

MICROSOFT SQL SERVER

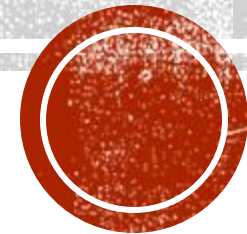


Prof. Maria EL HAIBA

Docteur en Informatique



*Email: **m.elhaiba@emsi.ma***



4

INTRODUCTION AU LANGAGE T-SQL (PART 2)

- Éléments du langage T-SQL (Variables, déclaration, affectation,...)
- Les structures de contrôle
- **Les curseurs, Les transactions**
- Les procédures stockées, Les fonctions système
- La gestion des exceptions, les déclencheurs

Curseurs

SECTION 1



Les Curseurs

➤ Définition

- Un curseur dans SQL Server est un **mécanisme de base de données** qui nous permet de **récupérer chaque** ligne à la fois et de **manipuler** ses données.
- Autrement dit, c'est une **zone mémoire** utilisée par le SGBD pour récupérer un **ensemble de résultats** issu d'une **requête SELECT**.
- Ainsi, un curseur est **toujours utilisé** conjointement avec une **instruction SELECT**.
- Pour **utiliser** un curseur, nous avons besoin de le **déclarer** (**SANS le symbole @ devant son nom**).
- Syntaxe :


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



Exemple: Déclaration

- Le curseur **CUR1** contiendra le **résultat** de la requête : `SELECT NumArt, PUArt FROM Article;`

```
DECLARE  
cur1 CURSOR FOR  
SELECT NumArt, PUArt FROM Article ;
```



NumArt	PUArt
1	12
2	20
3	5
4	7000
5	500
10	10000

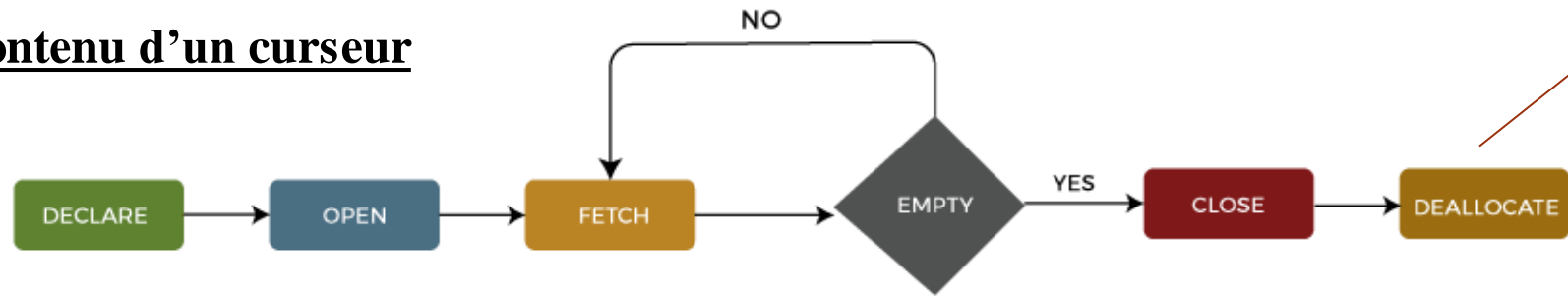
- Les données sont **disponibles** dans le curseur au moment de sa déclaration.
- Il y a **un pointeur** avant la première ligne du curseur.

L'intérêt d'un **curseur SQL Server** est de mettre à jour les données **ligne par ligne**, de les modifier ou d'effectuer des calculs qui ne sont pas possibles lorsque nous récupérons tous les enregistrements en même temps.



