respectez les dates et les modalités de remises
  - Point de vue de l'étudiant
- Les curseurs
  - Définition
  - Syntaxe de déclaration
  - Exemples.

# Les curseurs

- › Définition: Les curseurs sont des zones mémoire 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.
- › Les données sont disponibles dans le curseur au moment de sa déclaration.

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

Un curseur a la même structure qu'une table.

Un curseur a besoin de son propre DECLARE et n'a pas @ devant son nom.

# Les curseurs

- › Lecture du contenu d'un curseur:
  - Pour lire un curseur, il faut l'ouvrir avec la fonction OPEN. (comme un fichier)
  - La commande **FETCH NEXT FROM.....INTO** (un peu comme READ) permet de lire l'enregistrement suivant. (une seule ligne à la fois).
  - On utilise une boucle WHILE pour parcourir l'ensemble des enregistrements du curseur. On arrêter lorsque le curseur est vide.
  - On ferme le curseur(très important)
  - On libère les ressources occupées par le curseur. (très important)

# Les curseurs

› Exemple:

Voici le contenu de la table panierAchat qui représente un panier d'achat dans votre BD stockClg.

	idArticle	idClient	qteAchat
1	1	1	5
2	1	2	15
3	2	1	10
4	3	1	15
5	4	2	10

Lorsque, je déclare le curseur curPanier  
declare curPanier cursor for select idArticle, qteAchat  
from PanierAchat where idClient =1;

idArticle	qteAchat
1	5
2	10
3	15

# Les curseurs

Comment lire le contenu du curseur ? En transférant son contenu dans des variables et on lit les variables) . On procède par étape.

**Étape 0:** On ouvre le curseur avec **OPEN**.

**Étape 1:** on se déplace au premier enregistrement et on met le contenu dans des variables déjà déclarées.

```
fetch next from curPanier into @idArticle,@qtAchat;
```


**Étape 3:** On utilise une boucle WHILE pour parcourir tout le curseur. À chaque itération, le pointeur (rouge) va à l'enregistrement suivant. On arrête lorsqu'il n'y a plus de ligne à lire.

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.


```
WHILE(@@FETCH_STATUS=0)
```

**Étape 4:** On ferme le curseur avec **CLOSE**

**Étape 5:** On libère les ressources occupées par le curseur: **DEALLOCATE**



idArticle	qteAchat
1	5
2	10
3	15



idArticle	qteAchat
1	5
2	10
3	15

# Les curseurs.

- › Par défaut, les curseurs sont Forwrad ONLY, avec un parcours en avant seulement.
- › Lorsque vous récupérer le contenu du curseur dans des variables, ces variables sont utilisées comme des variables ordinaires.
- › Dans l'exemple suivant, lorsque le panier d'un client est supprimé alors la quantité en stock des articles qui étaient dans le panier est mise à jour.

# Les curseurs

```
create or alter procedure deletePanier(@idclient smallint) as
begin
declare @qtAchat int, @idArticle int;
declare curPanier cursor for select idArticle, qtAchat from panierAchat where idClient =@idClient;
open curPanier
fetch next from curPanier into @idArticle,@qtAchat;
    while @@FETCH_STATUS=0
    begin
        update Articles set quantiteStock = quantiteStock + @qtAchat where idArticle =@idArticle;
        fetch next from curPanier into @idArticle,@qtAchat;
    end;
delete from PanierAchat where idClient =@idClient;
close curPanier;
DEALLOCATE curPanier;
end;
```



# Les curseurs.

Curseur scrollable:

- › 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.
- › L'accès au contenu du curseur se fait directement à l'endroit souhaité (accès direct).

# Les curseurs.

Quelques fonctions du curseur scrollable:

- › FETCH FIRST : va à la première ligne du curseur
- › FETCH LAST : va au dernier enregistrement
- › FETCH ABSOLUTE n: va à la nième position dans le curseur
- › FETCH RELATIVE n :pas à la nième ligne après la position courante (n peut-être positif ou négatif).
- › FETCH PRIOR :va à la ligne immédiatement avant la position courante

# Les curseurs

```
declare curempPermanent scroll cursor for select nom, prenom, salaire from employeesClg C  
inner join EmpPermanent P on c.empno =P.empno order by salaire desc;
```

Le contenu du curseur est le suivant:

	nom	prenom	salaire
1	Patoche	Alain	349998,00
2	Monsieur	Spock	200000,00
3	Saturne	Lune	120000,00
4	Fafar	Kelly	55000,00
5	Postras	Fred	45001,00
6	Lechat	Simba	45000,00

Le code de la page suivante, va me permettre d'accéder directement aux lignes du curseur.

# Les curseurs

```
begin
declare @nom varchar(30),
        @prenom varchar(30),
        @salaire money;

declare curempPermanent scroll cursor for select nom, prenom, salaire
from employesClg C inner join EmpPermanent P on c.empno =P.empno order by salaire desc;

open curempPermanent;

        print(' la premiere ligne');

        fetch first from curempPermanent into @nom,@prenom, @salaire;
        print concat (@nom,'----', @prenom,'----', @salaire);

close curempPermanent;
deallocate curempPermanent;
end;
```

# Les curseurs

Le résultat sera: Patoche----Alain----349998.

Avec le même code, mais on met:

```
fetch absolute 3 from curempPermanent into @nom,@prenom, @salaire;
```

**Le résultat sera: Saturne----Lune----120000.00**

Nous sommes à la 3ème ligne, puis on fait relative de 2

```
fetch absolute 3 from curempPermanent into @nom,@prenom, @salaire;
```

```
fetch relative 2 from curempPermanent into @nom,@prenom, @salaire;
```

Le résultat sera: (on avance de 2 par rapport à Saturne.

