

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
- **AUTO-INCREMENTATION 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

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

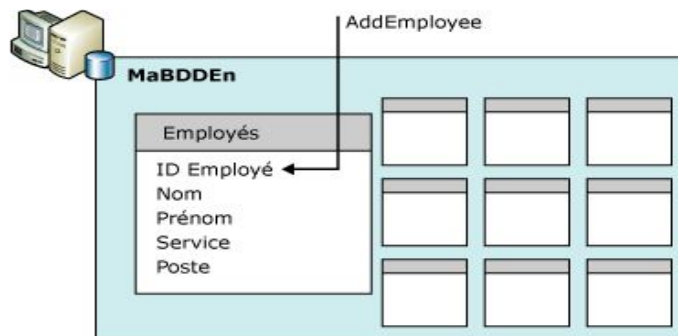
L'arrivée de l'ordinateur et des programmes informatiques a ensuite permis de **stocker les données dans des fichiers informatiques**, simplifiant et allégeant déjà énormément la gestion des données. Le gros désavantage des fichiers informatiques tels que ceux utilisés dans Excel ou Access est bien entendu 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**

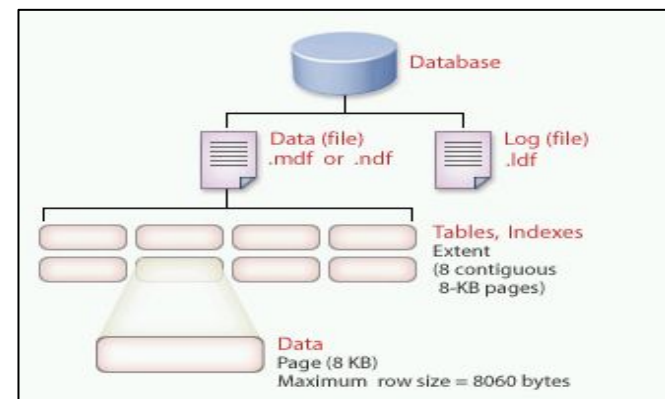
Base de données et SGBD

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

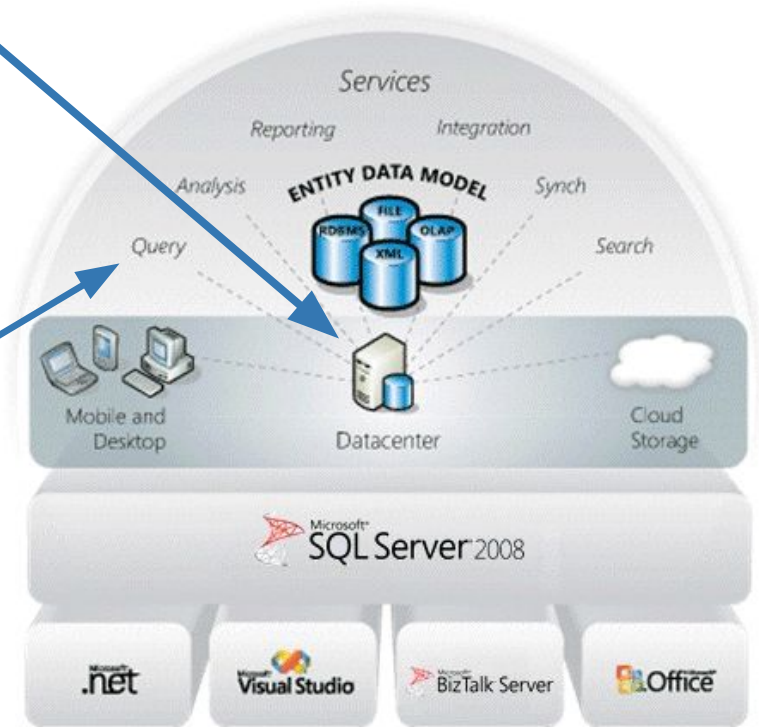
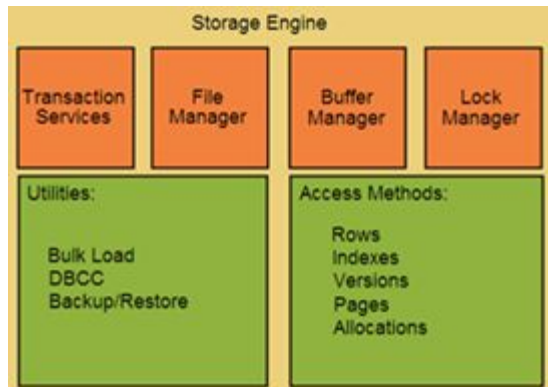
- **Sécurité des données**

