

COURS DE BASES DE DONNÉES

Conception et Gestion des Bases de Données

Transactionnel Structured Query Language

Professeur Laachfoubi Nabil

Département des Mathématiques et Informatique

Table des matières

A. L'Algèbre de Boole et Diagrammes de Venn (Rappels).....	4
A.1. Définition.....	4
A.2. Les opérateurs logiques	4
A.3. Quelques théorèmes et lois de l'algèbre de boole	5
A.4. Diagrammes de Venn	7
B. Les similitudes entre la vision ensembliste et la vision relationnelle.....	9
C. Passage d'une vision ensembliste à une vision relationnelle.....	12
D. Le langage TSQL (Transactionnel Structured Query Language)	16
D.1. Les opérateurs et leur priorité	16
D.2. Les variables	17
D.3. Les structures conditionnelles	18
D.4. Les structures itératives	18
D.5. Les mots clés usuels et leur priorité.....	18
D.6. Les catégories de commandes TSQL	19
E. La base de données «Boutique».....	19
F. Le TSQL pour manipuler les données (Data Manipulation Langage).....	22
F.1. Syntaxe du «SELECT»	22
F.1.a) Les options usuelles du «SELECT»	22
F.1.b) Le «SELECT» utilisant les fonctions d'agrégation usuelles	26
F.1.c) Les options usuelles du «FROM»	28
F.1.d) Les options usuelles du «WHERE».....	31
F.1.e) Les options usuelles du «GROUP BY».....	35
F.1.f) L'option «INTO»	42
F.2. Syntaxe de l'«INSERT»	43
F.3. Syntaxe de l'«UPDATE»	45
F.4. Syntaxe du «DELETE».....	48
G. Le TSQL pour manipuler les structures (Data Defintion Langage)	52
G.1. Manipulations usuelles des bases de données	52
G.1.a) Le «CREATE DATABASE»	52
G.1.b) L'«ALTER DATABASE»	54
G.1.c) Le «DROP DATABASE».....	56
G.2. Manipulations usuelles des tables	56
G.2.a) Le «CREATE TABLE».....	56
G.2.b) Le «UPDATE TABLE»	58
G.2.c) Le «DROP TABLE».....	59
G.3. Manipulations usuelles des vues	60

G.3.a) Le «CREATE VIEW» / «ALTER VIEW» / «DROP VIEW»	60
G.4. Manipulations usuelles des procédures stockées.....	61
G.4.a) Le «CREATE PROCEDURE» / «ALTER PROCEDURE» / «DROP PROCEDURE»	61
G.5. Manipulations usuelles des fonctions.....	64
G.5.a) Le «CREATE FUNCTION» / «ALTER FUNCTION » / «DROP FUNCTION»	64
G.6. Manipulations usuelles des triggers.....	67
G.6.a) Le «CREATE TRIGGER» / «ALTER TRIGGER » / «DROP TRIGGER»	67
G.6.b) Le «ENABLE TRIGGER» / «DISABLE TRIGGER »	68
H. Les curseurs.....	68
I. Conception des Bases de Données avec Merise	72
I.1. Le modèle conceptuel des données «MCD»	72
I.1.a) Les entités et les identifiants.....	72
I.1.b) Les associations et les cardinalités	72
(1). Les associations binaires.....	73
(2). Les associations réflexives	73
(3). Les associations n-aires	74
(4). Les associations plurielles.....	74
I.1.c) Les règles de normalisation	75
I.1.d) Les formes normales	77
I.2. Les règles de passage du model conceptuel au modèle logique des données «MLD».....	79
I.3. Le modèle physique des données «MPD».....	80

A. L'Algèbre de Boole et Diagrammes de Venn (Rappels)

A.1. Définition

L'algèbre de Boole est le chapitre des mathématiques qui permet d'adapter les techniques du calcul algébrique à des variables booléennes (vrai ou faux, true ou false, ou encore 0 ou 1) et plus généralement à des expressions booléennes.

Exemples illustrant l'utilisation de l'algèbre de boole dans une communication téléphonique entre deux personnes :

Communication = (Émetteur décroche) ET (Récepteur décroche)

Communication est « vrai » si l'émetteur décroche ET que le récepteur décroche aussi

Décrocher = (Sonnerie ET Décision de répondre) OU (décision d'appeler)

Décrocher est « vrai » si on entend la sonnerie ET que l'on décide de répondre, ou alors si l'on décide tout simplement d'appeler.

A.2. Les opérateurs logiques

ET Logique			OU Logique : inclusif		
ET	0	1	OU	0	1
0	0	0	0	0	1
1	0	1	1	1	1

NOT Logique : négation		XOR Logique : ou exclusif		
Val	Val	XOR	0	1
0	1	0	0	1
1	0	1	1	0

Soit A et B deux variables booléennes prenant soit la valeur 0, soit la valeur 1.

Le résultat de l'opération A ET B est vrai uniquement lorsqu'à la fois A et B sont vrais. Dans les autres cas A ET B donne un résultat faux.

Le résultat de l'opération A OU B est faux uniquement lorsque A et B sont tous les deux faux. Dans les autres cas A OU B donne un résultat vrai.

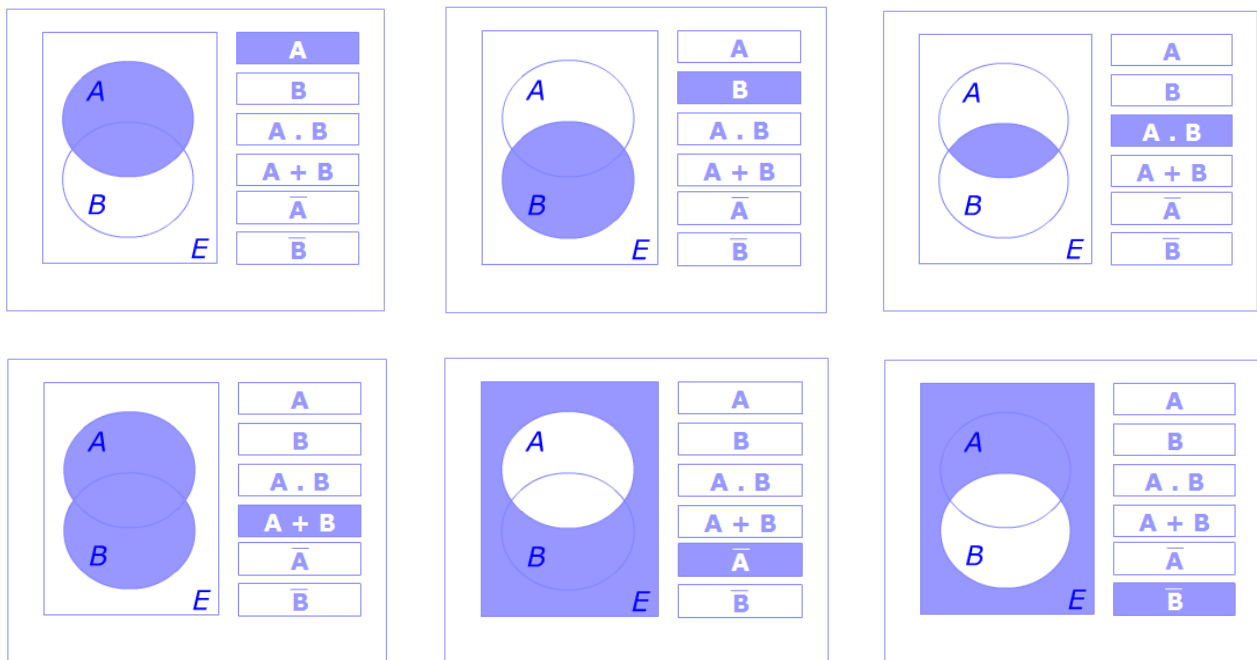
Le résultat de l'opération NOT A (négation de A) est vrai uniquement lorsque A est faux. Réciproquement, le résultat de l'opération NOT A est faux uniquement lorsque A est vrai.

Le résultat de l'opération A XOR B est vrai uniquement lorsque A est la négation de B ou inversement. Dans le cas où A et B sont égaux, le résultat de l'opération est faux.

On verra un peu plus loin dans ce cours que les opérateurs logiques sont très utilisés dans les requêtes de manipulation des données, notamment pour la clause where. La maîtrise de ces opérateurs joue un rôle primordial dans l'optimisation des requêtes SQL.

Pour mieux comprendre ces opérateurs, on va étudier leur similitude avec les opérateurs d'intersection, d'union et de négation, car finalement quand on manipule les données des tables dans une base de données avec le langage SQL, en réalité on est en train de travailler sur des ensembles.

Soit A et B deux ensemble, l'opérateur "." sera assimilé à l'intersection " \cap " pour les ensembles et à l'opérateur "ET" pour l'algèbre de boole, l'opérateur "+" sera assimilé à l'union "U" pour les ensembles et à l'opérateur "OU" pour l'algèbre de boole, l'opérateur "-" sera assimilé à la négation pour les ensembles et à l'opérateur "NOT" pour l'algèbre de boole.



Interprétation :

$A \cdot B$: représente l'intersection, ce sont donc les éléments appartenant en même temps à l'ensemble A et à l'ensemble B (A ET B).

$A + B$: représente l'union, ce sont donc les éléments appartenant à l'ensemble A auxquels sont ajoutés les éléments de l'ensemble B (A OU B).

\bar{A} : représente la négation, ce sont donc tous les éléments n'appartenant pas à l'ensemble A (NOT A)

\bar{B} : représente la négation, ce sont donc tous les éléments n'appartenant pas à l'ensemble B (NOT B)

Etant donné que dans l'algèbre de boole une variable ne peut contenir qu'une seule valeur, et qu'on ne traite que deux valeurs possibles (vrai : 1) et (faux : 0), l'ensemble E est constitué donc des valeurs {vrai, faux} ou encore {0,1}, ce qui a pour conséquence le fait suivant : la négation de (vrai : 1) c'est (faux : 0), et la négation de (faux : 0) c'est (vrai : 1).

Dans le langage "C" à titre d'exemple, la seule valeur équivalente à faux est la valeur zéro, toutes les valeurs différentes de zéro sont considérées comme vrai. En effet, les instructions if (-150) et if (150) sont toutes les deux considérées comme vraie, alors que l'instruction if (0) est considérée comme fausse. Il est donc primordial de ne pas confondre les termes négation et opposé.

A.3. Quelques théorèmes et lois de l'algèbre de boole

Remarque importante : ne pas confondre les opérateurs + et * utilisés dans le cadre de l'algèbre de boole avec ceux utilisés pour les opérations arithmétiques.

- La commutativité : $a + b = b + a$
 $a.b = b.a$
- L'associativité : $(a + b) + c = a + (b + c) = a + b + c$
 $(a.b).c = a.(b.c) = a.b.c$
- La distributivité : $a.(b + c) = a.b + a.c$
 $a + (b.c) = (a + b).(a + c)$
- L'idempotence : $a + a + a + \dots + a = a$
 $a.a.a. \dots .a = a$
- L'élément neutre : $a + 0 = a$
 $a.1 = a$
- L'élément nullité : $0.a = 0$
 $1 + a = 1$
- L'absorption : $a + a.b = a$
 $a.(a + b) = a$
- La simplification : $a.(\bar{a} + b) = a.b$
 $a + \bar{a}.b = a + b$
- La redondance : $a.b + \bar{a}.c = a.b + \bar{a}.c + b.c$
- La complémentarité : $a + \bar{a} = 1$
 $a = \bar{\bar{a}}$
 $a.\bar{a} = 0$
- La dualité : Le théorème dual est formulé à partir du théorème de base en remplaçant les éléments 0 par des 1 (respectivement, les 1 par des 0) et les (\cdot) par des ($+$) (respectivement, les ($+$) par des (\cdot)).
- Exemple :
 $a + a = a$ possède un équivalent dual $\rightarrow a.a = a$

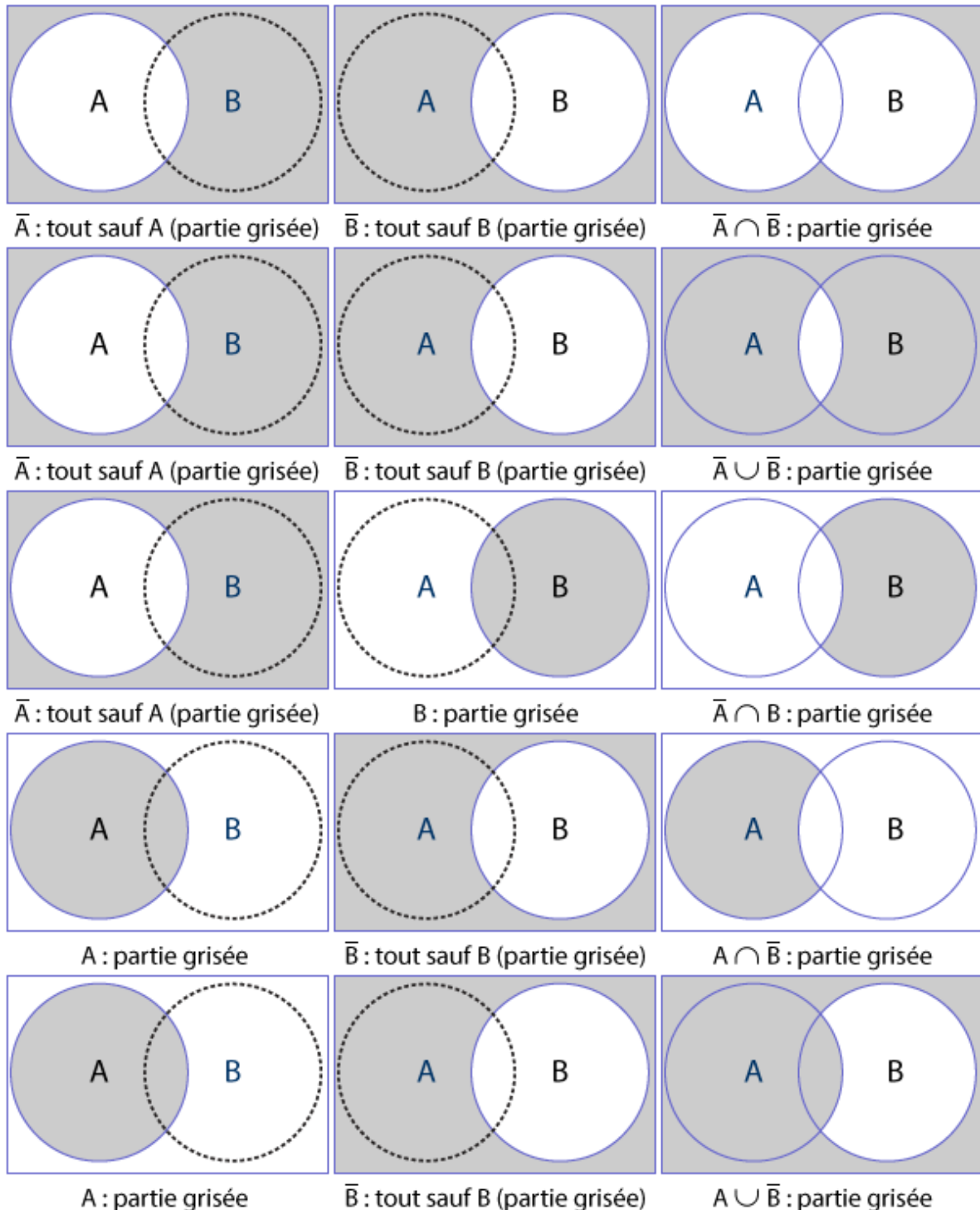
La loi de Morgan :

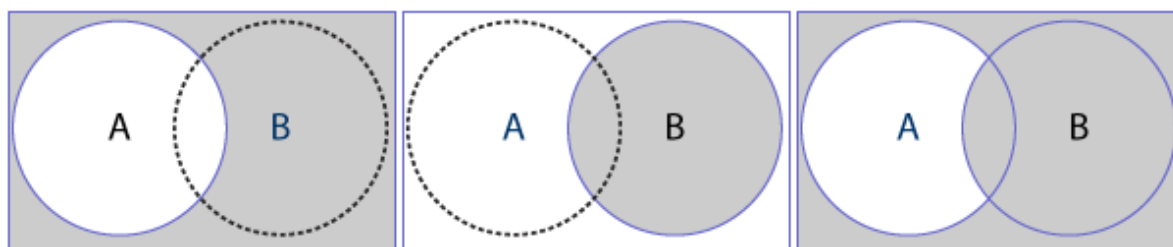
Fonction	Table de vérité/Table de fonctionnement																																			
$\overline{a + b} = \bar{a}.\bar{b}$	<table><tr><th>a</th><th>b</th><th>a+b</th><th>$\overline{a + b}$</th><th>\bar{a}</th><th>\bar{b}</th><th>$\bar{a}.\bar{b}$</th></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	a	b	a+b	$\overline{a + b}$	\bar{a}	\bar{b}	$\bar{a}.\bar{b}$	0	0	0	1	1	1	1	0	1	1	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0
a	b	a+b	$\overline{a + b}$	\bar{a}	\bar{b}	$\bar{a}.\bar{b}$																														
0	0	0	1	1	1	1																														
0	1	1	0	1	0	0																														
1	0	1	0	0	1	0																														
1	1	1	0	0	0	0																														
Fonction	Table de vérité/Table de fonctionnement																																			
$\overline{a.b} = \bar{a} + \bar{b}$	<table><tr><th>a</th><th>b</th><th>$a.b$</th><th>$\overline{a.b}$</th><th>\bar{a}</th><th>\bar{b}</th><th>$\bar{a} + \bar{b}$</th></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	a	b	$a.b$	$\overline{a.b}$	\bar{a}	\bar{b}	$\bar{a} + \bar{b}$	0	0	0	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0
a	b	$a.b$	$\overline{a.b}$	\bar{a}	\bar{b}	$\bar{a} + \bar{b}$																														
0	0	0	1	1	1	1																														
0	1	0	1	1	0	1																														
1	0	0	1	0	1	1																														
1	1	1	0	0	0	0																														

A.4. Diagrammes de Venn

Etant donné que dans les bases de données relationnelles les tables sont liées à travers des attributs ayant des valeurs communes (clés primaires et clés étrangères), extraire des données à partir de ces tables consiste à évaluer des expressions algébriques sur des ensembles.

Ci-dessous des exemples d'opérations sur deux ensembles A et B :





\bar{A} : tout sauf A (partie grisée)

B : partie grisée

$\bar{A} \cup B$: partie grisée

B. Les similitudes entre la vision ensembliste et la vision relationnelle

Une jointure permet de mettre en relation plusieurs tables afin d'en extraire des données conditionnées par des comparaisons de colonnes. A l'image des diagrammes de Venn, il existe plusieurs types de jointures. Savoir quel type de jointure utiliser dépend en réalité des informations que l'on souhaite extraire.

Pour simplifier les choses dans un premier temps, considérons la table TA avec une seule colonne ida, et la table TB avec une seule colonne idb, contenant respectivement les valeurs suivantes :

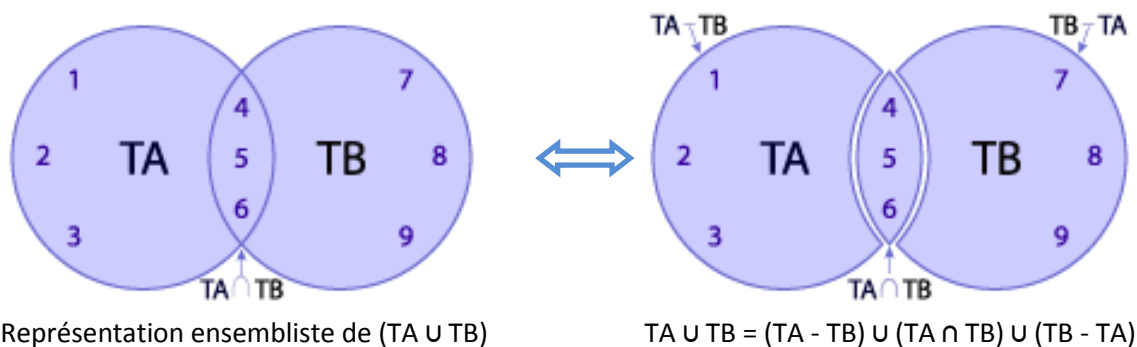
$$TA(ida) = \{1, 2, 3, 4, 5, 6\} \quad \text{et} \quad TB(idb) = \{4, 5, 6, 7, 8, 9\}$$

TA	TB
ida	idb
1	4
2	5
3	6
4	7
5	8
6	9

La jointure de la table TA avec la table TB est une opération permettant d'associer les lignes de la table TA avec celles de la table TB par le biais d'un prédicat pour créer ainsi une troisième table virtuelle (en mémoire) constituée de deux colonnes ida et idb. Généralement, ce prédicat représente le lien sémantique existant entre les deux tables.