**Poitras----Fred----45001.00**

```
fetch absolute 3 from curempPermanent into @nom,@prenom, @salaire;
```

```
fetch relative -1 from curempPermanent into @nom,@prenom, @salaire;
```

Le résultat sera: On recule de 1, par rapport à saturne

**Monsieur----Spock----200000.00**

# Les curseurs.

## Conclusion.

- Les curseurs sont des variables qui se déclarent avec la commande SELECT:  
**DECLARE nomCurseur CURSOR FOR SELECT**
- Pour lire un curseur, il faut d'abord l'ouvrir. Tout curseur ouvert doit-être obligatoirement fermé. Les curseurs ouverts non fermés provoquent des erreurs graves.
- La méthode FETCH ..NEXT permet de passer à l'enregistrement suivant. Dans ce cas, on parle d'accès **séquentiel**.
- Par défaut, les curseurs ne sont pas scrollables.
- Si le curseur est scrollable alors il est possible d'accéder directement aux lignes de celui-ci. On parle d'accès **direct**.
- N'utilisez pas des curseurs pour faire un simple SELECT. Ce n'est pas utile et c'est déconseillé.
  - Ça occupe des ressources. Si vous oubliez le deallocate → problème
  - Si vous oubliez de fermer le curseur → problème.

# programmation de bases de données

# Sécurité des données

- › Introduction
- › Menaces courantes
- › Rôles de serveur
- › Rôles de base de données



# Sécurité des données

## Introduction

- Aucune méthode universelle n'existe pour créer une application cliente SQL Server sécurisée.
- Chaque application est unique au niveau de sa configuration, de son environnement de déploiement et de ses utilisateurs.
- Une application relativement sécurisée lors de son déploiement initial peut devenir moins sécurisée avec le temps.
- Il est impossible d'anticiper avec précision sur les menaces qui peuvent survenir dans le futur.

# Sécurité des données

## Menaces courantes:

La sécurité peut être envisagée comme une chaîne dans laquelle un maillon manquant compromet la solidité de l'ensemble. La liste suivante comprend quelques menaces de sécurité courantes évoquées plus en détail dans les rubriques de cette section.

### > **Injection SQL:**

L'injection SQL est le processus qui permet à un utilisateur malveillant d'entrer des instructions Transact-SQL au lieu d'une entrée valide. Si l'entrée est transmise directement au serveur sans validation et si l'application exécute accidentellement le code injecté, l'attaque risque d'endommager ou de détruire des données.

### Solutions:

- Valider toutes les entrées
- Utiliser des procédures stockées ou des requêtes paramétrées

# Sécurité des données

Menaces courantes:

Exemples :

`SELECT * from utilisateurs where nom = @nom;` (requête avec paramètre) → Bonne pratique

`SELECT * from utilisateurs where nom ='Patoche' ;` → Mauvaise pratique

Il suffit que quelqu'un de malintentionné remplace la requête par:

`SELECT * from utilisateurs where nom ='Patoche' OR 1=1;`

Dans l'exemple 2, Il suffit que quelqu'un de malintentionné va ajouter la requête UNION ALL

`SELECT nom, Description FROM produits`

`WHERE Description like '%Chaises'`

**UNION ALL**

*`SELECT username ,password FROM dba_users`*

# Sécurité des données

Menaces courantes:

## > **Élévation de privilège :**

Les attaques d'élévation de privilège se produisent lorsqu'un utilisateur s'empare des privilèges d'un compte approuvé, un administrateur ou un propriétaire.

## > Solutions:

- Évitez l'utilisation des comptes d'administrateur (comme Sa pour SQL Server, root pour MySQL et system pour Oracle) pour l'exécution du code.
- Supprimer les comptes utilisateurs non utilisés
- Supprimer les comptes utilisateurs par défaut
- Donnez les privilèges selon les besoins.
- Idéalement, interdire les accès distants aux comptes administrateurs par défaut (root, sa, sys)

# Sécurité des données

Menaces courantes:

- › **Détection des attaques et surveillance intelligente**

Une attaque de détection peut utiliser des messages d'erreur générés par une application pour rechercher des vulnérabilités dans la sécurité.

- › Solutions:

- Ne pas afficher de messages d'erreur explicites affichant la requête ou une partie de la requête SQL. Personnalisez vos messages erreur.

# Sécurité des données

Menaces courantes:

## > Mots de passe

De nombreuses attaques réussissent lorsqu'un intrus a su deviner ou se procurer le mot de passe d'un utilisateur privilégié. Les mots de passe représentent la première ligne de défense contre les intrus, la définition de mots de passe forts est donc un élément essentiel de la sécurité de votre système.

## > Solutions:

- Renforcer les mots de passe.
- Utilisez la stratégie des mots de passe pour les comptes sa, root et system.
- Supprimer les comptes sans mot de passe.

# Sécurité des données

Les privilèges, les rôles ou les droits peuvent être attribués:

- › sur le serveur au complet
- › sur une ou plusieurs bases de données
- › sur une ou plusieurs tables, vues, procédures
- › sur une colonne ou plusieurs colonnes.

La plus part des SGBD regroupent un ensemble de privilèges dans des ROLES prédéfinis . Certains rôles sont sur le serveur, d'autres sont sur la base de données.

# Sécurité des données

## Rôle du serveur:

SQL Server fournit des rôles au niveau du serveur pour vous aider à gérer les autorisations sur les serveurs

- › SQL Server fournit neuf rôles serveur fixes. Les autorisations accordées aux rôles serveur fixes (à l'exception de **public**) ne peuvent pas être changées.

Les rôles du serveur sont attribués aux connexions



# Sécurité des données

## Rôle niveau serveur

Rôles	Description
<b>sysadmin</b>	Les membres du rôle serveur fixe <b>sysadmin</b> peuvent effectuer n'importe quelle activité sur le serveur.
<b>serveradmin</b>	Les membres du rôle serveur fixe <b>serveradmin</b> peuvent modifier les options de configuration à l'échelle du serveur et arrêter le serveur.
<b>securityadmin</b>	Les membres du rôle serveur fixe <b>securityadmin</b> gèrent les connexions et leurs propriétés. Ils peuvent attribuer des autorisations GRANT, DENY et REVOKE au niveau du serveur. Ils peuvent également attribuer des autorisations GRANT, DENY et REVOKE au niveau de la base de données, s'ils ont accès à une base de données. En outre, ils peuvent réinitialiser les mots de passe pour les connexions SQL Server .
<b>processadmin</b>	Les membres du rôle serveur fixe <b>processadmin</b> peuvent mettre fin aux processus en cours d'exécution dans une instance de SQL Server.
<b>setupadmin</b>	Les membres du rôle serveur fixe <b>setupadmin</b> peuvent ajouter et supprimer des serveurs liés à l'aide d'instructions Transact-SQL. (L'appartenance au rôle <b>sysadmin</b> est nécessaire pour utiliser Management Studio.)
<b>bulkadmin</b>	Les membres du rôle serveur fixe <b>bulkadmin</b> peuvent exécuter l'instruction BULK INSERT.
<b>diskadmin</b>	Le rôle serveur fixe <b>diskadmin</b> permet de gérer les fichiers disque.
<b>dbcreator</b>	Les membres du rôle serveur fixe <b>dbcreator</b> peuvent créer, modifier, supprimer et restaurer n'importe quelle base de données.
<b>public</b>	Chaque connexion SQL Server appartient au rôle serveur <b>public</b> . Lorsqu'un principal de serveur ne s'est pas vu accorder ou refuser des autorisations spécifiques sur un objet sécurisable, l'utilisateur hérite des autorisations accordées à public sur cet objet. Vous ne devez affecter des autorisations publiques à un objet que lorsque vous souhaitez que ce dernier soit disponible pour tous les utilisateurs. Vous ne pouvez pas modifier l'appartenance au rôle public.

