



Concepteur Développeur en Informatique

Assurer la persistance des données

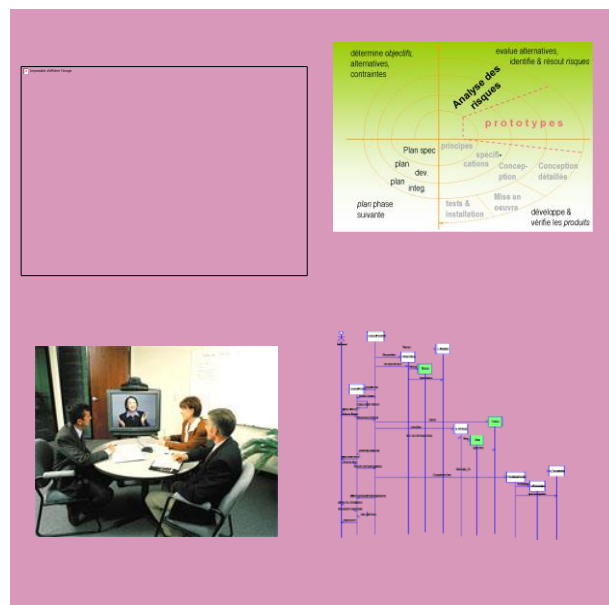
SQL – Programmer le SGBDR – Partie 2

Accueil

Apprentissage

PAE

Evaluation



Module 3 : Développer la persistance des données

Sommaire

1	Introduction	3
2	Transaction	3
2.1	Syntaxe relative aux transactions.....	4
3	Gestion des verrouillages	6
3.1	Principe du verrouillage transactionnel	6
3.2	Verrous et Niveau d'isolation des transactions.....	7
4	Le journal des transactions	12
4.1	Les modes de récupération.....	12
5	Les triggers	14
5.1	Une seule transaction	15
5.2	Programmation des triggers	15
6	Manipulation de curseurs	22
6.1	Déclaration d'un curseur	22
6.2	Ouverture et lecture	25
6.3	Un exemple à la loupe.....	26
6.4	Processus des curseurs en bref	27

1 Introduction

Ce support de cours aborde :

- La programmation des transactions et le verrouillage transactionnel,
- la programmation des triggers
- l'utilisation de curseurs de données.

Il complète les premiers éléments de connaissance apportés dans le support « Programmer le SGBDR ».

2 Transaction

SQL comporte un moteur transactionnel, une transaction correspondant à la modification d'une donnée. On parle, pour évoquer un système transactionnel, de système OLTP (On Line Transaction Processing).

Les transactions peuvent être implicites ou déclarées explicitement afin de recouvrir un **ensemble de modifications solidaires**.

La définition explicite de transactions permet de s'assurer de la complétude d'une demande de modification pouvant comporter plusieurs opérations de modification sur plusieurs lignes de plusieurs tables.

Il est ainsi possible de garantir une bonne cohérence à la base de données.

Pour expliciter ce propos, prenons l'exemple d'une transaction bancaire :

Elle est toujours composée du débit d'un compte et du crédit d'un autre compte. Il convient donc de s'assurer que les deux opérations ont bien été effectuées. Si l'une des deux opérations n'aboutit pas, nous aurons alors un système « bancal ». Si nous ne spécifions pas explicitement que ces deux opérations constituent une seule transaction, le risque d'erreur demeure.

Les données d'une transaction sont validées dans leur ensemble par l'application d'un ordre **COMMIT**, ou invalidées (elles reprennent alors leur état initial) par l'ordre **ROLLBACK**.

On appelle ce mécanisme la validation en deux temps.

La validation transactionnelle s'appuie sur le journal des transactions associé à toute base de données.

Une transaction est caractérisée les critères **ACID**

- Atomique : si une des instructions échoue, toute la transaction échoue
- Cohérente, car la base de données est dans un état cohérent avant et après la transaction, c'est-à-dire respectant les règles de structuration énoncées.
- Isolée : Les données sont verrouillées : il n'est pas possible depuis une autre transaction de visualiser les données en cours de modification dans une transaction.
- Durable : les modifications apportées à la base de données par une transaction sont validées

2.1 Syntaxe relative aux transactions

BEGIN TRAN ou BEGIN TRANSACTION marque le point de référence du début de la transaction. La transaction peut éventuellement être nommée.

Les instructions émises dans le cadre la transaction sont verrouillées par le système et l'accès aux données sous jacentes restreint pour les autres connexions.

Les données sont déverrouillées lors de l'exécution :

- d'un ordre de validation (application des modifications) COMMIT
- d'un ordre d'invalidation (retour version précédente des données) ROLL BACK ou relatif à une erreur système.

Les transactions peuvent être imbriquées.

Les transactions peuvent être nommées et être ainsi référencées plus facilement. Toutefois dans le cadre de transactions imbriquées, seule la transaction la plus externe peut être nommée.

L'utilisation de la clause WITH MARK ['description'] lors de la déclaration d'une transaction indique qu'elle est marquée dans le journal des transactions. Si WITH MARK est utilisé, un nom de transaction doit être spécifié.

WITH MARK permet de restaurer un journal de transactions par rapport à une marque nommée.

L'exemple suivant illustre l'intérêt et les conditions de mise ne place d'une transaction. Il s'agit ici de l'insertion de lignes de commandes dans la table [Order details] du comptoir anglais.

La clé primaire de la table [Order Details] est constituée de deux colonnes [OrderID] et [ProductID].

Deux lignes d'une même commande ne peuvent donc pas porter sur le même article.

2.1.1 Exemple sans transaction

Dans cet exemple, 3 lignes seront insérées et 1 rejetée.

Nous nous trouverons donc avec une commande dans le système non conforme à l'originale.

```
INSERT INTO [Order Details] ([OrderID], [ProductID], [UnitPrice], [Quantity], [Discount])  
VALUES (11099, 1, 10,10, 0.2)  
INSERT INTO [Order Details] ([OrderID], [ProductID], [UnitPrice], [Quantity], [Discount])  
VALUES (11099, 2, 10,10, 0.2)  
INSERT INTO [Order Details] ([OrderID], [ProductID], [UnitPrice], [Quantity], [Discount])  
VALUES (11099, 3, 10,10, 0.1)  
INSERT INTO [Order Details] ([OrderID], [ProductID], [UnitPrice], [Quantity], [Discount])  
VALUES (11099, 2, 10,10, 0.1)
```

Figure 1 : Insertion de lignes de commandes

```
(1 ligne(s) affectée(s))
```

```
(1 ligne(s) affectée(s))
```

```
(1 ligne(s) affectée(s))
```

Serveur : Msg 2627, Niveau 14, État 1, Ligne 1

Violation de la contrainte PRIMARY KEY 'PK_Order_Details'.

Impossible d'insérer une clé en double dans l'objet 'Order Details'.

L'instruction a été arrêtée.

