

# Rencontre 10

Insert avec Select, Update avec jointure, déclencheurs et transactions

Bases de données et programmation Web



- ❖ Retour sur les INSERT INTO ...SELECT
- ❖ Update avec jointure
- ❖ Déclencheurs
- ❖ Déclencheurs sur plusieurs instructions DML
- ❖ Contrôle de transactions

**Précision** : Nous pourrions passer **le reste de la session** sur ces notions car elles peuvent être exploitées de manière très sophistiquée. Cela dit, nous les aborderons seulement en surface pour être au moins capable de comprendre leur utilité et leur fonctionnement et les exploiter dans le contexte de la programmation Web.



❖ Pour insérer des données dans une table à partir des données d'une autre table.

◆ Avec les déclencheurs qu'on va voir bientôt, on va souvent insérer des données dans des tables d'archives ou d'audit. Ces données proviendront des tables existantes.

○ Exemple 1: Insérer la quantité et le prix des produits achetés dans la commande 1 dans une table d'audit appelée IngredientsTransaction

```
INSERT INTO Ingredients.IngredientsTransaction (IngredientID, QtyEnTransaction, Prix, DateETHeureTransaction)
SELECT IngredientID, Quantite, PrixVente, GETDATE()
FROM Commandes.DetailsCommande
WHERE CommandeID=1
```



## ❖ Un standard est utilisé fréquemment pour les tables d'archives

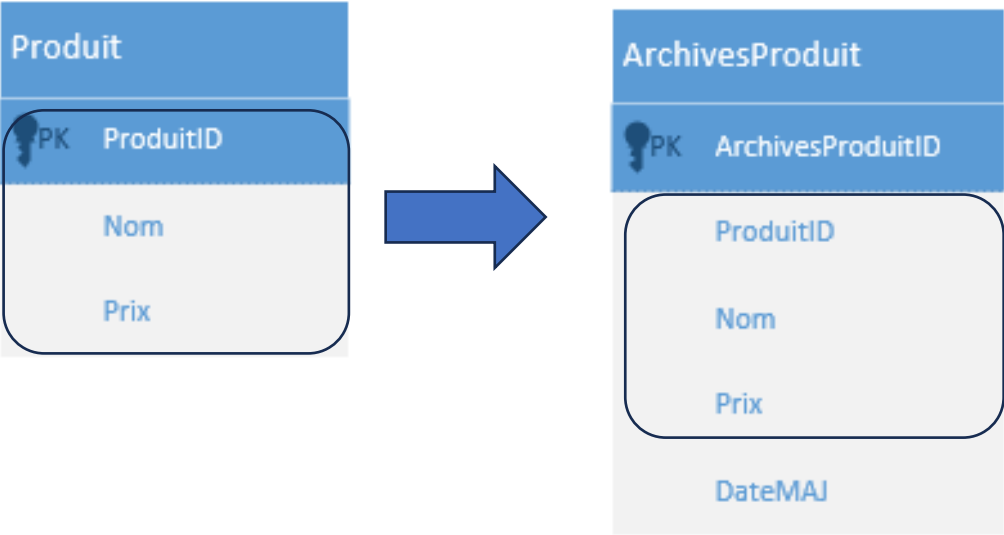
❖ Si la table Produit contient les champs (ProduitID, Nom, Prix, Description)

La table ArchivesProduit contiendra (ArchivesProduitID, ProduitID, Nom, Prix, Description, DateMAJ)

❖ Si on a la table ProduitSpec, qui contient (ProduitSpecID, Largeur, Hauteur, Poids, ProduitID)

La table ArchivesProduitSpec contiendra (ArchivesProduitSpecID, ProduitSpecID, Largeur, Hauteur, Poids, ProduitID, DateMAJ)