Les Curseurs

➤ Lecture du contenu d'un curseur



Cycle de vie
d'un curseur

1. Ouvrir le curseur avec **OPEN** (Bien évidemment après **déclaration** du curseur).
2. Lire le curseur avec la commande **FETCH NEXT FROM.....INTO** qui permet de lire l'enregistrement suivant (Une seule ligne à la fois)
3. Utiliser une **boucle WHILE** pour parcourir l'ensemble actif des enregistrements du curseur, et qui s'arrête lorsque le curseur est **VIDE**.
4. Fermer le curseur avec la commande **CLOSE**
5. Supprimer la référence au Curseur avec **DEALLOCATE**



La lecture du contenu d'un curseur se fait en transférant son contenu dans des variables et on lit les variables



Syntaxe de lecture d'un curseur

```
DECLARE liste_variables;
```

Déclarer le curseur avec son nom

```
DECLARE nom_curseur CURSOR  
FOR SELECT_statement ;
```

Ouvrir le curseur

```
OPEN nom_curseur;
```

Initialiser les variables auparavant déclarées
avec le premier FETCH (la première ligne)

```
FETCH NEXT FROM nom_cursor INTO liste_variables;
```

Tant que le FETCH se fait normalement,
parcourir tous les autres enregistrements

```
WHILE @@FETCH_STATUS = 0
```

```
BEGIN
```

```
    FETCH NEXT FROM nom_curseur INTO liste_variables;
```

```
END;
```

La variable système @@FETCH_STATUS : Renvoie
l'état de la dernière instruction FETCH effectuée sur un
curseur. Elle renvoie 0 si tout s'est bien passé,

```
CLOSE nom_curseur;
```

Fermer le curseur

```
DEALLOCATE nom_curseur;
```

Supprimer la référence du curseur et libérer les
ressources utilisées par ce curseur



Exemple: Curseur pour affichage

➤ Soit le programme suivant qui **affiche** les **numéros des articles** et leurs **prix** :

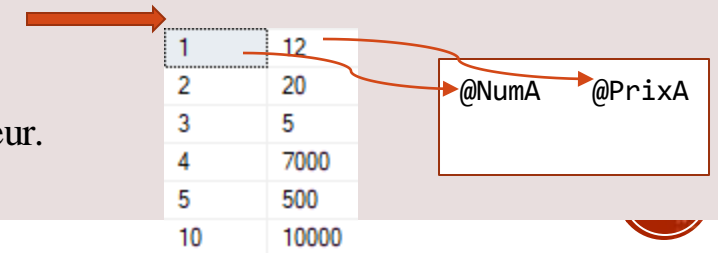
```
DECLARE @NumA int, @PrixA int;
DECLARE cur1 CURSOR FOR
    SELECT NumArt, PUArt FROM Article ;
BEGIN
OPEN cur1;
PRINT concat('Numéro', '---', 'Prix');
FETCH NEXT FROM cur1 INTO @NumA, @PrixA

WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT concat(@NumA, '-----', @PrixA);
        FETCH NEXT FROM cur1 INTO @NumA, @PrixA;
    END

CLOSE cur1;
DEALLOCATE cur1;
END;
```

Explication

- 1- Lorsque le curseur **cur1** est déclaré, il contient le résultat de la requête SELECT (Avec un pointeur avant la 1ère ligne du curseur).
- 2- On ouvre le curseur avec la commande OPEN.
- 3- La commande **FETCH NEXT FROM cur1 INTO @NumA, @PrixA** fait **avancer le pointeur** à la prochaine ligne. Après le **FETCH NEXT** la flèche orange avance à la première ligne du curseur.
- 4- La première ligne du curseur **contient des données**. Ces données sont **affectées** respectivement à @NumA, @PrixA
- 5- **Tant** que le curseur **n'est pas vide** (**WHILE @@FETCH_STATUS = 0**), on passe à la ligne suivante avec **FETCH NEXT FROM cur1 INTO @NumA, @PrixA**
- 6- On ferme le curseur
- 7- On supprime la référence au curseur.



1	12
2	20
3	5
4	7000
5	500
10	10000

Exemple: Curseur pour affichage

➤ Le résultat de l'affichage est le suivant :

Numéro	Prix
1	12
2	20
3	5
4	7000
5	500
10	10000



Toutefois

Il est **inutile** d'utiliser un curseur pour ne faire qu'**afficher** son contenu. La sortie de votre SELECT est suffisante. Il est même **déconseillé** d'utiliser un curseur **que pour** l'affichage du résultat.