Ce lien est caractérisé par l'égalité de certaines valeurs de la colonne ida de la table TA avec celles de la colonne idb de la Table TB (égalité entre la clé primaire et sa clé étrangère sans tenir compte de l'intégrité référentielle¹). En partant donc de l'hypothèse que TA et TB contiennent chacune une seule colonne, on peut faire les constatations suivantes :

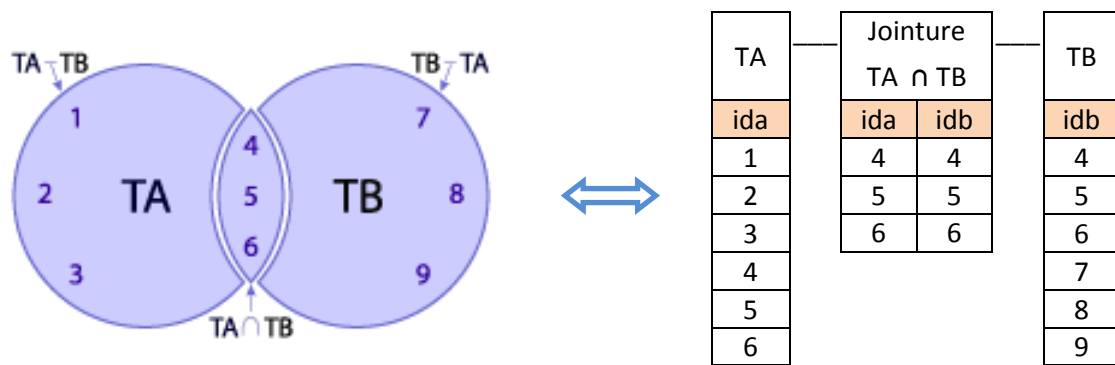
- ida et idb possèdent des valeurs communes : $TA \cap TB = \{(4,4), (5,5), (6,6)\}$
- ida possède des valeurs que idb ne possède pas : $TA \cap \overline{TB} = \{(1,\emptyset), (2,\emptyset), (3,\emptyset)\} = TA - TB$
- idb possède des valeurs que ida ne possède pas : $\overline{TA} \cap TB = \{(\emptyset,7), (\emptyset,8), (\emptyset,9)\} = TB - TA$



Notons que lorsqu'on cherche à extraire les éléments de l'ensemble $(TA - TB)$ à travers une jointure entre TA et TB, en recherche en réalité à faire correspondre les éléments de TA qui ne sont pas contenus dans TB c.à.d. $(TA - TB)$ avec des éléments de TB, mais sachant que l'intersection entre TB et $(TA - TB)$ ne contient aucun élément, la jointure va donc associer chaque élément de l'ensemble $(TA - TB)$ avec des valeurs nulles pour

¹ Toute valeur de la clé étrangère doit exister comme valeur de clé primaire (si idb est une clé étrangère et ida une clé primaire, cela signifie que les valeurs de idb doivent exister dans ida : $idb \subseteq ida$).

chaque colonne de TB. Si l'on souhaite extraire uniquement les éléments de $(TA - TB)$ sans faire référence aux valeurs nulles, il suffit de faire une projection² uniquement sur la colonne ida de la table $(TA - TB)$.



Représentation ensembliste de TA, TB et $TA \cap TB$

Représentation de la jointure $TA \cap TB$

Syntaxe simplifiée pour la jointure entre les tables TA et TB :

```
SELECT { * | TA.ida | TB.idb | TA.ida , TB.idb | TA.* | TB.* | TA.* , TB.* | ida | idb | ida , idb }
FROM TA { [ INNER ] | [ LEFT | RIGHT | FULL [ OUTER ] ] } JOIN TB
ON TA.ida = TB.idb
```

Remarques importantes :

- Le mot clé **JOIN** est obligatoire dans la requête de jointure,
- Les mots clés **INNER** et **OUTER** sont optionnels,
- Le mot clé **INNER** est utilisé dans le cadre des jointures internes, mais il reste optionnel, c'est la valeur par défaut,
- Le mot clé **OUTER** est utilisé dans le cadre des jointures externes, mais il reste optionnel, c'est la valeur par défaut. Toutefois, il est obligatoire de préciser la nature de cette jointure externe, il existe trois sortes de jointures externes :
- LEFT JOIN** : pour jointure gauche, elle consiste à extraire tous les éléments de la table située à gauche de cette instruction même si tous les éléments de cette table n'ont pas de correspondance dans la table de droite. Les éléments qui ne possèdent pas de correspondance seront associés à des valeurs nulles,
- RIGHT JOIN** : pour jointure droite, elle consiste à extraire tous les éléments de la table située à droite de cette instruction même si tous les éléments de cette table n'ont pas de correspondance dans la table de gauche. Les éléments qui ne possèdent pas de correspondance seront associés à des valeurs nulles,
- FULL JOIN** : il s'agit de l'union entre la jointure gauche et la jointure droite, elle consiste à extraire tous les éléments de la table située à gauche de cette instruction même si tous les éléments de cette table n'ont pas de correspondance dans la table de droite, et de les rajouter aux éléments de la table située à droite de cette instruction même si tous les éléments de cette table n'ont pas de correspondance dans la table de gauche. Les éléments qui ne possèdent pas de correspondance dans l'autre table seront associés à des valeurs nulles.

² La projection consiste à afficher que les données de certaines colonnes d'une table donnée.

- Les requêtes suivantes sont équivalentes :

```
SELECT * FROM TA INNER JOIN TB ON TA.ida = TB.idb  
SELECT * FROM TA JOIN TB ON TA.ida = TB.idb
```

Idem pour :

```
SELECT * FROM TA LEFT OUTER JOIN TB ON TA.ida = TB.idb  
SELECT * FROM TA LEFT JOIN TB ON TA.ida = TB.idb
```

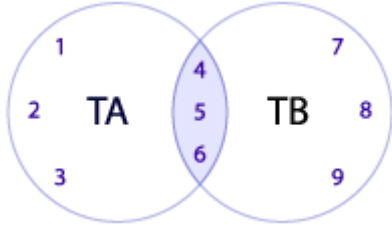
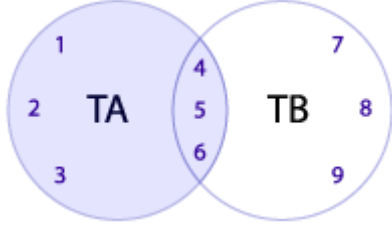
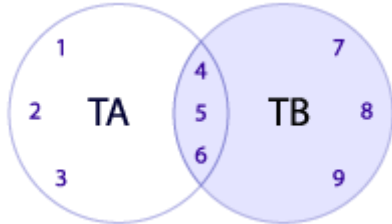
Idem pour :

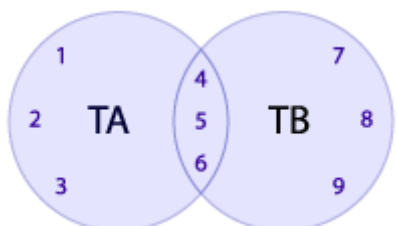
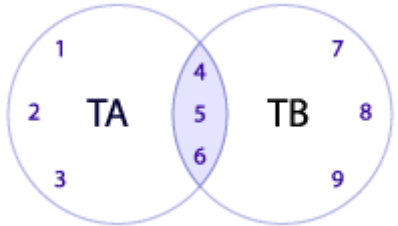
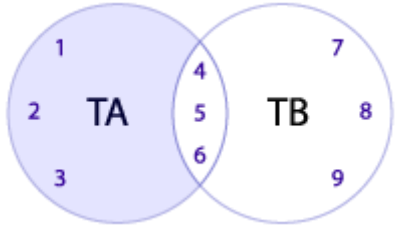
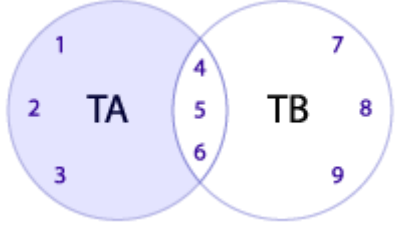
```
SELECT * FROM TA RIGHT OUTER JOIN TB ON TA.ida = TB.idb  
SELECT * FROM TA RIGHT JOIN TB ON TA.ida = TB.idb
```

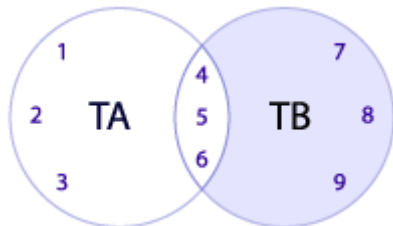
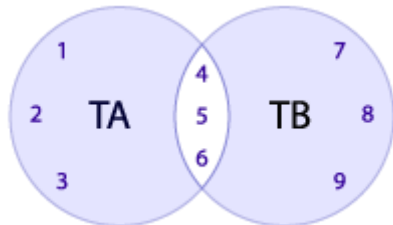
Idem pour :

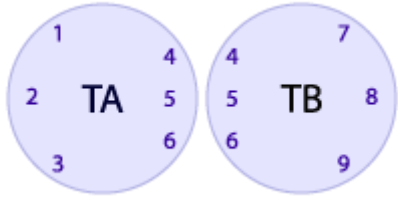
```
SELECT * FROM TA FULL OUTER JOIN TB ON TA.ida = TB.idb  
SELECT * FROM TA FULL JOIN TB ON TA.ida = TB.idb
```

C. Passage d'une vision ensembliste à une vision relationnelle

N°	Schéma	Requêtes fournissant le même résultat	Résultat														
1)		<p>INNER JOIN est identique à JOIN</p> <pre>SELECT * FROM TA JOIN TB ON TA.ida = TB.idb ----- SELECT * FROM TA INNER JOIN TB ON TA.ida = TB.idb ----- SELECT * FROM TA , TB WHERE TA.ida = TB.idb</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>5</td></tr><tr><td>6</td><td>6</td></tr></table>	ida	idb	4	4	5	5	6	6						
ida	idb																
4	4																
5	5																
6	6																
2)		<p>LEFT JOIN est identique à LEFT OUTER JOIN :</p> <pre>SELECT TA.ida , TB.idb FROM TA LEFT JOIN TB ON TA.ida = TB.idb ----- SELECT TA.ida , TB.idb FROM TA LEFT OUTER JOIN TB ON TA.ida = TB.idb ----- SELECT TA.ida , NULL AS idb FROM TA WHERE TA.ida NOT IN (SELECT DISTINCT TB.idb FROM TB) UNION SELECT TA.ida , TB.idb FROM TA , TB WHERE TA.ida = TB.idb</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>1</td><td>NULL</td></tr><tr><td>2</td><td>NULL</td></tr><tr><td>3</td><td>NULL</td></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>5</td></tr><tr><td>6</td><td>6</td></tr></table>	ida	idb	1	NULL	2	NULL	3	NULL	4	4	5	5	6	6
ida	idb																
1	NULL																
2	NULL																
3	NULL																
4	4																
5	5																
6	6																
3)		<p>RIGHT JOIN est identique à RIGHT OUTER JOIN :</p> <pre>SELECT TA.ida , TB.idb FROM TA RIGHT JOIN TB ON TA.ida = TB.idb ----- SELECT TA.ida , TB.idb FROM TA RIGHT OUTER JOIN TB ON TA.ida = TB.idb ----- SELECT TA.ida , TB.idb FROM TA , TB WHERE TA.ida = TB.idb UNION SELECT NULL AS ida , TB.idb FROM TB WHERE TB.idb NOT IN (SELECT DISTINCT TA.ida FROM TA)</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>5</td></tr><tr><td>6</td><td>6</td></tr><tr><td>NULL</td><td>7</td></tr><tr><td>NULL</td><td>8</td></tr><tr><td>NULL</td><td>9</td></tr></table>	ida	idb	4	4	5	5	6	6	NULL	7	NULL	8	NULL	9
ida	idb																
4	4																
5	5																
6	6																
NULL	7																
NULL	8																
NULL	9																

N°	Schéma	Requêtes fournissant le même résultat	Résultat																				
4)		<p>FULL JOIN est identique à FULL OUTER JOIN (union avec intersection)</p> <pre>SELECT * FROM TA FULL JOIN TB ON TA.ida = TB.idb ----- SELECT * FROM TA FULL OUTER JOIN TB ON TA.ida = TB.idb</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>1</td><td>NULL</td></tr><tr><td>2</td><td>NULL</td></tr><tr><td>3</td><td>NULL</td></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>5</td></tr><tr><td>6</td><td>6</td></tr><tr><td>NULL</td><td>7</td></tr><tr><td>NULL</td><td>8</td></tr><tr><td>NULL</td><td>9</td></tr></table>	ida	idb	1	NULL	2	NULL	3	NULL	4	4	5	5	6	6	NULL	7	NULL	8	NULL	9
ida	idb																						
1	NULL																						
2	NULL																						
3	NULL																						
4	4																						
5	5																						
6	6																						
NULL	7																						
NULL	8																						
NULL	9																						
5)		<p>Semi-jointure (intersection de TA et de TB)</p> <pre>SELECT TA.ida FROM TA WHERE EXISTS (SELECT 1 FROM TB WHERE TA.ida = TB.idb) ----- SELECT TA.ida FROM TA INTERSECT SELECT TB.idb FROM TB</pre>	<table><tr><th>ida</th></tr><tr><td>4</td></tr><tr><td>5</td></tr><tr><td>6</td></tr></table>	ida	4	5	6																
ida																							
4																							
5																							
6																							
6)		<p>Anti semi-jointure avec NOT EXISTS, NOT IN ou EXCEPT (TA - TB) :</p> <pre>SELECT TA.ida FROM TA WHERE NOT EXISTS (SELECT 1 FROM TB WHERE TA.ida = TB.idb) ----- SELECT TA.ida FROM TA WHERE TA.ida NOT IN (SELECT DISTINCT TB.idb FROM TB) ----- SELECT TA.ida FROM TA EXCEPT SELECT TB.idb FROM TB</pre>	<table><tr><th>ida</th></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	ida	1	2	3																
ida																							
1																							
2																							
3																							
7)		<p>LEFT JOIN Exclusif (TA sans intersection) :</p> <pre>SELECT TA.ida , TB.idb FROM TA LEFT JOIN TB ON TA.ida = TB.idb WHERE TB.idb IS NULL ----- SELECT TA.ida , NULL AS idb FROM TA WHERE TA.ida NOT IN (SELECT DISTINCT TB.idb FROM TB)</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>1</td><td>NULL</td></tr><tr><td>2</td><td>NULL</td></tr><tr><td>3</td><td>NULL</td></tr></table>	ida	idb	1	NULL	2	NULL	3	NULL												
ida	idb																						
1	NULL																						
2	NULL																						
3	NULL																						

N°	Schéma	Requêtes fournissant le même résultat	Résultat														
8)		<p>RIGHT JOIN Exclusive (TB sans intersection) :</p> <pre>SELECT TA.ida , TB.idb FROM TA RIGHT JOIN TB ON TA.ida = TB.idb WHERE TA.ida IS NULL</pre> <p>-----</p> <pre>SELECT NULL AS ida , TB.idb FROM TB WHERE TB.idb NOT IN (SELECT DISTINCT TA.ida FROM TA)</pre> <p>-----</p> <pre>SELECT NULL AS ida , TB.idb FROM TB WHERE NOT EXISTS (SELECT 1 FROM TA WHERE TA.ida = TB.idb)</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>NULL</td><td>7</td></tr><tr><td>NULL</td><td>8</td></tr><tr><td>NULL</td><td>9</td></tr></table>	ida	idb	NULL	7	NULL	8	NULL	9						
ida	idb																
NULL	7																
NULL	8																
NULL	9																
9)		<p>FULL JOIN Exclusif (union sans intersection) : représente l'union entre le left join exclusif et le right join exclusif.</p> <pre>SELECT TA.ida , TB.idb FROM TA FULL JOIN TB ON TA.ida = TB.idb WHERE TA.ida IS NULL OR TB.idb IS NULL</pre> <p>-----</p> <pre>SELECT TA.ida , TB.idb FROM TA LEFT JOIN TB ON TA.ida = TB.idb WHERE TB.idb IS NULL UNION SELECT TA.ida , TB.idb FROM TA RIGHT JOIN TB ON TA.ida = TB.idb WHERE TA.ida IS NULL</pre>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>1</td><td>NULL</td></tr><tr><td>2</td><td>NULL</td></tr><tr><td>3</td><td>NULL</td></tr><tr><td>NULL</td><td>7</td></tr><tr><td>NULL</td><td>8</td></tr><tr><td>NULL</td><td>9</td></tr></table>	ida	idb	1	NULL	2	NULL	3	NULL	NULL	7	NULL	8	NULL	9
ida	idb																
1	NULL																
2	NULL																
3	NULL																
NULL	7																
NULL	8																
NULL	9																

N°	Schéma	Requêtes fournissant le même résultat	Résultat																																																																										
10)		<p>CROSS JOIN (produit cartésien) :</p> <pre>SELECT TA.ida , TB.idb FROM TA CROSS JOIN TB</pre> <p>-----</p> <pre>SELECT TA.ida , TB.idb FROM TA , TB</pre> <p>Il est à noter que dans les deux requêtes la condition de jointure est absente. Dans la première requête elle est à proscrire car elle va générer une erreur de syntaxe, mais dans la deuxième requête elle a été retirée volontairement. En effet, l'idée consiste à associer chaque ligne de la table TB avec toutes les lignes de la table TA sans se préoccuper du lien sémantique entre les données.</p> <p>Le résultat du croisement contient 36 lignes :</p> <p>6 lignes de TA x 6 lignes de TB = 36 lignes</p>	<table><tr><th>ida</th><th>idb</th></tr><tr><td>1</td><td>4</td></tr><tr><td>2</td><td>4</td></tr><tr><td>3</td><td>4</td></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>4</td></tr><tr><td>6</td><td>4</td></tr><tr><td>1</td><td>5</td></tr><tr><td>2</td><td>5</td></tr><tr><td>3</td><td>5</td></tr><tr><td>4</td><td>5</td></tr><tr><td>5</td><td>5</td></tr><tr><td>6</td><td>5</td></tr><tr><td>1</td><td>6</td></tr><tr><td>2</td><td>6</td></tr><tr><td>3</td><td>6</td></tr><tr><td>4</td><td>6</td></tr><tr><td>5</td><td>6</td></tr><tr><td>6</td><td>6</td></tr><tr><td>1</td><td>7</td></tr><tr><td>2</td><td>7</td></tr><tr><td>3</td><td>7</td></tr><tr><td>4</td><td>7</td></tr><tr><td>5</td><td>7</td></tr><tr><td>6</td><td>7</td></tr><tr><td>1</td><td>8</td></tr><tr><td>2</td><td>8</td></tr><tr><td>3</td><td>8</td></tr><tr><td>4</td><td>8</td></tr><tr><td>5</td><td>8</td></tr><tr><td>6</td><td>8</td></tr><tr><td>1</td><td>9</td></tr><tr><td>2</td><td>9</td></tr><tr><td>3</td><td>9</td></tr><tr><td>4</td><td>9</td></tr><tr><td>5</td><td>9</td></tr><tr><td>6</td><td>9</td></tr></table>	ida	idb	1	4	2	4	3	4	4	4	5	4	6	4	1	5	2	5	3	5	4	5	5	5	6	5	1	6	2	6	3	6	4	6	5	6	6	6	1	7	2	7	3	7	4	7	5	7	6	7	1	8	2	8	3	8	4	8	5	8	6	8	1	9	2	9	3	9	4	9	5	9	6	9
ida	idb																																																																												
1	4																																																																												
2	4																																																																												
3	4																																																																												
4	4																																																																												
5	4																																																																												
6	4																																																																												
1	5																																																																												
2	5																																																																												
3	5																																																																												
4	5																																																																												
5	5																																																																												
6	5																																																																												
1	6																																																																												
2	6																																																																												
3	6																																																																												
4	6																																																																												
5	6																																																																												
6	6																																																																												
1	7																																																																												
2	7																																																																												
3	7																																																																												
4	7																																																																												
5	7																																																																												
6	7																																																																												
1	8																																																																												
2	8																																																																												
3	8																																																																												
4	8																																																																												
5	8																																																																												
6	8																																																																												
1	9																																																																												
2	9																																																																												
3	9																																																																												
4	9																																																																												
5	9																																																																												
6	9																																																																												

D. Le langage TSQL (Transactionnel Structured Query Language)

Avant toute chose, il est important de rappeler que TSQL est un langage de programmation spécifique à Microsoft SQL Server. Comme pour tous les langages de programmation, il utilise un ensemble d'opérateurs, de mots clés et de structures. Nous allons étudier dans cette section les principaux éléments de ce langage.

D.1. Les opérateurs et leur priorité

Le TSQL dispose d'un certain nombre d'opérateurs et de mots clés préfinis qui s'exécutent en respectant un ordre de priorité bien établi. Voici donc une idée non exhaustive sur la liste des opérateurs, la liste des mots clés et l'ordre de priorité les concernant :

Opérateurs Arithmétiques	
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (Reste de la division)

Opérateurs de Comparaison et d'Affectation	
=	Comparaison ou affectation selon l'utilisation
>	Supérieur
>=	Supérieur ou égal
<	Inférieur
<=	Inférieur ou égal
<>	Différent
expression IN (exp ₁ , exp ₁ , ..., exp _n)	Compare expression à toutes les expressions de la liste
expression IS NULL	Renvoie True si expression est NULL, False dans le cas contraire
expression BETWEEN valmin AND valmax	Recherche si la valeur de l'expression est comprise entre la valeur valmin et valmax. Les bornes sont comprises
EXISTS (sous-requête)	Renvoie True si la sous requête renvoie au moins une ligne
expression LIKE modèle	Permet de filtrer des données suivant un modèle

Opérateurs Logiques	
expression_1 AND expression_2	Retourne True si les deux expressions sont vraies
expression_1 OR expression_2	Retourne True si l'une au moins des deux expressions est vraie
NOT expression	Retourne True si l'expression est fausse
expression IS NULL	Retourne True Si expression est NULL, False dans le cas contraire
expression IS NOT NULL	Retourne False Si expression est NULL, True dans le cas contraire

Priorité	Opérateurs arithmétiques, logiques, de comparaison et d'affectation
1	* (Multiplication), / (Division), % (Modulo)
2	+ (Positif), - (Négatif), + (Addition), + (Concaténation), - (Soustraction)
3	= , > , < , >= , <= , <> , != , !> , !< (comparaisons)
4	NOT (Négation)
5	AND
6	ALL , ANY , BETWEEN , IN , LIKE , OR , SOME
7	= (Affectation)

N.B : Pour les opérateurs ayant la même priorité, l'exécution se fera en fonction de l'ordre de leur apparition dans les instructions de gauche vers la droite. Pour imposer un ordre d'exécution particulier, l'usage des parenthèses devient obligatoire.