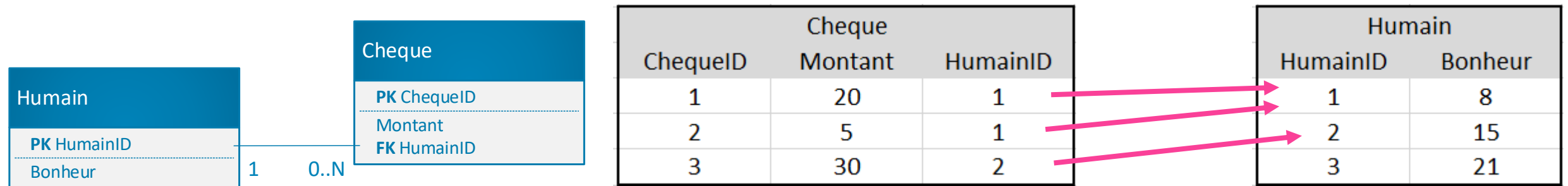
❖ Comme cela les tables liées dans les archives **gardent leurs liens entre elles**



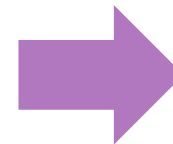


## ❖ UPDATE avec jointure💡

- ◆ En résumé : On met seulement à jour les rangées qui trouvent une correspondance avec l'autre table pendant la jointure.



```
UPDATE Humain
SET Bonheur += Montant
FROM Cheque C INNER JOIN Humain H
ON C.HumainID = H.HumainID
```



HumainID	Bonheur
1	28
2	45
3	21

- Le bonheur de l'humain #1 a augmenté de 20. (Pas de 25, car seule la première valeur trouvée a été prise en compte)
- Le bonheur de l'humain #2 a augmenté de 30.



## ❖ Déclencheur

- ◆ Un déclencheur (**trigger**) est comme une procédure interne à la base de données dont l'exécution est liée à la soumission d'une autre instruction de type **INSERT**, **UPDATE** ou **DELETE** (déclenchée après ou à la place de l'instruction) .
- ◆ Exemple : À chaque fois qu'on supprime une donnée d'une certaine table, on a un déclencheur pour que cette donnée soit automatiquement insérée dans une autre table à des fins d'archives.



## ❖ Créer un déclencheur :

GO

CREATE TRIGGER schema.trg\_XnomDéclencheur

ON nom\_table

<INSTEAD OF ou AFTER> <INSERT ou UPDATE ou DELETE>

AS

Instruction(s) SQL

} Peut contenir des SELECT, des INSERT, UPDATE, DELETE, etc.

GO





❖ Il y a deux types de triggers:

❖ **AFTER trigger** : permet d'exécuter du code après qu'une instruction ait été exécutée.

❖ **INSTEAD trigger** : permet d'exécuter du code à la place d'une autre instruction.  
Surtout utilisée pour remplacer les DELETE par quelque chose d'autre.



- ❖ 2 tables temporaires sont créées pour la durée du trigger seulement.
- ❖ **deleted** et **inserted**
- ❖ **deleted**, image des données AVANT l'update ou le delete.
- ❖ **inserted**, image des données APRÈS l'insert ou l'update.



- ❖ **deleted**, image des données AVANT l'update ou le delete.
- ❖ **inserted**, image des données APRÈS l'insert ou l'update.

DML	deleted (AVANT)	inserted (APRES)
Insert	Vide	Enregistrements insérés
Update	Enregistrements AVANT la mise à jour	Enregistrements mis-à-jour
Delete	Enregistrement AVANT la suppression	Vide



## ❖ Standards de nommage

❖ Il y a différents standards utilisés. Ici:

◆ schema.**trg\_**nomDuTrigger.