	OrderID	ProductID	UnitPrice	Quantity	Discount	TS
1	11099	1	10.0000	10	0.2	0x00000000000005AB7
2	11099	2	10.0000	10	0.2	0x00000000000005AB8
3	11099	3	10.0000	10	0.1	0x00000000000005AB9

Figure 2 : Messages et résultat

2.1.2 Exemple avec transaction explicite

Dans cet exemple, la différence réside dans le fait qu'aucune ligne ne sera insérée en cas de problèmes. La cohérence des informations reste ainsi assurée.

```

P-BOSTON\SQL... - TRAN-01.sql  Résumé
USE COMPTOIRANGLAIS
GO
-- Suppression des lignes existantes
SET NOCOUNT ON
DELETE FROM DBO.[ORDER DETAILS] WHERE ORDERID = 11099
SET NOCOUNT OFF
BEGIN TRAN
BEGIN TRY
INSERT INTO [Order Details] (OrderID,ProductID,UnitPrice,Quantity,Discount)
VALUES (11099,1,10,10,0.2)
INSERT INTO [Order Details] (OrderID,ProductID,UnitPrice,Quantity,Discount)
VALUES (11099,2,10,10,0.2)
INSERT INTO [Order Details] (OrderID,ProductID,UnitPrice,Quantity,Discount)
VALUES (11099,3,10,10,0.2)
INSERT INTO [Order Details] (OrderID,ProductID,UnitPrice,Quantity,Discount)
VALUES (11099,1,10,10,0.2)
COMMIT TRAN
END TRY
BEGIN CATCH
ROLLBACK TRAN
END CATCH
SELECT * FROM dbo.[Order Details] Where OrderID = 11099
  
```

OrderID	ProductID	UnitPrice	Quantity	Discount	TS
11099	1	10.0000	10	0.2	0x00000000000005AB7
11099	2	10.0000	10	0.2	0x00000000000005AB8
11099	3	10.0000	10	0.1	0x00000000000005AB9
11099	1	10.0000	10	0.2	0x00000000000005AB7

Figure 3 : Recours aux transactions explicites

3 Gestion des verrouillages

3.1 Principe du verrouillage transactionnel

Le système recourt à la mise en place de verrous pour réduire les préjudices susceptibles d'affecter les ressources en cours de modification.

Ainsi, lorsqu'un utilisateur effectue des transactions sur une ressource, SQL Server n'autorise pas d'autres utilisateurs à effectuer sur cette ressource des opérations qui nuiraient aux dépendances de l'utilisateur détenteur du verrou. Les verrous sont gérés de manière interne par le logiciel système et sont placés et libérés en fonction des actions entreprises par l'utilisateur.

Il s'agit là d'un système complexe que nous n'aborderons pas en détail mais dont il faut comprendre les enjeux et les conséquences éventuelles sur le niveau de performance de votre système.

Les verrous peuvent être appliqués à différents niveaux de la base de données depuis la ligne (niveau le plus fin) jusqu'à la table voire la base de données dans son ensemble (par exemple lors d'une opération de restauration, un verrou exclusif est mis en place sur la base de données).

Vous pouvez modifier la nature des verrous et leur impact, en modifiant le niveau d'isolation des transactions. Mais avant d'aborder ce sujet, nous allons, par un schéma, visualiser une situation qui illustre les situations de blocages engendrés par ces mécanismes de verrouillage et les demandes d'accès concurrentielles à une même ressource.

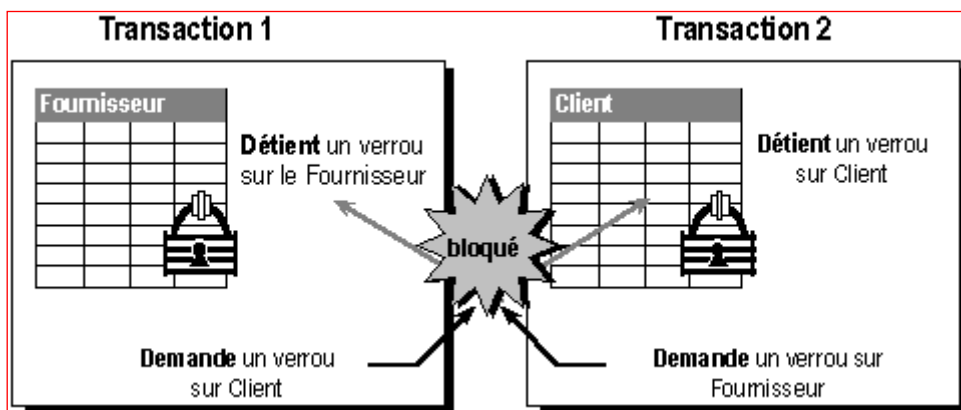


Figure 4 : Exemple de blocage sur accès concurrents à une même ressource

La transaction 1 a posé un verrou sur le fournisseur et demande un verrou sur le client sur laquelle la transaction 2 est déjà détentrice d'un verrou et qui demande à son tour un verrou sur le fournisseur. Il s'agit là d'une situation où les deux demandes se bloquent : on parle d'étreinte fatale...Le système peut mettre fin à cette situation de blocage en annulant la requête du thread victime du verrou (déterminée par des règles complexes visant à tuer le « plus faible »...): on parle alors de DEADLOCK.

3.2 Verrous et Niveau d'isolation des transactions

Il existe plusieurs modes de verrouillage : partagé, mise à jour et exclusif. Le mode de verrouillage indique le niveau de dépendance de la connexion sur l'objet verrouillé.

SQL Server contrôle l'interaction des modes de verrouillage. Par exemple, il est impossible d'obtenir un verrou exclusif si d'autres connexions disposent de verrous partagés sur la ressource.

Les verrous sont maintenus le temps nécessaire pour protéger la ressource au niveau demandé mais la durée des verrous partagés utilisés pour protéger les lectures dépend des niveaux d'isolement de la transaction.

Les verrous exclusifs utilisés pour protéger les mises à jour sont maintenus jusqu'à la fin de la transaction.

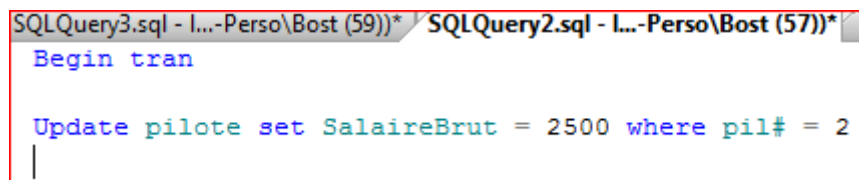
Si une connexion essaye d'appliquer un verrou qui entre en conflit avec un verrou placé par une autre connexion, elle est bloquée jusqu'à ce que le verrou en conflit soit libéré et que la connexion applique le verrou demandé ou si le délai d'attente de la connexion expire.

Par défaut, il n'y a pas de délai d'attente, mais certaines applications en définissent un pour éviter d'attendre indéfiniment.

L'attribution des verrous se fait selon la règle du premier arrivé premier servi.

3.2.1 Un exemple pour comprendre

La première transaction demande la mise à jour du salaire d'un pilote.



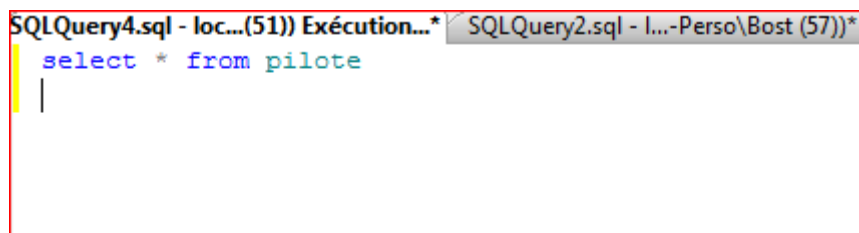
```
SQLQuery3.sql - I...-Perso\Bost (59))* SQLQuery2.sql - I...-Perso\Bost (57))*  
Begin tran  
  
Update pilote set SalaireBrut = 2500 where pil# = 2  
|
```

Figure 5 : Première transaction : verrou exclusif / MAJ

Une deuxième transaction demande la liste des pilotes.

Cette demande reste sans effet...

