

Rencontre 11

Déclencheurs (suite)

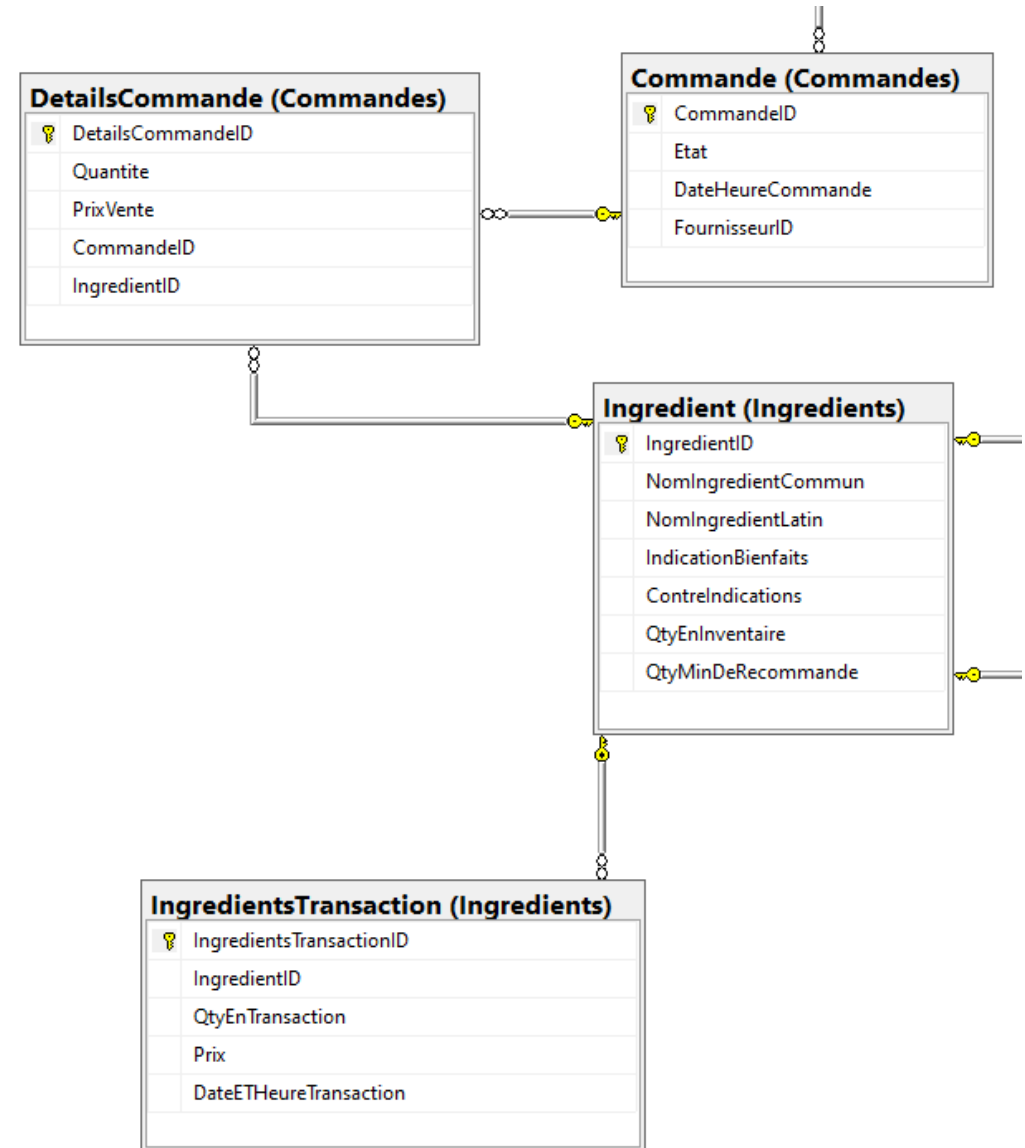
Bases de données et programmation Web



- ❖ **Un déclencheur complexe**
- ❖ Retour sur les standards de nommage
- ❖ Exemples d'autres déclencheurs
- ❖ Déclencheurs `INSTEAD OF`

Nous sommes un salon de thé et nous passons régulièrement des commandes à nos fournisseurs.