❖ Pour le nom du trigger : **trg\_** suivi d'un :

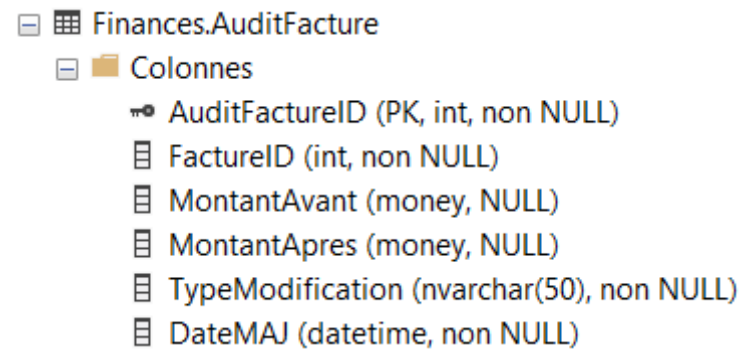
**i**, si c'est pour un insert. Ex : schema.**trg\_i**nomDuTrigger ,

**u**, si c'est pour un update,

**d**, si c'est pour un delete,

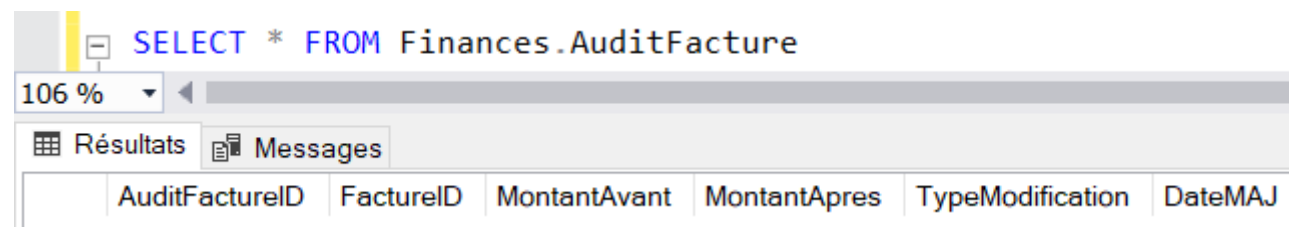
**iu**, si c'est pour un insert et un update. Ex: Biens.**trg\_iu**Bien

Observons que dans **BD\_DemoTrigger** nous avons 2 tables sans données:  
**Facture** et **AuditFacture**.



The screenshot shows the 'Columns' folder expanded for the 'Finances.AuditFacture' table. The columns listed are:

AuditFactureID (PK, int, non NULL)
FactureID (int, non NULL)
MontantAvant (money, NULL)
MontantApres (money, NULL)
TypeModification (nvarchar(50), non NULL)
DateMAJ (datetime, non NULL)



The screenshot shows a query window with the following SQL statement:

```
SELECT * FROM Finances.AuditFacture
```

The results pane shows a table with the following columns:

AuditFactureID	FactureID	MontantAvant	MontantApres	TypeModification	DateMAJ
----------------	-----------	--------------	--------------	------------------	---------



# Trigger AFTER INSERT

Nous allons insérer une nouvelle facture dans la table **Facture**.

Après l'insertion d'un enregistrement dans **Facture**, le trigger **Finances.trg\_iFacture** se déclenchera.

Celui-ci insèrera dans la table **AuditFacture** un enregistrement qui précisera qu'on vient de faire un INSERT comme type de modification et enregistrera la date de cette insertion.



Voyons les infos qu'on veut entrer dans la table **AuditFacture** après l'ajout d'une nouvelle facture dans la table **Facture**:

Le champ **AuditFactureID** est un compteur qui s'augmente tout seul. On n'entre pas d'infos là.

