1. CREATION DE LA BASE «COMPAGNIE AERIENNE»

Doc. en ligne : « Créer et maintenir des bases de données »

1.1 Introduction

Dans ce chapitre, nous allons construire complètement la base de données « Compagnie Aérienne » qui a été présentée au début.

L'administrateur (compte **sa**) vous a accordé le droit de créer une nouvelle base de donnée, dont vous serez propriétaire (compte **dbo**, « data base owner ») : le dbo a tous les droits sur sa base, de même que l'administrateur a tous les droits sur la totalité du SGBDR.

Tous les exercices ci après peuvent être exécuté :

En mode texte en tapant des odres sql *Create*

Avec l'outil graphique.

Bien que l'outil graphique soit particulièrement performant, il est interessant de savoir construire un base en mode texte, car tous les SGBD ne propose pas d'outils graphique, et aussi ils sont tous différents . Nous proposons de faire d'abord les exercices en mode texte, puis ensuite avec l'outil graphique.

Loguez vous avec Management studio et ouvirez une fenêtre sql (nouvelle requête)

```
CREATE DATABASE AirXXX
```

1.2 Création des domaines (types de données utilisateur)

Doc. en ligne : « Accéder aux données et les modifier » : « Eléments de syntaxe Transact SQL » -> « Utilisation des types de données »

La première étape consiste à créer des « domaines », c'est-à-dire des types de données, qui s'appliqueront aux colonnes ou attributs des tables et des colonnes. Comme en C ou en Pascal, les domaines sont définis par le propriétaire de la base de données, à partir des types prédéfinis du langage SQL :

bit	entier dont la valeur est 1 ou 0.
tinyint	entier dont la valeur est comprise entre 0 et 255.
smallint	entier dont la valeur est comprise entre -2^{15} (-32 768) et $2^{15} - 1$ (32 767).
int	entier dont la valeur est comprise entre -2^{31} (-2 147 483 648) et $2^{31} - 1$ (2 147 483 647).
decimal	Données numériques fixes de précision et d'échelle comprises entre -10-1 et 10^{38} 10^{38}-1.
numeric	Synonyme de **decimal**.

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

smallmoney	Valeurs de données monétaires comprises entre - 214 748,3648 et +214 748,3647, avec une précision d'un dix-millième d'unité monétaire.
money	Valeurs de données monétaires comprises entre -2 (-922 337 203 685 477,580 8) et 2^{63} - 1 (+922 337 203 685 477,580 7), avec une précision d'un dix-millième d'unité monétaire.
real	Données numériques de précision en virgule flottante comprises entre -3.40E + 38 et 3.40E + 38.
float	Données numériques de précision en virgule flottante comprises entre -1.79E + 308 et 1.79E + 308.
smalldatetime	Données de date et d'heure comprise entre le 1 janvier 1900 et le 6 juin 2079, avec une précision d'une minute.
datetime	Données de date et d'heure comprises entre le 1 janvier 1753 et le 31 décembre 9999, avec une précision de trois centièmes de seconde ou de 3,33 millisecondes.
char	chaîne de caractères de longueur fixe d'un maximum de 8 000 caractères.
varchar	chaîne de caractères de longueur variable d'un maximum de 8 000 caractères.
text	texte de longueur variable ne pouvant pas dépasser 2^{31} - 1 (2 147 483 647) caractères.
binary	données binaires de longueur fixe ne pouvant pas dépasser 8 000 octets.
varbinary	Données binaires de longueur variable ne pouvant pas dépasser 8 000 octets.
image	Données binaires de longueur variable ne pouvant pas dépasser 2^{31} - 1 (2 147 483 647) octets.

La syntaxe pour créer des nouveaux types ne fait pas partie de la norme ANSI du SQL. Sous SQL SERVER il faut appeler la procédure système sp_addtype (system procedure add type), par le mot clé**execute**

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

execute sp_addtype TypeNom, 'VARCHAR(50)', 'NOT NULL'

```
sp_addtype nom_de_type, type_physique [, contrainte_de_valeur_NULL]

     nom_de_type : Nom du type de données défini par l'utilisateur.

     type_physique : Nom du type de données physique prédéfini de SQL
     Server sur lequel repose le type de données défini par l'utilisateur.

     contrainte_de_valeur_NULL : Paramètre indiquant comment gérer des
     valeurs NULL dans un type de données défini par l'utilisateur (vaut
     NULL ou NOT NULL).
```

SQL Server ne fait pas de différence entre les majuscules et les minuscules pour les noms de type de données système.

Exercice 20

Les exercices qui suivent vont vous guider pour écrire un script de création de base de données : Creer.SQL.

On commentera chacune des étapes du script(commentaire Transact SQL : **/*** plusieurs lignes ***/** ou **--** fin de ligne*)*

On développera en parallèle un script Supprimer.SQL qui supprime tous les objets que l'on vient de traiter. Après chaque exercice, on relancera les scripts de suppression et de création, pour s'assurer que tous les objets se créent correctement.

A la fin de ce chapitre, on doit donc aboutir à deux scripts complets qui permettent de maintenir la base (en la recréant complètement en cas de crash système…)

Etape 1 : création des types utilisateurs :

TypeDate *: type prédéfini****datetime****. Obligatoire. (=> date du vol dans la table Affectation).*

TypeHoraire *: type prédéfini smalldatetime. Obligatoire (=> heures de départ et d'arrivée, dans la table Vol)*

TypeNom *: chaîne variable limitée à 50 caractères. Obligatoire. (=> nom du pilote dans la table Pilote et du constructeur dans la table Constructeur)*

TypePrenom *: chaîne variable limitée à 30 caractères. Optionnel. (=> prénom du pilote dans la table Pilote)*

TypeAvion *: chaîne variable limitée à 20 caractères. Obligatoire. (=> type d'avion dans les tables Type et Avion)*

TypeIdConstructeur *: entier (smallint). Obligatoire (=> identificateur du constructeur dans la table Constructeur)*

TypeIdAeroport *: chaîne fixe de 3 caractères. Obligatoire. (=> identificateur de l'aéroport dans la table Aéroport)*

TypeIdPilote *: entier (smallint). Obligatoire (=> identificateur du pilote dans la table Pilote)*

TypeNumVol *: chaîne fixe de 5 caractères. Obligatoire (=> identificateur du vol dans la table Vol)*

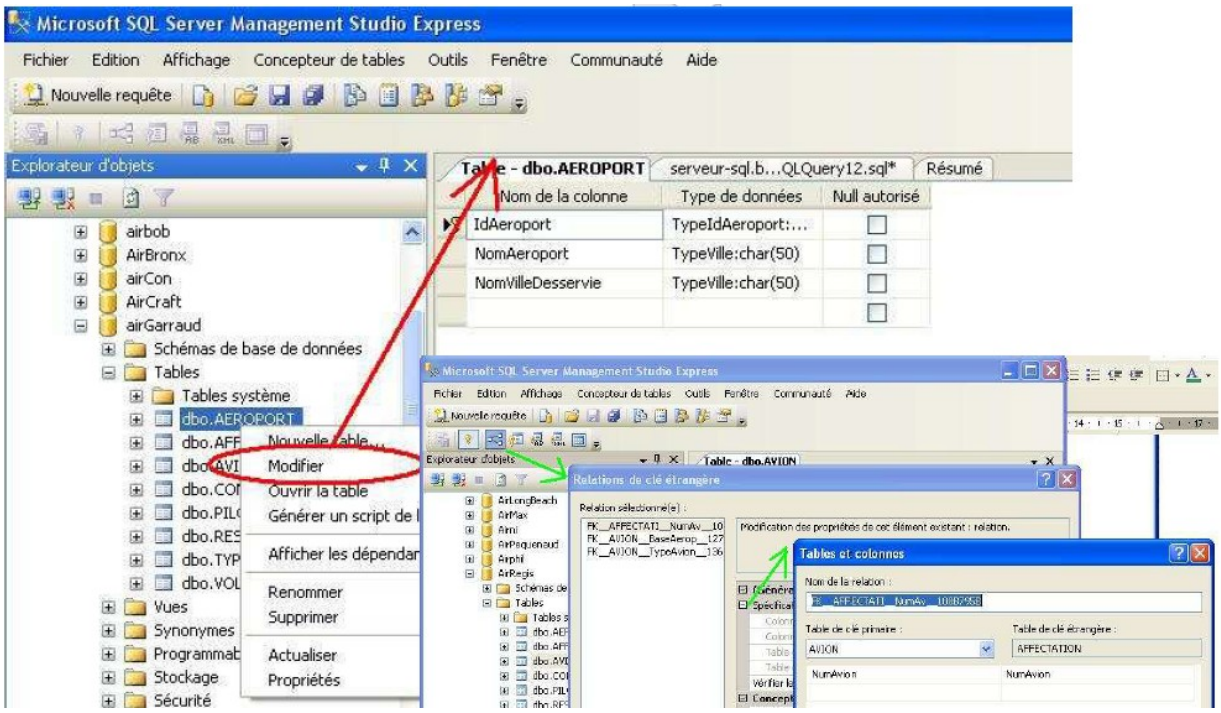
TypeVille *: chaîne variable limitée à 50 caractères. Optionnel. (=> nom de l'aéroport et de la ville desservie dans la table Aeroport)*

Avec sql server management studio express (voir page 10)

lancer l'outil par le menu « SQL Server 2017 Management studio express»

A l'invite loguer vous avec votre identifiant sql serveur (user :votre nom / password :votre nom).

Développer « Bases de données » : seules les bases de données auxquelles vous avez accès apparaissent. Développer la base en cours de construction, et le répertoire « Types de données utilisateurs ».



1.3 Création et suppression de tables. Contraintes d'intégrité

Doc. en ligne : « Créer et maintenir des bases de données » : « Tables »

Clés primaires et étrangères

La *clé primaire*est une colonne ou une combinaison de colonnes dont les valeurs identifient de façon unique chaque ligne dans la table : elle ne peut pas être nulle .

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

On crée une clé primaire grâce à la contrainte PRIMARY KEY lors de la création ou de la modification d'une table. SQL crée automatiquement un objet « index » pour la clé primaire : l'index assure l'unicité de la clé primaire dans la table, et permet un accès rapide aux données par la clé primaire.

Une *clé étrangère*est une colonne ou une combinaison de colonnes dont les valeurs correspondent à la clé primaire d'une autre table. Elle n'est pas obligatoirement unique ni définie (peut être NULL), mais ses valeurs doivent correspondre à des valeurs existantes de la clé primaire. On crée une clé étrangère par la contrainte FOREIGN KEY lors de la création ou de la modification d'une table.

Intégrité des données dans une base

Assurer l'intégrité des données, c'est préserver la cohérence et l'exactitude des données stockées dans une base de données en validant le contenu des différents champs, en vérifiant la valeur des champs l'un par rapport à l'autre, en validant les données dans une table par rapport à une autre, et en vérifiant que la mise à jour d'une base de données est efficacement et correctement effectuée pour chaque transaction.

4 contraintes d'intégrité :

Intégrité d'entité : unicité de la clé primaire (PRIMARY KEY) ou d'autres clés (UNIQUE)

Intégrité de domaine : plage des valeurs possibles pour une colonne. On peut restreindre cette plage en attribuant à la colonne un type défini par l'utilisateur, ou par une contrainte CHECK avec une règle.

L'intégrité référentielle garantit la relation entre une clé primaire unique dans une table et les clés étrangères qui y font référence dans les autres tables. Exemple : avant de supprimer une ligne dans la table PILOTE, il faut supprimer toutes les lignes de la table VOL qui font référence à ce pilote.

Pour assurer l'intégrité référentielle, SQL Server interdit de :

- ajouter des enregistrements à une table liée lorsqu'il n'y a aucun enregistrement associé dans la table primaire ;
- changer des valeurs ou effacer des enregistrements dans une table primaire qui engendrerait des enregistrements «orphelins» dans une table liée;

Par exemple, avec les tables *ventes* et *titres* dans la base de données*pubs*, l'intégrité référentielle est basée sur la relation entre la clé étrangère*id*(*_titre*) de la table *ventes* et la clé primaire (*id_titre*) de la table *titres*, comme le montre l'illustration suivante:

Exemples de création de table : CreateTable

Lorsque vous créez une table, nommez ses colonnes, définissez un type de donnée pour chaque colonne, précisez si elles peuvent contenir des valeurs NULL, définissez des contraintes par l'option CHECK, des valeurs par défaut par DEFAULT

```
CREATE TABLE emplois                                        Clé primaire
(
    id_emploi       smallint        PRIMARY KEY,                    Valeur par défaut
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
        desc_emploi     varchar(50)     NOT NULL DEFAULT 'Nouveau poste',
        niv_min         tinyint         NOT NULL    CHECK (niv_min >= 10),
        niv_max         tinyint         NOT NULL    CHECK (niv_max <= 250)
)
```

Contrainte

FOREIGN KEY : la clé fait référence à la clé primaire d'une autre table.

id_emploi smallint NOT NULL DEFAULT 1 REFERENCES emplois(id_emploi)

ou id_emploi smallint et plus loin FOREIGN KEY (id_emploi) REFERENCES emplois (id_emploi)

UNIQUE : assure l'unicité d'une clé secondaire

pseudonyme varchar(30) NULL UNIQUE

DEFAULT : valeur par défaut lorqu'aucune valeur n'est donnée dans INSERT ou UPDATE

CHECK :

```
CHECK (id_éditeur IN ('1389', '0736', '0877', '1622', '1756')
        OR id_éditeur LIKE '99[0-9][0-9]')

CHECK (niv_min >= 10)
```

Nom contrainte

ou avec mot clé CONSTRAINT :

CONSTRAINT CK_id_employé CHECK (id_employé LIKE '[0-9][0-9][0-9][0-9][FM]')

IDENTITY : valeur numérique autoincrémentale (valeur initiale et valeur de l'incrément)

```
IDENTITY(10,3)      (=> valeurs: 10, 13, 16…)
```

Exercice 21 : Création des tables et des contraintes

Compléter le script de création pour créer les tables de la base « Compagnie aérienne », décrite dans le schéma général. Pour chaque table, on définira les liens clé étrangère-clé primaire en suivant le schéma.

Définitions de contraintes supplémentaires sur les tables :

AVION : la clé primaire NumAvion est autoincrémentée à partir de 100, par pas de 1.