Elle ne sera exécutée que lors de la libération du verrou posé sur la table pilote par la première transaction.



```
SQLQuery4.sql - loc...(51)) Exécution...* SQLQuery2.sql - I...-Perso\Bost (57))*  
select * from pilote  
|
```

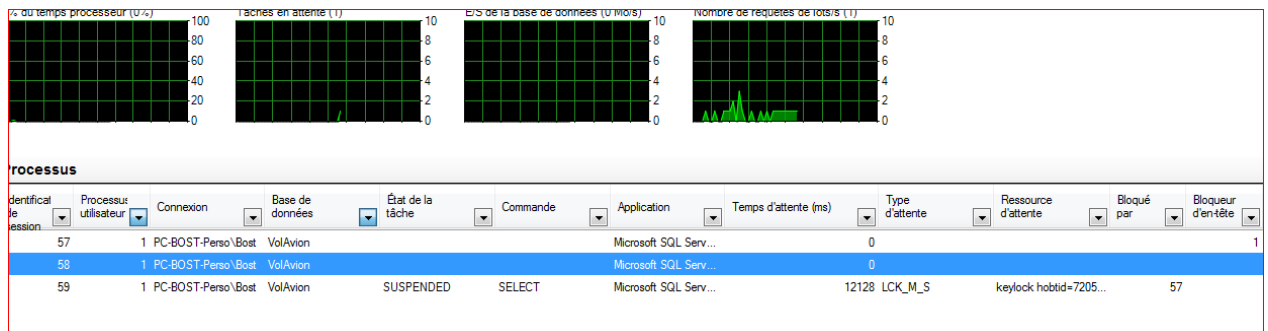
Figure 6 : Demande extraction bloquée par Transaction 1

Programmation des SGBDR-Partie 2

Par l'intermédiaire de l'onglet Moniteur d'activité du logiciel Enterprise Manager, présent dans la barre des tâches standard :

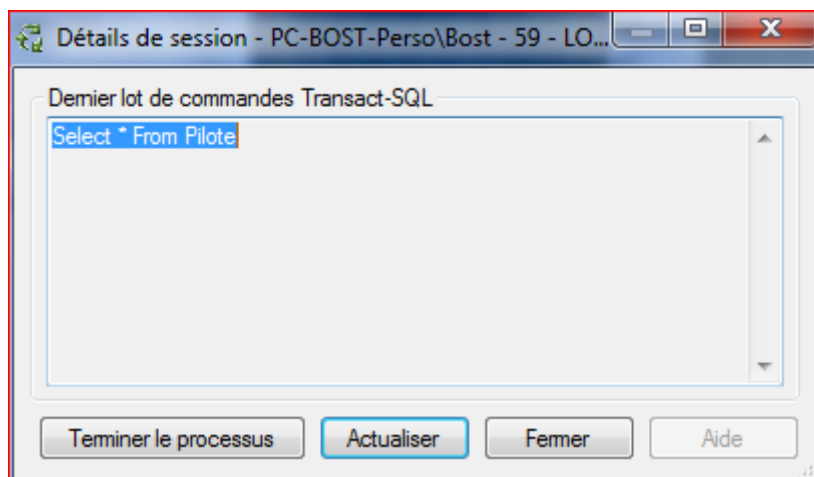


Nous pouvons comprendre la nature des opérations en cours et des verrous posés par le processus exécutant la requête de mise à jour.



État de la tâche	Commande	Application	Temps d'attente (ms)	Type d'attente
		Microsoft SQL Serv...	0	
		Microsoft SQL Serv...	0	
SUSPENDED	SELECT	Microsoft SQL Serv...	142230	LCK_M_S

Nous pouvons aussi avoir le détail des commandes exécutées.

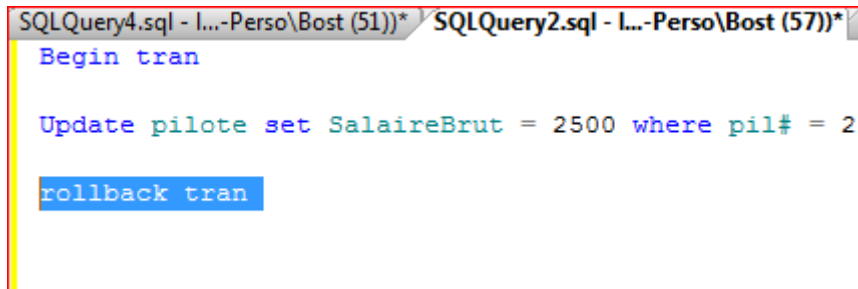


Programmation des SGBDR-Partie 2

Que faut-il faire pour débloquent la situation et permettre à la requête d'extraction de s'exécuter ?

Mais c'est bien sûr ! Terminer la transaction 1 par un COMMIT !

Après exécution du COMMIT TRAN, l'extraction est réalisée :

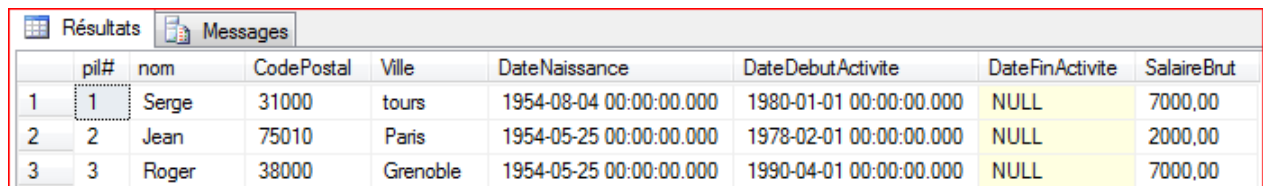


```
SQLQuery4.sql - I...-Perso\Bost (51))* SQLQuery2.sql - I...-Perso\Bost (57))*
Begin tran

Update pilote set SalaireBrut = 2500 where pil# = 2

rollback tran
```

Figure 7 : Fin de Transaction 1 – abandon par Rollback



	pil#	nom	CodePostal	Ville	DateNaissance	DateDebutActivite	DateFinActivite	SalaireBrut
1	1	Serge	31000	tours	1954-08-04 00:00:00.000	1980-01-01 00:00:00.000	NULL	7000,00
2	2	Jean	75010	Paris	1954-05-25 00:00:00.000	1978-02-01 00:00:00.000	NULL	2000,00
3	3	Roger	38000	Grenoble	1954-05-25 00:00:00.000	1990-04-01 00:00:00.000	NULL	7000,00

Figure 8 : La deuxième transaction est libérée et la liste réalisée

A noter : il est donc essentiel de ne pas maintenir des transactions trop longues ou qui exigent une intervention de l'utilisateur, comme c'est le cas ici ! Il s'agit évidemment d'un cas d'école...

1.1.1. Les verrous en bref

Les verrous empêchent les conflits de mise à jour :

- Ils permettent la sérialisation des transactions de façon à ce qu'une seule personne à la fois puisse modifier un élément de données ; dans un système d réservation de places, les verrous garantissent que chaque place n'est attribuée qu'à une seule personne.
- SQL Server définit et ajuste de manière dynamique le niveau de verrouillage approprié pendant une transaction : il est également possible de contrôler manuellement le mode d'utilisation de certains verrous.
- Les verrous sont nécessaires aux transactions concurrentes pour permettre aux utilisateurs d'accéder et de mettre à jour les données en même temps.

Le contrôle de la concurrence permet de s'assurer que les modifications apportées par un utilisateur ne vont pas à l'encontre de celles apportées par un autre.

- – Le contrôle pessimiste verrouille les données qui sont lues en vue d'une mise à jour, empêchant tout autre utilisateur de modifier ces données
- – Le contrôle optimiste ne verrouille pas les données : si les données ont été modifiées depuis la lecture, un message d'annulation est envoyé à l'utilisateur.

Le choix entre ces deux types de contrôle est effectué en fonction de l'importance du risque de conflit de données et de l'évaluation du coût de verrouillage des données (nombre d'utilisateurs, nombre de transactions, temps de réponse ...)

Le type de contrôle sera explicité en spécifiant le niveau d'isolement de la transaction.

3.2.2 Niveau d'isolation des transactions

Lors de transactions concurrentes, des verrous peuvent être posés pour éviter les situations suivantes :

- – .Mise à jour perdue : une mise à jour peut être perdue lorsqu'une transaction écrase les modifications effectuées par une autre transaction
- – Lecture incorrecte : Se produit lorsqu'une transaction lit des données non validées provenant d'une autre transaction
- – Lecture non renouvelable : Se produit lorsque, une transaction devant lire la même ligne plusieurs fois, la ligne est modifiée par une autre transaction et donc produit des valeurs différentes à chaque lecture
- – Lectures fantômes : Se produit lorsqu'une autre transaction insert une ligne au sein de la transaction en cours

Un verrou partagé appliqué à une ressource par une première transaction autorise l'acquisition par une deuxième transaction d'un verrou partagé sur cette ressource, même si la première transaction n'est pas terminée. Un verrou partagé sur une ligne est libéré dès la lecture de la ligne suivante.