# Sécurité des données

Rôle niveau base de données:

- › Les rôles niveau base de données sont attribués à un utilisateur mappé sur la connexion

# Les Rôles BD

Rôles	Description
<b>db_owner</b>	Les membres du rôle de base de données fixe <b>db_owner</b> peuvent effectuer toutes les activités de configuration et de maintenance sur la base de données et peuvent également supprimer la base de données dans SQL Server. (Dans SQL Database et SQL Data Warehouse, certaines activités de maintenance requièrent des autorisations de niveau serveur et ne peuvent pas être effectuées par db_owners.)
<b>db_securityadmin</b>	Les membres du rôle de base de données fixe <b>db_securityadmin</b> peuvent modifier l'appartenance au rôle pour les rôles personnalisés uniquement et gérer les autorisations. Les membres de ce rôle peuvent potentiellement élever leurs privilèges et leurs actions doivent être supervisées.
<b>db_accessadmin</b>	Les membres du rôle de base de données fixe <b>db_accessadmin</b> peuvent ajouter ou supprimer l'accès à la base de données des connexions Windows, des groupes Windows et des connexions SQL Server.
<b>db_backupoperator</b>	Les membres du rôle de base de données fixe <b>db_backupoperator</b> peuvent sauvegarder la base de données.
<b>db_ddladmin</b>	Les membres du rôle de base de données fixe <b>db_ddladmin</b> peuvent exécuter n'importe quelle commande DDL (Data Definition Language) dans une base de données.
<b>db_datawriter</b>	Les membres du rôle de base de données fixe <b>db_datawriter</b> peuvent ajouter, supprimer et modifier des données dans toutes les tables utilisateur.
<b>db_datareader</b>	Les membres du rôle de base de données fixe <b>db_datareader</b> peuvent lire toutes les données de toutes les tables utilisateur.
<b>db_denydatawriter</b>	Les membres du rôle de base de données fixe <b>db_denydatawriter</b> ne peuvent ajouter, modifier ou supprimer aucune donnée des tables utilisateur d'une base de données.
<b>db_denydatareader</b>	Les membres du rôle de base de données fixe <b>db_denydatareader</b> ne peuvent lire aucune donnée des tables utilisateur d'une base de données.
<b>Public</b>	Permet de faire un USE sur la BD

# Sécurité des données

## Les privilèges sur les objets:

Les privilèges sur les objets sont attribués aux utilisateurs et non aux connexions.

En général un privilège ou une permission correspond à une commande SQL: SELECT, INSERT, DELETE, UPDATE, EXECUTE, CREATE...

Un objet, correspond à une table, une vue, une colonne, une procédure stockée ou une fonction.

# Sécurité des données

## **Retour sur la dernière séance:**

- › Point de vue des étudiants
- › Point de vue des enseignant

## **Rappels:**

- › Attaques possibles
- › Rôles serveur et rôles BD
- › Permissions sur les objets.

## **Suite du cours:**

- › Création d'un user mappé sur une connexion
- › Attribution de rôles
- › Les commande GRANT, REVOKE et DENY

# Sécurité des données

## À retenir absolument (Rappels)

- › Éviter le code sql (utiliser des procédures stockées ou requêtes paramétrées) dans vos applications clientes pour prévenir les injection SQL
- › Valider vos entrées des formulaires
- › Utiliser une stratégie de mot de passes
- › Restreindre les privilèges utilisateurs
- › Éviter d'afficher les messages erreur du SGBD
- › N'utilisez pas vos comptes admin pour vos opérations courantes.
- › Créer des connexions avec des rôles suffisants. Pas plus
- › Pour les user, donner les rôles BD suffisants . Pas plus

# Sécurité des données

## Attribution de ROLE

Lorsqu'une connexion est créée, elle a les rôles public sur le serveur et sur la BD qui lui est associée.

Un rôle PUBLIC sur le serveur, veut dire que l'on peut se connecter et uniquement se connecter au serveur

Un rôle public sur la BD, veut dire que l'on peut faire un USE et uniquement un USE sur la BD.

