



Dotnet France
Technologies Sharepoint, SQL Server & .NET

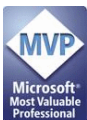
Association Dotnet France

Les bases fondamentales du langage Transact SQL

Version 1.0



Grégory CASANOVA



James RAVAILLE

<http://blogs.dotnet-france.com/jamesr>

Sommaire

1	Introduction.....	4
2	Pré-requis	5
2.1	Présentation	5
2.2	Les expressions	5
2.3	Les opérateurs	6
2.4	Les fonctions.....	7
3	Les instructions DML	12
3.1	Présentation	12
3.2	Création, modification et suppression de données	12
3.2.1	L'instruction INSERT	12
3.2.2	L'instruction UPDATE.....	14
3.2.3	L'instruction DELETE	15
3.3	Lire et trier des données	16
3.3.1	L'instruction SELECT	16
3.3.2	Changer le nom des colonnes (ALIAS)	16
3.3.3	La condition WHERE	17
3.3.4	Les projections de données.....	18
3.3.5	Les calculs simples.....	20
3.3.6	Le produit cartésien.....	20
3.3.7	Les jointures	21
3.3.8	La close ORDER BY	23
3.3.9	L'opérateur UNION.....	24
3.3.10	L'opérateur EXCEPT	25
3.3.11	L'opérateur INTERSECT.....	25
3.3.12	La clause TOP	26
3.3.13	Créer une table grâce à SELECT INTO	26
3.3.14	La clause COMPUTE et COMPUTE BY	27
3.3.15	Les opérateurs ROLLUP et CUBE	28
3.3.16	L'opérateur OVER	29
3.3.17	L'opérateur NTILE	30
3.3.18	Les sous-requêtes.....	30

3.3.19	Les instructions PIVOT et UNPIVOT.....	30
3.3.20	L'instruction MERGE.....	32
4	Le SQL Procédural.....	33
4.1	Les variables	33
4.1.1	Les variables utilisateur	33
4.1.2	Les variables système	33
4.2	Les transactions.....	33
4.3	Les lots et les scripts.....	34
4.4	Le contrôle de flux.....	35
4.4.1	L'instruction RETURN.....	35
4.4.2	L'instruction PRINT	35
4.4.3	L'instruction CASE.....	36
4.4.4	Les blocs BEGIN ... END.....	36
4.5	La gestion des curseurs	38
4.6	Les exceptions	41
4.6.1	Lever une exception	41
4.6.2	Gestion des erreurs dans le code	42
5	Conclusion	43

1 Introduction

Dans ce cours, nous allons étudier les bases du langage Transact SQL. La version du langage Transact SQL utilisée est celle de SQL Server 2008. Pour ce faire, nous allons définir les différentes parties du langage (DML, DDL, DCL), puis détailler la partie DML, qui est celle qui sert à manipuler les données de façon générale.

2 Pré-requis

Avant de lire ce cours, nous vous conseillons :

- D'avoir déjà utilisé l'interface d'administration de SQL Server 2008 : SQL Server Management Studio (Chapitre 1).
- D'avoir les bases dans la construction d'un modèle relationnel de données (Chapitre 2).
- Les bases fondamentales du langage T-SQL

2.1 Présentation

Le T-SQL (Transact Structured Query Language) est un langage de communication avec une base de données relationnelle SQL Server. Il définit une batterie « simple » mais complète de toutes les opérations exécutables sur une base de données (lecture de données, opérations d'administration du serveur, ajout, suppression et mises à jour d'objets SQL - tables, vues, procédures stockées, déclencheurs, types de données personnalisés ... -). Ce langage est composé d'instructions, réparties dans de 3 catégories distinctes :

- **DML : Data Modification Language**, soit *langage de manipulation de données*. Dans cette catégorie, s'inscrivent les instructions telles que l'instruction SELECT ou encore les instructions qui nous permettent la création, la mise à jour et la suppression de données stockées dans les tables de la base de données. Il est important de retenir que le DML sert simplement pour les données, et en aucun cas pour la création, mise à jour ou suppression d'objets dans la base de données SQL Server.
- **DDL : Data Definition Language**, soit *langage de définition de données*. Les instructions de cette catégorie, permettent d'administrer la base de données, ainsi que les objets qu'elle contient. Elles ne permettent pas de travailler sur les données. Aussi, elles ne seront pas traitées dans ce chapitre.
- **DCL : Data Control Language**, soit *langage de contrôle d'accès*. Cette catégorie d'instructions nous permet de gérer les accès (autorisations) aux données, aux objets SQL, aux transactions et aux configurations générales de la base.

Ces trois catégories combinées permettent que le langage T-SQL prenne en compte des fonctionnalités algorithmiques, et admette la programmabilité. Le T-SQL est non seulement un langage de requêtage, mais aussi un vrai langage de programmation à part entière. Sa capacité à écrire des procédures stockées et des déclencheurs (Triggers), lui permet d'être utilisé dans un environnement client de type .NET, au travers d'une application en C# ou en VB.NET. Dans ce chapitre, nous allons détailler la partie DML du T-SQL exclusivement. Auparavant, nous étudierons différents éléments syntaxiques qui composeront la syntaxe de ce langage, à savoir les expressions, les opérateurs et les fonctions. Par la suite, nous traiterons l'aspect procédural (algorithmique) de ce langage.

2.2 Les expressions

Dans le T-SQL, nous pouvons utiliser des expressions, permettant de mettre en œuvre l'aspect algorithmique du langage. Les expressions peuvent prendre plusieurs formes.

- **Les constantes** : une constante est une variable, dont la valeur ne peut être changée lors de l'exécution d'instructions T-SQL.

- **Les noms de colonnes** : ils pourront être utilisés comme expressions. La valeur de l'expression étant la valeur stockée dans une colonne pour une ligne donnée.
- **Les variables** : il s'agit d'entités qui peuvent être employées en tant qu'expressions ou dans des expressions. Les variables sont préfixées par le caractère @. Les variables systèmes sont préfixées par les caractères @@. La valeur de l'expression variable est la valeur de la variable elle-même.
- **Les fonctions** : il est possible d'utiliser comme expression n'importe quelle fonction. Elles permettent d'exécuter des blocs d'instructions T-SQL, et de retourner une valeur.
- **Les expressions booléennes** : elles sont destinées à tester des conditions. Elles sont utilisées dans des structures algorithmiques de type WHILE, IF ou encore dans la clause WHERE d'une requête SQL, à affiner de permettre d'afficher une recherche, ou bien à poser une condition d'exécution.
- **Les sous-requêtes** : une sous requête SELECT peut être placée en tant qu'expression. La valeur de l'expression est la valeur renvoyée par la requête.

2.3 Les opérateurs

Les opérateurs nous permettent de combiner des expressions, des expressions calculées ou des expressions booléennes. Il existe plusieurs types d'opérateurs, que nous allons détailler :

- Les opérateurs arithmétiques :

+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste de division)

- Les opérateurs de bits :

&	ET
	OU
^	OU exclusif
~	NON

- Les opérateurs de comparaison :

=	Égale
>	Supérieur
>=	Supérieur ou égal
<	Inférieur
<=	Inférieur ou égal
<>	Différent
Exp1 IN (exp2, exp3, ...)	Compare l'expression seule à toutes les expressions de la liste
IS NULL	Renvoie True si l'expression est NULL. False le cas échéant
Exp1 BETWEEN minimum AND maximum	Recherche si la valeur de Exp1 est comprise entre la valeur « minimum » et « maximum ».

	Les bornes minimum et maximum sont incluses
EXISTS (Sous Requête)	Renvoie True, si et seulement si la sous requête renvoie au moins une ligne
Exp1 LIKE	Permet de filtrer des données suivant un modèle

Pour l'opérateur de comparaison LIKE, les expressions permettent de définir un modèle de recherche pour la correspondance des données :

_	Un caractère quelconque
%	N caractères quelconques
[ab...]	Un caractère dans la liste ab...
[a-z]	Un caractère dans l'intervalle a-z
[^ab...]	Un caractère en dehors de la liste ou de l'intervalle spécifié
ab...	Le ou les caractères eux-mêmes

- Les opérateurs logiques :

OR	Retourne True si une expression des deux expressions (opérandes) est vraie
AND	Retourne True si les deux expressions (opérandes) sont vraies.
NOT	True si l'expression est fausse.

2.4 Les fonctions

Les fonctions se distinguent en deux catégories : celles créées par l'utilisateur, ou les fonctions système. Nous allons détailler ci-dessous les fonctions système, les fonctions utilisateur seront traitées dans un autre cours. Les fonctions système se divisent en différentes catégories :

- Les fonctions d'agrégation :

COUNT (*)	Dénombre les lignes sélectionnées
COUNT ([ALL DISTINCT] exp1)	Dénombre toutes les expressions non nulles ou les expressions non nulles distinctes
COUNT_BIG	Possède le même fonctionnement que la fonction COUNT, simplement, le type de données de sortie est de type bigint au lieu de int
SUM ([ALL DISTINCT] exp1)	Somme de toutes les expressions non nulles ou des expressions non nulles distinctes
AVG ([ALL DISTINCT] exp1)	Moyenne de toutes les expressions non nulles ou des expressions non nulles distinctes
MIN (exp1) OU MAX (exp1)	Valeur MIN ou valeur MAX d'exp1
STDEV ([ALL DISTINCT] exp1)	Ecart type de toutes les valeurs de l'expression donnée
STDEVP ([ALL DISTINCT] exp1)	Ecart type de la population pour toutes les valeurs de l'expression donnée
VAR ([ALL DISTINCT] exp1)	Variance de toutes les valeurs de l'expression donnée

VARP ([ALL DISTINCT] exp1)	Variance de la population pour toutes les valeurs donnée
GROUPING	S'utilise avec ROLLUP ou CUBE. Indique 1 quand la ligne est générée par un ROLLUP ou un CUBE et 0 dans un autre cas
CHECKSUM (* [exp1...])	Permet de calculer un code de contrôle par rapport à une ligne de la table ou par rapport à une liste d'expression. Cette fonction permet la production d'un code de hachage
CHECKSUM_AGG ([ALL DISTINCT] exp1)	Permet le calcul d'une valeur de hachage par rapport à un groupe de données. Ce code de contrôle permet de savoir rapidement si des modifications ont eu lieu sur un groupe de données, car cette valeur de contrôle n'est plus la même après modification des données

- Les fonctions mathématiques :

ABS (exp1)	Valeur absolue d'exp1.
CEILING (exp1)	Plus petit entier supérieur ou égal à exp1.
FLOOR (exp1)	Plus grand entier supérieur ou égal à exp1.
SIGN (exp1)	Renvoie 1 si exp1 est positive, -1 si elle est négative, et 0 si elle est égale à 0.
SQRT (exp1)	Racine carrée d'exp1.
POWER (exp1, n)	Exp1 à la puissance n.
SQUARE (exp1)	Calcul du carré d'exp1.

- Les fonctions trigonométriques :

PI ()	Valeur de PI.
DEGREES (exp1)	Conversion d'exp1 de radian vers degrés.
RADIANS (exp1)	Conversion d'exp1 de degrés vers radians.
SIN (exp1), COS (exp1), TAN (exp1), COT (exp1)	Sin, cos ou tangente d'exp1.
ACOS (exp1), ASIN (exp1), ATAN (exp1)	Arc cos, arc sin ou arc tan d'exp1.
ATN2 (exp1, exp2)	Angle dont la tangente se trouve dans l'intervalle exp1 et exp2.

- Les fonctions logarithmiques :

EXP (exp1)	Exponentielle d'exp1.
LOG (exp1)	Logarithme d'exp1.
LOG10 (exp1)	Logarithme base 10 d'exp1.