D.2. Les variables

Essentiellement, TSQL dispose de deux catégories de variables : celles définies par le développeur (elles doivent être préfixées par le caractère «@»), et celles prédéfinies dans le langage, appelées variables système (elles sont préfixées par le terme «@@»).

```

DECLARE
{
    {
        {@nom_variable [AS] type_donnée_scalaire [= valeur]}
        |
        {@nom_curseur CURSOR}
    }[, ... ]
    |
    {@nom_table [AS] TABLE ({définition_colonnei | définition_contraintei} [, ... ])}
}

définition_colonnei :
nom_colonnei type_donnée_scalaire [{précision [, échelle] | max}]
[COLLATE identifiant_du_jeu_de_caractères]
[[DEFAULT valeur] | IDENTITY [(valeur_de_départ, valeur_du_pas)]]
[[NULL | NOT NULL] | [PRIMARY KEY | UNIQUE] | CHECK (expression_logique)]

```

```

définition_contraintei :
{
  {PRIMARY KEY | UNIQUE} (nom_colonnei [, ... ])
  |
  CHECK (expression_logique)
}

```

D.3. Les structures conditionnelles

D.4. Les structures itératives

D.5. Les mots clés usuels et leur priorité

Priorité	Mot clé
1	FROM
2	ON
3	JOIN
4	WHERE
5	GROUP BY
6	HAVING
7	SELECT
8	DISTINCT
9	ORDER BY
10	TOP

N.B : Cet ordre est généralement respecté. Toutefois, il existe des cas très rares où il est perturbé, comme par exemple lorsqu'il s'agit d'une conversion de types dans la clause «select». Dans ce cas de figure, la conversion avec la fonction «convert» s'exécutera avant la clause «where».

A l'instar des langages SQL permettant de réaliser des programmes, le TSQL offre des instructions pour manipuler les données, des instructions pour manipuler les structures et puis d'autres pour créer des programmes élaborés (procédures stockées, fonctions, déclencheurs, curseurs, nouveaux types de données, boucles, ...)

D.6. Les catégories de commandes TSQL

Les commandes du langage «TSQL» sont répertoriées dans plusieurs catégories. Nous allons dans ce qui suit, présenter les quatre catégories de ce langage :

La catégorie DML (DATA MANIPULATION LANGUAGE) qui inclue des instructions permettant de manipuler les données :

- SELECT, INSERT, UPDATE, DELETE, MERGE

La catégorie DDL (DATA DEFINITION LANGUAGE) qui inclue des instructions permettant de manipuler les structures :

- CREATE, ALTER, DROP, RENAME, TRUNCATE, COMMENT

La catégorie DCL (DATA CONTROL LANGUAGE) qui inclue des instructions permettant de manipuler les droits des utilisateurs :

- GRANT, REVOKE

La catégorie TCL (TRANSACTION CONTROL LANGUAGE) qui inclue des instructions permettant de manipuler les transactions :

- COMMIT, ROLLBACK, SAVEPOINT

E. La base de données «Boutique»

Pour comprendre le résultat produit par chaque requête, nous allons nous baser sur la base de données «BOUTIQUE» dont nous illustrons ci-dessous le schéma relationnel et les tables :

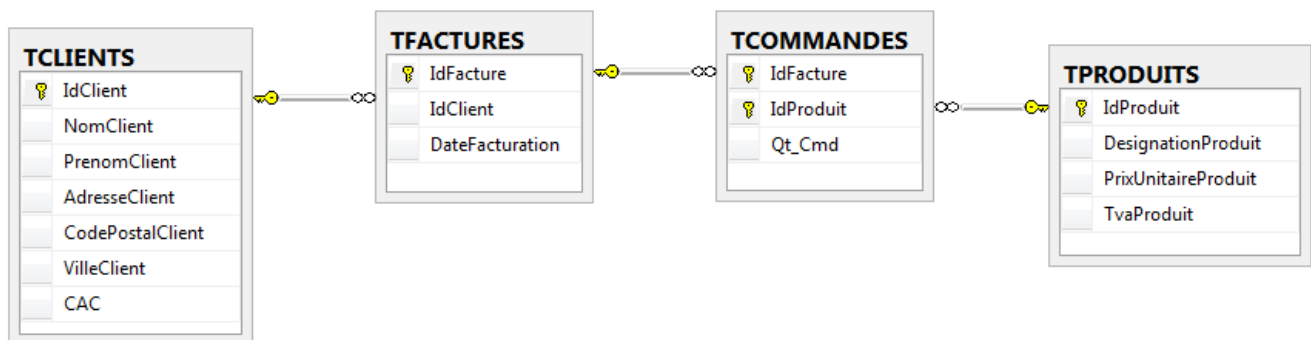


Table «TClients»

IdClient	NomClient	PrenomClient	AdresseClient	CodePostalClient	VilleClient	CAC
1	NA	PA	AA	CA	CASABLANCA	0.00
2	NB	PB	AB	CB	SALE	0.00
3	NC	PC	AC	CC	RABAT	0.00
4	ND	PD	AD	CD	CASABLANCA	0.00
5	NE	PE	AE	CE	SALE	0.00

Table «TFactures»

IdFacture	IdClient	DateFacturation
1	1	01/10/2016
2	1	07/10/2016
3	1	09/10/2017
4	2	11/10/2017
5	2	12/10/2017
6	2	13/10/2018
7	3	05/11/2018
8	3	05/12/2018
9	3	10/12/2018

Table «TProduits»

IdProduit	DesignationProduit	PrixUnitaireProduit	TvaProduit
1	PA	10,00	0,20
2	PB	15,00	0,30
3	PC	20,00	0,15
4	PD	25,00	0,20
5	PE	30,00	0,10
6	PF	40,00	0,30
7	PG	50,00	0,25

Table «TCommandes»

IdFacture	IdProduit	Qt_Cmd
1	1	10
1	2	20
1	3	30
2	2	10
2	3	20
2	4	30
3	3	10
3	4	20
3	5	30
4	1	5
4	2	10
4	3	15
5	2	5
5	3	10
5	4	15
6	3	20
6	4	20
6	5	20
7	3	3
7	4	4
7	5	5
8	1	10
8	2	20
8	3	30
9	2	2
9	3	3
9	4	4

F. Le TSQL pour manipuler les données (Data Manipulation Language)

F.1. Syntaxe du «SELECT»

```
SELECT      <options_du_select>
[ INTO      nouvelle_table ]
[ FROM      tables_sources ]
[ WHERE     conditions_de_recherche ]
[ GROUP BY  expression_de_reroupage ]
[ HAVING    condition_du_filtre ]
[ ORDER BY  ordre_affichage [ ASC | DESC ] ]
```

Notons, qu'en dehors du mot clé «select», les autres mots clés sont optionnels. Il existe donc une multitude de possibilités pour les combiner ensemble. Pour mieux illustrer ces cas de figures, nous allons donner des exemples d'utilisation pour chaque mot clé. Notons que l'ordre des options doit être respecté.

F.1.a) Les options usuelles du «SELECT»

```
SELECT
[ DISTINCT | ALL ]
[ TOP ( expression ) [ PERCENT ] [ WITH TIES ] ]
< liste_des_attributs >
```

Notons que le seul argument obligatoire pour le «select» est la liste des attributs. Ces attributs ne sont pas forcément des noms de colonnes de tables existantes, mais ça peut être également des noms de colonnes virtuelles (Alias) associées à des expressions faisant référence à des sous requêtes ou à des calculs, ou tout simplement un moyen pour stocker un résultat ou une valeur dans une variable.

Exemple 1 : Select ne faisant référence à aucune table physique

```
SELECT Message = 'Bonjour tout le monde !!!'
```

Résultat :

Message
Bonjour tout le monde !!!

```
SELECT 'Bonjour tout le monde !!!' AS Message
```

Résultat :

Message
Bonjour tout le monde !!!

```
DECLARE @VAL AS INT          /*Déclaration d'une variable de type entier*/
SELECT @VAL = 100             /*Affectation de la valeur 100 à cette variable*/
SELECT @VAL AS Message
```

Résultat :

Message
100

```

SELECT
    *
FROM
    (
        VALUES (1, 'Bonjour'), (2, 'Tout'), (3, 'Le'), (4, 'Monde')
    ) AS TableDerivee (IdMot, Mot)

```

Résultat :

IdMot	Mot
1	Bonjour
2	Tout
3	Le
4	Monde

Le dernier «SELECT» s'appuie sur une table créée virtuellement pour être exploitée durant d'exécution de la requête.

Exemple 2 : Select utilisant le mot clé «FROM» sur une table physique

```

SELECT
    IdClient, NomClient, VilleClient
FROM
    TCLIENTS

```

Résultat :

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
2	NB	SALE
3	NC	RABAT
4	ND	CASABLANCA
5	NE	SALE

Notons qu'on peut utiliser le mot clé «FROM» sur plusieurs tables sans faire référence à la clause «WHERE» pour établir les jointures qui lie ces table (voir produit cartésien plus haut). Mais on peut également rajouter cette clause pour établir les liens sémantiques entre ces tables, ou pour spécifier des conditions particulières à respecter pour extraire les résultats.

Exemple 3 : Select utilisant le mot clé «DISTINCT» sur une, ou plusieurs colonnes

```

SELECT
    DISTINCT
    VilleClient
FROM
    TCLIENTS

```

Résultat :

VilleClient
CASABLANCA
SALE
RABAT

Notons que ce terme peut s'appliquer également sur plusieurs colonnes simultanées :

```

SELECT  DISTINCT
        IdClient, DateFacturation
FROM
        TFACTURES

```

Résultat :

IdClient	DateFacturation
1	01/09/2016
1	03/09/2016
2	02/10/2016
2	03/10/2016
3	05/11/2016

Exemple 4 : Select utilisant le mot clé «FROM» sur plusieurs tables physiques

```

SELECT  DISTINCT
        DesignationProduit
FROM
        TCLIENTS, TFACTURES, TCOMMANDES, TPRODUITS
WHERE
        TCLIENTS.IdClient      = 1 AND
        TCLIENTS.IdClient      = TFACTURES.IdClient AND
        TFACTURES.IdFacture    = TCOMMANDES.IdFacture AND
        TCOMMANDES.IdProduit   = TPRODUITS.IdProduit

```

Cette requête permet d'afficher la liste des produits achetés par le client N° 1. Etant donné qu'elle n'affiche aucun attribut de la table des clients, et que le code client se trouve également dans la table des factures en tant que clé étrangère, on aurait pu l'optimiser en retirant toute référence à cette table et en exploitant le code du client se trouvant dans la table des factures. Cette opération va nous permettre d'éviter de faire un produit cartésien supplémentaire inutilement.

```

SELECT  DISTINCT
        DesignationProduit
FROM
        TFACTURES, TCOMMANDES, TPRODUITS
WHERE
        TFACTURES.IdClient      = 1 AND
        TFACTURES.IdFacture     = TCOMMANDES.IdFacture AND
        TCOMMANDES.IdProduit    = TPRODUITS.IdProduit

```

Résultat :

DesignationProduit
PA
PB
PC
PD
PE

Exemple 5 : Select utilisant le mot clé «ALL»

«ALL» permet de comparer chaque valeur d'une colonne pour savoir si elle répond à un critère donné vis-à-vis d'une liste de valeurs retournées par une sous-requête. Autrement dit, ce mot permet de vérifier si une valeur donnée répond à une condition "=", "<>", ">", ">=", "<", ou "<=" pour l'ensemble des valeurs retournées par la sous-requête.


```

SELECT      DesignationProduit, PrixUnitaireProduit
FROM        TPRODUITS
WHERE       PrixUnitaireProduit <> ALL
            (/*sous-requête pour l'extraction des prix multiples de 10*/
             /*pour utiliser le modulo «%», il faut convertir le prix en entier*/
             SELECT
                PrixUnitaireProduit
            FROM
                TPRODUITS
            WHERE
                CONVERT(int, PrixUnitaireProduit)%10 = 0
            )

```

Pour retrouver le même résultat, il est évident qu'on aurait pu faire simplement avec la requête qui suit, mais on n'aurait pas pu expliquer le fonctionnement du mot clé «ALL» :

```

SELECT      DesignationProduit, PrixUnitaireProduit
FROM        TPRODUITS
WHERE       CONVERT(int, PrixUnitaireProduit)%10 <> 0

```

Résultat :

DesignationProduit	PrixUnitaireProduit
PB	15,00
PD	25,00

Exemple 6 : Select utilisant le mot clé «TOP»

«TOP» permet d'extraire les n premières lignes, ou alors un pourcentage de lignes d'une table donnée selon l'ordre de lecture de ces lignes. Si la clause «ORDER BY» est spécifiée avec le «SELECT», les lignes sont tout d'abord triées avant l'extraction.

Lorsque le mot clé «WITH TIES» est utilisé conjointement avec «TOP», la clause «ORDER BY» devient obligatoire, car le rôle de ce mot clé est de rajouter parmi les lignes triées, toutes celles ayant le même classement que la dernière ligne extraire par «TOP». pour mieux comprendre voyons les résultats des 3 requêtes suivantes :

```

SELECT TOP (4)
    DesignationProduit, TvaProduit
FROM
    TPRODUITS

```

Résultat : le «TOP» renvoie les 4 premières lignes selon l'ordre de lecture.

DesignationProduit	TvaProduit
PA	0,20
PB	0,30
PC	0,15
PD	0,20

```

SELECT TOP (4)
    DesignationProduit, TvaProduit
FROM
    TPRODUITS
ORDER BY
    TvaProduit ASC

```

Résultat : le «TOP» renvoie les 4 premières lignes selon l'ordre du tri.

DesignationProduit	TvaProduit
PE	0,10
PC	0,15
PA	0,20
PD	0,20

```

SELECT TOP (3) WITH TIES
    DesignationProduit, TvaProduit
FROM
    TPRODUITS
ORDER BY
    TvaProduit ASC

```

Résultat : le «TOP WITH TIES» renvoie 3 + 1 premières lignes selon l'ordre du tri.

DesignationProduit	TvaProduit	
PE	0,10	
PC	0,15	
PA	0,20	Dernière ligne du TOP(3)
PD	0,20	Ligne ajoutée par l'instruction WITH TIES

Le «TOP» renvoi les 3 premières lignes selon l'ordre de tri + les lignes ayant le même classement que la dernière ligne du TOP(3). Dans notre cas, il s'agit d'une seule ligne concernant le produit PD avec une tva de 0.20.

F.1.b) Le «SELECT» utilisant les fonctions d'agrégation usuelles

```

SELECT
    { COUNT | MAX | MIN | AVG } ( { [ ALL | DISTINCT ] expression } | * ) [ AS alias ]

```

Notons que les accolades sont là uniquement pour désigner les termes obligatoires, et les crochets pour désigner les termes optionnels. Le symbole «*» fait référence à toutes les colonnes des tables sur lesquelles opère la sélection. Dans le cas des fonctions d'agrégation, ce symbole ne fonctionne qu'avec la fonction «COUNT». Par ailleurs, le terme «ALL» est la valeur par défaut, il permet d'appliquer la fonction d'agrégation à toutes les valeurs.

COUNT

expression : fait référence à un argument de tous les types, sauf les fonctions d'agrégation, les sous-requêtes, les types image, text et ntext.

COUNT(*) : renvoie le nombre de lignes d'un select. Les valeurs nulles «NULL» et les doublons sont comptabilisés.

COUNT(ALL expression) : évalue l'expression pour chaque ligne puis renvoie le nombre de valeurs non nulles «NULL». «ALL» est la valeur par défaut.

COUNT(DISTINCT expression) évalue l'expression pour chaque ligne d'un groupe et renvoie le nombre de valeurs uniques et non nulles «NULL».

Pour un nombre de valeurs supérieur à $2^{31} - 1$, cette fonction renvoie une erreur. A ce moment, il est préférable d'utiliser la fonction COUNT_BIG qui peut renvoyer un nombre de valeurs allant jusqu'à $2^{63} - 1$.

MAX

Cette fonction renvoie la valeur maximale de l'expression. Elle ignore toutes les valeurs nulles «NULL». Pour les colonnes de type chaîne de caractères, elle renvoie la plus grande valeur dans l'ordre alphabétique. Lorsque l'expression est évaluée à nulle «NULL», elle renvoie «NULL».

expression : fait référence au nom d'une colonne, au nom d'une fonction, à une expression arithmétique ou de concaténation, ou encore à une constante. Cette fonction peut être appliquée sur des valeurs numériques, chaînes de caractères ou encore de type «datetime», à l'exception des fonctions d'agrégation, des sous-requêtes et le type bit.

MAX(ALL expression) : évalue l'expression puis renvoie la valeur maximale. «ALL» est la valeur par défaut.

MAX(DISTINCT expression) évalue l'expression puis renvoie la valeur maximale. le terme «DISTINCT» n'a aucun effet sur le résultat.

MIN

Cette fonction renvoie la valeur minimale de l'expression. Elle ignore toutes les valeurs nulles «NULL». Pour les colonnes de type chaîne de caractères, elle renvoie la plus petite valeur dans l'ordre alphabétique. Lorsque l'expression est évaluée à nulle «NULL», elle renvoie «NULL».

expression : fait référence au nom d'une colonne, au nom d'une fonction, à une expression arithmétique ou de concaténation, ou encore à une constante. Cette fonction peut être appliquée sur des valeurs numériques, chaînes de caractères ou encore de type «datetime», à l'exception des fonctions d'agrégation, des sous-requêtes et le type bit.

MIN(ALL expression) : évalue l'expression puis renvoie la valeur minimale. «ALL» est la valeur par défaut.

MIN(DISTINCT expression) évalue l'expression puis renvoie la valeur minimale. le terme «DISTINCT» n'a aucun effet sur le résultat.

SUM

Cette fonction renvoie la somme des valeurs relatives à l'expression. Elle ignore toutes les valeurs nulles «NULL», et ne peut être utilisée que sur des colonnes contenant des valeurs numériques.

expression : fait référence au nom d'une colonne, au nom d'une fonction, à une expression arithmétique ou de concaténation, ou encore à une constante. Cette fonction ne peut être appliquée que sur des valeurs numériques. Les fonctions d'agrégation, les sous-requêtes et le type bit ne sont pas compatibles avec cette fonction.

SUM(ALL expression) : évalue l'expression puis renvoie la somme des valeurs non nulles «NULL». «ALL» est la valeur par défaut.

SUM(DISTINCT expression) évalue l'expression puis renvoie la somme des valeurs distinctes non nulles «NULL».

AVG

Cette fonction renvoie la moyenne des valeurs relatives à l'expression. Elle ignore toutes les valeurs nulles «NULL», et ne peut être utilisée que sur des colonnes contenant des valeurs numériques.

expression : fait référence au nom d'une colonne, au nom d'une fonction, à une expression arithmétique ou de concaténation, ou encore à une constante. Cette fonction ne peut être appliquée que sur des valeurs numériques. Les fonctions d'agrégation, les sous-requêtes et le type bit ne sont pas compatibles avec cette fonction.

AVG(ALL expression) : évalue l'expression puis renvoie la moyenne des valeurs non nulles «NULL». «ALL» est la valeur par défaut.

AVG(DISTINCT expression) évalue l'expression puis renvoie la moyenne des valeurs distinctes non nulles «NULL».

F.1.c) Les options usuelles du «FROM»

FROM < table_source₁ , table_source₂, ..., table_source_n >

Options usuelles pour «table_source_i» :

```
{
  table_physique  [ [ AS ] tab_alias ]
| table_view      [ [ AS ] tab_alias ]
| table_dérivée   [ [ AS ] tab_alias ] [ ( col1, col2, ... colp ) ]
| <jointure>
}
```

table_physique : fait référence à une table physique existante,

table_view : fait référence à une vue prédéfinie (view : table virtuelle),

table_dérivée : fait référence à une sous requête faisant office de table virtuelle,

<jointure> : expression exprimant les jointures entre des tables.

Une «**table_physique**» est une table dont la structure et les données sont mémorisés de façon permanente sur le disque dur de l'ordinateur. Pour les exemples de requêtes utilisant des tables physiques, prière de se référer aux exemples cités plus haut (en page 22).

Une «**table_view**» est une table virtuelle associée aux résultats d'une requête «select» prédéfinie. En effet, contrairement à une «view» qui est ni plus ni moins qu'une requête «select» portant un nom et préenregistrée physiquement au niveau de la base de données, la «table_view» représente le résultat obtenu suite à l'exécution de la «view» (vue en français) au niveau de la mémoire virtuelle. Ce résultat n'est visible que par la requête faisant appel à cette «table_view», et sa durée de vie est par conséquent limitée à cet appel. Autrement dit, on parlera d'une vue comme étant le résultat d'une requête préenregistrée qui peut être exploité par d'autres requêtes comme étant une table temporaire qui existera - au niveau de la RAM - le temps d'exécution de ces requêtes. Pour les exemples de requêtes utilisant des vues, prière de se référer aux exemples cités dans la section (G.3 plus bas).

Une «**table_dérivée**» est identique à une «table_view» à la différence majeure qu'elle n'est pas associée à une requête préenregistrée. Son code est imbriqué à l'intérieur d'une requête appelante, on parle alors de sous-requête. Notons qu'avec les tables dérivées, on peut utiliser plusieurs niveaux d'imbrication.

Une «**jointure**» fait référence aux instructions de jointure permettant de lier les «table_source_i» entre elles, à travers une syntaxe spécifique. Pour les exemples de requêtes utilisant des jointures, veuillez-vous référer aux exemples cités plus haut.