- Ça occupe des ressources. Si vous oubliez le DEALLOCATE → **Problème**
- Si vous oubliez de fermer le curseur → **Problème**



Exemple: Curseur pour Update

- Soit le programme suivant qui met à jour le prix des articles :

```
DECLARE @Prix1 int;
DECLARE prix_cursor CURSOR FOR SELECT PUArt FROM Article ;
BEGIN
OPEN prix_cursor ;
FETCH NEXT FROM prix_cursor INTO @Prix1 ;
    WHILE @@FETCH_STATUS = 0
        BEGIN
            IF @Prix1 < 300 UPDATE Article
            set PUArt = PUArt+ (PUArt*0.1) WHERE CURRENT OF prix_cursor ;
            ELSE IF @Prix1 BETWEEN 300 AND 500 UPDATE Article
            set PUArt = PUArt+ (PUArt*0.05) WHERE CURRENT OF prix_cursor;
            ELSE UPDATE Article
            set PUArt = PUArt+ (PUArt*0.01) WHERE CURRENT OF prix_cursor;
            FETCH NEXT FROM prix_cursor INTO @Prix1;
        END;
CLOSE prix_cursor;
DEALLOCATE prix_cursor;
END;
```

Explication

- Même principe que l'exemple dernier. Mais au lieu d'afficher le contenu du curseur, on met à jour la base de données selon le contenu du curseur qui est le Prix des articles dans ce cas.
- La clause **CURRENT OF** permet une opération de mise à jour ou de suppression à la **position actuelle** du curseur sans qu'il soit nécessaire de spécifier une clause WHERE pour qualifier la ligne à mettre à jour.



Série 2 d'exercices : Curseurs

En gardant toujours la même base de données '**GestionC**' et en utilisant les **curseurs**:



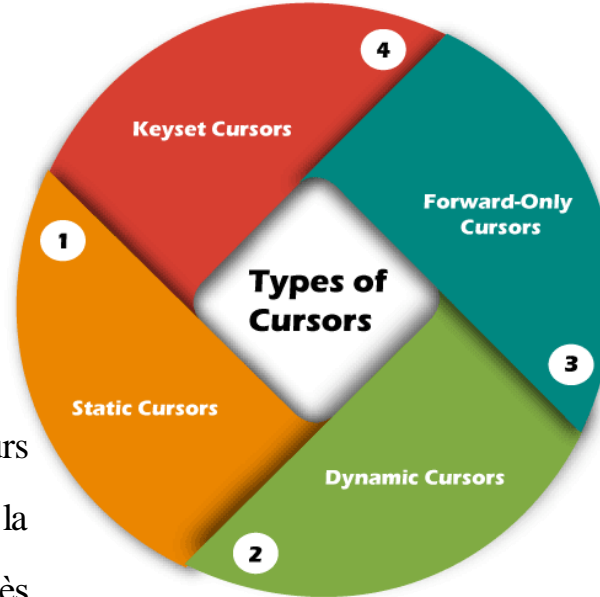
Voir Série N°2: Curseurs

- 1) Ecrire un programme qui affiche la liste des articles sous la forme
 - **L'article Numéro portant la désignationcoûte**
- 2) Ecrire un programme qui pour chaque commande affiche :
 - Le numéro et la date de commande sous la forme :
Commande N° : Effectuée le : ...
 - Le montant de cette commande, **Son montant est :**
- 3) Ecrire un programme qui pour **chaque commande vérifie** si **cette commande a au moins un article**.
 - **Si c'est le cas** : affiche la liste de ses articles (Numéro, Désignation et Prix) et la quantité commandée,
 - **Sinon** : affiche un message d'erreur : « *Aucun article pour la commande N°.... Elle sera supprimée* », Et supprime cette commande.

Types de curseurs SQL Server

Ensemble de valeurs clés (Keyset) : Chaque ligne du curseur est référencée par une clé, dans tempDB, permettant d'accéder aux données en temps réel à la lecture ou à la manipulation.

