

# SQL déclaratif

## du standard à la pratique

Langage d'interrogation des Bases de  
Données

# Plan du cours

## Partie 1 : Introduction aux concepts

- Base de données et SGBD 10
- De l'analyse au relationnel 14
- Notions de tables 23
- Contraintes 25

# Plan du cours

## Partie 2 : DDL – Data Definition Language

*(Langage de définition des données)*

- **CREATE TABLE** **35**
- **IDENTITY et DEFAULT** **43**
- **Contraintes** **46**
- **ALTER TABLE** **58**
- **TRUNCATE TABLE** **61**
- **DROP TABLE** **62**

# Plan du cours

## Partie 3 : DRL – Data Retrieval Language

*(Langage de d'extraction des données)*

- La clause « SELECT » 67
- Limiter et ordonner 78
- Les fonctions 96
- GROUP BY 126
- Jointures 137
- Sous-requêtes 164

# Plan du cours

## Partie 4 : DML – Data Manipulation Language

*(Langage de manipulation des données)*

- Insertion de données 187
- Mise à jour de données 193
- Suppression de données 196
- OUTPUT 197

## Partie 5 : Notions avancées

- Gestion des transactions 199
- Fusion de données 202

# Auto-Evaluation

Afin de vous assurer que vous êtes en phase avec la formation et que vous intégrez la matière correctement et à un bon rythme, nous vous proposerons, à la fin de chaque module du cours, un petit tableau d'évaluation. Ce tableau a pour but de rappeler les notions phares du module et nous vous invitons à le remplir pour vous-même afin de vous rendre compte des notions que vous devez peut-être revoir, de celles pour lesquelles vous devrez demander plus d'explications au formateur ou encore que vous avez tout compris !

Dans ces petites auto-évaluations, les lettres suivantes sont utilisées pour vous aider évaluer le niveau avec lequel vous sentez avoir compris la matière :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit. Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Résumé de l'ensemble des notions

Notions	P	S	V	I
Base de données et SGBD				
Schéma relationnel (création, fonctionnement, lecture)				
Contrainte de « PRIMARY KEY »				
Contrainte de « FOREIGN KEY »				
Contraintes « UNIQUE », « CHECK », « NOT NULL »				
Création de table et contraintes (CREATE TABLE)				
Auto-incrémentation				
Modification de la structure d'une table existante (ALTER TABLE)				
Réinitialisation d'une table et suppression (TRUNCATE et DROP)				

# Auto-Evaluation

## Résumé de l'ensemble des notions

Notions	P	S	V	I
Ordre « SELECT ... FROM » simple				
Clauses WHERE, GROUP BY et HAVING, ORDER BY				
Fonctions simples et fonctions d'agrégation				
Jointures				
Requêtes imbriquées				
Ordres DML (INSERT, UPDATE, DELETE)				
Transaction (principes, loi ACID, ordres COMMIT et ROLLBACK)				



Partie 1

# INTRODUCTION AUX CONCEPTS

Base de données et SGBD

De l'analyse au relationnel

Notions de tables

Contraintes

# Base de données et SGBD

Une **Base de données** est une structure, le plus souvent informatique, permettant le stockage et l'exploitation de données

**Pendant longtemps**, nous nous sommes contentés de stocker nos données sur **des supports papiers**, dans des classeurs, dans des livres, dans des bibliothèques. Sans autre ressource technologique, cette technique rendait parfois le stockage, l'organisation et l'exploitation des données **un peu lourd**, sans compter **l'espace nécessaire** pour entreposer ces données

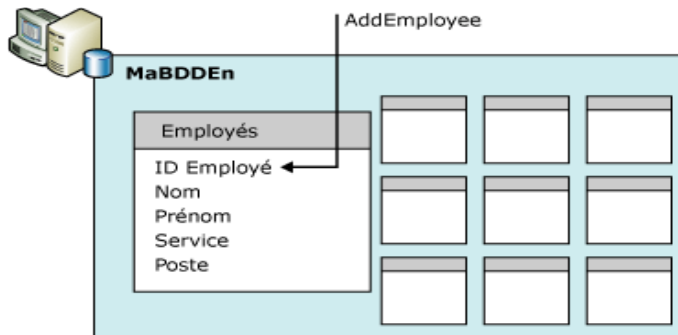
**L'arrivée de l'ordinateur** et des programmes informatiques a ensuite permit de **stocker les données dans des fichiers informatiques**, simplifiant et allégeant déjà énormément la gestion des données. Le gros désavantage des fichiers informatiques tels que ceux utilisés dans Excel ou Access est bien entendu que **ces fichiers ne se mettent pas à jour simultanément** pour l'ensemble des utilisateurs, chacun travaillant avec sa propre copie des données, la faisant évoluer à sa guise

Aujourd'hui, nous nous dirigeons donc vers des **bases de données dites « relationnelles »**, stockées sur un ou plusieurs **serveurs centralisés** qui peuvent être accédés par **de nombreux clients** de façon simultanée. Cela permet de rendre l'information disponible en temps réel, **partout et tout le temps**

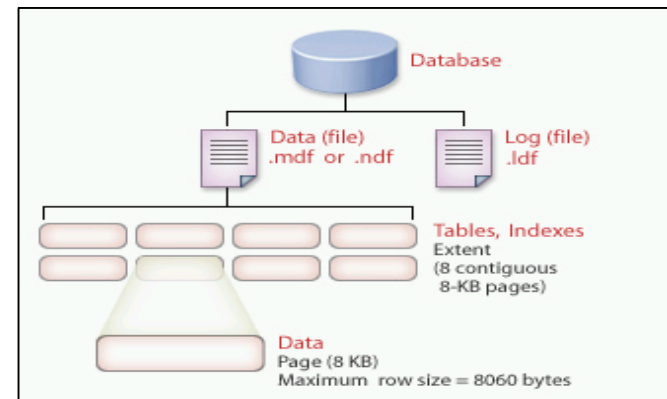
# Base de données et SGBD

Un **Système de Gestion de Base de Données (SGBD)** est un programme informatique permettant de gérer des bases de données

Aujourd'hui, nombreux sont les systèmes permettant de créer des bases de données relationnelles. Un SGBD a la caractéristique de venir renforcer une base de données en lui apportant **un certains nombre de fonctionnalités supplémentaires**. C'est le nombre de fonctionnalités, leur efficacité et leur fluidité qui font qu'un SGBD sera plus populaire ou meilleur qu'un autre



*Structure logique*



*Structure physique*

# Base de données et SGBD

## Fonctionnalités principales d'un SGBD :

- **Cohérence des données**

Un SGBD doit garantir que les données contenues dans les bases de données respectent et respecteront toujours les **règles de logique relationnelles établies**

- **Concurrence d'accès aux données**

Lorsque deux utilisateurs essayent d'accéder simultanément aux données, le système doit être capable d'assurer un ordre logique d'exécution des requêtes de chaque utilisateur, **en respectant les règles de transactions** qu'il a établi

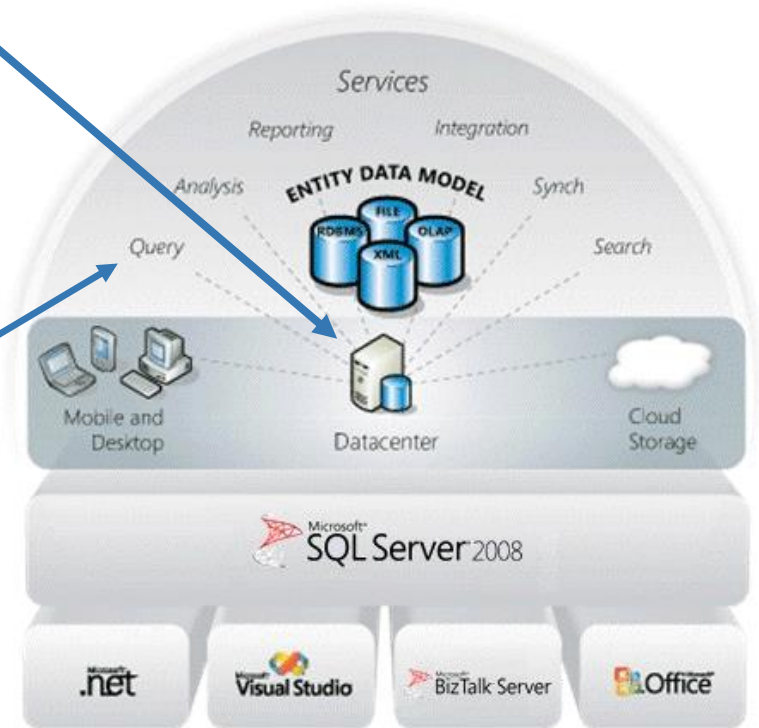
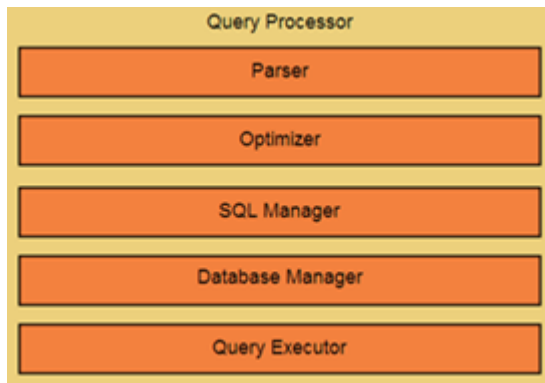
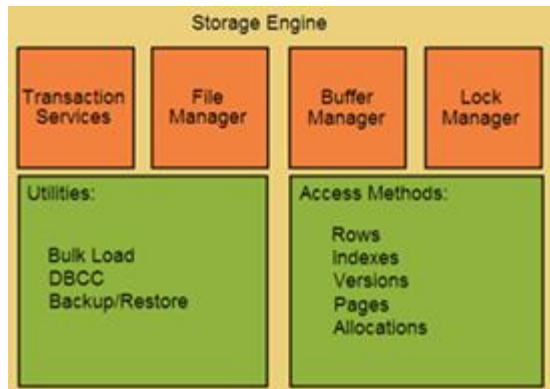
- **Sécurité des données**

Certainement le point le plus délicat qu'un SGBD doit pouvoir gérer, **la sécurité et l'accès aux données** est bien souvent la chose la plus importantes pour les entreprises. Le système doit pouvoir garantir que seuls les utilisateurs autorisés accèderont aux données dont ils ont besoin

- **Pérennité des données**

Avoir des données, c'est bien, les récupérer après un crash, c'est mieux ! Un SGBD doit donner accès à des outils ou des **méthodes de sauvegarde et de récupération** des données afin de pouvoir faire face à n'importe quelle panne logicielle, matérielle ou autre

# Base de données et SGBD



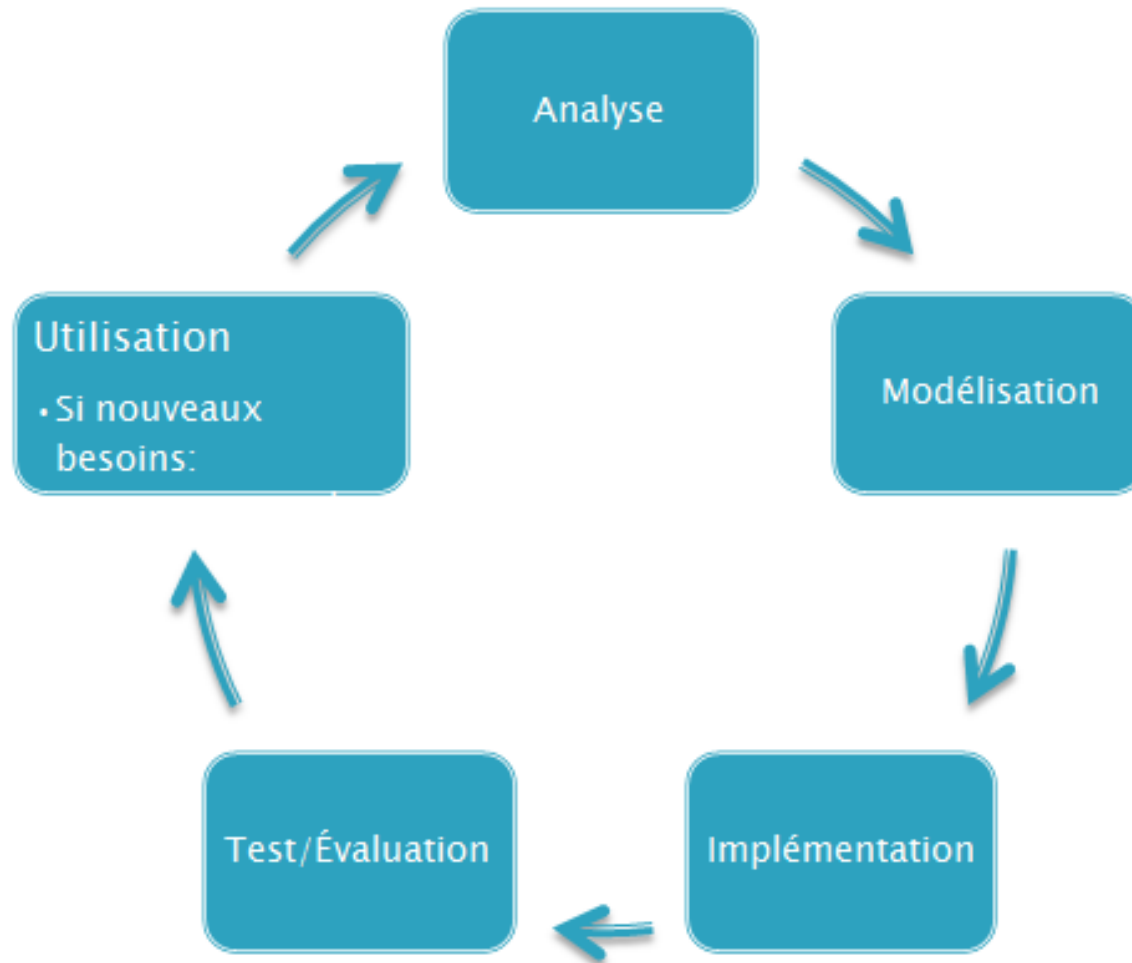
# De l'analyse au relationnel

Avant d'arriver à la création de la base de données elle-même, **il est nécessaire d'analyser en profondeur le problème rencontré**. Cette phase d'analyse est nécessaire afin de ne rien oublier et de gagner un temps précieux au niveau du développement et de l'implémentation de la solution

La phase d'analyse passera par **plusieurs étapes** contenant chacune **un certain nombre de schémas et de diagrammes**. Il s'agira la plupart du temps d'appliquer un modèle d'analyse tel que *UML*, dans son intégralité ou partiellement du moins. Ces différents schémas permettront d'établir **le « schéma relationnel » de la base de données**, qui doit permettre aux développeurs de générer la base de données elle-même

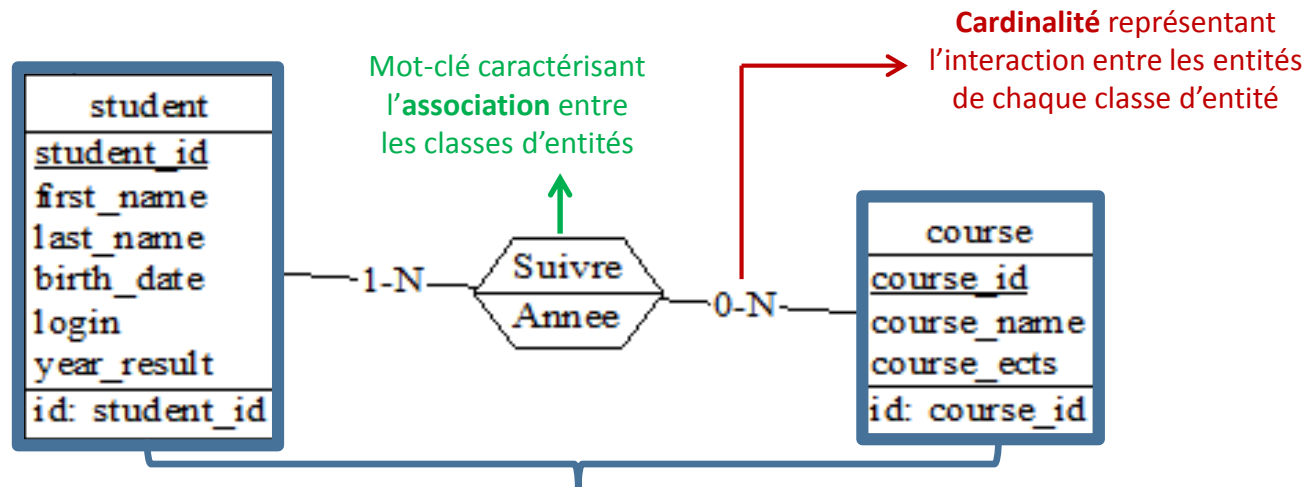
Dans certains cas simples, il est possible de se limiter à deux schémas. Le « **schéma Entités-Associations** » donnera alors directement la possibilité de passer au **schéma relationnel**

# De l'analyse au relationnel



# De l'analyse au relationnel

**Le schéma entités-associations (EA)** modélise simplement chaque acteur (**entité**) d'un système donné, en spécifiant chacun de leurs attributs qu'il est nécessaire d'enregistrer. Il indique **également la façon dont les acteurs interagissent** les uns avec les autres

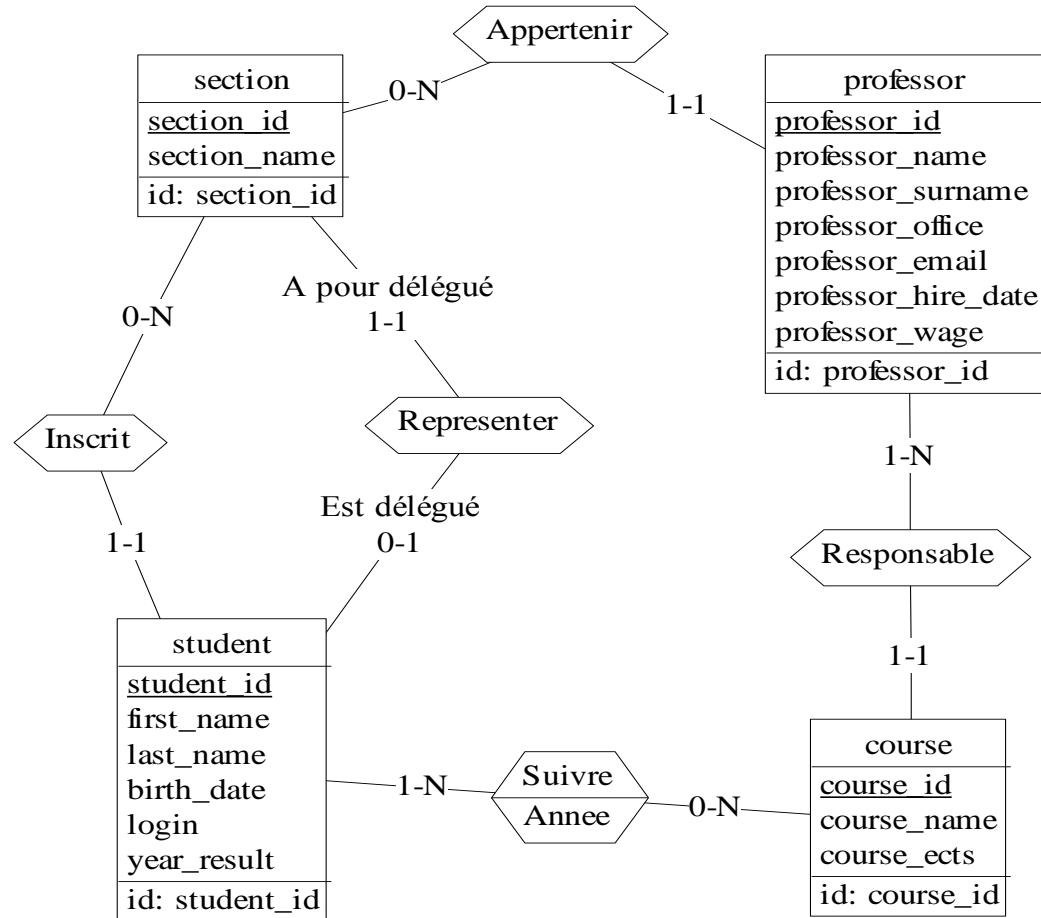


Classes d'entités reprenant les **attributs** (caractéristiques) de chacune des **entités** (acteurs) du système modélisé



# De l'analyse au relationnel

Exemple de  
**schéma EA**  
représentant le  
système  
d'information  
d'une université



# De l'analyse au relationnel

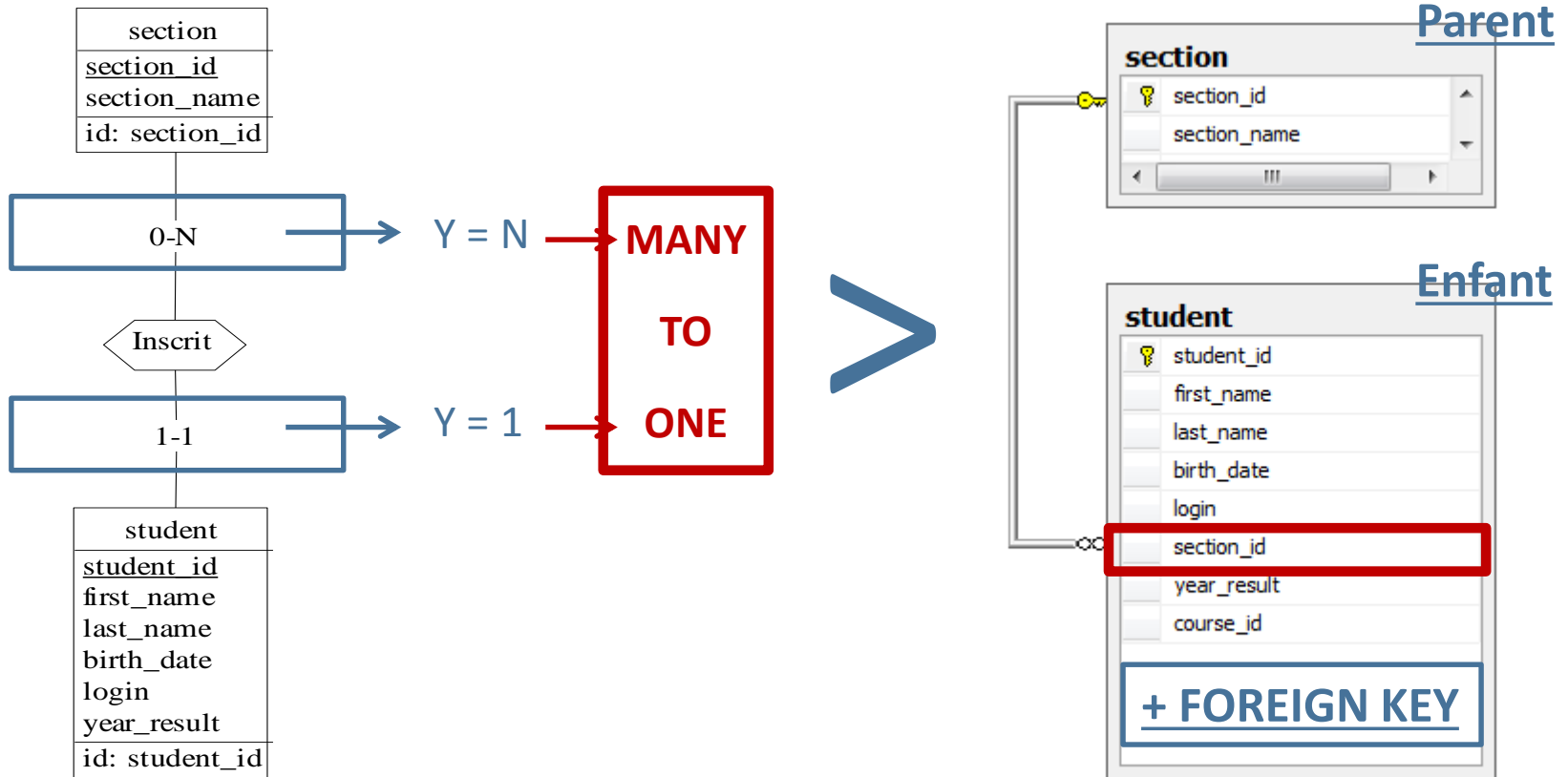
**Le schéma relationnel** est la traduction du schéma entités-associations  
Un schéma relationnel représente le plan de la base de données  
Il peut être lu aussi bien par l'analyste que par le programmeur