À la réception de la commande, nous voulons augmenter la quantité en inventaire des produits qui sont dans la commande reçue.



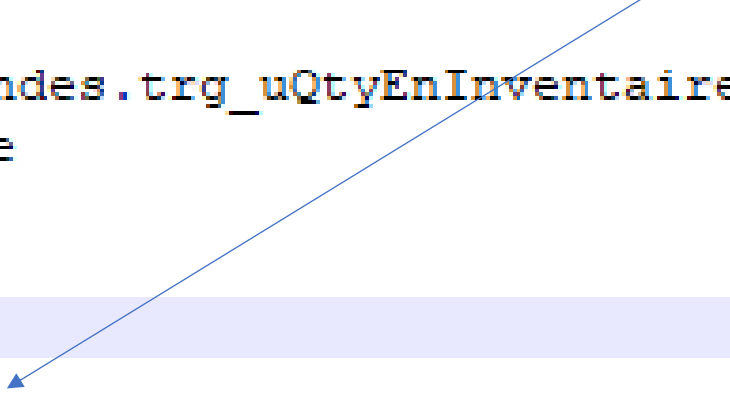


Actions	Valeur de l'état de la commande	Ce qu'on veut faire alors
Création d'une commande Et des enregistrements des détails de la commande	ETAT = 'passée'	
Notre fournisseur nous avise que la commande est envoyée	ETAT = 'attenteDeLivraison'	
On reçoit notre commande	ETAT = 'livrée'	On veut alors augmenter la quantité en inventaire des produits de notre commande. On veut aussi ajouter à la table Ingredients.IngredientsTransaction les modifications faites aux quantités en inventaires des produits de notre commande.



- ❖ Étape 1 : Créer le bon type de trigger, sur la bonne table et vérifier qu'il y a eu une modification du champ **Etat**

```
GO
CREATE TRIGGER Commandes.trg_uQtyEnInventaire
ON Commandes.Commande
AFTER UPDATE
AS
[BEGIN
    IF (UPDATE (Etat))
    BEGIN
```

A blue arrow originates from the text 'Étape 1 : Créer le bon type de trigger, sur la bonne table et vérifier qu'il y a eu une modification du champ Etat' and points to the 'IF (UPDATE (Etat))' line in the SQL code. A light blue horizontal bar highlights the 'BEGIN' line in the code.



- ❖ Étape 2 : Aller chercher la valeur de CommandeID et de Etat de la table **inserted** et vérifier que la valeur de Etat est 'Livrée'

...

```
DECLARE @CommandeID int, @Etat nvarchar(20);  
SELECT @CommandeID=CommandeID, @Etat=Etat FROM inserted;  
  
IF (@Etat = 'Livrée')  
BEGIN  
    ...  
END
```

...



- ❖ Étape 3 : Mettre à jour les QtyEnInventaire des produits qu'on a reçu dans notre commande.

...*On a besoin d'un update avec jointure:*

```
UPDATE Ingredients.Ingredient
SET QtyEnInventaire += DC.Quantite
FROM Ingredients.Ingredient I
INNER JOIN Commandes.DetailsCommande DC
ON DC.IngredientID = I.IngredientID
WHERE CommandeID=@CommandeID
```

Nécessaire pour savoir
quels produits doivent
augmenter leur
quantité en inventaire,
bref ceux qui sont dans
la bonne commande.

...



- ❖ Étape 4 : Insérer dans notre table Ingredients.IngredientsTransaction les modifications faites aux QtyEnInventaire suite à notre commande qu'on a reçue.

...

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

} Nos données à insérer
proviennent de la table
DetailsCommande.

...