SQL Server utilise des verrous exclusifs pour les modifications de données (INSERT, UPDTAT, DELETE). Une seule transaction peut acquérir un verrou exclusif sur une ressource ; une transaction ne peut pas acquérir un verrou exclusif sur une ressource tant qu'il existe des verrous partagés ; une transaction ne peut pas acquérir un verrou partagé sur une ressource déjà pourvue d'un verrou exclusif.

SQL Server permet de définir un niveau d'isolement qui protège une transaction vis-à-vis des autres.

La syntaxe est la suivante

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED  
| REPEATABLE READ | SNAPSHOT | SERIALIZABLE}
```

Read UnCommitted

Indique à SQL Server de ne pas placer de verrous partagés : une transaction peut lire des données modifiées non encore validées par une autre transaction.

Des lectures incorrectes peuvent se produire.

Read Committed (Option par défaut)

Indique à SQL Server d'utiliser des verrous partagés pendant la lecture : une transaction ne peut pas lire les données modifiées mais non validées d'une autre transaction.

La lecture incorrecte ne peut pas se produire.

Repeatable Read

SQL Server place des verrous partagés sur toutes les données lues par chaque instruction de la transaction et les maintient jusqu'à la fin de la transaction : une transaction ne peut pas lire des données modifiées mais pas encore validées par une autre transaction, et ne peut modifier les données lues par la transaction active tant que celle-ci n'est pas terminée.

La lecture incorrecte et la lecture non renouvelable ne peuvent pas se produire.

Snapshot

Une transaction n'a pas accès aux modifications de données apportées par une autre transaction : les données vues par la première transaction sont les données antérieures au début de la deuxième transaction.

Serializable

Une transaction ne peut pas lire des données modifiées mais pas encore validées par une autre transaction, et ne peut modifier les données lues par la transaction active tant que celle-ci n'est pas terminée.

Une transaction ne peut pas insérer de nouvelles lignes avec des valeurs de clés comprises dans le groupe de clés lues par des instructions de la transaction active, tant que celle-ci n'est pas terminée.

La lecture incorrecte la lecture non renouvelable et les lectures fantômes ne peuvent pas se produire.

En conclusion :

Le niveau d'isolement spécifie le comportement de verrouillage par défaut de toutes les instructions de la session (connexion).

Plus le niveau d'isolement est élevé, plus les verrous sont maintenus longtemps et plus ils sont restrictifs.

La commande DBCC USEROPTIONS renvoie, entre autres informations, le niveau d'isolement de la session.

Il est possible de définir la durée maximale pendant laquelle SQL Server permet à une transaction d'attendre le déverrouillage d'une ressource bloquée.

La variable globale @@lock_timeout donne le temps d'attente défini.

SET LOCK_TIMEOUT délai_attente positionne le délai, en millisecondes, avant que ne soit renvoyé un message d'erreur (-1 indique qu'il n'y a pas de délai – par défaut)

4 Le journal des transactions

Le journal des transactions enregistre les modifications de données au fur et à mesure qu'elles sont effectuées.

Lorsqu'une modification de données est envoyée par l'application, si les pages de données concernées n'ont pas été chargées en mémoire cache à partir du disque lors d'une précédente modification, elles sont alors amenées en mémoire.

Chaque instruction est enregistrée dans le journal et la modification est effectuée en mémoire.

Le processus de point de contrôle reporte périodiquement dans la base de données toutes les modifications effectuées

A un instant t , il existe dans le journal :

1. des transactions validées avant le dernier point de contrôle et donc reportées dans la base
2. des transactions validées après le dernier point de contrôle, et donc non reportées dans la base
3. des transactions non validées

Les conditions 1 et 2 n'existent que si vous avez choisi un mode de récupération de données autre que simple.

Dans ce cas, en cas de défaillance du système, le processus de restauration automatique utilise le journal des transactions pour reprendre toutes les transactions validées non reportées dans la base, et annuler les transactions non validées.

4.1 Les modes de récupération

4.1.1 Mode de récupération simple

Le mode de récupération simple réduit les tâches d'administration du journal des transactions car celui-ci n'est pas sauvegardé. Il entraîne un risque de perte de travail assez élevé en cas d'endommagement de la base de données. Les données ne sont récupérables que jusqu'à la sauvegarde la plus récente des données perdues. Par conséquent, en mode de récupération simple, les intervalles de sauvegarde doivent être suffisamment courts pour empêcher la perte d'un volume significatif de données. Toutefois, les intervalles doivent être suffisamment longs pour que la charge de sauvegarde n'affecte pas la production. L'insertion de sauvegardes différentielles dans la stratégie de sauvegarde permet de réduire la surcharge.

En règle générale, pour une base de données utilisateur, le mode de récupération simple est utile pour les bases de données de test et de développement ou pour celles contenant essentiellement des données accessibles en lecture seule, telles qu'un Data Warehouse.

Le mode de récupération simple ne convient pas aux systèmes de production où la perte de récentes modifications est inacceptable. Dans ce cas, nous recommandons l'utilisation du mode de restauration complète.

4.1.2 Mode de restauration complète et de récupération utilisant les journaux de transactions

Le mode de restauration complète et le mode de récupération utilisant les journaux de transactions offrent beaucoup plus de protection pour les données que le mode de récupération simple. Ces deux modes dépendent de la sauvegarde du journal des transactions pour permettre une récupération complète et empêcher une perte de travail dans le plus grand nombre possible de scénarios d'échec.

Mode de restauration complète

Fournit le mode de maintenance de base de données normal pour les bases de données où la durabilité des transactions est nécessaire.

Des sauvegardes du journal sont requises. Ce mode consigne complètement toutes les transactions et conserve les enregistrements du journal des transactions après leur sauvegarde. Le mode de restauration complète permet de récupérer une base de données jusqu'au moment de la défaillance, en partant du principe que la fin du journal peut être sauvegardée après la défaillance. Le mode de restauration complète prend également en charge la restauration des pages de données individuelles.

Mode de récupération utilisant les journaux de transactions

Ce mode de récupération consigne en bloc la plupart des opérations en bloc. Il a été conçu uniquement comme complément au mode de restauration complète.

Pour certaines opérations en bloc à grande échelle, telles qu'une importation en bloc ou une création d'index, le basculement temporaire en mode de récupération utilisant les journaux de transactions augmente les performances et réduit la consommation d'espace du journal.

Des sauvegardes du journal sont encore requises. Comme le mode de restauration complète, le mode de récupération utilisant les journaux de transactions conserve les enregistrements du journal des transactions après leur sauvegarde.

En revanche, il génère des sauvegardes du journal plus volumineuses et un risque accentué de perte de travail, car le mode de récupération utilisant les journaux de transactions ne prend pas en charge la récupération dans le temps. Pour plus d'informations, consultez Sauvegarde avec le mode de récupération utilisant les journaux de transactions.

Il s'agit de programmes déclenchés automatiquement lors d'opérations de mises à jour sur une table.

5 Les triggers

Ils représentent une catégorie spécialisée de procédures stockées définies pour s'exécuter automatiquement lors d'une transaction sur une table ou une vue.

Ils seront donc associés à une opération d'insertion, de mise à jour ou de suppression d'une ligne d'une table particulière.

Ce sont des outils puissants qui peuvent être utilisés pour mettre automatiquement en application des règles de gestion lorsque des données sont modifiées. Ils sont notamment utilisés pour programmer des transactions dépendantes d'autres transactions.

Si nous prenons l'exemple de la prise de commande dans une application de gestion commerciale, dans le cas de l'insertion d'une nouvelle ligne de commande, nous pourrions mettre à jour les quantités de produits réservés pour commande et générer une ligne dans la table mouvements de stocks.

Le stock de réservations dépend des transactions sur la table commande, en autres.

Les déclencheurs peuvent aussi étendre la logique de contrôle d'intégrité des données et permettent de dépasser les limites fixées aux contraintes d'intégrité, valeurs par défaut et règles programmées par ailleurs. Nous reviendrons sur ces notions dans le chapitre consacré à la création des structures de données.

Il est possible de programmer un Trigger spécifique pour une opération ou pour un ensemble d'opérations. Il est toutefois plus aisé de créer un trigger pour chacune des opérations de base sur une table.

Il existe deux catégories de Triggers :

Les triggers classiques qui se déclenchent après l'opération associée. Ils sont désignés sous le terme de triggers AFTER

Les triggers qui se substituent à l'opération associée, dits INSTEAD OF.

Dans certains systèmes et depuis la norme SQL 3 la tendance est à la généralisation, vous trouverez des déclencheurs BEFORE, comprenez avant insertion dans le journal des transactions.

Restrictions interdites au sein d'un programme déclencheur :

ALTER DATABASE	CREATE DATABASE	DISK INIT
DISK RESIZE	DROP DATABASE	LOAD DATABASE
LOAD LOG	RECONFIGURE	RESTORE DATABASE
RESTORE LOG		

5.1 Une seule transaction

Il est fondamental de comprendre que les triggers qui se déclenchent sur une opération et cette opération font partie d'une seule et même transaction, ainsi que tous les autres triggers éventuellement déclenchés par la suite.

Ainsi, si un trigger échoue, l'opération à l'origine de son exécution sera elle aussi annulée.

Les règles statiques définies sur les données (contraintes d'intégrité, valeurs par défaut, ...) sont toujours vérifiées avant l'exécution d'un trigger.

Il n'est donc pas possible de réaliser des suppressions en cascade par le biais d'un trigger.

Par exemple, sur suppression d'une ligne dans la table des commandes (orders), je ne peux pas supprimer les lignes détail (order details) par le biais d'un Trigger. Le système émettra un message d'erreur précisant la violation d'une contrainte d'intégrité référentielle.

Il faut éviter la récursivité directe dans un trigger. Elle n'est d'ailleurs pas autorisée par défaut. Exemple de récursivité directe.

Un Trigger T1 se déclenche sur l'ajout d'une ligne dans une table Tab1 et comporte une instruction d'ajout d'une ligne dans cette même table.

5.1.1 Les triggers « AFTER »

Vous pouvez associer plusieurs triggers de cette nature pour une même opération sur une même table.

Il n'est toutefois pas possible de préciser l'ordre d'exécution des différents triggers. Il est donc nécessaire d'être prudent sur la nature des opérations à faire prendre en charge par les différents triggers.

5.1.2 Les triggers « INSTEAD OF »

Essentiellement conçus pour prendre en charge les opérations de modifications de données sur les vues multi-tables qui, par nature, sont en lecture seule.

Seul un programme déclencheur de cette nature par opération.

Ce type de trigger est toutefois peu usité car non conforme aux standards.

5.2 Programmation des triggers

5.2.1 Les tables temporaires INSERTED et DELETED

Ces tables sont manipulées au sein des triggers et stockent les lignes en cours de modification.

Ces tables ont des structures (définition) identiques à la table associée au Trigger.

La table INSERTED comprend les lignes en cours d'ajout (INSERT) ou les nouvelles valeurs des lignes modifiées (UPDATE).

La table DELETED comprend les lignes supprimées (DELETE) ou les valeurs des lignes avant modification (UPDATE).

Une transaction UPDATE est en fait assimilée à une opération de suppression suivie d'une opération d'insertion ; les anciennes lignes figurent dans la table **deleted** et les nouvelles lignes dans la table **inserted**.

Ces tables sont uniquement accessibles en lecture. Elles permettent d'extraire les valeurs des lignes sur lesquelles porte la transaction.

5.2.2 Exemple de trigger

J'envisage de tracer les opérations de modification de salaire des pilotes et de consigner la trace de ces dernières dans une table nommée Augmentations lorsque l'augmentation est supérieure à 10%.

Je crée donc un programme déclencheur sur la table PILOTES pour l'opération UPDATE (les autres opérations ne m'intéressent pas).

Je souhaite conserver la trace :

- du salaire d'origine et celui nouvellement alloué
- l'identifiant et le nom du pilote
- le compte utilisateur ayant réalisé la transaction
- la date de la transaction

Ce trigger va systématiquement se déclencher sur toute opération de mise à jour de la table PILOTES.

En fait, je ne souhaite effectuer une mise en historique que lorsque la colonne Salaire a fait l'objet d'une modification. Je peux recourir à la fonction UPDATE qui permet de déterminer si une colonne est mise à jour.

Création de la table historique des augmentations

Je récupère les valeurs du compte utilisateur et la date de transaction en recourant à des fonctions systèmes exécutées si aucune valeur n'est insérée dans ces colonnes (valeurs par défaut).