Exemple 7 : Le «FROM» utilisant une table dérivée constituée de valeurs constantes

```
SELECT
    IdMot, Mot
FROM
    (
        VALUES (1, 'Bonjour'), (2, 'Tout'), (3, 'Le'), (4, 'Monde')
    ) AS TableDerivee (IdMot, Mot)
```

Résultat :

IdMot	Mot
1	Bonjour
2	Tout
3	Le
4	Monde

«TableDerivee», «IdMot» et «Mot» sont des alias représentant respectivement : le nom de l'ensemble constitué par les couples de valeurs, le nom associé à la première colonne de chaque couple de valeurs, le nom associé à la deuxième colonne de chaque couple de valeurs. «TableDerivee» est donc assimilée à une variable de type table, créée virtuellement à partir d'une liste fixe de valeurs pour être exploitée durant l'exécution du «SELECT». La portée et la durée de vie de cette table sont donc limitées au processus de la requête appelante.

Exemple 8 : Le «FROM» utilisant une table dérivée non constante (sous-requête)

```
SELECT
    TC.IdClient, TC.NomClient, TC.VilleClient
FROM
    (
        SELECT
            *
        FROM
            TCLIENTS
    ) AS TC
```

Résultat :

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
2	NB	SALE
3	NC	RABAT
4	ND	CASABLANCA
5	NE	SALE

Exemple 9 : Le «FROM» utilisant une jointure entre une table et une sous-requête

```
SELECT
    TCLIENTS.IdClient, NomClient, VilleClient
FROM
    TCLIENTS,
    (
        SELECT DISTINCT
            IdClient
        FROM
            TFACTURES
    ) AS TF
WHERE
    TCLIENTS.IdClient = TF.IdClient
```

Résultat :

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
2	NB	SALE
3	NC	RABAT

Exemple 10 : Le «FROM» utilisant une vue (view)

```
CREATE VIEW TF AS
(
    SELECT DISTINCT
        IdClient
    FROM
        TFACTURES
)

-----

SELECT
    *
FROM
    TF
```

Résultat :

IdClient
1
2
3

Exemple 11 : Le «FROM» utilisant la jointure entre une table et une vue

```
CREATE VIEW TF AS          /* création d'une vue nommée TF*/
(
  SELECT DISTINCT
    IdClient
  FROM
    TFACTURES
)

-----

SELECT
  TCLIENTS.IdClient, NomClient, VilleClient
FROM
  TCLIENTS, TF
WHERE
  TCLIENTS.IdClient = TF.IdClient
```

Résultat :

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
2	NB	SALE
3	NC	RABAT

F.1.d) Les options usuelles du «WHERE»

WHERE < conditions_de_recherche >

Options usuelles pour «conditions_de_recherche» :

```
{
  { [ NOT ] <prédicat> | ( < conditions_de_recherche > ) }
  [ { AND | OR } [ NOT ] { <prédicat> | ( < conditions_de_recherche > ) } ]
  [ ...n ]
}
```

Options usuelles pour «prédicat» :

```
{
  {
    expression { = | < > | != | > | >= | < | <= } expression
    | expression [ NOT ] BETWEEN expression AND expression
    | expression_chaine [ NOT ] LIKE <modèle_chaine> [ ESCAPE 'caractère' ]
    | expression [ NOT ] IN (sous-requête | expression [, ...n ])
    | expression [ NOT ] EXISTS (sous-requête)
    | expression IS [ NOT ] NULL
    | scalaire { = | < > | != | > | >= | < | <= } [ SOME | ANY | ALL ] (sous-requête)
  }
}
```

Option usuelles pour «modèle_chaine» :

Modèle	Pouvant être remplacé par
%	Chaîne de de caractères quelconque (même vide),
_	Un caractère quelconque,
[chaine]	Un caractère parmi ceux figurant dans la chaine,
[^chaine]	Un caractère en dehors de ceux figurant dans la chaine,
[car ₁ –car _n]	Un caractère parmi ceux se trouvant dans l'intervalle car ₁ à car _n
[^car ₁ –car _n]	Un caractère en dehors de ceux se trouvant dans l'intervalle car ₁ à car _n

Exemple 12 : Le «WHERE» utilisant des opérateurs logiques et de comparaison

```

SELECT
    IdClient, NomClient, VilleClient
FROM
    TCLIENTS
WHERE
    ((IdClient >= 1) AND (IdClient < 3)) OR (IdClient >= 5))

```

Résultat :

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
2	NB	SALE
5	NE	SALE

Exemple 13 : Le «WHERE» utilisant BETWEEN

```

SELECT
    IdClient,
    NomClient,
    VilleClient
FROM
    TCLIENTS
WHERE
    IdClient BETWEEN 2 AND 4

```

Résultat :

IdClient	NomClient	VilleClient
2	NB	SALE
3	NC	RABAT
4	ND	CASABLANCA

Exemple 14 : Le «WHERE» utilisant LIKE

```

SELECT
    IdClient,
    NomClient,
    VilleClient
FROM
    TCLIENTS
WHERE
    VilleClient LIKE 'Casablanca'

```


Résultat :

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
4	ND	CASABLANCA

On aurait pu obtenir le même résultat avec l'opérateur égal (de comparaison)

```
SELECT
    IdClient,
    NomClient,
    VilleClient
FROM
    TCLIENTS
WHERE
    VilleClient LIKE '%' + 'ab' + '%'
```

Résultat : Toutes les villes contenant la chaîne de caractères 'ab'

IdClient	NomClient	VilleClient
1	NA	CASABLANCA
3	NC	RABAT
4	ND	CASABLANCA

On aurait pu obtenir le même résultat avec LIKE '%ab%'

Exemple 15 : Le «WHERE» utilisant IN

```
SELECT
    IdClient ,
    NomClient ,
    VilleClient
FROM
    TCLIENTS
WHERE
    IdClient IN (2, 4, 5)
```

Résultat :

IdClient	NomClient	VilleClient
2	NB	SALE
4	ND	CASABLANCA
5	NE	SALE

```
SELECT
    IdClient ,
    NomClient ,
    VilleClient
FROM
    TCLIENTS
WHERE
    IdClient NOT IN (SELECT DISTINCT IdClient FROM TFACTURES)
```

Résultat : Tous les clients qui n'ont pas de factures

IdClient	NomClient	VilleClient
4	ND	CASABLANCA
5	NE	SALE

Exemple 16 : Le «WHERE» utilisant EXISTS

```
SELECT
    C.IdClient ,
    C.NomClient ,
    C.VilleClient
FROM
    TCLIENTS AS C
WHERE
    NOT EXISTS (
        SELECT F.IdClient
        FROM   TFACTURES AS F
        WHERE  F.IdClient = C.IdClient
    )
```

Résultat : Tous les clients qui n'ont pas de factures

IdClient	NomClient	VilleClient
4	ND	CASABLANCA
5	NE	SALE

Exemple 17 : Le «WHERE» utilisant IS NULL

```
SELECT
    TCLIENTS.IdClient ,
    NomClient ,
    VilleClient
FROM
    TCLIENTS LEFT JOIN TFACTURES ON TCLIENTS.IdClient = TFactures.IdClient
WHERE
    TFactures.IdClient IS NULL
```

Résultat : Tous les clients qui n'ont pas de factures

IdClient	NomClient	VilleClient
4	ND	CASABLANCA
5	NE	SALE

Exemple 18 : Le «WHERE» utilisant SOME (alias de ANY)

```
SELECT DISTINCT
    IdClient
FROM
    TFACTURES,
    TCOMMANDES
WHERE
    TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
    IdProduit = SOME (
        SELECT
            IdProduit
        FROM
            TPRODUITS
        WHERE
            PrixUnitaireproduit BETWEEN 20 AND 40
    )
```

Résultat : Clients ayant acheté un produit dont le prix est compris entre 20 et 40

IdClient
1
2
3

Exemple 19 : Le «WHERE» utilisant ALL (incompatible avec Order By et Into dans un Select)

```
SELECT DISTINCT
    IdClient
FROM
    TFACTURES,
    TCOMMANDES
WHERE
    TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
    Qt_cmd >= ALL ( SELECT DISTINCT
                    Qt_cmd
                    FROM
                        TCOMMANDES
                    WHERE
                        IdProduit IN (1 , 2, 3)
                    )
```

Résultat : Clients ayant commandé une quantité d'un produit, supérieure ou égale à celles relatives aux produits 1, 2 et 3.

IdClient
1
3

F.1.e) Les options usuelles du «GROUP BY»

```
GROUP BY
{
    [ < colonne1 , ... , colonnen> ]
    | [ {ROLLUP | CUBE | GROUPING SETS}(< colonne1 , ... , colonnen>) ]
}
```

Le «GROUP BY» est une instruction qui s'exécute après l'évaluation des conditions liées à la clause «WHERE», et qui permet d'effectuer des regroupements de lignes selon les valeurs d'une ou de plusieurs colonnes. En effet, toutes les valeurs identiques sur une colonne ou sur plusieurs colonnes seront ramenées à une seule ligne, avec la possibilité d'effectuer des calculs sur les lignes objet du groupement.

< colonne_i> : Désigne la colonne d'une table, d'une table dérivée ou encore d'une vue. Les tables, les tables dérivées ou encore les vues contenant ces colonnes doivent obligatoirement figurer dans la clause «FROM». Toute colonne non calculable figurant dans le «SELECT» doit figurer obligatoirement dans la clause «GROUP BY». Toutefois, les alias figurant dans le «SELECT» ne sont pas autorisés, exception faite pour les alias des tables dérivées figurant dans la clause «FROM». Par ailleurs, les tables dérivées et les colonnes de type image, text ou ntext, ne sont pas autorisées.

<ROLLUP> : Permet d'évaluer la fonction d'agrégation pour chaque combinaison de colonnes du groupement en retirant à chaque fois une colonne en partant de la droite vers la gauche pour évaluer la fonction d'agrégation.

A titre d'exemple, dans le cas de la fonction d'agrégation «SUM» définie dans le «SELECT», ROLLUP(colonne₁, colonne₂, colonne₃) crée des sous-totaux pour chaque combinaison de de colonnes en diminuant le nombre de colonnes de la droite vers la gauche :

colonne₁, colonne₂, colonne₃ : sous-total(par valeur du triplet (colonne₁, colonne₂, colonne₃))

colonne₁, colonne₂, NULL : sous-total(par valeur du doublet (colonne₁, colonne₂))

colonne₁, NULL, NULL : sous-total(par valeur de (colonne₁))

NULL, NULL, NULL : total global

<CUBE> : Permet d'évaluer la fonction d'agrégation pour chaque combinaison de colonnes du groupement.

A titre d'exemple, dans le cas de la fonction d'agrégation «SUM» définie dans le «SELECT», CUBE(colonne₁, colonne₂, colonne₃) crée des sous-totaux pour chaque combinaison de colonnes :

colonne₁, colonne₂, colonne₃ : sous-total(par valeur du triplet (colonne₁, colonne₂, colonne₃))

colonne₁, colonne₂, NULL : sous-total(par valeur du doublet (colonne₁, colonne₂))

colonne₁, NULL, colonne₃ : sous-total(par valeur du doublet (colonne₁, colonne₃))

NULL, colonne₂, colonne₃ : sous-total(par valeur du doublet (colonne₂, colonne₃))

colonne₁, NULL, NULL : sous-total(par valeur de (colonne₁))

NULL, colonne₂, NULL : sous-total(par valeur de (colonne₂))

NULL, NULL, colonne₃ : sous-total(par valeur de (colonne₃))

NULL, NULL, NULL : total global

<GROUPING SETS> : Permet d'évaluer la fonction d'agrégation pour chaque colonne du groupement.

A titre d'exemple, dans le cas de la fonction d'agrégation «SUM» définie dans le «SELECT», GROUPING SETS(colonne₁, colonne₂, colonne₃) crée des sous-totaux pour chaque colonne :

colonne₁, NULL, NULL : sous-total(par valeur de (colonne₁))

NULL, colonne₂, NULL : sous-total(par valeur de (colonne₂))

NULL, NULL, colonne₃ : sous-total(par valeur de (colonne₃))

L'exemple ci-dessous illustre le regroupement de la table de commandes selon le code de la facture

Table «TCommandes»

IdFacture	IdProduit	Qt_Cmd
1	1	10
1	2	20
1	3	30
2	2	10
2	3	20
2	4	30
3	3	10
3	4	20
3	5	30
4	1	5
4	2	10

4	3	15
5	2	5
5	3	10
5	4	15
6	3	20
6	4	20
6	5	20
7	3	3
7	4	4
7	5	5
8	1	10
8	2	20
8	3	30
9	2	2
9	3	3
9	4	4

Exemple 20 : Le «GROUP BY» sur une colonne, utilisant une seule table

```
SELECT
  IdFacture
FROM
  TCOMMANDES
GROUP BY
  IdFacture
```

```
SELECT
  IdFacture, COUNT(IdProduit) AS NBP
FROM
  TCOMMANDES
GROUP BY
  IdFacture
```

Résultat : groupement par IdFacture, puis en calculant le nombre de produits par facture

IdFacture
1
2
3
4
5
6
7
8
9

IdFacture	NBP
1	3
2	3
3	3
4	3
5	3
6	3
7	3
8	3
9	3

Exemple 21 : Le «GROUP BY» sur une colonne, utilisant deux tables

```
SELECT
  IdFacture ,
  SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS MF
FROM
  TCOMMANDES,
  TPRODUITS
WHERE
  TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY IdFacture
```

Résultat :

IdFacture	MF
1	1200,00
2	1555,00
3	1820,00
4	600,00
5	777,50
6	1720,00
7	354,00
8	1200,00
9	228,00

Exemple 22 : Le «GROUP BY» sur deux colonnes, utilisant plusieurs tables

```

SELECT
  IdClient ,
  TCOMMANDES.IdFacture,
  SUM (Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS MF
FROM
  TFACTURES, TCOMMANDES, TPRODUITS
WHERE
  TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
  TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
  IdClient , TCOMMANDES.IdFacture

```

Résultat : groupement par IdFacture et par IdClient et calcul du montant de la facture.

IdClient	IdFacture	MF
1	1	1200,00
1	2	1555,00
1	3	1820,00
2	4	600,00
2	5	777,50
2	6	1720,00
3	7	354,00
3	8	1200,00
3	9	228,00

Exemple 23 : Le «GROUP BY» sur deux colonnes avec «ROLLUP», utilisant plusieurs tables

```

SELECT
  IdClient, TFACTURES.IdFacture,
  SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS MF
FROM
  TFACTURES, TCOMMANDES, TPRODUITS
WHERE
  TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
  TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
  ROLLUP(IdClient , TFACTURES.IdFacture)

```

Résultat : groupement par IdFacture et par IdClient avec «ROLLUP» pour calculer les montants des factures, le chiffre d'affaire par client, puis le chiffre d'affaire réalisé avec l'ensemble des clients.

IdClient	IdFacture	MF
1	1	1200,00
1	2	1555,00
1	3	1820,00
1	NULL	4575,00
2	4	600,00
2	5	777,50
2	6	1720,00
2	NULL	3097,50
3	7	354,00
3	8	1200,00
3	9	228,00
3	NULL	1782,00
NULL	NULL	9454,50

Exemple 24 : Le «GROUP BY» sur deux colonnes avec «CUBE», utilisant plusieurs tables

```

SELECT
  IdClient, TFACTURES.IdFacture,
  SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS MF
FROM
  TFACTURES, TCOMMANDES, TPRODUITS
WHERE
  TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
  TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
  CUBE(IdClient , TFACTURES.IdFacture)

```

Résultat : groupement par IdFacture et par IdClient avec «CUBE» pour calculer les montants des factures, le chiffre d'affaire par client, puis le chiffre d'affaire réalisé avec l'ensemble des clients.

IdClient	IdFacture	MF
1	1	1200,00
NULL	1	1200,00
1	2	1555,00
NULL	2	1555,00
1	3	1820,00
NULL	3	1820,00
2	4	600,00
NULL	4	600,00
2	5	777,50
NULL	5	777,50
2	6	1720,00
NULL	6	1720,00
3	7	354,00

NULL	7	354,00
3	8	1200,00
NULL	8	1200,00
3	9	228,00
NULL	9	228,00
NULL	NULL	9454,50
1	NULL	4575,00
2	NULL	3097,50
3	NULL	1782,00

Exemple 25 : Le «GROUP BY» utilisant «GROUPING SETS»

```

SELECT
  idClient,
  TFACTURES.IdFacture,
  DATEPART(yyyy,DateFacturation) AS Année,
  SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS CA_MF
FROM
  TFACTURES,
  TCOMMANDES,
  TPRODUITS
WHERE
  TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
  TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
  GROUPING SETS(idClient, TFACTURES.IdFacture, DATEPART(yyyy,DateFacturation))

```

Résultat : calcul des montants des factures, du chiffre d'affaire par année et du chiffre d'affaire par client.

IdClient	IdFacture	Année	CA_MF
NULL	NULL	2016	2755,00
NULL	NULL	2017	3197,50
NULL	NULL	2018	3502,00
NULL	1	NULL	1200,00
NULL	2	NULL	1555,00
NULL	3	NULL	1820,00
NULL	4	NULL	600,00
NULL	5	NULL	777,50
NULL	6	NULL	1720,00
NULL	7	NULL	354,00
NULL	8	NULL	1200,00
NULL	9	NULL	228,00
1	NULL	NULL	4575,00
2	NULL	NULL	3097,50
3	NULL	NULL	1782,00

Exemple 26 : Le «GROUP BY» sur une expression, utilisant plusieurs tables et «ORDER BY»

```
SELECT
    DATEPART(yyyy,DateFacturation) AS Année,
    SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS CA
FROM
    TFACTURES, TCOMMANDES, TPRODUITS
WHERE
    TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
    TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
    DATEPART(yyyy,DateFacturation)
ORDER BY
    Année ASC
```

Résultat : calcul du chiffre d'affaire par année.

Année	CA
2016	2755,00
2017	3197,50
2018	3502,00

Exemple 27 : Le «GROUP BY» utilisant «HAVING» et «ORDER BY»

```
SELECT
    DATEPART(yyyy,DateFacturation) AS Année,
    SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS CA
FROM
    TFACTURES, TCOMMANDES, TPRODUITS
WHERE
    TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
    TCOMMANDES.IdProduit = TPRODUITS.IdProduit AND
    DATEPART(yyyy,DateFacturation) > 2016
GROUP BY
    DATEPART(yyyy, DateFacturation)
HAVING
    SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) > 3000
ORDER BY
    CA DESC
```

Résultat : calcul du chiffre d'affaire par année avec des conditions dans les clauses «WHERE» et «HAVING»

Année	CA
2018	3502,00
2017	3197,50

F.1.f) L'option «INTO»

```
INTO < nom_nouvelle_table >
```

Le «INTO» est une instruction qui permet de créer une nouvelle table physique, dont les attributs seront ceux qui figurent dans la clause «SELECT».

Exemple 28 : Le «SELECT INTO» utilisant «GROUP BY» et «IDENTITY»

```
SELECT
  IDENTITY(INT , 1, 1) AS IdNewTab,
  IdClient,
  DesignationProduit,
  SUM(Qt_Cmd*PrixUnitaireProduit*(1 + TvaProduit)) AS CAP
INTO
  NewTab
FROM
  TFACTURES, TCOMMANDES, TPRODUITS
WHERE
  TFACTURES.IdFacture = TCOMMANDES.IdFacture AND
  TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
  IdClient,
  DesignationProduit
GO
SELECT * FROM NewTab
```

Résultat : création d'une nouvelle table « NewTab » contenant les chiffres d'affaires des clients par produit.

IdNewTab	IdClient	DesignationProduit	CAP
1	1	PA	120,00
2	2	PA	60,00
3	3	PA	120,00
4	1	PB	585,00
5	2	PB	292,50
6	3	PB	429,00
7	1	PC	1380,00
8	2	PC	1035,00
9	3	PC	828,00
10	1	PD	1500,00
11	2	PD	1050,00
12	3	PD	240,00
13	1	PE	990,00
14	2	PE	660,00
15	3	PE	165,00

F.2. Syntaxe de l'«INSERT»

L'«INSERT» permet d'ajouter des lignes à une table. Pour illustrer les différentes manières d'utilisation de cette instruction, nous allons nous appuyer sur la table «NewTab» créée précédemment en marquant le champ «IdNewTab» comme étant clé primaire. Rappelons que ce dernier champ était définie uniquement comme un champ auto-incrémentale et non comme une clé primaire.

Par ailleurs, en plus de l'insertion des lignes dans la table de destination, le processus d'insertion stocke les lignes insérées dans une table virtuelle spéciale nommée «INSERTED». Cette table adopte la structure et les attributs de la table d'origine. En effet, les colonnes de cette table sont automatiquement mappées sur les colonnes de la table sur laquelle est effectuée l'insertion.

Notons que dans les différents exemples que nous allons présenter dans cette section, les opérations d'insertion n'ont pas forcément un sens logique et/ou fonctionnel. Elles sont présentées uniquement pour énumérer les différentes possibilités d'ordre syntaxique.

Exemple 29 : Insertion d'une nouvelle ligne constante.

```
INSERT INTO NewTab
(IdClient, DesignationProduit, CAP)
VALUES
(20, 'PA', 200)
```

Exemple 30 : Insertion de plusieurs lignes constantes.

```
INSERT INTO NewTab
(IdClient, DesignationProduit, CAP)
VALUES
(20, 'PA', 200), (30, 'PB', 300), (40, 'PC', 400)
```

Exemple 31 : Insertion de plusieurs lignes constantes sans spécifier les noms des colonnes.