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



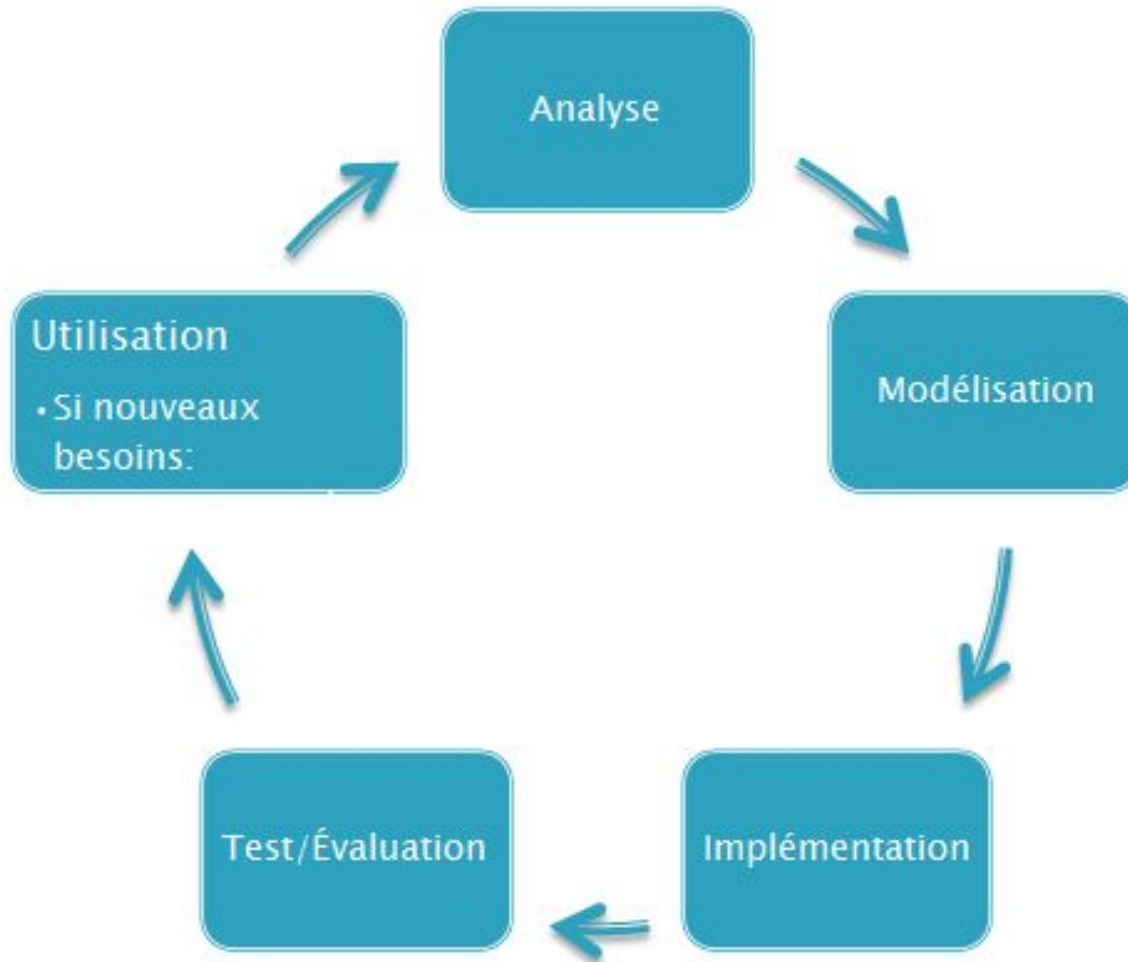
De l'analyse au relationnel

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**

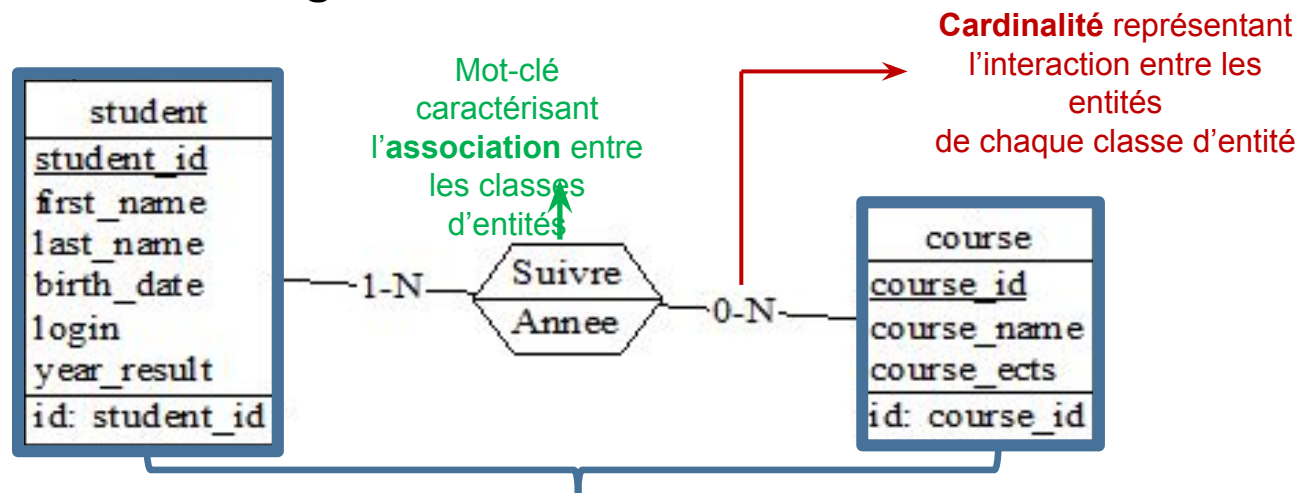