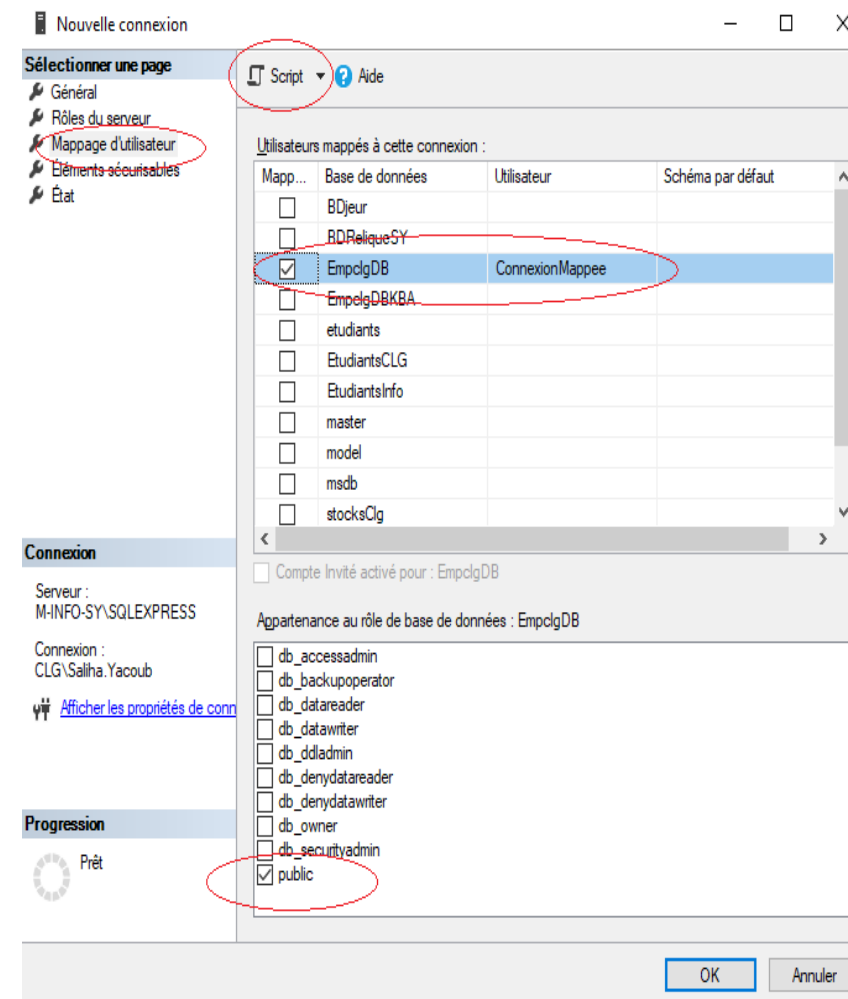
Pour attribuer des permissions ou des privilèges, il faudra mapper la connexion à un utilisateur (car les permissions sont données aux utilisateurs).

Pour garantir la sécurité des données il est conseillé de donner les rôles et les permissions au besoin.

# Sécurité des données

## Création d'une connexion mappée sur un utilisateur (par l'interface)

- On suit les mêmes étapes pour créer une connexion, puis avant de cliquer sur OK, il faudra mapper un utilisateur à cette connexion
- L'utilisateur mappé a le même nom que la connexion.
- On lui affecte une base de données, sinon ce sera un utilisateur « orphelin »
- Un utilisateur utilise un login pour se connecter et il est rattaché à une base de données.
- Son rôle est PUBLIC, dans ce cas il peut faire un USE sur Empcldb





# Sécurité des données

## Attribution de ROLE

Pour la connexion plus haut, voici le code SQL correspondant:

```
USE Empc1gDB;  
CREATE LOGIN [connexionMappee] WITH PASSWORD=N'123456',  
DEFAULT_DATABASE=[Empc1gDB], CHECK_EXPIRATION=OFF, CHECK_POLICY=OFF  
USE [Empc1gDB]  
  
CREATE USER [connexionMappee] FOR LOGIN [connexionMappee]
```

# Sécurité des données

## Attribution de ROLE

Lorsqu'un ROLE est attribué, nous dirons qu'un MEMBRE a été RAJOUTÉ au ROLE

```
CREATE LOGIN [connexionMappee] WITH PASSWORD=N'123456',  
DEFAULT_DATABASE=[master], CHECK_EXPIRATION=OFF, CHECK_POLICY=OFF
```

```
USE [EmpclgDB]
```

```
CREATE USER [connexionMappee] FOR LOGIN [connexionMappee]
```

```
ALTER ROLE [db_datareader] ADD MEMBER [connexionMappee]
```

# Sécurité des données

Pour mieux voir que les ROLES sont attribués aux USERS et non aux connexions, remarquez le nom du USER mappé sur la connexion

USE [EmpcldgDB]

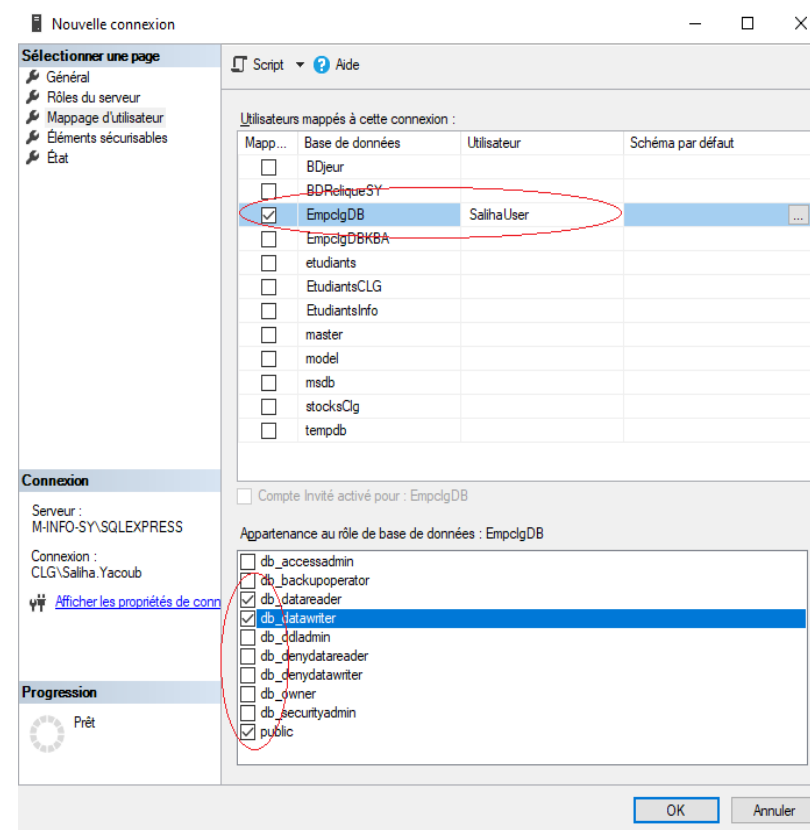
```
CREATE LOGIN [ETUDIANTConnexion] WITH  
PASSWORD=N'123456',  
DEFAULT_DATABASE=[EmpcldgDB],  
CHECK_EXPIRATION=OFF, CHECK_POLICY=OFF
```

```
CREATE USER [SalihaUser] FOR LOGIN  
[SalihaConnexion]
```

```
ALTER ROLE [db_datareader] ADD MEMBER  
[ETUDIANTUser]
```

```
ALTER ROLE [db_datawriter] ADD MEMBER  
[ETUDIANTUser]
```

GO



# Sécurité des données

Nous pouvons attribuer tous les rôles (serveur ou base de données par l'interface graphique) mais il est important de faire l'ensemble des opérations par des commandes SQL.