❖ Étape 5 : Tests Tests Tests Tests Tests Tests

- TESTS TESTS TESTS TESTS
- testez votre déclencheur en insérant une commande avec l'état 'passée' et des détails pour cette commande
- Montrez la quantité en inventaire des produits de la commande (qu'on n'a pas encore reçus)
- Faites un UPDATE sur Commandes.Commande pour changer l'état de la commande à 'AttenteDeLivraison'
- Montrez la quantité en inventaire des produits de la commande n'ont pas changés (car on ne les a pas encore reçus)
- Faites un UPDATE sur Commandes.Commande pour changer l'état de la commande à 'Livrée'
- Montrez que la quantité en inventaire des produits de la commande ont changés(car on les a reçus)
- Faites la requête pour vérifier le contenu de Ingredients.IngredientsTransaction par la suite



❖ Étape 5 : Tests Tests Tests Tests Tests Tests

```
-- testez votre déclencheur en insérant une commande avec l'état 'passée' et des détails pour cette commande
INSERT INTO Commandes.Commande (Etat, DateHeureCommande, FournisseurID)
VALUES ('passée', GETDATE(), 1)
GO
INSERT INTO Commandes.DetailsCommande ( Quantite, PrixVente, CommandeID, IngredientID)
VALUES (10, 50, 1, 1),
       (100, 50, 1, 2)
-- Montrez la quantité en inventaire des produits de la commande (qu'on n'a pas encore reçus)
SELECT IngredientID, QtyEnInventaire
FROM Ingredients.Ingredient
WHERE IngredientID IN (1,2)
-- Faites un UPDATE sur Commandes.Commande pour changer l'état de la commande à 'AttenteDeLivraison'
UPDATE Commandes.Commande
SET Etat = 'AttenteDeLivraison'
WHERE CommandeID = 1
-- Montrez la quantité en inventaire des produits de la commande n'ont pas changés (car on ne les a pas encore reçus)
SELECT IngredientID, QtyEnInventaire
FROM Ingredients.Ingredient
WHERE IngredientID IN (1,2)
-- Faites un UPDATE sur Commandes.Commande pour changer l'état de la commande à 'Livrée'
UPDATE Commandes.Commande
SET Etat = 'Livrée'
WHERE CommandeID = 1
-- Montrez que la quantité en inventaire des produits de la commande ont changés(car on les a reçus)
SELECT IngredientID, QtyEnInventaire
FROM Ingredients.Ingredient
WHERE IngredientID IN (1,2)
-- Faites la requête pour vérifier le contenu de Ingredients.IngredientsTransaction par la suite
SELECT *
FROM Ingredients.IngredientsTransaction
WHERE IngredientID IN (1,2)
```



❖ Étape 5 : Tests Tests Tests Tests Tests Tests

	IngredientID	QtyEnInventaire
1	1	500
2	2	600

QtyEnInventaire initiale des produits de ma commande

	IngredientID	QtyEnInventaire
1	1	500
2	2	600

QtyEnInventaire inchangée après que l'Etat de la commande passe de 'Passée' à 'AttenteDeLivraison'

	IngredientID	QtyEnInventaire
1	1	510
2	2	700

QtyEnInventaire modifiée après que l'Etat de la commande passe à 'Livrée'

	IngredientsTransactionID	IngredientID	QtyEnTransaction	Prix	DateETHeureTransaction
1	1	1	10	4.25	2025-02-24 16:24:18.893
2	2	1	4	8.00	2025-02-24 16:24:18.893
3	3	1	10	50.00	2025-02-24 16:24:28.460
4	4	2	100	50.00	2025-02-24 16:24:28.460



Insertion des modifications à la qty en inventaire des produits dans notre table Ingredients.IngredientTransaction



- ❖ Retour sur un déclencheur complexe
- ❖ **Retour sur les standards de nommage**
- ❖ Exemples d'autres déclencheurs
- ❖ Déclencheurs `INSTEAD OF`



- ❖ Tous les objets de la BD ont des standards de nommage avec des préfix (sauf les tables).
 - ◆ Nous avons vu les préfix pour les contraintes (PK, FK, UC, CK, DF)
 - ◆ Pour une vue: **vw_**
 - ◆ Pour une fonction: **ufn_**, **udf_**
 - ◆ Pour une procédure stockée: **usp_**
 - ◆ Pour un déclencheur **trg_** , **trg_i**, **trg_u**, **trg_d**, **trg_iu**