De l'analyse au relationnel



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

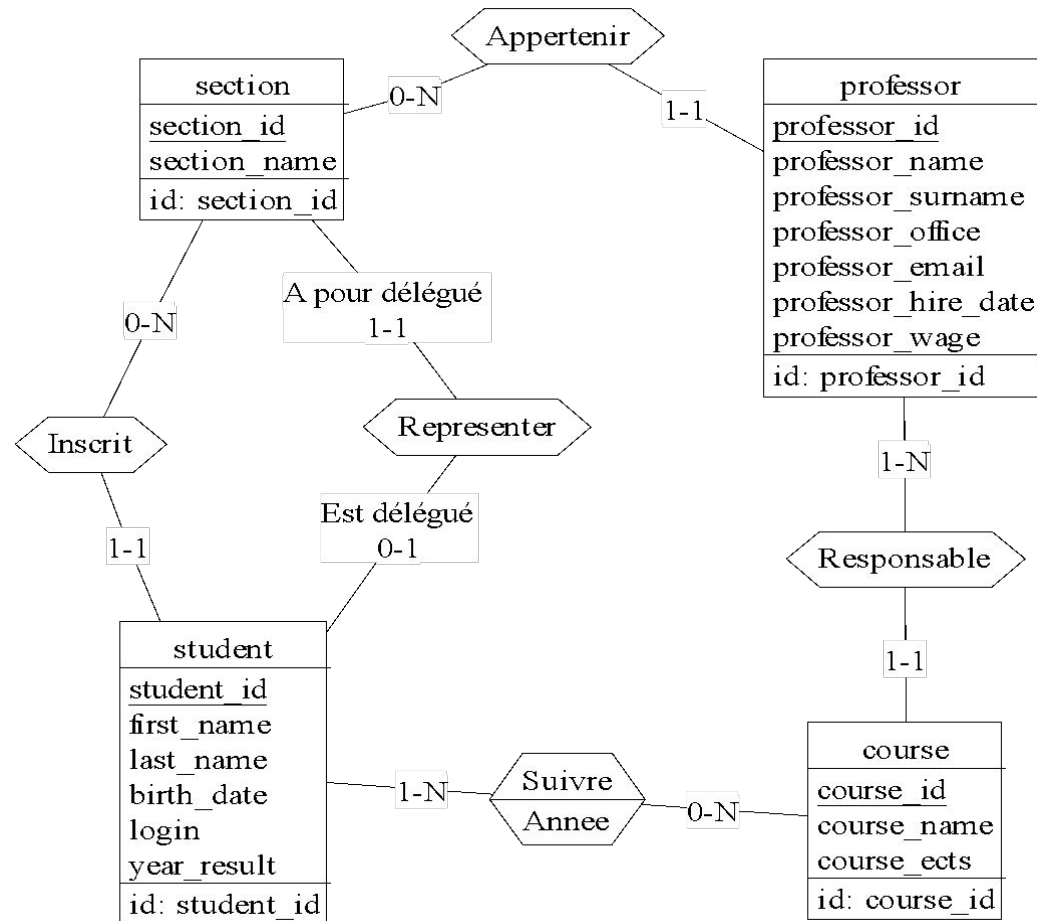
d'enregistrer. Il indique **également la façon dont les acteurs interagissent** les uns avec les autres