La traduction du schéma entités-associations se fait en appliquant certaines règles simples dont voici un aperçu succinct :

- Les **classes d'entités** deviennent des **tables**
- Les **attributs** deviennent des **colonnes**
- Les **associations** deviennent des **contraintes d'intégrité référentielles** selon les règles établies dans les slides suivants

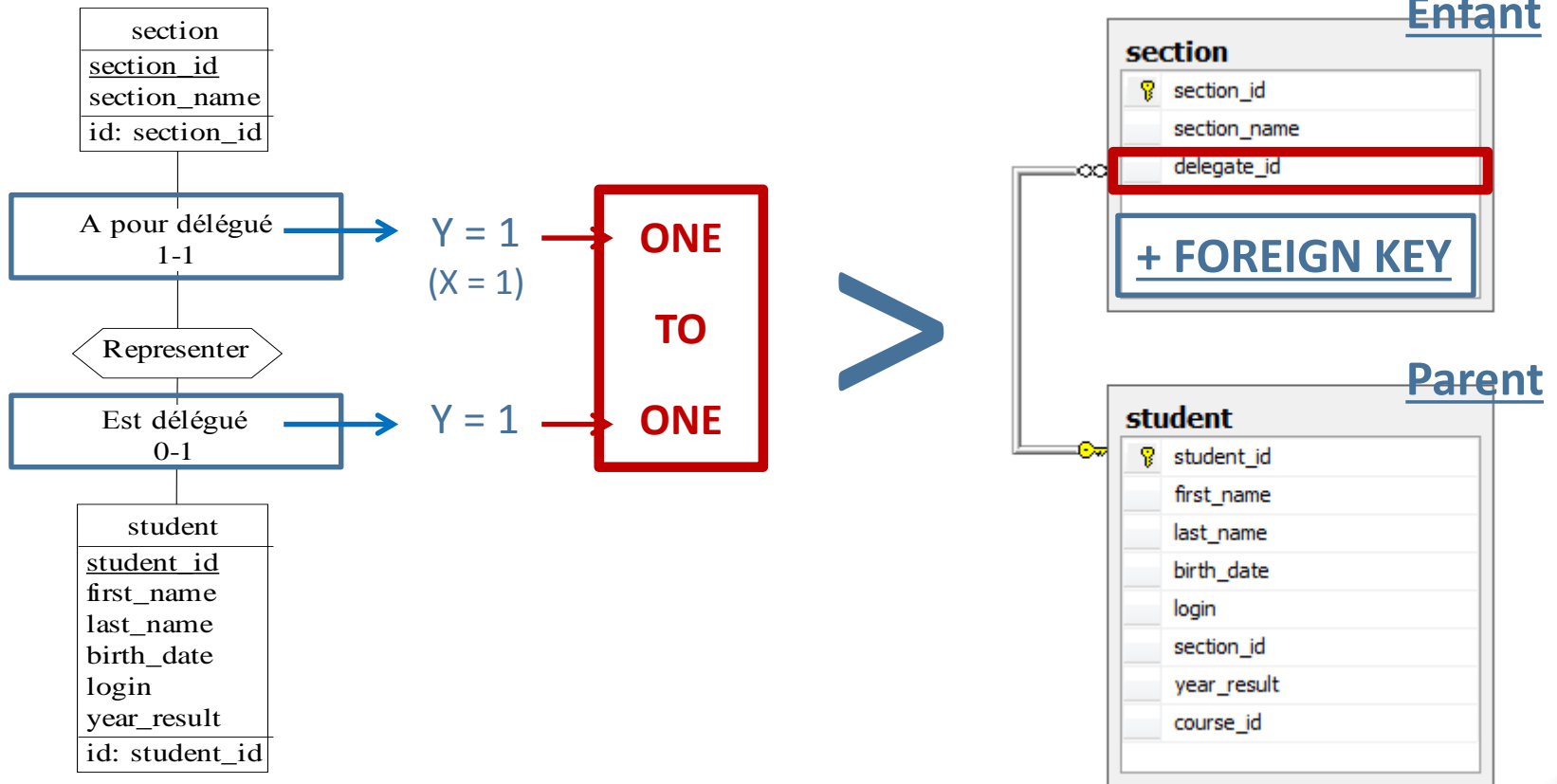
# De l'analyse au relationnel

## Traduction d'une association de type « One-to-Many » :



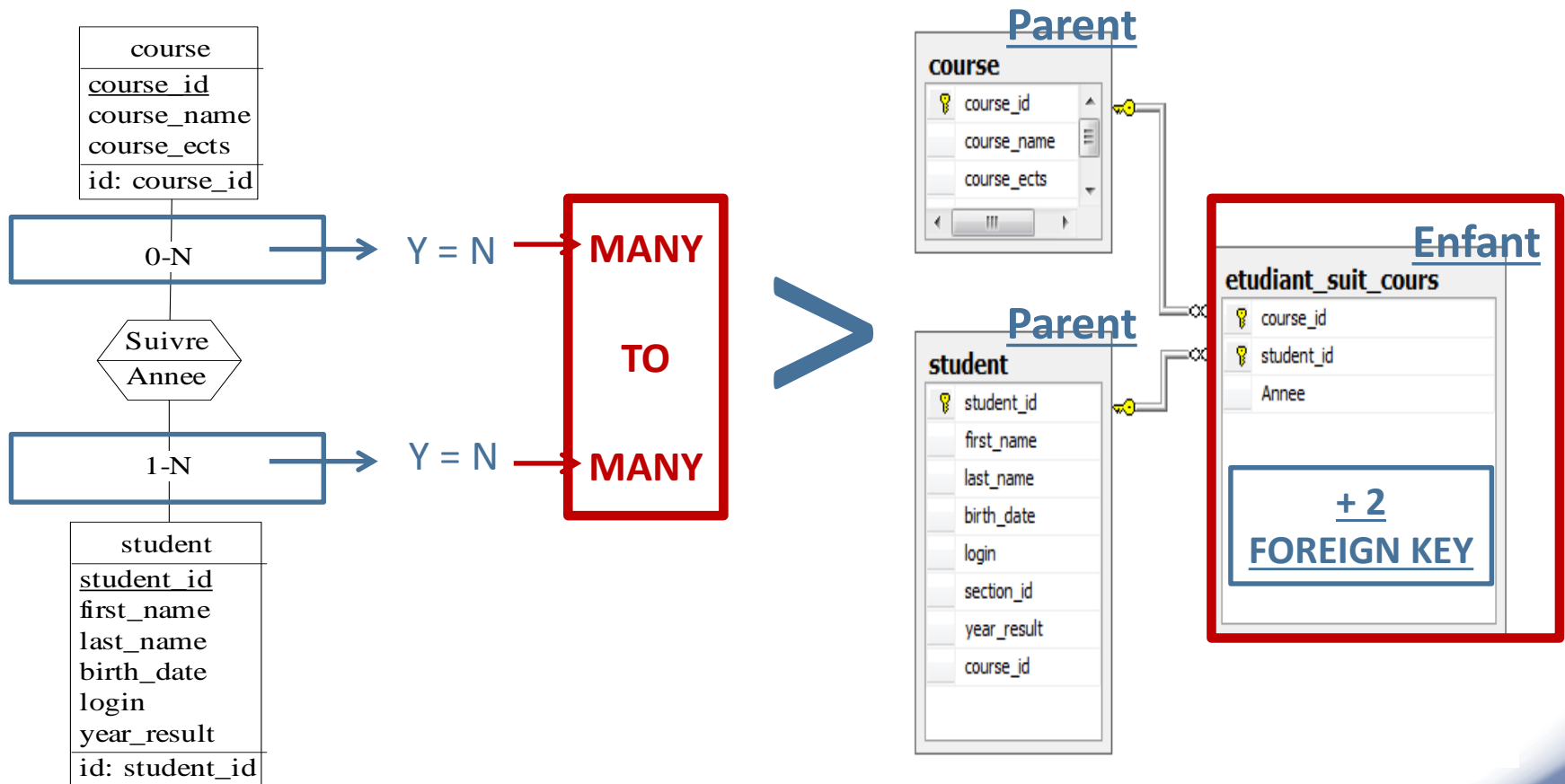
# De l'analyse au relationnel

## Traduction d'une association de type « One-to-One » :

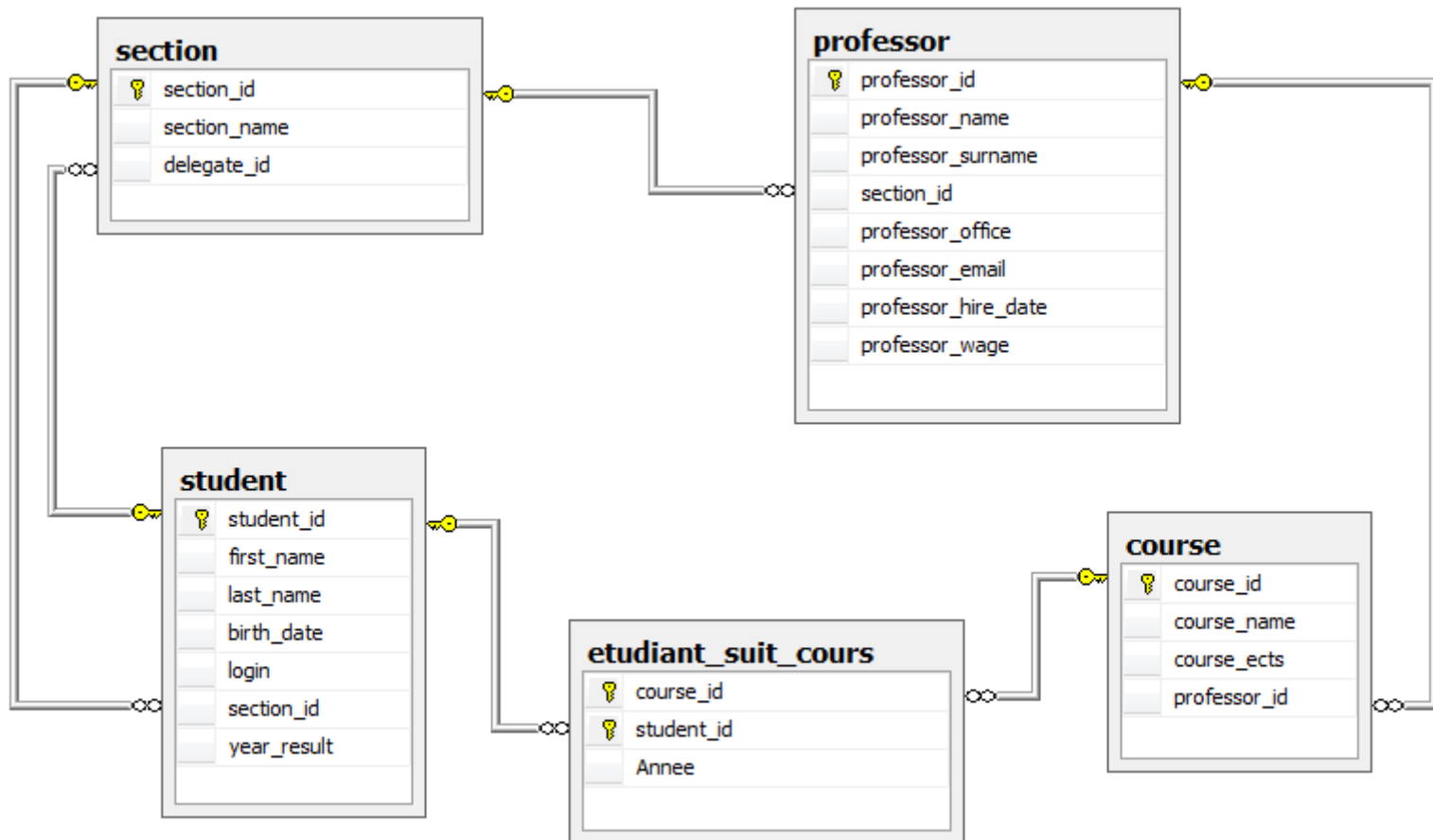


# De l'analyse au relationnel

## Traduction d'une association de type « Many-to-Many » :

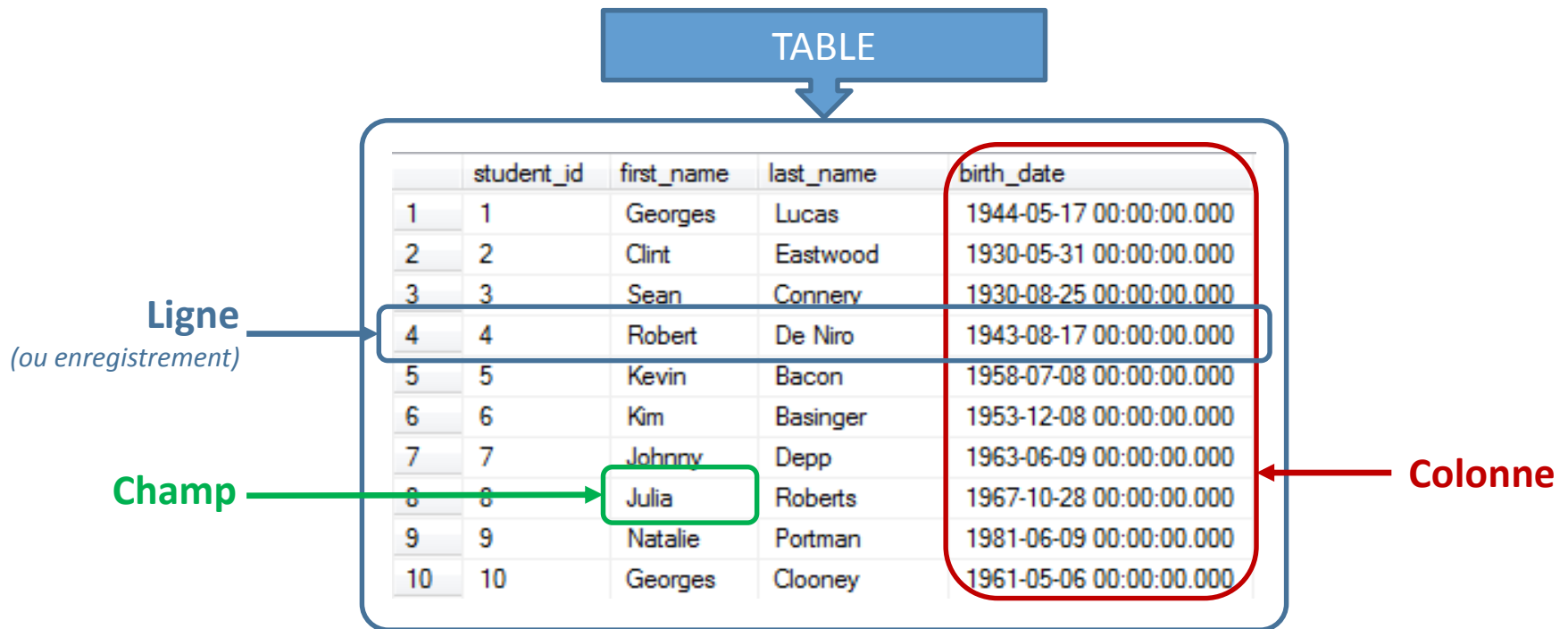


# De l'analyse au relationnel



# Notions de tables

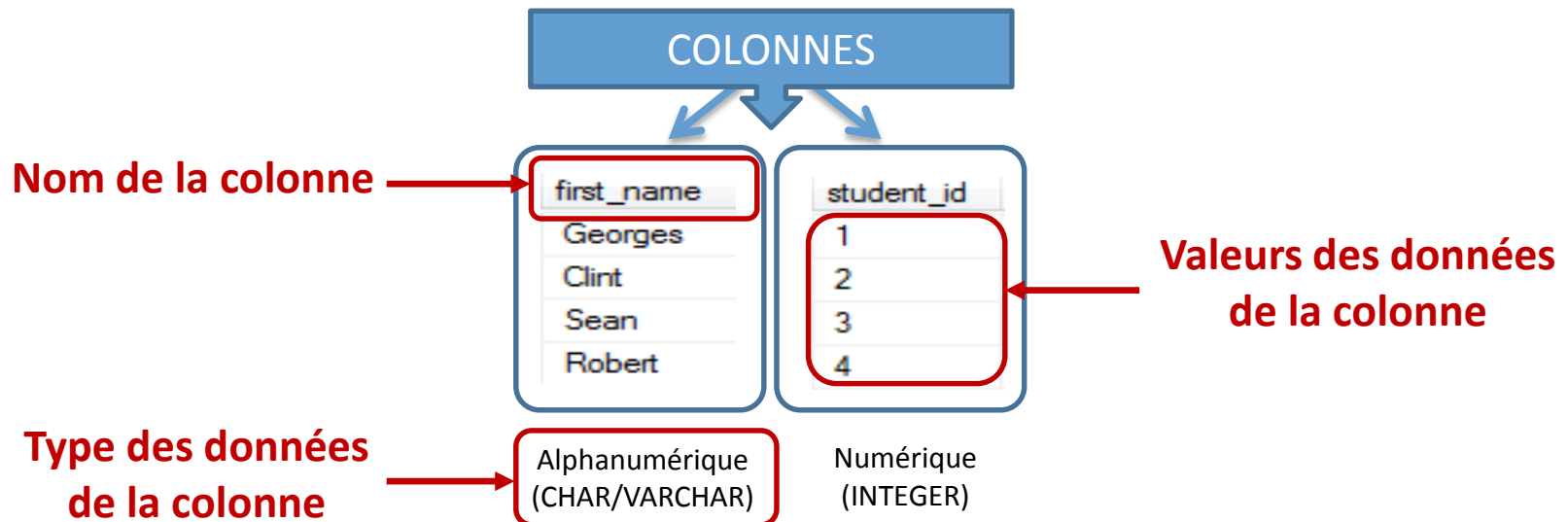
Une **Table** regroupe des **ensembles de données (d'informations)** stockés de façon permanente et décrivant chacun un acteur du système réel modélisé (**Entité**)



*La Table étudiant contient des informations sur les étudiants uniquement*

# Notions de tables

Une **Colonne** est un **attribut** d'une table, elle représente une **caractéristique particulière de l'objet réel** représenté





# Contraintes

Une **Contrainte** est objet intégré à une table (*comme les colonnes*)  
Une contrainte possède **un nom, un type**  
et **concerne au moins une colonne** de la table

On distingue principalement **5 types de contraintes différentes**

Seule la contrainte « **NOT NULL** » n'aura **pas de nom**

Il n'existera au maximum **qu'une seule contrainte de « Clé Primaire »** pour chaque table

Contraintes	Description
<b>NOT NULL</b>	Force la présence d'une valeur (valeur obligatoire)
<b>UNIQUE</b>	Empêche les valeurs-doublons
<b>CHECK</b>	Conditionne les valeurs (expression conditionnelle)
<b>FOREIGN KEY</b>	Conditionne les valeurs par rapport à une autre table

# Contraintes : NOT NULL

Ajouter la contrainte « **NOT NULL** » à la colonne d'une table obligera cette colonne à **contenir une valeur** pour chacune des lignes de la table

Une colonne non-obligatoire est **facultative**, elle peut contenir la valeur **NULL**

La valeur **NULL** spécifie que aucune valeur n'est attribuée

**NULL** représente l'absence de valeur, elle n'est ni 0 ni une case vide

The diagram shows a table with 9 columns. Annotations highlight specific constraints:

- Pas de valeur renseignée**: Points to the `section_id` column.
- Colonne facultative acceptant des valeurs NULL**: Points to the `year_result` column, which contains a `NULL` value in row 6.
- Colonne obligatoire NULL**: Points to the `course_id` column, which contains `0` values in rows 5, 6, and 8.

	student_id	first_name	last_name	birth_date	login	section_id	year_result	course_id
1	1	Georges	Lucas	1944-05-17 00:00:00.000	glucas	1320	10	EG2210
2	2	Clint	Eastwood	1930-05-31 00:00:00.000	ceastwoo	1010	4	EG2210
3	3	Sean	Connery	1930-08-25 00:00:00.000	sconnery	1020	12	EG2110
4	4	Robert	De Niro	1943-08-17 00:00:00.000	rde niro	1110	3	EG2210
5	5	Kevin	Bacon	1958-07-08 00:00:00.000	kbacon	1120	16	0
6	6	Kim	Basinger	1953-12-08 00:00:00.000	kbasinge	1310	NULL	0
7	7	Johnny	Depp	1963-06-09 00:00:00.000	jdepp	1110	11	EG2210
8	8	Julia	Roberts	1967-10-28 00:00:00.000	jroberts	1120	17	0

# Contraintes : **UNIQUE**

La contrainte d'unicité « **UNIQUE** » a la particularité d'empêcher **une colonne** ou **une combinaison de colonnes**, d'accepter deux valeurs identiques pour deux lignes différentes de la table

Colonne **non candidate** contient des **doublons**

Colonne **UNIQUE** aucun doublon

	student_id	first_name	last_name	birth_date	login	section_id	year_result	course_id
1	1	Georges	Lucas	1944-05-17 00:00:00.000	glucas	1320	10	EG2210
2	2	Clint	Eastwood	1930-05-31 00:00:00.000	ceastwoo	1010	4	EG2210
3	3	Sean	Connery	1930-08-25 00:00:00.000	sconnery	1020	12	EG2110
4	4	Robert	De Niro	1943-08-17 00:00:00.000	rde niro	1110	NULL	EG2210
5	5	Kevin	Bacon	1958-07-08 00:00:00.000	kbacon	1120	16	0
6	6	Kim	Basinger	1953-12-08 00:00:00.000	kbasinge	1310	NULL	0
7	7	Johnny	Depp	1963-06-09 00:00:00.000	jdepp	1110	11	EG2210
8	8	Julia	Roberts	1967-10-28 00:00:00.000	jroberts	1120	17	0
9	9	Natalie	Portman	1981-06-09 00:00:00.000	nportman	1010	4	EG2210
10	10	Georges	Clooney	1961-05-06 00:00:00.000	gclooney	1020	4	EG2110

Combinaison de colonnes **UNIQUE** aucun doublon

# Contraintes : **PRIMARY KEY**

**La contrainte de clé primaire** d'une table désigne un ensemble (souvent minimal) de colonnes qui identifient de manière unique les enregistrements d'une table. La combinaison des colonnes qui la composent doit être **UNIQUE** et **NON NULL**.