**Dans les travaux et examens, seules les commandes SQL seront acceptées.**

D'après ce que nous avons compris:

1. Il faut créer le login
2. Créer l'utilisateur mappé sur le login
3. Et si nous ne voulons pas que le USER soit orphelin, il faudra faire USE uneBD avant de créer le USER.

Recommandé : Faire le USE avant de créer la connexion.

# Sécurité des données

```
use stocksClg;  
CREATE LOGIN connexion2020 with password = '123456',  
default_database = stocksClg, CHECK_POLICY=OFF ;  
  
CREATE USER connexion2020 for login connexion2020;  
  
ALTER ROLE [db_datareader] ADD MEMBER connexion2020;
```

Avec ce role l'utilisateur **connexion2020** ne pourra pas créer des objets (Tables) sur la base de données et ne pourra pas faire des INSERT , UPDATE, DELETE sur les tables de la BD

Pour permettre à connexion2020 de faire des MAJ, il faut l'ajouter au **role db\_datawriter**

```
alter role db_datawriter ADD MEMBER connexion2020;
```

Pour retirer un membre d'un ROLE c'est : ALTER ROLE nomRole DROP MEMBER nomMembre.

```
alter role db_ddladmin DROP MEMBER PatocheUser
```

# Sécurité des données

› Rappels: création d'un LOGIN:

lorsqu'un login est créé, le nom et le mot de passe sont obligatoires les autres paramètres ne le sont pas , mais nous pouvons les fournir séparés par une virgule. Comme on l'a vu la dernière fois, les paramètres importants à fournir sont:

- La BD par défaut,
- Stratégie du mot de passe activée ou non. Par défaut cette stratégie est activée
- Le mot de passe va-t-il expiré ?
- On peut aussi décider si l'on souhaite changer le mot de passe après la première connexion. Dans ce cas, l'option expiration du mot de passe doit-être activée.

# Sécurité des données

## Exemple 1

```
CREATE LOGIN kbaLog with password = 'kba$420CLG' MUST_CHANGE,  
check_expiration =on,  
check_policy =on,  
default_database =stocksClg;
```

## Exemple 2

```
CREATE LOGIN kbaLog with password = 'kba$420CLG' ,  
check_expiration =off,  
check_policy =on,  
default_database =stocksClg;
```

# Sécurité des données

Pour avoir accès à la base de données par défaut, il faut créer un utilisateur de même nom que la connexion après avoir fait un USE sur la BD

```
use stocksClg;
```

```
CREATE LOGIN kbaLog with password = 'kba$420CLG' ,
```

```
check_expiration =off,
```

```
check_policy =on,
```

```
default_database =stocksClg;
```

```
create user kbaLog for login kbaLog;
```

Une connexion crée de cette façon a uniquement le role public sur le serveur et le user mappé sur cette connexion et le rôle public sur la BD.

Pour ajouter un User à un ROLE

```
Alter ROLE nomrole ADD MEMBER nomuser
```



## Sécurité des données: les privilèges sur les objets:

Lorsque vous attribuez un ROLE sur la BD, les autorisations qui viennent avec le rôle s'appliquent à toutes la base de données et donc à toutes les tables, vues.

Ce qui veut dire , que si on fait:

```
Alter role db_datareader add member kbalog;
```

kbaLog a le droit de faire SELECT sur toutes les tables de la base de données stockCLG.

Pour donner des permissions sur des objets un seul à la fois , on utilise la commande **GRANT**.

# Sécurité des données, GRANT

## › La commande GRANT:

La commande GRANT permet d'attribuer des privilèges ou des permissions à un utilisateur sur un **objet**. Au lieu de donner des droits sur toutes les tables de la base de données par attribution de ROLE, on utilise la commande GRANT pour cibler les objets de la base de données qui seront affectés et restreindre les privilèges sur les objets ciblés pour les utilisateurs.

```
GRANT { ALL [ PRIVILEGES ] }  
      | permission [ ( column [ ,...n ] ) ] [ ,...n ]  
      [ ON [ class :: ] securable ] TO principal [ ,...n ]  
      [ WITH GRANT OPTION ] [ AS principal ]
```

# Sécurité des données, GRANT

- Si l'élément sécurisable est une fonction scalaire, ALL représente SELECT et REFERENCES.
- Si l'élément sécurisable est une fonction table, ALL représente DELETE, INSERT, REFERENCES, SELECT et UPDATE.
- Si l'élément sécurisable est une procédure stockée, ALL représente EXECUTE.
- Si l'élément sécurisable est une table, ALL représente DELETE, INSERT, REFERENCES, SELECT et UPDATE.
- Si l'élément sécurisable est une vue, ALL représente DELETE, INSERT, REFERENCES, SELECT et UPDATE.
- WITH GRANT OPTION, signifie que l'utilisateur qui a reçu le privilège peut donner le même privilège à un autre utilisateur sur le même objet.

# Sécurité des données, GRANT

## Exemples:

- `GRANT SELECT , INSERT ON articles TO kbaLog;`
- `GRANT SELECT, INSERT, UPDATE(adresse) ON Clients TO user1,user2;`
- `GRANT SELECT ON Commandes TO kbaLog WITH GRANT OPTION;`

# Sécurité des données: REVOKE

- › La commande REVOKE:
- › La commande REVOKE permet de retirer des droits. (Des privilèges) . En principe les privilèges ont été attribuer par la commande GRANT

```
REVOKE [ GRANT OPTION FOR ] <permission> [ ,...n ] ON  
    [ OBJECT :: ] [ schema_name ]. object_name [ ( column [ ,...n ] ) ]  
    { FROM | TO } <database_principal> [ ,...n ]  
    [ CASCADE ]  
    [ AS <database_principal> ]
```

**CASCADE**, lorsque vous avez : WITH GRANT OPTION, enlève les privilèges en cascade

```
REVOKE INSERT ON articles FROM kbaLog
```

```
REVOKE GRANT OPTION FOR SELECT on Commandes  
FROM kbaLog CASCADE;
```