<u>DML</u>	Table DELETED (Image AVANT)	Table INSERTED (Image APRÈS)
Insert	Vide	Enregistrements insérés

Le **FactureID** nous viendra de la table **inserted**

Le **MontantAvant** sera NULL

Le **MontantAprès** nous viendra de la table **inserted**

Le **TypeModification** sera **'INSERT'**

La **DateMAJ** sera **GetDate()**

# Création du trigger AFTER INSERT

```
GO
CREATE TRIGGER Finances.trg_iFacture
ON Finances.Facture
AFTER INSERT
AS
BEGIN
    DECLARE @FactureID int;
    DECLARE @Montant money;

    SELECT @FactureID=FactureID, @Montant=Montant FROM inserted;

    INSERT INTO Finances.AuditFacture (FactureID, MontantAvant, MontantAprès, TypeModification, DateMAJ)
    VALUES (@FactureID, NULL, @Montant, 'INSERT', GETDATE());
END
GO
```



# Pour tester:

Données des tables Finances.Facture et Finances.AuditFacture AVANT l'insert

La commande INSERT sur la table Finances.Facture

Données des tables Finances.Facture et Finances.AuditFacture APRÈS l'Insert

```
--Derniers enregistrements de la table Facture
SELECT TOP(1) 0 AS [AVANT INSERT], FactureID, ClientID, Montant, Taxes, EmployeID
FROM Finances.Facture
ORDER BY FactureID DESC
--Derniers enregistrements de la table AuditFacture
SELECT TOP(1) 0 AS [AVANT INSERT], AuditFactureID, FactureID, MontantAvant, MontantApres, TypeModification, DateMAJ
FROM Finances.AuditFacture
ORDER BY AuditFactureID DESC
GO
--Insertion d'une nouvelle facture
INSERT INTO Finances.Facture (ClientID, Montant, Taxes, EmployeID)
VALUES (1, 1000, 150,2);
GO
--Derniers enregistrements de la table Facture
SELECT TOP(2) 0 AS [APRÈS INSERT], FactureID, ClientID, Montant, Taxes, EmployeID
FROM Finances.Facture
ORDER BY FactureID DESC
--Derniers enregistrements de la table AuditFacture
SELECT TOP(2) 0 AS [APRÈS INSERT], AuditFactureID, FactureID, MontantAvant, MontantApres, TypeModification, DateMAJ
FROM Finances.AuditFacture
ORDER BY AuditFactureID DESC
```

106 %

Résultats Messages

	AVANT INSERT	FactureID	ClientID	Montant	Taxes	EmployeID

	AVANT INSERT	AuditFactureID	FactureID	MontantAvant	MontantApres	TypeModification	DateMAJ

	APRÈS INSERT	FactureID	ClientID	Montant	Taxes	EmployeID
1	0	1	1	1000,00	150,00	2

	APRÈS INSERT	AuditFactureID	FactureID	MontantAvant	MontantApres	TypeModification	DateMAJ
1	0	1	1	NULL	1000,00	INSERT	2025-02-17 15:21:04.330



# Trigger AFTER UPDATE

Quand nous allons **modifier le montant d'une facture** dans la table **Facture**, le trigger **Finances. trg\_uFacture** se déclenchera. Celui-ci insèrera dans la table **AuditFacture** un enregistrement avec UPDATE comme type de modification et la date de cette mise-à-jour.



Voyons les infos qu'on veut entrer dans la table **AuditFacture** après la **modification du montant d'une facture** dans la table **Facture**:

Le champ **AuditFactureID** est un compteur qui s'augmente tout seul. On n'entre pas d'infos là.

DML	Table deleted (Image AVANT)	Table inserted (Image APRÈS)
Update	Enregistrements avant l'update	Enregistrements après l'update

Le **FactureID** nous viendra de la table **inserted**

Le **MontantAvant** nous viendra de la table **deleted**

Le **MontantAprès** nous viendra de la table **inserted**

Le **TypeModification** sera **'UPDATE'**

La **DateMAJ** sera **GetDate()**

# Trigger AFTER UPDATE :

```
GO
CREATE TRIGGER Finances.trg_uFacture
ON Finances.Facture
AFTER UPDATE
AS
BEGIN
    DECLARE @FactureID int;
    DECLARE @MontantAVANT money, @MontantAPRES money;

    SELECT @FactureID=FactureID, @MontantAPRES=Montant FROM inserted;
    SELECT @MontantAVANT=Montant FROM deleted;

    INSERT INTO Finances.AuditFacture (FactureID, MontantAvant, MontantApres, TypeModification, DateMAJ)
    VALUES (@FactureID, @MontantAVANT, @MontantAPRES, 'UPDATE', GETDATE());
END
GO
```