Statique : Ce curseur crée une copie statique, toujours en lecture seule, de toutes les lignes concernées de la base de données source dans tempDB. Il permet l'accès à partir d'une ligne dans différents sens.



À défilement en avant (Forward Only) : Ce type de curseur ne met, à la disposition de l'utilisateur, qu'une seule ligne à la fois avec un déplacement que vers la ligne suivante (accès séquentiel) .

Dynamique : Ce type de curseur permet la navigation et la prise en compte de tout type de modification sur les données des tables sous-jacentes. En revanche, il consomme plus de ressources système.



Déclaration - Types de Curseurs

➤ Pour **utiliser** un curseur d'un type donné, nous avons besoin de le **déclarer** et notamment le **spécifier**.

▪ Syntaxe :

```
DECLARE nomCurseur CURSOR Static FOR SELECT ... FROM ...  
                                Keyset  
                                Dynamic
```

Chacun est **plus adapté qu'un autre** selon le besoin voulu :

- Modes de navigation possibles (Déplacement vers l'avant / dans tous les sens)
- Prise en compte des mises-à-jour (Read-only / manipulation)
- ...



Curseur scrollable

- Par défaut, les **curseurs** sont **Forward ONLY** : ils ne sont pas scrollables (Options de parcours).
- 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 (Ex. : La ligne 3),*
 - *Une position relative à partir d'une position prédéfinie.*
- L'accès au contenu du curseur se fait directement à l'endroit souhaité (accès direct).



Fonctions du curseur scrollable

➤ Quelques fonctions :

- Atteindre le **premier** enregistrement du curseur `FETCH First FROM nom_curseur INTO variable1, variable2, ..`

- Atteindre le **dernier** enregistrement du curseur `FETCH Last FROM nom_curseur INTO variable1, variable2, ..`

- Atteindre l'enregistrement **précédent** du curseur **de celui en cours**

```
FETCH Prior FROM nom_curseur INTO variable1, variable2, ..
```

- Atteindre l'enregistrement **se trouvant à la position n** dans le curseur

```
FETCH Absolute n FROM nom_curseur INTO variable1, variable2, ..
```

- Atteindre l'enregistrement **se trouvant après n positions** de la **ligne en cours** (!!! Pas à la nième ligne)

```
FETCH Relative Num_Ligne FROM nom_curseur INTO variable1, variable2, ..
```



Exemple: Curseur scrollable

- Soit le **curseur scrollable** suivant :

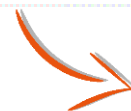
```
declare com_cli scroll cursor for select Nom, Prenom, Sum(PUArt*QteCommandee) as 'Montant'  
From Client Cl inner join Commande C on Cl.NumCl=C.NumCl  
inner join Com_Art CA on C.NumCom=CA.NumCom  
inner join Article A on A.NumArt=CA.NumArt  
Group by Nom, Prenom order by Montant desc
```

- Son **contenu** est le suivant :

Nom	Prenom	Montant
Oussama	Fihri	14000
Touria	Karam	400
Karim	Yahyaoui	240



Voir Série N°2: Curseurs



Ecrire le programme permettant d'**accéder directement aux lignes du curseur**
(Ex. : Première ligne).



Exemple: Curseur scrollable

- Le programme permettant d'accéder directement à la première ligne du curseur relatif aux clients (Nom, Prénom) avec le montant de leurs commandes par ordre décroissant, est le suivant :

```
begin
declare @nom varchar(30),@prenom varchar(30),@mt float;

declare com_cli scroll cursor for select Nom, Prenom, Sum(PUArt*QteCommandee) as 'Montant'
from Client Cl inner join Commande C on Cl.NumCl=C.NumCl
            inner join Com_Art CA on C.NumCom=CA.NumCom
            inner join Article A on A.NumArt=CA.NumArt
            Group by Nom, Prenom order by Montant desc

open com_cli;
print(' La première ligne');
fetch first from com_cli into @nom,@prenom, @mt;
print concat (@nom,'----', @prenom,'----', @mt);
close com_cli;
deallocate com_cli;
end;
```