# Sécurité des données, DENY

- › La commande DENY:
- › La commande DENY permet de **retirer des droits hérités** d'un **ROLE**.

```
DENY    { ALL [ PRIVILEGES ] }  
        | <permission> [ ( column [ ,...n ] ) ] [ ,...n ]  
        [ ON [ <class> :: ] securable ]  
        TO principal [ ,...n ]  
        [ CASCADE ] [ AS principal ]  
[;]
```

Exemples

DENY SELECT on Clients to kbaLog;

# Sécurité des données

Conclusion pour cette partie:

- › Ne pas donner un rôle serveur autre que PUBLIC à n'importe quelle connexion.
- › Ne pas donner un rôle BD, autre que PUBLIC à n'importe quel user.
- › Il vaut mieux ajouter un privilège au besoin, plutôt que d'en retirer après un problème. Il faut cibler l'octroi des privilèges aux utilisateurs.
- › La commande GRANT permet de mieux cibler les permissions sur les objets.
- › La commande REVOKE permet de retirer des privilèges attribués par GRANT.
- › La commande DENY permet d'enlever un privilège hérité d'un ROLE
- › Ces trois commandes font partie du SQL standard. Disponibles dans tous les SGBDs.

# Sécurité des données, Démonstration

- › Création de login
- › Création de user mappés sur le login
- › Attribution de ROLE DB
- › Tester la commande DENY
- › Tester la commande GRANT
- › Tester la commande REVOKE



# Sécurité des données

- › Retour sur la dernière séance:
  - Point de vue de l'étudiant
  - Point de vue des enseignants
- › Rappels
- › Les rôles non prédéfinis;
- › Les vues pour la sécurité des données.
- › Chiffrement des données
  - Définition
  - Le hachage
  - L'encryption.
- › Démonstration

# Sécurité des données, créer un rôle

**Les ROLES non prédéfinis.** (ceux créés par les utilisateurs)

Rappel: Un rôle va regrouper plusieurs privilèges . L'avantage de créer des rôles est de pouvoir attribuer le même ensemble de privilèges à plusieurs utilisateurs .

Imaginer la situation suivante:

- Vous avez 20 programmeurs qui travaillent sur le même projet utilisant une base de données VenteProduits
- Vous avez donné les permissions nécessaires aux 20 employés sur les tables de cette base de données.
- Deux semaines plus tard, on se rend compte que nous avons oublié d'attribuer le droit UPDATE du prix sur la table Articles , et le droit INSERT sur la table Commissions .

# Sécurité des données, créer un rôle

› **Les ROLES non prédéfinis.** (ceux créés par les utilisateurs)

La solution serait de retrouver les 20 programmeurs et de leur ajouter ces privilèges . Pas évident.

Imaginez maintenant que vous avez recruté deux autres programmeurs pour travailler sur le même projet.

Comment allez-vous faire pour retrouver toutes les permissions attribuées aux 20 premiers pour les donner aux deux derniers ? Chercher dans la documentation à condition que vous ayez tout garder.

Une solution simple serait de créer un ROLE pour lequel on donne les permissions ensuite, on donne le role à qui en a besoin. Facile.

# Sécurité des données, créer un role

## > Les ROLES non prédéfinis: Étapes:

- On crée le ROLE → CREATE ROLE nomrole
- On attribue les permissions au ROLE → GRANT
- On ajoute un user (MEMBRE) au ROLE en utilisant la commande suivante:

```
ALTER ROLE nomrole ADD MEMBER nomUser
```

# Sécurité des données

## > Exemple:

```
USE StocksClg;
```

```
CREATE ROLE roleprojet;
```

```
GRANT SELECT , INSERT ON Articles TO roleprojet;
```

```
GRANT SELECT , INSERT, UPDATE(adresse) on Clients to roleprojet;
```

```
ALTER ROLE roleprojet add member Simba;
```

Ce rôle, je peux l'attribuer à toute une équipe de projet.

# Sécurité des données

Maintenant, si nous avons oublié les privilèges UPDATE du prix sur la table Articles , et le privilège INSERT sur la table LigneCommande, alors on fait:

```
GRANT UPDATE(prix) ON Articles TO roleprojet;
```

```
GRANT INSERT ON lignecommande TO roleprojet;
```

On réhausse le ROLE roleprojet en ajoutant les privilèges nécessaires.

Evidemment, si on veut enlever des privilèges à toute l'équipe de projet, il suffit de faire un REVOKE sur le rôle.

```
REVOKE insert ON Clients FROM roleprojet
```

# Sécurité des données

- › Nous avons abordé les vues comme étant des objets de la base de données permettant la simplification de requêtes.
- › Dans ce qui suit, nous allons voir comment les vues peuvent contribuer à la sécurité des données.

```
CREATE [ OR ALTER ] VIEW [ schema_name . ] view_name  
  
AS select_statement    [ WITH CHECK OPTION ]    [ ; ]
```

- › L'option WITH CHECK OPTION permet d'assurer que les modifications apportées à la table (dont la vue est issue) via la vue respectent la CLAUSE WHERE.
- › Lorsqu'une ligne est modifiée via la vue, alors les données devraient rester cohérentes.

# Sécurité des données

Exemple :

```
create view VSport as select * from questions where  
codecategorie =3 with check option;
```

```
grant select, update, insert on Vsport to user1;
```

L'instruction suivante exécutée par le user1 va marcher car le code catégorie est 3:

```
insert into VSport values('une question',1,'facile',3);
```

L'instruction suivante exécutée par le user1 NE va PAS marcher car le code catégorie est 2. Ne respecte pas la clause WHERE:

```
insert into VSport values('une autre question',1,'facile',2);
```

De plus, le USER1, ne voit pas TOUT le contenu de la table Questions



# Le chiffrement des données: Le hachage

Définition: Le chiffrement est un procédé de la cryptographie qui consiste à rendre les données illisible ou impossible à lire sauf si vous avez une clé de déchiffrement.