- Les fonctions de dates :

Format	Abréviation	signification
Year	Yy, yyyy	Année (1753 à 9999)
quarter	Qq, q	Trimestre (1 à 4)
Month	Mm, m	Mois (1 à 12)
Day of year	Dy, y	Jour de l'année (1 à 366)

Day	Dd, d	Jour dans le mois (1 à 31)
Weekday	Dw, ww	Jour de la semaine (1 à 7)
Hour	Hh	Heure (0 à 23)
Minute	Mi, n	Minute (0 à 59)
Seconds	Ss, s	Seconde (0 à 59)
milliseconds	Ms	Milliseconde (0 à 999)

GETDATE ()	Date et Heure système.
DATENAME (format, exp1)	Renvoie la partie date sous forme de texte.
DATEPART (format, exp1)	Renvoie la valeur de la partie date selon le format donné.
DATEDIFF (format, exp1, exp2)	Différence entre les deux tables selon le format donné.
DATEADD (format, p, exp1)	Ajoute p format à la date exp1.
DAY (exp1)	Retourne le numéro du jour dans le mois.
MONTH (exp1)	Retourne le numéro du mois.
YEAR (exp1)	Retourne l'année.
SWITCHOFFSET (datetimeoffset, zone_horaire)	Convertis le type datetimeoffset en le type passé en second paramètre.
SYSDATETIME	Retourne la date et l'heure usuelle du serveur dans le format datetime2.
SYSDATETIMEOFFSET	Fonctionne de la même manière que SYSDATETIME, mais il prend en compte le décalage GMT.

- Les fonctions de chaîne de caractères :

ASCII (exp1)	Valeur du code ASCII du premier caractère d'exp1.
UNICODE (exp1)	Valeur numérique correspondant au code UNICODE d'exp1.
CHAR (exp1)	Caractère correspondant au code ASCII d'exp1.
NCHAR (exp1)	Caractère UNICODE correspondant au code numérique d'exp1.
LTRIM (exp1), RTRIM (exp1)	Supprime les espaces à droite pour RTRIM et à gauche pour LTRIM d'exp1.
STR (exp1, n, p)	Convertit le nombre exp1, en chaîne de longueur maximale n dont p caractères seront à droite de la marque décimale.
SPACE (n)	Renvoie n espaces.
REPLICATE (exp1, n)	Renvoie n fois exp1.
CHARINDEX ('masque', exp1) PATINDEX ('%masque%', exp1)	Renvoie la position de départ de la première expression 'masque' dans exp1. PATINDEX permet d'utiliser des caractères génériques et de travailler avec certains type comme TEXT, CHAR ou encore VARCHAR.
LOWER (exp1), UPPER (exp1)	Change la casse. LOWER va convertir exp1 en minuscules et UPPER va convertir exp1 en majuscules.
REVERSE (exp1)	Retourne les caractères d'exp1 dans le sens inverse.



RIGHT (exp1, n)	Renvoie les n caractères les plus à droite d'exp1.
LEFT (exp1, n)	Renvoie les n caractères les plus à gauche d'exp1.
SUBSTRING (exp1, n, p)	Renvoie p caractères d'exp1 à partir de n.
STUFF (exp1, n, p, exp2)	Supprime p caractères d'exp1, à partir de n, puis insère exp2 à la position n.
SOUNDEX (exp1)	Renvoie le code phonétique d'exp1.
DIFFERENCE (exp1, exp2)	Compare les SOUNDEX des deux expressions. La valeur, qui peut être renvoyée va de 1 à 4, valeur pour laquelle, les deux expressions possèdent la plus grande similitude.
LEN (exp1)	Retourne le nombre de caractères d'exp1.
QUOTENAME (exp1)	Permet de transformer exp1 en identifiant valide pour SQL Server.
REPLACE (exp1, exp2, exp3)	Permet de remplacer dans exp1 toutes les occurrences d'exp2 par exp3.

- Les Fonctions systèmes :

COALESCE (exp1, exp2...)	Renvoie la première expression non NULL.
COL_LENGTH (nom_table, nom_colonne)	Longueur de la colonne.
COL_NAME (id_table, id_colonne)	Nom de la colonne.
DATALLENGTH (exp1)	Longueur en octet de l'expression.
DB_ID (Nom_base)	Numéro d'identification de la base de données.
DB_NAME (id_base)	Nom de la base.
GETANSINULL (nom_base)	Renvoie 1 si l'option 'ANSI NULL DEFAULT' est positionné pour la base.
HOST_ID ()	Numéro d'identification du poste.
HOST_NAME ()	Nom du poste.
IDENT_INCR (nom_table)	Valeur de l'incrémenté défini pour la colonne identité de la table spécifiée.
IDENT_SEED (nom_table)	Valeur initiale définie pour la colonne identité de la table indiquée.
IDENT_CURRENT (nom_table)	Retourne la dernière valeur de type identité utilisé par cette table.
INDEX_COL (nom_table, id_index, id_cle)	Nom de la colonne indexé correspondant à l'index.
ISDATE (exp1)	Renvoie 1 si l'expression de type varchar possède un format date valide.
ISNULL (exp1, valeur)	Renvoie valeur si exp1 est NULL.
ISNUMERIC (exp1)	Renvoie 1 si l'expression de type varchar a un format numérique valide.
NULLIF (exp1, exp2)	Renvoie NULL si exp1 = exp2.
OBJECT_ID (objet)	Numéro d'identification de l'objet.
OBJECT_ID (name)	Nom de l'objet dont l'id est placé en argument.
STATS_DATE (id_table, id_index)	Date de la dernière mise à jour de l'index.
SUSER_SID (nom_accès)	Numéro d'identification correspondant au nom_accès.
SUSER_SNAME (id)	Nom d'accès identifié par l'id.
USER_NAME (id)	Nom de l'utilisateur dont l'id est placé en

	argument.
CURRENT_TIMESTAMP	Date et heure système, équivalent à GETDATE ().
SYSTEM_USER	Nom d'accès.
CURRENT_USER, USER, SESSION_USER	Nom de l'utilisateur de la session.
OBJECT_PROPERTY (id, propriété)	Permet de retrouver les propriétés de la base.
ROW_NUMBER	Permet de connaître le numéro d'une ligne issue d'une partition depuis un jeu de résultats.
RANK	Permet de connaître le rang d'une ligne issue d'une partition dans une série de résultats.
DENSE_RANK	Fonctionne comme RANK, mais ne s'applique qu'aux lignes de la série de résultat.
HAS_DBACCESS (nom_base)	Permet de savoir si, avec le contexte de sécurité actuel, il est possible d'accéder à la base. (retourne 1 dans ce cas, dans le cas contraire, 0)
HAS_PERMS_BY_NAME	Permet de savoir par programmation, si l'on dispose d'un privilège ou non.
KILL	Cette fonction permet de mettre fin à une session utilisateur.
NEWID ()	Permet de gérer une valeur de type UniqueIdentifier.
NEWSEQUENTIALID ()	Permet de gérer la prochaine valeur de type UniqueIdentifier.
PARSENAME (nom_objet, partie_à_extraire)	Permet d'extraire à partir du nom complet de l'objet, le nom de l'objet. La partie partie_à_extraire peut prendre la valeur 1, 2, 3, 4 selon si l'on veut extraire le nom de l'objet, le schéma, la base, ou encore le nom du serveur.
PUBLISHINGSERVERNAME	Permet de savoir qui est à l'origine d'une publication.
STUFF (chaine1, n, p, chaine2)	Permet de supprimer p caractères de la chaîne chaine1, à partir des positions n, puis d'y insérer chaine2

- Les fonctions conversion de types :

CAST (exp1 AS types_données)	Permet de convertir une valeur dans le type spécifié en argument
CONVERT (types_données, exp1, style)	Conversion de l'expression dans le type de données spécifié. Un style peut être spécifié dans le cas d'une conversion date ou heure

- Les fonctions diverses :

RAND (exp1)	Nombre aléatoire compris en 0 et 1. Exp1 est la valeur de départ
ROUND (exp1, n)	Arrondis exp1 à n chiffres après la virgule

3 Les instructions DML

3.1 Présentation

Pour toutes les instructions du DML, il existe dans SQL Server un outil simple pour retrouver la syntaxe voulue rapidement (Pour des instructions simples, telle le SELECT, UPDATE...). La démarche est simple. Via le menu contextuel d'une table, sélectionnez « Générer un script de la table en tant que... ». Il nous est alors proposé de sélectionner l'action que nous voulons accomplir : SELECT, INSERT, UPDATE ou DELETE. Cette action peut aussi être réalisée sur d'autres objets SQL de la base de données.

3.2 Création, modification et suppression de données

3.2.1 L'instruction INSERT

L'instruction `INSERT`, comme son nom l'indique, va nous permettre d'ajouter une ligne de données dans une table de la base de données. Le code générique, d'ajout d'une ligne de données est la suivante :

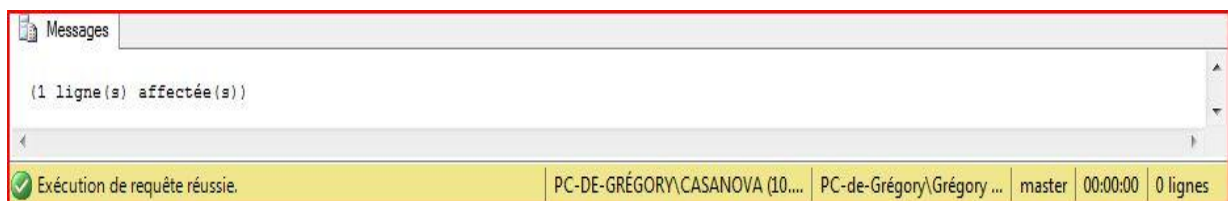
```
INSERT INTO [Entreprise].[dbo].[Client]
    ([Nom_Client]
    , [Prenom_Client]
    , [Numero_Client]
    , [Adresse_Client]
    , [Mail_Client])
VALUES
    (<Nom_Client, varchar(50),>
    , <Prenom_Client, varchar(50),>
    , <Numero_Client, varchar(20),>
    , <Adresse_Client, varchar(50),>
    , <Mail_Client, varchar(50),>)
GO
```

Dans ce code générique, nous demandons à SQL Server d'ajouter un enregistrement à la table Client, appartenant au schéma dbo dans la base de données Entreprise. Pour préciser les colonnes pour lesquelles nous allons ajouter des données, il est nécessaire de préciser le nom des colonnes, après l'instruction `INSERT INTO`. Le mot clé `VALUES` nous permet de fournir des valeurs aux champs. Il est impératif que les valeurs soient dans le même ordre que celui des colonnes, tout d'abord pour la cohérence des données, mais aussi pour respecter la compatibilité des données avec le type que vous avez assigné à votre table au moment de sa création. Dans le cas où certaines de vos colonnes acceptent des valeurs NULL, il existe deux méthodes pour obtenir cette valeur. La première, est d'omettre le nom de la colonne et la valeur correspondante dans l'instruction. La seconde vise à laisser la colonne dans la description, mais à préciser le mot clé NULL dans la clause `VALUES`. Pour des chaînes de caractères, il faut placer celles-ci entre simples cotes. Dans le cas d'un champ de type identité (possédant une incrémentation automatique grâce à la contrainte IDENTITY), il n'est pas nécessaire de spécifier ni le nom du champ, ni sa valeur.