TYPE : la clé primaire TypeAvion commence obligatoirement par une lettre
la capacité est comprise entre 50 et 400 : cette colonne est obligatoire, avec un défaut de 100

CONSTRUCTEUR : la clé primaire IdConstructeur est autoincrémentée à partir de 1, par pas de 1

PILOTE : la clé primaire IdPilote est autoincrémentée à partir de 1, par pas de 1.
La combinaison (nom, prénom) est unique.

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

AEROPORT : la clé primaire IdAeroport ne comporte que des lettres
la colonne NomVilleDesservie est optionnelle
la colonne NomAeroport est obligatoire

VOL : la clé primaire NumVol est constituée du préfixe " IT" suivi de trois chiffres

AFFECTATION : la clé primaire est composée des colonnes NumVol et DateVol
la clé étrangère IdPilote peut être NULL (en attente d'affectation à un pilote)

Commencer par établir la liste de dépendance :

- *numéroter 0 les tables indépendantes, 1 les tables qui ne dépendent que de tables indépendantes, 2 celles qui dépendent des précédentes…*
- *créer les tables en commençant par les indépendantes, et en suivant l'ordre de la liste de dépendance.*

Vérifier la création des tables sous Entreprise Manager

Exercice 22 : Script de test

Tester les contraintes d'intégrité en faisant des essais de remplissage par l'instruction INSERT et des suppressions par DELETE : vérifier les valeurs par défaut, les domaines de validité des colonnes…
Partir des valeurs décrites dans le chapitre I, en remplissant les tables dans l'ordre de la liste de dépendance.
Exercice 23 : Script de destruction des types utilisateurs et des tables

Pour détruire une table :

```
DROP TABLE AVION
GO
```

Le GO oblige le moteur à faire la suppression immédiatement et permet de recréer la table (sinon erreur « Il y a déjà un objet Avion dans la base de données »)

Attention : il faut détruire les tables dans l'ordre inverse de la création, en supprimant d'abord les tables dépendantes

Détruire les types à la fin, quand ils ne sont plus utilisés par aucune table :

```
exec sp_droptype  TypeNom
```

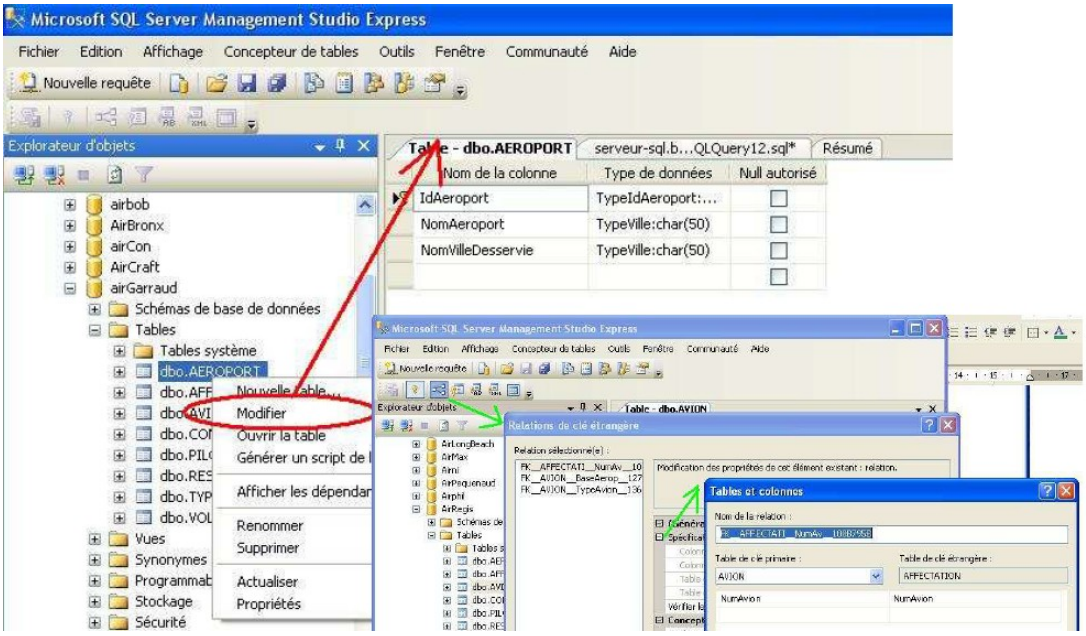
☺ Avec sql server management studio express (voir page 7 à 11)

Dans le répertoire « Bases de données », « Tables », visualisez et vérifiez les tables et leurs contraintes.

Pour visualiser les propriétés d'une table, avec le bouton droit de la souris, choisir « Modifier une table »

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Pour obtenir les détails sur les contraintes, et les clés étrangères : cliquer l'outil relation.



1.4 Création de vues

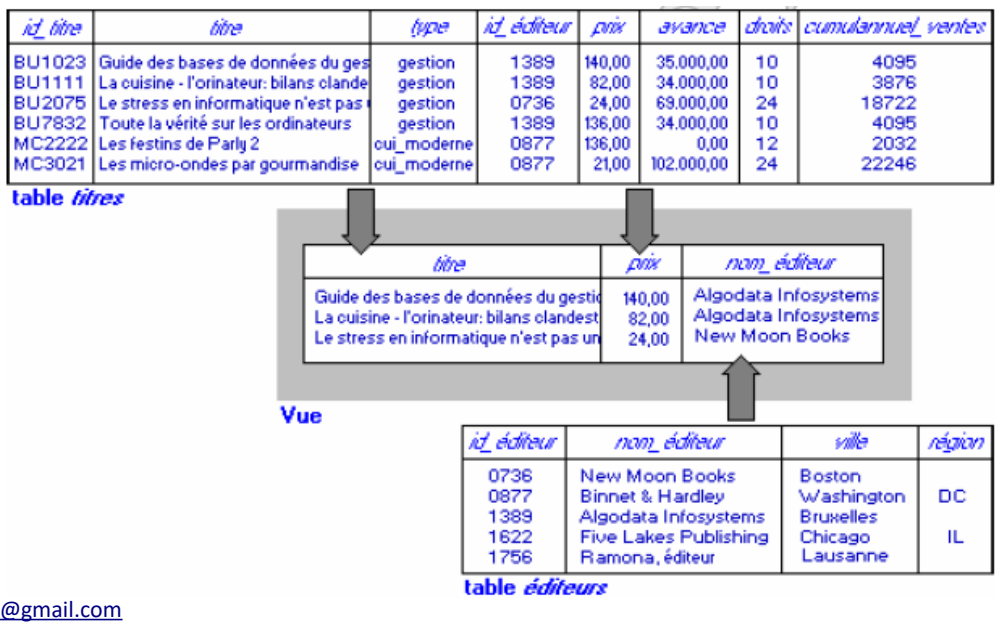
Doc. en ligne : « Créer et maintenir des bases de données » : « Vues »
SQL2 : p. 133 à 149

Quand utiliser des vues ?

Une *vue* est une table virtuelle dont le contenu est défini par une requête SELECT. Comme une table réelle, la vue possède des colonnes nommées et des lignes de données, mais elle n'est pas stockée dans la base de données, et n'a pas d'existence propre, indépendamment des tables.

Les vues améliorent la sécurité en permettant de contrôler les données que les utilisateurs peuvent visualiser, et l'ergonomie, en présentant les données sous une forme familière, indépendamment de la structure interne de la base .

Une vue peut être manipulée exactement comme une table : lorsqu'on modifie une vue, on modifie en fait les tables sur lesquelles elle s'appuie. Inversement, si on modifie les tables, les modifications sont reportées automatiquement dans toutes les vues qui leur font référence. La suppression d'une vue n'influe pas sur le contenu de la table associée. Par contre la suppression d'une table supprime toutes les vues qui lui font référence.



Paul EDIDE
Ingénieur S
paul.edide@gmail.com

Fonction CREATE VIEW

```
CREATE VIEW vuetitres
AS
SELECT titre, type, prix, datepub
FROM titres
```

Ou avec les colonnes explicites

```
CREATE VIEW vueEtiquette (TitreOuvrage, PrixOuvrage)
AS
SELECT titre, prix
FROM titres
```

Limitations :

Vous pouvez créer des vues uniquement dans la base de données courante.

Vous pouvez construire des vues à partir d'autres vues ou de procédures qui font référence à des vues.

Si vous remplissez une vue par une instruction SELECT, vous n'avez en général pas besoin de donner les noms des colonnes, qui seront ceux du SELECT, sauf dans les cas suivants :

- colonnes dérivées d'une expression arithmétique, d'une fonction ou d'une constante.
- plusieurs colonnes issues de tables différentes auraient le même nom.
- on veut donner un nom à la colonne de la vue, différent du nom de la colonne dans la table (présentation pour l'utilisateur de la base de données)

Vous ne pouvez pas associer des règles, des valeurs par défaut ou des Déclencheurs à des vues, ni construire des Index à partir de vues.

Vous ne pouvez pas créer de vues temporaires et vous ne pouvez pas créer de vues à partir de tables temporaires.

Vous ne pouvez pas modifier par UPDATE, INSERT ou DELETE les vues qui dérivent de plusieurs tables par jointure, ou qui contiennent les clauses GROUP BY, ou DISTINCT, ou qui omettent un attribut obligatoire de la table (déclaré NOT NULL) : ces vues ne seront accessibles qu'en lecture.

Pour une table modifiable, la clause WITH CHECK OPTION contrôle les modifications sur la vue.

Exercice 24 création d'une vue « Depart »

On veut disposer d'une vue qui visualise les départs du jour, à partir de Paris, avec la présentation ci-dessous :

NumVol	*De*	*A*	*Heure de Départ*	*Heure d'arrivée*
IT101	*ORL*	*BLA*	*11H15*	*12H40*
IT102	*CDG*	*NIC*	*12H31*	*14H00*
IT110	*ORL*	*NIC*	*15H24*	*16H00*

Remarques :

- *La table AEROPORT donne la liste des aéroports desservant chaque ville : ORLY (ORL) et Charles de Gaulle (CDG) pour la ville de Paris. La table AFFECTATION donne les dates de tous les vols.*
- *Utiliser la fonction **DATENAME()** de Transact SQL pour extraire l'heure et les minutes des colonnes VOL.Hdépart et VOL.HArrivée, qui sont stockées en smalldatetime **DATENAME**: **(hh, Hdépart)** renvoie l'heure du départ, **DATENAME (mi, Hdépart)** renvoie les minutes*

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

- *On obtient la date du jour par la fonction **GETDATE()**. Cette date ne peut pas être comparée directement à AFFECTATION.DateVol, car elle comprend les heures, minutes, secondes. Le plus simple est de comparer le quantième du jour de la date courante, au quantième de la date du vol : la fonction **DATENAME (dayofyear, uneDate**) extrait le quantième du jour de la date « uneDate »*

Tester la vue par un select simple. Rajouter la création de la vue dans le script de création de la base, et sa suppression dans le script de suppression.

1.5 Définition des permissions sur les objets d'une base de donnée

Contrôle d'accès aux données dans un SGBD

L'accès aux données est contrôlé par le SGBD au moyen de l'identification de l'utilisateur qui lors de la connexion doit fournir le nom d'utilisateur et le mot de passe associé. Chaque objet d'une base de données mémorise la liste des utilisateurs autorisés à le manipuler : son créateur et les autres utilisateurs qui auront été habilités par lui.

L'administrateur système de SQL SERVER (compte sa) est le seul à posséder tous les droits sur toutes les bases de données : création d'une nouvelle base, suppression… Ces droits ne peuvent pas lui être retirés (indépendants des droits explicites sur la base)

Le propriétaire d'une base de données (DBO : Data Base Owner) a tous les droits sur les objets de sa base, et peut les donner à d'autres utilisateurs par la commande GRANT ou leur retirer des droits par la commande REVOKE

Attribution de droits : GRANT

SQL2 : p. 171-173

La requête GRANT permet de donner à un utilisateur ou un groupe d'utilisateurs la permission d'utiliser certaines instructions de création ou de suppression d'objet (CREATE TABLE, CREATE VIEW, DROP TABLE, DROP VIEW…) ou des droits d'accès sur des objets donnés par SELECT, INSERT, UPDATE, DELETE, ALTER, ou ALL .

Pour autoriser l'accès à un objet pour tous les utilisateurs on utilisera le groupe PUBLIC

Permissions d'instruction :

GRANT {ALL |liste_d'instructions} TO {PUBLIC | liste_de_noms}

GRANT CREATE DATABASE, CREATE TABLE
TO Marie, Jean

Permissions d'objet :

GRANT {ALL | liste_de_permissions}ON {nom_de_table [(liste_de_colonnes)] | nom_de_vue [(liste_de_colonnes)] | nom_de_procédure_stockée | nom_de_procédure_stockée_étendue}TO {PUBLIC | liste_de_noms}

Il faut d'abord accorder des permissions générales au groupe public, puis préciser en accordant ou en retirant des permissions spécifiques à certains utilisateurs :

GRANT SELECT
ON auteurs
TO public
go

Marie , Jean et Pierre auront tous les privilèges sur la table auteurs
GRANT INSERT, UPDATE, DELETE ON
auteurs
TO Marie, Jean, Pierre

Suppression de droits : REVOKE

Doc. en ligne : chercher « REVOKE (T-SQL) »

Permissions d'instruction :

REVOKE {ALL | liste_instructions} FROM {PUBLIC | liste_de_noms}

REVOKE CREATE TABLE, CREATE DEFAULT
FROM Marie, Jean

Permissions d'objet :

REVOKE {ALL | liste_de_permissions} ON {nom_de_table [(liste_de_colonnes)] | nom_de_vue [(liste_de_colonnes)] | nom_de_procédure_stockée | nom_de_procédure_stockée_étendue} FROM {PUBLIC | liste_de_noms}

Exercice 25 : définition des permissions sur les tables et les vues

Dans chaque groupe de quatre, un des comptes représentera l'usager de base qui consulte les horaires, les deux autres seront des employés de l'aéroport dotés de permissions limitées, qui assistent le propriétaire de la base : un responsable des horaires et un responsable de la planification

Sous le compte du propriétaire de la base (DBO) qui a tous les droits sur la base :

- ***Créer les trois nouveaux utilisateurs dans votre base (usager, Responsable_horaires, Responsable_planification), correspondant aux trois comptes d'accès de votre groupe, avec la syntaxe :***

exec sp_adduser nom_de_connexion, nom_utilisateur_base

- ***Pour l'usager, permettre la lecture de la vue Depart, pour consultation dans l'aéroport. Aucun droit sur les tables !***

- ***Le responsable des horaires a tous les droits sur Vol, et peut consulter Avion, Aeroport et Type***

- ***Le responsable planification a le droit de lire, modifier, ajouter et supprimer sur la table Affectation, et de lire les tables Avion, Vol, Pilote, et Type***

- ***Les deux assistants peuvent créer des vues .***

En se loggant sous les trois comptes, faire des requêtes manuelles, pour vérifier que les permissions fonctionnent comme prévu.