- Il ne peut exister qu'une et une seule clé primaire par table **MAIS** celle-ci peut être composée d'une combinaison de plusieurs colonnes
- **Une clé primaire est dite « naturelle »** si la ou les colonnes qui la composent sont des attributs représentant réellement une caractéristique de l'objet que la table décrit
- **Une clé primaire sera « artificielle »** si elle est représentée par une colonne dont les valeurs ne représentent rien dans le monde réel. Ces colonnes artificielles sont régulièrement utilisées car elles sont faciles à manipuler et, comme elles ont le plus souvent comme valeurs des nombres, leur contenu peut être géré directement par le SGBD. On dira alors, qu'en plus d'être une clé primaire artificielle, **les valeurs de la colonne sont auto-incrémentées**, le plus souvent de 1 à l'infini

# Contraintes : PRIMARY KEY

Combinaison **non candidate**  
(contient des valeurs **NULL**)

	student_id	first_name	last_name	birth_date	login	section_id	year_result	course_id
1	1	Georges	Lucas	1944-05-17 00:00:00.000	glucas	1320	10	EG2210
2	2	Clint	Eastwood	1930-05-31 00:00:00.000	ceastwoo	1010	4	EG2210
3	3	Sean	Connery	1930-08-25 00:00:00.000	sconnery	1020	12	EG2110
4	4	Robert	De Niro	1943-08-17 00:00:00.000	rde niro	1110	NULL	EG2210
5	5	Kevin	Bacon	1958-07-08 00:00:00.000	kbacon	1120	16	0
6	6	Kim	Basinger	1953-12-08 00:00:00.000	kbasinge	1310	NULL	0
7	7	Johnny	Depp	1963-06-09 00:00:00.000	jdepp	1110	11	EG2210
8	8	Julia	Roberts	1967-10-28 00:00:00.000	jroberts	1120	17	0
9	9	Natalie	Portman	1981-06-09 00:00:00.000	nportman	1010	4	EG2210
10	10	Georges	Clooney	1961-05-06 00:00:00.000	gclooney	1020	4	EG2110

**PRIMARY KEY**  
dite « **artificielle** »  
(**PEUT** être **auto-incrémentée**)

Combinaison **UNIQUE ET NOT NULL**  
candidate à la création d'une  
**PRIMARY KEY** « **naturelle** »

# Contraintes : **FOREIGN KEY**

**La contrainte de clé étrangère** d'une table (**enfant**) permet d'éviter la redondance d'information au niveau de cette table par le biais d'une colonne de référence pointant vers une colonne identifiante d'une autre table (**parent**)  
Contenant, elle, le détail de l'objet référencé

- La clé étrangère est une contrainte de la table, elle concerne une ou plusieurs colonnes de la table, mais ces colonnes ne sont pas implicitement créées lors de la création de la clé étrangère. De la même manière, modifier ou supprimer la clé étrangère ne modifie pas les caractéristiques de la colonne concernée
- Il peut exister plusieurs clés étrangères dans chaque table
- Plusieurs colonnes peuvent composer la même clé étrangère d'une table, si cette clé étrangère fait référence à plusieurs colonnes d'une autre table (clé primaire composite, clé d'unicité composite)
- Les colonnes référencées entre elles doivent être du même type
- Les valeurs d'une colonne utilisée comme clé étrangère peuvent être **NULL**

# Contraintes : FOREIGN KEY

## Table Livres

Information redondante

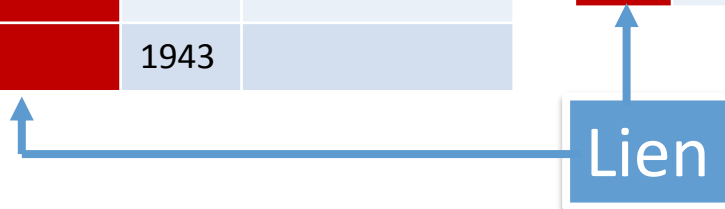
Titre	PrenomAuteur	NomAuteur	PseudoAuteur	NaissAuteur	DateLivres	ISBN
La Peste	Albert	Camus	Bebert	07/11/1913	1974	978-20703604
La Chute	Albert	Camus	Bebert	07/11/1913	1973	978-20703109
Huis-clos	Jean-Paul	Sartre	Jy-Pé	21/06/1905	1943	

## Table Livres

Titre	RefAuteur	Date	ISBN
La Peste	1	1974	978-20703604
La Chute	1	1973	978-20703109
Huis-clos	2	1943	

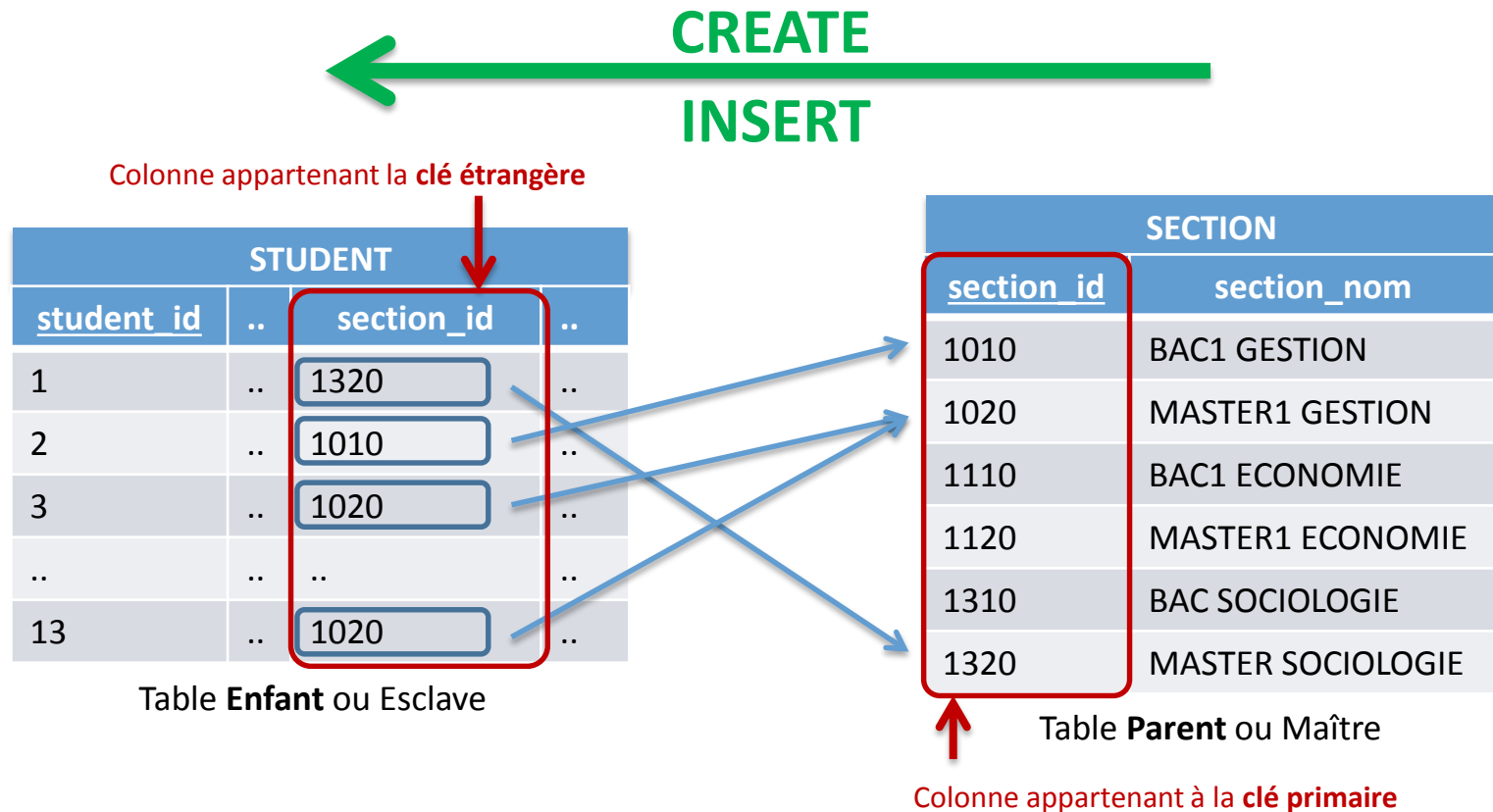
## Table Auteurs

Ref	Nom	Prenom	Pseudo	DateNaiss
1	Albert	Camus	Bebert	07/11/1913
2	Jean-Paul	Sartre	Jy-Pé	21/06/1905



La table **Livres** est liée à la table **Auteurs** par le champ **RefAuteur**.

# Contraintes : FOREIGN KEY



**DROP**  
**DELETE**



# Contraintes : CHECK

La contrainte **CHECK** d'une table permet de poser une condition spécifique sur les colonnes de la tables, afin d'y empêcher l'insertion de n'importe quelle valeur

	student_id	first_name	last_name	birth_date	login	section_id	year_result	course_id
1	1	Georges	Lucas	1944-05-17 00:00:00.000	glucas	1320	10	EG2210
2	2	Clint	Eastwood	1930-05-31 00:00:00.000	ceastwoo	1010	4	EG2210
3	3	Sean	Connery	1930-08-25 00:00:00.000	sconnery	1020	12	EG2110
4	4	Robert	De Niro	1943-08-17 00:00:00.000	rde niro	1110	NULL	EG2210
5	5	Kevin	Bacon	1958-07-08 00:00:00.000	kbacon	1120	16	0
6	6	Kim	Basinger	1953-12-08 00:00:00.000	kbasinge	1310	NULL	0
7	7	Johnny	Depp	1963-06-09 00:00:00.000	jdepp	1110	11	EG2210
8	8	Julia	Roberts	1967-10-28 00:00:00.000	jroberts	1120	17	0
9	9	Natalie	Portman	1981-06-09 00:00:00.000	nportman	1010	4	EG2210
10	10	Georges	Clooney	1961-05-06 00:00:00.000	gclooney	1020	4	EG2110

$0 \leq \text{year\_result} \leq 20$

$\text{Birth\_date} > 01-01-1930$

## Partie 2

# DDL – DATA DEFINITION LANGUAGE

CREATE TABLE

IDENTITY et DEFAULT

Contraintes

ALTER TABLE

TRUNCATE TABLE

DROP TABLE

# CREATE TABLE

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  nom_colonne3 TYPE  
)
```

```
CREATE TABLE student (  
  student_id int,  
  first_name varchar(50),  
  last_name varchar(50),  
  birth_date datetime,  
  login varchar(50),  
  section_id int,  
  year_result int,  
  course_id varchar(6)  
)
```

# CREATE TABLE

- **Une table possède un NOM**

Ce nom peut être composé de **128 caractères** et est choisi par l'utilisateur. Il est conseillé de choisir des **noms clairs et concis**. Les **caractères spéciaux** comme les accents et les espaces blancs sont **interdits**

- **Une table possède des COLONNES**

Le nom de ces colonnes est limité par les mêmes contraintes que celles énoncées pour le nom de la table

- **Les colonnes ont un TYPE**

Le type de la colonne définit le type qu'auront les valeurs de cette colonne. Les types principaux sont **INT** (pour entiers), **VARCHAR(taille)** (pour une chaîne de caractères), **DECIMAL(X,Y)** ou **FLOAT** (pour un chiffre décimal), **DATE** (pour une date). Une liste plus détaillée des différents types de données est fournie dans les slides suivants

# CREATE TABLE

## Types principaux de données

Nom	Taille sur le disque (octets)	Description
INTEGER ( <i>INT</i> )	4	Valeurs entières de $-2^{31}$ à $2^{31}$
DECIMAL( <i>x,y</i> ) ( <i>DEC</i> )	5 à 17	Valeurs numériques contenant « x » chiffres au total, dont « y » après la virgule
VARCHAR( <i>x</i> )	Nombre de caractères + 2	Chaîne de caractères d'une taille variable pouvant aller de 1 caractère à 8000 caractères (non-Unicode)
DATE	3	Dates du 01/01/0001 au 31/12/9999
BIT	1	Valeurs 1, 0 ou NULL
VARBINARY( <i>x</i> )	Nombres de bits + 2	Valeurs binaires d'une taille de 1 à 8000 (stockage de fichiers de type son, image ou vidéo)

# CREATE TABLE

## Types de données numériques

Nom	Taille sur le disque (octets)	Description
TINYINT	1	Valeurs entières de 0 à 255
SMALLINT	2	Valeurs entières de $-2^{15}$ à $2^{15} - 1$
BIGINT	8	Valeurs entières de $-2^{63}$ à $2^{63}$
NUMERIC(x,y)	5 à 17	(Idem DECIMAL)

## Types de données d'approximation

Nom	Taille sur le disque (octets)	Description
FLOAT(x) ( $1 \leq x \leq 53$ )	4 à 8 (dépend de x)	Valeurs numériques d'approximation (e)
REAL	4	Valeurs numériques d'approximation

# CREATE TABLE

## Types de données textuelles

Nom	Taille sur le disque (octets)	Description
CHAR(x)	4	<i>(Idem VARCHAR mais de taille fixe, non-Unicode)</i>
NCHAR(x)	Nombre de caractères * 2	Chaîne de caractères d'une taille fixe pouvant aller de 1 caractère à 4000 caractères (Unicode)
NVARCHAR(x)	Nbre de caractères * 2 + 2	Chaîne de caractères d'une taille variable pouvant aller de 1 caractère à 4000 caractères (Unicode)
TEXT <b>DÉPRÉCIÉ</b>	2 <sup>31</sup> -1 (2 147 483 647)	Chaîne de caractères d'une taille variable pouvant aller jusqu'à 2 <sup>31</sup> -1 caractères (non-Unicode)
NTEXT <b>DÉPRÉCIÉ</b>	Nombre de caractères * 2	Chaîne de caractères d'une taille variable pouvant aller jusqu'à 2 <sup>30</sup> -1 (1 073 741 823) caractères (Unicode)

# CREATE TABLE

## Types de données de dates et heures

Nom	Taille sur le disque (octets)	Description
TIME	5	Heures allant de 00:00:00 à 23:59:59.9999999
SMALLDATETIME	4	Dates allant du 01/01/1900 au 06/06/2079 et heures allant de 00:00:00 à 23:59:59
DATETIME	8	Dates allant du 01/01/1753 au 31/12/9999 et heures allant de 00:00:00 à 23:59:59.997
DATETIME2	6 à 8	Dates allant du 01/01/0001 au 31/12/9999 et heures allant de 00:00:00 à 23:59:59.9999999



# CREATE TABLE

## Autres types de données

Nom	Taille sur le disque (octets)	Description
TIMESTAMP <i>(DÉPRÉCIÉ, utiliser « ROWVERSION »)</i>	8	Champ auto-incrémenté qui génère une nouvelle valeur chaque fois que la ligne est mise à jour (ex : 0x00000000000007D4)
UNIQUEIDENTIFIER	16	Valeurs hexadécimales uniques (ex : 6F9619FF-8B86-D011-B42D-00C04FC964FF)
XML	Maximum 2Go	Données XML

Vous trouverez une liste complète des types utilisés sur SQL-Server à l'adresse suivante :  
<http://msdn.microsoft.com/en-us/library/ms187752.aspx>

# CREATE TABLE

## Types principaux de données sous Oracle 12c

Nom	Taille sur le disque (octets)	Description
NUMBER(x,y)	1 à 22	Valeurs numériques contenant « x » chiffres au total, dont « y » après la virgule. (x,y) non-obligatoire, valeur max par défaut si non spécifiés Valeurs possibles : $-1.0 \times 10^{130}$ à $1.0^{126}$
VARCHAR2(x)	Nombre de caractères	Chaîne de caractères d'une taille variable pouvant aller de 1 caractère à 4000 caractères (non-Unicode)
DATE	7	Dates du 01/01/0001 au 31/12/9999

Vous trouverez une liste complète des types utilisés sur Oracle 12c à l'adresse suivante :  
[http://docs.oracle.com/cd/E16655\\_01/server.121/e17209/sql\\_elements001.htm#SQLRF50972](http://docs.oracle.com/cd/E16655_01/server.121/e17209/sql_elements001.htm#SQLRF50972)

# IDENTITY et DEFAULT

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE IDENTITY(1,1),  
  nom_colonne2 TYPE DEFAULT valeur_par_défaut,  
  nom_colonne3 TYPE  
)
```

```
CREATE TABLE student (  
  student_id int IDENTITY(1,1),  
  first_name varchar(50),  
  last_name varchar(50) DEFAULT 'Smith',  
  birth_date datetime,  
  login varchar(50),  
  section_id int,  
  year_result int DEFAULT 0,  
  course_id varchar(6)  
)
```

# IDENTITY et DEFAULT

- **Les valeurs d'une colonne peuvent être AUTO-INCRÉMENTÉES**

Sous SQL-Server, le mot-clé « **IDENTITY** » permet de demander au système d'attribuer lui-même des valeurs à la colonne concernée. Ces valeurs commencent bien souvent à 1 et sont incrémentées de 1 à chaque nouvelle ligne « **(1,1)** ». Il ne faut pas s'occuper de cette colonne lors de l'insertion d'une nouvelle ligne dans la table

**ATTENTION :** Sous SQL Server, il n'est pas possible (*en utilisant du code*) de rajouter l'auto-incrémentation d'une colonne lorsque la table est déjà créée, il faut y penser à la création ! Sinon il sera nécessaire de supprimer et recréer la table...

**Remarque :** Comme expliqué plus loin, l'ordre DDL « **TRUNCATE TABLE** » permet de vider la table de ses données et de réinitialiser le compteur de l'auto-incrémentation par la même occasion

- **Une colonne peut posséder une VALEUR PAR DÉFAUT**

Le mot-clé « **DEFAULT** » permet d'insérer une valeur par défaut dans la colonne concernée, si aucune valeur n'est spécifiée pour cette colonne lors de l'insertion d'une nouvelle ligne dans la table. Si une valeur est renseignée, elle remplace bien entendu la valeur par défaut. La contrainte « **NOT NULL** » n'a aucun sens lorsque « **DEFAULT** » est utilisé et peut être source d'erreur sur certaines plateformes

# IDENTITY et DEFAULT

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE GENERATED ALWAYS AS IDENTITY,  
  nom_colonne2 TYPE DEFAULT valeur_par_défaut,  
  nom_colonne3 TYPE  
)
```

```
CREATE TABLE student (  
  student_id NUMBER GENERATED ALWAYS AS IDENTITY,  
  first_name VARCHAR2(50),  
  last_name VARCHAR2(50) DEFAULT 'Smith',  
  birth_date DATE,  
  login VARCHAR2(50),  
  section_id INT,  
  year_result INTEGER DEFAULT 0,  
  course_id CHAR(6)  
);
```

# Contraintes

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  CONSTRAINT nom_contrainte TYPE CONTRAINTE (colonne_concernée)  
)
```

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE CONSTRAINT nom_contrainte TYPE CONTRAINTE,  
  nom_colonne3 TYPE  
)
```

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE TYPE CONTRAINTE,  
  nom_colonne3 TYPE  
)
```

# Contraintes : NOT NULL

```
CREATE TABLE student (  
    student_id int NOT NULL,  
    first_name varchar(50),  
    last_name varchar(50) NOT NULL,  
    birth_date datetime,  
    login varchar(50),  
    section_id int,  
    year_result int,  
    course_id varchar(6) NOT NULL  
)
```

***La contrainte NOT NULL est la seule contrainte qui n'est pas nommée,  
elle est incluse dans la définition de la colonne***

# Contraintes : UNIQUE

```
CREATE TABLE student (  
  student_id int NOT NULL UNIQUE,  
  first_name varchar(50),  
  last_name varchar(50) NOT NULL,  
  birth_date datetime,  
  login varchar(50) CONSTRAINT UK_login UNIQUE,  
  section_id int,  
  year_result int,  
  course_id varchar(6) NOT NULL,  
  CONSTRAINT UK_Lname_Bdate UNIQUE (last_name, birth_date)  
)
```

Nom contrainte

Type contrainte

Combinaison de colonnes concernées

***Toute contrainte aura un nom dans le système. Si nous ne la nommons pas nous-même, le système choisira lui-même un nom pour la contrainte***



# Contraintes : PRIMARY KEY

```
CREATE TABLE student (  
    student_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date datetime,  
    login varchar(50),  
    section_id int,  
    year_result int,  
    course_id varchar(6),  
    CONSTRAINT PK_student PRIMARY KEY (student_id)  
)
```

# Contraintes : **PRIMARY KEY**

```
CREATE TABLE student (  
    student_id int PRIMARY KEY,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date datetime,  
    login varchar(50),  
    section_id int,  
    year_result int,  
    course_id varchar(6)  
)
```

# Contraintes : **FOREIGN KEY**

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  CONSTRAINT nom_contrainte TYPE CONTRAINTE (colonne_concernée)  
    REFERENCES table_référencée (colonne_référencée)  
)
```

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE REFERENCES table_référencée,  
  nom_colonne3 TYPE  
)
```

*Le paramètre « **colonne\_référencée** » peut être omis  
si les colonnes portent le même nom dans la table enfant et la table parent*

# Contraintes : FOREIGN KEY

```
CREATE TABLE student (  
    student_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date datetime,  
    login varchar(50),  
    section_id int,  
    year_result int,  
    course_id varchar(6),  
    CONSTRAINT FK_student_section FOREIGN KEY (section_id)  
        REFERENCES section (section_id)  
)
```

# Contraintes : FOREIGN KEY

```
CREATE TABLE student (  
    student_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date datetime,  
    login varchar(50),  
    section_id int  
        CONSTRAINT FK_student_section  
        REFERENCES section (section_id),  
    year_result int,  
    course_id varchar(6)  
)
```

# Contraintes : FOREIGN KEY

```
CREATE TABLE student (  
    student_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date datetime,  
    login varchar(50),  
    section_id int REFERENCES section,  
    year_result int,  
    course_id varchar(6)  
)
```

# Contraintes : FOREIGN KEY

Les options « **ON DELETE** » et « **ON UPDATE** » peuvent être rajoutées à la contrainte de clé étrangère de façon à ne pas lever d'erreur lorsqu'une ligne de la table référencée est supprimée ou mise à jour. Dans ce cas, selon l'action spécifiée, la ligne de la table enfant sera supprimée ou modifiée.

- L'action « **CASCADE** » répercute la modification (ou suppression de ligne) sur la table enfant.
- L'action « **SET NULL** » met à NULL les valeurs correspondantes dans la table enfant (possible uniquement si la colonne accepte les valeurs NULL...)
- L'action « **SET DEFAULT** » remplace les valeurs correspondantes dans la table enfant par la valeur prévue par défaut (si aucune valeur par défaut n'a été renseignée, la valeur sera NULL).
- Si rien n'est spécifié, l'action par défaut est « **NO ACTION** » qui lève une erreur.