# UPDATE(column)

- ❖ Pour vérifier si une colonne précise a été mise à jour, vous pouvez faire un test en utilisant le fait qu'**UPDATE (column)** retourne **TRUE** si la colonne a été mise à jour.
- ❖ Ex: **IF (UPDATE(prixDemandé) )**  
**BEGIN....END**

# Trigger AFTER UPDATE complet:

```
GO
CREATE TRIGGER Finances.trg_uFacture
ON Finances.Facture
AFTER UPDATE
AS
BEGIN
-- inserted APRÈS FactureID, ClientID, Montant, Taxes, EmployeID
-- deleted AVANT FactureID, ClientID, Montant, Taxes, EmployeID
    IF(UPDATE(Montant))
    BEGIN
        DECLARE @FactureID int;
        DECLARE @MontantAVANT money, @MontantAPRES money;

        SELECT @FactureID=FactureID, @MontantAPRES=Montant FROM inserted;
        SELECT @MontantAVANT=Montant FROM deleted;

-- Champs de AuditFacture FactureID, MontantAvant, MontantApres, TypeModification, DateMAJ
        INSERT INTO Finances.AuditFacture (FactureID, MontantAvant, MontantApres, TypeModification, DateMAJ)
        VALUES (@FactureID, @MontantAVANT, @MontantAPRES, 'UPDATE', GETDATE());
    END
END
GO
```

# Pour tester:

Données des tables Facture et AuditFacture AVANT l'UPDATE

La commande UPDATE sur la table Facture

Données des tables Facture et AuditFacture APRÈS l'UPDATE

```
--Test trigger
--Enregistrement de la table Facture dont le montant sera modifié
SELECT 0 AS [AVANT UPDATE], FactureID, Montant FROM Finances.Facture WHERE FactureID=1;
--Dernier enregistrement de la table AuditFacture
SELECT TOP(1) 0 AS [AVANT UPDATE], AuditFactureID, FactureID, MontantAvant, MontantAprès, TypeModification, DateMAJ
FROM Finances.AuditFacture
ORDER BY AuditFactureID DESC
GO
--Modification du montant de la facture no 1
UPDATE Finances.Facture
SET Montant=4000, Taxes=600
WHERE FactureID=1;
GO
--Enregistrement de la table Facture dont le montant a été modifié
SELECT 0 AS [APRÈS UPDATE], FactureID, Montant FROM Finances.Facture WHERE FactureID=1;
--Derniers enregistrements de la table AuditFacture
SELECT TOP(2) 0 AS [APRÈS UPDATE], AuditFactureID, FactureID, MontantAvant, MontantAprès, TypeModification, DateMAJ
FROM Finances.AuditFacture
ORDER BY AuditFactureID DESC
```

Résultats		Messages					
AVANT UPDATE		FactureID	Montant				
1	0	1	1000,00				
AVANT UPDATE		AuditFactureID	FactureID	MontantAvant	MontantAprès	TypeModification	DateMAJ
1	0	1	1	NULL	1000,00	INSERT	2023-02-21 22:12:07.410
APRÈS UPDATE		FactureID	Montant				
1	0	1	4000,00				
APRÈS UPDATE		AuditFactureID	FactureID	MontantAvant	MontantAprès	TypeModification	DateMAJ
1	0	2	1	1000,00	4000,00	UPDATE	2023-02-21 22:12:30.957
2	0	1	1	NULL	1000,00	INSERT	2023-02-21 22:12:07.410



# Trigger AFTER DELETE

Nous voulons que quand nous allons **SUPPRIMER** une facture dans la table **Facture**, le trigger **Finances.trg\_dFacture** se déclenche. Celui-ci insèrera dans la table **AuditFacture** un enregistrement, avec DELETE comme type de modification et la date de la suppression.