Procédons à un exemple pour mieux comprendre :

```
INSERT INTO [Client]
(Nom_Client,
Prenom_Client,
Numero_Client,
Adresse_Client,
Mail_Client)
VALUES
('CASANOVA',
'Grégory',
+33563456764,
'31 place de la chance',
'75554@supinfo.com')
GO
```

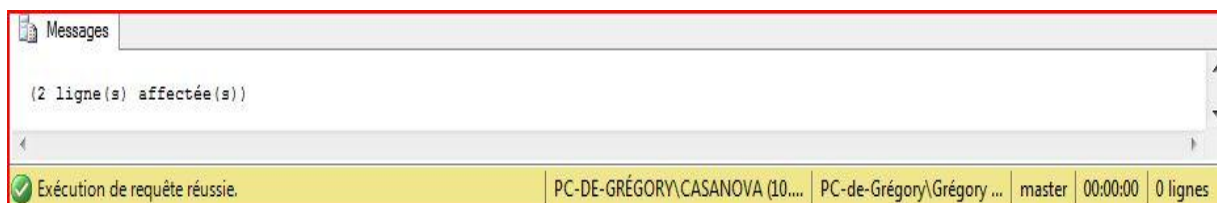
Après avoir exécuté le code ci-dessus, le message suivant apparaît, confirmant de sa bonne exécution :



Dans le cas d'une insertion multiple d'enregistrements, la syntaxe sera la même, à l'exception près qu'au lieu d'une seule série de données après le mot clé **VALUES**, vous en spécifier le nombre voulu. Si nous voulons ajouter deux enregistrements dans une même instruction Insert, alors la syntaxe est la suivante :

```
INSERT INTO [Client]
(Nom_Client,
Prenom_Client,
Numero_Client,
Adresse_Client,
Mail_Client)
VALUES
('CASANOVA',
'Grégory',
+33563456764,
'31 place de la chance',
'75554@supinfo.com'),
('RAVAILLE',
'James',
+33567876435,
'34 Avenue de le paix',
'James.Ravaille@Domaine.fr')
GO
```

Le message suivant s'affiche, après l'exécution de cette instruction, ce qui confirme bien que l'enregistrement multiple a été exécuté sans erreur :



Enfin, il est possible d'ajouter des enregistrements à l'aide de l'instruction **SELECT**, qui va copier les enregistrements d'une table (source) vers une autre table (destination). Voici un exemple :

```
INSERT Commande
SELECT  Id_Client, GETDATE(), Id_Stock, 1
FROM Client, Stock
WHERE Id_Client = 3
AND Id_Stock = 5
```

Dans ce cas, nous allons ajouter dans la table commande, les informations sélectionnées. Ici, **Id_Client**, la date du jour grâce à la fonction **GETDATE()**, **Id_Stock**, et le chiffre 1 qui correspond à la quantité que nous voulons ajouter à la commande de notre client. Les informations concernant **Id_Client** et **Id_Stock** seront sélectionnées en fonction des conditions précisées après la clause **WHERE**. Grâce à ce lot, nous allons ajouter la troisième ligne présente dans le résultat présenté ci-dessous.

	Id_Commande	Id_Client	Date_Commande	Id_Stock	Quantite
1	1	1	2009-06-04	1	1
2	2	4	2009-06-04	5	20
3	3	3	2009-06-04	5	1

3.2.2 L'instruction UPDATE

L'instruction **UPDATE**, permet de mettre à jour un ou plusieurs enregistrements. La syntaxe générique de cette instruction est la suivante :

```
UPDATE [Entreprise].[dbo].[Client]
SET [Nom_Client] = <Nom_Client, varchar(50),>
, [Prenom_Client] = <Prenom_Client, varchar(50),>
, [Numero_Client] = <Numero_Client, varchar(20),>
, [Adresse_Client] = <Adresse_Client, varchar(50),>
, [Mail_Client] = <Mail_Client, varchar(50),>
WHERE <Conditions de recherche,,>
GO
```

L'instruction ci-dessus permet de mettre à jour la table **Client** de la base de données **Entreprise**. La clause **SET** permet d'indiquer les champs à mettre à jour. La clause **WHERE**, sert à cibler les enregistrements à mettre à jour. Voici l'enregistrement de la table **Client** dont le champ **Id-Client** vaut 3 :

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	3	DOLLON	Julien	33563765514	17 Rue du cotton	Jul.Dollon@dotnetfrance.com

Voici une instruction SQL permettant de modifier le nom de ce client :

```
UPDATE [Entreprise].[dbo].[Client]
SET [Adresse_Client] = '18 Rue du cotton'
WHERE Id_Client = 3
GO
```

Après l'exécution de l'instruction ci-dessus, voici les données de l'enregistrement modifié :

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	3	DOLLON	Julien	33563765514	18 Rue du cotton	Jul.Dollon@dotnetfrance.com

Il est aussi possible d'effectuer des opérations grâce à un **UPDATE**. Par exemple, on peut augmenter les prix des articles d'un magasin de 10%, en multipliant le prix de tous les articles par 1,1.

3.2.3 L'instruction DELETE

L'instruction **DELETE** permet de supprimer des enregistrements. La syntaxe générique est la suivante :

```
DELETE FROM [Entreprise].[dbo].[Client]
WHERE <Conditions de recherche,>
GO
```

L'instruction **DELETE FROM** va permettre la suppression de données dans la table Client de la base de données Entreprise, dans la seule condition que les contraintes dans **WHERE** soient respectées. Voici une instruction permettant de supprimer l'enregistrement de la table Client dont l'identifiant est 4 :

```
DELETE FROM [Entreprise].[dbo].[Client]
WHERE Id_Client = 4
GO
```

Après avoir exécuté le code, on remarque que le client dont l'identifiant est 4, n'existe plus :

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	1	CASANOVA	Grégory	33563456764	31 place de la chance	75554@supinfo.com
2	2	RAVAILLE	James	33567876435	34 Avenue de le paix	James.Ravaille@Domaine.fr
3	3	DOLLON	Julien	33563765514	18 Rue du cotton	Jul.Dollon@dotnetfrance.com
4	5	VASSELON	Jean Christophe	33567654342	19 avenue du sac	JCVD@leursdomaine.com
5	6	HOLLEBEC	Mathieu	33578763450	26 place des canons	MathHol@votredomaine.com

La suppression multiple de données est possible, par exemple si dans notre cas, nous avons précisé une plage d'identifiants dans notre clause **WHERE**.

3.3 Lire et trier des données

3.3.1 L'instruction SELECT

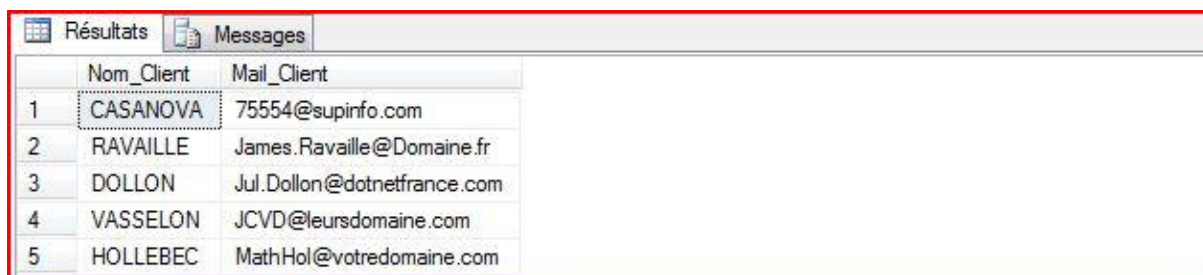
L'instruction **SELECT** permet de sélectionner des données (tout ou partie d'enregistrements), d'une ou plusieurs tables. Elle offre aussi la possibilité de les trier, et de les regrouper. La syntaxe générale de cette instruction est la suivante :

```
SELECT [Id_Client]
      , [Nom_Client]
      , [Prenom_Client]
      , [Numero_Client]
      , [Adresse_Client]
      , [Mail_Client]
FROM [Entreprise].[dbo].[Client]
GO
```

Voici une instruction **SELECT** permettant de lire le nom et l'adresse Email de tous les clients (si notre but avait été de sélectionner toutes les colonnes, au lieu de lister toutes celles-ci, il est possible d'indiquer que nous les sélectionnons toutes avec le simple caractère « * ») :

```
SELECT [Nom_Client]
      , [Mail_Client]
FROM [Entreprise].[dbo].[Client]
GO
```

Le résultat sera le suivant :



	Nom_Client	Mail_Client
1	CASANOVA	75554@supinfo.com
2	RAVAILLE	James.Ravaille@Domaine.fr
3	DOLLON	Jul.Dollon@dotnetfrance.com
4	VASSELON	JCVD@leursdomaine.com
5	HOLLEBEC	MathHol@votredomaine.com

3.3.2 Changer le nom des colonnes (ALIAS)

Par défaut, le nom de la colonne est celui du nom de la colonne dans la table. Il est possible d'en changer en utilisant des alias. Voici un exemple d'utilisation d'alias :

```

-----
--Il existe deux manières de renommer les colonnes.
--Celle-ci :
-----

SELECT 'Nom Client' = [Nom_Client]
      , 'Mail Client' = [Mail_Client]
FROM [Entreprise].[dbo].[Client]
GO

-----
--Ou encore celle là :
-----

SELECT [Nom_Client] AS 'Nom Client'
      , [Mail_Client] AS 'Mail Client'
FROM [Entreprise].[dbo].[Client]
GO

```

Le nom des colonnes est changé par un nom « plus explicite » :

	Nom Client	Mail Client
1	CASANOVA	75554@supinfo.com
2	RAVAILLE	James.Ravaille@Domaine.fr
3	DOLLON	Jul.Dollon@dotnetfrance.com
4	VASSELON	JCVD@leursdomaine.com
5	HOLLEBEC	MathHol@votredomaine.com

3.3.3 La condition WHERE

Il est alors possible d'ajouter des conditions à notre recherche pour l'affiner, au travers de la clause **WHERE**. Les restrictions servent à limiter le nombre d'enregistrements à sélectionner. Les conditions contenues dans le **WHERE** sont des expressions booléennes qui peuvent être composées de noms de colonnes, de constantes, de fonctions, d'opérateurs de comparaison et d'opérateurs logiques. Prenons un exemple concret :

```

SELECT [Nom_Client] AS 'Nom Client'
      , [Mail_Client] AS 'Mail Client'
FROM [Entreprise].[dbo].[Client]
WHERE Id_Client IN (1,2,3,6)
GO

```

Cette instruction **SELECT** sélectionne tous les champs de tous les enregistrements pour lesquels la colonne **Id_Client** est égale soit à 1, 2, 3 et 6. On remarque alors que dans notre code, nous avons utilisé la condition **WHERE**, une colonne, un opérateur de comparaison et un opérateur logique. Le résultat est le suivant :

	Nom Client	Mail Client
1	CASANOVA	75554@supinfo.com
2	RAVAILLE	James.Ravaille@Domaine.fr
3	DOLLON	Jul.Dollon@dotnetfrance.com
4	VASSELON	JCVD@leursdomaine.com
5	HOLLEBEC	MathHol@votredomaine.com

```

SELECT [Nom_Client] AS 'Nom Client'
      , [Mail_Client] AS 'Mail Client'
FROM [Entreprise].[dbo].[Client]
WHERE Id_Client BETWEEN 1 AND 10
GO

```

L'instruction ci-dessus présente l'utilisation des clauses `WHERE` et `BETWEEN`, qui permet de lire tous les enregistrements dont l'identifiant est compris entre 1 et 10 (bornes incluses). Le résultat est le suivant :

	Nom Client	Mail Client
1	CASANOVA	75554@supinfo.com
2	RAVAILLE	James.Ravaille@Domaine.fr
3	DOLLON	Jul.Dollon@dotnetfrance.com