- Les valeurs doivent être fournies dans l'ordre des champs dans la table.

```
INSERT INTO NewTab
VALUES
(20, 'PA', 200), (30, 'PB', 300), (40, 'PC', 400)
```

Exemple 32 : Insertion de plusieurs lignes constantes en spécifiant les valeurs pour la clé primaire.

- Les valeurs 32, 33, et 34 pour la clé primaire ne doivent pas exister dans la table.

```
SET IDENTITY_INSERT NewTab ON
INSERT INTO NewTab
(IdNewTab, IdClient, DesignationProduit, CAP)
VALUES
(32, 2, 'PA', 20), (33, 3, 'PA', 30), (34, 4, 'PA', 40)
SET IDENTITY_INSERT NewTab OFF
```

Exemple 33 : Insertion de lignes constantes sans respecter l'ordre des colonnes dans la table.

- Les valeurs 35, 36, et 37 pour la clé primaire ne doivent pas exister dans la table.

```
SET IDENTITY_INSERT NewTab ON
INSERT INTO NewTab
(DesignationProduit, IdClient, CAP, IdNewTab)
VALUES
('PA', 2, 20, 35), ('PA', 3, 30, 36), ('PA', 4, 40, 37)
SET IDENTITY_INSERT NewTab OFF
```

Exemple 34 : Insertion à travers un «SELECT» sans préciser les colonnes de destination.

- L'ordre des colonnes dans le select doit correspondre à celui de la table.
- La colonne de la clé primaire ne doit pas être insérée (risque de doublons)

```
INSERT INTO NewTab
SELECT
    IdClient, DesignationProduit, CAP
FROM
    NewTab
WHERE
    IdNewTab > 30
```

Exemple 35 : Insertion à travers un «SELECT» en précisant les colonnes de destination.

```
INSERT INTO NewTab
    (IdClient, DesignationProduit, CAP)
SELECT
    IdClient, DesignationProduit, CAP
FROM
    NewTab
WHERE
    IdNewTab > 30
```

Exemple 36 : Insertion à travers un «SELECT» sans respecter l'ordre des colonnes dans la table.

```
INSERT INTO NewTab
    (DesignationProduit , IdClient, CAP)
SELECT
    DesignationProduit , IdClient, CAP
FROM
    NewTab
WHERE
    IdNewTab > 30
```

Exemple 37 : Insertion à travers un «SELECT» en utilisant «INSERT TOP» sans «ORDER BY»

- Order by dans l'insert n'a aucun impact sur l'ordre des lignes qui vont être ajoutées.

```
INSERT TOP(3) INTO NewTab
    (DesignationProduit , IdClient, CAP)
SELECT
    DesignationProduit , IdClient, CAP
FROM
    NewTab
WHERE
    IdNewTab > 10
```

Exemple 38 : Insertion à travers un «SELECT TOP» en utilisant «INSERT» avec «ORDER BY»

- Order by dans le select à un impact sur l'ordre des lignes qui vont être ajoutées.

```
INSERT INTO NewTab
    (DesignationProduit , IdClient, CAP)
SELECT TOP(3)
    DesignationProduit , IdClient, CAP
FROM
    NewTab
WHERE
    IdNewTab > 10
ORDER BY
    CAP DESC
```

Exemple 39 : Insertion et affichage des lignes ajoutées avec «OUTPUT INSERTED.*»

```
- INSERTED est le nom de la table temporaire qui contient les lignes ajoutées ou modifiées.  
INSERT INTO NewTab  
  (DesignationProduit , IdClient, CAP)  
OUTPUT INSERTED.*  
SELECT TOP(3)  
  DesignationProduit , IdClient, CAP  
FROM  
  NewTab  
WHERE  
  IdNewTab > 10  
ORDER BY  
  CAP DESC
```

F.3. Syntaxe de l'«UPDATE»

L'«UPDATE» permet de mettre à jour les lignes d'une table. Pour illustrer les différentes manières d'utilisation de cette instruction, nous allons nous appuyer sur la table «NewTab» créée précédemment en marquant le champ «IdNewTab» comme étant une clé primaire. Rappelons que ce dernier champ était définie uniquement comme un champ auto-incrémentale et non comme une clé primaire.

Par ailleurs, le processus de modification stocke les lignes avant leur modification dans une table virtuelle spéciale nommée «DELETED», et les lignes modifiées dans une table virtuelle nommée «INSERTED». Ces tables adoptent la structure et les attributs de la table d'origine. En effet, les colonnes des tables «INSERTED» et «DELETED» sont automatiquement mappées sur les colonnes de la table sur laquelle est effectuée la mise à jour.

Notons que dans les différents exemples que nous allons présenter dans cette section, les opérations de modification n'ont pas forcément un sens logique et/ou fonctionnel. Elles sont présentées uniquement pour énumérer les différentes possibilités d'ordre syntaxique.

Exemple 40 : Modification de toutes les valeurs d'une colonne sans la clause «WHERE».

```
UPDATE NewTab  
SET CAP = CAP * 1.20
```

Exemple 41 : Modification des données conditionnée à travers la clause «WHERE».

```
UPDATE NewTab  
SET  
  CAP = 100.00 ,  
  DesignationProduit = 'PA',  
  IdClient = 3  
WHERE  
  IdNewTab = 11
```

Exemple 42 : Modification des données en utilisant «DEFAULT».

```
/* DEFAULT fait référence en priorité à la valeur par défaut, mais si aucune valeur par défaut n'est définie et que  
NULL est autorisé, c'est le NULL qui sera attribué au champ CAP */  
UPDATE NewTab  
SET  
  CAP = DEFAULT  
WHERE IdNewTab > 15
```

Exemple 43 : Modification des données en utilisant la clause «TOP» sans la clause «ORDER BY».

```
- TOP (3) renvoi les 3 premières lignes de façon aléatoire
UPDATE
  TOP (3) NewTab
SET
  CAP = CONVERT(INT, CAP)
```

Exemple 44 : Modification des données en utilisant la clause «TOP» avec la clause «ORDER BY».

```
- TOP (3) dans le select renvoi les 3 premières lignes selon le classement
UPDATE
  NewTab
SET
  CAP = CONVERT(INT, CAP)
FROM
  (SELECT TOP (3) IdNewTab FROM NewTab ORDER BY CAP DESC) AS NT
WHERE
  NT.IdNewTab = NewTab.IdNewTab
```

Exemple 45 : Modification des données en utilisant les clauses «TOP», «ORDER BY» avec «OUTPUT».

```
- TOP (3) dans le select renvoi les 3 premières lignes selon le classement
- OUTPUT Affiche les lignes modifiées à partir de deleted et inserted
UPDATE
  NewTab
SET
  CAP = CONVERT(INT, CAP)
OUTPUT
  DELETED.IdNewTab, DELETED.CAP AS OLD_CAP, INSERTED.CAP AS NEW_CAP
FROM
  (SELECT TOP (3) IdNewTab FROM NewTab ORDER BY CAP DESC) AS NT
WHERE
  NT.IdNewTab = NewTab.IdNewTab
```

Exemple 46 : Modification des données à travers une sous-requête retournant un scalaire.

```
- OUTPUT Affiche les lignes modifiées à partir de deleted et inserted
UPDATE
  NewTab
SET
  CAP = (
    SELECT
      AVG(T.MCAP)
    FROM
      (
        SELECT MIN(CAP) AS MCAP
        FROM NewTab
        GROUP BY DesignationProduit
      ) AS T
  )
OUTPUT DELETED.IdNewTab, DELETED.CAP AS OLD_CAP, INSERTED.CAP AS NEW_CAP
WHERE
  IdNewTab IN (13, 14, 15)
```

Exemple 47 : Modification des données à travers une fonction.

```
/*Création d'une fonction retournant le chiffre d'affaire par client et par produit*/
CREATE FUNCTION fct_ca_par_client_produit
(
    @IdClient          AS INT,
    @DesignationProduit AS NVARCHAR(30)
)
RETURNS FLOAT
AS
BEGIN
    DECLARE @CAP AS FLOAT
    SELECT
        @CAP = SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit))
    FROM
        TFACTURES
        JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
        JOIN TPRODUITS  ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
    WHERE
        IdClient = @IdClient AND
        DesignationProduit = @DesignationProduit
    RETURN @CAP
END
/*Mise à jour de la table à travers la fonction, ne pas oublier d'utiliser le préfixe dbo */
UPDATE
    NewTab
SET
    CAP = dbo.fct_ca_par_client_produit(IdClient , DesignationProduit)
```

Exemple 48 : Modification des données à travers une sous-requête retournant plusieurs lignes.

```
UPDATE NewTab
SET
    NewTab.CAP = TSR.NCAP
FROM
    ( SELECT
        DesignationProduit,
        SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS NCAP
    FROM TFACTURES
        JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
        JOIN TPRODUITS  ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
    GROUP BY
        DesignationProduit
    ) AS TSR
WHERE
    NewTab.DesignationProduit = TSR.DesignationProduit
```

Exemple 49 : Modification des données à travers une «VIEW».

```
/*La vue peut porter sur plusieurs tables, mais la modification ne doit porter que sur les colonnes d'une seule table*/
CREATE VIEW View_NewTab AS /* Création d'une vue nommée View_NewTab */
(
    SELECT
        *
    FROM
        NewTab
    WHERE
        IdNewTab > 15
)
```

```

UPDATE
    View_NewTab
SET
    CAP = 100

```

Exemple 50 : Modification des données à travers un alias.

```

UPDATE
    NT
SET
    NT.CAP = 100
FROM
    NewTab AS NT
    JOIN TCLIENTS      ON NT.IdClient = TCLIENTS.IdClient
    JOIN TFACTURES     ON TCLIENTS.IdClient = TFACTURES.IdClient
    JOIN TCOMMANDES    ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
    JOIN TPRODUITS     ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
WHERE
    NT.IdClient IN (1, 3) AND
    TvaProduit = 0.2      AND
    Qt_Cmd >= 10

```

Exemple 51 : Modification des données dans une variable de type table.

```

/*La modification des données dans une variable de type table, n'a aucune répercussion sur la table physique*/
/*Déclaration d'une variable locale de type table*/
DECLARE @VarTab TABLE (
    VarIdTab      INT,
    VarIdClient   INT,
    VarDesignation NVARCHAR(30) ,
    VarCAP        FLOAT
)
/*Insertion des données dans la variable locale*/
INSERT INTO @VarTab SELECT * FROM NewTab WHERE IdNewTab > 15
/*Mise à jour des données dans la variable locale*/
UPDATE
    @VarTab
SET
    VarCAP += 10
/*Les données ont été modifiées au niveau de la variable locale*/
SELECT * FROM @VarTab
/*Les données restent inchangées au niveau de la table physique*/
SELECT * FROM NewTab WHERE IdNewTab > 15

```

F.4. Syntaxe du «DELETE»

Le «DELETE» permet de supprimer les lignes d'une table. Pour illustrer les différentes manières d'utilisation de cette instruction, nous allons nous appuyer sur la table «NewTab» créée précédemment en marquant le champ «IdNewTab» comme clé primaire. Rappelons que ce dernier champ était définie uniquement comme un champ auto-incrémentale et non comme une clé primaire.

Par ailleurs, le processus de suppression stocke les lignes supprimées dans une table virtuelle spéciale nommée «DELETED». Cette table adopte la structure et les attributs de la table d'origine. En effet, les colonnes de la table «DELETED» sont automatiquement mappées sur les colonnes de la table sur laquelle est effectuée la suppression.

Notons que dans les différents exemples que nous allons présenter dans cette section, les opérations de suppression n'ont pas forcément un sens logique et/ou fonctionnel. Elles sont présentées uniquement pour énumérer les différentes possibilités d'ordre syntaxique.

Exemple 52 : Suppression de toutes les lignes de la table.

DELETE NewTab		TRUNCATE TABLE NewTab
----------------------	---	------------------------------

Exemple 53 : Suppression conditionnée à travers la clause «WHERE».

DELETE NewTab WHERE IdNewTab = 11
--

Exemple 54 : Suppression utilisant la clause «TOP» sans la clause «ORDER BY».

- TOP (3) renvoi les 3 premières lignes de façon aléatoire DELETE TOP (3) NewTab
--

Exemple 55 : Suppression utilisant la clause «TOP» avec la clause «ORDER BY».

- TOP (3) dans le select renvoi les 3 premières lignes selon le classement DELETE NewTab FROM (SELECT TOP (3) IdNewTab FROM NewTab ORDER BY CAP DESC) AS NT WHERE NT.IdNewTab = NewTab.IdNewTab
--

Exemple 56 : Suppression utilisant les clauses «TOP», «ORDER BY» avec «OUTPUT».

- TOP (3) dans le select renvoi les 3 premières lignes selon le classement - OUTPUT Affiche les lignes supprimées à partir de deleted DELETE NewTab OUTPUT DELETED.* FROM (SELECT TOP (3) IdNewTab FROM NewTab ORDER BY CAP DESC) AS NT WHERE NT.IdNewTab = NewTab.IdNewTab
--

Exemple 57 : Suppression conditionnée à travers une sous-requête retournant un scalaire.

- OUTPUT Affiche les lignes supprimées à partir de deleted DELETE NewTab OUTPUT DELETED.* WHERE CAP >= (SELECT AVG(T.MCAP) FROM (SELECT MIN(CAP) AS MCAP FROM NewTab GROUP BY DesignationProduit) AS T))
--

Exemple 58 : Suppression conditionnée à travers une fonction retournant un scalaire.

```
/*Création d'une fonction retournant le chiffre d'affaire moyen par client et par produit*/
CREATE FUNCTION fct_ca_moy_par_client_produit
(
    @IdClient          AS INT,
    @DesignationProduit AS NVARCHAR(30)
)
RETURNS FLOAT
AS
BEGIN
    DECLARE @CAMP AS FLOAT
    SELECT
        @CAMP = AVG(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit))
    FROM
        TFACTURES
        JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
        JOIN TPRODUITS  ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
    WHERE
        IdClient = @IdClient AND
        DesignationProduit = @DesignationProduit
    RETURN @CAMP
END
/*Suppression conditionnée à travers le calcul de la fonction, ne pas oublier le préfixe dbo */
DELETE
    NewTab
WHERE
    dbo.fct_ca_moy_par_client_produit (IdClient , DesignationProduit) <= 150
```

Exemple 59 : Suppression à travers une sous-requête retournant plusieurs lignes.

```
DELETE
    NewTab
FROM
    (
        SELECT
            DesignationProduit,
            SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS NCAP
        FROM TFACTURES
            JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
            JOIN TPRODUITS  ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
        GROUP BY
            DesignationProduit
    ) AS TSR
WHERE
    NewTab.DesignationProduit = TSR.DesignationProduit AND
    NewTab.CAP BETWEEN 0.5*TSR.NCAP AND TSR.NCAP
```

Exemple 60 : Suppression à travers une «VIEW».

```
/*La vue peut porter sur plusieurs tables, mais la suppression ne doit porter que sur les lignes d'une seule table*/
CREATE VIEW View_NewTab AS /* Création d'une vue nommée View_NewTab */
(
    SELECT
        *
    FROM
        NewTab
    WHERE
        IdNewTab > 15
)
```

```
DELETE View_NewTab
```

Exemple 61 : Suppression à travers un alias.

```
DELETE
  NT
FROM
  NewTab AS NT
  JOIN TFACTURES ON NT.IdClient = TFACTURES.IdClient
  JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
  JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
WHERE
  NT.DesignationProduit = TPRODUITS.DesignationProduit AND
  TvaProduit < 0.2 AND
  Qt_Cmd > 20
```

Exemple 62 : Suppression de données dans une variable de type table.

```
/*La suppression des données dans une variable de type table, n'a aucune répercussion sur la table physique*/
/*Déclaration d'une variable locale de type table*/
DECLARE @VarTab TABLE (
    VarIdTab          INT,
    VarIdClient       INT,
    VarDesignation    NVARCHAR(30) ,
    VarCAP            FLOAT
)
/*Insertion des données dans la variable locale*/
INSERT INTO @VarTab SELECT * FROM NewTab
/*Suppression des lignes dans la variable locale*/
DELETE
  @VarTab
WHERE
  VarIdTab > 15
/*Les lignes ont été supprimées au niveau de la variable locale*/
SELECT * FROM @VarTab
/*Les lignes restent inchangées au niveau de la table physique*/
SELECT * FROM NewTab WHERE IdNewTab > 15
```

G. Le TSQL pour manipuler les structures (Data Definition Language)

G.1. Manipulations usuelles des bases de données

G.1.a) Le «CREATE DATABASE»

Exemple 63 : Création d'une base de données avec les valeurs par défaut.

```
IF DB_ID ('TEST') IS NOT NULL /*Teste si la base 'TEST' existe déjà*/  
    DROP DATABASE TEST        /*Supprime la base 'TEST' sil elle existe*/  
  
CREATE DATABASE TEST          /*Création de la base 'TEST' */
```

Cela revient à définir les valeurs par défaut pour un certain nombre de paramètres :

```
CREATE DATABASE TEST  
ON /*Création du fichier de données*/  
(  
    NAME          = test ,  
    FILENAME      = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER  
                    \MSSQL\DATA\test.mdf' ,  
    SIZE          = 4160KB, /*Taille de la base au départ*/  
    MAXSIZE       = UNLIMITED, /*Taille maximale que la base peut atteindre*/  
    FILEGROWTH    = 1024KB /*Augmentation automatique de taille de la base*/  
)  
LOG ON /*Création du fichier journal*/  
(  
    NAME          = test_log,  
    FILENAME      = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER  
                    \MSSQL\DATA\test_log.ldf',  
    SIZE          = 1040KB, /*Taille du journal au départ*/  
    MAXSIZE       = 2048GB, /*Taille maximale du journal*/  
    FILEGROWTH    = 10% /*Augmentation automatique de la taille du journal*/  
)
```

Exemple 64 : Création d'une base avec des valeurs utilisateur dans le groupe par défaut.

```
IF DB_ID ('TEST') IS NOT NULL  
    DROP DATABASE TEST  
  
CREATE DATABASE TEST  
ON PRIMARY /*Création du fichier de données dans le groupe primaire*/  
(  
    NAME          = test_data ,  
    FILENAME      = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER  
                    \MSSQL\DATA\test_data.mdf' ,  
    SIZE          = 5MB , /*Taille de la base au départ*/  
    MAXSIZE       = 50MB, /*Taille maximale que la base peut atteindre*/  
    FILEGROWTH    = 1MB /*Augmentation automatique de taille de la base*/  
)  
LOG ON /*Création du fichier journal*/  
(  
    NAME          = test_log,  
    FILENAME      = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER  
                    \MSSQL\DATA\test_log.ldf' ,  
    SIZE          = 1MB , /*Taille du journal au départ*/  
)
```

```

MAXSIZE      = 50MB, /*Taille maximale du journal*/
FILEGROWTH   = 1MB   /*Augmentation automatique de la taille du journal*/
)

```

Exemple 65 : Création d'une base avec plusieurs fichiers de données et de journaux.

```

IF DB_ID ('TEST') IS NOT NULL
    DROP DATABASE TEST

CREATE DATABASE TEST
ON PRIMARY
(
    NAME          = test1_data ,
    FILENAME      = 'D:\DATA\test1_data.mdf' ,
    SIZE          = 5MB ,
    MAXSIZE       = 50MB,
    FILEGROWTH    = 1MB
),
(
    NAME          = test2_data ,
    FILENAME      = 'D:\DATA\test2_data.ndf' ,
    SIZE          = 5MB ,
    MAXSIZE       = 50MB,
    FILEGROWTH    = 1MB
),
(
    NAME          = test3_data ,
    FILENAME      = 'D:\DATA\test3_data.ndf' ,
    SIZE          = 5MB ,
    MAXSIZE       = 50MB,
    FILEGROWTH    = 1MB
)
LOG ON
(
    NAME          = test1_log,
    FILENAME      = 'E:\DATA\test1_log.ldf' ,
    SIZE          = 1MB ,
    MAXSIZE       = 50MB,
    FILEGROWTH    = 1MB
),
(
    NAME          = test2_log,
    FILENAME      = 'E:\DATA\test2_log.ldf' ,
    SIZE          = 1MB ,
    MAXSIZE       = 50MB,
    FILEGROWTH    = 1MB
)

```

Exemple 66 : Création d'une base avec plusieurs groupes de fichiers.

```

IF DB_ID ('TEST') IS NOT NULL
    DROP DATABASE TEST

CREATE DATABASE TEST
ON PRIMARY
(
    NAME          = test_pri_data ,
    FILENAME      = 'D:\DATA\test_pri_data.mdf' ,
    SIZE          = 5MB ,
    MAXSIZE       = 50MB,
    FILEGROWTH    = 1MB
)

```

```

) ,
(
    NAME          = tes_sec1_data ,
    FILENAME       = 'D:\DATA\test_sec1_data.ndf' ,
    SIZE           = 5MB ,
    MAXSIZE        = 50MB,
    FILEGROWTH     = 1MB
) ,
FILEGROUP FG1
(
    NAME          = test_fg1_sec1_data ,
    FILENAME       = 'D:\DATA\test_fg1_sec1_data.ndf' ,
    SIZE           = 5MB ,
    MAXSIZE        = 50MB,
    FILEGROWTH     = 1MB
) ,
(
    NAME          = test_fg1_sec2_data,
    FILENAME       = 'D:\DATA\test_fg1_sec2_data.ndf' ,
    SIZE           = 5MB ,
    MAXSIZE        = 50MB,
    FILEGROWTH     = 1MB
)
FILEGROUP FG2
(
    NAME          = test_fg2_sec1_data ,
    FILENAME       = 'D:\DATA\test_fg2_sec1_data.ndf' ,
    SIZE           = 5MB ,
    MAXSIZE        = 50MB,
    FILEGROWTH     = 1MB
)
LOG ON
(
    NAME          = test1_log,
    FILENAME       = 'E:\DATA\test1_log.ldf' ,
    SIZE           = 1MB ,
    MAXSIZE        = 50MB,
    FILEGROWTH     = 1MB
) ,
(
    NAME          = test2_log,
    FILENAME       = 'E:\DATA\test2_log.ldf' ,
    SIZE           = 1MB ,
    MAXSIZE        = 50MB,
    FILEGROWTH     = 1MB
)
)