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

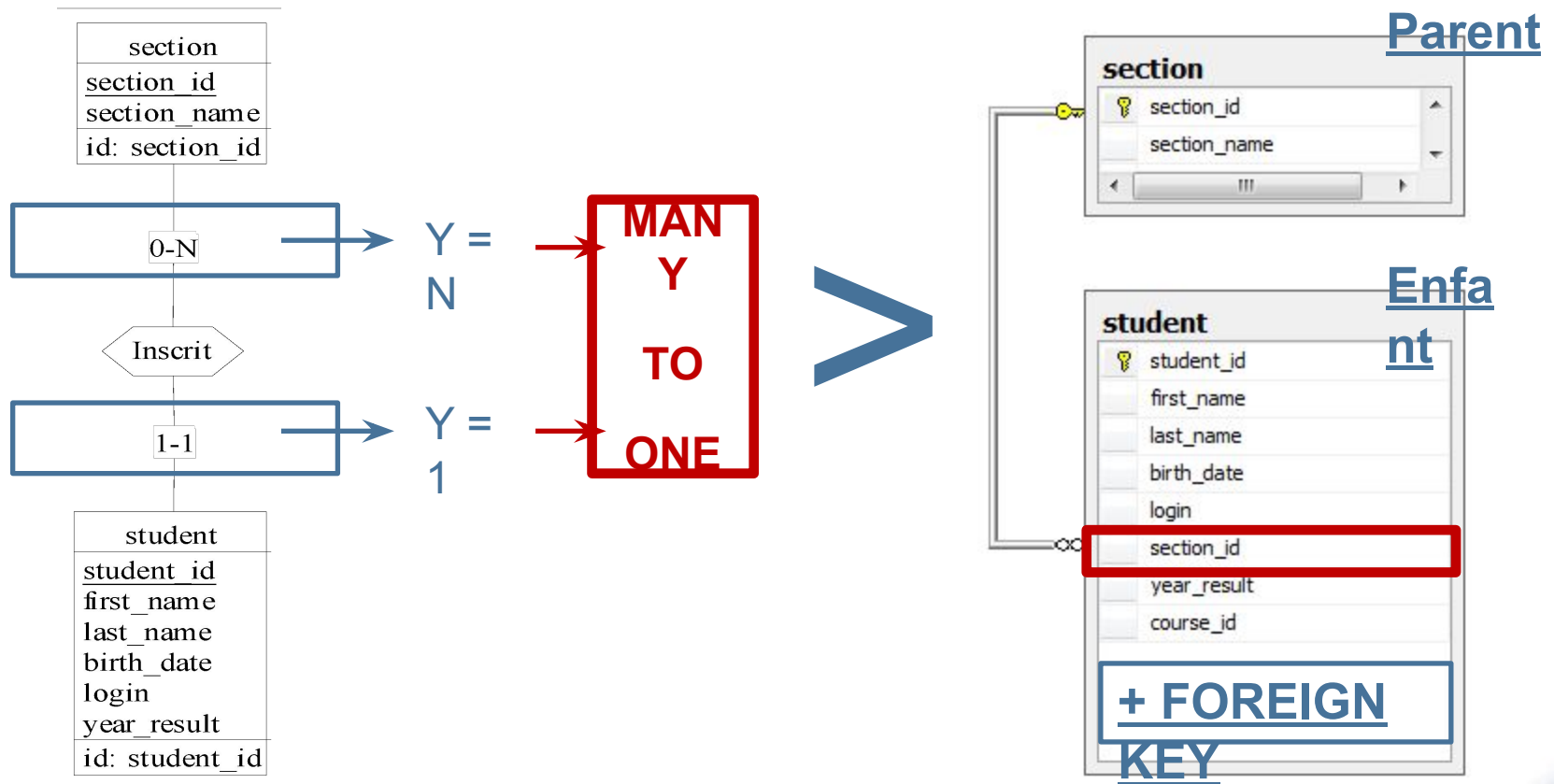
Il peut être lu aussi bien par l'analyste que par le programmeur