3.3.4 Les projections de données

Les projections de données sont utiles dans certains cas, par exemple lorsque vous voulez lister les villes dans lesquelles sont présents vos clients. Une projection va grouper les enregistrements identiques dans un seul et même enregistrement. Voici les deux cas possibles de projection :

```

-----
-- Deux façons de grouper les colonnes identiques :
-- Celle-ci :
-----

SELECT Mesure, COUNT(Mesure) AS 'Nombre article avec cette mesure'
FROM Stock
GROUP BY Mesure
GO

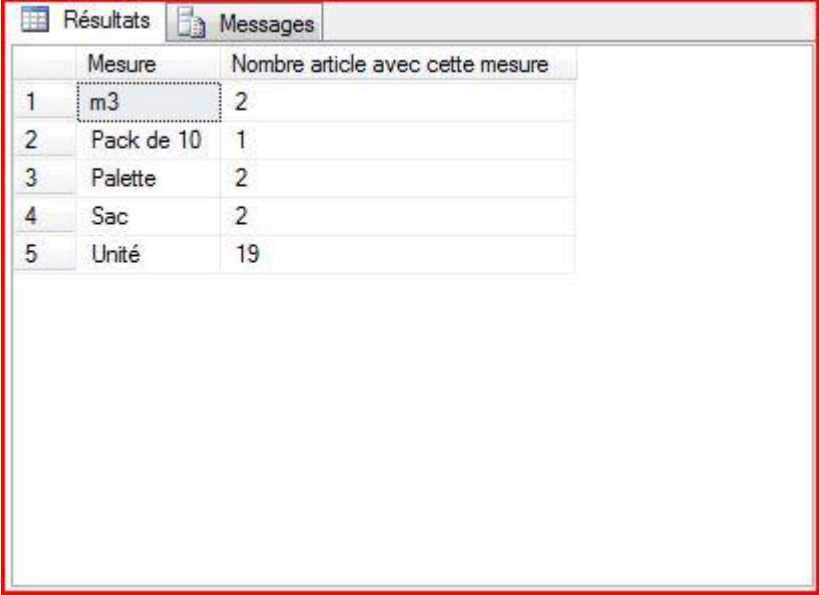
-----
-- Ou celle là :
-----

SELECT DISTINCT Mesure
FROM Stock
GO

```

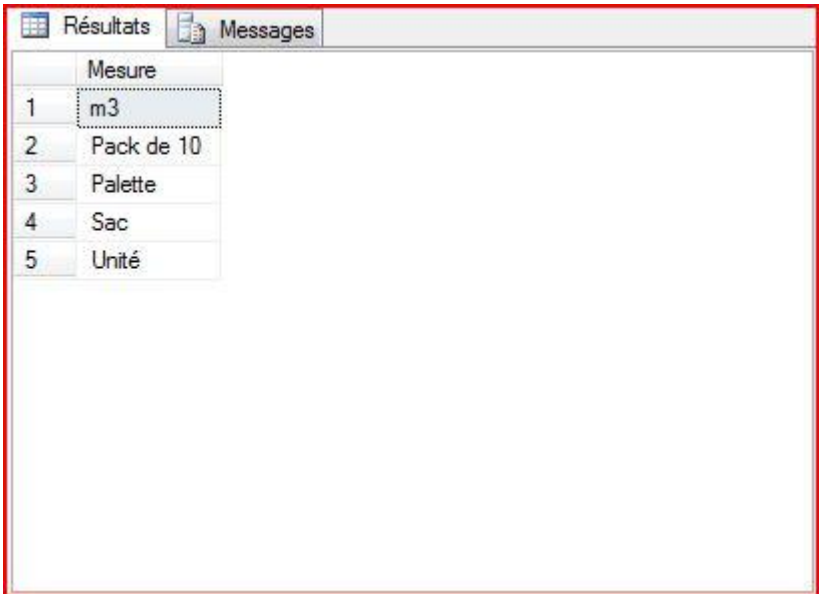
Dans le premier morceau de code, nous allons afficher une seule ligne de chaque résultat, même si plusieurs résultats existent pour la colonne `Mesure`, et nous comptons le nombre d'occurrence qui intervient pour chaque `Mesure`, grâce à la fonction `COUNT()`, associée à la clause

GROUP BY. Ce genre d'instruction peu être pratique dans le cas où l'on veut calculer le pourcentage de vente en fonction de la localisation d'un magasin par exemple. On n'affichera qu'une seule fois la localisation du magasin grâce à la clause **GROUP BY**, et on affichera pour chaque localisation, le nombre de vente effectuée. On peut alors facilement en déduire lequel des magasins est le plus productif. Pour revenir à notre exemple, nous pouvons déduire du résultat que nous vendons plus d'articles à l'unité, que tout le reste des articles.



	Mesure	Nombre article avec cette mesure
1	m3	2
2	Pack de 10	1
3	Palette	2
4	Sac	2
5	Unité	19

Pour le second morceau de code, on pourra seulement afficher les résultats de façon distincte, c'est-à-dire en évitant les doublons comme dans le premier exemple. En revanche, il ne sera pas possible d'utiliser une fonction d'agrégation, type **COUNT()**, car elle doit être contenue dans une clause **GROUP BY**. On obtiendra alors le résultat identique au premier exemple, hors mis le fait que nous ne pouvons pas compter le nombre d'occurrence de chaque mesure dans la colonne Mesure.



	Mesure
1	m3
2	Pack de 10
3	Palette
4	Sac
5	Unité

3.3.5 Les calculs simples

Les calculs, comme nous les appelons, regrouperont les calculs numériques mais aussi les manipulations sur les chaînes de caractères, par exemple la concaténation. Les modèles sont les suivants :

```
SELECT Id_Stock,
'Quantité Produit' = Quantite * 3
FROM Stock
```

Ici, la quantité de chaque Stock sera multipliée par trois dans le résultat de la recherche par l'instruction `SELECT`. Mais la valeur de la quantité de produit ne sera en aucun cas changer dans la base de données.

```
SELECT Nom_Client + ' ' + Prenom_Client AS 'NOM COMPLET'
FROM Client
```

Dans l'instruction ci-dessus, nous concaténons les champs `Nom_Client` et `Prenom_Client` en une seule colonne que nous appellerons `NOM COMPLET`. Le résultat est le suivant :

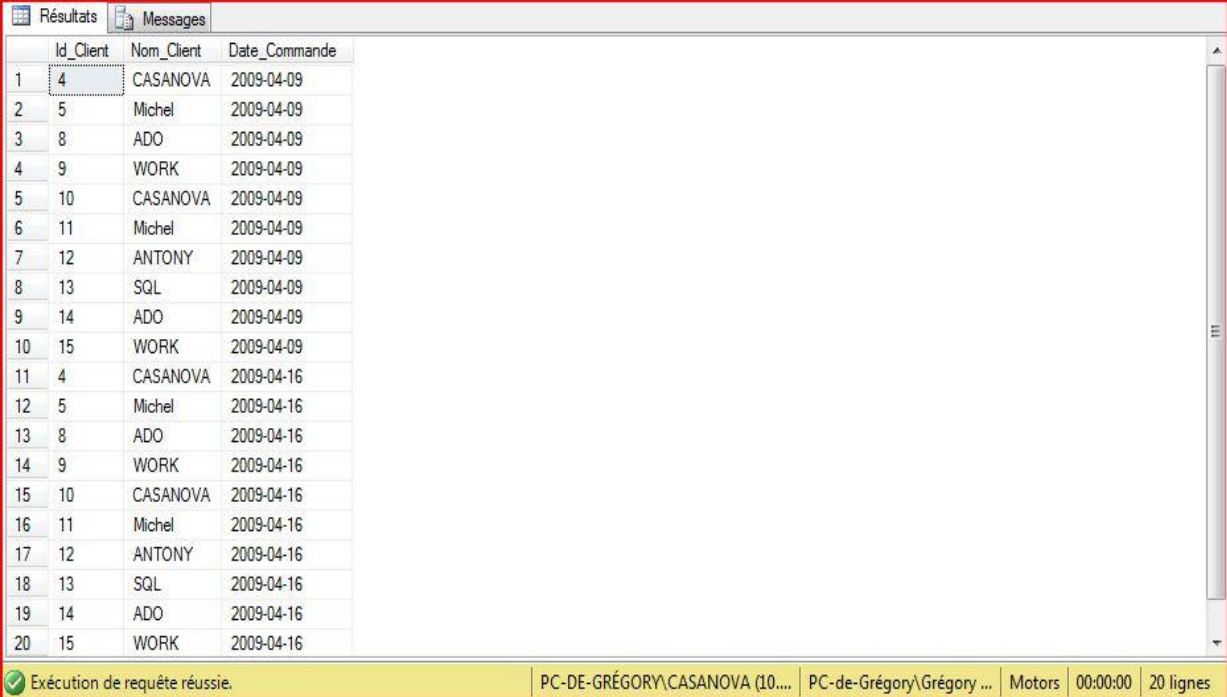
Résultats		Messages
NOM COMPLET		
1	CASANOVA grégory	
2	Michel Galabru	
3	ADO DOTNET	
4	WORK FLOWS	
5	CASANOVA grégory	
6	Michel Galabru	
7	ANTONY michel	
8	SQL SERVER	
9	ADO DOTNET	
10	WORK FLOWS	
Exécution de requête réussie.		PC-DE-GRÉGORY\CASANOVA (10....) PC-de-Grégory\Grégory ... Motors 00:00:00 10 lignes

3.3.6 Le produit cartésien

Le but du produit cartésien est de croiser des données de plusieurs tables, de manière à obtenir toutes les combinaisons possibles. Il y aura autant d'enregistrements de retour que le produit du nombre de lignes de chaque table. Donnons un exemple :

```
-----  
-- Il existe deux manières de faire un produit cartésien :  
-- La syntaxe classique :  
-----  
  
SELECT Id_Client, Nom_Client, Date_Commande  
FROM Client, Commande  
  
-----  
-- La syntaxe en SQL ANSI :  
-----  
  
SELECT Id_Client, Nom_Client, Date_Commande  
FROM Client CROSS JOIN Commande
```

Le résultat est le suivant :



	Id_Client	Nom_Client	Date_Commande
1	4	CASANOVA	2009-04-09
2	5	Michel	2009-04-09
3	8	ADO	2009-04-09
4	9	WORK	2009-04-09
5	10	CASANOVA	2009-04-09
6	11	Michel	2009-04-09
7	12	ANTONY	2009-04-09
8	13	SQL	2009-04-09
9	14	ADO	2009-04-09
10	15	WORK	2009-04-09
11	4	CASANOVA	2009-04-16
12	5	Michel	2009-04-16
13	8	ADO	2009-04-16
14	9	WORK	2009-04-16
15	10	CASANOVA	2009-04-16
16	11	Michel	2009-04-16
17	12	ANTONY	2009-04-16
18	13	SQL	2009-04-16
19	14	ADO	2009-04-16
20	15	WORK	2009-04-16

Nous obtenons 20 enregistrements, ce qui est concluant puisque les deux tables contiennent respectivement 10 et 2 enregistrements. Les deux syntaxes, ANSI ou classique, retournent bien évidemment le même résultat.

3.3.7 Les jointures

Une jointure est un produit cartésien avec une restriction. Une jointure permet d'associer logiquement des lignes de tables différentes. Les jointures sont généralement (pour des raisons de performances) utilisées pour mettre en relation les données de lignes comportant une clé étrangère avec les données de lignes comportant une clé primaire. Voyons-le en détail avec un exemple concret :

```
-----  
-- Il existe deux manières de faire une jointure :  
-- La syntaxe classique :  
-----  
  
SELECT Client.Id_Client, Date_Commande  
FROM Client, Commande  
WHERE Client.Id_Client = Commande.Id_Client  
-----  
-- La syntaxe SQL ANSI :  
-----  
  
SELECT Client.Id_Client, Date_Commande  
FROM Client INNER JOIN Commande  
ON Client.Id_Client = Commande.Id_Client
```

Le résultat est le suivant et il est le même pour les deux instructions SQL :



	Id_Client	Date_Commande
1	1	2009-06-04
2	3	2009-06-04
3	2	2009-06-04
4	1	2009-06-04

3.3.7.1 Les jointures externes

Les jointures externes sont des jointures dans lesquelles la condition est fausse. Dans ce cas, le résultat retourné sera celui d'une des deux tables. Le résultat sera celui de la première table citée si on utilise l'option `LEFT`, et celui de la seconde table citée si l'on utilise l'option `RIGHT`. La syntaxe est la suivante :

```
SELECT Client.Id_Client, Date_Commande  
FROM Client LEFT OUTER JOIN Commande  
ON Client.Id_Client = Commande.Id_Client  
  
SELECT Client.Id_Client, Date_Commande  
FROM Client RIGHT OUTER JOIN Commande  
ON Client.Id_Client = Commande.Id_Client
```

Et le résultat est le suivant :

	Id_Client	Date_Commande
1	1	2009-06-04
2	1	2009-06-04
3	2	2009-06-04
4	3	2009-06-04
5	5	NULL
6	6	NULL

	Id_Client	Date_Commande
1	1	2009-06-04
2	3	2009-06-04
3	2	2009-06-04
4	1	2009-06-04

On remarque alors clairement que suivant qu'on utilise l'option `RIGHT` ou `LEFT`, le résultat est différent, et qu'il respecte le comportement décrit auparavant. Les valeurs `NULL` présentes dans le premier résultat sont dues au fait que les clients dont l'Id est 5 et 6 n'ont pas de commandes. Ces valeurs `NULL` disparaissent dans le second résultat, tout simplement parce qu'il n'existe pas de commande qui n'a pas de client, alors que l'inverse existe. En revanche, Il est obligatoire d'utiliser les jointures externes avec la syntaxe ANSI, c'est pourquoi je vous recommande d'apprendre les jointures selon le modèle ANSI et non le modèle classique, bien que le modèle classique soit plus logique. Dans les versions antérieures, le modèle classique était supporté grâce aux signes `*=` et `=*`, mais ceci ne sont plus supportés sous SQL Server 2008.