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  CONSTRAINT nom_contrainte TYPE CONTRAINT (colonne_concernée)  
    REFERENCES table_référencée (colonne_référencée)  
    ON DELETE action ON UPDATE action  
)
```

# Contraintes : FOREIGN KEY

```
CREATE TABLE student (  
    student_id int PRIMARY KEY,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date datetime,  
    login varchar (50),  
    section_id int  
    CONSTRAINT FK_student_section REFERENCES section  
        ON DELETE SET NULL ON UPDATE CASCADE,  
    year_result int,  
    course_id varchar(6)  
)
```



# Contraintes : CHECK

```
CREATE TABLE student (  
    student_id int PRIMARY KEY,  
    first_name varchar(50),  
    last_name varchar(50)  
    CONSTRAINT CK_last_name CHECK (last_name IS NOT NULL),  
    birth_date datetime,  
    login varchar (50),  
    section_id int,  
    year_result int CHECK (year_result BETWEEN 0 AND 20),  
    course_id varchar(6),  
    CONSTRAINT CK_birth_date CHECK (YEAR(birth_date) > 1970)  
)
```

# ALTER TABLE

**L'ordre DDL « ALTER TABLE »** permet de modifier la structure d'une table déjà existante. Certaines actions nécessiteront parfois de lourdes procédures, notamment lorsque l'on souhaite changer le type d'une colonne contenant déjà des données. *C'est pourquoi il est très important d'avoir réaliser une bonne analyse au préalable...*

## Ajouter/modifier une colonne

```
ALTER TABLE nom_table ADD nom_colonne TYPE contraintes  
ALTER TABLE nom_table ALTER COLUMN nom_colonne NOUVEAU_TYPE ...
```

```
ALTER TABLE student  
ADD year_result int CHECK (year_result BETWEEN 0 AND 20)
```

```
ALTER TABLE student ALTER COLUMN birth_date date
```

# ALTER TABLE

## Ajouter une contrainte

```
ALTER TABLE nom_table ADD CONSTRAINT nom_contrainte TYPE (colonne) ...
```

```
ALTER TABLE student  
ADD CONSTRAINT FK_student_section FOREIGN KEY (section_id)  
REFERENCES section ON DELETE SET NULL
```

## Désactiver/réactiver une contrainte

```
ALTER TABLE nom_table NOCHECK CONSTRAINT nom_contrainte  
ALTER TABLE nom_table CHECK CONSTRAINT nom_contrainte
```

```
ALTER TABLE student NOCHECK CONSTRAINT FK_student_section
```

# ALTER TABLE

## Supprimer une colonne

```
ALTER TABLE nom_table DROP COLUMN nom_colonne
```

```
ALTER TABLE student DROP COLUMN year_result
```

## Supprimer une contrainte

```
ALTER TABLE nom_table DROP CONSTRAINT nom_contrainte
```

```
ALTER TABLE student DROP CONSTRAINT PK_student
```

# TRUNCATE TABLE

L'ordre DDL « **TRUNCATE TABLE** » permet de vider une table de son contenu. Cet ordre permet par la même occasion de réinitialiser le compteur utilisé pour la colonne éventuellement auto-incrémentée

Cette opération est **plus rapide et efficace qu'un DELETE** s'il s'agit de supprimer l'ensemble des données d'une table car l'ordre DELETE va créer une entrée par ligne supprimée au niveau du journal de transactions

```
TRUNCATE TABLE nom_table
```

```
TRUNCATE TABLE student
```

# DROP TABLE

L'ordre DDL « **DROP TABLE** » permet de supprimer une table. Attention aux contraintes de clés étrangères...

```
DROP TABLE nom_table
```

```
DROP TABLE student
```

# Auto-Evaluation

Ce premier module contenait une série de notions importantes, autant théoriques que pratiques. Nous vous invitons (suite à la réalisation des exercices) à évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant ces notions

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit. Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Notions à évaluer (1/2)

Notions	P	S	V	I
Base de données ( <i>en théorie</i> )				
Système de gestion de bases de données ( <i>en théorie</i> )				
Cycle d'analyse d'un projet ( <i>en théorie</i> )				
Schéma Entité-Association ( <i>en théorie</i> )				
Transformation du schéma EA vers les schéma Relationnel				
Schéma Relationnel (utilité, création, lecture)				
Structure d'une table ( <i>en théorie</i> )				
Notion de « contrainte » de façon générale ( <i>en théorie</i> )				
Ordre « CREATE TABLE »				



# Auto-Evaluation

## Notions à évaluer (2/2)

Notions	P	S	V	I
Contrainte de « PRIMARY KEY »				
Contrainte de « FOREIGN KEY »				
Contraintes « UNIQUE », « NOT NULL » et « CHECK »				
Auto-incrémentation				
Valeur par défaut				
Ordre « ALTER TABLE »				
Ordre « DROP TABLE »				
Ordre « TRUNCATE TABLE »				

## Partie 3

# **DRL – DATA RETRIEVAL LANGUAGE**

La clause « SELECT »

Limiter et ordonner

Les fonctions

GROUP BY

Jointures

Sous-requêtes

# La clause « **SELECT** »

```
SELECT colonne1, colonne2, colonne3, ...  
FROM nom_table
```

- La casse n'a pas d'importance, mais on prendra l'habitude d'écrire les mots-clés du langage en majuscules
- On écrira généralement les clauses (« **SELECT** », « **FROM** », etc.) sur des lignes différentes afin d'indenter au mieux le code

# La clause « SELECT »

## Sélectionner toutes les colonnes et toutes les lignes

```
SELECT *  
FROM student
```

## Sélectionner toutes les lignes de certaines colonnes uniquement

```
SELECT first_name, last_name, year_result  
FROM student
```

# La clause « SELECT » : Alias

```
SELECT first_name as [Prénom]  
      , last_name 'Nom de famille'  
      , section_id Section  
      , year_result as "Résultat annuel"  
FROM student
```

- Les alias servent à renommer l'intitulé des colonnes à l'affichage des données. Cela ne change rien au niveau des données contenues dans les tables, bien entendu
- Le mot-clé « **as** » n'est pas obligatoire
- **Sous SQL-Server**, les alias doivent être accompagnés de guillemets simples, doubles ou de crochets s'ils contiennent des caractères spéciaux, des accents ou des espaces

Prénom	Nom de famille	Section	Résultat annuel
Georges	Lucas	1320	10
Clint	Eastwood	1010	4
Sean	Cannon	1020	12

# La clause « SELECT » : Alias

```
SELECT first_name AS Prénom  
       , last_name "Nom de famille"  
FROM student ;
```

**Sous Oracle**, les alias tenant en un seul mot ne doivent pas être délimités. Il sera par contre nécessaire d'utiliser les doubles-guillemets lorsqu'ils en contiennent plus d'un

PRÉNOM	Nom de famille
Georges	Lucas
Clint	Eastwood
Sean	Connery
Robert	De Niro
Kevin	Bacon
Kim	Basinger
Johnny	Depp
Julia	Roberts
Natalie	Portman

# La clause « SELECT » :

## Opérations Arithmétiques

```
SELECT first_name, year_result  
      , (year_result/20)*100 as [Nouveau Résultat]  
FROM student
```

### Opérateurs autorisés

Opérateur	Signification
/	Division
*	Multiplication
+	Addition / <b>Concaténation</b>
-	Soustraction

first_name	year_result	Nouveau Résultat
Georges	10	50
Clint	4	20
Sean	12	60
Robert	3	15
Kevin	16	80
Kim	19	95
Johnny	11	55
Julia	17	85
Natalie	4	20
Georges	4	20
André	10	50

# La clause « SELECT » : Concaténation

```
SELECT first_name + ' ' + last_name as [Nom complet]  
      , [login] + student_id as 'Code étudiant'  
FROM student
```

Nom complet	Code étudiant
Georges Lucas	glucas1
Clint Eastwood	ceastwoo2
Sean Connery	sconnery3
Robert De Niro	rde niro4
Kevin Bacon	kbacon5
Kim Basinger	kbasinge6
Johnny Depp	jdepp7
Julia Roberts	jroberts8
Natalie Portman	nportman9
Georges Clooney	gclooney10
Andy Garcia	agarcia11



# La clause « SELECT » : Concaténation

```
SELECT first_name || ' ' || last_name AS "Nom complet"  
      , login || student_id "Code étudiant"  
FROM student ;
```

*Sous Oracle*, la concaténation se fait grâce au symbole « || »

Nom complet	Code étudiant
Georges Lucas	glucas1
Clint Eastwood	ceastwoo2
Sean Connery	sconnery3
Robert De Niro	rde niro4
Kevin Bacon	kbacon5
Kim Basinger	kbasinge6
Johnny Depp	jdepp7
Julia Roberts	jroberts8
Natalie Portman	nportman9
Georges Clooney	gclooney10

# La clause « SELECT » : DISTINCT

```
SELECT DISTINCT first_name  
FROM student
```

Sans  
DISTINCT

first_name
Alyssa
Andy
Bruce
Clint
David
Georges
Georges
Halle
Jennifer
Johnny

Avec  
DISTINCT

first_name
Alyssa
Andy
Bruce
Clint
David
Georges
Halle
Jennifer
Johnny
Julia

# La clause « SELECT » : DISTINCT

```
SELECT DISTINCT first_name, year_result  
FROM student
```

Sans  
DISTINCT

first_name	year_result
Tom	4
Tom	4
Sophie	6
Shannen	2
Sean	12
Sarah	7
Sandra	2
Robert	3
Reese	7
Natalie	4

Avec  
DISTINCT

first_name	year_result
Tom	4
Sophie	6
Shannen	2
Sean	12
Sarah	7
Sandra	2
Robert	3
Reese	7
Natalie	4
Michael .I	3

# La clause « SELECT » : SANS FROM

```
SELECT col1 as alias_col1, col2, col3, ...
```

Sous SQL Server, la clause « **SELECT** » peut s'utiliser seule, si le but est simplement d'afficher un résultat structuré dans un tableau

```
SELECT GETDATE() as [Date du jour]  
      , 'Vive le SQL !'
```

Date du jour	(No column name)
2014-06-30 15:18:00.263	Vive le SQL !

# La clause « SELECT » : DUAL

```
SELECT col1 as alias_col1, col2, col3, ... FROM DUAL
```

Sous Oracle, la clause « **SELECT** » ne peut pas s'utiliser seule, mais il est également possible d'afficher un résultat divers structuré dans un tableau, grâce à la table temporaire « **DUAL** »

```
SELECT CURRENT_TIMESTAMP AS "Date du jour"  
      , 'Vive le SQL !'  
FROM DUAL ;
```

Date du jour	'VIVELESQL!'
30/06/14 19:15:06,195000000 EUROPE/BERLIN	Vive le SQL !

# Limiter et ordonner

```
SELECT colonne1, colonne2, colonne3, ...  
FROM nom_table  
WHERE conditions  
ORDER BY liste_colonnes
```

- La clause « **WHERE** » permet de limiter le nombre de lignes sélectionnées
- La clause « **ORDER BY** » permet de trier les résultats affichés, selon une ou plusieurs colonnes données
- Comme vu précédemment, ***ces clauses ne sont pas obligatoires*** mais si elles sont présentes, elles apparaissent dans cet ordre
- ***Il n'y a qu'une seule clause de chaque type.*** Il n'y aura donc jamais deux « **WHERE** » dans une même requête

# Limiter et ordonner : « **WHERE** »

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE year_result >= 16
```

## Opérateurs de comparaison

Opérateur	Signification
>	Plus grand
<	Plus petit
>=	Plus grand ou égal
<=	Plus petit ou égal
<>	Différent
!	Négation (« !> » = « pas plus grand »)

student_id	first_name	last_name	year_result
5	Kevin	Bacon	16
6	Kim	Basinger	19
8	Julia	Roberts	17
11	Andy	Garcia	19
18	Jennifer	Gamer	18
25	Halle	Berry	18

# Limiter et ordonner : **BETWEEN**

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE year_result BETWEEN 10 AND 16
```

Liste des étudiants ayant obtenu un résultat annuel compris entre 10 et 16, ces valeurs incluses

Cela revient à demander la liste des étudiants qui ont un résultat  
***plus grand ou égal à 10 ET plus petit ou égal à 16***

student_id	first_name	last_name	year_result
1	Georges	Lucas	10
3	Sean	Connery	12
5	Kevin	Bacon	16
7	Johnny	Depp	11
23	Keanu	Reeves	10



# Limiter et ordonner : BETWEEN

```
SELECT first_name, last_name, birth_date
FROM student
WHERE birth_date BETWEEN '1960-01-01' AND '1970-12-31'
```

**Les bornes de l'intervalle doivent être du même type que la valeur comparée.** Dans cet exemple, les chaînes de caractères '1960-01-01' et '1970-12-31' seront automatiquement converties en dates afin de pouvoir être comparées aux valeurs de la colonne « **birth\_date** »

first_name	last_name	birth_date
Johnny	Depp	1963-06-09 00:00:00.000
Julia	Roberts	1967-10-28 00:00:00.000
Georges	Clooney	1961-05-06 00:00:00.000
Tom	Cruise	1962-07-03 00:00:00.000
Sophie	Marceau	1966-11-17 00:00:00.000
Michael J.	Fox	1969-06-20 00:00:00.000
Sandra	Bullock	1964-07-26 00:00:00.000
Keanu	Reeves	1964-09-02 00:00:00.000
Halle	Berry	1966-08-14 00:00:00.000

# Limiter et ordonner : IN

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE year_result IN(12,16,18)
```

Liste des étudiants dont le résultat annuel est ***égal à 12*** OU ***égal à 16*** OU ***égal à 18***

student_id	first_name	last_name	year_result
3	Sean	Connery	12
5	Kevin	Bacon	16
18	Jennifer	Gamer	18
25	Halle	Berry	18

# Limiter et ordonner : IN

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE first_name IN('Tom', 'Jennifer', 'Halle')
```

L'opérateur « **IN** » permet de comparer tous types de données, tant que les valeurs entre parenthèses sont bien du même type que la valeur comparée. **La casse n'a PAS d'importance**

student_id	first_name	last_name	year_result
13	Tom	Cruise	4
18	Jennifer	Gamer	18
20	Tom	Hanks	8
25	Halle	Berry	18

# Limiter et ordonner : **LIKE**

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE first_name LIKE 'j%'
```

L'opérateur « **LIKE** » est utilisé pour comparer des chaînes de caractères entre elles

Le symbole « **%** » peut être utilisé pour remplacer **de 0 à N caractères**

Le symbole « **\_** » peut être utilisé pour remplacer **1 caractère**

student_id	first_name	last_name	year_result
7	Johnny	Depp	11
8	Julia	Roberts	17
18	Jennifer	Gamer	18

# Limiter et ordonner : LIKE

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE last_name LIKE '%oo_'
```

Liste des étudiants dont les trois dernières lettres du nom de famille sont « o », « o » et un caractère indéfini

student_id	first_name	last_name	year_result
2	Clint	Eastwood	4
14	Reese	Witherspoon	7

# Limiter et ordonner : **NOT**

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE year_result NOT BETWEEN 10 AND 15
```

L'opérateur « **NOT** » marque la négation des opérateurs « **BETWEEN** », « **IN** » et « **LIKE** »

student_id	first_name	last_name	year_result
2	Clint	Eastwood	4
4	Robert	De Niro	3
5	Kevin	Bacon	16
6	Kim	Basinger	19
8	Julia	Roberts	17
9	Natalie	Portman	4
10	Georges	Clooney	4
11	Andy	Garcia	19
12	Bruce	Willis	6
12	Tom	Cruise	4

# Limiter et ordonner : NOT

Liste des étudiants dont le nom de famille ne contient pas de « e »

```
SELECT student_id, first_name, last_name, year_result  
FROM student  
WHERE last_name NOT LIKE '%e%'
```

Liste des étudiants dont le résultat annuel est différent de 12, 16 ou 18

```
SELECT student_id, first_name, last_name, year_result  
FROM student  
WHERE year_result NOT IN (12,16,18)
```

# Limiter et ordonner : IS (NOT) NULL

```
SELECT student_id, first_name, last_name, year_result  
FROM student  
WHERE year_result IS NULL
```

Afin de déterminer si une valeur est « **NULL** » ou non, il faudra utiliser la syntaxe « **IS NULL** » dont la négation sera « **IS NOT NULL** »

student_id	first_name	last_name	year_result
5	Kevin	Bacon	NULL
7	Johnny	Depp	NULL
10	Georges	Clooney	NULL



# Limiter et ordonner : **AND**

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE first_name like 'J%'
      AND year_result >= 10
```

L'opérateur « **AND** » permet de combiner plusieurs conditions en même temps

***Une ligne doit répondre simultanément à toutes les conditions pour faire partie du résultat***

student_id	first_name	last_name	year_result
7	Johnny	Depp	11
8	Julia	Roberts	17
18	Jennifer	Gamer	18

# Limiter et ordonner : **OR**

```
SELECT student_id, first_name, last_name, year_result
FROM student
WHERE first_name like 'J%'
      OR year_result >= 10
```

Tout comme son compère « **AND** », l'opérateur « **OR** » permet également de combiner plusieurs conditions en même temps

*Il suffit qu'une ligne réponde à l'une des conditions pour faire partie du résultat*

student_id	first_name	last_name	year_result
1	Georges	Lucas	10
3	Sean	Connery	12
5	Kevin	Bacon	16
6	Kim	Basinger	19
7	Johnny	Depp	11
8	Julia	Roberts	17
11	Andy	Garcia	19

# Limiter et ordonner : Précédence

Il est conseillé d'**utiliser des parenthèses** afin de forcer le système à tester les conditions dans l'ordre souhaité. Sous SQL-Server, si aucune parenthèse n'est utilisée, l'ordre de précedence suivant est appliqué

Ordre	Opérateurs évalués
1	Multiplication, Division, Modulo
2	Addition, Soustraction, Concaténation
3	Opérateurs de comparaisons (=,>,<!=,...)
4	NOT
5	AND
6	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
7	Affectation (=)

<http://msdn.microsoft.com/fr-fr/library/ms190276.aspx>

# Limiter et ordonner : « ORDER BY »

```
SELECT student_id, first_name, last_name, year_result  
FROM student  
ORDER BY last_name ASC
```

La clause « **ORDER BY** » permet de trier le résultat d'une requête selon une ou plusieurs colonnes

Le tri peut se faire de façon croissante (« **ASC** » – ascendant) ou décroissante (« **DESC** » – descendant) sur les valeurs. La valeur par défaut est « **ASC** »

*Il est possible de trier selon une colonne qui n'est pas affichée*

student_id	first_name	last_name	year_result
5	Kevin	Bacon	16
6	Kim	Basinger	19
25	Halle	Berry	18
22	Sandra	Bullock	2
10	Georges	Clooney	4
3	Sean	Connery	12
12	Tom	Cruise	4

# Limiter et ordonner : « ORDER BY »

```
SELECT section_id  
       , first_name + ' ' + last_name as 'Nom complet'  
FROM student  
ORDER BY section_id, 'Nom complet' DESC
```

Il est possible de trier les résultats *en fonction d'un alias* de colonne ainsi que sur *une combinaison de plusieurs colonnes*

section_id	Nom complet
1010	Sandra Bullock
1010	Natalie Portman
1010	Clint Eastwood
1010	Bruce Willis
1020	Tom Hanks
1020	Tom Cruise
1020	Sean Connery
1020	Sarah Michelle Gellar

# Auto-Evaluation

N'oubliez pas de prendre le temps d'évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant les notions abordées dans ce module !

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit.  
Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Notions à évaluer

Notions	P	S	V	I
Ordre « SELECT ... FROM »				
Alias				
Concaténation et conversion de type de données (CONVERT)				
Clause « WHERE »				
Opérations arithmétiques				
Opérateurs « BETWEEN », « IN », « LIKE » et leur négation				
Comparaison avec une valeur « NULL »				
Opérateurs « AND » et « OR »				
Clause « ORDER BY »				

# Les fonctions

**Une fonction** est un ensemble de lignes de code stocké dans le système, qui exécute à la demande, une tâche pour l'utilisateur et renvoie un résultat. Une fonction demande souvent que **des paramètres soient fournis** en entrée.

**Le résultat renvoyé** doit ensuite être affiché ou inclus dans une expression ou une requête. Une fonction peut bien sûr utiliser le résultat d'une autre fonction

- Le nom de la fonction est ***toujours suivi de parenthèses***, même si aucun paramètre n'est fourni ou attendu
- Les fonctions renvoient des valeurs de types divers. Le tableau suivant classe les fonctions présentées dans cette formation, selon le type de valeur qu'elles renvoient

Type retourné	Noms des fonctions
numérique	datepart, charindex, len, abs, modulo
chaîne de caractères	substring, upper, lower, replace, trim
datetime	getdate()



# Les fonctions : **CONVERT**

**CONVERT** (NOUVEAU\_TYPE, valeur\_à\_convertir)

**CONVERT** (TYPE\_CHAÎNE\_DE\_CARACTÈRES, date\_à\_convertir, format\_date)

La fonction « **CONVERT** » attend 2 paramètres en entrée (éventuellement 3 lorsque l'élément à convertir est une date) et *renvoie une valeur correspondant au deuxième paramètre dans le type spécifié par le premier* (et dans le format précisé par le troisième, le cas échéant [100-114])

```
SELECT CONVERT (varchar, birth_date, 110)
           as [Date de naissance]
FROM student
```

Date de naissance
05-17-1944
05-31-1930
08-25-1930
05-17-1944

# Les fonctions : **GETDATE()**

```
SELECT  GETDATE ()  
        , CONVERT (varchar, GETDATE (), 109)  
        , CONVERT (date, GETDATE ())  
        , CONVERT (time, GETDATE ())
```

*Sous SQL-Server*, la fonction « **GETDATE()** » renvoie la date et l'heure actuelles

Type retourné : DATETIME

Date du jour	Date du jour formatée	Date uniquement	Heure uniquement
2014-05-14 14:21:09.210	May 14 2014 2:21:09:210PM	2014-05-14	14:21:09.2130000

# Les fonctions : **DATEPART**

**DATEPART** (*partie\_de\_date\_à\_extraire, date\_traitée*)

La fonction « **DATEPART** » extrait une partie d'une date donnée

Type retourné : NOMBRE

```
SELECT DATEPART (mm, GETDATE ())  
      , DATEPART (dy, GETDATE ())  
      , DATEPART (ns, GETDATE ())
```

Mois	Jour de l'année	Nanosecondes
5	134	153000000

<http://msdn.microsoft.com/fr-be/library/ms174420.aspx>

# Les fonctions : **CHARINDEX**

**CHARINDEX** (*chaine\_de\_caractères\_recherchée, valeur\_à\_évaluer*)

La fonction « **CHARINDEX** » renvoie la position du début de l'occurrence d'une chaîne de caractère dans une autre

Type retourné : NOMBRE

```
SELECT CHARINDEX('i', 'Kim Basinger')
       , CHARINDEX('08', 'Basinger 08/12/1953')
       , CHARINDEX('y', 'Kim Basinger')
       , CHARINDEX('', 'Kim Basinger')
```

position du premier i	position du 08	pas de y	chaîne vide
2	10	0	0

# Les fonctions : **LEN**

**LEN** (*chaine\_de\_caractères\_à\_mesurer*)

La fonction « **LEN** » renvoie le nombre de lettres composant une chaine de caractères donnée, espaces blancs compris

Type retourné : NOMBRE

```
SELECT LEN('Kim Basinger')
```

Longueur de la chaine de caractères

12

# Les fonctions : **ABS**

**ABS** (*nombre*)

La fonction « **ABS** » renvoie la valeur absolue du nombre passé en paramètre

Type retourné : NOMBRE

```
SELECT ABS (-1.0) , ABS (0.0) , ABS (1.0)  
SELECT ABS (-2147483648)
```

val1	val2	val3
1.0	0.0	1.0

```
Msg 8115, Level 16, State 2, Line 1  
Arithmetic overflow error converting expression to data type int.
```

# Les fonctions : Modulo

*dividende % diviseur*

Le « % », qui représente la fonction « **modulo** » que l'on rencontre fréquemment dans d'autres langages également, **renvoie le reste de la division ENTIÈRE du premier nombre (dividende) par le second (diviseur)**. Cette fonction permet de savoir si le premier chiffre est multiple du second

Type retourné : NOMBRE