Déclarer le curseur avec
l'attribut scroll

Atteindre la première ligne
du curseur

Résultat

```
La première ligne
Oussama----Fihri----14000
```



Exemple: Curseur scrollable

➤ Quelques fonctions :

- Avec le même code, mais on met: `fetch absolute 3 from com_cli into @nom,@prenom, @mt;`

- **Le résultat sera: Karim---Yahyaoui---240**

- Nous sommes à la 3ème ligne, puis on fait relative de -1

```
fetch absolute 3 from com_cli into @nom,@prenom, @mt;  
fetch relative -1 from com_cli into @nom,@prenom, @mt;
```

- **Le résultat sera:** (On recule de 1, par rapport à Karim) **Touria---Karam---400**

- Il est notamment possible d'être à la 3ème ligne, et faire relative de 2

```
fetch absolute 3 from com_cli into @nom,@prenom, @mt;  
fetch relative 2 from com_cli into @nom,@prenom, @mt;
```

- Dans ce cas **le résultat** avancera de 2 par rapport à Karim



À retenir



- 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**.
- La méthode **FETCH ..NEXT** permet de passer à **l'enregistrement suivant**. Dans ce cas, on parle **d'accès séquentiel**.
- La variable système **@@FETCH_STATUS** est utilisée pour **détecter la fin** du curseur. Tant que cette variable a la valeur 0, on a pas encore atteint la fin du curseur.
- Par défaut, les curseurs sont **Forward ONLY** et **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**.
- Les curseurs sont très pratiques de par leur **souplesse**, néanmoins, il faut les utiliser à **bon escient** et le **moins souvent possible**.





Transactions

SECTION 2



Les Transactions

➤ Contexte

- Paul veut acheter une Tablette



- Etapes pour effectuer l'achat:

1. Débiter argent sur le compte de Paul
2. Créditer argent sur le compte du vendeur
3. Diminuer le stocke des tablettes de 1

1. UPDATE Client SET solde=solde-1000 WHERE nom='Paul'
2. UPDATE Vendeur SET solde=solde+1000 WHERE nom='Vendeur'
3. UPDATE Article SET stock=stock-1 WHERE des='Tablette'



Les Transactions

➤ Contexte - Problèmes

- Lorsqu'on débute, on fait les requêtes **les unes après les autres**. Toutefois, il est possible que **certains problèmes surgissent** au moment de l'exécution :



- *Les requêtes échouent ;*
- *Panne pendant l'opération ;*
- *Arrêt du serveur des données ;*
- *Données incohérentes, violation d'une contrainte amenant le système à rejeter les opérations demandées ;*
- *Conflits avec des requêtes faites en parallèle ;*
- ...



Les Transactions

➤ Contexte - Vérification

- De ce fait, il est **nécessaire de vérifier** les points suivants :



- Le compte de Paul doit **bien être** débité
- Le compte du vendeur n'est crédité **qu'une** fois le compte de Paul débité
- Aucune valeur **incohérente ne doit être** générée
 - *Compte de Paul doit être positif*
 - *Stock des tablettes ne peut pas être négatif*
- Si un **problème** survient, l'**ensemble** des opérations doit être **annulé**

Pour cela, faire appel aux
TRANSACTIONS



Transactions - Définition

➤ Définition

- Une transaction dans SQL Server est un **bloc d'instructions** LMD (Select, Insert, Delete, Update) exécuté à la fois et qui laisse la base de données dans un état **cohérent**.
- Si pour une raison ou une autre, l'une de ces instructions n'a pas pu être exécutée, tout le groupe d'instructions est annulé (**le tout ou rien**)
- Autrement dit, Si une **seule instruction** dans le bloc **n'est pas cohérente** alors la transaction est **annulée**, et de ce fait, toutes les opérations LMD sont annulées.