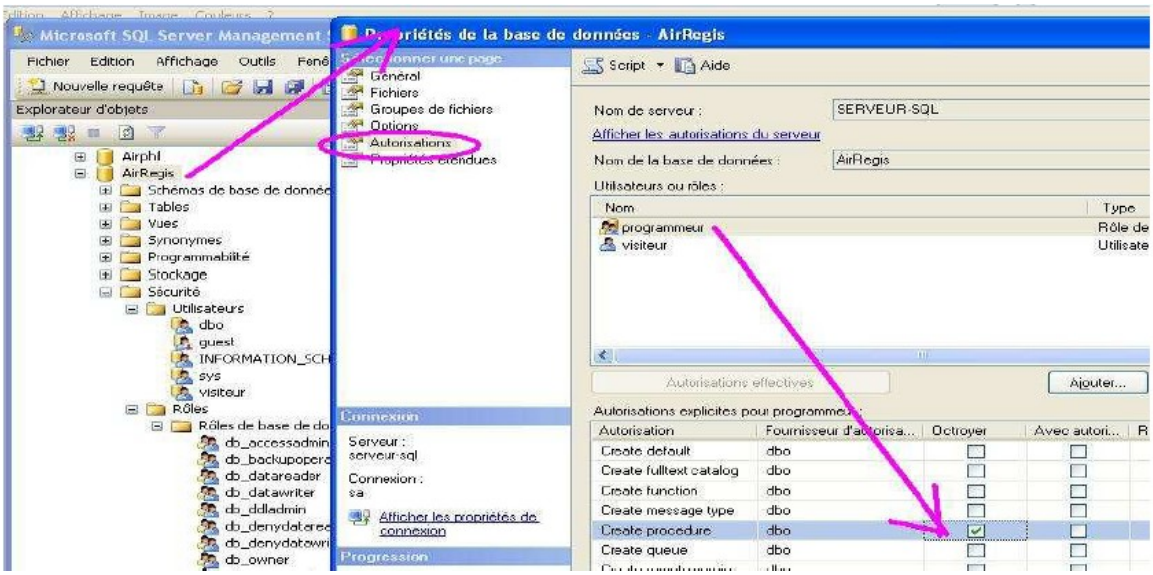
Rajouter la création des nouveaux utilisateurs et la définition des permissions à la fin du script de création de la base, et la suppression des utilisateurs dans le script de suppression.

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

☺ Avec sql server management studio express (voir page 15 à 23)

Permissions sur la base :

En vous loggant avec le compte propriétaire de la base (dbo), vous pouvez visualiser et modifier les permissions d'instruction (CREATE TABLE, CREATE VIEW…) des différents utilisateurs de votre base : sélectionnez le nom de votre base, et choisissez « Propriétés » avec le bouton droit de la souris.

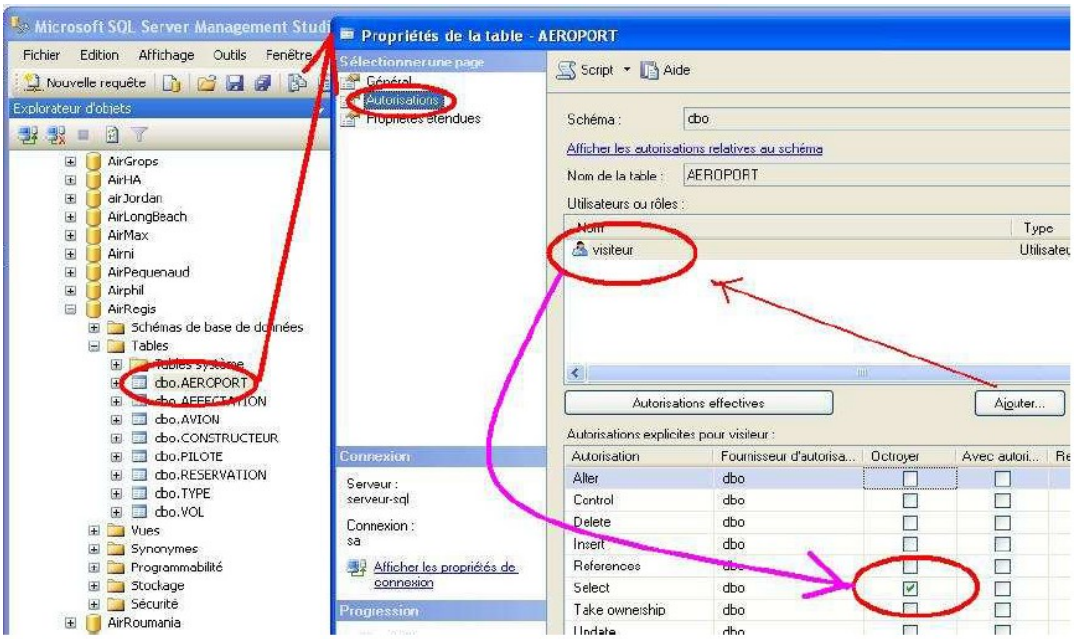


Permission sur les objets

Vous pouvez aussi visualiser et modifier les **permissions d'objet** sur les tables (SELECT, INSERT, UPDATE, DELETE) :

choisissez « Propriétés » avec le bouton droit de la souris.

puis clickez sur le bouton « Autorisations »



Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

1.6 Les procédures stockées

Utilité des procédures stockées

Une procédure stockée est un ensemble d'instructions compilées, qui peut s'exécuter plus rapidement. Les procédures stockées augmentent la puissance et la performance du langage SQL. Elles peuvent :

- recevoir et renvoyer des paramètres à une procédure appelante,
- appeler d'autres procédures,
- renvoyer une valeur de sortie à la procédure appelante

Elles supportent les boucles et les if traditionnels de l'algo. Elles allient donc la puissance de SQL et de la programmation. Les noms de variables commencent par '@' afin de les distinguer des noms de table et de champs. Les variables sont typées.

Les procédures stockées sur d'autres serveurs SQL auxquels le processus client n'est pas directement connecté peuvent être exécutées si le serveur distant autorise les accès distants.

Les procédures stockées diffèrent des autres instructions et lots d'instructions SQL parce qu'elles sont préanalysées et prénormalisées. Lorsque vous exécutez une procédure pour la première fois, le programme de traitement des requêtes de SQL Server l'analyse et prépare une structure interne normalisée pour la procédure qui est stockée dans une table système. Lors de la première exécution de la procédure au démarrage du serveur SQL, elle est chargée en mémoire et compilée complètement (elle n'est plus analysée ou mise en ordre puisque cela a été fait lors de sa création). Le plan compilé reste alors en mémoire (sauf si le besoin de mémoire l'oblige à en sortir) pour que sa prochaine exécution (effectuée par le même client ou par un autre) puisse être traitée sans que le plan de compilation consomme des ressources système.

Les procédures stockées peuvent servir de mécanismes de sécurité, car un utilisateur peut avoir la permission d'exécuter une procédure stockée même s'il n'a aucune permission sur les tables ou les vues auxquelles elle fait référence Il est de bonne pratique de faire toutes les mises à jour via des procédures stockées.

Création de procédures stockées : CREATE PROCEDURE.

Doc. en ligne : « Créer et maintenir des bases de données » : « Procédures Stockées »
Aide *Transact-SQL* : CREATE PROCEDURE et EXECUTE
SQL2 : p.114

Les procédures stockées sont des objets de base de données : la permission d'exécuter CREATE PROCEDURE revient par défaut au propriétaire de base de données qui peut la transmettre à d'autres utilisateurs.

Vous ne pouvez créer une procédure stockée que dans la base de données courante.

Une procédure stockée peut contenir toutes les instructions SQL, sauf CREATE. Elle peut appeler une autre procédure stockée jusqu'à 16 niveaux d'imbrications.

Pour créer une procédure stockée

Dans Enterprise Manager, dans votre base de données, choisissez l'option «Procédures stockées», et ensuite «Nouvelle Procédure Stockée» avec le bouton droit de la souris.

Ou

Utilisez l'instruction SQL**CREATE PROC** :

la procédure stockéeinfo_auteur reçoit en paramètres le nom et le prénom d'un auteur, et affiche le titre et l'éditeur de chacun des livres de cet auteur :

```
CREATE PROC info_auteur @nom varchar(40), @prénom varchar(20)
AS
    SELECT    nom_auteur, pn_auteur, titre, nom_éditeur
    FROM      auteurs, titres, éditeurs, titreauteur
    WHERE     pn_auteur = @prénom
       AND    nom_auteur = @nom
       AND    auteurs.id_auteur = titreauteur.id_auteur
       AND    titres.id_titre = titreauteur.id_titre
       AND    titres.id_éditeur = éditeurs.id_éditeur
go
```

@ avant le nom du paramètre

Pour exécuter la procédure :

```
Execute  info_auteur 'Chevalier', 'Bernard'
```

nom_auteur	pn_auteur	titre	nom_éditeur
Chevalier	Bernard	La colère : notre ennemie ?	New Moon Books
Chevalier	Bernard	Vivre sans crainte	New Moon Books

la procédure info_éditeur2 affiche le nom des auteurs qui ont écrit un livre publié par l'éditeur passé en paramètre : si aucun nom d'éditeur n'est précisé, la procédure fournit les auteurs publiés par Algodata Infosystems.

```
CREATE PROC info_éditeur2 @nom_éditeur varchar(40) = 'Algodata Infosystems'
AS
   SELECT nom_auteur, pn_auteur, nom_éditeur
   FROM      auteurs a, éditeurs e, titres t, titreauteur ta
   WHERE @nom_éditeur = e.nom_éditeur
      AND      a.id_auteur = ta.id_auteur
      AND      t.id_titre = ta.id_titre
      AND      t.id_éditeur = e.id_éditeur
```

Paramètre par défaut

Si la valeur par défaut est une chaîne de caractères contenant des espaces ou des marques de ponctuation, ou si elle débute par un nombre (par exemple, 6xxx), elle doit figurer entre guillemets anglais simples.

La procédure peut spécifier les actions à accomplir si l'utilisateur ne fournit pas de paramètre :

```
CREATE PROC montre_index3 @table varchar(30) = NULL
AS
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
 IF @table is NULL
     PRINT 'Entrez un nom de table'
 ELSE
   -- traitement du paramètre…
```

L'instruction 'select' permet de créer un recordset en sortie.
L'instruction 'print' permet d'afficher un message sous isql.

Dans la pratique les procédures stockées seront appelées par un client VB et l'instruction print sera inutilisable.

L'option OUTPUT derrière un paramètre, indique un paramètre en sortie, qui est renvoyé à la procédure appelante : *somme_titres* est créée avec un paramètre d'entrée facultatif *titre_sélectionné*et un paramètre de retour *somme*

```
CREATE PROC somme_titres @titre_sélectionné varchar(40) = '%', @somme money
OUTPUT AS
        SELECT 'Liste des Titres ' = titre
        FROM titres
        WHERE titre LIKE @titre_sélectionné

       SELECT @somme = SUM(prix)
         FROM titres
         WHERE titre LIKE @titre_sélectionné
go
```

Appel d'une procédure stockée avec un paramètre en sortie :

Déclaration de variable

```
 DECLARE @coût_total money

 EXECUTE somme_titres 'Les%', @coût_total OUTPUT

 IF @coût_total < 1000
```

Appel procédure

```
                                                BEGIN
                                                    PRINT ' '
        PRINT 'L''ensemble de ces titres peut s''acheter pour moins de 1 000 FF.'
 END
 ELSE
        PRINT 'Le coût total de ces livres s''élève à ' + convert (varchar(20), @coût_total)
 go
```

```
 Liste des Titres
 ----------------------------------------------
 Les festins de Parly 2
 Les micro-ondes par gourmandise
 Les secrets de la Silicon Valley

 L'ensemble de ces livres peut s'acheter pour moins de 1 000 FF.
```

Dans une procédure stockée, vous pouvez aussi retourner une valeur par l'instruction **RETURN**, et la récupérer dans l'appelant par la syntaxe suivante :

```
DECLARE @coût_total money

EXECUTE @coût_total = somme_titre 'Les%'
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Ce code retour exprime souvent le bon ou mauvais déroulement de la procédure.

Exercice 26 : création et test de procédures stockées

Créer une procédure stockée « Planning_perso » qui affiche, pour un Id pilote passé en paramètre, les numéros de vol, avec les aéroports de départ et d'arrivée, classés par date et heure de départ.