3.3.8 La clause `ORDER BY`

La clause `ORDER BY` est utilisée dans une instruction `SELECT` pour trier les données d'une table (ou plusieurs tables) en fonction d'une ou plusieurs colonnes. Par défaut, le rangement se fera par ordre croissant ou par ordre alphabétique. Avec le mot clé `ASC`, le rangement se fera dans l'ordre ascendant. Avec le mot clé `DESC`, le rangement se fera dans l'ordre descendant. Prenons un exemple :

```
--Rangement dans l'ordre ascendant :

SELECT Client.Id_Client, Client.Nom_Client, Date_Commande
FROM Client LEFT OUTER JOIN Commande
ON Client.Id_Client = Commande.Id_Client
ORDER BY Nom_Client ASC
GO

--Rangement dans l'ordre descendant :

SELECT Client.Id_Client, Client.Nom_Client, Date_Commande
FROM Client LEFT OUTER JOIN Commande
ON Client.Id_Client = Commande.Id_Client
ORDER BY Nom_Client DESC
GO
```

Avec la clause `ORDER BY`, nous obtiendrons le même résultat que précédemment, trié dans un ordre différent : les enregistrements sont triés selon le champ `Nom_Client` de façon croissante pour le premier lot, de façon décroissante pour le second lot. Le résultat est le suivant :

	Id_Client	Nom_Client	Date_Commande
1	1	CASANOVA	2009-06-04
2	1	CASANOVA	2009-06-04
3	3	DOLLON	2009-06-04
4	6	HOLLEBEC	NULL
5	2	RAVAILLE	2009-06-04
6	5	VASSELON	NULL

	Id_Client	Nom_Client	Date_Commande
1	5	VASSELON	NULL
2	2	RAVAILLE	2009-06-04
3	6	HOLLEBEC	NULL
4	3	DOLLON	2009-06-04
5	1	CASANOVA	2009-06-04
6	1	CASANOVA	2009-06-04

Les enregistrements sont bien rangés dans l'ordre inverse, suivant la colonne Nom_Client.

3.3.9 L'opérateur UNION

L'opérateur **UNION** va nous permettre d'obtenir un ensemble de ligne provenant de plusieurs requêtes différentes. Toutes les requêtes doivent fournir le même nombre de colonnes avec les mêmes types de données pour chaque colonne (correspondance deux à deux).

```
SELECT Id_Stock, Quantite
FROM Stock

UNION

SELECT Id_Stock, Quantite
FROM Commande
```

Le résultat est le suivant :

	Id_Stock	Quantite
1	1	1
2	1	24
3	2	15
4	3	1
5	3	19
6	4	2
7	4	298
8	5	1
9	5	229
10	6	10
11	7	10
12	8	25
13	9	30
14	10	10

Les résultats des tables sont associés, et les enregistrements s'ajoutent. La première table citée dans la première instruction `SELECT` sera associée à la première table citée dans la seconde instruction `SELECT`, de même pour les secondes tables. Dans le résultat, on obtient alors 2 colonnes au lieu de 4. Une option est possible avec l'opérateur `UNION`, `UNION ALL` qui va permettre de retourner toutes les lignes résultats, même celles qui seront en double. Il est bon de savoir que lorsque cet opérateur n'est pas précisé, les lignes dupliquées ne sont retournées qu'une seule fois.

3.3.10 L'opérateur EXCEPT

L'opérateur `EXCEPT` permet d'extraire d'une solution les éléments que l'on ne veut pas y retrouver, c'est-à-dire, enlever une valeur précise ou un domaine que l'on ne veut pas retrouver dans notre solution finale. Il est donc évident que si on exclut des valeurs, les deux expressions `SELECT` séparées par le mot clé `EXCEPT` doivent avoir le même nombre de colonnes en argument. Prenons un exemple :

```
SELECT Id_Stock, Quantite
FROM Stock

EXCEPT

SELECT Id_Stock, Quantite
FROM Stock
WHERE Id_Stock = 3
```

Ici, on sélectionnera les colonnes `Id_Stock` et `Quantite` de la table `Stock`, excepté celle pour lesquelles `Id_Stock` est égal à 3.

3.3.11 L'opérateur INTERSECT

Grace à cet opérateur, il va être possible d'identifier en une seule requête, des lignes d'informations simultanément présentes dans deux jeux de résultats distincts, mais de mêmes structures.

```
SELECT * FROM Client
WHERE Id_Client BETWEEN 1 AND 3

SELECT * FROM Client
WHERE Prenom_Client = 'Julien'

SELECT * FROM Client
WHERE Id_Client BETWEEN 1 AND 3

INTERSECT

SELECT * FROM Client
WHERE Prenom_Client = 'Julien'
```

Le résultat est le suivant :

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	1	CASANOVA	Grégory	33563456764	31 place de la chance	75554@supinfo.com
2	2	RAVAILLE	James	33567876435	34 Avenue de le paix	James.Ravaille@Domz
3	3	DOLLON	Julien	33563765514	18 Rue du cotton	Jul.Dollon@dotnetfranc

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	3	DOLLON	Julien	33563765514	18 Rue du cotton	Jul.Dollon@dotnetfrance.com

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	3	DOLLON	Julien	33563765514	18 Rue du cotton	Jul.Dollon@dotnetfrance.com

Le jeu de données obtenu donne tous les clients dont l'Id est compris entre 1 et 3, et dont le nom est Julien. L'opérateur **INTERSECT**, fait l'intersection des deux jeux de résultats, et ne donne en sortie, que les valeurs communes aux deux jeux. Dans l'exemple donné, les deux jeux de résultats n'ont en résultat le client dont l'Id est 3. Le résultat final ne donnera donc que le client dont l'Id est 3, comme montré sur l'exemple ci-dessus.

3.3.12 La clause TOP

La close **TOP** permet d'extraire grâce à l'instruction **SELECT**, que les premiers enregistrements de la sélection. Elle est utilisable avec les instructions **INSERT**, **UPDATE**, **DELETE**. Prenons un exemple avec l'instruction **SELECT** :

```
SELECT TOP 5 *
FROM dbo.Client
```

Cette instruction permet de sélectionner les 5 premiers enregistrements de la table Client, dans l'ordre de lecture des enregistrements dans la table. Si nous spécifions la clause **ORDER BY**, alors les enregistrements sélectionnés respectent cet ordre de tri.

```
SELECT TOP 50 PERCENT WITH TIES *
FROM dbo.Client
ORDER BY Nom Client
```

Cette instruction permet de sélectionner 50% des enregistrements dans l'ordre de lecture des enregistrements. Dans le cas où nous avons utilisé un pourcentage, la close **WITH TIES** ne s'utilise que si une close **ORDER BY** est appliquée au **SELECT**. Elle a pour effet de ne sélectionner les enregistrements qu'après la mise en leur tri.

3.3.13 Créer une table grâce à SELECT INTO

Il est possible de créer une table à l'aide de colonnes de tables déjà existantes. Grâce à un simple **SELECT INTO**, nous aurons à choisir les colonnes qui constitueront les champs de la nouvelle table. Toutes les closes et conditions disponibles pour l'instruction **SELECT** sont applicables pour l'instruction **SELECT INTO**. Voici un exemple :

```
SELECT Id_Client, Nom_Client, Id_Commande
INTO dbo.Exemple
FROM dbo.Client, dbo.Commande
WHERE Client.Id_Client = Commande.Id_Client
```

Dans l'instruction précédente, la clause **INTO** permet de préciser que nous allons créer une table, ici, `dbo.Exemple`, et que nous allons ajouter les lignes trouvées dans l'instruction **SELECT**, à l'intérieur de cette nouvelle table. Il est utile de préciser que la table n'est pas définie en tant que tel par la clause **INTO**, mais plus par le **SELECT**, car c'est cette instruction qui va donner à la table ses caractéristiques (nombre de colonnes, type de données des colonnes...). Dans le cas où des colonnes sont calculées, il est impératif de donner un nom à ces colonnes. Si l'on fait précéder le nom de la table créée par un #, la table sera temporaire locale, si elle est précédée d'un ##, elle sera temporaire globale. On rappelle que ces deux types de tables temporaires sont stockés dans la base de données Tempdb qui est une table prédéfinie en tant que table système dans SQL Server 2008. Les tables temporaires locales sont accessibles que par la session qui l'a créée et disparaît à la déconnexion alors que les tables globales, elles sont ensuite accessibles par toutes les sessions, et enfin elles sont détruites lors de la déconnexion de la dernière session à l'avoir utilisée. Ce genre de table est pratique, pour des travaux de transferts de données, ou encore si nous avons besoin de garder les données contenues dans une table, tout en voulant supprimer la structure de la table en question.

3.3.14 La clause COMPUTE et COMPUTE BY

La clause **COMPUTE** est utilisée à la suite de la clause **ORDER BY**, afin de retourner un sous résultat, en rapport avec le résultat principal. Le sous résultat est obligatoirement généré par une fonction d'agrégation telle que **COUNT**, **SUM**... Il est bon de noter que ces clauses sont maintenues pour des raisons de compatibilités, mais sont vouées à disparaître dans les versions futures. L'exemple suivant retourne un résultat principal, et un sous résultat. Le résultat principal sélectionne toutes les colonnes de la table Client, ordonnées par le nom des clients, tandis que le sous résultat va compter le nombre de client. L'intérêt de la clause **COMPUTE** est de pouvoir générer un sous résultat, grâce à une même requête.

```
SELECT *
FROM Entreprise.dbo.Client
ORDER BY Nom_Client
COMPUTE COUNT(Id_Client)
```

Résultats		Messages				
	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	1	CASANOVA	Grégory	33563456764	31 place de la chance	75554@supinfo.com
2	3	DOLLON	Julien	33563765514	17 Rue du coton	Jul.Dollon@dotnetfrance.com
3	6	HOLLEBEC	Mathieu	33578763450	26 place des canons	MathHol@votredomaine.com
4	2	RAVAILLE	James	33567876435	34 Avenue de la paix	James.Ravaille@Domaine.fr
5	5	VASSELON	Jean Christophe	33567654342	19 avenue du sac	JCVD@leursdomaine.com
6	4	VERGNAULT	Bertrand	33567654323	26 rue de la vie	Bertrand@Sondomaine.com
cnt						
1	6					

Le mot clé **BY** de la clause **COMPUTE**, nous permet de retourner les sous résultats en fonction des différentes valeurs d'une colonne spécifique. Dans l'exemple, à la suite, on peut remarquer que l'on donne la quantité du stock, pour chaque `Id_Stock` en sous résultat. La colonne que l'on précise donc après le mot clé **BY**, nous permet de dire, de quelle manière nous allons découper les sous résultats.

```
SELECT *
FROM Entreprise.dbo.Stock
ORDER BY Id_Stock
COMPUTE SUM(Quantite) BY Id_Stock
```

Id_Stock	Libelle	Quantite	Mesure	Id_Entrepos
1	Echelle 2m	25	Unité	1
sum				
1		25		
1	Echelle 3m	15	Unité	1
sum				
1		15		
1	Brique	20	Palette	2
sum				
1		20		

3.3.15 Les opérateurs ROLLUP et CUBE

Les opérateurs **ROLLUP** et **CUBE** sont utilisés avec la clause **GROUP BY**, dans le but d'obtenir des lignes supplémentaires affichant les calculs de la fonction.