- ❖ De plus, **tous** les objets de la BD sont dans des **schémas**
 - ◆ CREATE VIEW **Recettes.vw_RecetteCategorieTheme**
 - ◆ CREATE PROCEDURE **Fournisseurs.usp_CompterNbIngredientsParFournisseur**
 - ◆ CREATE TRIGGER **Ingredients.trg_iAugmenterQtyEnInventaire**



- ❖ Retour sur un déclencheur complexe
- ❖ Retour sur les standards de nommage
- ❖ **Exemples d'autres déclencheurs**
- ❖ Déclencheurs `INSTEAD OF`

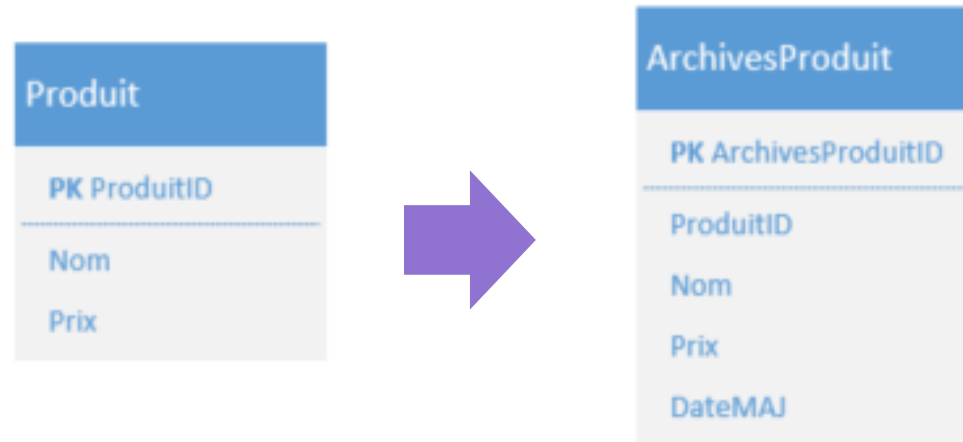


- ❖ Maintenant que vous connaissez le mécanisme des déclencheurs, nous vous donnons divers exemples de cas où des déclencheurs seront très utiles (sans les tests cette fois-ci)
 - ◆ Exemple #1 : Quand un produit est **supprimé**, on l'ajoute automatiquement dans une table spéciale servant d'archives pour les produits supprimés.
 - ◆ Exemple #2: Dans un jeu en équipe, quand un joueur marque un point, on doit également mettre à jour le pointage total de l'équipe, qui est une valeur **dérivée**.
 - ◆ Exemple #3: Reproduire le comportement de **ON DELETE CASCADE** pour une table avec deux **FK** ou plus pointant vers la même **PK**.



❖ Exemple #1

- ◆ Quand un produit est **supprimé**, on l'ajoute automatiquement dans une table spéciale servant d'archives pour les produits supprimés.
 - Pour ce genre d'opération, il est préférable d'octroyer une **nouvelle clé primaire** (avec **IDENTITY(1,1)** par exemple) pour être sûr que dans tous les cas, les rangées seront uniques et identifiables.
 - Si les données risquent d'être réutilisées ou que l'on doive préserver des relations avec d'autres entités, un « **SOFT DELETE** » peut être préférable. (Ex : ajouter une colonne de type **bit** nommée **EstSupprime** dans la table pour spécifier si un produit existe encore)





❖ 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**)