```
**** Planning personnel de LECU Régis

NumVOL      De    Vers  partant le
-----------------------------------------
IT105                 LYS            ORL             6 avril         6:0
IT102                 CDG            NIC             6 avril         12:0
IT108                 BRI            ORL             6 avril         19:0
IT103                 GRE            BLA             7 avril         9:0
IT107                 NIC            BRI             6 mai           7:0
IT107                 NIC            BRI             7 juin          7:0
```

Si le paramètrese est omis, la procédure affichera le message d'erreur : « Erreur : Idpilote non défini ». Si le pilote n'existe pas, la procédure affichera « Erreur : pilote inexistant ». Tester la procédure, et rajouter la au script de création de la base.

NB dans la pratique, pour plus d'un paramètre, il n'est pas possible de toujours se protèger contre un parmètre manquant, un paramètre manquant dans la séquence d'appel provoque souvent une exception dans la procédure stockée. Par contre il est de bonne pratique de vérifier que les paramètres d'entrée n'ont pas de valeur nulle.

Exercice 27 : Procédures imbriquées avec passage de paramètres, et valeur de retour

1) Faire un SELECT qui affiche la somme des heures de vol d'un pilote donné, pour un numéro de semaine donné, avec la présentation suivante :

```
-------------------------------------------------------------------
Semaine 49,  Heures de Vols de LECU Régis =  3
```

On utilisera la fonction DATEDIFF pour calculer la différence entre l'heure de départ et d'arrivée, et DATENAME avec l'option week pour extraire le numéro de semaine de la date du vol.

2) Ranger le nom et le prénom du pilote, et le numéro de semaine dans des variables intermédiaires Transact SQL. Stocker la somme des heures de vol dans une variable de type INT, et préparer le texte à afficher dans une variable de type VARCHAR.

Pour formater le texte à afficher, on aura besoin de la fonction CONVERT (Ressources : exemple précédent et aides sur Convert, et Print dans Transact SQL)

3) En partant du SELECT précédent, faire une procédure Horaire qui renvoie la somme des heures de vol d'un pilote quelconque pour une semaine donnée, et tester la en affichant son résultat.

Une procédure stockée doit être une boîte noire : ne pas afficher de messages d'erreurs mais retourner un code d'erreur à l'appelant : 0 si succès, -1 si pilote inexistant. Le nom, le prénom du pilote et le numéro de semaine seront définis comme paramètres obligatoires.

4) Réaliser une procédure ObjectifHebdomadaire qui
- affiche le numéro de semaine demandée, le nom et le prénom du pilote, et son nombre d'heures hebdomadaire (renvoyé par Horaire)
- compare cette somme à un objectif passé en paramètre : nombre d'heures demandées

- affiche « Objectif atteint » ou « non atteint »

Ajouter Horaire et ObjectifHebdomadaire, aux scripts de création et de suppression

Remarques :

Les procédures stockées ci avant donneront des résultats faux si il existe des pilotes de même nom et prénom. Il sera donc prudent de vérifier que il y a un et un seul pilote.

Une alternative intéressante consiste à fournir en paramètre d'entrée l' ID du pilote, plutôt que nom et prénom

1 .7 Les déclencheurs (trigger)

Doc. en ligne : « Créer et maintenir des bases de données » : « Respect des règles d'entreprise à l'aide de déclencheurs »
Aide Transact SQL : CREATE TRIGGER
SQL2 : p. 107 à 111

Définition

Un *déclencheur* (trigger) est un type particulier de procédure stockée qui s'exécute automatiquement lorsque vous modifiez ou supprimez des données dans une table.

Les trigger garantissent la cohérence des données liées logiquement dans différentes tables : par exemple, un trigger permettra de mettre à jour toutes les lignes de la table AFFECTATION, liées à l'avion que l'on va supprimer dans la table AVION.

Chaque trigger est spécifique à certaines opérations sur les données : UPDATE, INSERT ou DELETE. Il s'exécute immédiatement après l'opération qui l'a lancé : le trigger et l'opération forment une seule transaction qui peut être annulée par le trigger. Si une erreur grave est détectée, toute la transaction est automatiquement annulée (en langage de bases de données, une transaction désigne une suite indivisible d'instructions SQL, qui s'effectuent toutes ou pas du tout ; cette notion sera détaillée dans les chapitres suivants)

Exemples d'utilisation :

Pour effectuer des changements en cascade dans des tables liées de la base de données : un déclencheur UPDATE sur **titres.id_titre** provoque une mise à jour des lignes correspondant à ce titre dans les tables **titreauteur**, **ventes** et **droits_prévus**.

Pour assurer des restrictions plus complexes que par la contrainte CHECK : contrairement à CHECK, les déclencheurs peuvent faire référence à des colonnes dans d'autres tables .

Pour trouver la différence entre l'état d'une table avant et après une modification des données, et accomplir une ou plusieurs actions en fonction de cette différence .

Syntaxe du CREATE TRIGGER

Un déclencheur est un objet de base de données. Pour créer un déclencheur, vous devez spécifier la table courante et les instructions de modification des données qui l'activent. Ensuite, vous devez spécifier la ou les actions que le déclencheur devra entreprendre.

Une table peut avoir au maximum trois déclencheurs: un déclencheur UPDATE, un déclencheur INSERT et un déclencheur DELETE.

Chaque déclencheur peut avoir de nombreuses fonctions et appeler jusqu'à 16 procédures. Un déclencheur ne peut être associé qu'à une seule table, mais il peut s'appliquer aux trois opérations de modifications de données (UPDATE, INSERT et DELETE).Exemple 1 :

```
CREATE TRIGGER pense_bête
                                                    Nom de la table
ON  titres

FOR INSERT, UPDATE                                  Conditions de déclenchemen

AS
   RAISERROR ('Insertion Livres',15,9)

go                  Actions associée
```

Ce trigger envoie un message au client lorsqu'on ajoute ou on modifie des données dans *titres.* L'instruction SQL :

```
insert titres (id_titre, titre)
values ('BX100', 'test')
```

provoque le déclenchement du trigger, avec le message :

insert titres (id_titre, titre)

values ('BX100', 'test')

Msg. 50000, niveau 15, état 9

Insertion Livres

Exemple 2 : ce déclencheur INSERT met à jour la colonne *cumulannuel_ventes* de la table *titres* chaque fois qu'une nouvelle ligne est ajoutée à la table *ventes*.

```
CREATE TRIGGER déclench_ins
ON ventes
FOR INSERT
AS
   UPDATE titres
   SET cumulannuel_ventes = cumulannuel_ventes + qt
   FROM inserted
                                Table temporaire = lignes modifiées de VENTES
   WHERE titres.id_titre = inserted.id_titre
Go
```

- Affichage du cumul de ventes par livres, avant la nouvelle vente

```
SELECT id_titre, cumulannuel_ventes
FROM titres
WHERE cumulannuel_ventes is NOT NULL
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

id_titre	*cumulannuel_ventes*
BU1032	*15*
MC3021	*40*
PS2091	*30*

-- Insertion d'une nouvelle vente : qt = 100 sur le livre 'BU1032'

```
INSERT ventes
VALUES ('6380', '6700', '1994', 100, 'Net 60', 'BU1032' )
```

- Provoque la mise à jour automatique de la table TITRES par le trigger, et donc aussi le déclenchement du trigger d'avertissement précédent :

Msg. 50000, niveau 15, état 9
Insertion Livres

--Réaffichage du cumul de ventes par livres, après la vente

id_titre	*cumulannuel_ventes*
BU1032	***115***
MC3021	*40*
PS2091	*30*

☺ Avec Management studio

Pour visualiser et mettre à jour un Trigger :
Dans l'explorateur du serveur, ouvrir la base, déployer les tables, puis déclencheur, dans la table qui déclenche.

Exercice 28 : sécurisation des suppressions, avec modification en cascade

Dans la table PILOTE, créer un trigger en suppression, qui met à NULL tous les champs IdPilote de la table AFFECTATION, liés au pilote à supprimer : avec cette méthode, on connaît les vols sans affectation, qui devront être affectés à un autre pilote. Tester le trigger en supprimant un pilote, et en réaffichant la table AFFECTATION.

Les lignes à supprimer sont stockées dans la table temporaire deleted. Pour cet exercice, il faut supprimer la contrainte de clé étrangère sur la colonne IdPilote de la table AFFECTATION : comme les contraintes de clé étrangère sont évaluées avant le lancement des triggers, le lien entre AFFECTATION et PILOTE interdit la suppression des pilotes référencés dans la table AFFECTATION

Exercice 29 : test de cohérence en insertion sur la table VOL

Dans la table VOL, créer un trigger en insertion qui vérifie que la ville d'arrivée est différente de la ville de départ, et que l'heure d'arrivée est postérieure à l'heure de départ. Dans le cas contraire, on annule l'insertion en appelant l'instruction ROLLBACK.

Exercice 29 bis : test de cohérence en insertion sur la table AFFECTATION

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Interdire à un pilote de voler plus de 35 heures par semaine, par un Trigger en insertion sur la table AFFECTATION.

Lors de l'affectation d'un pilote à un vol, on vérifie que la somme des heures de vol hebdomadaire de ce pilote, pour la semaine considérée, ne dépasse pas 35 heures

La somme des heures de vol du pilote pour une semaine représentée par son numéro, est donnée par la procédure Horaire.

Remarques

Le rollback dans un trigger

Le rollback annule les actions du trigger et celles de la requête déclenchante

Si la requête appelante est elle-même dans une transaction, la transaction est annulée

De plus l'ensemble du lot appelant est abandonné, il est donc impossible de gérer l'incident.

Le raise error dans un trigger

Si la gravité est>10, cela provoque une exception dans le lot appelant.

Conclusion.

Compte tenu des complications évoquées ci avant, il paraît souhaitable de ne pas utiliser de triggers dans des contextes où un risque d'erreur existe.

1.8 Les index

Doc. en ligne : « Créer et maintenir des bases de données » : « Index »
Doc. en ligne : « Optimiser la performance des bases de donnée »
Aide Transact SQL : CREATE INDEX

Présentation simplifiée et conseils

Les index permettent d'optimiser le fonctionnement d'une base de données : ils accélèrent les opérations SELECT, INSERT, UPDATE et DELETE en fournissant un accès direct sur certaines colonnes ou ensembles de colonnes.

Un index ordonné (option CLUSTERED) fournit un accès direct aux lignes physiques de la table, sur le disque. La création d'un index ordonné modifie l'ordre physique des lignes : il faut donc le créer avant les index non ordonnés, et il ne peut y avoir qu'un index ordonné par table (souvent sur la clé primaire).

Un index non ordonné(sans l'option CLUSTERED) est un index qui s'appuie sur l'ordre logique de la table, en mémoire. Une table peut contenir plusieurs index non ordonnés. Employez un index non ordonné pour :

- des colonnes qui contiennent un nombre élevé de valeurs uniques;
- des requêtes qui renvoient de petits ensembles de résultats

· La recherche de données au moyen d'un index ordonné est presque toujours plus rapide qu'avec un index non ordonné. L'index ordonné est particulièrement utile lorsque l'on veut extraire en une seule instruction plusieurs lignes qui possèdent des clés contiguës : l'index garantit que ces lignes seront physiquement adjacentes.

- Pour écrire un index efficace, il faut visualiser la « sélectivité » des colonnes = le nombre de valeurs uniques par rapport au nombre de lignes de la table, données par l'instruction SQL :

SELECT COUNT (DISTINCT *nom_de_colonne***) FROM** *nom_de_table*

Conseils pour une table de 10 000 lignes:

Nombre de valeurs uniques	Choix de l'index
5000	Index non ordonné
20	Index ordonné
3	Pas d'index

Index simple : sur la colonne id_d'auteur de la table auteurs.

```
CREATE INDEX index_id_auteur
        ON auteurs (id_auteur)
```

Index ordonné unique : sur la colonne id_d'auteur de la table auteurs pour assurer l'unicité des données. L'option CLUSTERED étant spécifiée, cet index classera physiquement les données sur le disque.

```
CREATE UNIQUE CLUSTERED INDEX index_id_auteur
        ON auteurs (id_auteur)
```

Index composé simple : crée un index sur les colonnes id_d'auteur et id_titre de la table titreauteur.

```
CREATE INDEX ind1
        ON titreauteur (id_auteur, id_titre)
```

Exercice 30 : optimisation de la base « Compagnie aérienne »

Créer un Index non ordonné composé sur les champs NomPilote et PrenomPilote de la table Pilote.

Vérifier l'existence de l'index sous Microsoft SQL Management Studio : sélectionner la table, puis index.

1.9 Récapitulation

Reprendre tous les exercices et compléter le script de création de la base, dans l'ordre :

- définition des types utilisateurs
- création des tables, en commençant par les tables indépendantes
- création des vues
- création des procédures stockées
- définition des permissions sur les tables, les vues, les procédures stockées
- création des triggers

- création des index

Ecrire un script d'initialisation complète des tables (INSERT…)

Ecrire un script de destruction complète de la base, dans l'ordre inverse de la création :

- suppression des index, triggers, procédures stockées
- suppression des vues
- suppression des tables, en commençant par les tables avec des clés étrangères
- suppression des types utilisateurs

☺ Avec SQL Management studio

Conservez votre base et créez une autre base vide, que vous allez gérer entièrement sous Enterprise Manager

- **Types de données utilisateur**: bouton droit => « Nouveau type de données défini par l'utilisateur »

- **Tables** : sélectionnez l'objet « Tables » : bouton droit => « Nouvelle Table »

  L'utilisation est assez intuitive : pour définir les contraintes de clé primaire => cliquez sur la clé jaune dans la barre d'outil (comme sous ACCESS)

- **Vues** : sélectionnez l'objet « Vues » : bouton droit => « Nouvelle Vue ».

  Vous pouvez définir votre Vue en langage SQL, ou avec un outil graphique comparable à celui d'ACCESS et de VISUAL BASIC.

- **Procédures stockées**: sélectionnez l'objet « Procédures Stockées » : bouton droit => « Nouvelle Procédure Stockée ».

  Pas d'aide graphique : il faut l'écrire en SQL dans la fenêtre !

- **Triggers** : sélectionnez une à une les tables concernées : bouton droit « Modifier une table », puis bouton « Déclencheurs » dans la barre d'outil de la fenêtre « Créer la table… »

Demander une création automatique de scripts de création et de destruction, et comparez les scripts générés aux scripts que vous venez d'écrire

Sélectionner votre base : avec le bouton droit, choisir « Toutes Tâches », « Générer des scripts SQL »

A titre d'exemple : lire le script de création de la base PUBLI utilisé au début de ce cours (Extrait)