```
SELECT 38 / 5, 38 % 5
```

$$\begin{array}{r} 38,0 \\ 35 \\ \hline 30 \\ 30 \\ \hline 0 \end{array} \quad \begin{array}{r} 5 \\ 7,6 \end{array}$$

Division entière	Reste
7	3

# Les fonctions : **SUBSTRING**

**SUBSTRING** (*chaine\_de\_caractères, position\_départ, nombre\_caractères*)

La fonction « **SUBSTRING** » renvoie une chaîne de caractère d'une longueur souhaitée, à partir d'une position donnée, à l'intérieur d'une chaîne de caractères passée en paramètre

Type retourné : CHAÎNE DE CARACTÈRES

```
SELECT SUBSTRING ('Basinger', 4, 3)
SELECT last_name, SUBSTRING(first_name, 1, 1)
FROM student
```

Caractères 4, 5 et 6  
ing

last_name	Initiale du prenom
Lucas	G
Eastwood	C
Connery	S
De Niro	R
Bacon	K



# Les fonctions : **UPPER** et **LOWER**

**UPPER** (*chaîne\_de\_caractères*)  
**LOWER** (*chaîne\_de\_caractères*)

Les fonctions « **UPPER** » et « **LOWER** » renvoient la chaîne de caractères passée en paramètres, respectivement en majuscules ou en minuscules

Type retourné : CHAÎNE DE CARACTÈRES

```
SELECT UPPER(last_name), LOWER(first_name)
FROM student
WHERE UPPER(first_name) LIKE 'TOM'
```

Nom de famille	Prénom
CRUISE	tom
HANKS	tom

# Les fonctions : **REPLACE**

**REPLACE** (*chaine\_de\_caractères\_traitée, caract\_à\_remplacer, nouveau\_caract*)

La fonction « **REPLACE** » remplace les caractères demandés par d'autres

Type retourné : CHAÎNE DE CARACTÈRES

```
SELECT REPLACE(' Kim Basinger ', ' ', '')  
      , REPLACE('11110000101010', '1', '0')
```

Sans espaces	Sans 1
KimBasinger	00000000000000

# Les fonctions : **LTRIM** et **RTRIM**

**LTRIM** (*chaine\_de\_caractères*)

**RTRIM** (*chaine\_de\_caractères*)

Les fonctions « **LTRIM** » et « **RTRIM** » renvoient la chaîne de caractères passée en paramètres épurée des espaces blancs éventuellement présents en début ou en fin de chaîne, respectivement

Type retourné : CHAÎNE DE CARACTÈRES

```
SELECT LTRIM('    Kim Basinger    ')
       , RTRIM('    Kim Basinger    ')
       , LTRIM(RTRIM('    Kim Basinger    '))
```

LTRIM	RTRIM	LTRIM de RTRIM
Kim Basinger	Kim Basinger	Kim Basinger

# Les fonctions : Agrégations

**Une fonction d'agrégation** est une fonction particulière qui attend comme paramètres **un ensemble de valeurs** (une colonne) et qui présente en retour **un seul** résultat agrégé (regroupé) sur ces valeurs. Sauf exceptions, les valeurs **NULL** ne sont pas prises en compte

## Fonctions d'agrégation principales

Fonction	Description
<b>COUNT</b>	Nombre total de valeurs contenues dans la table/colonne
<b>MAX</b>	Valeur numérique la plus élevée
<b>MIN</b>	Plus petite valeur numérique disponible
<b>SUM</b>	Somme de l'ensemble des valeurs de la colonne
<b>AVG</b>	Moyenne de l'ensemble des valeurs de la colonne

# Les fonctions : **COUNT**

**COUNT (\*)**

**COUNT (colonne)**

**COUNT (DISTINCT colonne)**

La fonction d'agrégation « **COUNT** » renvoie le nombre total de valeur contenues dans la table ou la colonne à laquelle on applique la fonction. Les valeurs « **NULL** » ne sont prises en compte que dans l'utilisation du « **COUNT(\*)** »

Type retourné : NOMBRE (INTEGER)

```
SELECT COUNT(*), COUNT(first_name), COUNT(DISTINCT first_name)
FROM student
```

Total des lignes	Total des prénoms	Total des prénoms sans doublons
25	25	23

# Les fonctions : **MAX** et **MIN**

**MAX** (*colonne*)

**MIN** (*colonne*)

Les fonctions d'agrégation « **MAX** » et « **MIN** » renvoient respectivement la plus grande ou la plus petite des valeurs contenues dans une colonne donnée

Type retourné : NOMBRE

```
SELECT MAX (year_result), MIN (year_result*5)  
      , MAX (LEN(last_name))  
FROM student
```

Résultat le plus élevé	Pourcentage le plus faible	Taille du nom le plus long
19	10	15

# Les fonctions : **SUM**

**SUM** (*colonne*)

La fonction « **SUM** » renvoie la somme des valeurs d'une colonne

Type retourné : NOMBRE

```
SELECT SUM (year_result), SUM (year_result) / COUNT (*)  
FROM student
```

Somme des résultats annuels	Moyenne générale
219	8

# Les fonctions : **AVG**

## **AVG** (colonne)

La fonction « **AVG** » renvoie la moyenne de l'ensemble des valeurs contenues dans une colonne

Type retourné : NOMBRE

```
SELECT AVG (year_result)
        , AVG (DATEPART(yy,GETDATE()) - DATEPART(yy,birth_date))
FROM student
```

Moyenne générale	Moyenne d'âge
8	53



# Les fonctions : **CASE**

```
CASE  
  WHEN expression1 THEN valeur1  
  WHEN expression2 THEN valeur2  
  ...  
  WHEN expressionN THEN valeurN  
  ELSE valeur_par_défaut  
END
```

- **L'instruction « CASE »** peut être utilisée afin de modifier l'affichage des éléments d'une colonne selon ce que l'on souhaite
- Dès qu'une expression contenue dans l'une des clauses « **WHEN** » est évaluée à « **TRUE** », la valeur contenue après la clause « **THEN** » est affichée dans la colonne et **l'instruction « CASE » se termine** et est réévaluée en totalité pour la ligne suivante
- Si aucune des expressions évaluées après les clauses « **WHEN** » n'est validée, la valeur affichée dans la colonne correspond à la valeur présentée dans la clause « **ELSE** »

# Les fonctions : CASE

```
SELECT last_name, first_name, year_result,  
CASE  
    WHEN year_result BETWEEN 18 AND 20 THEN 'Excellent'  
    WHEN year_result BETWEEN 16 AND 17 THEN 'Très Bien'  
    WHEN year_result BETWEEN 14 AND 15 THEN 'Bien'  
    WHEN year_result BETWEEN 12 AND 13 THEN 'Suffisant'  
    WHEN year_result BETWEEN 10 AND 11 THEN 'Faible'  
    WHEN year_result BETWEEN 8 AND 9 THEN 'Insuffisant'  
    ELSE 'Insuffisance Grave'  
END AS [Note globale]  
FROM student
```

last_name	first_name	year_result	Note globale
Lucas	Georges	10	Faible
Eastwood	Clint	4	Insuffisance Grave
Connery	Sean	12	Suffisant
De Niro	Robert	3	Insuffisance Grave
Bacon	Kevin	16	Très Bien
Basinger	Kim	19	Excellent
Denn	Johnny	11	Faible

# Les fonctions : CASE

```
CASE colonne_à_évaluer  
  WHEN valeur_de_comparaison1 THEN valeur1  
  ...  
  ELSE valeur_par_défaut  
END
```

Lorsque les expressions à évaluer sont des **égalités strictes**, il est possible de simplifier l'écriture du « **CASE** » en reprenant le nom de la colonne à évaluer directement après le mot-clé « **CASE** »

```
SELECT student_id, first_name,  
CASE section_id  
  WHEN 1010 THEN 'BSc Management'  
  WHEN 1320 THEN 'MA Sociology'  
  ELSE NULL  
END AS [Nom de section section]  
FROM student
```

last_name	first_name	Nom de section section
Lucas	Georges	MA Sociology
Eastwood	Clint	BSc Management
De Niro	Robert	NULL
Depp	Johnny	NULL
Portman	Natalie	BSc Management
Garcia	Andy	NULL
Willie	Bruce	BSc Management

# Les fonctions : **NULLIF**

**NULLIF** (*colonne\_considerée*, *valeur\_à\_mettre\_à\_NULL*)

La fonction « **NULLIF** » est un cas particulier du « **CASE** » qui renvoie les mêmes valeurs que la colonne passée en paramètre, sauf pour les valeurs équivalentes au deuxième paramètre fourni, pour lesquelles la valeur **NULL** sera affichée

```
CASE colonne_considerée  
  WHEN valeur_à_mettre_à_NULL THEN NULL  
  ELSE valeur_colonne_considerée  
END
```

# Les fonctions : NULLIF

```
SELECT last_name, first_name, year_result  
      , NULLIF (year_result, 7)  
FROM student
```

```
SELECT last_name, first_name, year_result  
      , CASE year_result  
          WHEN 7 THEN NULL  
          ELSE year_result  
        END AS [Résultats sauf les 4/20]  
FROM student
```

last_name	first_name	year_result	Résultats sauf les 7/20
Clooney	Georges	4	4
Garcia	Andy	19	19
Willis	Bruce	6	6
Cruise	Tom	4	4
Witherspoon	Reese	7	NULL
Marceau	Sophie	6	6
Michelle Gellar	Sarah	7	NULL
Milano	Alyssa	7	NULL
Gamer	Jennifer	19	19

# Les fonctions : **COALESCE**

**COALESCE** (*colonne1, colonne2, ..., colonneN*)

La fonction « **COALESCE** » est un autre cas particulier du « **CASE** » qui renvoie la première valeur non NULL rencontrée parmi les différentes colonnes fournies en paramètres

**CASE**

**WHEN** *colonne1 IS NOT NULL THEN colonne1*

**WHEN** *colonne2 IS NOT NULL THEN colonne2*

...

**WHEN** *colonneN-1 IS NOT NULL THEN colonneN-1*

**ELSE** *colonneN*

**END**

# Les fonctions : COALESCE

Table  
« WAGES »

emp_id	hourly_wage	salary	commission	num_sales
1	10	NULL	NULL	NULL
2	20	NULL	NULL	NULL
3	30	NULL	NULL	NULL
4	40	NULL	NULL	NULL
5	NULL	10000	NULL	NULL
6	NULL	20000	NULL	NULL
7	NULL	30000	NULL	NULL
8	NULL	40000	NULL	NULL
9	NULL	NULL	15000	3
10	NULL	NULL	25000	2
11	NULL	NULL	20000	6
12	NULL	NULL	14000	4

Résultat  
du COALESCE

Total Salary
10000,00
20000,00
20800,00
30000,00
40000,00
41600,00
45000,00
50000,00
56000,00
62400,00
83200,00
120000,00

```
SELECT CAST(COALESCE(hourly_wage * 40 * 52
                     , salary
                     , commission * num_sales)
           AS money) AS 'Total Salary'
FROM wages
ORDER BY 'Total Salary'
```

# Les fonctions : Imbrication

Table  
« BUDGETS »

dept	current_year	previous_year
1	100000	150000
2	NULL	300000
3	0	100000
4	NULL	150000
5	300000	250000

*NULL = même budget que l'année précédente*

*0 = budget non défini (valeurs à ne pas considérer dans la moyenne)*

*« current\_year » et « previous\_year » sont de type DECIMAL*



```
SELECT AVG(NULLIF(COALESCE(current_year, previous_year), 0.00)) AS 'Average Budget'
FROM budgets
```

Average Budget
212500.000000



# Les fonctions : Microsoft vs Oracle

T-SQL	Oracle	Utilité
AVG()	AVG()	Faire une moyenne
COUNT()	COUNT()	Compte le nombre d'enregistrement
MAX() & MIN()	MAX() & MIN()	Plus grande/petite valeur
SUM()	SUM()	Faire la somme
CAST() & CONVERT()	CAST(<donnée> AS <type>)	Conversion de données
DATEADD() & DATEDIFF()	+ & -	Additionner/soustraire des dates
COALESCE()	COALESCE()	Retourne la valeur reprise si elle est NON NULL
GETDATE()	CURRENT_DATE	Retourne la date et heure actuelle
DATEPART(<extraction> , <date>)	EXTRACT (<extraction> FROM <date>)	Retourne un entier Datepart, ex: yy,yyyy, mm...

# Les fonctions : Microsoft vs Oracle

T-SQL	Oracle	Utilité
DAY(), MONTH(), YEAR()		Retourne le jour, le mois, l'année
NULLIF()	NULLIF()	Retourne NULL si <nom_colonne> est égale à <valeur_à_éliminer>
ABS()	ABS()	Valeur absolue
<dividende> % <diviseur>	MOD(<dividende> , <diviseur>)	Modulo
RAND()	dbms_random.random	Génère un nombre aléatoire
LEN()	LENGTH()	Nombre de caractères
LOWER() & UPPER()	LOWER() & UPPER()	Mettre en minuscule/majuscule
LTRIM(RTRIM())	TRIM	Supprime les espaces à gauche et à droite
CHARINDEX()	INSTR()	Retourne la position d'une chaîne de caractères dans une autre

# Les fonctions : Microsoft vs Oracle

T-SQL	Oracle	Utilité
SUBSTRING()	SUBSTR ()	Extraire une chaîne
+ ou CONCAT(exp1, exp2)		Concaténation

# Auto-Evaluation

N'oubliez pas de prendre le temps d'évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant les notions abordées dans ce module !

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

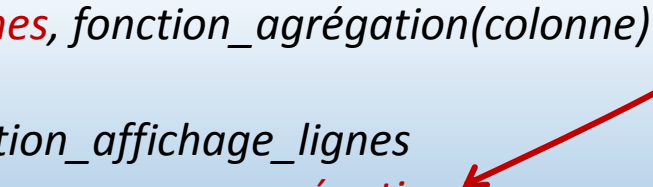
- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit.  
Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Notions à évaluer

Notions	P	S	V	I
Fonction (fonctionnement interne, utilité, mise en pratique)				
Imbrication de fonctions				
Fonctions d'agrégation				
Expression « CASE »				
Fonctions « NULLIF » et « COALESCE »				

# GROUP BY



```
SELECT colonnes, fonction_agrégation(colonne)
FROM table
WHERE condition_affichage_lignes
GROUP BY sous_groupes_agrégation
HAVING condition_affichage_groupes
ORDER BY ordre_tri_affichage
```

COUNT
MAX
MIN
SUM
AVG

- La clause « **GROUP BY** » permet de créer des sous-regroupements de lignes au niveau de la table, afin de leur appliquer une même fonction d'agrégation
- La clause « **HAVING** » ne peut être présente que si la clause « **GROUP BY** » est présente également. Le « **HAVING** » pose une condition d'affichage sur les groupes créés par la clause « **GROUP BY** ». Cette condition doit porter sur une fonction d'agrégation également
- Première règle d'or  
*Dès que la clause « **SELECT** » combine l'affichage d'une ou plusieurs fonctions d'agrégation **ET** des colonnes non-agrégées, la clause « **GROUP BY** » est obligatoire*
- Seconde règle d'or  
*Toutes les colonnes non-agrégées présentes dans la clause « **SELECT** » doivent impérativement se retrouver dans la clause « **GROUP BY** »*

# GROUP BY

```
SELECT section_id, AVG(year_result)
FROM student
GROUP BY section_id
```

section_id	Moyenne par section
1010	4
1020	7
1110	8
1120	17
1310	11
1320	10

Sans le « **GROUP BY** », le système produit l'erreur suivante :

Column 'student.section\_id' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

# GROUP BY : + WHERE

```
SELECT section_id, AVG(year_result)
FROM student
WHERE LEFT(last_name,1) IN ('B', 'C', 'D')
GROUP BY section_id
```

section_id	last_name	year_result
1020	Connery	12
1110	De Niro	3
1120	Bacon	16
1310	Basinger	19
1110	Depp	11
1020	Clooney	4
1020	Cruise	4
1010	Bullock	2
1320	Doherty	2
1320	Berry	18

Moyenne = 6,67

section_id	Moyenne par section
1010	2
1020	6
1110	7
1120	16
1310	19
1320	10

*Solution de la requête*

Ensemble de lignes triées grâce à la clause « **WHERE** »  
et sur lequel la clause « **GROUP BY** » sera appliquée



# GROUP BY : + WHERE + HAVING

```
SELECT section_id, AVG(year_result)
FROM student
WHERE LEFT(last_name,1) IN ('B', 'C', 'D')
GROUP BY section_id
HAVING AVG(year_result) >= 10
```

section_id	last_name	year_result
1020	Connery	12
1110	De Niro	3
1120	Bacon	16
1310	Basinger	19
1110	Depp	11
1020	Clooney	4
1020	Cruise	4
1010	Bullock	2
1320	Doherty	2
1320	Berry	18

« WHERE » uniquement

section_id	Moyenne par section
1010	2
1020	6
1110	7
1120	16
1310	19
1320	10

WHERE + GROUP BY

section_id	Moyennes > 10
1120	16
1310	19
1320	10

WHERE + GROUP BY  
+ HAVING

# GROUP BY : colonnes multiples

```
SELECT section_id, course_id, AVG(year_result)
FROM student
WHERE section_id IN (1010, 1020)
GROUP BY section_id, course_id
HAVING SUM(year_result) >= 2
ORDER BY section_id
```

- *Toutes les colonnes non-agrégées présentes dans la clause « **SELECT** » doivent impérativement se retrouver dans la clause « **GROUP BY** »*
- La condition du « **HAVING** », portant sur l'affichage des groupes créés par la clause « **GROUP BY** », peut utiliser d'autres fonctions d'agrégation et d'autres colonnes que celles utilisées dans la clause « **SELECT** »

section_id	course_id	Moyenne
1010	EG1020	2
1010	EG2210	4
1020	EG1020	7
1020	EG2110	7
1020	EG2210	10

# GROUP BY : ROLLUP et CUBE

```
SELECT colonnes, fonction_agrégation(colonne)
FROM table
GROUP BY ROLLUP (sous_groupes_agrégation)
```

```
SELECT colonnes, fonction_agrégation(colonne)
FROM table
GROUP BY CUBE (sous_groupes_agrégation)
```

- Les mots-clés « **ROLLUP** » ou « **CUBE** » peuvent être rajoutés à la clause « **GROUP BY** » de façon à afficher des sous-totaux
- « **ROLLUP** » applique un sous-total en remontant dans les colonnes indiquées, présentant un sous-total à partir de la colonne la plus détaillée, en remontant vers la colonne présentant des résultats groupés de façon plus vaste (sous-total par section et global)
- « **CUBE** » permet d'appliquer la fonction d'agrégation sur tout regroupement possible au niveau des données agrégées (sous-total par section, global et par cours, sans tenir compte des sections). Le « **CUBE** » englobe le « **ROLLUP** »

# GROUP BY : ROLLUP

```
SELECT section_id, course_id, AVG(year_result)
FROM student
WHERE section_id IN (1010, 1020)
GROUP BY ROLLUP (section_id, course_id)
```

section_id	course_id	Moyenne
1010	EG1020	2
1010	EG2210	4
1020	EG1020	7
1020	EG2110	7
1020	EG2210	10

*Sans « ROLLUP »*

section_id	course_id	Moyenne
1010	EG1020	2
1010	EG2210	4
1010	NULL	4
1020	EG1020	7
1020	EG2110	7
1020	EG2210	10
1020	NULL	7
NULL	NULL	6

*Total section 1010*

*Total section 1020*

*Total général*

*Avec « ROLLUP »*

# GROUP BY : CUBE

```
SELECT section_id, course_id, SUM(year_result)
FROM student
WHERE section_id IN (1010, 1020)
GROUP BY CUBE (section_id, course_id)
```

section_id	course_id	Somme
1010	EG1020	2
1020	EG1020	7
1020	EG2110	35
1010	EG2210	14
1020	EG2210	10

Sans « CUBE »

section_id	course_id	Somme
1020	EG2210	10
1020	EG2110	35
1020	EG1020	7
1020	NULL	52
1010	EG2210	14
1010	EG1020	2
1010	NULL	16
NULL	EG2210	24
NULL	EG2110	35
NULL	EG1020	9
NULL	NULL	68

Total section 1020

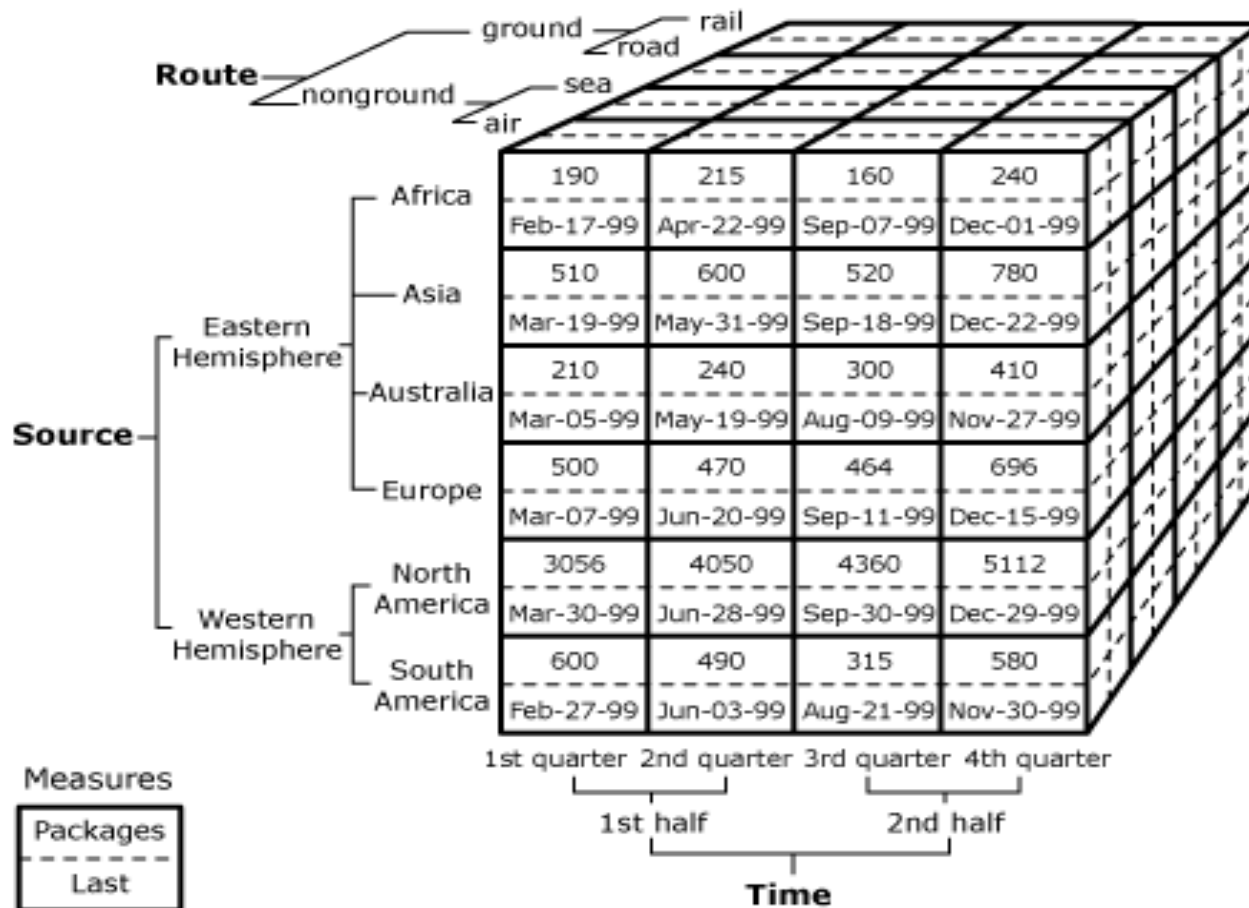
Total section 1010

Total par cours

Total général

Avec « CUBE »

# GROUP BY : CUBE OLAP



# Auto-Evaluation

N'oubliez pas de prendre le temps d'évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant les notions abordées dans ce module !

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit.  
Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Notions à évaluer

Notions	P	S	V	I
Clause « GROUP BY ... HAVING » et règles d'or				
Différence entre les clauses « WHERE » et « HAVING »				
« GROUP BY » sur plusieurs colonnes				
Clause « ROLLUP »				
Clause « CUBE »				



# Jointures

## Jointures horizontales