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

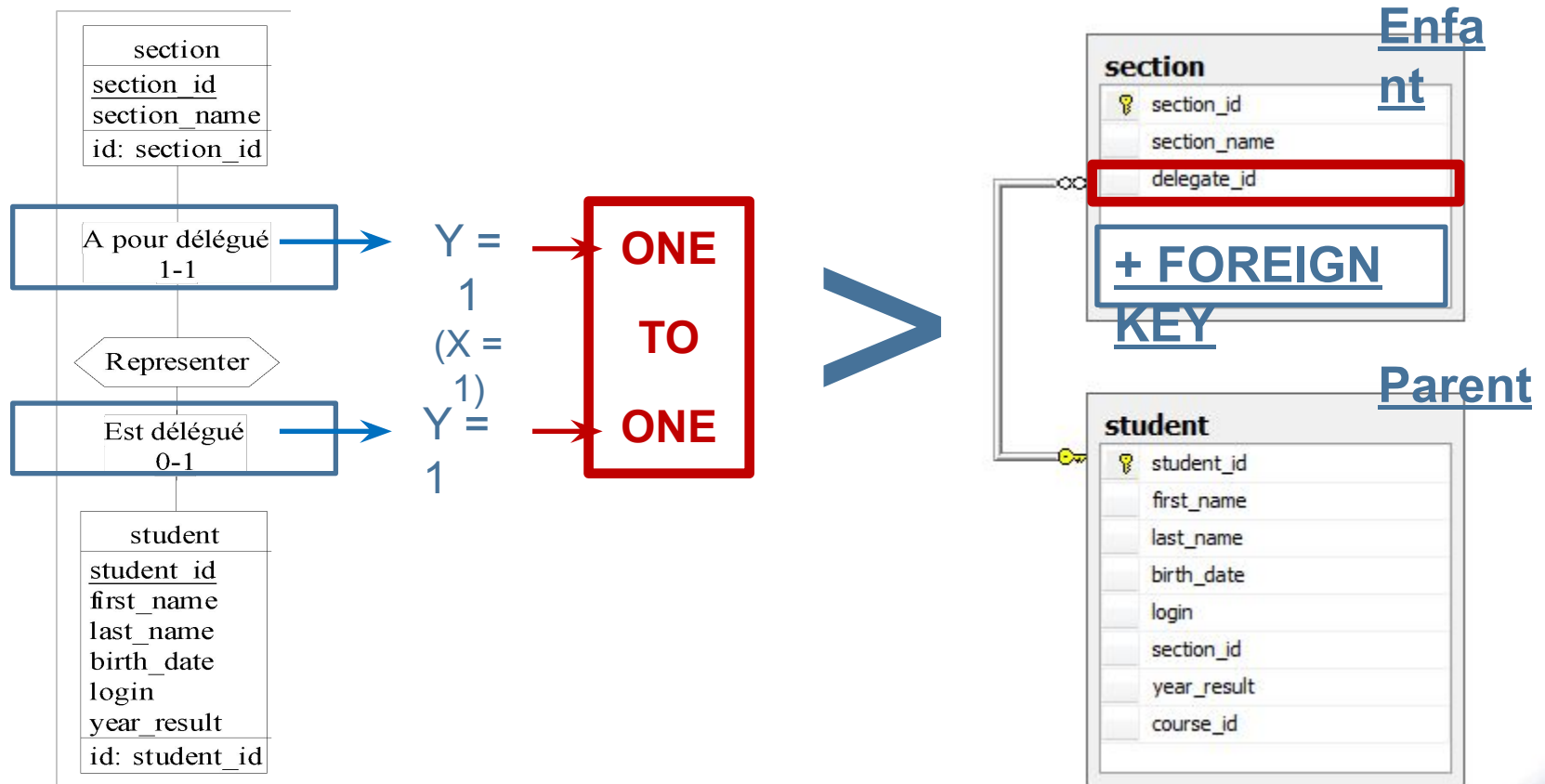
De l'analyse au relationnel

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



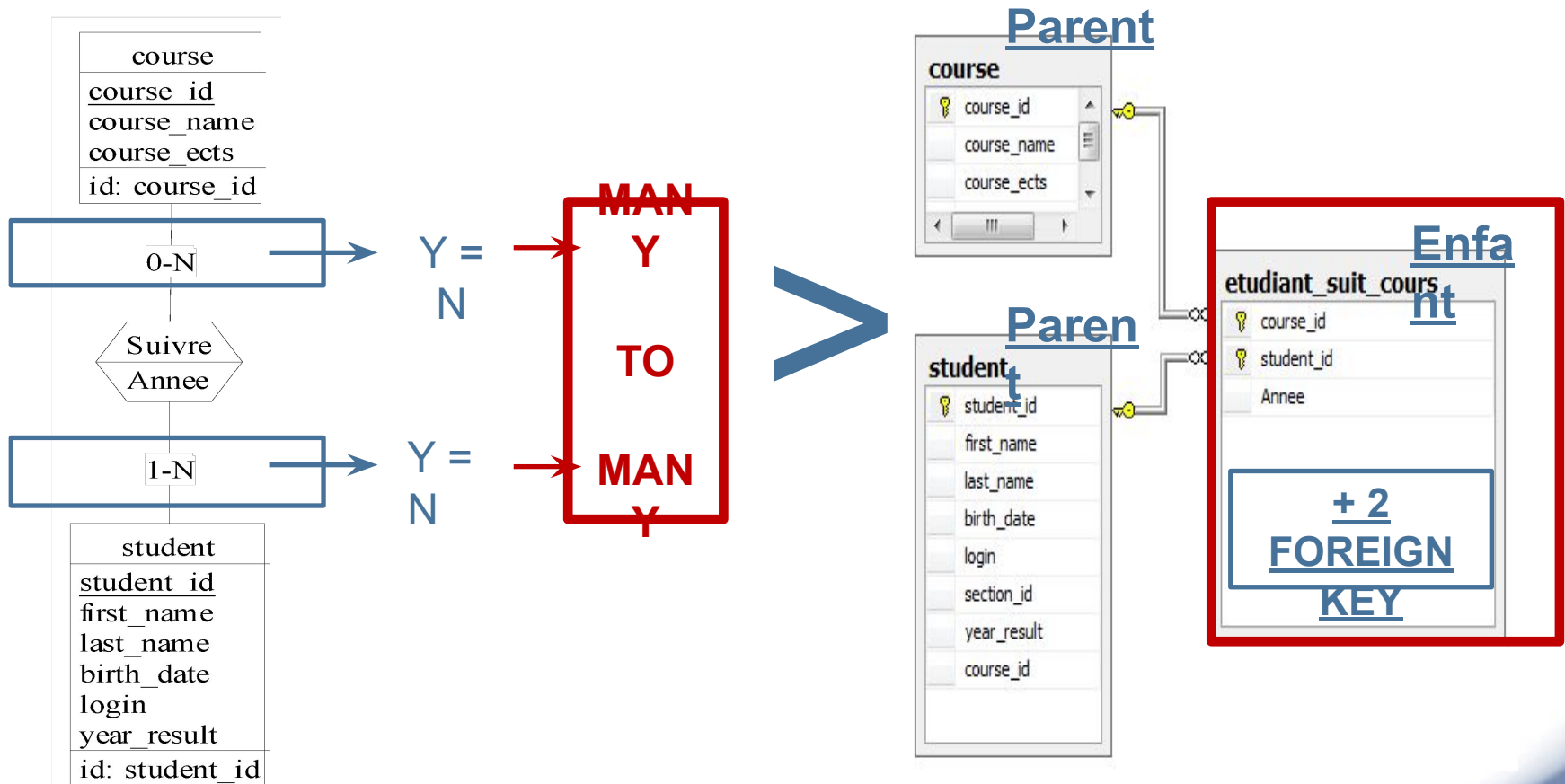
De l'analyse au relationnel

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

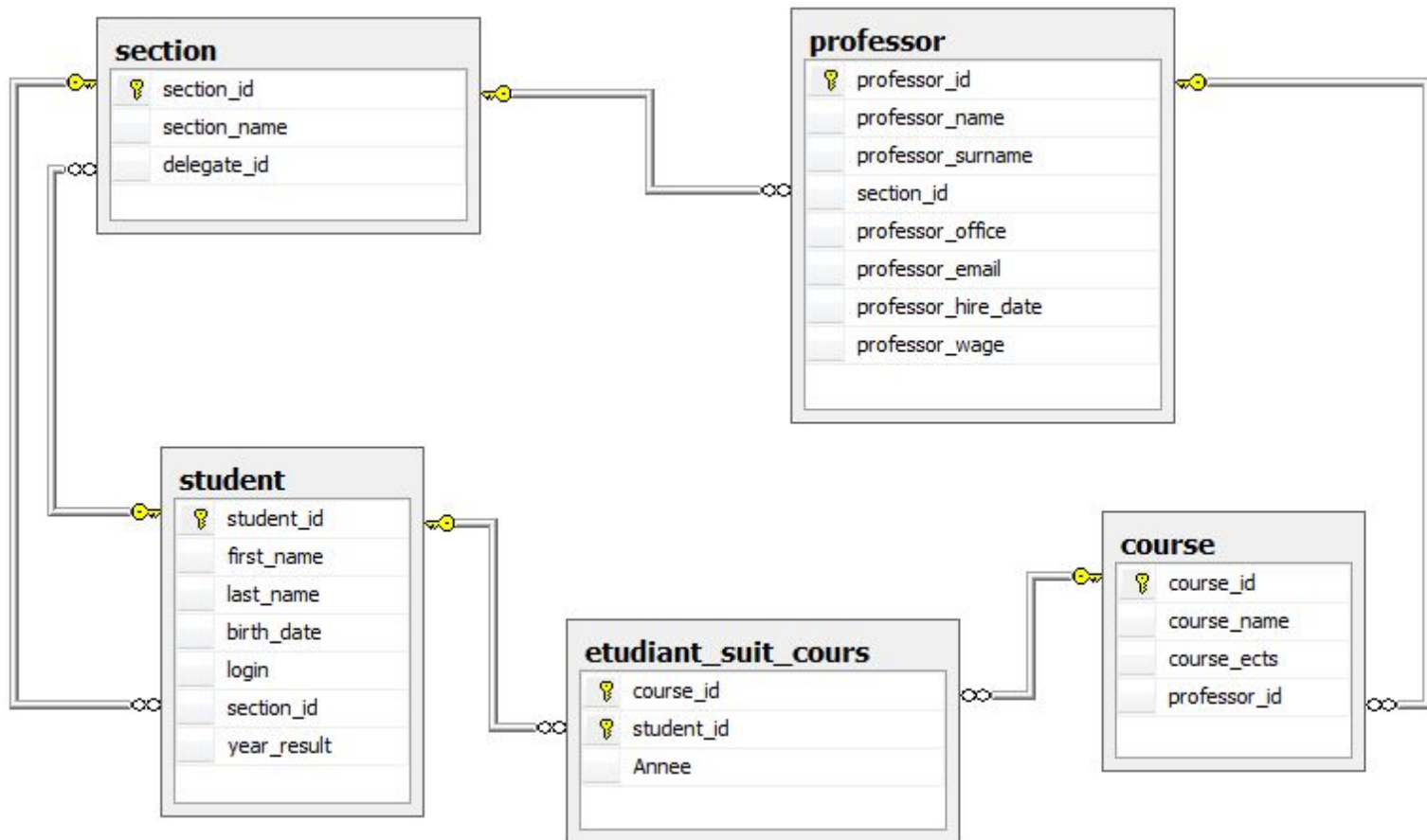


De l'analyse au relationnel

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