Faire des recherches documentaires sur les syntaxes qui n'ont pas été vues dans la construction guidée. Ce script peut aussi vous servir d'exemples pour la plupart des syntaxes.

```
--    InstPubli.SQL   2001
-
GO
set nocount    on
set dateformat mdy

USE master
GO
declare @dttm varchar(55)
select  @dttm=convert(varchar,getdate(),113)
raiserror('Début de InstPubli.SQL à %s ....',1,1,@dttm) with nowait
if exists (select * from sysdatabases where name='publi')
  DROP database publi
GO

CHECKPOINT
GO
CREATE DATABASE publi
   on master = 3
GO
CHECKPOINT
GO
USE publi
GO
if db_name() = 'publi'
   raiserror('Base de données ''publi'' créée et contexte actuellement
utilisé.',1,1) else
   raiserror('Erreur dans InstPubli.SQL, ''USE publi'' a échoué! Arrêt immédiat du
            SPID.' ,22,127) with log

go

execute sp_dboption 'publi' ,'trunc. log on chkpt.' ,'true'

EXECUTE  sp_addtype  id,  'varchar(11)',  'NOT
NULL'  EXECUTE  sp_addtype  idt,  'varchar(6)',
'NOT   NULL'   EXECUTE   sp_addtype   empid,
'char(9)', 'NOT NULL' go

raiserror('Etape de création de la section de table ....',1,1)
GO

CREATE TABLE auteurs
(
      id_auteur   id
            CHECK (id_auteur like '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]')
            CONSTRAINT UPKCL_auidind PRIMARY KEY CLUSTERED,
      nom_auteur        varchar(40)     NOT NULL,
      pn_auteur        varchar(20)     NOT NULL,
      téléphone   char(12)        NOT NULL
            DEFAULT ('INCONNU'),
      adresse varchar(40)     NULL,
      ville    varchar(20)     NULL,
      pays   char(2) NULL,
      code_postal     char(5) NULL
            CHECK (code_postal like '[0-9][0-9][0-9][0-9][0-9]'),
      contrat        bit     NOT NULL
)
go

CREATE TABLE éditeurs
(
      id_éditeur  char(4) NOT NULL
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
            CONSTRAINT UPKCL_pubind PRIMARY KEY CLUSTERED
            CHECK (id_éditeur in ('1389', '0736', '0877', '1622', '1756')
                  OR id_éditeur like '99[0-9][0-9]'),
      nom_éditeur       varchar(40)     NULL,
      ville    varchar(20)     NULL,
      région   char(2) NULL,
      pays varchar(30)     NULL
            DEFAULT('FR')
)
go

CREATE TABLE titres
(
      id_titre        idt
            CONSTRAINT UPKCL_titleidind PRIMARY KEY CLUSTERED,
      titre   varchar(80)     NOT NULL,
      type    char(12)        NOT NULL
            DEFAULT ('SANS TITRE'),
      id_éditeur  char(4) NULL
            REFERENCES éditeurs(id_éditeur),
      prix   money   NULL,
      avance money   NULL,
      droits int     NULL,
      cumulannuel_ventes       int     NULL,
      notes   varchar(200)    NULL,
      datepub datetime        NOT NULL
            DEFAULT (getdate())
)
go

CREATE TABLE titreauteur
(
      id_auteur   id
            REFERENCES auteurs(id_auteur),
      id_titre        idt
            REFERENCES titres(id_titre),
      cmd_auteur  tinyint NULL,
      droits_pourcent      int     NULL,
      CONSTRAINT UPKCL_taind PRIMARY KEY CLUSTERED(id_auteur, id_titre)
)
go


CREATE TABLE magasins
(
      id_mag char(4) NOT NULL
            CONSTRAINT UPK_storeid PRIMARY KEY CLUSTERED,
      nom_mag       varchar(40)     NULL,
      adresse_mag    varchar(40)     NULL,
      ville    varchar(20)     NULL,
      pays   char(2) NULL,
      code_postal     char(5) NULL
)
go


CREATE TABLE ventes
(
      id_mag   char(4) NOT NULL
               REFERENCES magasins(id_mag),
      num_cmd   varchar(20)     NOT NULL,
      date_cmd  datetime        NOT NULL,
      qt       smallint        NOT NULL,

      id_titre  idt
               REFERENCES titres(id_titre),
      CONSTRAINT UPKCL_sales PRIMARY KEY CLUSTERED (id_mag, num_cmd, id_titre)
)
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
go
CREATE TABLE droits_prévus
(
	id_titre	idt
		REFERENCES titres(id_titre),
	minimum int	NULL,
	maximum int	NULL,
	droits int	NULL
)
go

CREATE TABLE remises
(
	typeremise	varchar(40)	NOT NULL,
	id_mag char(4) NULL
		REFERENCES magasins(id_mag),
	qtémin  smallint	NULL,
	qtémax smallint	NULL,
	remise dec(4,2) NOT NULL
)
go

CREATE TABLE emplois
(
	id_emploi  smallint
		IDENTITY(1,1)
		PRIMARY KEY CLUSTERED,
	desc_emploi	varchar(50)	NOT NULL
		DEFAULT 'Nouveau poste - pas de dénomination officielle',
	niv_min tinyint NOT NULL
		CHECK (niv_min >= 10),
	niv_max tinyint NOT NULL
		CHECK (niv_max <= 250)
)
go

CREATE TABLE pub_info
(
	pub_id char(4) NOT NULL
		REFERENCES éditeurs(id_éditeur)
		CONSTRAINT UPKCL_pubinfo PRIMARY KEY CLUSTERED,
	logo	image	NULL,
	pr_info text  NULL
)
go




CREATE TABLE employé
(
	id_employé  empid
		CONSTRAINT PK_id_employé PRIMARY KEY NONCLUSTERED
		CONSTRAINT CK_id_employé CHECK (id_employé LIKE
			'[A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9][0-9][FM]' or
			id_employé LIKE '[A-Z]-[A-Z][1-9][0-9][0-9][0-9][0-9][FM]'),

	pn_employé	varchar(20)	NOT NULL,
	init_centrale	char(1) NULL,
	nom_employé	varchar(30)	NOT NULL,
	id_emploi  smallint	NOT NULL
		DEFAULT 1

		REFERENCES emplois(id_emploi),
	position_employé tinyint
		DEFAULT 10,
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
       id_éditeur  char(4) NOT NULL
              DEFAULT ('9952')
              REFERENCES éditeurs(id_éditeur),

       date_embauche       datetime        NOT NULL
              DEFAULT (getdate())
)
go

raiserror('Etape de création de la section de trigger ....',1,1)
GO

CREATE TRIGGER employé_insupd
ON employé
FOR INSERT, UPDATE
AS
--Get the range of level for this job type from the jobs table.
DECLARE @niv_min tinyint,
       @niv_max tinyint,
       @niv_emploi tinyint,
       @id_emploi smallint
        SELECT @niv_min =
                 niv_min,
    @niv_max = niv_max,
       @niv_emploi = i.position_employé,
       @id_emploi = i.id_emploi
FROM employé e, emplois j, inserted i
WHERE e.id_employé = i.id_employé AND i.id_emploi =
j.id_emploi IF (@id_emploi = 1) and (@niv_emploi <> 10)
BEGIN

       RAISERROR ('L''identificateur d''emploi 1 attend le niveau par défaut
       10.',16,-1) ROLLBACK TRANSACTION
END
ELSE
IF NOT (@niv_emploi BETWEEN @niv_min AND @niv_max)
BEGIN
       RAISERROR ('Le niveau de id_emploi:%d doit se situer entre %d et
              %d.', 16, -1, @id_emploi, @niv_min, @niv_max)
       ROLLBACK TRANSACTION
END
go


raiserror('Etape d''insertion des auteurs ....',1,1)
GO

INSERT auteurs
       VALUES('807-91-6654','Bec','Arthur','22.47.87.11','4, chemin de la Tour de
       Campel', 'Bruges','BE','05006',1)
go

raiserror('Etape d''insertion des éditeurs ....',1,1)

GO
INSERT éditeurs VALUES('9901', 'GGG&G', 'Munich', NULL, 'GER')


INSERT éditeurs VALUES('9999', 'Editions Lucerne', 'Paris', NULL, 'FR')
go

raiserror('Etape d''insertion des pub_info ....',1,1)

GO
INSERT  pub_info  VALUES('9901',  0xFFFFFFFF,  'Aucune  information
actuellement')  INSERT  pub_info  VALUES('9999',  0xFFFFFFFF,  'Aucune
information actuellement') go

raiserror('Etape d''insertion des titres ....',1,1)

GO
INSERT titres VALUES('PC8888',"Les secrets de la Silicon
```

```
Valley",'informatique','1389',$136.00,$54000.00,10,4095,"Deux femmes courageuses
dévoilent tous les scandales qui jonchent l'irrésistible ascension des pionniers de
l'informatique. Matériel et logiciel : personne n'est épargné.",'12/06/85' ) go

raiserror('Etape d''insertion de titreauteur ....',1,1)

GO
INSERT      titreauteur      VALUES('172-32-1176',
'PS3333', 1, 100) INSERT titreauteur VALUES('213-
46-8915', 'BU1032', 2, 40) go

raiserror('Etape d''insertion des magasins ....',1,1)

GO
INSERT magasins VALUES('7066','Librairie spécialisée','567, Av. de la
Victoire','Paris','FR','75016')
go

raiserror('Etape d''insertion des ventes ....',1,1)

GO
INSERT ventes VALUES('7066', 'QA7442.3', '11/09/94', 75,
'Comptant','PS2091') go

raiserror('Etape d''insertion de droits_prévus ....',1,1)

GO
INSERT  droits_prévus  VALUES('BU2075',  1001,
3000,      12)      INSERT      droits_prévus
VALUES('BU2075', 3001, 5000, 14) go

raiserror('Etape d''insertion des remises ....',1,1)

GO
INSERT remises VALUES('Remise volume', NULL, 100,
1000, 6.7) INSERT remises VALUES('Remise client',
'8042', NULL, NULL, 5.0) go

raiserror('Etape d''insertion des emplois ....',1,1)

GO
INSERT emplois VALUES ('Directeur des opérations', 75, 150)
INSERT emplois VALUES ('Rédacteur', 25, 100)
go

raiserror('Etape d''insertion des employé ....',1,1)

GO
INSERT employé VALUES ('M-P91209M', 'Manuel', '', 'Pereira', 8, 101, '9999',
'09/01/89') go

raiserror('Etape de création de la section d''index ....',1,1) with nowait

GO
CREATE CLUSTERED INDEX employé_ind ON employé(nom_employé, pn_employé,
init_centrale) go

CREATE NONCLUSTERED INDEX aunmind ON auteurs (nom_auteur, pn_auteur)

go
CREATE NONCLUSTERED INDEX titleidind ON ventes (id_titre)
go
CREATE NONCLUSTERED INDEX titleind ON titres (titre)
go
CREATE NONCLUSTERED INDEX auidind ON titreauteur (id_auteur)
go
CREATE NONCLUSTERED INDEX titleidind ON titreauteur (id_titre)
go
CREATE NONCLUSTERED INDEX titleidind ON droits_prévus (id_titre)
go

raiserror('Etape de création de la section d''view ....',1,1) with nowait
GO
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
CREATE VIEW vuetitre
AS
SELECT titre, cmd_auteur, nom_auteur, prix, cumulannuel_ventes,
id_éditeur FROM auteurs, titres, titreauteur
WHERE auteurs.id_auteur =
       titreauteur.id_auteur AND
       titres.id_titre = titreauteur.id_titre
go

raiserror('Etape de création de la section de procedure ....',1,1) with nowait

GO

CREATE PROCEDURE pardroits @pourcentage int
AS
SELECT id_auteur FROM titreauteur
WHERE titreauteur.droits_pourcent =
@pourcentage go

raiserror('(Ensuite, premier octroi incorporé dans la section de proc.)',1,1)

GRANT EXECUTE ON pardroits TO public
go

CREATE PROCEDURE reptq1 AS
SELECT id_éditeur, id_titre, prix, datepub
FROM titres
WHERE prix is NOT NULL
ORDER BY id_éditeur
COMPUTE avg(prix) BY id_éditeur
COMPUTE avg(prix)
go

GRANT EXECUTE ON reptq1 TO public
go

CREATE PROCEDURE reptq2 AS
SELECT type, id_éditeur, titres.id_titre, cmd_auteur,
       Name = substring (nom_auteur, 1,15),
cumulannuel_ventes FROM titres, auteurs, titreauteur
WHERE titres.id_titre = titreauteur.id_titre AND auteurs.id_auteur =
       titreauteur.id_auteur AND id_éditeur is NOT NULL
ORDER BY id_éditeur, type
COMPUTE avg(cumulannuel_ventes) BY id_éditeur,
type COMPUTE avg(cumulannuel_ventes) BY
id_éditeur go

GRANT EXECUTE ON reptq2 TO public
go

CREATE PROCEDURE reptq3 @limite_inférieure money, @limite_supérieure money,
@type char(12)
AS
SELECT id_éditeur, type, id_titre, prix
FROM titres
WHERE prix >@limite_inférieure AND prix <@limite_supérieure AND type = @type OR type
LIKE '%cui%'
ORDER BY id_éditeur, type
COMPUTE count(id_titre) BY id_éditeur, type
go

GRANT EXECUTE ON reptq3 TO public

go
GRANT CREATE PROCEDURE TO public
go

raiserror('Etape de la section de GUEST ....',1,1)

GO

EXECUTE sp_adduser guest
go
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
GRANT ALL ON éditeurs TO guest
… idem sur toutes les tables…

GRANT EXECUTE ON pardroits TO guest
GRANT CREATE TABLE TO guest
GRANT CREATE VIEW TO guest
GRANT CREATE RULE TO guest
GRANT CREATE DEFAULT TO guest
GRANT CREATE PROCEDURE TO guest
go

UPDATE STATISTICS éditeurs
... idem sur toutes les tables…
GO
CHECKPO
INT go
declare @dttm varchar(55)
select  @dttm=convert(varchar,getdate(),113)
raiserror('Fin de InstPubli.SQL à %s ....',1,1,@dttm) with nowait
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Exercices complémentaires sur la base "Compagnie aérienne"

Pour faire cette suite d'exercices, commencer par ajouter une colonne Ville à la table Pilote : ville où réside le pilote. Remplir cette colonne avec un jeu de test, qui comprend certaines des villes où sont localisés des avions (corrigés en annexe)

1 Noms des pilotes qui habitent dans la ville de localisation d'un A320 (dans une ville desservie par un aéroport où sont basés des A320)

2 Noms des pilotes planifiés pour des vols sur A320

3 Noms des pilotes planifiés sur A320, qui habitent dans la ville de localisation d'un A320

4 Noms des pilotes planifiés sur A320, qui n'habitent pas la ville de localisation d'un A320

5 Noms des pilotes planifiés sur A320 ou qui habitent la ville de localisation d'un A320

6 Pour chaque ville desservie, donner la moyenne, le minimum et le maximum des capacités des avions qui sont localisés dans un aéroport desservant cette ville

7 Même question, limitée aux villes où sont localisés plus de deux avions

8 Afficher les noms des pilotes qui ne pilotent que des A320

9 Afficher les noms des pilotes qui pilotent tous les A320 de la compagnie

2. LES TRANSACTIONS

Doc. en ligne : « Accéder aux données et les modifier » : « Transactions »
SQL2 : chap. 13, « La gestion des transactions »

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Principes

Une transaction est une unité logique de travail, un ensemble d'instructions d'interrogation ou de mise à jour des données que SQL traite comme une seule action indivisible : soit toutes les instructions comprises dans la transaction sont traitées, soit aucune.

Ce mécanisme garantit la fiabilité des accès au SGBD et la cohérence des données : il pare à toute interruption non désirée d'une séquence d'instructions (arrêt programme, panne machine ou volonté d'annuler les dernières opérations).

Une unité logique de travail doit posséder quatre propriétés appelées propriétés **ACID** (Atomicité, Cohérence, Isolation et Durabilité), pour être considérée comme une transaction :

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

Soit une transaction bancaire qui débite un compte A et crédite un compte B : sans le mécanisme de transaction, si une erreur survenait entre le débit et le crédit, la somme versée serait perdue pour tout le monde. Le mécanisme de transaction va lier les deux opérations : en cas de crash du système central ou de coupure du réseau entre l'agence bancaire et le central, les deux opérations sont annulées.

- Cohérence: lorsqu'elle est terminée, une transaction doit laisser les données dans un état cohérent.

C'est une conséquence de l'atomicité : si l'on arrive sans erreur à la fin de la transaction, les comptes débiteur et créditeur sont actualisés par une opération indivisible ; si une erreur survient pendant la transaction, les deux opérations sont annulées.Dans les deux cas, les deux soldes demeurent cohérents.

Remarquons que le mécanisme de transaction ne fait pas tout : si le problème survient immédiatement après la fin de la transaction, le virement a eu lieu, et il n'est pas question de le faire une seconde fois ; au contraire, si l'erreur survient juste avant, il faut refaire le virement.

Ce sera donc à l'application de tenir à jour un « journal » des virements, pour savoir dans tous les cas si le virement a eu lieu ou non : pour restituer fidèlement les opérations, le journal doit évidemment être actualisé da nsla transaction.

- Isolation : les modifications effectuées par des transactions concurrentes doivent être isolées transaction par transaction. Une transaction accède aux données soit dans l'état où elles étaient avant d'être modifiées par une transaction concurrente, soit telles qu'elles se présentent après exécution de cette dernière, mais jamais dans un état intermédiaire.

- *Tant qu'une transaction n'est pas finie, on ne peut pas être sûr qu'elle aboutisse sans erreur : un utilisateur qui consulte les deux soldes depuis un autre terminal de gestion, doit donc voir les anciens soldes = les dernières données dont la cohérence est garantie.*

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

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Remarques :

1) Ce type de fonctionnement est possible parce que le S.G.B.D. dispose d'une copie des tables avant modification (souvent appelée SNAPSHOT = l'état instantané). Lors du COMMIT, le SGBDR recopie le fichier SNAPSHOT sur le fichier de bases de donnée, en une seule opération indivisible.

2) Un ROLLBACK automatique est exécuté en cas de panne de la machine.

3) ROLLBACK et COMMIT ne concernent pas les commandes de définition de données : CREATE, DROP, ALTER. Toute modification portant sur la structure de la base de données, est donc irréversible.

4) L'exécution fréquente d'un COMMIT peut éviter la perte d'informations.

Mise en œuvre en Transact SQL

Selon les SGBD, le début d'une transaction est déclenché différemment : sous SQL SERVER, il est marqué par l'instruction :

BEGIN TRANSACTION

La fin d'une transaction est marquée par les instructions **COMMIT** ou **ROLLBACK.**

Les modifications ne sont rendues permanentes que lors de l'instruction **COMMIT** : tant que les changements ne sont pas validés par un **COMMIT,** seul l'utilisateur travaille sur les données qu'il vient de modifier alors que les autres utilisateurs travaillent sur les valeurs avant modification.

L'instruction **ROLLBACK** annule toutes les modifications effectuées depuis le début de la transaction en cours.

En l'absence de transaction explicite marquée par BEGIN TRANSACTION, toute instruction SQL constitue une transaction élémentaire.

Exercice 31 : vérification du mécanisme ACID, observation des verrouillages

Travaillez sur votre base « Compagnie Aerienne » avec l'analyseur de requête : on ouvrira deux sessions par binôme à deux noms différents.

- ***pour que SQL SERVER gère correctement les transactions, commencer par spécifier dans les deux sessions :***

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

1) Créer un nouveau pilote, en terminant la transaction par ROLLBACK. Vérifier que le nouveau pilote n'est pas conservé dans la base (faire un SELECT avant et après le ROLLBACK)

2) Ouvrir une transaction par session. Faire une lecture sur la table Pilote (SELECT) dans chaque session. Que constatez-vous ?

3) Ouvrir une transaction par session. Dans une des sessions, on lit les données (SELECT sur Pilote) ; dans l'autre on essaie de les changer (UPDATE sur Pilote). Que constatez-vous ?

4) Ouvrir une transaction par session. On commence par modifier dans une session et on lit dans l'autre. Que constatez-vous ?

Pour expliquer ce qui a été observé, nous allons détailler les deux utilisations des transactions :

- dans la gestion des erreurs (utilisation du ROLLBACK) ;

- dans la gestion des accès « concurrents » (plusieurs clients accédant aux mêmes tables en lecture ou en écriture, entraînant un verrouillage)

L'utilisation des transactions dans la gestion d'erreur

TRANSACT SQL range un compte-rendu d'erreur, après chaque requête, dans la variable globale : **@@error** . Si l'instruction s'est bien effectuée, **@@error = 0,** sinon il s'agit d'une erreur ou d'un Warning.

Pour écrire une transaction correcte, il faut donc tester **@@error** et faire un **ROLLBACK** dès qu'une erreur est détectée.

Voici un exemple complet de procédure sécurisée :