Une **transaction** est une **unité logique de traitement** qui est soit complètement **exécutée**, soit complètement **abandonnée**



Transactions - Propriété ACID

➤ Propriété d'une Transaction: ACID

Atomicité

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



Isolation

Les modifications effectuées par des transactions concurrentes doivent **être isolées** transaction par transaction → **Système de Verrous**.



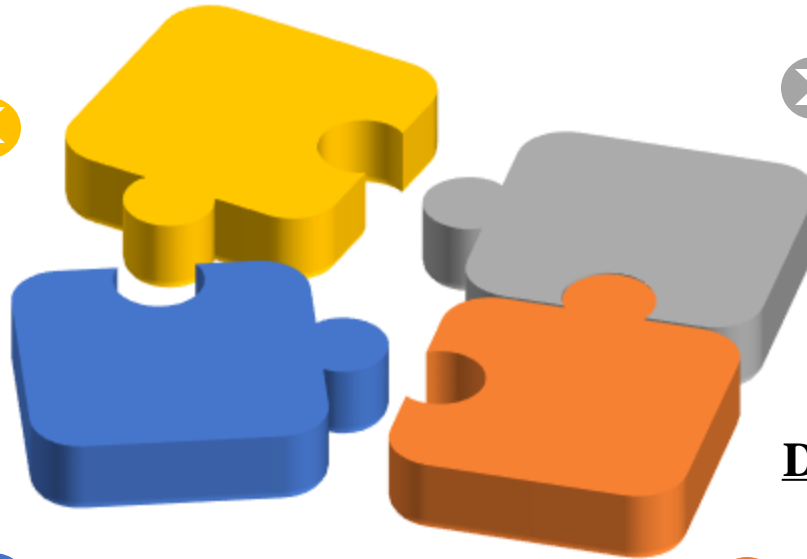
Cohérence

Lorsqu'elle est terminée, une transaction doit s'assurer de garder les données dans un **état cohérent**, et conforme aux **règles métiers**



Durabilité

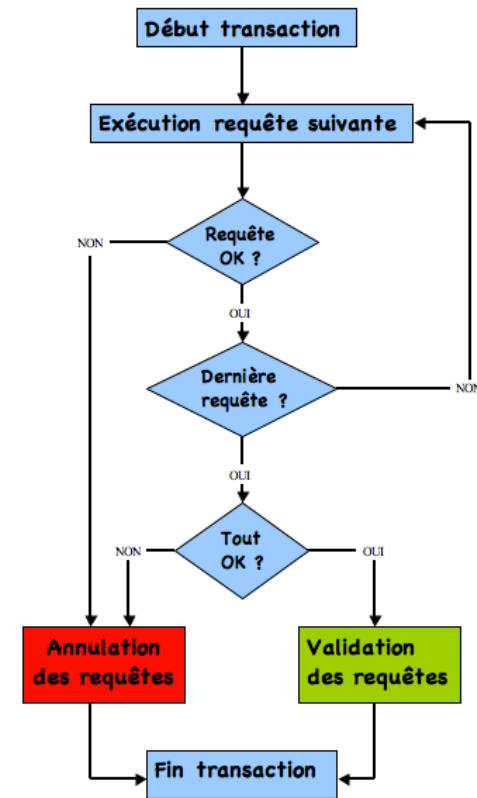
Lorsqu'une transaction **durable** est terminée, ses effets sur le système sont **permanents**. Les **modifications sont conservées** même en cas de défaillance du système



Transactions - Déroulement

➤ Déroulement d'une transaction

- On **démarre** une transaction.
- On **exécute les requêtes** désirées une à une.
- **Si** une des requêtes **échoue**, on **annule toutes** les requêtes, et on **termine** la transaction.
- Par contre, si à **la fin** des requêtes, **tout s'est bien passé**, on **valide tous** les changements, et on **termine** la transaction.
- Si le traitement est **interrompu** (entre deux requêtes par exemple), les changements **ne sont jamais validés**, et donc les données de la base restent les mêmes qu'avant la transaction.



Une transaction doit **NECESSAIREMENT** être soit **validée** ou **annulée**



Transactions - Principe

➤ Principe

- Une transaction débute par un **BEGIN TRANSACTION** et termine par un **COMMIT** ou un **ROLLBACK**.