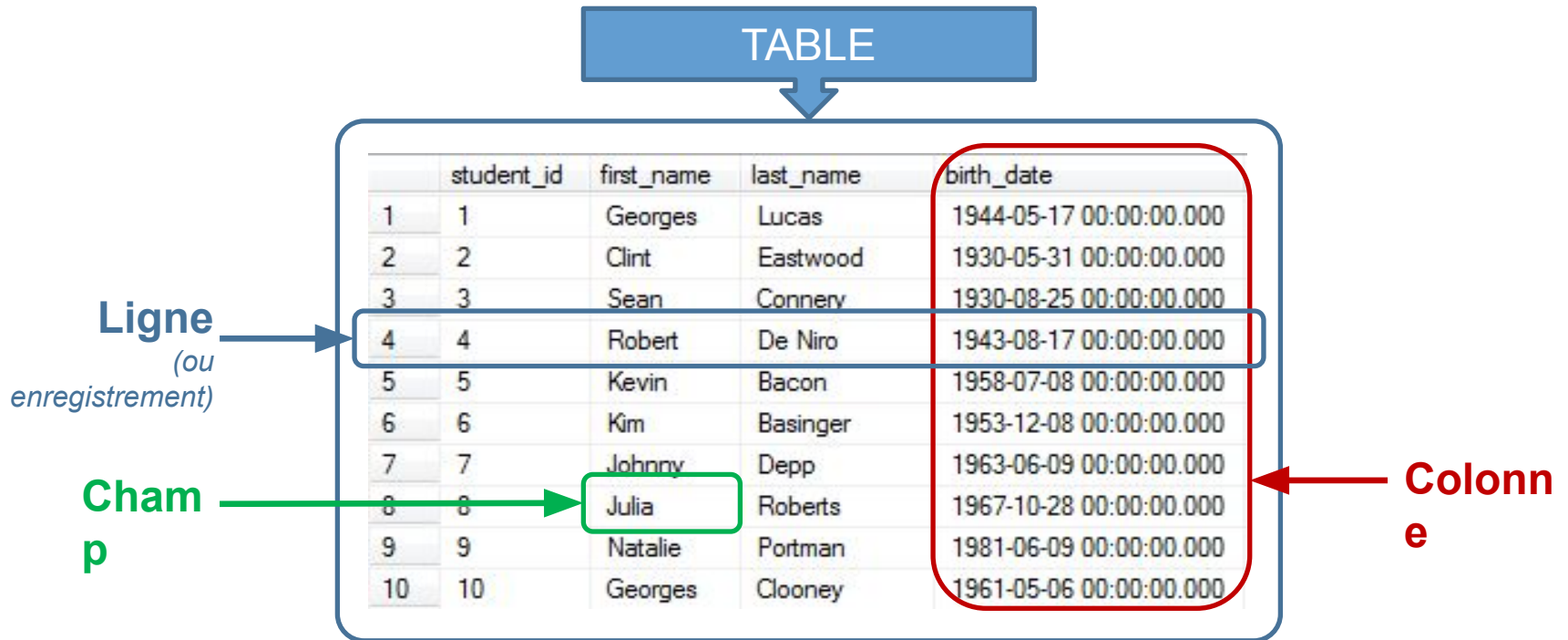


De l'analyse au relationnel



Notions de tables

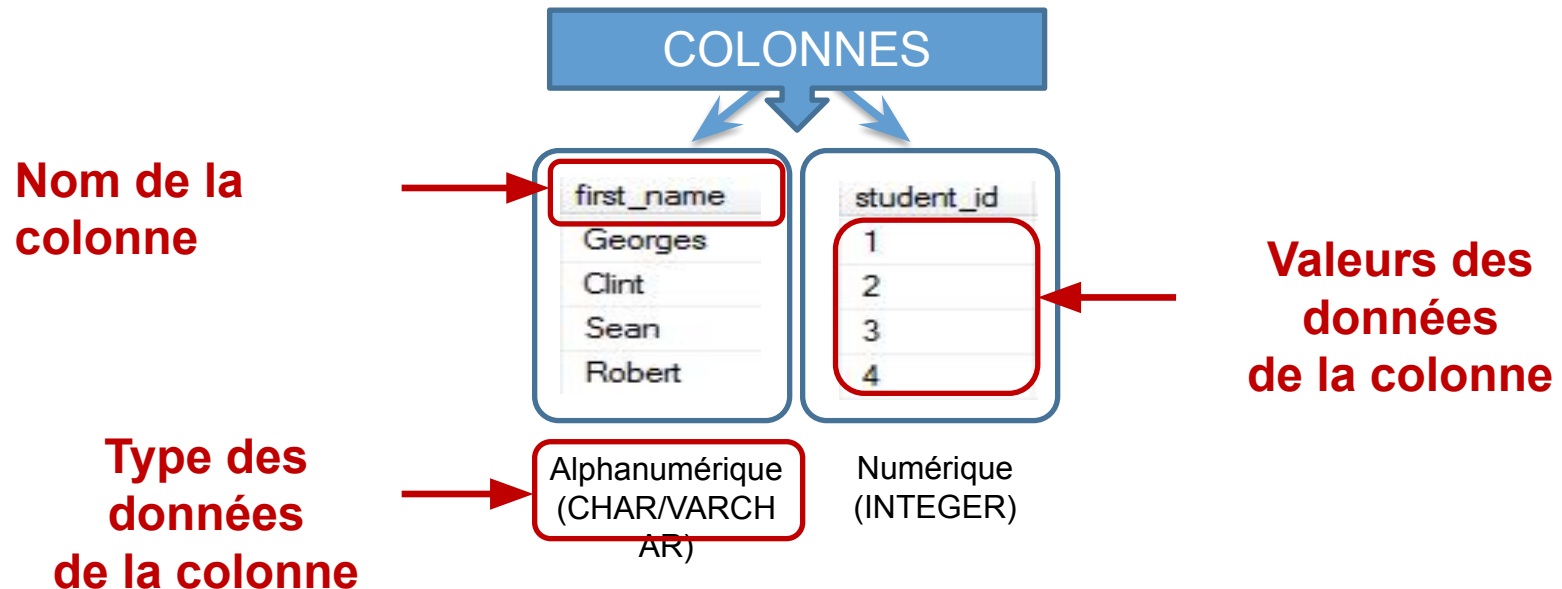
Une **Table** regroupe des **ensembles de données (d'informations)** stockés de façon permanente et décrivant chacun un acteur du système réel modélisé (**Entité**)



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

} **CLE
PRIMAIRE**

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 qu'aucune valeur n'est attribuée