3.3.15.1 L'opérateur ROLLUP

La clause **WITH ROLLUP** permet de créer des lignes comportant des résultats pour le groupement des colonnes contenues dans la clause **GROUP BY**, en les combinants de la gauche vers la droite.

```
USE Entreprise
GO

SELECT a.Id_Entrepos, b.Id_Stock
FROM dbo.Entrepos a
INNER JOIN dbo.Stock b
ON a.Id_Entrepos = b.Id_Entrepos
GROUP BY a.Id_Entrepos, b.Id_Stock
WITH ROLLUP
```

Id_Entrepos	Nombre éléments
1	8
2	8
3	8
4	8
5	8
6	8
7	8
8	8
9	6
10	6
11	6

3.3.15.2 L'opérateur CUBE

L'opérateur **CUBE** permet de créer des résultats pour toutes les combinaisons possibles des colonnes contenues dans la clause **GROUP BY**.

```
USE Entreprise
GO

SELECT a.Id_Entrepos, b.Id_Stock
FROM dbo.Entrepos a
INNER JOIN dbo.Stock b
ON a.Id_Entrepos = b.Id_Entrepos
GROUP BY a.Id_Entrepos, b.Id_Stock
WITH CUBE
```

	Id_Entrepos	Id_Stock
1	1	1
2	NULL	1
3	1	2
4	NULL	2
5	2	3
6	NULL	3
7	2	4
8	NULL	4
9	2	5
10	NULL	5
11	1	6
12	NULL	6
13	3	7
14	NULL	7
15	3	8
16	NULL	8

3.3.16 L'opérateur OVER

L'opérateur **OVER** permet de partitionner les données ou encore de les trier avant d'appliquer une fonction de calcul d'agrégat par exemple (Voir les fonctions dans ce chapitre), ou encore les fonctions de tri tel que ROW_NUMBER, NTILE, que nous verrons plus tard, ou encore DENSE_RANK. Dans le cas d'une fonction de tri, l'opérateur **OVER** va pouvoir contenir un partitionnement ou une clause **ORDER BY**, ce qui va nous permettre de ranger les données avant d'effectuer une fonction. Il est important de noter que les fonctions d'agrégation ne sont applicables avec un **OVER** dans le seul cas d'un partitionnement. Prenons un exemple :

```
SELECT a.Id_Entrepos, COUNT(b.Id_Stock)
OVER (PARTITION BY b.Id_Entrepos) AS 'Nombre éléments'
FROM Entrepos a
INNER JOIN Stock b
ON a.Id_Entrepos = b.Id_Entrepos
```

	Id_Entrepos	Nombre éléments
1	1	8
2	1	8
3	1	8
4	1	8
5	1	8
6	1	8
7	1	8
8	1	8
9	2	6
10	2	6
11	2	6

3.3.17 L'opérateur NTILE

Cette fonction est utilisée en addition à **OVER**, et permet de diviser la partition en différents groupes de données équilibrées. **NTILE** s'utilise avec une clause **ORDER BY** de la façon suivante :

```
SELECT a.Id_Client, Nom_Client, Prenom_Client, Adresse_Client,
       Id_Commande, Date_Commande,
       NTILE(5) OVER (PARTITION BY Nom_Client, Prenom_Client ORDER BY
                       Date_Commande) AS 'Ensemble'
FROM Entreprise.dbo.Client a
INNER JOIN Entreprise.dbo.Commande b
ON a.Id_Client = b.Id_Client
```

3.3.18 Les sous-requêtes

Il est possible d'imbriquer une requête **SELECT** dans une requête **SELECT** (**UPDATE** ou **DELETE**). Les sous requêtes peuvent être utilisées avec les clauses **HAVING** ou **WHERE**. Il existe trois types de sous requêtes différentes :

- Les sous requêtes qui ne renvoient qu'une seule valeur unique (sous-requête scalaire) :

```
SELECT Id_Client FROM Motors.dbo.Client
WHERE Id_Client = (SELECT Id_Client FROM Motors.dbo.Client WHERE
                  Id_Client = 10)
```

- Les requêtes renvoyant une liste d'enregistrements. Elles sont utilisées avec **IN**, **EXIST**, **ANY**, **SOME** ou encore **ALL** :

```
SELECT *
FROM Motors.dbo.Client
WHERE EXISTS (SELECT * FROM Motors.dbo.Client WHERE Id_Client = 4)
```

- La sous requête externe utilisée au travers de la clause **WHERE**, fait référence à une table de la requête interne. Dans ce cas là, la requête externe est exécutée pour chaque ligne extraite de la requête interne.

```
SELECT Nom_Client
FROM Motors.dbo.Client
WHERE EXISTS (SELECT Id_Client_Commande, Id_Client
                FROM Motors.dbo.Commande INNER JOIN Motors.dbo.Client
                ON Id_Client = Id_Client_Commande)
```

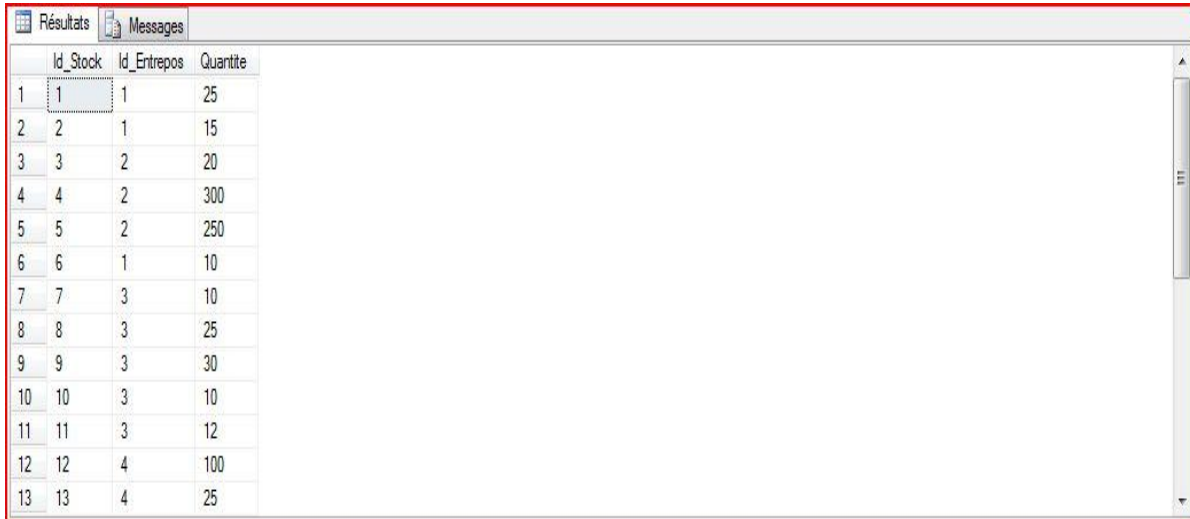
3.3.19 Les instructions PIVOT et UNPIVOT

Ces instructions sont très puissantes et sont faciles à utiliser. L'instruction **PIVOT** aura la capacité de transformer un résultat présenté sous forme de ligne en colonne et **UNPIVOT** aura la capacité inverse. La clause **PIVOT** fait partie de la clause **FROM** de l'instruction **SELECT**. L'utilisation de **PIVOT** va permettre la création d'un pseudo table interne à la requête. On pourra donc lui assigner un alias avec la clause **AS** si nécessaire. Le résultat qui suit est la sélection de toutes les colonnes de notre table **STOCKS**. La requête **SELECT** suivante nous servira de comparaison.

```
USE Entreprise
GO

SELECT Id_Stock, Id_Entrepos, Quantite
FROM dbo.Stock
```

Voici son résultat :



	Id_Stock	Id_Entrepos	Quantite
1	1	1	25
2	2	1	15
3	3	2	20
4	4	2	300
5	5	2	250
6	6	1	10
7	7	3	10
8	8	3	25
9	9	3	30
10	10	3	10
11	11	3	12
12	12	4	100
13	13	4	25

Avec le code qui va suivre, nous allons nous proposer d'améliorer la lisibilité de notre résultat en affichant, grâce à un `PIVOT`, la quantité en fonction des dépôts (1, 2, 3 ou 4). Il suffit de faire un `SELECT` des valeurs que nous voulons passer en colonne. Les alias présents dans l'exemple servent évidemment à donner un nom aux colonnes créées, car par défaut, elles n'ont pas de noms. Pour l'instruction `PIVOT`, comme pour l'instruction `UNPIVOT`, nous allons dans un premier temps appliquer une fonction d'agrégation à la colonne passée en paramètre de la colonne de pivot et la colonne par laquelle nous allons effectuer le pivot après la clause `FOR`. La clause `IN` indiquera simplement les valeurs pour lesquelles nous allons effectuer le pivot. Il est important de remarquer que l'alias que nous donnons au pivot n'est pas optionnel. Si vous n'en donnez pas, une erreur sera levée.

```
USE Entreprise
GO

SELECT Id_Stock,
[1] AS "D1", [2] AS "D2", [3] AS "D3", [4] AS "D4"
FROM dbo.Stock
PIVOT (SUM(Quantite) FOR Id_Entrepos
IN ([1],[2],[3],[4])) AS PVT
```

Le résultat est le suivant :

	Id_Stock	D1	D2	D3	D4
1	1	25	NULL	NULL	NULL
2	2	15	NULL	NULL	NULL
3	3	NULL	20	NULL	NULL
4	4	NULL	300	NULL	NULL
5	5	NULL	250	NULL	NULL
6	6	10	NULL	NULL	NULL
7	7	NULL	NULL	10	NULL
8	8	NULL	NULL	25	NULL
9	9	NULL	NULL	30	NULL
10	10	NULL	NULL	10	NULL
11	11	NULL	NULL	12	NULL
12	12	NULL	NULL	NULL	100
13	13	NULL	NULL	NULL	25

3.3.20 L'instruction MERGE

L'instruction **MERGE** permet en une action Transact SQL, de modifier, ajouter, ou même supprimer sur une même table de destination, si la condition est respectée. On pourra alors grâce à une instruction **MERGE**, modifier des tours à tour, chaque ligne de notre table en fonction d'une autre table. Voici un exemple de structure de l'instruction **MERGE** :

```
USE Entreprise
GO

MERGE INTO dbo.Stock
USING dbo.Commande
ON Stock.Id_Stock = Commande.Id_Stock
WHEN MATCHED THEN
UPDATE
SET Stock.Quantite = Stock.Quantite - Commande.Quantite;
```

Description de la requête précédente : On se propose de modifier la table Stock avec la table Commande, avec pour condition d'arrêt, le fait que : `Stock.Id_Stock = Commande.Id_Stock`. Les mots clés **WHEN MATCHED THEN** vont permettre de dire, si la condition d'arrêt est respectée, alors on fait l'instruction qui suit le **THEN**. Ici, on soustraira à la quantité du stock, la quantité de la commande passée, pour chaque Id_Stock. Il est possible d'utiliser les mots clés **WHEN NOT MATCHED THEN**, qui vont nous permettre de modifier les lignes de la table cible pour lesquelles la condition d'arrêt n'est pas vraie.

4 Le SQL Procédural

4.1 Les variables

4.1.1 Les variables utilisateur

Une variable est une zone mémoire caractérisée par un type et un nom, et permettant de stocker une valeur respectant le type. Dans SQL Server, les variables doivent être obligatoirement déclarées avant d'être utilisées.