```
SELECT table1.col1, table1.col2, table2.col1, table2.col2  
FROM table1 JOIN table2 ON table1.col1 = table2.col2  
WHERE ... GROUP BY ... ORDER BY ...
```

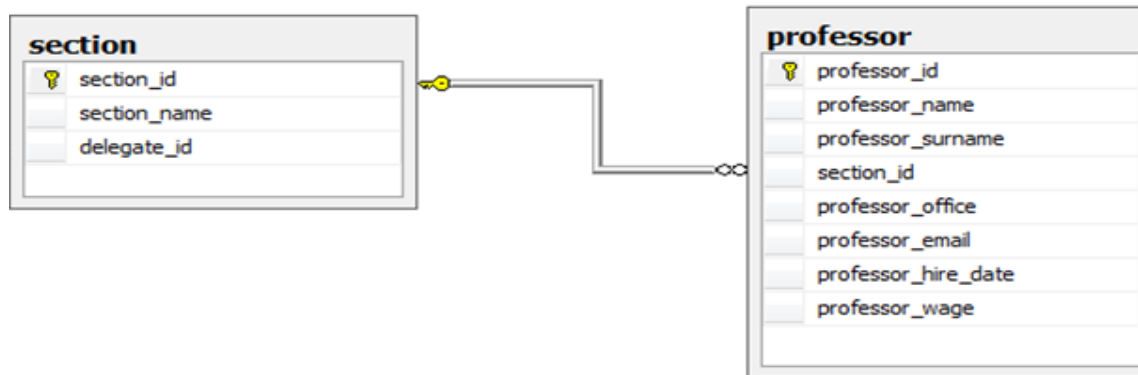
Comparaison des colonnes des tables entre elles

## Jointures verticales

```
SELECT ... FROM ... WHERE ... GROUP BY ...  
opérateur_comparaison_requêtes  
SELECT ... FROM ... WHERE ... GROUP BY ...
```

Comparaison du résultat de deux requêtes entre eux

# Jointures horizontales

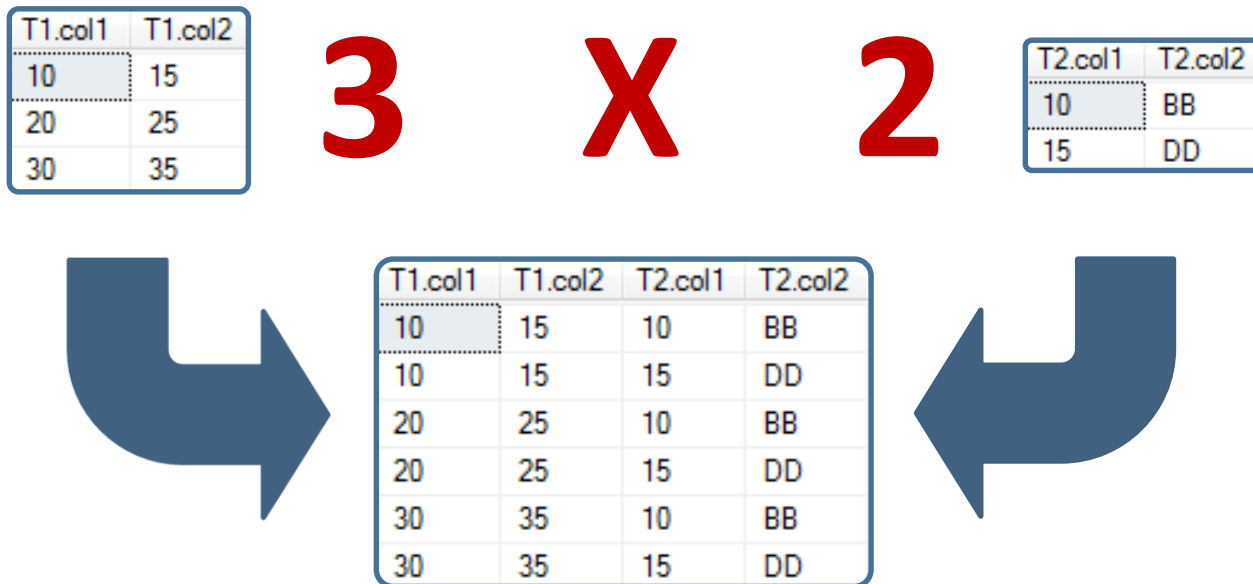


- La jointure horizontale compare deux colonnes entre elles et affiche les colonnes souhaitées pour chaque concordance trouvée
- La condition de la jointure (c'est-à-dire la comparaison à faire) utilisera souvent les **clés primaires et étrangères** liant les tables (mais ce n'est pas obligatoire)
- Si les colonnes utilisées dans la requête ont le même nom dans plus d'une table participant à la jointure, il faudra faire précéder ces colonnes du nom de la table. Nous prendrons donc l'habitude de **donner un alias aux tables** et de faire précéder chaque colonne d'un alias de table créé
- Lorsqu'une table a reçu un alias, il n'est plus possible d'utiliser le nom de la table dans la requête car le système travail désormais avec une copie de la table d'origine, portant l'alias comme nom

# Jointures : CROSS JOIN

```
SELECT T1.col1, T1.col2, T2.col1, T2.col2  
FROM table1 T1 CROSS JOIN table2 T2
```

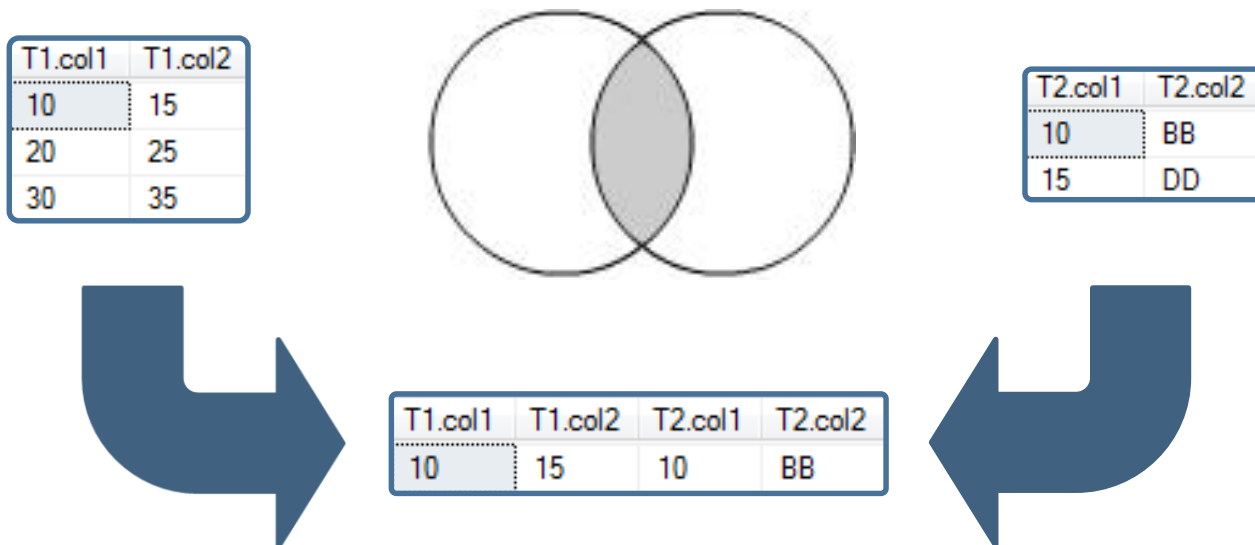
Le « **CROSS JOIN** » effectue simplement un produit cartésien des lignes de chacune des tables



# Jointures : **INNER JOIN**

```
SELECT *  
FROM table1 T1 JOIN table2 T2 ON T1.col1 = T2.col1
```

Le « **INNER JOIN** » compare les éléments des colonnes indiquées après le « **ON** » et affiche les informations demandées à chaque correspondance (mot-clé « **INNER** » facultatif *sous SQL-Server*)



# Jointures : INNER JOIN

```
SELECT S.section_id, S.section_name, P.professor_name
FROM section S JOIN professor P
ON S.section_id = P.section_id
```

Table  
« SECTION »

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	BSc Economics	15
1120	MSc Economics	6
1310	BA Sociology	23
1320	MA Sociology	8

Table  
« PROFESSOR »

professor_id	professor_name	section_id
1	zidda	1020
2	decrop	1120
3	giot	1310
4	lecourt	1310
5	scheppens	1020
6	louveaux	1110

Aucun professeur n'appartient aux sections 1010 et 1320  
2 professeurs font partie des sections 1020 et 1310

section_id	section_name	professor_name
1020	MSc Management	zidda
1120	MSc Economics	decrop
1310	BA Sociology	giot
1310	BA Sociology	lecourt
1020	MSc Management	scheppens
1110	BSc Economics	louveaux

Résultat de la jointure

# Jointures : INNER JOIN

```
SELECT S.section_id, S.section_name, P.professor_name  
FROM section S, professor P  
WHERE S.section_id = P.section_id
```



```
SELECT S.section_id, S.section_name, P.professor_name  
FROM section S JOIN professor P  
ON S.section_id = P.section_id
```

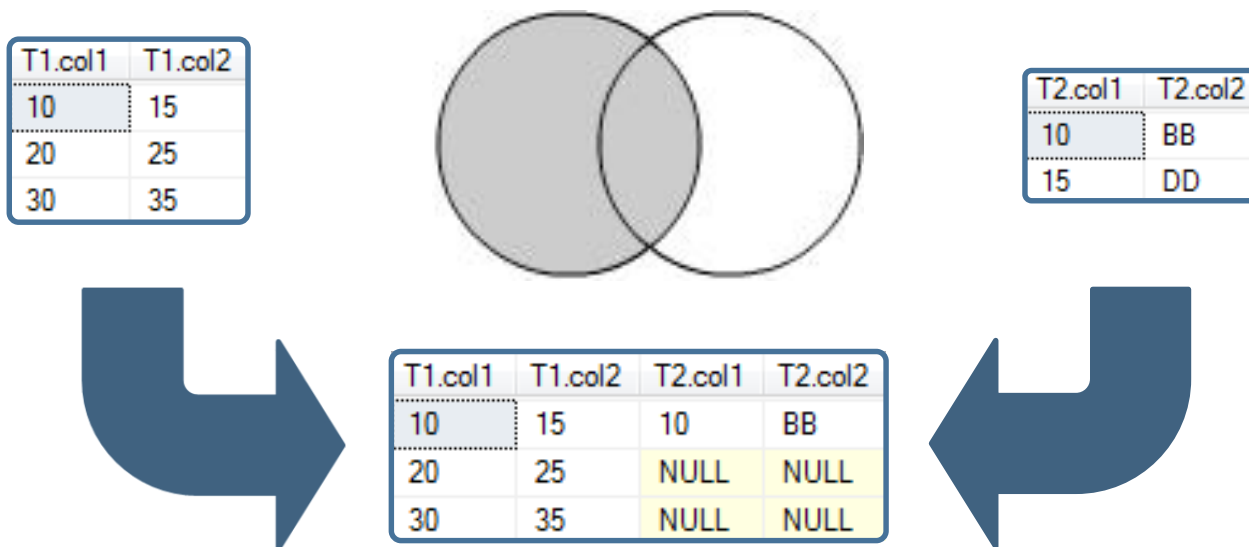
Sans faire précéder la colonne « ***section\_id*** » de l'alias de l'une ou l'autre table, le système produit l'erreur suivante :

Ambiguous column name 'section\_id'.

# Jointures : **LEFT OUTER JOIN**

```
SELECT *  
FROM table1 T1 LEFT JOIN table2 T2 ON T1.col1 = T2.col1
```

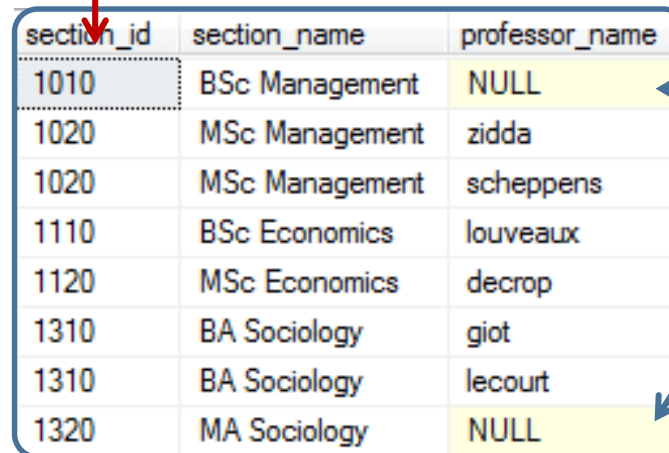
Le « **LEFT OUTER JOIN** » affiche les informations demandées à chaque correspondance, mais affiche aussi toutes les lignes de la première table, même si elles n'ont pas de correspondance dans la seconde (mot-clé « **OUTER** » facultatif *sous SQL-Server*)



# Jointures : LEFT OUTER JOIN

```
SELECT S.section_id, S.section_name, P.professor_name
FROM section S LEFT JOIN professor P
ON S.section_id = P.section_id
```

Aucun professeur n'appartient aux sections 1010 et 1320  
2 professeurs font partie des sections 1020 et 1310



section_id	section_name	professor_name
1010	BSc Management	NULL
1020	MSc Management	zidda
1020	MSc Management	scheppens
1110	BSc Economics	louveaux
1120	MSc Economics	decrop
1310	BA Sociology	giot
1310	BA Sociology	lecourt
1320	MA Sociology	NULL

On désire afficher les informations sur toutes les sections, qu'il y ai un professeur qui y soit inscrit ou non

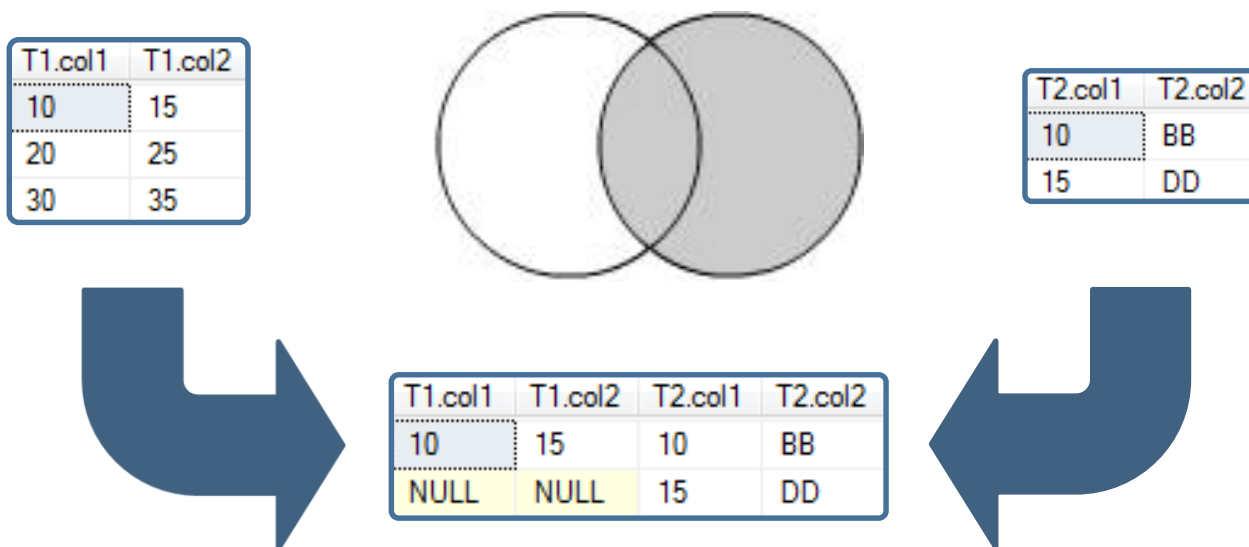
*Liste de toutes les sections avec les professeurs qui y sont inscrits, s'il y en a*



# Jointures : **RIGHT OUTER JOIN**

```
SELECT *  
FROM table1 T1 RIGHT JOIN table2 T2 ON T1.col1 = T2.col1
```

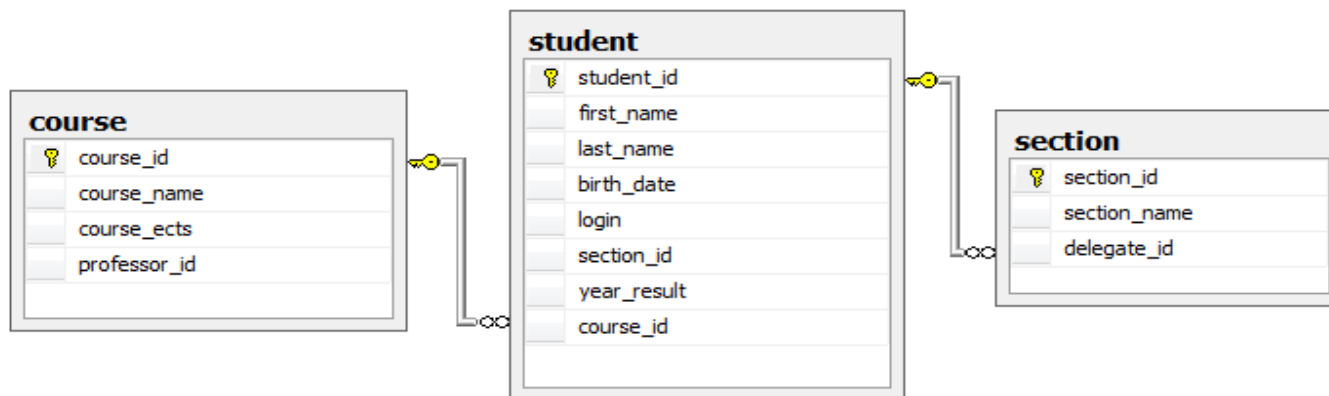
Le « **RIGHT OUTER JOIN** » fonctionne de la même manière que le **LEFT**, mais concerne la seconde table de la jointure (mot-clé « **OUTER** » facultatif *sous SQL-Server*)



# Jointures : RIGHT OUTER JOIN

```
SELECT first_name + ' ' + last_name  
      , section_name, course_name  
FROM   course C RIGHT JOIN student St  
      ON St.course_id = C.course_id  
   LEFT JOIN section S  
      ON St.student_id = S.delegate_id
```

Liste des étudiants, la section dont ils sont *éventuellement* délégués ainsi que le cours auquel ils sont *éventuellement* inscrits



# Jointures : RIGHT OUTER JOIN

course_id	course_name	course_ects	professor_id
EG1020	Derivatives	3.0	3
EG2110	Marketing management	3.5	2
EG2210	Financial Management	4.0	3
EING2283	Marketing engineering	4.0	1
EING2383	Supply chain management et e-business	2.5	5

student_id	first_name	last_name	course_id
1	Georges	Lucas	EG2210
2	Clint	Eastwood	EG2210
3	Sean	Connery	EG2110
4	Robert	De Niro	EG2210
5	Kevin	Bacon	0
6	Kim	Basinger	0
7	Johnny	Depp	EG2210

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	BSc Economics	15
1120	MSc Economics	6
1310	BA Sociology	23
1320	MA Sociology	6

```

SELECT first_name + ' ' + last_name
, section name, course name
FROM course C RIGHT JOIN student St
ON St.course id = C.course id
LEFT JOIN section S
ON St.student_id = S.delegate_id
    
```

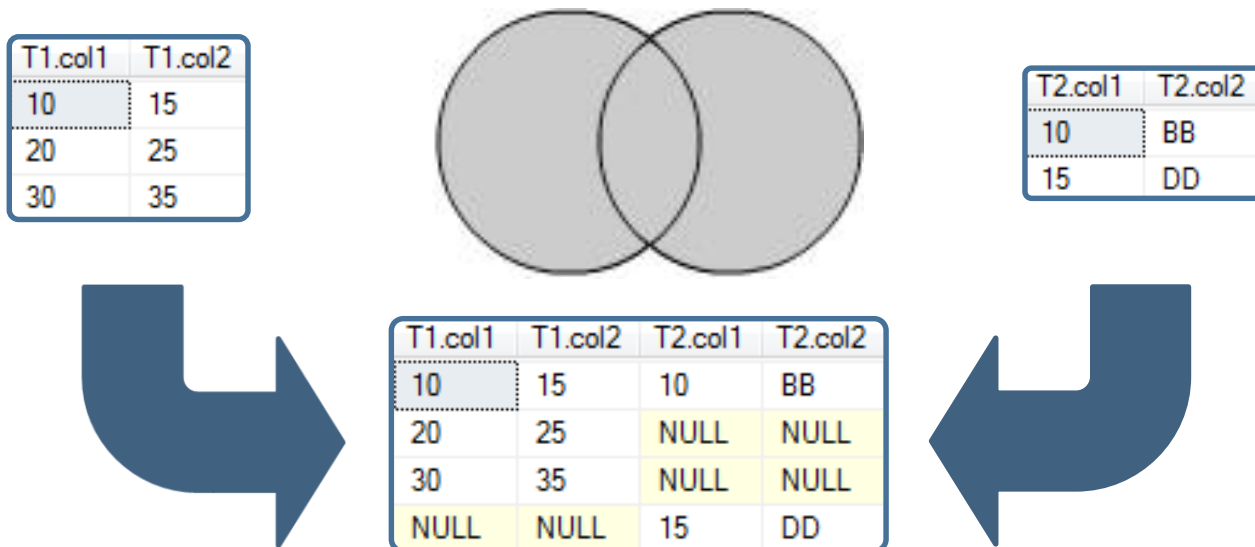
Nom étudiant	Section représentée	Cours choisi
Robert De Niro	NULL	Financial Management
Kevin Bacon	NULL	NULL
Kim Basinger	MSc Economics	NULL
Kim Basinger	MA Sociology	NULL
Johnny Depp	NULL	Financial Management
Julia Roberts	NULL	NULL
Natalie Portman	MSc Management	Financial Management
Georges Clooney	NULL	Marketing management

*Certains étudiants ne sont pas délégué de section, certains sont délégués de 2 sections, certains étudiants ne sont inscrits dans aucun cours, mais la liste de tous les étudiants doit apparaître quoiqu'il en soit*

# Jointures : **FULL OUTER JOIN**

```
SELECT *  
FROM table1 T1 FULL JOIN table2 T2 ON T1.col1 = T2.col1
```

Le « **FULL OUTER JOIN** » est une combinaison du **LEFT** et du **RIGHT** qui met en relation les lignes qui ont des éléments communs dans les colonnes indiquées et affiche toutes les autres lignes des deux tables, même si elles n'ont pas de point commun (mot-clé « **OUTER** » facultatif *sous SQL-Server*)



# Jointures : EQUI-JOIN

```
SELECT C.course_name, P.professor_name, S.section_name
FROM course C, professor P, section S
WHERE C.professor_id = P.professor_id
      AND P.section_id = S.section_id
```