```
-   ============================================
-   Author: EDIDE PAUL
-   Create date: 11 Juillet 2017
-   Description:faire un retrait d'espéces
--                          Version sql server 2017
-   Tous les paramètres doivent être fournis dans l'appel,
-   mais la valeurs nulles sont vérifiées.
-   Retour
-              -9 : un parmaètre possède une valeur nulle
-              -1  : le compte n'existe pas
-              -2  : le solde n'est pas suffisant
-              -3: l'instruction update pose probléme
-              -4  : le commit pose problème
-         1 :OK
-   ============================================
Create PROCEDURE [dbo].[retrait]
       @NumCompte int,
       @Montant int
AS

declare @soldeAvantOperation int;
declare @codeRet int;
       --hors transaction
       --Vérifier que les parmaètres ne sont pas
       nuls if @NumCompte is Null or @Montant is
       Null
              set @codeRet=-9
       else

       begin
              Begin Transaction
              --vérifer l'existence du compte
              if EXISTS (select * from COMPTE (holdlock,tablockx) where NumCompte =
              @NumCompte) begin
                 SELECT @soldeAvantOperation=solde
                 FROM COMPTE
                 WHERE NumCompte =  @NumCompte

                 if (@soldeAvantOperation > @Montant)
                 begin
                       --OK
                       --coder update
                        UPDATE COMPTE SET Solde = (Solde - @Montant) where NumCompte =
                                          @NumCompte
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
                --vérifier @@ERROR
               if @@ERROR = 0
                 begin
                      Commit transaction
                      if @@ERROR <>0
                      begin
                             rollback transaction
                             set @codeRet=-       --erreur commit
                             4
                      end
                      else
                      begin
                             set @codeRet=1       --OK
                      end
                 end
                 else
                 begin
                        rollback transaction
                        set                  --erreur update
                        @codeRet=-3
                 end
            end
            else
            begin
                  --KO solde insuffisant
                  rollback transaction
                  set @codeRet=-2
            end --fin solde insuffisant
         end
         else
         begin
            --KO compte inexistant
            rollback transaction
            set @codeRet=-1
         end
end -- fin test existance paramètre
return (@codeRet)
```

Remarques :
La procédure commence par vérifier que tous les paramètres sont renseignés. Noter qu'il n'est pas vraiment possible de se protéger contre un paramètre non transmis à l'appel, mais il faut vérifer que les valeurs ne sont pas nulles. Ces contrôles peuvent se faire hors transaction.

Ensuite la transaction débute.

Commencer par les vérifications fonctionnelles, le compte existe, le solde est suffisant. Dans les branches d'échec placer un rollback. Il n'y a ici que des select, il faut cependant les placer dans la transaction pour qu'ils provoquent des verrous.

Derriére chaque requête : tester @@error

Certaines erreurs fatales provoquent une exception et ne sont pas récupérables par le test sur @@ERROR, l'arrêt est alors brutal et le rollback n'est pas fait.

Plus précisement : une erreur de syntaxe sql provoque une erreur à la compilation, une erreur dans un nom de table provoque une erreur sql fatale et une exception chez le client.

Depuis sql server 2017, Microsoft a introduit un mécanisme d'exception. La limitation ci-dessus demeure cependant, les erreurs fatales ne sont pas attrapées.
A l'inverse les erreurs de gravité inférieure à 10 sont ignorées.

L'utilisation d'exception permet d'éviter des tests fastidieux à chaque instruction. Les tests fonctionnels préalables restent cependant nécessaires. Ci-dessous un exemple en utilisant les exceptions :

```
-- Description:    faire un retrait d'espéces
--                        Version sql server 2017
-  Les erreurs prévisibles sont géréees par algo
-  Les erreurs imprévisibles sont gérée par exception

-  Tous les paramètres doivent être fournis dans l'appel,
-  mais la valeurs nulles sont vérifiées.
-  Retour
-             -9 : un parmaètre possède une valeur nulle
-             -1  : le compte n'existe pas
-             -2  : le solde n'est pas suffisant
-             -3: le Update pose probléme !
-             -4  : le Commit pose probléme
-          -5  : un select pose probléme
-              0  : ne devrait pas se produire
-         1 :OK

-  ============================================
Create PROCEDURE [dbo].[retrait2]
@NumCompte int,
@Montant int
AS

declare @soldeAvantOperation int;
declare @codeRet int;
       set @codeRet=0
       --hors transaction
       --Vérifier que les parmaètres ne sont pas nuls
       if @NumCompte is Null or @Montant is Null
              set @codeRet=-9
       else
       begin
              BEGIN TRY
              Begin Transaction
              --vérifer l'existence du compte
              set @codeRet=-5
              if EXISTS (select * from COMPTE with (holdlock,tablock) where
                           NumCompte = @NumCompte)
              begin
                 SELECT @soldeAvantOperation=solde
                 FROM COMPTE
                 WHERE NumCompte =  @NumCompte

                 if (@soldeAvantOperation > @Montant)
                 begin
-                       Les contrôles préliminaires sont OK
-                       Le traitement devrait bien se passer
                       set @codeRet=-3
                      UPDATE COMPTE SET Solde = (Solde - @Montant) where NumCompte =
                       @NumCompte set @codeRet=-4
                       Commit
                       set @CodeRet=1
                 end
                 else
                 begin
                       --KO solde insuffisant
                       rollback transaction
                       set @codeRet=-2
                 end --fin solde insuffisant
              end
              else
              begin
                 --KO compte inexistant
                 rollback transaction
                 set @codeRet=-1
              end
```

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