```

G.1.b) L'«ALTER DATABASE»

Exemple 67 : Modification du nom de la base de données.

```
ALTER DATABASE TEST MODIFY NAME = TEST2
```

Exemple 68 : Modification du jeu de caractères.

```
ALTER DATABASE TEST COLLATE French_CI_AI
```

Exemple 69 : Ajout d'un fichier de données.

```
ALTER DATABASE TEST ADD FILE
(
  NAME          = tes_sec2_data ,
  FILENAME      = 'D:\DATA\test_sec2_data.ndf' ,
  SIZE          = 5MB ,
  MAXSIZE       = UNLIMITED,
  FILEGROWTH    = 5%
)
```

Exemple 70 : Ajout d'un groupe de deux fichiers de données.

```
ALTER DATABASE TEST ADD FILEGROUP FG3
ALTER DATABASE TEST ADD FILE
(
  NAME          = test_fg3_sec1_data,
  FILENAME      = 'D:\DATA\test_fg3_sec1_data.ndf' ,
  SIZE          = 1MB ,
  MAXSIZE       = UNLIMITED,
  FILEGROWTH    = 5%
),
(
  NAME          = test_fg3_sec2_data,
  FILENAME      = 'E:\DATA\test_fg3_sec2_data.ndf' ,
  SIZE          = 1MB ,
  MAXSIZE       = UNLIMITED,
  FILEGROWTH    = 5%
)
TO FILEGROUP FG3
```

Exemple 71 : Ajout de deux fichiers de journalisation.

```
ALTER DATABASE TEST ADD LOG FILE
(
  NAME          = test3_log,
  FILENAME      = 'E:\DATA\test3_log.ldf' ,
  SIZE          = 1MB ,
  MAXSIZE       = UNLIMITED,
  FILEGROWTH    = 5%
),
(
  NAME          = test4_log,
  FILENAME      = 'F:\DATA\test4_log.ldf' ,
  SIZE          = 1MB ,
  MAXSIZE       = UNLIMITED,
  FILEGROWTH    = 10%
)
```

Exemple 72 : Augmentation de la taille d'un fichier à 10 Mo.

```
ALTER DATABASE TEST MODIFY FILE
(
  NAME          = test1_data,
  SIZE          = 10MB ,
  MAXSIZE       = 60MB
)
```

Exemple 73 : Réduction de la taille d'un fichier à 6 Mo.

```
DBCC SHRINKFILE (test_data, 6)
```

Exemple 74 : Déplacement d'un fichier de donné vers un autre répertoire.

```
/*- il est nécessaire de détacher la base, de déplacer le fichier physiquement, puis d'attacher la base avant d'exécuter ce code */  
ALTER DATABASE TEST MODIFY FILE  
(  
    NAME          = test1_data,  
    FILENAME      = 'E:\DATABASE\test1_data.mdf' ,  
)
```

G.1.c) Le «DROP DATABASE»

Exemple 75 : Suppression d'une base de données.

```
DROP DATABASE TEST
```

Exemple 76 : Suppression de trois bases de données.

```
DROP DATABASE TEST1, TEST2, TEST3
```

G.2. Manipulations usuelles des tables

G.2.a) Le «CREATE TABLE»

Syntaxe :

```
CREATE TABLE nom_table  
(  
    {  
        < definition_et_contrainte_niveau_colonne >  
        | < contraintes_niveau_table > [, ... ]  
        | < index_niveau_table >  
    }  
    [, ... ]  
)  
[ON { 'groupe_fichiers' | DEFAULT} ]
```

```
< definition_et_contrainte_niveau_colonne > :  
nom_colonne type_donnée [(précision [, échelle] | max)]  
[COLLATE identifiant_du_jeu_de_caractères]  
[NULL | NOT NULL]  
[MASKED WITH (FUNCTION = 'nom_fonction_masque')]  
[IDENTITY [(valeur_de_départ, valeur_du_pas)]  
[INDEX nom_index [CLUSTERED | NONCLUSTERED] [ON {groupe_fichiers | DEFAULT}]]  
[CONSTRAINT nom_contrainte_valeur_par_defaut [DEFAULT valeur_par_défaut]]  
[  
    [CONSTRAINT nom_contrainte]  
    {  
        {PRIMARY KEY | UNIQUE}  
        [CLUSTERED | NONCLUSTERED] [ON {filegroup | DEFAULT}]  
        | FOREIGN KEY REFERENCES table_de_référence [(colonne_de_référence)]  
        [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]  
        [ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]  
        | CHECK (expression_logique)  
    } [, ... ]  
]
```



```

< contraintes_niveau_table > :

[
  [CONSTRAINT nom_contrainte]
  {
    {PRIMARY KEY | UNIQUE}
    [CLUSTERED | NONCLUSTERED] ({colonne1 [ASC | DESC]} [, ... ])
    [ON {filegroup | DEFAULT}]
    |
    [FOREIGN KEY] (colonne1 [, ... ])
    REFERENCES table_de_référence (colonne_de_référence1 [, ... ])
    [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
    [ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
    |
    CHECK (expression_logique)
  }[, ... ]
]

```

```

< index_niveau_table > :

[
  INDEX nom_index [CLUSTERED | NONCLUSTERED] ({colonne1 [ASC | DESC]} [, ... ])
  [ON {'groupe_fichiers' | DEFAULT}]
]

```

G.2.b) Le «UPDATE TABLE»

Exemple 77 : Modification d'une table.

```
ALTER TABLE nom_table
{
  [
    ALTER COLUMN nom_colonne
    /*modification d'une colonne*/
    {
      type_donnée [{précision [, échelle] | max}]
      [COLLATE identifiant_du_jeu_de_caractères]
      [NULL | NOT NULL]
    }
    |
    {ADD | DROP} MASKED [WITH (FUNCTION = 'nom_fonction_masque')]
  ]
  |
  [
    ADD
    /*ajout d'une ou de plusieurs colonnes*/
    nom_colonne type_donnée [{précision [, échelle] | max}]
    [COLLATE identifiant_du_jeu_de_caractères]
    [NULL | NOT NULL]
    [CONSTRAINT nom_contrainte_valeur_par_defaut [DEFAULT valeur_par_defaut]]
  ][, ... ] /*→ possibilité de rajouter d'autres colonnes*/
  |
  [
    [WITH {CHECK | NOCHECK}] ADD
    /*ajout d'une ou de plusieurs contraintes avec ou sans contrôle des données*/
    [CONSTRAINT nom_contrainte]
    {
      {PRIMARY KEY | UNIQUE}
      |
      FOREIGN KEY(colonne1 [, ... ]) REFERENCES table_réf [(colonne_réf1 [, ... ])]
      [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
      [ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
      |
      CHECK (expression_logique)
      |
      DEFAULT valeur_par_defaut FOR colonne
    }
  ][, ... ] /*→ possibilité de rajouter d'autres contraintes*/
  |
  [
    DROP
    /*suppression d'une ou de plusieurs contraintes et/ou colonnes*/
    [CONSTRAINT] {contrainte1} [, ... ]
    |
    COLUMN {colonne1} [, ... ]
  ][, ... ] /*→ possibilité de supprimer d'autres contraintes et/ou colonnes*/
  |
  /*activation ou désactivation de contraintes avec ou sans contrôle des données*/
  [[WITH {CHECK | NOCHECK}] {CHECK | NOCHECK} CONSTRAINT {ALL | contrainte1 [, ... ]}]
  |
  /*activation/désactivation de plusieurs ou de tous les triggers de la table*/
  [{ENABLE | DISABLE} TRIGGER {ALL | nom_trigger [, ... ]}]
}
```

G.2.c) Le «DROP TABLE»

Syntaxe :

```
DROP TABLE nom_table1 [, ... ]
```

Exemple 78 : Création et modification de plusieurs tables.

```
CREATE TABLE TCLIENTS2
(
  IdClient int NOT NULL IDENTITY(1,1),
  NomClient nvarchar(30) NOT NULL,
  PrenomClient nvarchar(30) NOT NULL,
  AdresseClient nvarchar(100) NOT NULL,
  CodePostalClient nvarchar(16) NOT NULL,
  VilleClient nvarchar(30) NOT NULL,
  CAC float NOT NULL,
  CONSTRAINT PK_TCLIENTS2 PRIMARY KEY CLUSTERED (IdClient ASC)
) ON 'PRIMARY'
GO
ALTER TABLE TCLIENTS2 ADD CONSTRAINT DF_TCLIENTS2_CAC DEFAULT ((0.0)) FOR CAC
GO
CREATE TABLE TPRODUITS2
(
  IdProduit int NOT NULL IDENTITY(1,1),
  DesignationProduit nvarchar(30) NOT NULL,
  PrixUnitaireProduit real NULL,
  TvaProduit real NOT NULL,
  PrixTTCProduit AS ((PrixUnitaireProduit*(1 + TvaProduit)))
  CONSTRAINT PK_TPRODUITS2 PRIMARY KEY CLUSTERED (IdProduit ASC)
) ON 'PRIMARY'
GO
CREATE TABLE TFACTURES2
(
  IdFacture int NOT NULL IDENTITY(1,1),
  IdClient int NULL,
  DateFacturation datetime NOT NULL CONSTRAINT DF_TFact2_DateFact DEFAULT (GETDATE()),
  CONSTRAINT PK_TFACTURES2 PRIMARY KEY CLUSTERED (IdFacture ASC)
) ON [PRIMARY]
GO
ALTER TABLE TFACTURES2 WITH CHECK ADD CONSTRAINT FK_TFACTURES2_TCLIENTS2
  FOREIGN KEY(IdClient) REFERENCES TCLIENTS2 (IdClient) ON DELETE CASCADE
GO
ALTER TABLE TFACTURES2 CHECK CONSTRAINT FK_TFACTURES2_TCLIENTS2
GO
CREATE TABLE TCOMMANDES2
(
  IdFacture int NOT NULL,
  IdProduit int NOT NULL,
  Qt_Cmd real NULL,
  CONSTRAINT PK_TCOMMANDES2 PRIMARY KEY CLUSTERED (IdFacture ASC, IdProduit ASC)
) ON [PRIMARY]
GO
ALTER TABLE TCOMMANDES2 WITH CHECK ADD CONSTRAINT FK_TCOMMANDES2_TFACTURES2
  FOREIGN KEY (IdFacture) REFERENCES TFACTURES2 (IdFacture) ON DELETE CASCADE
GO
ALTER TABLE TCOMMANDES2 CHECK CONSTRAINT FK_TCOMMANDES2_TFACTURES2
GO
ALTER TABLE TCOMMANDES2 WITH CHECK ADD CONSTRAINT FK_TCOMMANDES2_TPRODUITS2
  FOREIGN KEY(IdProduit) REFERENCES TPRODUITS2 (IdProduit) ON DELETE CASCADE
GO
ALTER TABLE TCOMMANDES2 CHECK CONSTRAINT FK_TCOMMANDES2_TPRODUITS2
```

G.3. Manipulations usuelles des vues

G.3.a) Le «CREATE VIEW» / «ALTER VIEW» / «DROP VIEW»

Une vue est une table virtuelle dont la structure et le contenu sont définis à travers une requête «select» à partir d'une ou plusieurs tables. Elle ne peut être créée que dans la base actuelle, et ses données ne sont chargées en mémoire qu'après son appel. Son rôle est multiple, elle permet :

- De synthétiser certaines données et traitements de façon compréhensible,
- D'Assurer l'évolution de la structure des tables, tout en restant compatible avec les applications clientes,
- De sécuriser l'accès aux tables physiques.

Une clause «select» - associée à la vue - doit respecter certaines règles :

- Elle ne peut contenir une clause «into»,
- Elle ne peut faire référence à des tables temporaires,
- Elle ne peut faire référence à des variables de type table,
- Elle ne peut contenir une clause «order by», à moins que cette dernière ne soit liée à un «select top(x)».

Par ailleurs, rappelons qu'il est tout à fait possible de modifier une table physique à travers une vue, mais cela n'est possible que sous des conditions :

- Si la vue porte sur plusieurs tables à travers des jointures, seule les données liées à une table peuvent faire l'objet d'une insertion, d'une modification ou d'une suppression.
- Les colonnes de la vue, obtenues à travers un calcul, à travers des fonctions d'agrégation ou à travers les clauses «union» «intersect», «except» et «crossjoin», ne peuvent faire l'objet d'une insertion, d'une modification ou d'une suppression,
- L'insertion, la modification ou encore la suppression, ne peut s'opérer sur une vue contenant les termes «distinct» ou «group by»,
- L'insertion, la modification ou encore la suppression, ne peut s'opérer sur une vue contenant à la fois les termes «top» et «with check option».

Syntaxe :

```
CREATE VIEW nom_vue [ (Colone1 , ... , Colonnen) ]  
AS instruction_select  
[ WITH CHECK OPTION ]
```

```
ALTER VIEW nom_vue [ (Colone1 , ... , Colonnen) ]  
AS instruction_select  
[ WITH CHECK OPTION ]
```

```
DROP VIEW nom_vue
```

Colone₁ : Lors de la création d'une vue, on a la possibilité de renommer les colonnes envoyées à travers la clause «select». La spécification des noms de colonnes et donc optionnelle. Par défaut, les noms des colonnes de la vue seront ceux renvoyés par le «select».

instruction_select : fait référence à requête sql valide.

with check option : Lors de la modification d'une table à travers la vue, cette options veille au respect des conditions définies dans la clause «select», et assure que les données modifiées soient toujours disponibles dans la vue après leur modification.

Exemple 79 : Création d'une vue.

```
CREATE VIEW view_client_ca
AS
SELECT
    IdClient,
    SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS CAC
FROM TFACTURES
JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
    IdClient
```

Exemple 80 : Modification d'une vue.

```
ALTER VIEW view_client_ca
AS
SELECT TOP(3)
    IdClient,
    AVG(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS CAMC
FROM TFACTURES
JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
GROUP BY
    IdClient
ORDER BY
    AVG(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) DESC
```

Exemple 81 : Suppression d'une vue après vérification de son existence.

```
IF DB_ID ('view_client_ca') IS NOT NULL
DROP VIEW view_client_ca
```

G.4. Manipulations usuelles des procédures stockées

G.4.a) Le «CREATE PROCEDURE» / «ALTER PROCEDURE» / «DROP PROCEDURE»

Une procédure permet de regrouper plusieurs traitements sous une seule et même entité, identifiable à travers un nom que l'on peut par la suite exploiter dans les autres traitements afin d'éviter de réécrire toutes les instructions de ce groupement.

Syntaxe :

```
CREATE PROCEDURE nom_procedure
[ { @Parami AS Typei } [= Valeur_par_defaut ] [ OUTPUT | READONLY ] ]
[ , ... ]
AS
{
[ BEGIN ]
    instructions_sql
    [ ... ]
[ END ]
}
```

```
ALTER PROCEDURE nom_procedure
[ { @Parami AS Typei } [= DEFAULT ] [ OUTPUT | READONLY ] ]
[ , ... ]
AS
{
[ BEGIN ]
    instructions_sql
    [ ... ]
[ END ]
}
```

```
DROP PROCEDURE nom_procedure
```

Exemple 82 : Création d'une procédure sans paramètres.

```
CREATE PROCEDURE P1
AS
SELECT
    TCLIENTS.*
FROM
    TCLIENTS
    LEFT JOIN TFACTURES ON TCLIENTS.IdClient = TFACTURES.IdClient
WHERE
    TFACTURES.IdClient IS NULL
GO
/*Appel de la procédure*/
EXECUTE P1
```

Exemple 83 : Création d'une procédure avec deux paramètres.

```
CREATE PROCEDURE P2
    @IdClient AS INT,
    @IdProduit AS INT
AS
SELECT
    IdClient,
    SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS CAP
FROM
    TFACTURES
    JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
    JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
WHERE
    IdClient = @IdClient AND
    TPRODUITS.IdProduit = @Produit
GROUP BY
    IdClient
```

```

GO
/*Possibilités d'appels de la procédure pour le client 1 et le produit 2*/
EXECUTE P2 1, 2
EXECUTE P2 @IdClient = 1, @IdProduit = 2
EXECUTE P2 @IdProduit= 2, @IdClient = 1

```

Exemple 84 : Création d'une procédure renvoyant deux jeux de résultats.

```

CREATE PROCEDURE P3
    @IdClient AS INT
AS
SELECT * FROM TCLIENTS WHERE IdClient = @IdClient
SELECT * FROM TFACTURES WHERE IdClient = @IdClient
GO
/*Appel de la procédure pour le client 1*/
EXECUTE P3 1

```

Exemple 85 : Création d'une procédure avec des valeurs par défauts pour les paramètres.

```

CREATE PROCEDURE P4
    @IdClient AS INT = 1,
    @VilleClient AS NVARCHAR(30) = 'CASA%'
AS
SELECT * FROM TCLIENTS
WHERE
    IdClient = @IdClient AND
    VilleClient LIKE @VilleClient + '%'
GO
/*Possibilités d'appels avec les différents paramètres*/
EXECUTE P4
EXECUTE P4 4
EXECUTE P4 2, 'Sa'

```

Exemple 86 : Création d'une procédure avec des paramètres scalaires en «OUTPUT».

```

CREATE PROCEDURE P5
    @IdClient AS INT = 1,
    @VilleClient AS NVARCHAR(30) OUTPUT
AS
SELECT @VilleClient = VilleClient FROM TCLIENTS
WHERE
    IdClient = @IdClient
GO
/*Possibilités d'appels avec les différents paramètres*/
DECLARE @VC AS NVARCHAR(30)
EXECUTE P5 3, @VC OUTPUT
PRINT @VC
EXECUTE P5 @VilleClient = @VC OUTPUT
PRINT @VC

```

Exemple 87 : Création d'une procédure avec un paramètres de type table.

```

/*Cela nécessite la création d'un nouveau type définit par le développeur*/
CREATE TYPE TAB AS TABLE (IdProd INT, DesProd NVARCHAR(30), Prix FLOAT)
GO
CREATE PROCEDURE P6
    @TAB AS TAB READONLY
AS
DECLARE @VarTab AS TAB
INSERT INTO @VarTab SELECT * FROM @TAB
SELECT * FROM @VarTab
GO

```

```

/*Possibilité d'appel*/
DECLARE @NewTab AS TAB
INSERT INTO @NewTab VALUES (1, 'Prod1', 100),
                             (2, 'Prod2', 200),
                             (3, 'Prod3', 300)

EXECUTE P6 @NewTab

```

Exemple 88 : Modification d'une procédure.

```

ALTER PROCEDURE P6
    @TAB AS TAB READONLY
AS
SELECT * FROM @TAB

```

Exemple 89 : Suppression d'une procédure.

```

DROP PROCEDURE P6

```

G.5. Manipulations usuelles des fonctions

G.5.a) Le «CREATE FUNCTION» / «ALTER FUNCTION » / «DROP FUNCTION»

Une fonction permet de regrouper plusieurs traitements sous une seule et même entité, identifiable à travers un nom unique que l'on peut par la suite exploiter dans les autres traitements afin d'éviter de réécrire toutes les instructions de ce groupement.

Il existe deux types de fonctions. Celles qui permettent de retourner une valeur scalaire, puis celles permettant de retourner une table ou une variable de type table.