Voici la déclaration d'une variable nommée *Id_Client* de type *Int* :

```
DECLARE @IdClient int
```

L'instruction suivante permet de valoriser cette variable via l'exécution d'une requête scalaire :

```
SELECT @IdClient = (SELECT Id_Client FROM Motors.dbo.Client WHERE
```

4.1.2 Les variables système

Les variables système sont définies par le système et ne peuvent être disponibles qu'en lecture. Elles se différencient syntaxiquement des variables utilisateur par le double @. L'exemple le plus courant est la variable @@ERROR, qui est à 0 en temps normal, et à 1 lorsqu'une erreur est levée.

4.2 Les transactions

Une transaction est caractérisée par le mot l'acronyme **ACID** (Atomic Consistency Isolation Durability) :

- **Atomique** car la transaction constitue une unité indivisible de travail pour le serveur.
- **Consistance** car à la fin d'une transaction, les données montrées sont soit celles d'avant transaction (dans le cas d'une annulation de la transaction) soit celle d'après transaction (dans le cas d'une validation).
- **Isolation**, car il est possible de verrouiller (isoler) les données pendant l'exécution de la transaction (verrouillage en lecture, en écriture, ...).
- **Durée** car les changements apportés sur des données par une transaction sont durables (non volatiles).

La syntaxe générique d'une transaction est la suivante :

```
BEGIN TRAN nom_transaction
--Démarrage de la transaction

COMMIT TRAN nom_transaction
--Validation de la transaction

SAVE TRAN nom_point_de_retour
--Déclaration d'un point de contrôle de la transaction

ROLLBACK TRAN nom_transaction OR nom_point_de_controle
--Annulation de la transaction
```

Voici un exemple de transaction :

```
BEGIN TRAN Transaction1
    UPDATE dbo.Client
        SET Nom_Client = 'ANDREO'
        WHERE Nom_Client = 'CASANOVA'

BEGIN TRAN Transaction2
    UPDATE dbo.Client
        SET Nom_Client = 'VASSELON'
        WHERE Nom_Client = 'HOLLEBECQ'
    COMMIT TRAN Transaction2
ROLLBACK TRAN Transaction1
```

Ici, dans notre exemple, nous avons deux transactions imbriquées. Il est très important de comprendre qu'une transaction est une unité indissociable, et que par conséquent, il est nécessaire de terminer par un `COMMIT` ou `ROLLBACK`, la dernière transaction en date. La fermeture des transactions se fait donc selon un modèle LIFO (Last In First Out). La dernière transaction écrite sera la première à devoir être fermée. Pour revenir à notre exemple, on peut désormais dire que le nom client égal à HOLLEBECQ sera changé par VASSELON, du fait du `ROLLBACK TRAN` qui termine la transaction 2, alors que CASANOVA ne sera pas changé par ANDREO, dans transaction1, car celle-ci se termine par un `ROLLBACK TRAN`.

Note Importante : Les instructions du DML doivent automatiquement comporter un `ROLLBACK TRAN` pour être prises en compte et être appliquées, alors que les instructions du DDL comportent un `ROLLBACK TRAN` implicite qui est opéré juste après que l'instruction du DDL soit faite. Il faut donc faire très attention à la suite d'instructions dans une transaction. Si jamais vous écrivez une transaction qui comporte deux instructions, une du DML puis une du DDL, même si vous mettez un `ROLLBACK` à la suite, les deux instructions seront « COMMIT », puisque les instructions du DDL comportent ce `ROLLBACK TRAN` implicite dont nous avons parlé précédemment.

4.3 Les lots et les scripts

Un lot est une suite de transactions et d'instructions qui seront exécutées en un seul et unique bloc. Un lot se termine par l'instruction `GO`. L'intérêt des lots réside dans les performances. Il faut bien entendu prendre en compte qu'une simple erreur de syntaxe fera que tout votre lot ne s'exécutera pas. En revanche, les lots possèdent certaines restrictions :

- Il est **impossible** d'utiliser deux des instructions suivantes, ensemble dans un même lot :
CREATE PROCEDURE, CREATE RULE, CREATE DEFAULT, CREATE TRIGGER, CREATE VIEW.
- Il **n'est pas possible** d'agir sur des définitions de colonnes ou d'agir sur une modification opérée dans un même lot.
- Il **n'est pas possible** de supprimer et de recréer un même objet dans un même lot.

Un script est un ensemble de lots, qui peut être enregistré dans un fichier dont l'extension est .sql. Comme exemple de script, vous avez le fichier **CoursSqlServer.sql**, disponible en annexe de ce cours, qui contient la structure de la base, des tables, certaines entrées de données et certains objets de la base tels qu'une procédure stockée ou un déclencheur...

4.4 Le contrôle de flux

Il existe quatre façons de contrôler les flux sur SQL Server 2008. Les instructions **RETURN**, **PRINT**, **CASE** et les blocs **BEGIN...END**, dans lesquels peuvent être contenus les structures de test **IF** et les boucles **WHILE**. Toutes ces instructions vont vous permettre de mettre en valeur vos données en les rendant plus présentables, ou bien, elles vous permettront de les manipuler avec plus de facilité, par exemple pour des actions répétitives, ou des actions nécessitant une condition. Dans cette partie, nous allons détailler tous les contrôles de flux possibles.

4.4.1 L'instruction RETURN

L'instruction **RETURN** vous permet de sortir d'une instruction ou d'une procédure sans condition particulière, en renvoyant ou non une valeur entière.

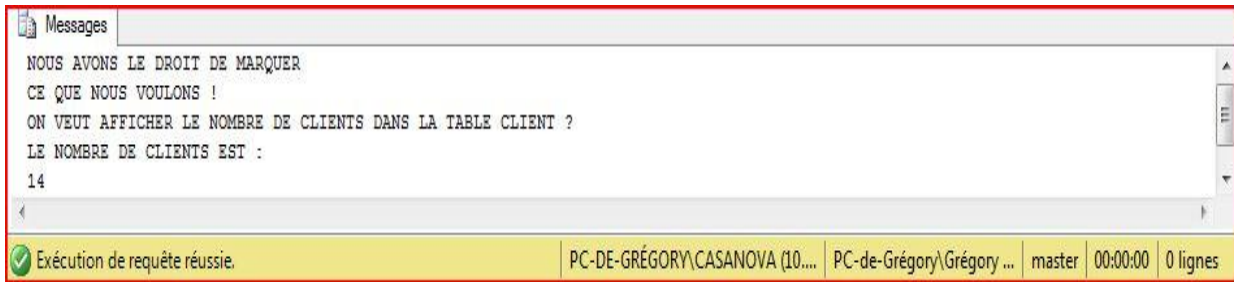
```
CREATE PROC Procedure1
AS
DECLARE @Variable int = 4
IF (@Variable > 2)
    RETURN 0;
ELSE
    RETURN 1;
GO
```

4.4.2 L'instruction PRINT

L'instruction **PRINT** est l'instruction d'affichage de message. Prenons un exemple :

```
PRINT 'NOUS AVONS LE DROIT DE MARQUER'
PRINT 'CE QUE NOUS VOULONS !'
PRINT 'ON VEUT AFFICHER LE NOMBRE DE CLIENTS DANS LA TABLE CLIENT ?'
DECLARE @Variable int
SELECT @Variable = COUNT(*) FROM Motors.dbo.Client
PRINT 'LE NOMBRE DE CLIENTS EST : '
PRINT @Variable
```

Lors de l'exécution, les traces suivantes sont affichées dans la fenêtre Messages :



4.4.3 L'instruction CASE

L'instruction CASE, permet d'attribuer des valeurs en fonction d'une condition. Voici un exemple :

```
USE Entreprise
SELECT 'Anciennete' = CASE Id_Client
    WHEN '3' THEN 'ANCIEN'
    WHEN '2' THEN 'PAS SI VIEUX'
    WHEN '1' THEN 'RECENT'
    ELSE 'ON SAIT PAS TROP'
END,
Id_Client, Nom_Client
FROM dbo.Client
ORDER BY Anciennete
```

Avec un case, on peut créer simplement une colonne en donnant des conditions pour les résultats, en fonction d'une autre colonne existante. Par exemple, ici, on détermine suivant l'Id_Client, si le client est Ancien, Pas si vieux, Récent, ou si l'on ne sait pas.

Son résultat d'exécution est le suivant :

	Anciennete	Id_Client	Nom_Client
1	ANCIEN	3	DOLLON
2	ON SAIT PAS TROP	4	VERGNAULT
3	ON SAIT PAS TROP	5	VASSELON
4	ON SAIT PAS TROP	6	HOLLEBEC
5	PAS SI VIEUX	2	RAVAILLE
6	RECENT	1	CASANOVA

4.4.4 Les blocs BEGIN ... END

Les blocs délimitent une série d'instructions, et ils peuvent être utilisés avec les conditions **IF** et les boucles **WHILE**. La structure générique est la suivante :

```
BEGIN
    --Les blocs peuvent contenir
    --Des instructions ou bien d'autres blocs
END
```

4.4.4.1 La condition IF

La structure de condition **IF** permet de poser une condition à une instruction. Si la condition est vraie, l'instruction sera exécutée. Dans le cas contraire, elle ne le sera pas. Voici un exemple d'utilisation de cette instruction :

```
SELECT * FROM Client

DECLARE @Variable int = 1
IF EXISTS (SELECT * FROM Client WHERE Id_Client = @Variable)
BEGIN
    DELETE FROM Client WHERE Id_Client = @Variable
    PRINT 'Le Client 11 a bien été supprimé !'
END
ELSE
    PRINT 'Pas de Client pour cet Id !'

SELECT * FROM Client
```

Dans ce script, on déclare dans un premier temps une variable `@Variable` de type `int`, et de valeur 1. On applique alors une condition **IF**, qui définit que s'il existe un client avec un `Id` égal à la valeur de notre variable déclarée préalablement, on le supprime et on écrit que le client a bien été supprimé. L'instruction **ELSE** définit en revanche que pour tous les autres cas, on écrit que le client n'existe pas.

Le résultat est le suivant dans le cas où le client à l'Id 1 existe :

Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	CASANOVA	Grégory	33563456764	31 place de la chance	75554@supinfo.com
2	RAVILLE	James	33567876435	34 Avenue de le paix	James.Ravaille@Domaine.fr
3	DOLLON	Julien	33563765514	17 Rue du cotton	Jul.Dollon@dotnetfrance.com
4	VERGNAULT	Bertrand	33567654323	26 rue de la vie	Bertrand@Sondomaine.com
5	VASSELON	Jean Christophe	33567654342	19 avenue du sac	JCVD@leursdomaine.com
6	HOLLEBEC	Mathieu	33578763450	26 place des canons	MathHol@votredomaine.com

Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
2	RAVILLE	James	33567876435	34 Avenue de le paix	James.Ravaille@Domaine.fr
3	DOLLON	Julien	33563765514	17 Rue du cotton	Jul.Dollon@dotnetfrance.com
4	VERGNA...	Bertrand	33567654323	26 rue de la vie	Bertrand@Sondomaine.com
5	VASSEL...	Jean Christo...	33567654342	19 avenue du sac	JCVD@leursdomaine.com
6	HOLLEBEC	Mathieu	33578763450	26 place des canons	MathHol@votredomaine.com

Dans l'onglet Messages du résultat de la requête, le message suivant est alors apparu :

```
(14 ligne(s) affectée(s))

(1 ligne(s) affectée(s))
Le Client 11 a bien été supprimé !

(13 ligne(s) affectée(s))
```

Le résultat est le suivant lorsque le client à l'Id 1 n'existe pas ou plus :

Résultats		Messages				
	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	2	RAVAILLE	James	33567876435	34 Avenue de le paix	James.Ravaille@Domaine.fr
2	3	DOLLON	Julien	33563765514	17 Rue du cotton	Jul.Dollon@dotnetfrance.com
3	4	VERGNAULT	Bertrand	33567654323	26 rue de la vie	Bertrand@Sondomaine.com
4	5	VASSELON	Jean Christophe	33567654342	19 avenue du sac	JCVD@leursdomaine.com
5	6	HOLLEBEC	Mathieu	33578763450	26 place des canons	MathHol@votredomaine.com