```
        END TRY
        BEGIN CATCH
              --unexpected error
              print ERROR_MESSAGE()
              rollback transaction
        END CATCH
end -- fin test existence paramètre
return (@codeRet)
```

Exercice 32 : réalisation d'une procédure stockée fiable, avec transaction

On veut gérer par une procédure stockée les départs de la société, et les réaffectations des vols au pilote remplaçant. La procédure doit être fiable : un pilote ne doit être supprimé des listes de la compagnie aérienne, qu'une fois le remplaçant entré dans les listes et tous ses vols réaffectés. Ces trois opérations doivent être indivisibles.

On propose l'interface suivante pour la procédure stockée :

ENTREE

@idAncien : identificateur de l'ancien pilote (OBLIGATOIRE)

@nom : nom du nouveau pilote (OBLIGATOIRE)

@prenom : prenom du nouveau pilote (OBLIGATOIRE)

SORTIE

@idNouveau : identificateur du nouveau pilote

RETOUR

(0) Remplacement OK, ***(1) Erreur : paramètres incorrects,***
(2) Erreur : l'ancien pilote, n'existe pas, ***(3) Erreur système***
La procédure sera livrée avec son programme de test…

Conseils pour une procédure stockée bien conçue

Affecter une valeur de code d'erreur, pour chaque annomlie possible.

Vérifier hors transaction la vraisemblance des paramètres d'entrée.

Dans une transaction.

Vérifier que la mise à jour envisagée est compatible avec l'intégrité de la base

Dans la négative faire un rollback.

Vérifier @@error après chaque reqûete, en cas d'échec faire un rollback.

Une requête select formellement correcte ne peut pas être en erreur.

Seule les requêtes de mise à jour doivent être controlées.

Dans la plupart des cas, ce sont les violations de contraintes qui positionnent @@error.

L'algo de la procédure comprendra des nombreuses branches, seule la branche succès comporte un commit

Vérifier @@error après le commit est probablement inutile avec SQL serveur, mais est indispensable avec les sgbd à évaluation des contraintes différées. Prendre l'habitude de le faire.

Malgré toutes nos précautions, une exception durant l'exécution d'une procédure stockée reste possible, en ce cas la transaction peut rester ouverte. Il faudra en tenir compte au niveau appelant.

L'accès « concurrent » : verrouillages dans les transactions

Un SGBD étant souvent utilisé sur une machine multi-utilisateurs, ou en Client-Serveur à partir de plusieurs clients, on peut imaginer ce qui arriverait si deux transactions pouvaient modifier en même temps la même ligne d'une table !

Reprenons l'exemple du débit/crédit, en supposant que la banque ne tolère aucun découvert. L'application qui effectue le virement doit tester si le compte A est approvisionné, avant de faire le débit.

Si l'on n'utilise pas de transaction, on tombe sur un problème classique en programmation « concurrente » : deux applications veulent débiter le même compte ; elles commencent par tester le solde qui est suffisant pour chacune d'entre elles, et débitent ensuite la somme demandée, en aboutissant à un solde négatif.

Le compte à débiter a un solde initial de 1200 F

Application A : Demande un débit de 1000 F	*Application B : Demande un débit de 500 F*
(1) Test du solde OK : 1200 > 1000	*(2) Test du solde OK : 1200 > 500*
(3) Débit de 1000 F => nouveau solde = 200 F	*(4) Débit de 500 F => solde final = -300 F*

Ce problème fondamental de l'accès concurrentiel aux données est géré au moyen d'un mécanisme de verrouillage, des lignes ou des tables entières *lock*().

Ce mécanisme permet de bloquer l'accès aux lignes ou aux tables concernées soit automatiquement (ORACLE, SQL Server), soit sur commande explicite (DB2, RDB). L'accès à une ligne ou une table verrouillées peut être traité soit par attente du déblocage (mode WAIT) soit par ROLLBACK de la transaction (mode NOWAIT) : le déblocage est effectif dès la fin de la transaction.

Reprise de l'exemple précédent avec transaction et verrouillage de la table Comptes

Application A : Demande un débit de 1000 F	*Application B : Demande un débit de 500 F*
(1) Début transaction avec verrouillage exclusif de la table Comptes (passant)	*(2) Début transaction avec verrouillage exclusif de la table Comptes (bloqué)*
(3) Test du solde OK : 1200 > 1000	*En attente*
(4) Débit de 1000 F => solde = 200 F	*En attente*
(5) Fin transaction => déverrouillage de la table Comptes	*(6) Test du solde : 200 < 500 => débit refusé*
	(7) Fin transaction => déverrouillage de la table Comptes

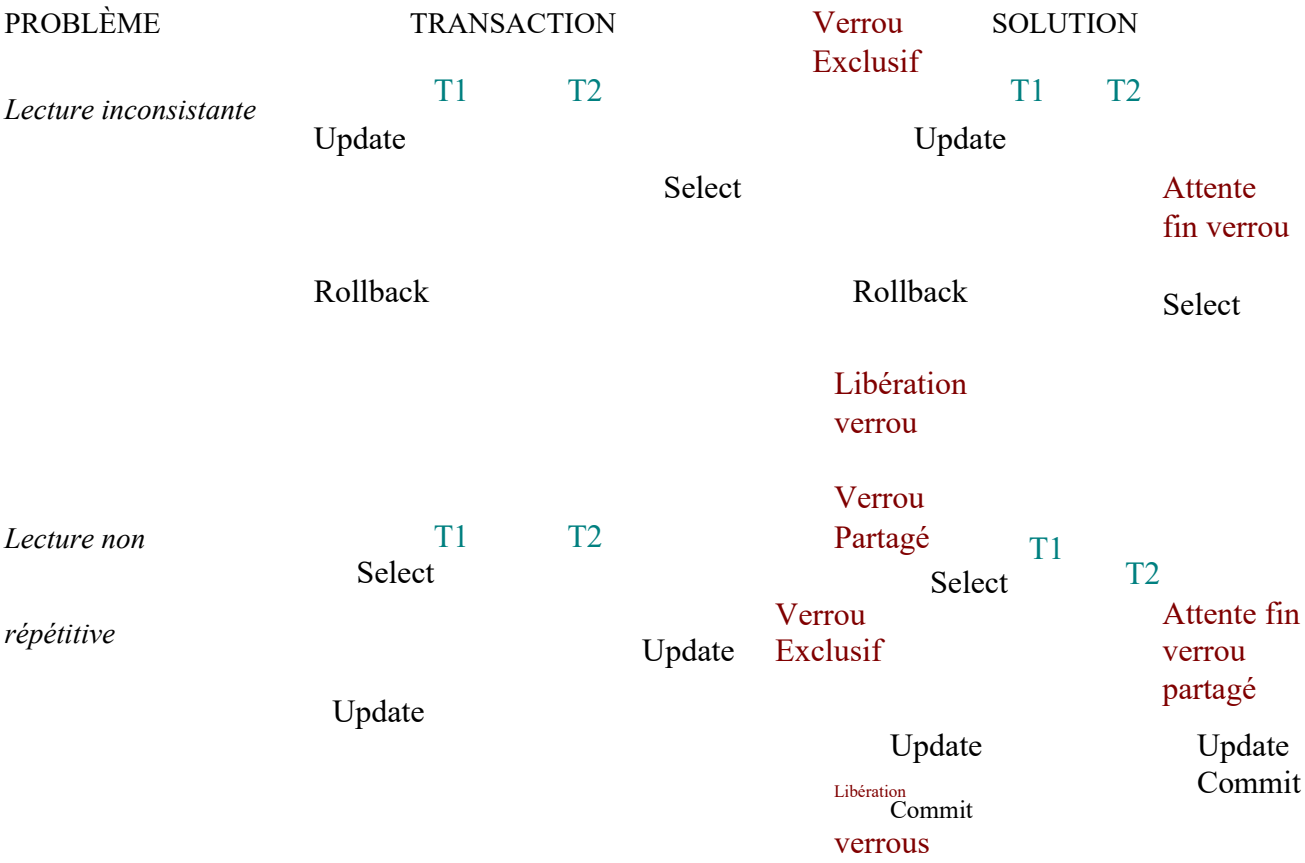
Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Les niveaux d'isolation dans les transactions

La norme SQL2 ne prévoie pas de demande explicite des verrous en début de transaction : les programmeurs risquant d'oublier de réserver les verrous, la norme considère que la prise automatique de verrous par le SGBD est préférable. Le SGBD choisit automatiquement les verrous à poser sur les tables et les lignes en fonction des requêtes reçues :

- SELECT : le SGBD pose un « verrou partagé » , c'est-à-dire un verrou qui autorise les autres lecteurs à consulter la table, mais qui bloque les demandes d'écriture (INSERT, DELETE, UPDATE)

- INSERT, DELETE, UPDATE : le SGBD pose un « verrou exclusif » qui interdit tout autre opération sur l'objet, y compris les lectures



Le SGBD pose des verrous différents selon le « niveau d'isolation des transactions » :

1. **Read Uncommited** : Aucun verrou. On peut lire les mises à jour dans d'autres applications, même si elles n'ont pas encore été validées par un COMMIT. Fortement déconseillé !

2. **Read Commited (défaut)**: verrou exclusif sur les lignes modifiées. Le SGBD cache les modifications en cours aux autres utilisateurs, tant qu'elles n'ont pas été validées par COMMIT. Mais il ne pose pas de verrous sur le SELECT : dans une même transaction, les lectures ne sont pas nécessairement « répétitives », les données pouvant être modifiées par une autre transaction entre des lectures qui se suivent.

3. **Repeatable Read (conseillé) :** corrige le problème précédent des lectures non répétitives, en posant un verrou partagé sur les lignes lues, en plus du verrou exclusif sur les lignes modifiées. Les lignes ne pourront donc pas être modifiées entre le premier SELECT et la fin de la transaction.

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

4. **Serializable :** le SGBD prend un verrou de table sur toute ressource utilisée

En résumé, le SGBD manipule quatre sortes de verrous, selon les opérations et le niveau d'isolation demandés :

	Verrou partagé	Verrou exclusif
Verrou de pages (lignes)	Lecture dans le cas 3	modification dans les cas 2 et 3
Verrou de tables	Lecture dans le cas 4	modification dans le cas 4

Verrou de pages : en pratique, un SGBD ne pose jamais un verrou sur une seule ligne, mais sur la page qui contient cette ligne.

Norme SQL 2 sur les niveaux d'isolation : p. 162

Mise en œuvre pratique en Transact SQL

Par défaut SQL Server travaille dans un niveau intermédiaire entre « Read UnCommited » et « Read Commited » : il pose des verrous exclusifs sur les pages modifiées; il pose des verrous partagés sur les pages lues (comme dans le cas « Repeatable Read ») mais il les libère dès la fin du SELECT. Les verrous partagés sont donc bloquants si une modification est en cours dans une autre transaction, mais ils n'empêchent pas les autres de modifier les données qui viennent d'être lues.

Trois options pour modifier le comportement des verrous : **HOLDLOCK**, **TABLOCK** et **TABLOCKX**

HOLDLOCK maintient le verrou jusqu'au COMMIT (au lieu de le relacher après le SELECT). Il interdit donc toute modification externe, et garantit la cohérence de toutes les lectures

TABLOCK pose un verrou partagé de table, au lieu d'un verrou de page

TABLOCKX pose un verrou exclusif de table

Exemple de transaction bancaire qui garantit la cohérence du solde (abandon si solde insuffisant) et la « répétitivité » des différentes lectures (solde non modifié par une autre transaction entre deux lectures)

-- fixer le mode d'isolation

set transaction isolation level SERIALIZABLE

-- début de la transaction

begin transaction

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

lecture de la table Compte avec verrouillage de la table en mode exclusif : écriture et lecture interdites jusqu'au COMMIT