- Syntaxe :

```
Begin Tran [Nom_Transaction]
...
If Condition
    RollBack Tran [Nom_Transaction]
...
Commit Tran [Nom_Transaction]
```



Si **plusieurs transactions** peuvent être en cours, on peut leur **attribuer des noms** pour les distinguer

Definition

- L'opération **COMMIT** détermine le point où la base de données est de nouveau cohérente (Permet **de valider la transaction** et rendre les traitements qui lui sont associés effectifs).
- 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 (Permet **d'interrompre une transaction** en cours qui n'a pas encore été validée).



Transactions - Exemple 1

- On souhaite supprimer la commande numéro 5 ainsi que la liste de ces articles. Le programme serait :

```
Delete from Commande where NumCom=5  
Delete from Com_Art where NumCom=5
```

Impossible: Conflit avec la contrainte
d'intégrité référentielle



- Si l'on suppose que la première instruction s'est bien exécutée, et alors que la deuxième n'a pas encore eu lieu, un problème survient : **Base de données incohérente** car on aura des lignes de commande pour une commande qui n'existe pas.
- En présence d'une transaction, le programme n'ayant pas atteint l'instruction Commit Tran, aurait annulé toutes les instructions depuis Begin Tran. Le programme devra être alors :

```
Begin Transaction  
Delete from Commande where NumCom=5  
Delete from Com_Art where NumCom=5  
Commit Transaction
```

Suppression temporaire jusqu'à validation.
Toutefois, **Exécution avec ERREUR**

Si la transaction ne contient pas de
Rollback, la deuxième instruction est
effectuée mais pas la première

Transactions - Bloc TRY... CATCH

- Il arrive que nous voulons que toutes les opérations à l'intérieur d'une transaction soit **TOUTES exécutées** si **toutes les opérations LMD sont correctes**, dans ce cas, il faudra les inclure à l'intérieur d'un **block TRY CATCH**.
- **Exemple 1:** Suppression d'une commande et la liste de ses articles

```
Begin Tran
Begin try
  Delete from Commande where NumCom=5
  Delete from Com_Art where NumCom=5
  Commit Tran
End try
Begin catch
  Rollback tran
  print 'impossible, instructions arrêtées';
End catch
```

Aucune instruction ne sera exécuté

Puisque la **première a été interrompue** suite au conflit d'intégrité, la deuxième n'a pas été effectuée : **Passage direct vers Rollback**



Transactions - Exemple 2

- Soit l'exemple relatif à l'achat d'une tablette. Le programme serait :

```
Begin Transaction achatArticle
-- Le compte de Paul est débité
UPDATE Client SET solde=solde-1000 WHERE nom='Paul'
-- Le compte du Vendeur est crédité
UPDATE Vendeur SET solde=solde+1000 WHERE nom='Vendeur'
-- Le stock est diminué
UPDATE Article SET stock=stock-1 WHERE des='Tablette'
Commit Transaction achatArticle
```



ATTENTION :

Nécessaire de vérifier si :

- *Compte de Paul doit être positif*
- *Stock des tablettes disponible*

- D'où le besoin d'ajouter des **CONDITIONS** et un **ROLLBACK** en cas d'erreur ou annulation



Transactions - Exemple 2

- Après prise en compte des conditions de vérification, le programme ressemblerait à ceci :

```
BEGIN TRANSACTION achatArticle
DECLARE @err int
SET @err= 0
--Le compte de Paul est débité
UPDATE client SET solde=solde-1000 WHERE nom='Paul'
IF(SELECT solde FROM client WHERE nom='Paul') < 0
SET @err= @err+ 1
...
/* Le reste des requêtes et des vérifications */
...
IF @err= 0
COMMIT TRANSACTION achatArticle
ELSE
ROLLBACK TRANSACTION achatArticle
```

Les transactions sont **beaucoup plus utiles** si nous les plaçons dans des instructions conditionnelles telles que IF .. ELSE.