```
CREATE TABLE [dbo].[Augmentations]
(
    [Pil#] [smallint] NOT NULL ,
    [Nom] [varchar] (75) NULL ,
    [SalaireInitial] [money] NULL ,
    [SalaireAlloué] [money] NULL ,
    [Utilisateur] [varchar] (255) NULL ,
    [DateHeureModification] [datetime] NULL
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Augmentations] ADD
    CONSTRAINT [DF_Augmentations_Utilisateur] DEFAULT (user_name()) FOR [Utilisateur],
    CONSTRAINT [DF_Augmentations_DateHeureModification] DEFAULT (getdate()) FOR [DateHeureModification]
GO
```

Figure 9 : Schéma relationnel de la table historique des augmentations

Programmation du Trigger historique des augmentations

```

CREATE TRIGGER TG_Pilote_Update_1 ON PILOTE
/* Sur mise à jour de la table PILOTE */
FOR UPDATE
AS
DECLARE @SalaireInitial Money
DECLARE @SalaireAlloue Money
DECLARE @PIL# Smallint
DECLARE @Nom varchar(75)

/* UPDATE permet de savoir si une colonne a été modifiée
dans le cadre d'une insertion ou d'un modification
Non significatif dans le cas d'une suppression */

IF UPDATE(SalaireBrut) AND (SELECT SalaireBrut FROM DELETED) > 0
BEGIN
/* récupération des valeurs du salaire + autres infos */
SELECT @SalaireInitial = SalaireBrut FROM DELETED
SELECT @SalaireAlloue = SalaireBrut, @PIL#=PIL#, @Nom=Nom FROM INSERTED

/* Augmentation supérieure à 10 % */
IF (@SalaireAlloue - @SalaireInitial) / @SalaireInitial > 0.10
BEGIN
/* Insertion dans table historique */
INSERT INTO Augmentations
(Pil#, Nom, SalaireInitial, SalaireAlloué, Utilisateur, DateHeureModification)
VALUES
(@PIL#, @Nom, @SalaireInitial, @SalaireAlloue, DEFAULT, DEFAULT)
END
END

```

Figure 10 : Trigger sur Pilote pour Mise à Jour

Les tables peuvent comporter plusieurs déclencheurs mais un déclencheur ne porte que sur une seule table ou vue.

L'instruction CREATE TRIGGER peut être définie avec les clauses FOR UPDATE, FOR INSERT ou FOR DELETE pour affecter un déclencheur à une catégorie spécifique d'actions de modification des données ou pour un ensemble d'opérations FOR INSERT, UPDATE par exemple.

Test du trigger historique des augmentations

Le premier test concerne la mise à jour d'une seule ligne identifiée par la clé primaire :