Syntaxe du «create / alter» d'une fonction retournant un scalaire :

```

{CREATE | ALTER} FUNCTION nom_fonction
(
    [ { @Parami AS Typei } [= Valeur_par_defaut ] [ READONLY ] ]
    [ , ... ]
)
RETURNS Type_scalaire
AS
BEGIN
    instructions_sql
    [ ... ]
    RETURN Val_scalaire
END

```

Syntaxe du «create / alter» d'une fonction retournant une table :

```

{CREATE | ALTER} FUNCTION nom_fonction
(
    [ { @Parami AS Typei } [= Valeur_par_defaut ] [ READONLY ] ]
    [ , ... ]
)
RETURNS TABLE
AS
RETURN
[ ( ]
    instructions_select
[ ) ]

```


Syntaxe du «create / alter» d'une fonction retournant une variable de type table :

```
{CREATE | ALTER} FUNCTION nom_fonction
(
  [ { @Parami AS Typei } [= Valeur_par_defaut ] [ READONLY ] ]
  [ , ... ]
)
RETURNS @TAB TABLE <Definition_de_la_table>
AS
BEGIN
    instructions_sql
    [ ... ]
    RETURN
END
```

Syntaxe du drop :

```
DROP FUNCTION nom_fonction
```

Exemple 90 : Création d'une fonction retournant un scalaire.

```
/*Création d'une fonction retournant le chiffre d'affaire par client et par produit*/
CREATE FUNCTION fct_scal_ca_par_client_produit
(
  @IdClient          AS INT,
  @DesignationProduit AS NVARCHAR(30)
)
RETURNS FLOAT
AS
BEGIN
    DECLARE @CAP AS FLOAT

    SELECT
        @CAP = SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit))
    FROM
        TFACTURES
        JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
        JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
    WHERE
        IdClient = @IdClient AND
        DesignationProduit = @DesignationProduit

    RETURN @CAP
END
GO
/*Possibilités d'appel*/
SELECT REP = dbo.fct_scal_ca_par_client_produit (1, 'PA')
DECLARE @RES AS FLOAT
SET @RES = dbo.fct_scal_ca_par_client_produit (1, 'PA')
SELECT REP = @RES
```

Exemple 91 : Création d'une fonction retournant une table.

```
/*Création d'une fonction retournant le chiffre d'affaire par client et par produit*/
CREATE FUNCTION fct_tab_ca_par_client_produit_v1
(
  @IdClient          AS INT
)
RETURNS TABLE
AS
RETURN
```

```

(
    SELECT
        IdClient,
        DesignationProduit,
        SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS CAP
    FROM
        TFACTURES
        JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
        JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
    WHERE
        IdClient = @IdClient
    GROUP BY
        IdClient,
        DesignationProduit
)
GO
/*Possibilités d'appel*/
SELECT * FROM dbo.fct_tab_ca_par_client_produit_v1 (1)
DECLARE @RES AS TABLE (IdClient INT, DesignationProduit NVARCHAR(30), CAP FLOAT)
INSERT INTO @RES SELECT * FROM dbo.fct_tab_ca_par_client_produit_v1 (2)
SELECT * FROM @RES

```

Exemple 92 : Création d'une fonction retournant une variable de type table.

```

/*Création d'une fonction retournant le chiffre d'affaire par client et par produit*/
CREATE FUNCTION fct_tab_ca_par_client_produit_v2
(
    @IdClient AS INT
)
RETURNS @RES AS TABLE (IdClient INT, DesignationProduit NVARCHAR(30), CAP FLOAT)
AS
BEGIN
    INSERT INTO @RES
    (
        SELECT
            @IdClient,
            DesignationProduit,
            SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS CAP
        FROM
            TFACTURES
            JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
            JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
        WHERE
            IdClient = @IdClient
        GROUP BY
            DesignationProduit
    )
    RETURN
END
GO
/*Possibilités d'appel*/
SELECT * FROM dbo.fct_tab_ca_par_client_produit_v2 (1)
DECLARE @RES AS TABLE (IdClient INT, DesignationProduit NVARCHAR(30), CAP FLOAT)
INSERT INTO @RES SELECT * FROM dbo.fct_tab_ca_par_client_produit_v2 (2)
SELECT * FROM @RES

```

Exemple 93 : Suppression d'une fonction.

```

DROP FUNCTION fct_scal_ca_par_client_produit

```

G.6. Manipulations usuelles des triggers

G.6.a) Le «CREATE TRIGGER» / «ALTER TRIGGER » / «DROP TRIGGER»

Un «Trigger» permet de regrouper des traitements devant être exécutés automatiquement suite à un événement sur un objet de la base de données. Il existe trois catégories de «triggers» :

- Les «triggers» DML liés aux actions «insert, update, delete» sur des tables ou des vues,
- Les «triggers» DDL liés aux actions «create, alter, drop, grant, deny, revoke» sur des bases de données ou sur le serveur. Il est important de noter que certaines procédures systèmes lance à travers leurs traitements des triggers DDL définis par l'utilisateur.
- Les «triggers» de connexion liés à l'action «logon» des utilisateurs sur la base de données.

Syntaxe usuelle d'un trigger DML : action «create / alter»

```
{CREATE | ALTER} TRIGGER nom_trigger
ON { nom_table | nom_view }
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
instructions_sql
```

Syntaxe usuelle d'un trigger DML : action «drop»

```
DROP TRIGGER nom_trigger
```

Syntaxe usuelle d'un trigger DDL : action «create / alter»

```
{CREATE | ALTER} TRIGGER nom_trigger
ON { ALL SERVER | DATABASE }
{ FOR | AFTER }
{ [ TYPE_EVENNEMENT ] | [ GROUPE_EVENNEMENTS ] } [ , ... ]
AS
instructions_sql
```

Syntaxe usuelle d'un trigger DDL : action «drop»

```
DROP TRIGGER nom_trigger
ON { ALL SERVER | DATABASE }
```

Syntaxe usuelle d'un trigger LOGON : action «create / alter»

```
{CREATE | ALTER} TRIGGER nom_trigger
ON ALL SERVER
{ FOR | AFTER } LOGON
AS
instructions_sql
```

Syntaxe usuelle d'un trigger LOGON : action «drop»

```
DROP TRIGGER nom_trigger
ON ALL SERVER
```

G.6.b) Le «ENABLE TRIGGER» / «DISABLE TRIGGER »

Syntaxe usuelle d'un trigger LOGON : action «anable / disable»

```
{ ENABLE | DISABLE } TRIGGER { nom_trigger1 [, ... ] | ALL }  
ON { nom_table_ou_vue | DATABASE | ALL SERVER }
```

Exemple 94 : Création d'un trigger DML.

```
CREATE TRIGGER tr_commandes_insert  
ON TCOMMANDES  
AFTER INSERT  
AS  
BEGIN  
    SET NOCOUNT ON  
    UPDATE  
        TCLIENTS  
    SET  
        CAC += ( SELECT PrixUnitaireProduit*Qt_Cmd*(1+TvaProduit)  
                FROM   INSERTED  
                JOIN   TPRODUITS ON INSERTED.IdProduit = TPRODUITS.IdProduit)  
    WHERE  
        IdClient = ( SELECT IdClient  
                    FROM   INSERTED  
                    JOIN   TFACTURES ON INSERTED.IdFacture = TFACTURES.IdFacture )  
END
```

H. Les curseurs

Un «curseur» est une variable un peu spéciale. Son contenu est constitué par le flux de données renvoyé par une requête de sélection. La différence majeure entre une variable de type «table» et un «curseur» est que ce dernier dispose d'un mécanisme lui permettant de parcourir ce flux - entre autre, ligne par ligne - tout en ayant la possibilité d'extraire les valeurs des colonnes pour chacune de ces lignes. Le parcours des curseurs se fait via l'instruction «FETCH». Cette dernière offre plusieurs possibilités de positionnement.

Contrairement aux variables de type «table», il est tout à fait possible de modifier les données d'une table à travers un «curseur». Toutefois, il n'est pas conseillé d'user de cette possibilité, car en général, l'utilisation des curseurs nécessite des ressources considérables.

Les «curseurs» disposent de plusieurs options, certaines sont même incompatibles entre elles. En effet, toutes les valeurs définies entre les crochets ouvrants et fermants sont optionnelles. Cependant, les options définies dans un même ensemble, et séparées par des barres obliques, s'excluent entre elles. A titre d'exemple, une variable de type «curseur» ne peut pas être à la fois locale et globale, ou alors «forward_only» et «scroll» ou encore «static» et «dynamique».

Syntaxe usuelle pour déclarer et manipuler une variable de type curseur :

```
DECLARE nom_curseur CURSOR                                /*déclare le curseur*/
[ LOCAL | GLOBAL ]                                       /*déclare le curseur*/
[ FORWARD_ONLY | SCROLL ]                               /*défini le type de parcours*/
[ STATIC | DYNAMIC | KEYSET | FAST_FORWARD ]            /*défini la classe mémoire*/
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ] /*défini le type de verrouillage*/
FOR instruction_select /*extraction des données pour remplir le curseur*/
[ FOR UPDATE [ OF colonne1 [, ... ] ] ] /*pour autoriser la modification via le*/
                                           /*curseur*/

OPEN nom_curseur      /*ouvre le curseur*/
CLOSE nom_curseur     /*ferme le curseur*/
DEALLOCATE nom_curseur /*libère les ressources occupées par le curseur*/
```

Syntaxe usuelle pour parcourir une variable de type curseur :

```
FETCH
[
  [ FIRST | NEXT | PRIOR | LAST | ABSOLUTE{ p | @var } | RELATIVE{ p | @var } ]
  FROM
]
{ { [ GLOBAL ] nom_curseur } | @nom_curseur }
[ INTO @var1 [, ... ] ]
```

FETCH FIRST FROM nom_curseur : se positionner sur la première ligne du curseur,

FETCH NEXT FROM nom_curseur : se positionner sur la ligne suivante,

FETCH PRIOR FROM nom_curseur : se positionner sur la ligne précédente,

FETCH LAST FROM nom_curseur : se positionner sur la dernière ligne,

FETCH ABSOLUTE N FROM nom_curseur : se positionner sur la N^{ième} ligne,

FETCH RELATIVE N FROM nom_curseur : se positionner sur la N^{ième} ligne plus loin que la ligne actuelle.

LOCAL

Permet de déclarer un curseur comme étant une variable locale à un traitement de type procédure, de type déclencheur, ou encore comme étant un paramètre OUTPUT d'une procédure stockée. Les ressources occupées sont désalloué automatiquement à la fin du traitement de la procédure stockée ou du déclencheur. Dans le cas d'un paramètre OUTPUT, le curseur est désalloué automatiquement à la fin du dernier traitement lui faisant référence.

GLOBAL

Permet de déclarer un curseur comme étant une variable globale à une connexion. Tous les traitements SQL peuvent utiliser ce curseur. Il ne peut être désalloué que de façon explicite durant la connexion, ou alors de façon implicite à la déconnexion.

FORWARD_ONLY

Avec cette option, le parcours du curseur ne peut se faire que du début vers la fin, ligne par ligne via l'instruction «FECTH NEXT» (aller à la ligne suivante).

«FORWARD_ONLY» est en opposition avec «SCROLL». En effet, ce dernier permet de parcourir le curseur avec toutes options possibles de l'instruction «FECTH».

En l'absence des options «STATIC», «DYNAMIC» et «KEYSET» - qui sont par défaut de type «SCROLL» -, le curseur sera par défaut défini comme étant «DYNAMIC». Durant la phase des traitements effectués par le curseur dans ce cas de figure, toutes les modifications survenues sur les tables adjacentes seront

reportées sur ce curseur. Toutefois, vu que le parcours ne peut se faire que vers l'avant, seules les lignes non encore traitées permettront de se rendre compte de cet aspect.

En l'absence à la fois des termes «FORWARD_ONLY», «SCROLL», «STATIC», «DYNAMIC» et «KEYSET», le curseur sera par défaut défini comme étant «FORWARD_ONLY» et «DYNAMIC».

SCROLL

Avec cette option, le parcours du curseur peut se faire avec toutes les options possibles de l'instruction «FECH» (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Elle est donc en opposition avec l'option «FORWARD_ONLY».

En l'absence de cette option, le curseur sera considéré - par défaut - comme étant «FORWARD_ONLY», à moins que l'une des options «STATIC», «DYNAMIC» ou «KEYSET» ne soit utilisées.

STATIC

Un curseur «STATIC» est par défaut de type «SCROLL», ce qui signifie qu'on peut utiliser toutes les options de l'instruction «FECH» » (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Cependant, si des modifications ont eu lieu sur les tables adjacentes, celles-ci ne seront pas retranscrites sur le curseur.

DYNAMIC

Un curseur «DYNAMIC» est par défaut de type «SCROLL», ce qui signifie qu'on peut utiliser toutes les options de l'instruction «FECH» » (FIRST, LAST, PRIOR, NEXT, RELATIVE) sauf (ABSOLUTE). A l'opposé du curseur «STATIC», si des modification ont eu lieu sur les tables adjacentes, celles-ci seront retranscrites sur le curseur.

KEYSET

Cette option - introduite à titre informatif - regroupe certains aspects liés à l'option «STATIC» et d'autres liés à l'option «DYNAMIC». Elle est très difficile à mettre en œuvre car la modification de la table adjacente dépend de plusieurs circonstances.

FAST_FORWARD

Cette option regroupe à la fois les options «FORWARD_ONLY» et «READ_ONLY». Ce qui signifie que le parcours ne peut se faire que ligne par ligne du début vers la fin, et qu'en plus, la modification à travers le curseur n'est pas autorisée. Par conséquent, cette option est incompatible avec les options «SCROLL» et «FOR_UPDATE».

READ_ONLY

Cette option interdit les mises à jour des données des tables adjacentes à travers le curseur.

SCROLL_LOCKS

Cette option permet de garantir les mises à jour à travers le curseur en verrouillant les lignes des tables adjacentes au moment de son ouverture. Elle est incompatible avec les options «FAST_FORWARD» et «SCROLL».

OPTIMISTIC

Cette option permet d'effectuer des mises à jour conditionnées à travers le curseur, et ce, sans verrouiller les lignes correspondantes dans la table adjacente. En effet, au moment de la demande de modification positionnée (clause WHERE CURRENT OF), SQL Server vérifie si la ligne dans la table adjacente a été modifiée depuis l'ouverture du curseur. Si tel est le cas, cette demande est rejetée, sinon la modification est acceptée. Cette option est incompatible avec l'option «FAST_FORWARD».

Exemple 95 : Création d'un curseur dans une procédure stockée.

```
CREATE PROCEDURE ps_cac_update
AS
BEGIN
    DECLARE
        @IdClient AS INT,
        @CAC AS FLOAT

    DECLARE CursCAC CURSOR LOCAL FAST_FORWARD
    FOR
        SELECT
            IdClient,
            SUM(PrixUnitaireProduit*Qt_Cmd*(1 + TvaProduit)) AS CAC
        FROM
            TFACTURES
            JOIN TCOMMANDES ON TFACTURES.IdFacture = TCOMMANDES.IdFacture
            JOIN TPRODUITS ON TCOMMANDES.IdProduit = TPRODUITS.IdProduit
        GROUP BY IdClient
    OPEN CursCAC
    FETCH NEXT FROM CursCAC INTO @IdClient, @CAC
    WHILE(@@FETCH_STATUS = 0)
    BEGIN
        UPDATE
            TCLIENTS
        SET
            CAC = @CAC
        WHERE
            IdClient = @IdClient
        FETCH NEXT FROM CursCAC INTO @IdClient, @CAC
    END
    CLOSE CursCAC
    DEALLOCATE CursCAC
END
```

Exemple 96 : modifier le type par défaut des curseurs (Local/Global) pour la base «boutique»

```
SELECT DATABASEPROPERTYEX('BOUTIQUE', 'IsLocalCursorsDefault') AS ISLOCAL
ALTER DATABASE BOUTIQUE SET CURSOR_DEFAULT LOCAL
SELECT DATABASEPROPERTYEX('BOUTIQUE', 'IsLocalCursorsDefault') AS ISLOCAL
ALTER DATABASE BOUTIQUE SET CURSOR_DEFAULT GLOBAL
SELECT DATABASEPROPERTYEX('BOUTIQUE', 'IsLocalCursorsDefault') AS ISLOCAL
```

I. Conception des Bases de Données avec Merise

Dans cette section nous allons nous intéresser à la modélisation conceptuelle des bases de données relationnelles avec «Merise». Cette méthode s'appuie sur le schéma entité-association à travers l'étude des dépendances fonctionnelles. Nous allons nous focaliser sur le modèle conceptuel des données «MCD», le modèle logique des données «MLD» et le modèle physique des données «MPD».

I.1. Le modèle conceptuel des données «MCD»

I.1.a) Les entités et les identifiants

Une entité est un regroupement d'informations (attributs) homogènes caractérisant un objet. A titre d'exemple, le nom d'un client, son prénom, son adresse, son code postal ou encore sa ville, sont des attributs qui caractérisent le client. De même, la désignation du produit, le prix du produit ou encore la tva du produit, sont des attributs qui caractérisent le produit. Grouper par exemple la tva d'un produit dans le même lot que le nom du client n'aura aucun sens. Par ailleurs, chaque ligne de l'entité doit être identifiable de façon unique. Si aucun attribut ne possède cette caractéristique, un identifiant numérique doit être rajouté pour jouer ce rôle. En effet, une entité doit contenir au moins un attribut (son identifiant). Car, si ce n'est pas le cas, cet attribut ne doit pas posséder des doublons, au risque d'instaurer une ambiguïté (imaginons le cas où plusieurs clients portent le même nom dans l'entité client et que le nom du client est le seul attribut de cette entité !!!, ou encore dans l'entité produit avec la désignation produit comme seul attribut !!!). On schématise les entités par des rectangles à coins carrés.

CLIENTS	PRODUITS
<u>IdClient</u>	<u>IdProduit</u>
NomClient	DésignationProduit
PrenomClient	PrixUnitaireProduit
AdresseClient	TvaProduit
CodePostalClient	QuantiteStock
VilleClient	QuantiteSeuil

I.1.b) Les associations et les cardinalités

Une association représente le lien sémantique qui existe entre deux ou plusieurs entités. A titre d'exemple, le lien naturel qui relie un client à un produit est l'action de commander. En effet, un client peut commander des produits, et un produit peut être commandé par des clients.

Le nombre minimum et maximum de fois que l'action est autorisée sont appelés respectivement cardinalité minimale et cardinalité maximale. Dans notre exemple on peut les déduire en se posant les questions suivantes :

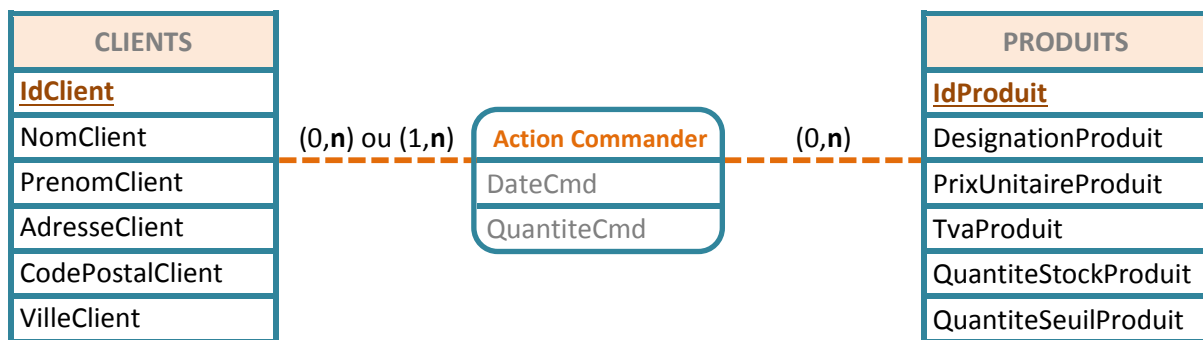
Pour connaître la cardinalité minimale des clients : combien de produits au minimum un client peut-il commander. La réponse est 1, si on considère qu'un client ne peut être inséré dans la table des clients que s'il est liée à un produit, ou alors 0 si on considère qu'un client est tout d'abord inséré dans la table des clients avant d'être associé à un produit. Cette dernière option est la plus naturelle et la plus facile à mettre en œuvre.

Pour connaître la cardinalité maximale des clients : combien de produits au maximum un client peut-il commander. La réponse est plusieurs.

Dans ce cas de figure on parle de cardinalités «1 , n» ou «0 , n» côté clients. On peut procéder de la même manière côté produit, pour déduire que les cardinalités sont «0 , n». En effet, un produit peut être commandé par plusieurs clients, mais d'autres produits en revanche - moins populaires - risquent de ne faire l'objet d'aucune action d'achat. Par ailleurs, au moment de l'introduction des produits dans l'entité produits, ces derniers ne sont pas encore liés à des clients.

Une association peut être porteuse ou dépourvue d'informations. Dans notre exemple, pour un client donné, on a besoin de connaître à quelle date la commande a été passée, ainsi que la quantité commandée de chaque produit. Dans d'autres cas, elle est définie uniquement pour symboliser le lien sémantique entre les entités.

On schématise les associations par des rectangles à coins arrondis. Notons au passage que la modélisation proposée ci-dessous n'est pas finalisée. Elle présente encore un certain nombre d'inconvénients, qu'on corrigera dans les sections suivantes.

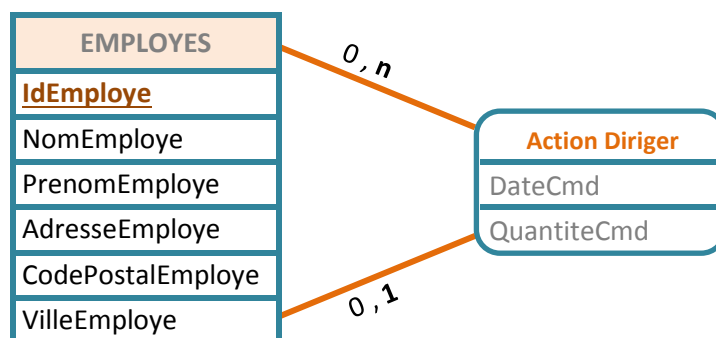


(1). Les associations binaires

Une association est dite binaire si elle lie deux entités distinctes, comme c'est le cas pour l'association «Action Commander» qui lie les entités clients et produits.

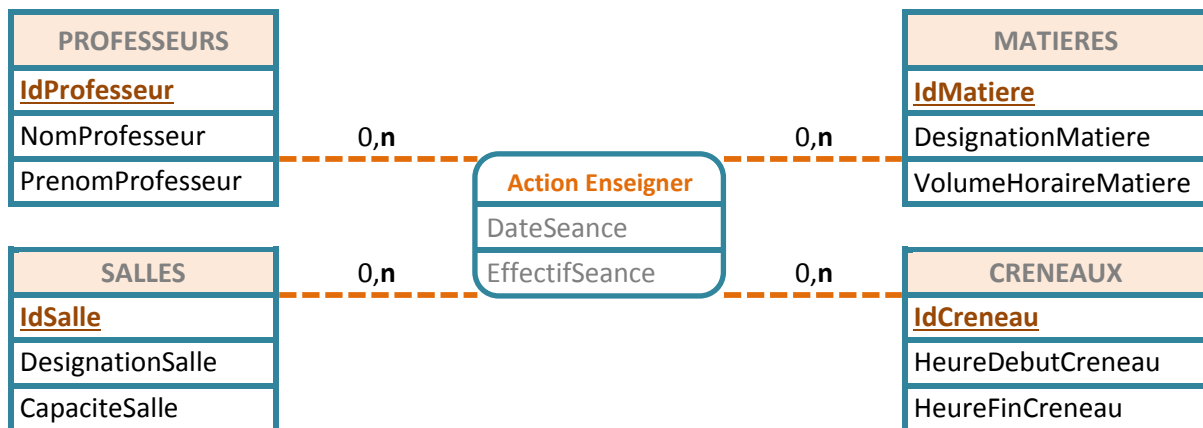
(2). Les associations réflexives

Une association réflexive est en quelque sorte une association binaire représentant le lien sémantique existant entre les éléments d'une même entité. A titre d'exemple, dans une entreprise on trouve généralement un président directeurs général, plusieurs directeurs, plusieurs chefs d'équipes, puis d'autres employés. Tous ces gens-là sont - en général - des employés de l'entreprise, et possèdent les mêmes attributs. Pour définir le lien hiérarchique qui existe entre eux, on peut définir une association «dirige» ou «être dirigé». Dans ce schéma, tous les employés - en dehors du pdg - ont un seul supérieur hiérarchique (le pdg n'a aucun supérieur hiérarchique), et toute personne avec des responsabilités peut avoir - ou pas - plusieurs subordonnés.