Voyons les infos qu'on veut entrer dans la table **AuditFacture** après la **suppression** d'une facture dans la table **Facture**:

Le champ **AuditFactureID** est un compteur qui s'augmente tout seul. On n'entre pas d'infos là.

DML	Table deleted (Image AVANT)	Table inserted (Image APRÈS)
Delete	Enregistrements supprimés	Vide

Le **FactureID** nous viendra de la table **deleted**

Le **MontantAvant** nous viendra de la table **deleted**

Le **MontantApres** sera NULL

Le **TypeModification** sera **'DELETE'**

La **DateMAJ** sera **GetDate()**

# Trigger AFTER DELETE complet:

```
GO
CREATE TRIGGER Finances.Factures_dtrgSupprimerFacture
ON Finances.Facture
AFTER DELETE
AS
BEGIN
    DECLARE @FactureID int;
    DECLARE @MontantAVANT money;

    SELECT @FactureID=FactureID, @MontantAVANT=Montant FROM deleted;

    INSERT INTO Finances.AuditFacture (FactureID, MontantAvant, MontantApres, TypeModification, DateMAJ)
    VALUES (@FactureID, @MontantAVANT, NULL, 'DELETE', GETDATE());
END
GO
```

# Pour tester:



Données des tables Facture et AuditFacture AVANT le DELETE

La commande DELETE sur la table Facture

Données des tables Facture et AuditFacture APRÈS le DELETE

```
--Test trigger
--Enregistrement de la table Facture qui sera supprimé
SELECT 0 AS [AVANT DELETE], FactureID, Montant FROM Finances.Facture WHERE FactureID=1;
--Derniers enregistrements de la table AuditFacture
SELECT TOP(2) 0 AS [AVANT DELETE], AuditFactureID, FactureID, MontantAvant, MontantAprès, TypeModification, DateMAJ
FROM Finances.AuditFacture
ORDER BY AuditFactureID DESC
GO
--Suppression de la facture no 1
DELETE FROM Finances.Facture
WHERE FactureID=1;
GO
--Enregistrement de la table Facture supprimé (devrait être vide)
SELECT 0 AS [APRÈS DELETE], FactureID, Montant FROM Finances.Facture WHERE FactureID=1;
--Derniers enregistrements de la table AuditFacture
SELECT TOP(2) 0 AS [APRÈS DELETE], AuditFactureID, FactureID, MontantAvant, MontantAprès, TypeModification, DateMAJ
FROM Finances.AuditFacture
ORDER BY AuditFactureID DESC
```

Résultats		Messages	
AVANT DELETE	FactureID	Montant	
0	1	4000,00	
AVANT DELETE	AuditFactureID	FactureID	MontantAvant
0	2	1	1000,00
0	1	1	NULL
AVANT DELETE	AuditFactureID	FactureID	MontantAprès
0	2	1	4000,00
0	1	1	1000,00
APRÈS DELETE	FactureID	Montant	
APRÈS DELETE	AuditFactureID	FactureID	MontantAvant
0	3	1	4000,00
0	2	1	1000,00
APRÈS DELETE	AuditFactureID	FactureID	MontantAprès
0	3	1	NULL
0	2	1	4000,00
APRÈS DELETE	AuditFactureID	FactureID	TypeModification
0	3	1	DELETE
0	2	1	UPDATE
APRÈS DELETE	AuditFactureID	FactureID	DateMAJ
0	3	1	2023-02-21 22:25:13.500
0	2	1	2023-02-21 22:25:51.360



# Faut-il toujours faire des tests pour les déclencheurs?

Est-ce que vous mettriez du code en production sans avoir fait des tests?