```

update pilote set salairebrut=20000 where pil#=1

select * from augmentations

```

Pil#	Nom	SalaireInitial	SalaireAlloué	Utilisateur	DateHeureModification
1	SERGE	12000.0000	20000.0000	dbo	2005-11-30 17:33:12.957

Figure 11 : Test du trigger Maj ligne unique

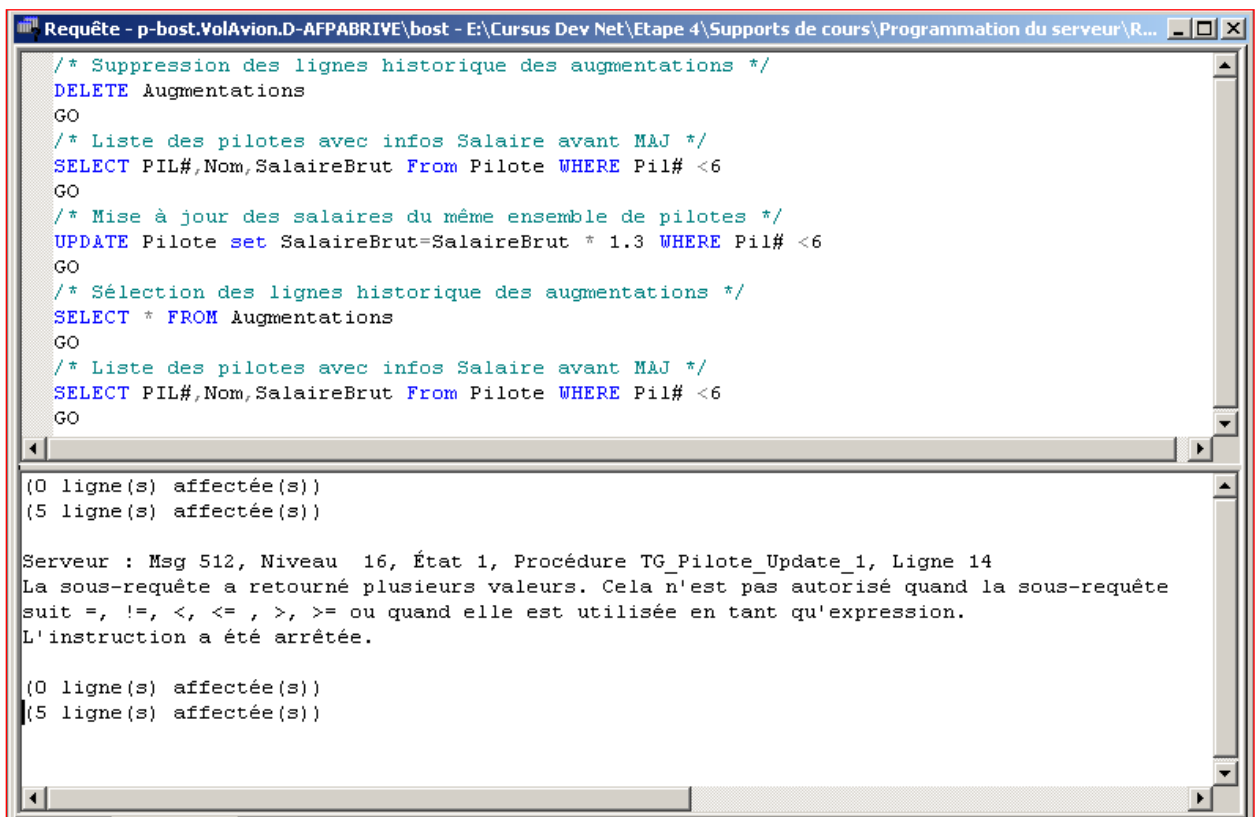
Une ligne est bien inscrite dans la table des augmentations.

A ce stade des tests, le trigger semble fonctionner correctement.

Mais que se passerait-il si je demandais la mise à jour d'un ensemble de lignes par le biais d'une seule transaction ?

Programmation des SGBDR-Partie 2

Le résultat par l'exemple ...



```
Requête - p-bost.VolAvion.D-AFPABRIVE\bost - E:\Cursus Dev Net\Etape 4\Supports de cours\Programmation du serveur\R...
/* Suppression des lignes historique des augmentations */
DELETE Augmentations
GO
/* Liste des pilotes avec infos Salaire avant MAJ */
SELECT Pil#,Nom,SalaireBrut From Pilote WHERE Pil# <6
GO
/* Mise à jour des salaires du même ensemble de pilotes */
UPDATE Pilote set SalaireBrut=SalaireBrut * 1.3 WHERE Pil# <6
GO
/* Sélection des lignes historique des augmentations */
SELECT * FROM Augmentations
GO
/* Liste des pilotes avec infos Salaire avant MAJ */
SELECT Pil#,Nom,SalaireBrut From Pilote WHERE Pil# <6
GO

(0 ligne(s) affectée(s))
(5 ligne(s) affectée(s))

Serveur : Msg 512, Niveau 16, État 1, Procédure TG_Pilote_Update_1, Ligne 14
La sous-requête a retourné plusieurs valeurs. Cela n'est pas autorisé quand la sous-requête
suit =, !=, <, <=, >, >= ou quand elle est utilisée en tant qu'expression.
L'instruction a été arrêtée.

(0 ligne(s) affectée(s))
(5 ligne(s) affectée(s))
```

Figure 12 : Augmentation de salaires : plusieurs lignes mises à jour au sein d'une même transaction

Le programme déclencheur s'est arrêté avec une erreur mettant fin à la transaction. La ligne en erreur dans ce trigger est l'expression conditionnelle qui contient une sous requête SQL :

```
(SELECT SalaireBrut FROM DELETED) > 0
```

En fait, elle ne renvoie pas une valeur scalaire, mais un ensemble de valeurs et la condition ne peut alors pas être évaluée correctement.

Ceci pour confirmer un point essentiel : Les tables temporaires SELECTED et DELETED contiennent l'ensemble des lignes sur lesquelles porte la transaction.

Programmation des SGBDR-Partie 2

Deuxième point : qu'en est-il de la mise à jour des lignes et de la table historique des augmentations ?

Réponse par l'illustration suivante :

	PIL#	Nom	SalaireBrut			
1	1	SERGE	20000.0000			
2	2	JEAN	12500.0000			
3	3	ROGER	25000.0000			
4	4	ROBERT	12000.0000			
5	5	MICHEL	12000.0000			
	Pil#	Nom	SalaireInitial	SalaireAlloué	Utilisateur	DateHeureModification
	PIL#	Nom	SalaireBrut			
1	1	SERGE	20000.0000			
2	2	JEAN	12500.0000			
3	3	ROGER	25000.0000			
4	4	ROBERT	12000.0000			
5	5	MICHEL	12000.0000			

Figure 13 : Résultat de la demande de mise à jour précédente

A noter : Aucune ligne n'a été mise à jour et aucun enregistrement n'a été inséré dans la table historique des augmentations. Cela confirme que la demande de mise à jour et les opérations effectuées par le trigger ne constituent qu'une seule et même transaction. Une erreur survenue dans le trigger provoque un roll back de la transaction.

Correction du trigger augmentation de salaire

Que faut-il faire pour que le trigger fonctionne correctement ?

Il convient de traiter chaque ligne des tables temporaires individuellement et, pour ce faire, recourir à la mise en place d'un curseur.

Vous trouverez dans le chapitre suivant des notions de cours sur ces curseurs.

Pour compléter notre mise au point et vérifier l'exactitude du trigger, j'ai de plus inséré une fonction d'impression des données traitées.

Vous trouverez page suivante le nouveau trigger correctement codé.

Programmation des SGBDR-Partie 2

```
CREATE TRIGGER TG_Pilote_Update_1 ON PILOTE
/* Sur mise à jour de la table PILOTE */
FOR UPDATE
AS
DECLARE @SalaireInitial Money
DECLARE @SalaireAlloue Money
DECLARE @PIL# Smallint
DECLARE @Nom varchar(75)

/* UPDATE permet de savoir si une colonne a été modifiée
dans le cadre d'une insertion ou d'une modification
Non significatif dans le cas d'une suppression */

IF UPDATE(SalaireBrut)
BEGIN
    Print 'Le salaire est à mettre à jour '
    DECLARE LignesImageAvant CURSOR
    LOCAL FORWARD_ONLY READ_ONLY
    FOR SELECT SalaireBrut FROM DELETED