```
Select solde From Comptes With (HOLDLOCK TABLOCKX) Where num = 1
 -- modification de la table Comptes
Update Comptes Set solde = solde – 500 Where num = 1
-- lecture de vérification, pour trouver le nouveau solde
Select solde From Comptes Where num = 1
 -- fin transaction
commit transaction
```

Attention aux « deadlocks »

L'utilisation des verrous peut conduire à des "deadlock" : situation où deux transactions ou plus peuvent se bloquer mutuellement.

Application A	*Application B*
begin transaction	begin transaction
Select solde From Comptes with (HOLDLOCK) Where num = 1 **verrou partagé, PASSANT**	
	Select solde From Comptes with (HOLDLOCK) Where num = 1 **verrou partagé, PASSANT**
Update Comptes Set solde = solde – 500 Where num = 1	
Demande de verrou exclusif, BLOQUANT (sur verrou partagé de B)	
Update Comptes Set solde = solde – 500 Where num = 1 **Demande de verrou exclusif, BlOQUANT (sur verrou partagé de A) => DEADLOCK**	

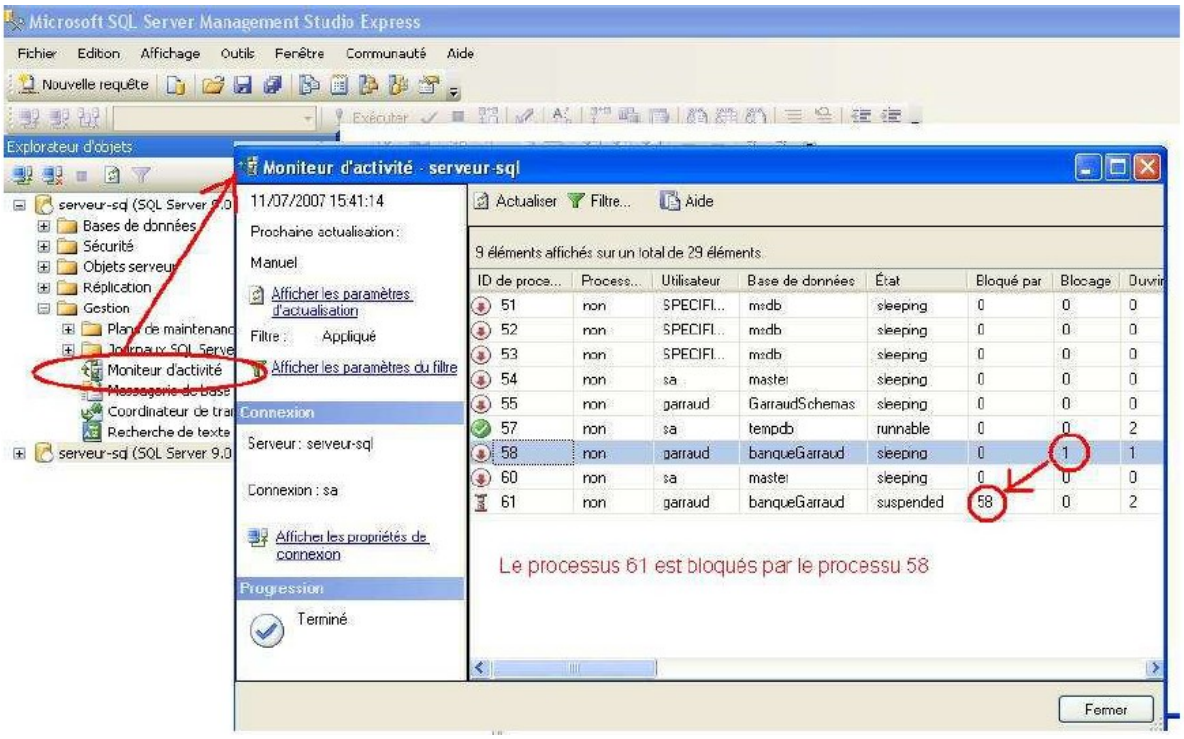
Exercice 33 : provoquer un DEADLOCK

Lancer deux fois le script SQL fourni Deadlock.sql sous deux connexions différentes.

Exécuter uniquement le SELECT sous chacune des connexions. Puis exécuter le UPDATE sous chacune des connexions. Que se passe-t-il ? Comment le SGBD sort-il du deadlock ?

Même exercice en observant les verrous qui sont posés par le SGBD. Sous Management studio, connectez vous sous le compte « Développeur » qui vous donne des droits suffisants pour observer l'état du système. Dans « Gestion / Moniteur d'activité », vous trouverez des informations sur les processus en cours, et les verrous posés sur les différents objets de la base.

Modifier le script avec l'option TABLOCKX qui évite le Deadlock en bloquant le deuxième processus dès le SELECT (=> c'est la solution à retenir)



3. LE MODELE CLIENT/SERVEUR

3.1 Introduction

A la base du dialogue, il y a un besoin du client : le client est l'application qui débute la "conversation", afin d'obtenir de la part du serveur (le programme "répondant") des données ou un résultat.

Le serveur est un programme générique dont la seule fonction est de répondre aux demandes des applications clientes, en fournissant en retour les données ou le résultat. Les demandes des applications sont appelées généralement des requê.tesLe serveur est, par nature, indépendant des applications qui l'interrogent.

Dialogue

Client Serveur

Initie et pilote le dialogue répond au client, participe au dialogue

Est conforme au modèle client-serveur, une application qui fait appel à des services distants par des requêtes, plutôt que par un échange de fichiers. Ce modèle exige que les ordinateurs soient reliés en réseau, et repose sur un protocole qui assure le bon fonctionnement des échanges :

- au niveau du client : émission de requêtes (les appels de service)
- au niveau du serveur : émission des réponses ou résultats

3.2 Les principes de base

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Comment peut-on définir un service?

La notion de service doit être comprise à travers la notion de traitement. Le modèle client-serveur permet de répartir les services utilisés par une application et non l'application elle-même.

Les traitements auxquels font appel les applications, requièrent souvent beaucoup de ressources machine : il est donc préférable, dans un souci de performance et d'efficacité, de les faire résider chacun sur un ordinateur dédié, appelé serveur.

Comment peut-on distinguer le cœur d'une application et les services annexes de celle-ci que l'on doit déporter?

Une application informatique se compose fondamentalement de 3 niveaux :

- l'interface avec l'utilisateur,
- les traitements,
- les données.

Ces niveaux sont eux-mêmes composés de couches :

Interface

Gestion de affichage

Logique de l'affichage

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Traitements	Logique fonctionnelle
	Exécution des procédures
Données	Intégrité des données
	Gestion des données

Quel est le rôle des différentes couches?

La gestion de l'affichage : est assurée par l'environnement d'exploitation (ex: Windows),

La logique de l'affichage : transmet à la gestion de l'affichage les directives de présentation (ex : les feuilles et les contrôles graphiques dans une application Visual Basic sous Windows),

La logique fonctionnelle : représente l'arborescence algorithmique de l'application et aiguille le traitement vers les procédures qui sont déroulées et exécutées dans la couche suivante (ex : les fonctions événementielles en Visual Basic associées aux messages souris, clavier..)

L'exécution des procédures : effectue les traitements de l'application (ex : des ordres SQL lancés par les fonctions événementielles, ou mieux des appels de procédures stockées),

L'intégrité des données : vérifie l'utilisation cohérente des données par les différents processus,

La gestion des données : permet la manipulation des données (sélection ou mise à jour des données).

3.3 Le dialogue entre client et serveur : l'IPC

Le dialogue s'effectue à travers le réseau qui relie le client et le serveur :

Une conversation de type client-serveur est un dialogue inter-processus qui s'appuie de part et d'autre sur 2 interfaces de niveau système. Ce dialogue s'appelle un IPC (Inter Processus Communication) ou encore Middleware.

L'IPC permet l'établissement et le maintien du dialogue. C'est un ensemble de couches logicielles qui :

- prend en compte les requêtes de l'application cliente,
- les transmet de façon transparente à travers le réseau,
- retourne les données ou les résultats à l'application cliente.

L'IPC est constitué par :

- des API (Application Programming Interface) ou interface de programmation au niveau applicatif : l'API est une interface entre un programme et un système. L'API encapsule des fonctions qui vont permettre à l'application de faire appel aux services proposés par le serveur (ex : les WinSockets pour faire de la communication IP sous Windows) ;

- des FAP (Format And Protocols) ou protocoles de communication et format des données : ce sont des procédures chargées d'assurer la présentation, les conversions de formats et de codification.

Application

Interface de programmation (API)

Protocole de communication et format (FAP)

Protocole de transport lié au réseau

L'application ne voit que l'API, l'API n'est en contact qu'avec la FAP et la FAP fait la liaison avec les couches réseau.

L'IPC est l'élément fondamental du fonctionnement client-serveur. Deux approches d'IPC existent :

- Les RPC (Remote Procédure Call ou appels de procédure à distance) pour les dialogues sans connexion ;
- Par Messages : il faut alors établir une session de communication entre Client et Serveur, et gérer cette session.

3.4 Les mécanismes utilisés par les interfaces de communication

Les RPC

Il s'agit de communication par échange synchrone : chacun reste en attente de l'autre et ne peut rien faire pendant ce temps. La fiabilité est médiocre : en cas d'incident, il n'y a pas de reprise. Mais la mise en œuvre est simple.

Client	Serveur
Appel de la procédure serveur et Attente de la réponse	
	prise en compte de la requête réveil du serveur exécution de la procédure émission d'un message réponse unique
Réception du résultat et reprise de l'exécution	

Les messages

Le Client et le Serveur effectuent un échange de messages. Le dialogue est asynchrone. C'est la demande de connexion du programme client qui est le point de départ du dialogue. Ce mode de communication représente un "investissement" important, et nécessite un bon niveau d'expertise. Il n'est justifié que pour des échanges intensifs.

Client	Serveur
Demande de connexion: envoi d'un message comportant l'identification du client	

Client	Serveur
	prise en compte de la demande et création d'un contexte
← acceptation de la connexion	
Envoi d'une requête	
	exécution de la requête
Réception du résultat	
Synchronisation	
Envoi d'une requête	
	exécution de la requête
Réception du résultat	
Synchronisation	

Schéma en couches d'un échange client-serveur :

Application	API
Présentation	FAP
Session	FAP
Transport	TCP
Réseau	IP
Liaison	Ethernet, Token Ring
Physique	Coaxial…

Le **protocole de communication** est contenu dans la couche FAP de l'IPC. Il gère l'ordonnancement de l'échange :

- ouvrir une session,
- envoyer une requête,
- faire remonter les résultats.

Le **protocole de transport** prend les messages émis par les applications pour les insérer dans une trame, qui va circuler sur le réseau. Ex : TCP/IP

3.5 Les différentes mises en oeuvre du modèle

Le type de relation Client-Serveur est déterminé par l'activité du site serveur, c'est à dire le service "déporté" du cœur de l'application. Selon la répartition évoquée entre Données, Traitements et Présentation, on peut distinguer plusieurs types de liaison Client-Serveur :

- le client-serveur de présentation,
- le client-serveur de données,
- le client-serveur de procédures.

Ces types ne sont pas exclusifs les uns des autres.

Le Client-Serveur de présentation

C'est le module Gestion de l'affichage, déporté sur un système spécialisé, qui est le service. Ceci suppose de pouvoir séparer la gestion de l'affichage et la logique de l'affichage : par exemple sous X-Windows. Windows est un système monolithique qui ne permet pas ce découpage.

Client	Serveur
logique de l'affichage	gestion de l'affichage
logique fonctionnelle	
Exécution des procédures	
intégrité des données	
gestion des données	

Le client-Serveur de données

C'est le module de gestion des données qui est déporté sur un serveur dédié ; on peut également déporter le module d'intégrité des données. Ex : les SGBDR.

Client	serveur
Gestion de l'affichage	intégrité des données
Logique de l'affichage	gestion des données
Logique fonctionnelle	
Exécution des procédures	

Le Client-Serveur de procédure

Ce type requiert plus de savoir faire pour sa mise en œuvre : il s'agit là de déporter vers le à serveur une partie des traitements, sans toucher la la localisation de la Gestion des données et de présentation.

La logique fonctionnelle de l'application réside toujours sur la station cliente. Cette partie logique effectue des appels ponctuels à des traitements ou services de la station serveur. Ces services sont soit des applicatifs écrits dans des langages classiques, soit des procédures cataloguées toutes prêtes, écrites en SQL (ex : « procédures stockées » appelées en Visual Basic).

Client	Serveur
Gestion de l'affichage	exécution des procédures
Logique de l'affichage	
Logique fonctionnelle	
Intégrité des données	
Gestion des données	

3.6 Conclusion

Le Client-Serveur représente la solution pour aller vers une utilisation uniforme des moyens informatiques.

Paul EDIDE
Ingénieur Système d'Informations et Décisionnels
paul.edide@gmail.com

Si le Client-Serveur ne permet pas de faire baisser les couts de projets, les avantages techniques apportés par le modèle client-serveur se traduisent par des atouts "stratégiques":

- une approche "évolutive" de la rénovation des systèmes d'information,
- un meilleur parti des ressources matérielles éparpillées dans l'entreprise,
- la prise en compte des nouvelles technologies.

Pour une présentation détaillée des techniques Client/Serveur, lire la documentation « Le Client/Serveur » de Marc Fayolle.

ANNEXES

GLOSSAIRE

Alias : Nom temporaire donné à une table ou à un attribut du résultat d'une requête. Un alias dure le temps de l'exécution de la requête, et n'a de sens que dans la requête où il est défini.

Attribut (*attribute*) : Chaque colonne d'une relation. Les attributs d'une relation doivent être uniques et sont normalement non ordonnés. Un attribut est toujours au moins caractérisé par son type et sa longueur. L'ensemble des valeurs que peut prendre un attribut est le domaine de l'attribut. Le degré d'une relation est le nombre d'attributs de la relation.

Base de données(*Data Base*) : Ensemble logique de données constitué de tables ou de fichiers avec leur index, et d'un dictionnaire centralisé permettant une gestion efficace des données.

Base de données relationnelle (*Relational Data Base*) : Base de données perçue par ses utilisateurs comme une collection de tables. Ces tables sont manipulées par un langage d'accès de quatrième génération, ou en SQL. L'utilisateur précise ce qu'il cherche dans la base et non comment il faut l'obtenir. Dans une base de données relationnelle, le SGBDR choisit le meilleur chemin d'accès pour optimiser l'utilisation des ressources (mémoire, disque, temps).

Champ (*Field*) : Synonyme d'attribut mais s'applique plutôt pour un enregistrement dans un système de gestion de fichiers classique (aussi appelé zone).

Clause (*Clause*) : Mot réservé du langage SQL utilisé pour spécifier ce qu'une commande doit faire, et souvent suivi d'une valeur. Dans la commande :
'SELECT DISTINCT clt_nom FROM clients'
les mots SELECT, DISTINCT et FROM sont des clauses.

Clé(*Key*) : Attribut ou combinaison d'attributs utilisé pour accéder à une table sur un critère particulier. Un index est construit sur une clé.

Clé étrangère(*Foreign Key*) : Attribut ou groupe d'attributs d'une relation, clé primaire dans une autre relation, et constituant un lien privilégié entre relations.

Clé primaire (*Primary Key*) : Clé donnant accès de façon univoque, pour chacune de ses valeurs, à un tuple de la table, (on dit parfois aussi « identifiant »).

Clé secondaire (*Secondary Key*) : Clé donnant accès pour chacune de ses valeurs, à un ensemble de tuples de la table, ceux-ci sont alors accédés en séquence.

Cluster (*Cluster*) : Voir Groupement (de tables).

Colonne (*Column*) : Voir attribut.

Corrélée (sous-requê te)(*Correlated query*) : Sous-requête exécutée de façon répétitive, pour chaque tuple sélectionné par la requête extérieure. A chaque exécution de la sous-requête, le(s) tuple(s) traité(s) dépend(ent) de la valeur de un ou plusieurs attributs du tuple sélectionné dans la requête extérieure.

Curseur (*Cursor*) : En SQL intégré, sorte de pointeur vers un tuple particulier de la table temporaire contenant le résultat d'une requête. Le curseur fait le lien entre SQL, langage ensembliste, et le langage hôte (C, COBOL,...) qui ne traite qu'un tuple à la fois.

Dictionnaire de données(*Data Dictionary*) : Ensemble de tables et de vues contenant toutes les informations sur la structure d'une base de données. Le dictionnaire est utilisé à chaque requête pour vérifier la légitimité et la cohérence de la commande.

Domaine (*Domain*) : Ensemble des valeurs que peut prendre un attribut.

Fonction de groupe (*Group Function*) : Fonction mathématique s'appliquant sur toutes les valeurs d'un attribut ou d'une expression combinant plusieurs attributs, dans les tuples des groupes générés par une requête. La fonction calcule une seule valeur pour chaque groupe.

Groupement (de tables) (*cluster*) : groupe de deux ou plusieurs tables ayant des valeurs d'attributs identiques, et partageant le même espace disque. Utilisé pour accélérer les requêtes multi-tables et pour diminuer l'espace occupé.

Index (*Index*) : Objet d'une base de données permettant de trouver un tuple ou un ensemble de tuples dans une table sans examiner la table entièrement. L'index possède une entrée pour chaque valeur de la clé associée, couplée à un (des) pointeur(s) vers le(s) tuple(s) qui ont cette valeur.

Intégrité d'entité (ou de relatio n)(*Integrity*) : Règle obligatoire dans une base de données relationnelle, spécifiant que les valeurs d'une clé primaire doivent être uniques et non NULL.

Intégrité de référence (ou référentielle)(*Referential Integrity*) : Règle de cohérence hautement souhaitée dans une base de données relationnelle, obligeant à vérifier la correspondance entre la valeur d'une clé étrangère et celle de la clé primaire associée. Cette règle doit être vérifiée à chaque mise à jour de la base.

Jointure (*join*) : Opération relationnelle couramment utilisée en SQL pour retrouver simultanément des parties de tuples dans plusieurs tables. On associe à cette jointure une condition (dite de jointure) spécifiant quels tuples des différentes tables doivent être associés.

Jointure 'externe' : Voir 'outer join'.

Ligne (d'une table) (*row*) : Voir tuple.

'Outer join' : Jointure incluant tous les tuples d'une table, même si l'autre table ne comprend aucun tuple satisfaisant la condition de jointure. Lorsque la condition de jointure ne peut être respectée, des valeurs NULL sont insérées pour les attributs manquants.

NULL : Valeur particulière d'un attribut, qui signifie "n'a pas de valeur" ou "valeur non applicable". Dans le premier cas, on ne connaît pas la valeur au moment de l'encodage, et dans le second, l'attribut n'a pas de sens pour le tuple concerné. Une requête peut sélectionner ou ignorer les tuples avec valeurs NULL. La valeur NULL se distingue de "blanc" et zéro.

Propriété(*Property*) : Voir attribut. S'emploie en conception de bases de données, notamment dans les méthodes MERISE, AXIAL, pour désigner les divers éléments d'une entité (ou d'une relation).

Privilège (*Privilege*) : Chacune des opérations qu'un utilisateur peut être autorisé à effectuer sur un objet de la base ; le privilège est accordé par le créateur de l'objet.

Relation (*Relation*) : Ensemble non ordonné de tuples. Une relation contient toujours une clé primaire. Chaque tuple est unique et est accessible de façon univoque par une valeur de la clé primaire. Une relation est toujours représentée au niveau conceptuel, donc en SQL, comme une table à deux dimensions, composées de lignes constituées d'attributs. Pour être

manipulable par SQL, une relation doit être en "première forme normale" (ou "normalisée"), c'est à dire que chaque attribut doit avoir une valeur atomique.

SGBDR (*RDBMS*) : Voir "Système de gestion de bases de données relationnelles".

Sous-requêt e ou **requête imbriquée**(*Subquery*) : Requête entourée de parenthèses apparaissant dans un autre requête (appelée requête extérieure), et permettant de conditionner la sélection des tuples de cette dernière.

Synonyme (*Synonym*) : Nom donné à une table ou une vue dans le but de simplifier les références à cet objet. Un synonyme est personnel et persiste jusqu'à ce que son créateur le détruise. Le synonyme est d'application dans toutes les requêtes où l'objet intervient. A ne pas confondre avec 'alias'.

Système de gestion de bases de données relationnelle s: ensemble de programmes permettant la mémorisation, la manipulation et le contrôle des données dans une ou plusieurs bases de données relationnelles. Outre le noyau relationnel qui gère les tables et optimise les requêtes, un SGBDR possède un langage d'exploitation de quatrième génération ainsi que divers outils (générateurs d'écrans, de documents, d'applications,...) favorisant le développement rapide d'applications. CODD et DATE ont défini un ensemble de règles strictes permettant de déterminer le niveau de 'relationnalité' d'un SGBDR.

Table (*Table*) : Voir relation.

Transaction (*Transaction*) : Ensemble logique d'événements et d'actions qui modifient le contenu d'une base de données, faisant passer la base d'un état cohérent à un autre état cohérent.

Tuple (*Tuple*) : Une ligne dans une relation (ou table). Un tuple est constitué de valeurs correspondant aux attributs qui constituent la relation. La cardinalité d'une relation est le nombre de tuples de la relation.

Vue (*View*) : Objet d'une base de données relationnelle et du langage SQL qui peut généralement être utilisé comme une table. Une vue réalise un résumé, une simplification, ou une personnalisation de la base de données en regroupant des données sélectionnées de plusieurs tables.