- Techniques: **Le hachage** et le **chiffrement**.
- > La technique de hachage des données a la particularité d'être **irréversible**; il n'est pas possible de retrouver les données originales après avoir été crypté avec une fonction de hachage.
- > Il s'agit d'une fonction mathématique qui prend en entrée des données (chaîne de caractères de longueur variable) et qui génère en sortie une chaîne de caractères de longueur fixe appelée « hash »;
- > La sortie (hash) est toujours la même pour des entrées identiques;
- > Cette technique de chiffrement est couramment utilisée pour conserver des **mots de passe** ou vérifier l'intégrité d'un document;
- > Algorithmes de hachage les plus utilisés: SHA2\_256, SHA2\_512
- > appelé également chiffrement unidirectionnel.
- > Dans **MS SQL Server** le chiffrement par hachage est fait avec la fonction **HASHBYTES**;
  - La fonction prend deux paramètres: **HASHBYTES ( '<algorithm>', { @input | 'input' } )**
    - > L'algorithme à utiliser et les données à chiffrer;
    - > L'algorithme ne chiffre pas des chaînes plus longues que 8000 caractères;

# Le chiffrement des données: Le hachage

```
create table fournisseurs (  
idfournisseur int identity,  
nom varchar(40),  
adresse varchar(50) not null,  
mpasse varbinary(128) not null,  
constraint pk_fournisseur primary key (idfournisseur));  
  
--Les insertions...  
  
insert into fournisseurs values  
('le iga', 'laval etc', HASHBYTES('SHA2_512', 'local$33'));  
insert into fournisseurs values  
('le metro', 'rue de la liune ici', HASHBYTES('SHA2_512', 'passww$33'));  
insert into fournisseurs  
values ('Lafleur', 'pas loin', HASHBYTES('SHA2_512', 'Motdepasse123'));
```

# Le chiffrement des données: Le hachage

```
select * from fournisseurs;
```

	idfournisseur	nom	adresse	mpasse
1	1	le iga	laval etc	0x2E33B2AB38861BF326645AC58A85B5BDCA8AFDE897D7F5A...
2	2	le metro	rue de la liune ici	0x97780E3B6AED0C1764151E77B2D0ED539675F1CFF52FE64C...
3	3	Lafleur	pas loin	0xF8AE481DCC4A9D8AC4317BAC2FCBA7729B2CF0DABB5EF4...
4	4	un fournisseur	blaba	0x2E33B2AB38861BF326645AC58A85B5BDCA8AFDE897D7F5A...

Si je veux chercher un fournisseur selon son mot de passe:

```
select * from fournisseurs where mpasse =  
(  
  select (HASHBYTES('SHA2_512', 'local$33'))  
) and idfournisseur =1;
```

## Exemple1: changer son mot de passe

```
create procedure changerMdp(@id int,  
                           @actuelMdp varchar(30),  
                           @nouveauMdp varchar(30)) AS  
  
Begin  
Declare @nouveauEncrypted varbinary(128), @currentMdp varbinary(128);  
---on cherche le mot de passe actuel  
select @currentMdp = mdp from fournisseurs where idfournisseur = @id;  
if(HASHBYTES('SHA2_512',@actuelMdp) = @currentMdp)  
begin  
select @nouveauEncrypted= HASHBYTES('SHA2_512',@nouveauMdp );  
update fournisseurs set mdp =@nouveauEncrypted where idfournisseur =@id;  
end;  
else print('les mots de passe ne correspondent pas');  
end;
```

# Le chiffrement des données: Le hachage

Pour executer:

```
execute changerMpassse
```

```
@id =1,
```

```
@actuelMpassse ='local$33',
```

```
@nouveMpassse ='Voila123';
```

## Exemple2: Vérifier si un fournisseur existe

```
create function existFournisseur(@id int,  
                                @mpasse varchar(30)) returns int as  
  
begin  
declare @mpasseHach varbinary(128);  
    select @mpasseHach = mpasse from fournisseurs  
    where idfournisseur =@id;  
        if(HASHBYTES('SHA2_512',@mpasse) = @mpasseHach)  
        begin  
            return 1;  
        end;  
  
    return 0;  
end;  
  
Pour tester:  
select dbo.existFournisseur(2, 'passww$33');
```

## Le chiffrement des données: Le chiffrement (l'encryption)

Chiffrement symétrique: chiffrement rapide qui utilise une seule clé; la même clé est utilisée pour crypter et décrypter les données;

ENCRYPTBYPASSPHRASE, DECRYPTBYPASSPHRASE: encryption des données en utilisant l'algorithme Triple DES (Data Encryption Standard)

Ces fonctions utilisent une clé privée pour chiffrer les données. La clé est fournie sous la forme d'une chaîne de caractères (un mot de passe par exemple).

**EncryptByPassPhrase ( { 'passphrase' | @passphrase } , { 'cleartext' | @cleartext } )**

**passphrase**, phrase secrète à partir de laquelle générer une clé symétrique

**@passphrase**, Variable de type nvarchar, char, varchar, binary, varbinary ou nchar contenant

une phrase de passe à partir de laquelle générer une clé symétrique.

**Cleartext** : le texte à chiffrer

## Le chiffrement des données: Le chiffrement (l'encryption)

DECRYPTBYPASSPHRASE fait l'opération inverse de **EncryptByPassPhrase**

```
DECRYPTBYPASSPHRASE ( { 'passphrase' | @passphrase } , { 'textecripte' | @textecripte } )
```

Passphrase: phrase secrète utilisée pour générer la clé

Textecripte: Texte encrpté.