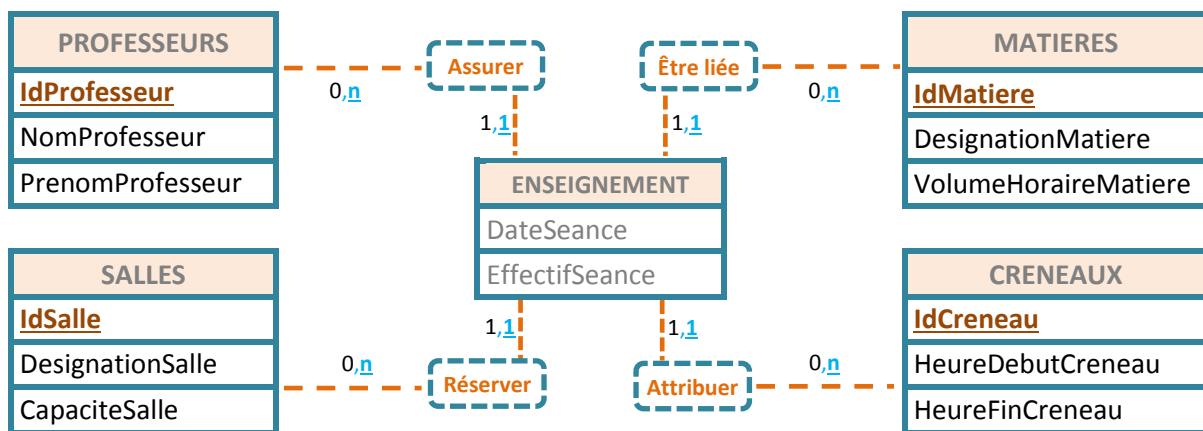
(3). Les associations n-aires

Une association non binaire est une association qui lie au minimum trois entités entre elles. L'exemple classique est celui où l'action enseigner lie les entités professeurs, matières, classes et créneaux.



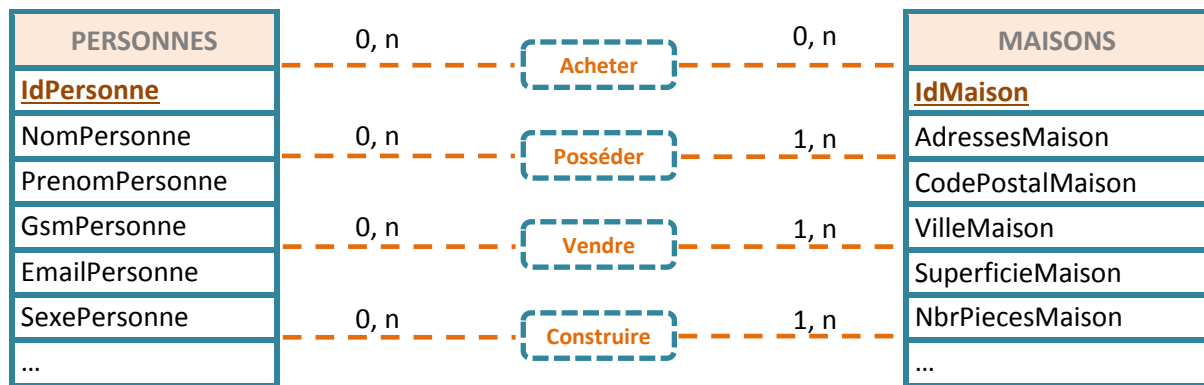
Il peut arriver des fois, de considérer - à tort - une association comme étant une entité, mais lorsqu'on établit les cardinalités, on s'aperçoit que les cardinalités maximales du côté de cette entité sont toutes à 1, alors que les cardinalités maximales du côté des autres entités sont à n. Dans ce cas de figure, l'entité mise en cause ainsi que toutes les associations l'entourant et vérifiant cette règle, vont fusionner pour former une seule association liant les autres entités intervenantes dans cette configuration.

Dans notre situation, cela revient à remplacer le scénario ci-dessous par celui établi juste avant.



(4). Les associations plurielles

Il peut arriver que deux entités soient liées à travers plusieurs associations, on parle alors d'associations plurielles. C'est le cas par exemple entre une entité personne et une entité maison. En effet, une ou plusieurs personnes peuvent acheter/posséder/vendre/construire une ou plusieurs maisons. Et inversement, une maison peut être achetée/possédée/vendue/construite par une ou plusieurs personnes. Les actions acheter, posséder, vendre et construire symbolisent chacune une association. Les cardinalités côté personne sont de (0, n) pour l'ensemble des actions, alors que côté maison, elles sont de (0, n) pour l'action achetée, et (1, n) pour les actions possédée/vendue/construite.



I.1.c) Les règles de normalisation

La conception d'une base de données nécessite la prise en compte d'un certain nombre de bonnes pratiques, permettant - entre autres - d'éviter la redondance et d'établir la cohérence entre les données. Voici donc la liste des règles usuelles :

- La normalisation des noms,
- La normalisation des attributs,
- La normalisation des identifiants,
- La normalisation des entités,
- La normalisation des cardinalités,
- La normalisation des associations.

La normalisation des noms :

Les noms utilisés pour identifier les entités, les associations ou encore les attributs de ces deux dernières doivent être uniques et compréhensibles. Pour les entités, il faut choisir un nom commun au pluriel et en majuscule (PERSONNES, MAISONS, CLIENTS, PRODUITS). Pour les associations, il faut choisir un verbe à l'infinitif (construire, vendre, posséder, acheter), ou à la forme passive (être construit, être vendu, être possédé, être acheté) ou encore accompagné d'un adverbe (avoir lieu durant, avoir lieu pendant, avoir lieu dans, avoir lieu à). Pour les attributs des entités et des associations, il faut choisir un nom commun au singulier (NomClient, PrenomClient, CodePostalClient).

La normalisation des attributs :

Les attributs des entités et des associations ne doivent pas être calculables à partir d'autres attributs, et ne doivent pas être redondants. A titre d'exemple, si un client possède plusieurs adresses, plusieurs téléphones, ou encore plusieurs contacts, il faut créer une association supplémentaire de cardinalité maximale n pour chacun de ces attributs. Cette redondance ne se limite pas uniquement à une entité ou à une association, mais concerne tout le modèle. Pour illustrer cette situation, considérons le cas où les clients possèdent deux adresses, une dans l'entité clients et l'autre dans l'association commander, ces deux adresses ne doivent pas avoir le même rôle, sinon cela risque de conduire à des confusions. Dans ce cas de figure, il est préférable de placer ces adresses (adresse de facturation et adresse de livraison) en fonction de leurs dépendances de la livraison ou de la facturation. En d'autres termes, si pour chaque client les deux adresses sont les mêmes, cela s'appelle de la redondance. Maintenant, si ces deux adresses sont généralement différentes et ne changent pas pour l'ensemble des livraisons, elles doivent être définies dans l'entité «clients». Mais si au

contraire, l'une de ces adresses varie en fonction des livraisons, cela va de soi, il faut la définir dans l'association «commander».

La normalisation des identifiants :

Chaque entité doit posséder un identifiant. Il faut éviter les identifiants composés de plusieurs attributs ou sous forme de chaînes de caractères, car cela nuit aux performances et risque de poser des problèmes dans le futur. En effet, une clé composée du nom, du prénom et de la date de naissance est loin d'être efficace, car lors des jointures (comparaison de la clé primaire avec la clé étrangère), comparer des chaînes de caractères est plus gourmand en termes de temps, d'autant plus qu'il y a trois comparaisons à faire. Par ailleurs, rien n'empêche que deux personnes qui naissent à la même date ne puissent pas avoir le même nom et le même prénom. Il faut garder à l'esprit qu'il est extrêmement rare de gagner au loto, mais ceci n'empêche pas qu'il est souvent des gagnants. Par ailleurs, il faut éviter les clés primaires susceptibles d'évoluer dans le temps comme les numéros de téléphone ou les numéros d'immatriculations. Pour faire simple et efficace, il faut choisir un entier - de préférence - auto-incrémenté.

La normalisation des entités :

Toute entité entourée par des associations avec des cardinalités maximales à 1 du côté de l'entité, et n de l'autre côté de chaque association, doit être remplacée par une association regroupant l'entité ainsi que l'ensemble des associations participantes à cette configuration (voir exemple des association plurielles).

La normalisation des cardinalités :

Une cardinalité minimale ne peut prendre que la valeur 0 ou 1. En effet, les seules valeurs significatives pour la cardinalité minimale sont 0 et 1 (0 : un client peut exister sans qu'il soit liée à un produit), (1 : un client ne peut exister que s'il est lié à un produit). Dans la mesure où cette cardinalité dépasse la valeur 1 (exemple de l'association entre l'entité joueurs et l'entité matchs : pour jouer un match de football, il faut au minimum 11 joueurs), il suffit de rajouter une entité équipe pour ramener cette cardinalité à la norme.

Une cardinalité maximale ne peut prendre que la valeur 1 ou n. Le terme n désigne toute valeur possible supérieure strictement à 1. Une cardinalité maximale valant 0, n'a aucun sens. Elle est synonyme d'une mauvaise conception. Les seules valeurs significatives pour la cardinalité maximale sont 1 ou n (1 : un enfant ne peut avoir au maximum qu'une mère génétique), (n : un produit peut être acheté par plusieurs clients).

Lors du passage à un schéma relationnel, pour les cardinalités (0, n) et (1, n), on ne fera pas de différence entre les cardinalités minimales 0 et 1. La cardinalité minimale valant 1 peut être vérifiée dans le modèle physique à travers un déclencheur.

La normalisation des associations :

Les attributs d'une association doivent dépendre directement des identifiants de toutes les entités qui l'entoure. En d'autres termes, les identifiants des entités en association doivent définir de façon unique chaque attribut de l'association. A titre d'exemple, dans l'association commander entre les entités clients et produits, les attributs «DateCmd» et «QuantiteCmd» ne respectent pas justement cette règle. En effet, le couple de valeurs («IdClient», «IdProduit») ne permet pas d'identifier de façon unique chacun de ces attributs. Il est de même pour l'association enseigner, puisque la connaissance de (IdEnseignant, IdMatiere, IdSalle, IdCreneau) ne permet pas d'identifier de façon unique «DateSeance» ou «EffectifSeance».

Lorsque les cardinalités maximales d’une association valent toutes n, et que les identifiants des entités qui l’entoure permettent d’identifier de façon unique chacun de ses attributs, l’association doit être transformée en entité avec un identifiant composé de l’ensemble des identifiants des entités en association.

Lorsque l’une des cardinalités maximales d’une association vaut 1, ses attributs doivent migrer vers l’entité présentant cette cardinalité.

Lorsque toutes les cardinalités maximales d’une association valent 1, l’association et les entités qui l’entoure doivent fusionner pour former une seule entité.

I.1.d) Les formes normales

Pour avoir un bon schéma relationnel, il est important d’adopter d’autres règles normalisation aussi importantes que les six règles étudiées dans la section précédente. Cela permettra en effet de consolider la robustesse de la conception du modèle, en évitant la redondance des données ainsi que les problèmes de mise à jour et de cohérence qui en découlent.

La première forme normale :

Cette forme concerne les attributs multivalués. Tout attribut doit être en effet atomique. Le terme atomique sous-entend que si on arrive à décomposer l’attribut en plusieurs informations, ce dernier n’aura plus de sens, ou alors, ces informations ne pourront pas être exploitables individuellement.

AdresseClient	EmailClient
20, Rue des roses, 20000, Casablanca, Maroc	r.c@ens-sup.com, r.c@hotmail.fr, r.c@gmail.com

Nous constatons ici que l’attribut «AdresseClient» est composée du numéro dans la rue, du nom de la rue, du code postal, de la ville et du pays, et que l’attribut «EmailClient» est composé de plusieurs emails séparés par des virgules. Si dans nos traitements actuels et futurs, on n’a nullement besoin de faire par exemple des recherches par nom de rue, par code postal, par ville, par pays ou par email, alors les attributs «AdresseClient» et «EmailClient» peuvent garder leurs formes pour des raisons d’optimisation. Mais si en revanche, cette décomposition permet ou permettra dans le futur de faire des traitements spécifiques à ces éléments, ces attributs doivent être décomposés en plusieurs entités distinctes pour «AdresseClient» et en une entité appart pour «EmailClient».

La deuxième forme normale :

Cette forme ne concerne que les identifiants composés. Un identifiant peut en effet être composé de plusieurs attributs, mais les autres attributs de l’entité doivent dépendre de la totalité de cet identifiant et non d’une partie uniquement.

<u>TitreFilm</u>	<u>DateProjection</u>	<u>HeureProjection</u>	<u>SalleProjection</u>	DureeFilm
Le roi lion	14/06/2016	20	3	75

Dans cet exemple, on a considéré un identifiant composé de (TitreFilm, DateProjection, Heureprojection, SalleProjection). On constate que la durée du film ne dépend que du titre du film et non de la totalité de l’identifiant. Cependant, pour des raisons de performances, on avait expliqué auparavant qu’il fallait éviter les identifiants composés, et qu’il fallait les remplacer par des identifiants entiers auto-incrémentés. Par conséquent, la deuxième forme normale doit être respectée si elle n’impacte pas négativement sur le temps des traitements. Le risque majeur du non-respect de cette forme est la possibilité de créer des incohérences.

En effet, si on projette le même film à une autre date par exemple, on risque de saisir une durée complètement différente.

La troisième forme normale :

Si on divise les attributs d'une entité en deux blocs, un bloc regroupant les attributs de l'identifiant, et un autre bloc regroupant le reste des attributs, aucune dépendance ne doit exister entre les attributs de ce dernier bloc. Autrement dit, tout attribut n'appartenant pas au bloc identifiant, ne doit dépendre que du bloc l'identifiant.

<u>DateProjection</u>	<u>HeureProjection</u>	<u>SalleProjection</u>	TitreFilm	DureeFilm
14/06/2016	20	3	Le roi lion	75

Dans cet exemple, on a considéré un identifiant composé de (DateProjection, Heureprojection, SalleProjection). On constate que la durée du film dépend du titre du film.

En résumé :

Une relation est normalisée en première forme normale si :

- 1) elle possède un identifiant unique et stable,
- 2) chaque attribut est monovalué (ne peut avoir qu'une seule valeur),
- 3) aucun attribut n'est décomposable en plusieurs attributs significatifs.

Une relation est normalisée en deuxième forme normale si et seulement si :

- 1) elle est en première forme normale,
- 2) tout attribut non identifiant est totalement dépendant de l'identifiant (aucun des attributs ne dépend que d'une partie de l'identifiant),
- 3) La deuxième forme normale ne concerne que les relations ayant un identifiant composé. Une relation en première forme normale n'ayant que l'identifiant comme attribut est considérée comme une relation en deuxième forme normale.

Une relation est normalisée en troisième forme normale si :

- 1) elle est en deuxième forme normale,
- 2) tout attribut doit dépendre directement de l'identifiant et uniquement de celui-ci (aucun attribut ne doit dépendre de l'identifiant par transitivité, ou dépendre d'un autre attribut non identifiant).

Remarque :

Il existe des règles de normalisation supplémentaires, mais on considère que le respect des celles présentées dans ce document est largement suffisant pour concevoir des modèles de bases de données relationnelles fiables. Par ailleurs, pour des raisons d'optimisation, on sera souvent amenés à faire des dérogations justifiées à certaines règles. C'est ce qu'on appelle la dénormalisation.

I.2. Les règles de passage du model conceptuel au modèle logique des données «MLD»

Pour passer du modèle conceptuel au modèle logique des données, on doit appliquer un certain nombre de règles :

Règle 1 :

Les entités deviennent des relations (tables) et les attributs des colonnes.

Règle 2 :

Les identifiants des entités deviennent des clés primaires des relations (tables).

Règle 3 :

Les associations doivent soit disparaître, soit être transformées en relations (tables). Nous allons illustrer ci-dessous les différents cas de figures :

- a) Une association binaire ou n-aire dont les cardinalités maximales sont toutes à 1, doit généralement fusionner avec les entités en association pour former une seule table. Cependant, certaines exigences fonctionnelles peuvent nous obliger à considérer d'autres possibilités. Nous allons énumérer ci-dessous les différentes possibilités :

- Le cas classique : faire migrer les attributs de l'association ainsi que ceux des entités vers l'une des entités la plus significative fonctionnellement. L'association et les entités qui ont été vidées de leurs attributs doivent être supprimées.

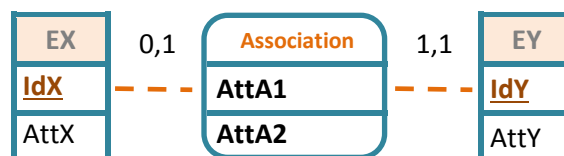


Nous devons obtenir à ce moment l'une des relations suivantes :

EX(IdX, AttX, AttY, AttA1, AttA2) si l'entité EX est plus significative que l'entité EY

EY(IdY, AttY, AttX, AttA1, AttA2) si l'entité EY est plus significative que l'entité EX

- Le cas où fonctionnellement on souhaite distinguer entre les entités en association, et ne pas procéder à une fusion. On doit alors faire migrer les attributs de l'association ainsi que les identifiants des entités de cardinalités 1,0 vers l'une des entités de cardinalité 1,1 la plus significative.

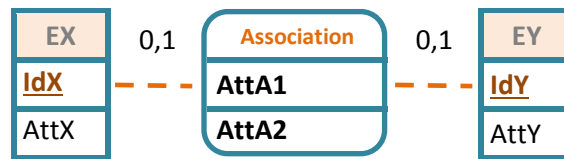


Nous devons à ce moment obtenir les relations suivantes :

EX(IdX, AttX)

EY(IdY, #IdX, AttY, AttA1, AttA2)

- Le cas où fonctionnellement les cardinalités peuvent être amenées à évoluer dans le temps, et que l'on souhaite anticiper ce changement. L'association devient alors une entité dont l'identifiant sera composé des identifiants des entités en association.



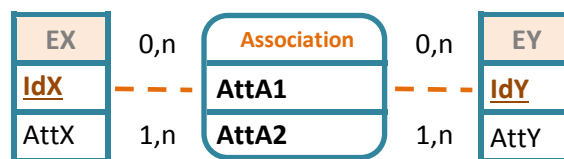
Nous devons à ce moment obtenir les relations suivantes :

EX(IdX, AttX)

EY(IdY, AttY)

EXY(#IdX, #IdY, AttA1, AttA2)

- b) Une association binaire ou n-aire dont les cardinalités maximales sont toutes à n, doit être transformée en une relation (table) dont la clé sera composée des identifiants des entités en association.



Nous devons à ce moment obtenir les relations suivantes :

EX(IdX, AttX)

EY(IdY, AttY)

EXY(#IdX, #IdY, AttA1, AttA2)

- c) Une association binaire dont les cardinalités maximales sont respectivement 1 et n, doit être éliminée en faisant migrer ses attributs vers l'entité présentant la cardinalité maximale 1. Une référence de la clé primaire de l'entité de cardinalité maximale n doit être ajoutée dans l'entité de cardinalité maximale 1. Cette référence à la clé primaire est nommée clé étrangère.



Nous devons à ce moment obtenir les relations suivantes :

EX(IdX, AttX)

EY(IdY, #IdX, AttY, AttA1, AttA2)

I.3. Le modèle physique des données «MPD»

Le passage au modèle physique de données est matérialisé par la traduction du modèle logique des données à travers un système de gestion de bases de données. Pour faire l'analogie avec l'algorithmique, cela correspond à la traduction de l'algorithme avec un langage de programmation compréhensible par l'ordinateur.

Cette traduction passe par la création physique de la base de données, la création des tables et la création des colonnes. Pour les colonnes, on doit préciser leurs natures (types de données), et éventuellement les contraintes qui leurs sont liées.

Etant donné qu'on est censé traiter des quantités importantes d'informations stockées dans une base de données, cette étape est également l'occasion pour penser à l'optimisation des performances en termes de temps de calcul. En effet, en faisant de la dénormalisation - malgré les risques d'incohérence des données et les problèmes de gestion que cela implique - on pourra réduire considérablement le temps de réponse de certaines requêtes. Cette optimisation se fait souvent aussi au détriment de l'espace mémoire. A titre d'exemple, l'indexation de certaines colonnes consomme de l'espace mémoire supplémentaire, mais permet d'accélérer les recherches tout en gardant la base de données normalisée. De la même manière, l'ajout de champs calculables permet d'éviter d'effectuer des jointures et des calculs sur une quantité importantes de données. Dans ce cas de figure, la base de données se retrouve dénormalisée, mais la cohérence pourra être assurée à travers des déclencheurs.