    DECLARE LignesImageAprès CURSOR
    LOCAL FORWARD_ONLY READ_ONLY
    FOR SELECT PIL#,Nom,SalaireBrut FROM INSERTED

    OPEN LignesImageAvant
    OPEN LignesImageAprès

    FETCH NEXT FROM LignesImageAvant INTO @SalaireInitial
    WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH NEXT FROM LignesImageAprès INTO @pil#,@Nom,@SalaireAlloue
        IF (@SalaireInitial > 0)
        BEGIN
            IF (@SalaireAlloue - @SalaireInitial) / @SalaireInitial > 0.10
            INSERT INTO Augmentations
            (Pil#,Nom,SalaireInitial,SalaireAlloué,Utilisateur,DateHeureModification)
            VALUES
            (@PIL#,@Nom,@SalaireInitial,@SalaireAlloue,DEFAULT, DEFAULT)
        END
        FETCH NEXT FROM LignesImageAvant INTO @SalaireInitial
    END
    CLOSE LignesImageAvant
    CLOSE LignesImageAprès
    DEALLOCATE LignesImageAvant
    DEALLOCATE LignesImageAprès
END
```

Figure 14 : Version corrigée du trigger

```
/* Suppression des lignes historique des augmentations */
DELETE Augmentations
GO
/* Liste des pilotes avec infos Salaire avant MAJ */
SELECT PIL#,Nom,SalaireBrut From Pilote WHERE Pil# <6
GO
/* Mise à jour des salaires du même ensemble de pilotes */
UPDATE Pilote set SalaireBrut=(SalaireBrut + 2000.0000) WHERE Pil# <6
GO
/* Sélection des lignes historique des augmentations */
SELECT * FROM Augmentations
GO
/* Liste des pilotes avec infos Salaire avant MAJ */
SELECT PIL#,Nom,SalaireBrut From Pilote WHERE Pil# <6
GO
```

Figure 15 : Test réalisé avec augmentation valeur absolue de 2000

	PIL#	Nom	SalaireBrut			
1	1	SERGE	27360.0000			
2	2	JEAN	18360.0000			
3	3	ROGER	33360.0000			
4	4	ROBERT	17760.0000			
5	5	MICHEL	17760.0000			
	Pil#	Nom	SalaireInitial	SalaireAlloué	Utilisateur	DateHeureModification
1	4	ROBERT	17760.0000	19560.0000	dbo	2005-12-01 12:15:50.500
2	5	MICHEL	17760.0000	19560.0000	dbo	2005-12-01 12:15:50.500
	PIL#	Nom	SalaireBrut			
1	1	SERGE	29160.0000			
2	2	JEAN	20160.0000			
3	3	ROGER	35160.0000			
4	4	ROBERT	19560.0000			
5	5	MICHEL	19560.0000			

Figure 16 : Résultats de l'exécution des lots précédents

Deux lignes ont été insérées dans la table de l'historique des augmentations de salaire hors normes et les salaires ont été correctement impactés par la mise à jour. Le trigger fonctionne !

Et si je réalise une mise à jour du patronyme des pilotes ?