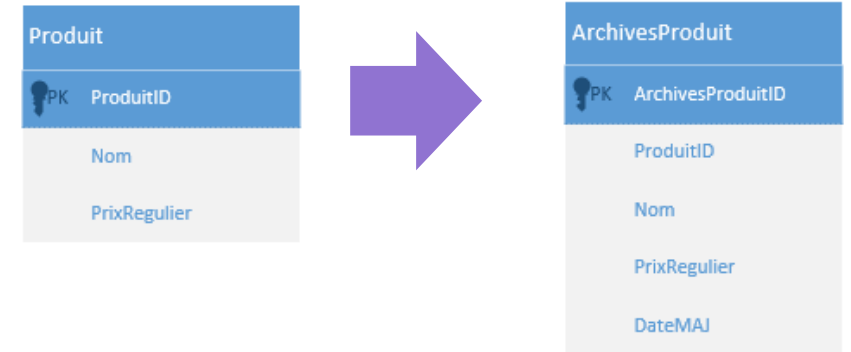
La table ArchivesProduitSpec contiendra (**ArchivesProduitSpecID**, **ProduitSpecID, Largeur, Hauteur, Poids, ProduitID, DateMAJ**)

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



❖ Exemple #1

```
GO
CREATE TRIGGER ExTriggers.trg_dProduitEtArchiver
ON ExTriggers.Produit
AFTER DELETE
AS
BEGIN
    INSERT INTO ExTriggers.ArchivesProduit (ProduitID, Nom, PrixRegulier, DateMAJ)
    SELECT ProduitID, Nom, PrixRegulier, GETDATE()
    FROM deleted
END
GO
```



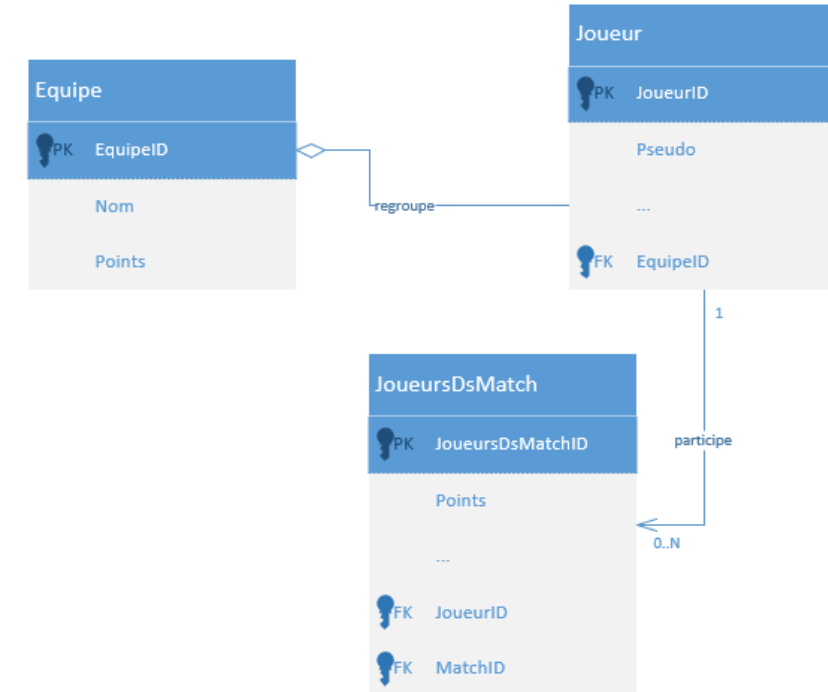
- Dans le cas d'un déclencheur activé à la suite d'une opération **INSERT** ou **UPDATE**, il existe une **table temporaire spéciale** similaire nommée **inserted**. (Même pour un **UPDATE**, la table avec les rangées modifiées s'appelle **inserted**)
- Gardez toujours à l'esprit que les tables **inserted** et **deleted** peuvent contenir plus d'une rangée car les opérations **INSERT**, **UPDATE** et **DELETE** peuvent impacter plus d'une rangée.

- **APRÈS (AFTER)** une opération **DELETE** sur la table **Produit**, on **INSERT** dans la table **ArchivesProduit** tous les produits qui sont dans la table « **deleted** ».
- La table « **deleted** » est une **table temporaire spéciale** qui contient toutes les rangées qui viennent d'être retirées par l'opération **DELETE**.



❖ Exemple #2

- ◆ Dans un jeu en équipe, quand un joueur marque un point dans un match, on doit également ajouter le point au pointage total de l'équipe, qui est une valeur **dérivée**.
 - Par exemple, si les quatre joueurs d'une équipe ont marqué 1, 2, 3 et 4 points, respectivement, l'équipe aura 10 points à la fin du match.
 - On décide donc de créer un déclencheur qui s'active lors d'un UPDATE sur la table JoueursDsMatch.