## Le chiffrement des données: Le chiffrement (l'encryption)

on ajoute la colonne pour la carte de crédit non cryptée

```
alter table fournisseurs add carteCredit varchar(20);
```

-- on met à jour normalement

```
update fournisseurs set carteCredit = '5555-5021-5789-4569'  
where idfournisseur=1;
```

```
update fournisseurs set carteCredit = '4567-8888-5789-9999'  
where idfournisseur=2;
```

```
update fournisseurs set carteCredit = '8888-8888-5789-2222'  
where idfournisseur=3;
```

--On ajoute une colonne carte de crédit encryptée (juste pour vérifier)

```
alter table fournisseurs add carteCryptee varbinary(128);
```

```
update fournisseurs set carteCryptee  
=ENCRYPTBYPASSPHRASE('local$33', '5555-5021-5789-4569') where  
idfournisseur=1;
```

# Le chiffrement des données: Le chiffrement (l'encryption)

Pour vérifier :

```
SELECT nom, adresse, carteCryptee AS 'carte encryptee',  
CONVERT(varchar, DECRYPTBYPASSPHRASE('local$33', carteCryptee)) AS  
'carte decryptée' FROM fournisseurs where idfournisseur =1;
```

nom	adresse	carte encryptee	carte decryptée
le iga	laval etc	0x020000002AA140B35692E931AAF74E9F4C925BD722590C1...	5555-5021-5789-4569

## Le chiffrement des données: Le chiffrement (l'encryption)

Exemple 2: on fait les choses proprement avec une procédure stockée

```
alter table fournisseurs drop column carteCredit;
```

```
alter table fournisseurs drop column carteCryptee ;
```

```
alter table fournisseurs add carteCryptee varbinary(128);
```

Contenue de la table fournisseurs:

idfournisseur	nom	adresse	mpasse	carteCryptee
1	le iga	laval etc	0x2E33B2AB38861BF326645AC58A85B5BDCA8AFDE897D7F5A...	NULL
2	le metro	rue de la lune ici	0x97780E3B6AED0C1764151E77B2D0ED539675F1CFF52FE64C...	NULL
3	Lafleur	pas loin	0xF8AE481DCC4A9D8AC4317BAC2FCBA7729B2CF0DABB5EF4...	NULL
4	un fournisseur	blaba	0x2E33B2AB38861BF326645AC58A85B5BDCA8AFDE897D7F5A...	NULL

La procédure stockée suivante va permettre de mettre à jour le numéro de carte de crédit uniquement si le mot de passe du fournisseur est valide.

## Exemple 1: ajouter info crédit

```
CREATE procedure ajouterCarteCredit (@id int,  
                                     @motdepasse varchar(30),  
                                     @carteNonCrypte varchar(20)) as  
  
begin  
declare @motdepasseHash varbinary(128);  
select @motdepasseHash = mpasse from fournisseurs where idfournisseur =@id;  
if(HASHBYTES('SHA2_512',@motdepasse ) = @motdepasseHash)  
    begin  
        update fournisseurs set carteCryptee =  
            ENCRYPTBYPASSPHRASE(@motdepasse, @carteNonCrypte)  
        where idfournisseur =@id;  
    end;  
end;
```

# Exemple 1: ajouter info crédit

```
execute ajouterCarteCredit
```

```
@id =1,
```

```
@motdepasse = 'Voila123',
```

```
@carteNonCrypte = '5555-5021-5789-4569'
```

```
execute ajouterCarteCredit
```

```
@id =2,
```

```
@motdepasse = 'passww$33',
```

```
@carteNonCrypte = '5555-1111-2222-4569';
```

```
--Ne marcher pas.
```

```
execute ajouterCarteCredit
```

```
@id =3,
```

```
@motdepasse = '123466',
```

```
@carteNonCrypte = '5555-3333-2222-4569'
```

idfournisseur	nom	adresse	mpasse	carteCryptee
1	le iga	laval etc	0x6E3976A1729F1E6EC0685ED08DE6FC973E2DAE6CAD61C16...	0x0200000038BFC8FFDBADF09AEA8EEC7F8956FDEE24A18...
2	le metro	rue de la liune ici	0x97780E3B6AED0C1764151E77B2D0ED539675F1CFF52FE64...	0x02000000F3228DC4CB33F117CB91DF92A0E6653C5EDBA45...
3	Lafleur	pas loin	0xF8AE481DCC4A9D8AC4317BAC2FCBA7729B2CF0DABB5EF48...	NULL

## Exemple 2: obtenir info crédit

```
create function obtenirInfoCreditFournisseur
(@idfournisseur varchar(50),
 @mdp varchar(30)) returns table as
return
select nom,
convert(varchar,DECRYPTBYPASSPHRASE(@mdp, carteCryptee)) as nocredit
      from fournisseurs where idfournisseur = @idfournisseur;

select * from obtenirInfoCreditFournisseur(1, 'Voila123');
```

## Le chiffrement des données: Le chiffrement (l'encryption)

Cette fonction utilise une clé privée pour chiffrer les données. Cette clé doit être préalablement créée avec la commande 'CREATE SYMMETRIC KEY'. C'est au moment de créer la clé symétrique que l'on peut spécifier l'algorithme de chiffrement.

Pour le chiffrement par clé symétrique, les algorithmes les plus utilisés sont: AES\_128, AES\_192 et AES\_256;

- Supporté par MS SQL Server;
  - Considéré le plus sécuritaire aujourd'hui;
  - Choisi par le gouvernement américain pour remplacer l'algorithme de chiffrement DES;
- 
- Les étapes:
    - On crée la clé symétrique protégée par un mot de passe
    - On ouvre la clé pour encrypter
    - On utilise la clé pour encrypter
    - On ferme la clé.

## Le chiffrement des données: Le chiffrement : Exemple (encrypter le NAS)

```
ALTER TABLE fournisseurs ADD nassCrypteByKey varbinary(128);  
  
CREATE SYMMETRIC KEY SymmetricKeyNass  
WITH ALGORITHM = AES_128  
ENCRYPTION BY password = 'Voila123';  
--Ouvrir la clé  
  
OPEN SYMMETRIC KEY SymmetricKeyNass  
Decryption BY password = 'Voila123';  
  
UPDATE fournisseurs SET nassCrypteByKey = EncryptByKey  
(Key_GUID('SymmetricKeyNass'), '999-120-506') where idfournisseur=1;  
--fermer la clé de chiffrement  
  
CLOSE SYMMETRIC KEY SymmetricKeyNass;  
  
select * from fournisseurs; --pour vérifier
```



## Le chiffrement des données: Le chiffrement : Exemple (encrypter le NAS)

Pour décrypter:

- On ouvre la clé
- On décrypte.

```
OPEN SYMMETRIC KEY SymmetricKeyNass
```

```
Decryption BY password = 'Voila123';
```

```
-- on affiche le nas décrypté
```

```
SELECT nom, nassCrypteByKey AS 'Nas encrypté',
```

```
CONVERT(varchar, DecryptByKey(nassCrypteByKey)) AS 'nas decrypté'
```

```
FROM fournisseurs where idfournisseur=1;
```



CONCLUSION



QUESTIONS ??