Bien oui, il faut toujours faire des tests pour vérifier que vos déclencheurs fonctionnent comme vous le pensez.

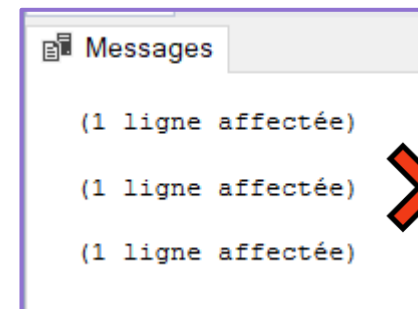




## ❖ Outils supplémentaires

### ◆ SET NOCOUNT ON;

- Cette instruction peut être glissée dans n'importe quel **déclencheur** ET **procédure stockée** qui modifie des rangées de données pour **EMPÊCHER** de donner un feedback sur le **nombre de lignes modifiées**.
- C'est une **excellente pratique** pour des raisons de **performance**. Ça semble banal, mais à grande échelle cette réponse peut ralentir considérablement les transactions réalisées sur une base de données.
  - Utilisez donc toujours **SET NOCOUNT ON** à moins que le nombre de lignes affectées soit un feedback nécessaire dans une certaine situation.





❖ Attention aux contraintes ou aux triggers que vous faites.

### ◆ Laisser de la flexibilité à la BD;

◆ Il faut faire attention à ce que les contraintes CHECK et les TRIGGERS que vous faites ne rendent pas trop difficile des modifications plus tard. Car il arrive que les règles d'affaire changent.

- Le fait qu'une école de ski ne démarre une classe que si elle a au moins 4 étudiants ne devrait pas être implémenté par une contrainte ni par un trigger car cela pourrait changer éventuellement.
- Le fait que le solde de la carte de crédit d'un client ne puisse pas dépasser sa limite de crédit est une bonne contrainte à implanter. C'est pas mal certain que cela ne changera jamais.



## ❖ Déclencheurs sur plusieurs instructions DML



- ❖ Nous avons vu des déclencheurs sur une seule instruction DML
  - ◆ AFTER INSERT
  - ◆ AFTER UPDATE
  - ◆ AFTER DELETE
  
- ◆ Parfois, selon les normes des entreprises, on veut faire un seul trigger AFTER par table
  - Donc si le déclencheur doit se déclencher après un INSERT et après un UPDATE, on aura
  - **AFTER INSERT, UPDATE**





- ❖ Les tables **inserted** et **deleted** nous permettront d'identifier quelle instruction **DML** a déclenché l'exécution du trigger:

DML	deleted (AVANT)	inserted (APRES)
Insert	Vide	Enregistrements insérés
Update	Enregistrements AVANT la mise à jour	Enregistrements mis-à-jour
Delete	Enregistrement AVANT la suppression	Vide

```
DECLARE @action CHAR(1);

SET @action =
CASE
    WHEN NOT EXISTS (SELECT * FROM deleted) AND EXISTS(SELECT * FROM inserted) THEN 'I'
    WHEN EXISTS(SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted) THEN 'U'
    WHEN EXISTS (SELECT * FROM deleted) AND NOT EXISTS(SELECT * FROM inserted) THEN 'D'
END
```



```
CREATE TRIGGER Biens.trg_iuBien
ON Biens.Bien
AFTER INSERT, UPDATE
AS
BEGIN
    DECLARE @action CHAR(1);

    SET @action =
    CASE
        WHEN EXISTS(SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted) THEN 'U'
        WHEN EXISTS(SELECT * FROM inserted) AND NOT EXISTS (SELECT * FROM deleted) THEN 'I'
    END

    IF @action = 'I'
    BEGIN
        INSERT INTO Biens.HistoriqueBienPrix (BienID, DatePrixDemande, PrixDemande)
        SELECT BienID, DateInscription, PrixDemande FROM inserted
    END
END
```