❖ Exemple #2

```
GO
CREATE TRIGGER ExTriggers.trg_uJoueursDsMatch
ON ExTriggers.JoueursDsMatch
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Points)
    BEGIN
        DECLARE @JoueurID int;
        SELECT @JoueurID FROM inserted;

        UPDATE ExTriggers.Equipe
        SET Points+=1
        WHERE EquipeID = (SELECT EquipeID
                           FROM ExTriggers.JoueurID
                           WHERE JoueurID = @JoueurID)
    END
END
GO
```

- APRÈS (AFTER) une opération UPDATE sur la table JoueursDsMatch (où on ajoute 1 point aux points d'un joueur qui vient de compter un but), le déclencheur UPDATE la table Equipe pour ajouter 1 point aux points de l'équipe associée au joueur qui a compté.
- La variable @JoueurID contient l'ID du joueur qui vient d'être modifié. (Trouvé dans la table spéciale inserted)
- L'opération UPDATE qui suit ajoute 1 point aux points de l'équipe du joueur avec l'ID @JoueurID.

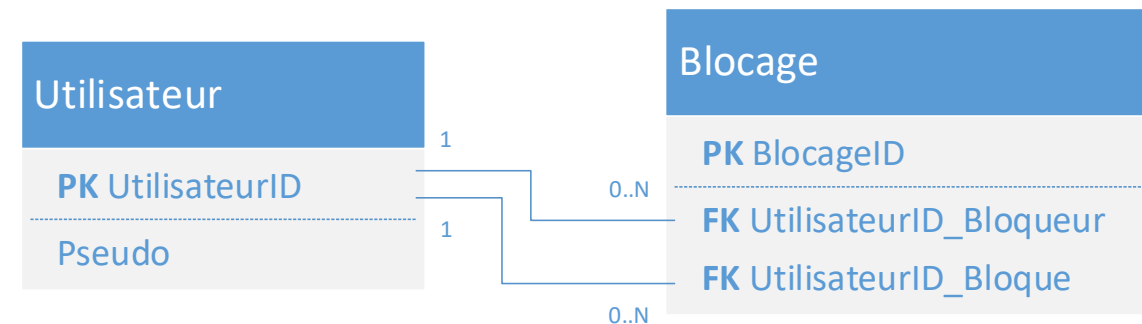
- **Attention !** Le fonctionnement de ce déclencheur implique plusieurs précautions :
 - Quand un Match commence (INSERT), les points de tous les joueurs du match doivent être initialisés à 0. (Car le déclencheur est seulement appelé lors d'un UPDATE). C'est probablement une contrainte par défaut qui fera cela.
 - Les UPDATE sur la table JoueursDsMatch ne peuvent **que cibler un joueur à la fois**. Ce qui ne devrait pas être un problème ici car c'est toujours un seul joueur qui compte un but à la fois. (Le déclencheur ici ne marche pas si la table temporaire inserted contient plusieurs rangées modifiées)



❖ Exemple #3

- ◆ Reproduire le comportement de **ON DELETE CASCADE** pour une table avec deux **FK** ou plus pointant vers la même **PK**.
 - Nous avons vu que **ON DELETE CASCADE** était parfois impossible à implémenter à l'aide d'une contrainte **FK**.
 - Quand on **supprime un utilisateur**, tous les **blocages** où cet utilisateur est le **bloqueur** OU le **bloqué** doivent être **supprimés** de la base de données.

```
ALTER TABLE GestionUtilisateurs.Blocage ADD CONSTRAINT FK_Blocage_UtilisateursID_Bloqueur  
FOREIGN KEY (UtilisateurID_Bloqueur) REFERENCES GestionUtilisateurs.Utilisateur(UtilisateurID)  
ON DELETE CASCADE } À retirer  
ON UPDATE CASCADE }  
ALTER TABLE GestionUtilisateurs.Blocage ADD CONSTRAINT FK_Blocage_UtilisateursID_Bloque  
FOREIGN KEY (UtilisateurID_Bloque) REFERENCES GestionUtilisateurs.Utilisateur(UtilisateurID)  
ON DELETE CASCADE } À retirer  
ON UPDATE CASCADE }
```



```
Msg 1785, Niveau 16, État 0, Ligne 29  
Introducing FOREIGN KEY constraint 'FK_Blocage_UtilisateursID_Bloque' on table 'Blocage' may cause cycles or multiple cascade paths.  
Msg 1750, Niveau 16, État 1, Ligne 29
```