```

/* Mise à jour du nom du même ensemble de pilotes */
UPDATE Pilote set Nom=LOWER(Nom) WHERE Pil# <6
GO
|
(5 ligne(s) affectée(s))

```

Figure 17 : Mise à jour du patronyme

Nous constatons que seule la table des pilotes a été mise à jour et nous n'avons pas d'informations mentionnant la mise à jour de la colonne salaire !

La fonction UPDATE() qui permet de savoir si une colonne (ici celle stockant le salaire brut) a fait l'objet d'une mise à jour a donc correctement été mise en œuvre.

6 Manipulation de curseurs

Ce chapitre traite de la manipulation de données au travers des curseurs. Les curseurs permettent de parcourir un jeu de données issu d'une requête de sélection et de traiter individuellement chaque ligne.

En règle générale, nous traitons les lignes par lot. Toutes les lignes du lot font l'objet d'un traitement similaire.

Mais nous avons besoin, parfois, de différencier les opérations à exécuter en fonction de la valeur des colonnes d'une ligne.

Les curseurs sont une extension logique des jeux de résultats qui permettent aux applications de manipuler le jeu de résultats ligne par ligne. Nous retrouverons les mêmes principes mis en œuvre dans le cas de la manipulation de données en mode connecté travers de composants tiers tels que ADO (dans la version précédent ADO Net mode déconnecté).

Nous allons découvrir dans ce chapitre les différentes instructions relatives aux curseurs :

- La déclaration du curseur
- L'ouverture et la fermeture
- Le parcours du curseur
- La destruction du curseur

Ce support n'a pas pour objectif d'approfondir les techniques de manipulation de curseurs mais d'apporter la connaissance strictement nécessaire.

6.1 Déclaration d'un curseur

Un curseur est une variable d'un type particulier CURSOR. Il se déclare donc à l'aide du mot clé DECLARE.

6.1.1 Portée

Il est possible de définir la portée du curseur.

Par défaut, la portée est locale à la procédure stockée ou au lot d'instructions dans lequel celui-ci est utilisé.

Mais la durée de vie d'un curseur peut être étendue à la durée de la connexion SQL qui l'a créée. Dans ce cas, il sera considéré comme de portée globale. Les mots-clés LOCAL, GLOBAL permettent de définir la portée du curseur.

Dans le cas d'un curseur de portée globale, la mémoire occupée par ce curseur sera libérée implicitement lors de la fermeture de la connexion SQL.

6.1.2 Défilement

Mots clés FORWARD_ONLY et SCROLL

Par défaut un curseur est en défilement vers l'avant uniquement. C'est le cas le plus fréquent d'utilisation : on lit le curseur séquentiellement depuis la première ligne jusqu'à la dernière au moyen de l'opération FETCH NEXT.

Mais il est possible d'étendre les fonctionnalités de parcours de ce dernier en complétant la définition du curseur à l'aide du mot clé **SCROLL**.

Il sera alors possible d'utiliser de multiples opérations de déplacement, de manière relative ou absolue, par le biais des instructions **FIRST**, **LAST**, **PRIOR**, **NEXT**, **RELATIVE** et **ABSOLUTE**.

6.1.3 Type du curseur

Les différents types de curseurs impactent les performances et le reflet des modifications apportées aux lignes présentes dans le curseur. Par défaut, le curseur est dynamique et reflète donc toutes les modifications apportées en cours de traitement sur les lignes présentes dans le curseur.

STATIC

Définit un curseur qui fait une copie temporaire des données qu'il doit utiliser. Toutes les réponses aux requêtes destinées au curseur sont effectuées à partir de cette table temporaire dans tempdb; par conséquent, les modifications apportées aux tables de base ne sont pas reflétées dans les données renvoyées par les extractions de ce curseur, et ce dernier n'accepte pas de modifications.

KEYSET

Spécifie que l'appartenance au curseur et l'ordre des lignes sont fixés lors de l'ouverture du curseur.

L'ensemble des clés qui identifient de manière unique les lignes est créé dans une table de tempdb, connue sous le nom de keyset.

Les modifications apportées aux valeurs non-clés dans les tables de la base, que ce soit celles effectuées par le propriétaire du curseur ou celles validées par les autres utilisateurs, sont visibles lorsque le propriétaire parcourt le curseur.

Les insertions effectuées par d'autres utilisateurs ne sont pas visibles. Si vous supprimez une ligne, une tentative d'extraction de la ligne renvoie la valeur -2 pour @@FETCH_STATUS.

Les mises à jour de valeurs clés effectuées hors du curseur sont semblables à la suppression de l'ancienne ligne suivie de l'insertion d'une nouvelle. La ligne comprenant les nouvelles valeurs n'est pas visible et si vous tentez d'extraire la ligne contenant l'ancienne valeur, le système renvoie la valeur -2 pour @@FETCH_STATUS. Les nouvelles valeurs sont visibles si la mise à jour s'effectue via le curseur en spécifiant la clause **WHERE CURRENT OF**.

DYNAMIC

Définit un curseur qui reflète toutes les modifications de données apportées aux lignes dans son jeu de résultats lorsque vous faites défiler le curseur. Les valeurs de données, l'ordre et l'appartenance aux lignes peuvent changer à chaque extraction.

6.1.4 Verrouillages des lignes

READ_ONLY

Interdit les mises à jour par l'intermédiaire de ce curseur. Le curseur ne peut être référencé dans une clause WHERE CURRENT OF dans une instruction UPDATE ou DELETE. Cette option supplante la fonction implicite de mise à jour d'un curseur.

SCROLL_LOCKS

Spécifie que les mises à jour ou les suppressions positionnées, effectuées par l'intermédiaire du curseur sont sûres de réussir. Les lignes sont verrouillées dès qu'elles sont lues par le curseur, de manière à garantir leur disponibilité pour des modifications ultérieures.

OPTIMISTIC

Spécifie que les mises à jour ou les suppressions positionnées, effectuées par l'intermédiaire du curseur, échouent si la ligne a été mise à jour depuis qu'elle a été lue par le curseur. SQL Server ne verrouille pas les lignes quand elles sont lues par le curseur. Au contraire, il compare les valeurs de la colonne TIMESTAMP, ou une valeur de CHECKSUM si la table ne comprend pas de colonne TIMESTAMP, afin de déterminer si la ligne a été modifiée après avoir été lue par le curseur. Si la ligne a été modifiée, la mise à jour ou la suppression positionnée que vous avez tentée échoue.

6.1.5 Génération du jeu de résultats

Le jeu de résultats est généré à partir de l'instruction SELECT spécifiée lors de la définition du curseur.

6.1.6 Opérations de mise à jour

Il est possible de préciser à ce niveau quelles sont les colonnes qui pourront faire l'objet d'une mise à jour.

UPDATE [OF Colonne1 [...Colonnen]]

Définit les colonnes qui peuvent être mises à jour par le curseur.

Si vous indiquez UPDATE sans préciser de liste de colonnes, toutes les colonnes peuvent être mises à jour.

Si vous spécifiez OF Colonne1 [...Colonnen], seules les colonnes énumérées dans la liste peuvent être modifiées.

6.2 Ouverture et lecture

L'instruction OPEN déclenche l'exécution de la requête SELECT sous jacente et charge le jeu de résultats.

6.2.1 Lecture

L'instruction FETCH charge les valeurs de la ligne dans la liste de variables précisée en complément au niveau du mot clé INTO.

L'instruction **FETCH** peut être utilisée avec les arguments suivants :

NEXT

Renvoie la ligne de résultats immédiatement après la ligne courante, et incrémente cette dernière de la ligne renvoyée. Si FETCH NEXT est la première extraction effectuée sur un curseur, cette instruction renvoie la première ligne dans le jeu de résultats. NEXT est l'option d'extraction du curseur par défaut.

PRIOR

Renvoie la ligne de résultats immédiatement avant la ligne courante, et décrémente cette dernière en fonction de la ligne renvoyée. Si FETCH PRIOR est la première extraction effectuée sur un curseur, aucune ligne n'est renvoyée et le curseur reste placé avant la première ligne.

FIRST

Renvoie la première ligne dans le curseur et la transforme en ligne courante.

LAST

Renvoie la dernière ligne dans le curseur et la transforme en ligne courante.

ABSOLUTE {n | @nvar}

Si n ou @nvar est un nombre positif, cela renvoie l'énème ligne depuis le début du curseur et transforme la ligne renvoyée en nouvelle ligne courante. Si n ou @nvar est un nombre négatif, cela renvoie l'énème ligne avant la fin du curseur et transforme la ligne renvoyée en nouvelle ligne courante. Si n ou @nvar est égal à 0, aucune ligne n'est renvoyée. n doit correspondre à une valeur constante de type entier et @nvar doit être de type smallint, tinyint ou int.

RELATIVE {n | @nvar}

Si n ou @nvar est un nombre positif, cela renvoie l'énème ligne à partir de la ligne courante et transforme la ligne renvoyée en nouvelle ligne courante. Si n ou @nvar est un nombre négatif, il renvoie l'énème ligne avant la ligne courante et transforme la ligne renvoyée en nouvelle ligne courante. Si n ou @nvar est égal à 0, la ligne courante est renvoyée.

INTO

Permet de préciser la liste des variables recevant la valeur des colonnes.

Cette liste doit respecter l'ordre des colonnes de l'ordre SELECT à l'origine du jeu de résultats

@@FETCH_STATUS

Programmation des SGBDR-Partie 2

Renvoie l'état de la dernière instruction FETCH effectuée sur un curseur actuellement ouvert par la connexion.

Valeurs renvoyées par l'instruction FETCH

0 : L'instruction FETCH a réussi.

-1 : L'instruction FETCH a échoué ou la ligne se situait au-delà du jeu de résultats.

-2 : La ligne recherchée est manquante.

6.3 Un exemple à la loupe

Afin de mieux comprendre la syntaxe relative au curseur, je vous propose d'étudier l'exemple suivant qui permet d'affecter des droits sur des objets de la base de données.

6.3.1 Déclaration

```
CREATE PROCEDURE DefinirDroits
    (@Operation          sysname
    ,@User               sysname
    ,@Droit              sysname
    ,@typeObjet          char(2) )
AS
DECLARE @NomObjet sysname, @REQSQL varchar(255)

DECLARE Liste_Objets CURSOR
LOCAL -- Uniquement accessible dans cette procédure
FORWARD_ONLY -- Déplacement vers l'avant uniquement
READ_ONLY -- En lecture seule

FOR -- Intruction source du jeu de résultats
SELECT name
FROM sysobjects
WHERE category = 'O' and type = @typeObjet
```

Figure 18 : Déclaration du curseur

6.3.2 Processus de parcours du curseur

```
OPEN Liste_Objets -- Ouverture du curseur

FETCH NEXT FROM Liste_Objets -- Lecture de la ligne suivante
INTO @NomObjet -- Chargement dans la variable @NomObjet

WHILE @@FETCH_STATUS = 0 -- Itérer tant que pas fin de curseur
BEGIN -- Bloc Instructions exécuté
    SET @REQSQL = @Operation + ' ' + @Droit + ' ON [' + @NomObjet + '] TO [' + @user + ']'
    PRINT @REQSQL
    EXEC (@REQSQL)
    FETCH NEXT FROM Liste_Objets into @NomObjet
END
```

Figure 19 : Parcours et traitement du jeu de résultats

Notez : La construction dynamique de la chaîne SQL @RESQL, dont la valeur sera exécutée avec l'instruction EXEC. Cette approche s'avère très utile dès lors que l'instruction SQL n'accepte pas d'arguments sous formes de variables : ici une instruction de définition de droits (GRANT, REVOKE ou DENY).

6.3.3 Fermeture et libération mémoire

```
CLOSE Liste_Objets -- Fermeture du curseur  
DEALLOCATE Liste_Objets -- Libération de la mémoire
```

Figure 20 : Fermeture du curseur et destruction

6.4 Processus des curseurs en bref

Pour utiliser un curseur, vous devez mettre en œuvre le processus suivant :

- Associez un curseur au jeu de résultats d'une instruction SELECT et définissez les caractéristiques du curseur, en indiquant par exemple, si les lignes contenues dans le curseur peuvent être mises à jour.
- Exécutez l'instruction OPEN pour remplir le curseur.
- Dans le curseur que vous voulez parcourir, extrayez les lignes au moyen de la commande FETCH. L'opération consistant à récupérer une ligne ou un bloc de lignes à partir d'un curseur est appelée une extraction. Le défilement est l'opération consistant à effectuer une série d'extractions afin d'extraire des lignes vers l'avant ou vers l'arrière.
- Vous pouvez éventuellement effectuer des opérations de modification sur la ligne à la position actuelle du curseur.
- Fermez le curseur à l'aide de l'instruction CLOSE.
- Libérez la mémoire allouée au curseur avec DEALLOCATE