## ❖ Transactions

- ◆ Une transaction est un ensemble d'opérations (« Unit of work ») effectuées sur une base de données.
- ◆ Le but derrière l'utilisation des transactions est de préserver la **cohérence** et l'**intégrité** d'une base de données.
  - Par exemple, on s'assure qu'une série d'opérations intimement liées sont **toutes réussies** ou **toutes échouées**. (Mais surtout pas réussies partiellement !)
- ◆ Les transactions sont souvent mises de l'avant avec les 4 qualités « **ACID** » suivantes :
  1. **Atomicité** : Toutes les opérations dans une même transaction sont considérées comme une seule « unité » de travail. Soit tout réussi, soit tout échoue.
  2. **Cohérence** : La base de données est dans un état cohérent une fois la transaction complétée.
  3. **Isolation** : Les changements générés par une transaction sont invisibles tant que la transaction n'est pas « **COMMIT** ».
  4. **Durabilité** : Une fois qu'une transaction est « **COMMIT** », les changements sont permanents et la transaction ne peut pas être annulée. (Pas de **ROLLBACK**)



## ❖ Exemple de transaction

◆ **BEGIN TRANSACTION** et **COMMIT TRANSACTION** délimitent le début et la fin de la transaction.

```
SELECT CompteID, NoCompte, TypeCompte, Solde, ClientID FROM dbo.CompteBancaire

BEGIN TRANSACTION
    UPDATE dbo.CompteBancaire
    SET Solde -= 500
    WHERE TypeCompte = 'Cheque' AND ClientID = 1
    UPDATE dbo.CompteBancaire
    SET Solde += 500
    WHERE TypeCompte = 'Epargne' AND ClientID = 1
COMMIT TRANSACTION

SELECT CompteID, NoCompte, TypeCompte, Solde, ClientID FROM dbo.CompteBancaire
```

100 %

Résultats Messages

	CompteID	NoCompte	TypeCompte	Solde	ClientID
1	1	1234565678781234	Cheque	2500.00	1
2	2	2345234534213456	Epargne	45000.00	1

	CompteID	NoCompte	TypeCompte	Solde	ClientID
1	1	1234565678781234	Cheque	2000.00	1
2	2	2345234534213456	Epargne	45500.00	1

On met les 2 UPDATE dans une transaction parce qu'on veut s'assurer que si on sort un montant de notre compte Chèque, on va bel et bien avoir ce montant déposé dans notre compte d'Épargne.

Si jamais il y a un problème, les données seront remises dans leur état initial.



## ❖ Transaction **non COMMIT**

◆ Si on **omet** le **COMMIT** qui conclut une transaction, les rangées de données touchées par la transaction sont « **LOCK** ». (Et c'est tant mieux !)


- Il devient impossible d'accéder aux rangées de données tant que la transaction n'a pas été COMMIT.

```
BEGIN TRANSACTION;
```

```
UPDATE Courses.Personnage  
SET Nom = 'Bébé Harmonie'  
WHERE Nom = 'Bébé Rosalina';
```

```
COMMIT TRANSACTION;
```

- Par exemple, si cette transaction n'avait pas été **COMMIT**, lancer les requêtes suivantes (dans une autre page SQL que celle de la transaction) « **bloquerait** » (impression de long chargement) aussi longtemps qu'il le faut, le temps que la transaction soit **COMMIT**.

 Exécution de la requête en cours...



- SELECT \* FROM Courses.Personnages;
- SELECT \* FROM Courses.Personnages WHERE Nom = 'Bébé Rosalina';
- SELECT \* FROM Courses.Personnages WHERE Nom = 'Bébé Harmonie';

- Dès que la transaction est finalement **COMMIT**, le **LOCK** est levé et les requêtes qui bloquaient peuvent s'exécuter.