Transactions - Mécanisme interne

➤ Mécanisme interne aux transactions

- SQL Server fonctionne par défaut en mode **AUTO COMMIT**, i.e. toute requête est automatiquement validée
- Tant **qu'on ne fait pas de COMMIT/ROLLBACK**, toutes les instructions internes à la transaction **sont stockées en mémoire** :
 - SQL server **trace** celles-ci dans son fichier journal **.ldf**
 - Il n'y a toujours **pas d'écriture définitive** des données dans la base à proprement parler (dans le fichier **.mdf**).
- A **fréquence régulière**, un programme interne à SQL Server vérifie le journal de façon répétitive depuis le dernier point de lecture
- Peut **reprendre** le cas échéant après un crash système à partir d'un **Savepoint** ou **point de sauvegarde**.



Transactions - Points de sauvegarde (Savepoints)

- Les points d'enregistrement / sauvegarde (savepoints) peuvent être utilisés pour annuler une **partie particulière** de la transaction plutôt que la transaction entière.
 - Cela permet **d'annuler** la partie entre les instructions qui viennent **après le 'Savepoint' et l'instruction d'annulation Rollback.**
 - Pour définir un point de sauvegarde dans une transaction, nous utilisons la syntaxe **SAVE TRANSACTION**, puis nous ajoutons un nom au point de sauvegarde.

```
BEGIN TRANSACTION
INSERT INTO Commande VALUES (6, '15-01-2023', 5);
SAVE TRANSACTION InsertStatement
DELETE FROM Commande WHERE NumCom=6
SELECT * FROM Commande
ROLLBACK TRANSACTION InsertStatement
COMMIT TRANSACTION
SELECT * FROM Commande
```



Transactions - Notions

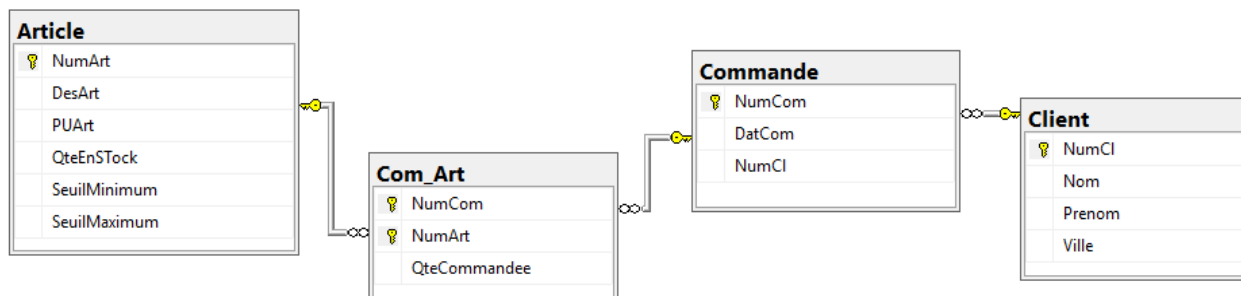
➤ Quelques Notions à retenir

- **@@TRANCOUNT** : Retourne le nombre de BEGIN TRANSACTION exécuté pendant la connexion en cours ;
- **BEGIN TRANSACTION** : Permet de **débuter** une transaction. Cette instruction **incrémente** la variable système @@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** (BD dans un état cohérent).
- Lorsque @@TRANCOUNT >0, cela veut dire qu'il y a encore des transactions **en suspend**. Il est fort possible que la table sur laquelle porte la transaction soit verrouillée.



Application : Gestion des Transactions

- 1) Écrire le programme 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 7. (Ou un client de votre choix, à insérer dans la table Commande)
 - b) On insère dans la table Com_Art la commande que l'on vient de créer pour l'article numéro 6 (Ou un article de votre choix)
 - c) Si la **quantité** dans la table Article **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 Article est suffisante) alors:
 - i. On **met à jour la table Article** par la nouvelle quantité en stock. 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 Com_Art.
 - iii. On **officialise** la transaction



**Voir Fichier :
Application Transaction**