❖ DÉCLENCHEUR **INSTEAD OF DELETE**

- ◆ Pour régler le problème de l'exemple 3, on va utiliser un nouveau type de déclencheur: le déclencheur **INSTEAD OF DELETE**
- ◆ Avec **INSTEAD OF DELETE**, l'instruction **DELETE** initiale **ne se fera pas**. C'est le code qui sera dans le déclencheur qui sera exécuté à la place.
- ◆ Dans ce code:
 - Parfois on va vouloir faire un '**Soft delete**': Ne pas supprimer l'enregistrement mais indiquer dans un champ de type bit que le produit est discontinué par exemple, ou que le client est inactif...
 - Parfois on va vouloir faire un vrai delete finalement mais seulement après avoir fait des actions auparavant.



- ❖ Retour sur un déclencheur complexe
- ❖ Retour sur les standards de nommage
- ❖ Exemples d'autres déclencheurs
- ❖ **Déclencheurs `INSTEAD OF`**

INSTEAD Triggers



- ❖ **INSTEAD trigger** : déclenché une fois par instruction et non à chaque enregistrement qui aurait été touché par l'instruction.
- ❖ Vous ne pouvez créer qu'UN SEUL INSTEAD trigger sur une même table pour le même type d'instruction.





❖ Exemple #3 (Suppression de plusieurs enregistrements)

```
GO
CREATE TRIGGER ExTriggers.dtrg_Utilisateur_SuppressionCascade
ON ExTriggers.Utilisateur
INSTEAD OF DELETE
AS
BEGIN;
    SET NOCOUNT ON;
    DELETE FROM ExTriggers.Blocage
    WHERE UtilisateurID_Bloqueur = ANY(SELECT UtilisateurID FROM deleted)
    OR UtilisateurID_Bloque = ANY(SELECT UtilisateurID FROM deleted);

    DELETE FROM ExTriggers.Utilisateur
    WHERE UtilisateurID = ANY(SELECT UtilisateurID FROM deleted);
END;
GO
```

- Cette fois-ci, on est obligé de créer un déclencheur de type **INSTEAD OF** plutôt que **AFTER**. C'est parce que si on essaye de **DELETE** un **Utilisateur**, nos contraintes **FK** vont **lancer une erreur** et annuler l'opération. Le déclencheur va donc **D'ABORD** supprimer les rangées impactées dans la table **Blocage** **PUIS** supprimer le ou les utilisateurs.

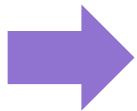
- Avec un déclencheur **INSTEAD OF**, l'opération est **REPLACÉE** par le code du **déclencheur**. Si on veut vraiment que les utilisateurs soient supprimés, il faudra **répéter l'opération** dans le **déclencheur**.

- Le premier **DELETE** supprime toutes les rangées de la table **Blocage** où l'id d'un utilisateur bloqueur ou bloqué correspond à un utilisateur qui vient d'être supprimé.

- Le deuxième **DELETE** supprime tous les utilisateurs de la table **Utilisateur** qui devaient être supprimés.

- Grâce à **ANY**, ce déclencheur peut gérer les opérations **DELETE** où plusieurs rangées sont affectées.