	Id_Client	Nom_Client	Prenom_Client	Numero_Client	Adresse_Client	Mail_Client
1	2	RAVAILLE	James	33567876435	34 Avenue de le paix	James.Ravaille@Domaine.fr
2	3	DOLLON	Julien	33563765514	17 Rue du cotton	Jul.Dollon@dotnetfrance.com
3	4	VERGNAULT	Bertrand	33567654323	26 rue de la vie	Bertrand@Sondomaine.com
4	5	VASSELON	Jean Christophe	33567654342	19 avenue du sac	JCVD@leursdomaine.com
5	6	HOLLEBEC	Mathieu	33578763450	26 place des canons	MathHol@votredomaine.com

Dans l'onglet message de la partie résultat, on obtient le message suivant :

```
(13 ligne(s) affectée(s))
Pas de Client pour cet Id !

(13 ligne(s) affectée(s))
```

4.4.4.2 La boucle WHILE

L'instruction **WHILE** est une structure algorithmique permettant d'exécuter un bloc d'instructions de manière répétitive, en fonction d'une condition. Tant que la condition est vraie, ce bloc d'instructions sera exécuté. Dans la syntaxe de la structure **WHILE**, deux instructions sont à connaître : l'instruction **BREAK** et l'instruction **CONTINUE**. La première permet de sortir de la structure en interrompant son exécution. La seconde nous permet de relancer immédiatement l'exécution du bloc d'instruction. Voici un exemple :

```
WHILE (SELECT COUNT(*) FROM Client) < 6
BEGIN
    INSERT INTO [Entreprise].[dbo].[Client]
        ([Nom_Client]
        , [Prenom_Client]
        , [Numero_Client]
        , [Adresse_Client]
        , [Mail_Client])
    VALUES
        ('DORDOLO',
        'Mathieu',
        33678765342,
        '9 Avenue des Peupliers',
        'MD@domaine.com')
END
```

Cet exemple permet d'ajouter des clients afin que la table Client en contienne 6.

4.5 La gestion des curseurs

Dans SQL Server, un curseur est un objet qui nous permet d'exécuter un traitement sur un ensemble d'enregistrements. Les curseurs sont des outils très puissants, mais aussi très gourmands en ce qui concerne les ressources. Il est donc conseillé de modifier des lignes de résultat de manière traditionnelle, avec un simple **UPDATE** ou une autre instruction du DML, afin de consommer le moins de ressources possibles.

- Voici la déclaration d'un curseur :

```
DECLARE Curseur (arguments1) CURSOR  
FOR SELECT  
FOR (arguments2) OF liste de colonnes
```

Argument1 peut être :

- INTENSITIVE: seules les opérations sur la ligne suivante sont permises.
- SCROLL: les déplacements dans les lignes du curseur peuvent se faire dans tous les sens.
- LOCAL : la portée du curseur est locale au lot, c'est-à-dire qu'il peut être utilisé que dans le lot dit.
- GLOBAL : la portée du curseur est globale, c'est-à-dire valable pour toute la connexion.
- FORWARD_ONLY : les données sont extraites du curseur dans leur ordre d'apparition.
- STATIC : une copie des données est faite de façon temporaire dans la base tempdb afin que le curseur ne soit pas affecté par les modifications qui peuvent être faites sur la base.
- SCROLL_LOCKS: garantit le succès des instructions DELETE et UPDATE.
- TYPE_WARNING: permet d'envoyer un message WARNING si des conversions de types implicites sont effectuées.
- KEYSET: les lignes et leur ordre dans le curseur sont fixés au moment de l'ouverture du curseur. Les références de chacune de ces lignes sont conservées dans tempb.
- DYNAMIC: le curseur représente exactement les données présentes dans la base. Donc le nombre de ligne, les valeurs qu'elles contiennent ou encore leur ordre peuvent changer de façon dynamique.
- FAST_FORWARD: permet de définir le curseur comme étant en avant et en lecture seule.
- READ_ONLY: est en lecture seule.

Argument2 peut être :

- UPDATE: Précise que des mises à jour vont être faites sur la table d'origine du curseur.
- READ ONLY: Précise qu'on se place en lecture seule.

- OPEN

Cette instruction permet de rendre le curseur utilisable, et créer des tables temporaires associées. La variable système @@CURSOR_ROWS est valorisée après cette instruction. Sa valeur passe de 0 à 1 après l'instruction OPEN.

Sa syntaxe est la suivante :

```
OPEN (arguments1) curseur_auteurs
```

Argument1 peut être :

- GLOBAL : la portée du curseur est globale, c'est-à-dire valable pour toute la connexion.
- FETCH

C'est l'instruction qui permet d'extraire une ligne du curseur et de valoriser les variables et leur contenu. Après cette instruction, la variable système @@FETCH_STATUS est à 0, si toutefois le **FETCH** c'est bien passé.

```
FETCH (arguments1) (FROM GLOBAL) Nom_Curseur INTO Liste_Variable
```

Argument1 peut être :

- NEXT: Lit la ligne suivante. C'est la seule option possible pour un INSENSITIVE CURSOR.
- PRIOR: Lit la ligne précédente.
- FIRST: Lit la première ligne.
- LAST: Lit la dernière ligne.
- ABSOLUTE p: Lit la Pième ligne de l'ensemble.
- RELATIVE p: Lit la Pième ligne à partir de la ligne courante.

- **CLOSE**

Cette instruction permet la fermeture du curseur et la libération de la place mémoire où il été contenu. Il est important de faire intervenir cette opération dès que possible dans le souci de libérer les ressources.

```
CLOSE Nom_Curseur
```

- **DEALLOCATE**

Cette instruction permet de supprimer le curseur et les ressources associées.

```
DEALLOCATE Nom_Curseur
```

Maintenant que nous avons expliqué la structure et le fonctionnement d'un curseur, nous allons montrer un exemple concret afin de comprendre leur fonctionnement en pratique :

```
DECLARE @Id_Client INT
DECLARE curseur CURSOR FOR
    SELECT Id_Client FROM Client

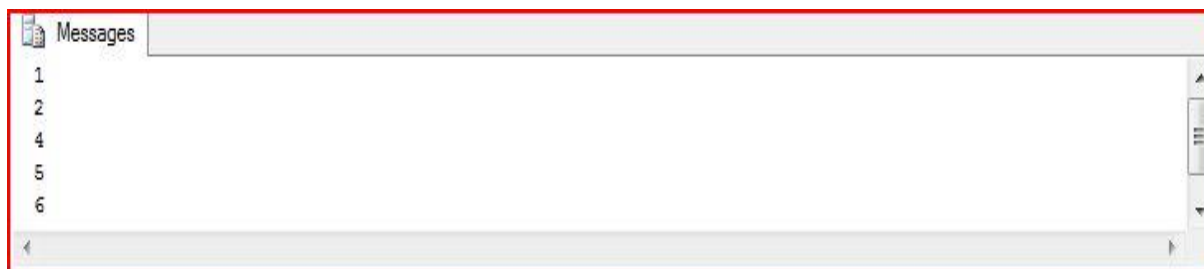
OPEN curseur

FETCH curseur INTO @Id_Client

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @Id_Client
        FETCH curseur INTO @Id_Client
    END

CLOSE curseur
DEALLOCATE curseur
```

Dans ce cas là, le curseur va nous permettre grâce à une boucle WHILE, de parcourir tous les `Id_Client` pour lesquels `@@FETCH_STATUS` sera égal à 0. Cette variable peut prendre trois états, 0, -1, -2, respectivement pour dire que soit l'instruction FETCH c'est déroulé normalement et a réussi, soit pour dire que l'instruction a échoué, sinon pour dire que la ligne recherchée est manquante. En temps normal, cette variable système est initialisée à -1. Après avoir parcouru tous les enregistrements de la table client, il est nécessaire de fermer le curseur et de le dé allouer. Le résultat est le suivant pour notre base de données d'exemple, Entreprise.



4.6 Les exceptions

4.6.1 Lever une exception

Pour chaque erreur qui survient dans SQL Server, SQL Server produit un message d'erreur. En règle générale, tous les messages possèdent la même structure : un numéro d'erreur, un message d'explication de l'erreur, un indicateur de sévérité, un état, le nom de la procédure associée à l'erreur et le numéro de la ligne ayant provoquée l'erreur. La gravité est un indicateur, un chiffre de 0 à 24 (gravité croissante). Il est possible de lever des exceptions personnalisées via l'instruction RAISERROR :

```
RAISERROR ('Le stock est négatif !', 12, 1)
```

Lorsqu'on veut lever une erreur, on peut soit donner l'identifiant de l'erreur en question, soit lui donner un message particulier. Si on lui donne un message particulier comme nous l'avons fait dans l'exemple ci-dessus, il faut automatiquement lui préciser une gravité et un état. On peut ajouter une clause WITH à la suite de l'instruction RAISERROR, pour appliquer une des trois options possibles :

- LOG : le message sera consigné dans l'observateur d'évènement Windows.
- NOWAIT : le message sera délivré sans attente à l'utilisateur.
- SETERROR : permet de valoriser @@ERROR et ERROR_NUMBER avec le numéro du message d'erreur.

On peut aussi définir un message d'erreur par la procédure stockée `sp_addmessage` et le supprimer par la procédure stockée `sp_dropmessage`. Voici la syntaxe de création d'un message d'erreur :

```
exec sp_addmessage @msgnum, @severity,  
@msgtext, @lang, @with log, @replace
```

Dans l'ordre, les paramètres correspondent aux données suivantes : identifiant, sévérité, message, langue, log et replace. Les paramètres Log et Replace ne sont pas obligatoires. Replace sert à remplacer le message d'erreur d'une erreur existante. En revanche, pour connaître le code de la langue à utiliser, utilisez la procédure stockée `sp_helplanguage`.

4.6.2 Gestion des erreurs dans le code

Il existe deux manières de gérer les erreurs. La première consiste à tester la valeur de la variable système @@ERROR, la seconde consiste à positionner dans un gestionnaire d'exception `TRY` le bloc d'instructions à tester, et dans le `CATCH`, l'erreur à lever. Voyons la syntaxe :

```
BEGIN TRY
-- ...
END TRY

BEGIN CATCH
-- ...
END CATCH
```

Les instructions `TRY CATCH` ne peuvent être dissociées.

Le bloc `TRY` permet de regrouper ensemble toutes les instructions susceptibles de lever une erreur. Si le cas se présente ou une instruction lève une erreur dans le bloc `TRY`, le contrôle est directement donné à la première instruction du bloc `CATCH`.

Le bloc `CATCH` suit toujours le bloc `TRY`. Celui-ci est exécuté si et seulement si, l'exécution d'une instruction du bloc `TRY` lève une erreur. Dans le bloc `CATCH`, le code permet de gérer l'erreur levée. Pour obtenir des informations sur cette dernière, il est possible d'utiliser les fonctions SQL suivantes :

- `ERROR_MESSAGE()` : Retourne le texte du message à communiquer à l'application. Ce texte comprend tous les paramètres mis en argument à l'erreur en question.
- `ERROR_NUMBER()` : Retourne le numéro de l'erreur.
- `ERROR_SEVERITY()` : Retourne le niveau de gravité.
- `ERROR_STATE()` : Retourne l'état.

5 Conclusion

Dans ce chapitre, nous avons donc vu la majorité des instructions possible en T-SQL DML, avec à chaque fois un exemple d'explication. Il est bon de répéter que ce chapitre ne détaille pas les deux autres facettes du Transact SQL qui sont le DDL et le DCL, tout simplement car on peu assimiler le DCL à l'administration de SQL Server, et parce que nous voyons le DDL au fur et à mesure que nous apprenons à créer les différents objets de la base dans SQL Server. Dans le chapitre suivant nous verrons de quelle manière il est possible de créer et gérer deux nouveaux objets de la base de données : les vues et les index.