```
SELECT C.course_name, P.professor_name, S.section_name
FROM (course C JOIN professor P
      ON C.professor_id = P.professor_id)
      JOIN section S
      ON P.section_id = S.section_id
```

Un « **ÉQUI-JOIN** » est le terme employé lorsque la condition de jointure est basée sur des **égalités strictes** entre les colonnes comparées

# Jointures : EQUI-JOIN

course_id	course_name	professor_id
ECGE2183	Financial Management	3
ECGE2184	Marketing management	2
EING2234	Derivatives	3
EING2283	Marketing engineering	1
EING2383	Supply chain management et e-business	5

Table « COURSE »

professor_id	professor_name	section_id
1	zidda	1020
2	decrop	1120
3	giot	1310
4	lescurt	1210
5	scheppens	1020
6	louveau	1110

Table  
« PROFESSOR »

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	BSc Economics	15
1120	MSc Economics	6
1310	BA Sociology	23
1320	MA Sociology	6

Table  
« SECTION »

```
SELECT C.course_name, P.professor_name, S.section_name
FROM (course C JOIN professor P
      ON C.professor_id = P.professor_id)
     JOIN section S
      ON P.section_id = S.section_id
```

course_name	prof_name	section_name
Financial Management	giot	BA Sociology
Marketing management	decrop	MSc Economics
Derivatives	giot	BA Sociology
Marketing engineering	zidda	MSc Management
Supply chain manage...	scheppens	MSc Management

# Jointures : **NON EQUI-JOIN**

```
SELECT S.last_name, S.year_result, G.Grade  
FROM Grade G, student S  
WHERE S.year_result BETWEEN G.Lower_bound AND G.Upper_bound
```



```
SELECT S.last_name, S.year_result, G.Grade  
FROM Grade G JOIN student S  
ON S.year_result BETWEEN G.Lower_bound AND G.Upper_bound
```

Un « **NON ÉQUI-JOIN** » est le terme employé lorsque la condition de jointure n'est pas basée sur des *égalités strictes* entre les colonnes comparées

# Jointures : NON EQUI-JOIN

```
SELECT S.last_name, S.year_result, G.Grade
FROM Grade G JOIN student S
ON S.year_result BETWEEN G.Lower_bound AND G.Upper_bound
```

Table  
« STUDENT »

last_name	year_result
Lucas	10
Eastwood	4
Connery	12
De Niro	3
Bacon	16
Basinger	19
Depp	11

Table  
« GRADE »

lower_bound	upper_bound	grade
14	15	B
18	20	E
10	11	F
8	9	I
0	7	IG
12	13	S
16	17	TB



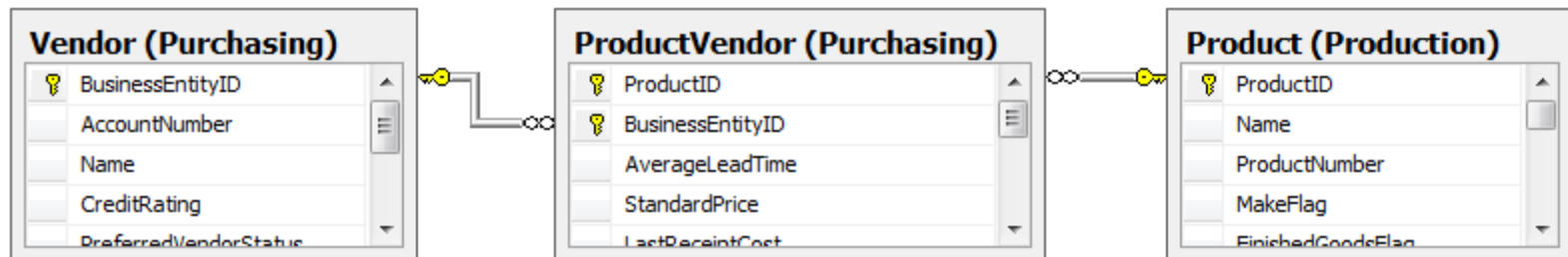
last_name	year_result	Grade
Basinger	19	E
Garcia	19	E
Gamer	18	E
Berry	18	E
Lucas	10	F
Depp	11	F
Reeves	10	F
Hanks	8	I
Eastwood	4	IG
De Niro	3	IG
Portman	4	IG
Clooney	4	IG



# Jointures : SELF-JOIN

```
SELECT *  
FROM table1 T1 JOIN table1 T2 ON T1.col1 = T2.col1
```

Le « **SELF-JOIN** » n'est rien d'autre qu'un « **INNER JOIN** » dans lequel les deux tables sont des copies de la même table d'origine. On utilise un « **SELF-JOIN** » lorsqu'on compare des éléments au sein de la même table. Les alias font en sorte que le système traite la requête comme un « **INNER JOIN** » classique, considérant les alias comme deux tables distinctes



La table « **ProductVendor** » de la base de données « **AdventureWorks** », représente le lien « **Many-to-Many** » entre les tables « **Vendor** » et « **Product** », mettant en relation quel vendeur a vendu quel produit  
À partir de la table « **ProductVendor** », nous aimerions savoir quel produit a été vendu par plus d'un vendeur

# Jointures : SELF-JOIN

```
SELECT pv1.ProductID, pv1.BusinessEntityID
FROM Purchasing.ProductVendor pv1
  INNER JOIN Purchasing.ProductVendor pv2
    ON pv1.ProductID = pv2.ProductID
      AND pv1.BusinessEntityID <> pv2.BusinessEntityID
```

Numero produit	Numero vendeur
1	1580
2	1688
4	1650
317	1578
317	1678
318	1578
318	1678
319	1556
319	1578
319	1678
320	1514
320	1602

Table « pv1 »

Numero Produit	Numero vendeur
317	1578
317	1678
318	1578
318	1678
319	1556
319	1578
319	1678
320	1602
320	1604
320	1514
321	1514
321	1602

Liste des produits vendus par plus d'un vendeur

Numero produit	Numero vendeur
1	1580
2	1688
4	1650
317	1578
317	1678
318	1578
318	1678
319	1556
319	1578
319	1678
320	1514
320	1602

Table « pv2 »

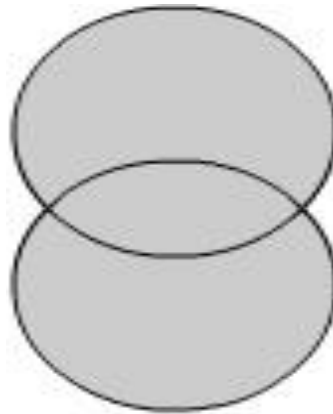
# Jointures verticales

```
SELECT ... FROM ... WHERE ... GROUP BY ...  
opérateur_comparaison_requêtes  
SELECT ... FROM ... WHERE ... GROUP BY ...  
ORDER BY ...
```

- Les jointures verticales comparent le résultat de deux requêtes indépendantes
- La comparaison des requêtes n'est possible que si chacune d'elles renvoie ***le même nombre de colonnes*** et que les colonnes en vis-à-vis sont du ***même type***
- L'affichage final résultant utilisera le nom des colonnes ou des alias utilisés ***dans la première requête***, il n'est donc pas nécessaire de donner des alias aux colonnes de la seconde
- Chaque requête peut contenir ***autant de clauses nécessaires*** à sa bonne réalisation (SELECT, FROM + JOIN, WHERE, GROUP BY, ...) à l'exception de la clause « ***ORDER BY*** » qui, si elle est utilisée, ***triera le résultat final*** résultant de la comparaison des deux requêtes. Il faudra toujours placer la clause « ***ORDER BY*** » à la suite de la deuxième requête

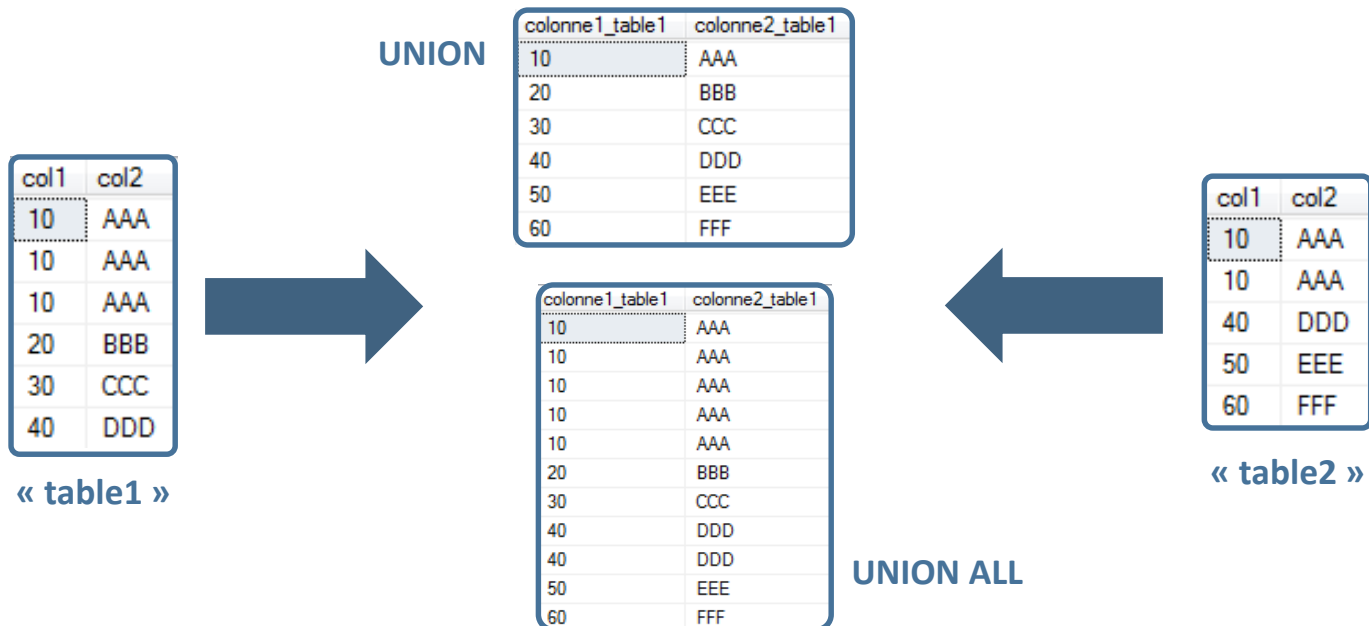
# Jointures : **UNION [ALL]**

- L'opérateur « **UNION** » applique un « **DISINCT** » aux résultats des deux requêtes et ajoute ensuite les lignes renvoyées par la seconde requête à celles présentées par la première, si elles sont différentes
- Le mot clé « **ALL** » peut être ajouté à l'opérateur « **UNION** » afin qu'absolument toutes les lignes ramenées par chacune des requêtes soient affichées, lignes déjà présentes dans le résultat de la première requête et doublons compris



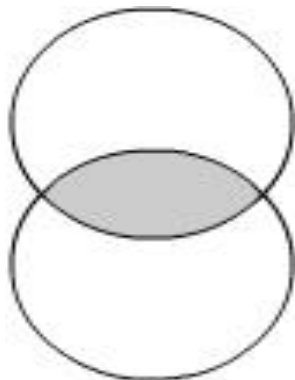
# Jointures : UNION [ALL]

```
SELECT col1 as [colonne1_table1], col2 as [colonne2_table1]
FROM table1
UNION
SELECT col1 as [colonne1_table2], col2 as [colonne2_table2]
FROM table2
ORDER BY [colonne2_table1]
```

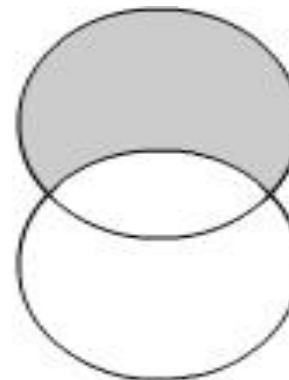


# Jointures : **INTERSECT** et **EXCEPT**

- L'opérateur « **INTERSECT** » n'affiche les lignes de la première requête que si elles se retrouvent également dans la seconde. *Les lignes en double ne sont comparées qu'une seule fois*
- L'opérateur « **EXCEPT** » n'affiche les lignes de la première requête que si elles **NE** se retrouvent **PAS** dans la seconde. *Les lignes en double ne sont comparées qu'une seule fois*
- « **INTERSECT ALL** » et « **EXCEPT ALL** » ne sont pas reconnus



*INTERSECT*



*EXCEPT*

# Jointures : INTERSECT

```
SELECT * FROM table1  
INTERSECT  
SELECT * FROM table2
```

col1	col2
10	AAA
10	AAA
10	AAA
20	BBB
30	CCC
40	DDD

« table1 »



col1	col2
10	AAA
40	DDD



col1	col2
10	AAA
10	AAA
40	DDD
50	EEE
60	FFF

« table2 »

The 'ALL' version of the INTERSECT operator is not supported.

# Jointures : EXCEPT

```
SELECT * FROM table1  
EXCEPT  
SELECT * FROM table2
```

*La version Oracle de l'opérateur « EXCEPT » est « MINUS »*

col1	col2
10	AAA
10	AAA
10	AAA
20	BBB
30	CCC
40	DDD

« table1 »



col1	col2
20	BBB
30	CCC



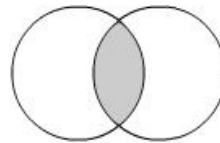
col1	col2
10	AAA
10	AAA
40	DDD
50	EEE
60	FFF

« table2 »

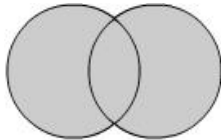
The 'ALL' version of the EXCEPT operator is not supported.



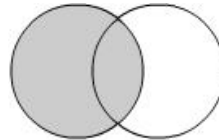
# Jointures : Résumé



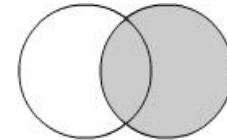
INNER JOIN



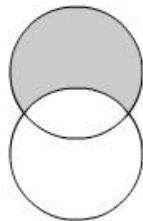
FULL OUTER JOIN



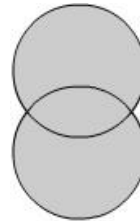
LEFT OUTER JOIN



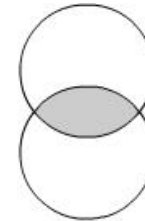
RIGHT OUTER JOIN



EXCEPT



UNION



INTERSECT

# Auto-Evaluation

N'oubliez pas de prendre le temps d'évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant les notions abordées dans ce module !

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit. Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Notions à évaluer

Notions	P	S	V	I
Différence entre jointures « horizontales » et « verticales »				
« CROSS JOIN »				
Equi-join (« INNER JOIN » entre plusieurs tables)				
« LEFT/RIGHT/FULL OUTER JOIN »				
SELF-JOIN				
Opérateurs « UNION », « INTERSECT », « EXCEPT »				

# Sous-Requêtes

```
SELECT ... FROM ...  
WHERE (SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...)  
GROUP BY ... ORDER BY ...
```

```
SELECT ...  
FROM (SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...) AS T1  
WHERE ... GROUP BY ... ORDER BY ...
```

```
SELECT ... FROM ... WHERE ...  
GROUP BY ... HAVING (SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...)  
GROUP BY ... ORDER BY ...
```

# Sous-Requêtes

- Une « **sous-requête** » est une **requête évaluée à l'intérieur d'une autre** requête et dont le résultat influence le résultat de la requête principale
- Une sous-requête est **toujours placée entre parenthèses**
- Il est possible d'utiliser une sous-requête **dans la clause « FROM », « WHERE » ou « HAVING »**
- Il est important de **bien visualiser le résultat produit par la requête imbriquée** afin de l'utiliser correctement dans la requête principale. Dans un premier, n'oublions pas qu'il est possible de n'exécuter qu'**une partie du code en le surlignant**, lorsqu'on travaille avec Microsoft SQL Server Management Studio

# Sous-Requêtes : **WHERE** et **HAVING**

```
SELECT ... FROM ...  
WHERE (SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...)  
GROUP BY ... ORDER BY ...
```

```
SELECT ... FROM ... WHERE ...  
GROUP BY ... HAVING (SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...)  
GROUP BY ... ORDER BY ...
```

- Lors de l'utilisation de requêtes imbriquées dans les conditions posées par le « **WHERE** » ou le « **HAVING** », il est indispensable que les données renvoyées soient **cohérente avec l'expression** dans laquelle elles sont utilisées (nombre de valeurs et type)
- Les données renvoyées seront de trois types :
  - **Scalaires** (une seule valeur)
  - **Multi-valeurs** (un ensemble de données scalaires, soit une colonne entière)
  - **Tabulaire** (un ensemble de lignes et de colonnes)

# Sous-Requêtes : WHERE et HAVING

## Scalar-valued subquery : « WHERE »

```
SELECT last_name, year_result
FROM student
WHERE year_result >= (SELECT year_result FROM student
                      WHERE last_name LIKE 'Bacon')
```

Si la valeur renvoyée par la sous-requête est **une valeur scalaire**, alors il est tout à fait possible d'utiliser les **opérateurs classiques d'inégalité** dans l'expression

last_name	year_result
Bacon	16
Basinger	19
Roberts	17
Garcia	19
Gamer	18
Berry	18

← Valeur renvoyée par la sous-requête

Liste des étudiants dont le résultat annuel est plus grand ou égal au résultat de M. Bacon

# Sous-Requêtes : WHERE et HAVING

## Scalar-valued subquery : « WHERE »

```
SELECT last_name, year_result
FROM student
WHERE year_result > (SELECT AVG(year_result)
                     FROM student) → 8
```

*Une agrégation globale* fonctionne bien également puisqu'elle renvoie **une seule valeur**

last_name	year_result
Lucas	10
Connery	12
Bacon	16
Basinger	19
Depp	11
Roberts	17
Garcia	19
Gamer	18
Reeves	10
Berry	18

*Liste des étudiants ayant un résultat plus élevé que la moyenne*



# Sous-Requêtes : WHERE et HAVING

## Scalar-valued subquery : « HAVING »

```
SELECT section_id, AVG(year_result) as [Moyenne]
FROM student
GROUP BY section_id
HAVING AVG(year_result) > (SELECT AVG(year_result)
                           FROM student)
```

section_id	Moyenne
1120	17
1310	11
1320	10

*Liste des sections dont la moyenne est plus grande que la moyenne globale*

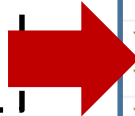
# Sous-Requêtes : WHERE et HAVING

## Multi-valued subquery : « [NOT] IN » operator

```
SELECT last_name, year_result
FROM student
WHERE year_result IN (SELECT MAX(year_result)
                      FROM student
                      GROUP BY section_id)
```

Si la sous-requête renvoie plus d'une valeur, il devient impossible d'utiliser les **opérateurs classiques d'inégalité**. L'opérateur « **IN** » permettra de comparer la valeur d'une colonne à chaque élément de la liste des valeurs renvoyées par la sous-requête, par exemple

```
SELECT MAX(year_result)
FROM student
GROUP BY section_id
```



Maximum par section
6
12
19
18
19
18

last_name	year_result
Connery	12
Basinger	19
Garcia	19
Willis	6
Marceau	6
Gamer	18
Bemy	18

**Si le résultat annuel de l'étudiant est égal à au moins l'une des valeurs renvoyées par la sous-requête, les données sont affichées**

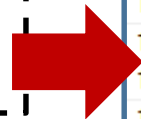
# Sous-Requêtes : WHERE et HAVING

## Multi-valued subquery : « ANY » operator

```
SELECT last_name, year_result
FROM student
WHERE year_result > ANY (SELECT MAX(year_result)
                        FROM student
                        GROUP BY section_id)
```

L'opérateur « **ANY** » peut être utilisé en plus des opérateurs d'*inégalité classiques* afin de comparer la valeur d'une colonne individuellement à chacune des valeurs de la liste renvoyée par la sous-requête. Si *au moins l'une des comparaisons* renvoie **TRUE**, les données sont affichées

```
SELECT MAX(year_result)
FROM student
GROUP BY section_id
```



Maximum par section
6
12
19
18
19
18

Si le résultat annuel de l'étudiant est supérieur à au moins l'une des valeurs renvoyées par la sous-requête, les données sont affichées

last_name	year_result
Witherspoon	7
Michelle Gellar	7
Milano	7
Hanks	8
Reeves	10
Lucas	10
Depp	11
Copper	12

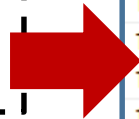
# Sous-Requêtes : WHERE et HAVING

## Multi-valued subquery : « ALL » operator

```
SELECT last_name, year_result
FROM student
WHERE year_result >= ALL (SELECT MAX(year_result)
                           FROM student
                           GROUP BY section_id)
```

L'opérateur « **ALL** » peut être utilisé en plus des opérateurs d'*inégalité classiques* afin de comparer la valeur d'une colonne individuellement à chacune des valeurs de la liste renvoyée par la sous-requête. Si **toutes les comparaisons** renvoient **TRUE**, les données sont affichées

```
SELECT MAX(year_result)
FROM student
GROUP BY section_id
```



Maximum par section	
6	
12	
19	
18	
19	
18	

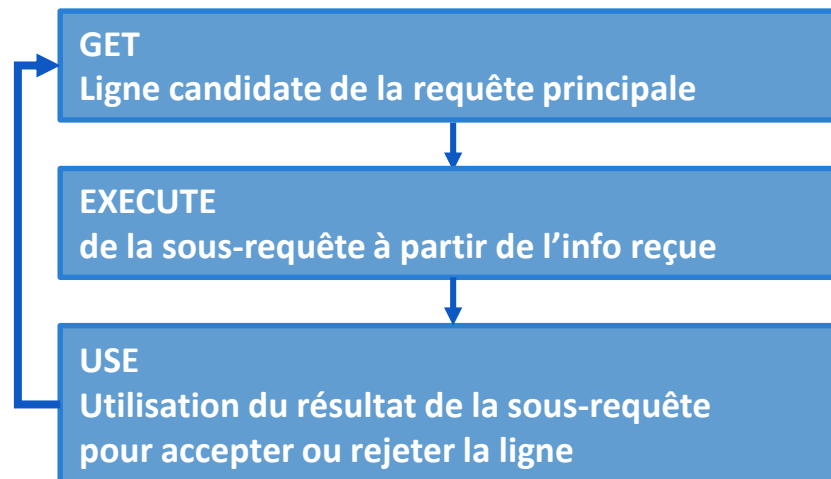
last_name	year_result
Basinger	19
Garcia	19

**Si le résultat annuel de l'étudiant est supérieur ou égal à toutes les valeurs renvoyées par la sous-requête, les données sont affichées**

# Sous-Requêtes : Corrélation

```
SELECT ... FROM table1 as T1  
WHERE (SELECT ... FROM table1 as T2 WHERE T1.col1 = T2.col1 ...) ...  
GROUP BY ... ORDER BY ...
```

Une requête « **corrélée** » signifie que le résultat renvoyé par la sous-requête dépend directement de la ligne actuellement rencontrée par la requête principale. Le résultat de la sous-requête est donc réévalué et potentiellement différent à chaque ligne rencontrée dans la requête principale



# Sous-Requêtes : Corrélation