UtilisateurID	Pseudo
2	Martin3
3	Simone48

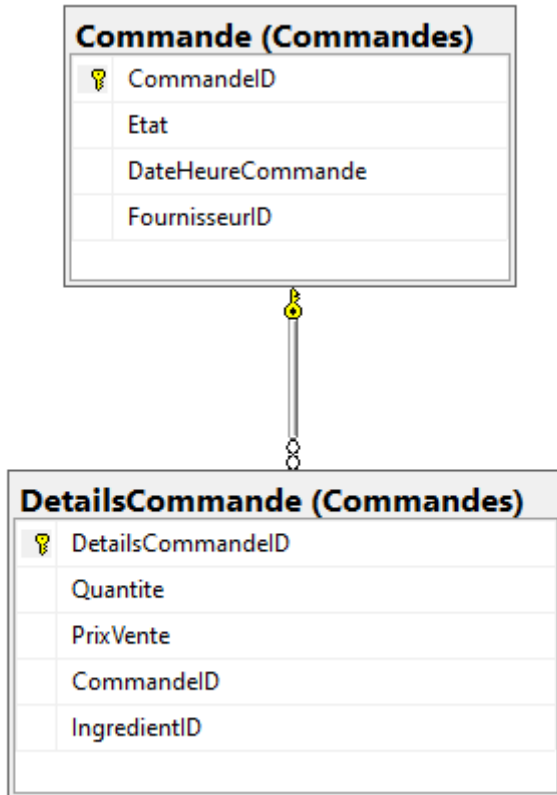


UtilisateurID_Bloqueur	UtilisateurID_Bloque
3	2

Si on supprime **Martin3** ou **Simone48**, il faut supprimer le **blocage** qui les relie.



❖ Exemple 4: **INSTEAD OF DELETE** pour une seule commande



- **Quand vous avez une relation 1-N entre deux tables et que vous n'avez pas mis une contrainte de clé étrangère avec ON DELETE CASCADE, vous aurez à faire un INSTEAD OF DELETE pour garder l'intégrité des données.**

- Ainsi, ce sera le cas pour une commande et le détail de cette commande car les entreprises doivent garder leurs infos plusieurs années pour des fins de vérification fiscale.

- On aura probablement à archiver les informations dans des tables d'archives AVANT de pouvoir supprimer une commande et le détail de cette commande.

- Avec un déclencheur INSTEAD OF, l'opération est REMPLACÉE par le code du déclencheur. Si on veut vraiment que les commandes soient supprimées, il faudra répéter l'opération dans le déclencheur.



❖ ATTENTION: ON supprime une seule commande

```
GO
;CREATE TRIGGER Commandes.dtrg_SupprimerUneCommande
ON Commandes.Commande
INSTEAD OF DELETE
AS
;BEGIN
    SET NOCOUNT ON;
    DECLARE @CommandeID int;
    SELECT @CommandeID = CommandeID FROM deleted;

    ; INSERT INTO Commandes.ArchivesCommande (CommandeID, Etat, DateHeureCommande, FournisseurID, DateMAJ)
    ; SELECT CommandeID, Etat, DateHeureCommande, FournisseurID, GETDATE()
    ; FROM Commandes.Commande
    ; WHERE CommandeID = @CommandeID

    ; INSERT INTO Commandes.ArchivesDetailsCommande (DetailsCommandeID, Quantite, PrixVente, CommandeID, IngredientID, DateMAJ)
    ; SELECT DetailsCommandeID, Quantite, PrixVente, CommandeID, IngredientID, GETDATE()
    ; FROM Commandes.DetailsCommande
    ; WHERE CommandeID = @CommandeID

    ; DELETE FROM Commandes.DetailsCommande -- on delete les données de l'entité N de la relation
    ; WHERE CommandeID = @CommandeID

    ; DELETE FROM Commandes.Commande -- on delete les données de l'entité 1 de la relation
    ; WHERE CommandeID = @CommandeID
END
GO
```

- Archiver les informations dans des tables d'archives **AVANT** de pouvoir supprimer une commande et le détail de cette commande.

- Le **premier DELETE** supprime toutes les rangées de la table DetailsCommande pour l'id de la commande qu'on veut supprimer.

- Le **deuxième DELETE** supprime la commande de la table Commande qu'on veut supprimer