```
SELECT last_name, section_id, year_result
FROM student AS s1
WHERE year_result > (SELECT AVG(year_result)
                     FROM student
                     WHERE section_id = s1.section_id )
```

last_name	section_id	year_result
Bacon	1120	16
Basinger	1310	19
Berry	1320	18
Bullock	1010	2
Clooney	1020	4
Connery	1020	12
Cruise	1020	4
De Niro	1110	3
Depp	1110	11
Doherty	1320	2
Garcia	1110	19
Gamer	1120	18

Table « S1 »

last_name	section_id	year_result
Bacon	1120	16
Basinger	1310	19
Berry	1320	18
Bullock	1010	2
Clooney	1020	4
Connery	1020	12
Cruise	1020	4
De Niro	1110	3
Depp	1110	11
Doherty	1320	2
Garcia	1110	19
Gamer	1120	18

Table « STUDENT »

section_id	Moyenne par section
1010	4
1020	7
1110	8
1120	17
1310	11
1320	10 < 18

Moyennes

last_name	section_id	year_result
Basinger	1310	19
Berry	1320	18
Connery	1020	12
Depp	1110	11
Garcia	1110	19
Gamer	1120	18
Hanks	1020	8
Reeves	1020	10
Willis	1010	6

Résultat

# Sous-Requêtes : [NOT] EXISTS

```
SELECT last_name
FROM student as s
WHERE EXISTS (SELECT * FROM inscriptions as i
              WHERE i.student_id = s.student_id
                 AND i.course_id = 'EING2234')
```

- L'opérateur « **EXISTS** » permet de n'afficher les données demandées que si le résultat de la sous-requête produit au moins une ligne de données (le nombre de lignes renvoyées par la sous-requête n'a pas d'importance)
- Ce résultat est donc de **type tabulaire** et bien souvent **corrélé**, c'est-à-dire qu'il tient compte des données contenues dans la requête principale
- Le mot-clé « **NOT** » peut être ajouté devant l'opérateur « **EXISTS** » afin de formuler la négation

# Sous-Requêtes : [NOT] EXISTS

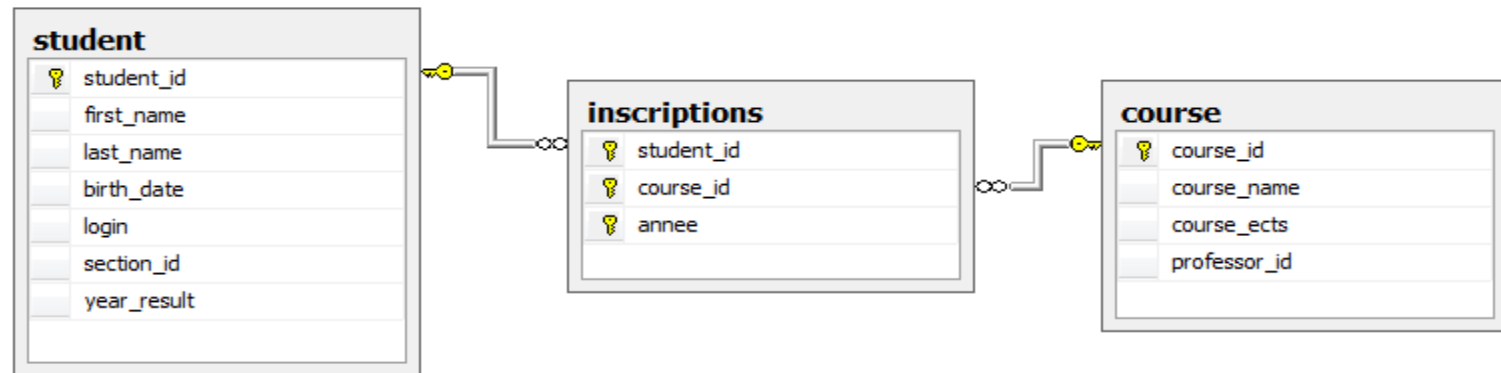


Table  
« STUDENT »

student_id	last_name
1	Lucas
2	Eastwood
3	Connery
4	De Niro
5	Bacon
6	Basinger
7	Depp
8	Roberts
9	Portman
10	Clooney

student_id	course_id	annee
1	ECGE2183	1960-09-01
1	EING2283	1960-09-01
3	EING2234	1960-09-01
3	EING2283	1960-09-01
3	EING2383	1960-09-01
4	EING2283	1960-09-01
6	EING2234	1960-09-01
9	EING2234	1960-09-01
9	EING2383	1960-09-01

Table  
« INSCRIPTIONS »

Étudiants inscrits au cours EING2234

```
SELECT student_id, last_name
FROM student as s
WHERE EXISTS (
  SELECT *
  FROM inscriptions as i
  WHERE i.student_id = s.student_id
  AND i.course_id = 'EING2234')
```

student_id	last_name
3	Connery
6	Basinger
9	Portman



# Sous-Requêtes : **FROM** et **WITH**

```
SELECT ...  
FROM (SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...) AS T1  
WHERE ... GROUP BY ... ORDER BY ...
```



```
WITH table_CTE (nom_col1, nom_col2, nom_col3, ..., nom_colN)  
AS  
(SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ...)  
SELECT ... FROM table_CTE WHERE ... GROUP BY ... ORDER BY ...
```

*CTE = Common Table Expression*

# Sous-Requêtes : FROM et WITH

- Une sous-requête de **type tabulaire** (renvoyant plusieurs lignes et plusieurs colonnes) peut être **traitée comme une table** à part entière et servir de référence pour la requête principale
- Dans le cas où la sous-requête est utilisée dans une clause « **FROM** », il est **nécessaire de lui donner un alias** afin de l'utiliser comme un nom de table dans la requête principale
- Il faudra toujours donner un alias aux colonnes affichant le résultat d'une expression
- Lors de l'utilisation d'une requête imbriquée avec la clause « **WITH** », la requête sert à fournir la table pré-déclarée et doit renvoyer le même nombre de colonnes qu'annoncé dans la clause « **WITH** »
- La plupart des systèmes montrent de **meilleures performances** avec l'utilisation de la clause « **WITH** », mais cela ne doit pas devenir une généralité. Certains cas d'utilisation peuvent démontrer le contraire au sein du même système

# Sous-Requêtes : FROM et WITH

```
SELECT section_name as [Section], Nbr as [Nombre d'étudiants]
FROM (SELECT section_id, COUNT(*) as [Nbr] FROM student
      GROUP BY section_id) as std
JOIN section as s ON s.section_id = std.section_id
WHERE Nbr > 5
```



```
WITH std (section_id, Nbr)
AS
( SELECT section_id, COUNT(*) as [Nbr]
  FROM student
 GROUP BY section_id)
SELECT section_name as [Section], Nbr as [Nombre d'étudiants]
FROM std JOIN section as s ON s.section_id = std.section_id
WHERE Nbr > 5
```

*Liste des sections contenant plus de 5 étudiants*

# Sous-Requêtes : FROM et WITH

```
WITH DirectReports(Name, Title, EmployeeID, EmployeeLevel, Sort, ManagerID)
AS (SELECT CONVERT(varchar(255), e.FirstName + ' ' + e.LastName),
      e.Title, e.EmployeeID, 1,
      CONVERT(varchar(255), e.FirstName + ' ' + e.LastName),
      ManagerID
  FROM dbo.MyEmployees AS e
 WHERE e.ManagerID IS NULL
 UNION ALL
  SELECT CONVERT(varchar(255), REPLICATE ('|      ', EmployeeLevel) +
    e.FirstName + ' ' + e.LastName),
    e.Title, e.EmployeeID, EmployeeLevel + 1,
    CONVERT (varchar(255), RTRIM(Sort) + '|      ' +
      FirstName + ' ' + LastName),
    e.ManagerID
  FROM dbo.MyEmployees AS e
  JOIN DirectReports AS d ON e.ManagerID = d.EmployeeID
)
SELECT EmployeeID, Name, Title, EmployeeLevel, ManagerID
FROM DirectReports
ORDER BY Sort
```

Clause « **WITH** » utilisée dans le cadre de l'**affichage hiérarchique** des employés d'une société

# Sous-Requêtes : FROM et WITH

Table  
« MYEMPLOYEES »

EmployeeID	FirstName	LastName	Title	DeptID	ManagerID
1	Ken	Sánchez	Chief Executive Officer	16	NULL
16	David	Bradley	Marketing Manager	4	273
23	Mary	Gibson	Marketing Specialist	4	16
273	Brian	Welcker	Vice President of Sales	3	1
274	Stephen	Jiang	North American Sale...	3	273
275	Michael	Blythe	Sales Representative	3	274
276	Linda	Mitchell	Sales Representative	3	274
285	Syed	Abbas	Pacific Sales Manager	3	273
286	Lynn	Tsoflias	Sales Representative	3	285

EmployeeID	Name	Title	EmployeeLevel	ManagerID
1	Ken Sánchez	Chief Executive Officer	1	NULL
273	Brian Welcker	Vice President of Sales	2	1
16	David Bradley	Marketing Manager	3	273
23	Mary Gibson	Marketing Specialist	4	16
274	Stephen Jiang	North American Sales Manager	3	273
276	Linda Mitchell	Sales Representative	4	274
275	Michael Blythe	Sales Representative	4	274
285	Syed Abbas	Pacific Sales Manager	3	273
286	Lynn Tsoflias	Sales Representative	4	285

Résultat de  
la requête du slide  
précédent

# Sous-Requêtes : JOIN VS Sous-requête

```
SELECT DISTINCT course_name  
  FROM course  
 WHERE course_id IN (SELECT course_id FROM inscriptions )
```

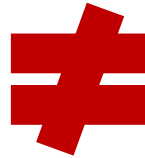


```
SELECT DISTINCT course_name  
FROM course C JOIN inscriptions I  
  ON C.course_id = I.course_id
```

# Sous-Requêtes : JOIN VS Sous-requête

```
SELECT DISTINCT course_name  
FROM course  
WHERE course_id NOT IN (SELECT course_id FROM inscriptions )
```

Retourne le nom du cours de la table « **Course** » s'il n'existe pas dans la table « **Inscriptions** »



```
SELECT DISTINCT course_name  
FROM course C JOIN inscriptions I  
ON C.course_id <> I.course_id
```

Retourne le nom du cours de la table « **Course** » s'il est différent de l'un des cours de la table « **Inscriptions** »

# Auto-Evaluation

N'oubliez pas de prendre le temps d'évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant les notions abordées dans ce module !

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit.  
Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !



# Auto-Evaluation

## Notions à évaluer

Notions	P	S	V	I
Types de sous-requêtes (scalaire, multi-valeur, tabulaire)				
Sous-requêtes dans les clauses « WHERE » et « HAVING »				
Opérateurs « ALL » et « ANY »				
Sous-requête corrélée				
Opérateur « EXISTS »				
Sous-requêtes dans la clause « FROM »				
Clause « WITH »				

## Partie 4

# **DML – DATA MANIPULATION LANGUAGE**

Insertion de données

Mise à jour de données

Suppression de données

OUTPUT

# Insertion de données

```
INSERT INTO table (col1, col2, ..., colN) VALUES  
(valeur1_col1, valeur1_col2, ..., valeur1_colN),  
(valeur2_col1, valeur2_col2, ..., valeur2_colN), ...
```

- L'ordre « **INSERT** » permet d'insérer des nouvelles lignes de données dans une table
- **La liste des colonnes** concernées par l'insertion n'est **pas obligatoire**, mais dans ce cas, les valeurs insérées doivent l'être **dans le même ordre** que celui dans lequel les colonnes apparaissent dans la table
- Il est possible de **ne pas insérer de valeur dans l'une des colonnes** de la table. Il suffit pour ce faire de ne pas indiquer le nom de la colonne dans la liste des colonnes spécifiées après le nom de la table
- Sous SQL Server, il est possible d'insérer **plusieurs lignes** en une seule requête en séparant les lignes à insérer par des virgules
- L'insertion doit respecter les contraintes posées sur la table...

# Insertion de données

```
INSERT INTO section (section_id, section_name, delegate_id)
VALUES (1415, 'SQL Déclaratif', 23),
       (1516, NULL, 12),
       (1617, 'Administration SQL Server', 4)
```

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	BSc Economics	15
1120	MSc Economics	6
1310	BA Sociology	23
1320	MA Sociology	6
1415	SQL Déclaratif	23
1516	NULL	12
1617	Administration SQL Server	4

*Insertion de 3 nouvelles lignes de données dans la table « **section** »*

# Insertion de données

```
INSERT INTO section (section_name, section_id)
VALUES ('SQL Procédural', 1718)
```

```
INSERT INTO section VALUES (1819, 'Business Intelligence', 23)
```

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	BSc Economics	15
1120	MSc Economics	6
1310	BA Sociology	23
1320	MA Sociology	6
1415	SQL Déclaratif	23
1516	NULL	12
1617	Administration SQL Server	4
1718	SQL Procédural	NULL
1819	Business Intelligence	23
1920	NULL	NULL

# Insertion de données : **DEFAULT**

```
INSERT INTO section VALUES (1920, DEFAULT, DEFAULT)
INSERT INTO section VALUES (2020, DEFAULT, NULL)
```

- Si l'une des colonnes possède une **valeur par défaut** ou accepte les valeurs « **NULL** », il est possible de ne pas insérer manuellement de valeur dans cette colonne en indiquant comme valeur le mot-clé « **DEFAULT** ». Il faut également procéder de cette façon pour fournir les valeurs à une colonne dont les valeurs sont **auto-incrémentées**
- L'instruction **INSERT INTO table DEFAULT VALUES** peut être utilisée si toutes les colonnes de la table peuvent prendre une valeur par défaut

section_id	section_name	delegate_id
1920	NULL	NULL
2020	NULL	NULL

# Insertion de données : **SELECT**

```
INSERT INTO section (section_id, section_name, delegate_id)
VALUES (2021
      , DEFAULT
      , (SELECT student_id FROM student
         WHERE last_name LIKE 'Willis') )
```

Le résultat d'un ordre « **SELECT** » peut être utilisé comme valeur pour l'une des colonnes si cet ordre renvoie bien ***une seule valeur***, du même type que la colonne correspondante

**Rappel** : un ordre « **SELECT** » utilisé comme sous-requête est ***toujours*** placé entre parenthèses

section_id	section_name	delegate_id
2021	NULL	12

# Insertion de données : SELECT

```
INSERT INTO section_archives (section_id, delegate_id)
SELECT DISTINCT s.section_id, s.delegate_id
FROM section S JOIN professor P ON P.section_id = S.section_id
```

- L'ordre « **SELECT** » permet également d'*insérer plusieurs lignes* en une seule fois dans une table
- Dans ce cas, le mot-clé « **VALUES** » *doit être omis*
- La requête doit bien entendu *renvoyer le même nombre de colonnes* que les colonnes à fournir

section_id	section_name	delegate_id
1020	NULL	9
1110	NULL	15
1120	NULL	6
1310	NULL	23



# Mise à jour de données

```
UPDATE table  
SET col1 = nouvelle_valeur_col1, col2 = nouvelle_valeur_col2, ...  
WHERE ...
```

- L'ordre « **UPDATE** » permet de mettre à jour des données existantes dans une table
- La clause « **WHERE** » n'est pas obligatoire, mais elle permet de spécifier la ou les lignes auxquelles les mises à jour doivent avoir lieu
- Un ordre « **SELECT** » peut bien sûr être utilisé pour renvoyer la valeur à utiliser pour la mise à jour

# Mise à jour de données

```
UPDATE section
SET delegate_id = (SELECT student_id FROM student
                   WHERE last_name LIKE 'Cruise')
    , section_name = 'SQL Déclaratif'
WHERE CONVERT(VARCHAR,section_id) LIKE '11%'
```

**Rappel :** un ordre « *SELECT* » utilisé comme sous-requête est **toujours** placé entre parenthèses

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	SQL Déclaratif	13
1120	SQL Déclaratif	13
1310	BA Sociology	23
1320	MA Sociology	6

# Mise à jour de données

```
UPDATE section
  SET delegate_id = std.student_id
    , section_name = 'SQL Déclaratif'
  FROM section s, student std
 WHERE CONVERT(VARCHAR,s.section_id) LIKE '11%'
    AND std.last_name LIKE 'Cruise'
```

Il est également possible d'utiliser une syntaxe semblable à celle de l'ordre « **SELECT** » pour l'exécution de l'ordre « **UPDATE** », jointures comprises. Le « **SELECT** » devient alors un « **SET** » et les colonnes ne sont pas affichées mais fournissent la valeur aux colonnes à mettre à jour

section_id	section_name	delegate_id
1010	BSc Management	12
1020	MSc Management	9
1110	SQL Déclaratif	13
1120	SQL Déclaratif	13
1310	BA Sociology	23
1320	MA Sociology	6

# Suppression de données

```
DELETE FROM table  
WHERE ...
```

- L'ordre « **DELETE** » permet de supprimer les lignes d'une table
- La clause « **WHERE** » n'est pas obligatoire, mais elle permet de spécifier la ou les lignes auxquelles la suppression s'applique

```
DELETE FROM student
```

```
DELETE FROM student WHERE student_id = 20
```

# OUTPUT

- **Sous SQL-Server**, la clause « **OUTPUT** » permet, à la suite d'un ordre DML, d'immédiatement renvoyer les lignes modifiées par l'ordre DML, comme si on exécutait un ordre « **SELECT** » sur ces données, à la suite de l'insertion
- Lors de l'utilisation d'un ordre DML, **2 tables sont utilisées** pour stocker momentanément l'information manipulées. La table « **INSERTED** » est utilisée pour stocker momentanément toutes les nouvelles données (lors des ordres « **INSERT** » et « **UPDATE** »). La table « **DELETED** » stockera les données amenées à disparaître (lors d'un « **UPDATE** » ou d'un « **DELETE** »). Notons qu'il n'existe pas de table « **UPDATED** »

```
INSERT INTO section OUTPUT INSERTED.* VALUES (3030, NULL, 10)
```

```
UPDATE section SET section_name = 'SQL Déclaratif'  
OUTPUT INSERTED.*, DELETED.section_id  
WHERE section_id IN (1010,1020)
```

```
DELETE FROM section OUTPUT DELETED.section_name, DELETED.delegate_id
```

Partie 5

# NOTIONS AVANCÉES

Gestion des transactions

Fusion de données

# Gestion des transactions

Une **Transaction** est représentée par un ordre ou un ensemble d'ordres qui **modifient l'état** de la base de données

Toute transaction au sein d'un SGBD relationnel, répond à **la loi « ACID »** :

## ATOMIQUE

*L'ensemble des ordres d'une transaction sont validés ou bien aucun ne l'est. Si un ordre échoue, l'ensemble de la transaction est annulée*

## COHÉRENTE

*Une transaction fait toujours passer le système d'un état valide à un autre état valide dans lequel l'ensemble des règles définies pour la base de données sont respectées*

## ISOLÉE

*Les transactions s'exécutent les unes après les autres et il n'existe qu'une seule transaction par programme client. Pour qu'une nouvelle transaction commence, la précédente doit se terminer*

## DURABLE

*Une transaction validée l'est de façon définitive, survivant à toute défaillance technique du système*

# Gestion des transactions

- La gestion des transactions s'effectue principalement selon deux ordres : « **COMMIT** » (pour valider une transaction) et « **ROLLBACK** » (pour annuler une transaction)
- La plupart des système travaillent en mode « **AUTO-COMMIT** », ce qui signifie qu'un ordre « **COMMIT** » implicite est exécuté à la suite de chaque ordre visant à modifier l'état de la base de données
- Afin d'éviter le mode « **AUTO-COMMIT** » sous *SQL-Server*, il sera nécessaire de commencer l'ensemble des ordres par l'instruction « **BEGIN TRANSACTION** » et de terminer la transaction explicite par un « **COMMIT TRANSACTION** » ou un « **ROLLBACK TRANSACTION** »
- Un ordre « **SELECT** » peut faire partie d'une transaction explicite, mais il n'a aucun impact sur la transaction elle-même
- Lorsque deux transactions concurrentes essayent d'atteindre la même information au même instant, un problème de concurrence d'accès peut avoir lieu (« **DEADLOCK** »). Ces problèmes de concurrence d'accès peuvent être géré par des verrous (« **LOCKS** ») ou en modifiant la **visibilité qu'une transaction** a d'une autre, c'est-à-dire le mode d'exécution des transactions. Ces notions ne seront pas abordées en détail dans le cadre de ce cours



# Gestion des transactions

Cette transaction explicite ne modifie rien au niveau du système :

```
BEGIN TRANSACTION
```

```
DELETE FROM student WHERE section_id IN  
( SELECT section_id FROM student  
GROUP BY section_id  
HAVING AVG(year_result) >= 10)
```

```
SELECT * FROM student
```

```
DELETE FROM student
```

```
SELECT * FROM student
```

```
ROLLBACK TRANSACTION
```

# Fusion de données

```
MERGE INTO table_cible AS alias_table_cible  
USING (données_à_comparer) AS alias_table_source  
ON alias_table_cible.colonne_comparée = alias_table_source.colonne_comparée  
WHEN MATCHED THEN ...  
WHEN NOT MATCHED THEN ...
```

- L'ordre « **MERGE** » permet de comparer deux jeux de données, en se basant sur une condition de jointure entre ces jeux de données et d'agir en fonction d'un résultat semblable ou différent
- Cet ordre est notamment utilisé pour mettre à jour une table, ne modifier les données que si elles existent déjà et rajouter les lignes qui n'existent pas encore

# Fusion de données

```
MERGE INTO dbo.A AS table_cible
  USING (SELECT * FROM B GROUP BY col1, col2)
  AS table_source (colonne1, colonne2)
  ON table_cible.col1 = table_source.colonne1
  WHEN MATCHED THEN UPDATE SET col2 = 'MATCH'
  WHEN NOT MATCHED THEN INSERT (col1,col2)
  VALUES (colonne1, colonne2);
```

col1	col2
10	AAA
10	AAA
20	BBB
30	CCC
40	DDD

Table « A »

col1	col2
10	AAA
10	AAA
10	AAA
40	DDD
50	EEE
60	FFF

Table « B »

col1	col2
10	MATCH
10	MATCH
20	BBB
30	CCC
40	MATCH
50	EEE
60	FFF

Table « A »  
Après le  
« MERGE »

# Auto-Evaluation

N'oubliez pas de prendre le temps d'évaluer le niveau de maîtrise que vous estimez avoir acquis personnellement concernant les notions abordées dans ce module !

Rappel de la signification des lettres dans les tableaux d'auto-évaluation :

- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit.  
Une explication supplémentaire du formateur ou une bonne révision de votre part s'impose
- **Insatisfaisant (I)** : Vous n'avez pas du tout compris la notion abordée, il faut tout faire pour y remédier !

# Auto-Evaluation

## Notions à évaluer

Notions	P	S	V	I
Insertion de données ligne par ligne (VALUES...)				
Insertion de données par lot (SELECT...)				
Mise à jour simple de données				
Mise à jour sous forme de jointure				
Suppression de données				
Clause « OUTPUT »				
Transactions et loi ACID (en théorie)				
Gestion de transactions explicites (« BEGIN/COMMIT/ROLLBACK »)				

# Références

- ELMASRI R., NAVATHE S., **Fundamentals of Databases Systems: Pearson New International Edition**, États-Unis, Pearson, 2013, 6th Edition
- BEN-GAN I., **Microsoft SQL Server 2012 T-SQL Fundamentals**, États-Unis, Microsoft Press, 2012, 1st Edition
- BEN-GAN I., KOLLAR L., SARKA D., KASS S., **Inside Microsoft SQL Server 2008 T-SQL Querying**, États-Unis, Microsoft Press, 2009, 1st Edition
- O'HEARN S., **OCA Oracle Database SQL SQL Certified Expert Exam Guide**, États-Unis, Microsoft Press, 2009, 1st Edition
- **MSDN-the microsoft developer network**, site de Microsoft : [msdn.microsoft.com](http://msdn.microsoft.com)
- **Oracle Database Online Documentation 12c Release (12.1)**, site d'Oracle : [docs.oracle.com](http://docs.oracle.com)