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

	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	

Pas de valeur renseignée

Colonne facultative acceptant des valeurs NULL

Colonne obligatoire e NULL

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 contenant des doublons

Colonne **UNIQUE** aucun doublon

Combinaison de colonnes **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

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 étrangère 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

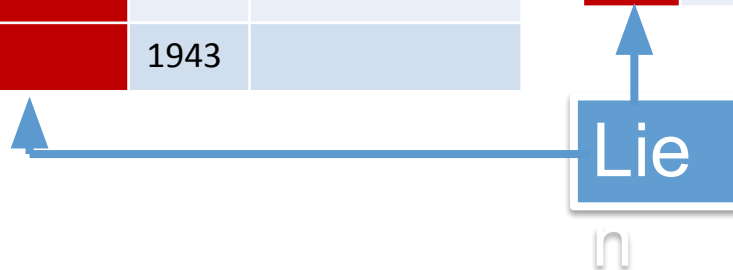
Titre	PrenomAuteur	NomAuteur	PseudoAuteur	NaissAuteur	DateLivres	ISBN
La Peste	Albert	Camus	Bebert	07/11/1913	1974	978-20703604
La Chute	Albert	Camus	Bebert	07/11/1913	1973	978-20703109
Huis-clos	Jean-Paul	Sartre	Jy-Pé	21/06/1905	1943	

Table Livres

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

Table Auteurs

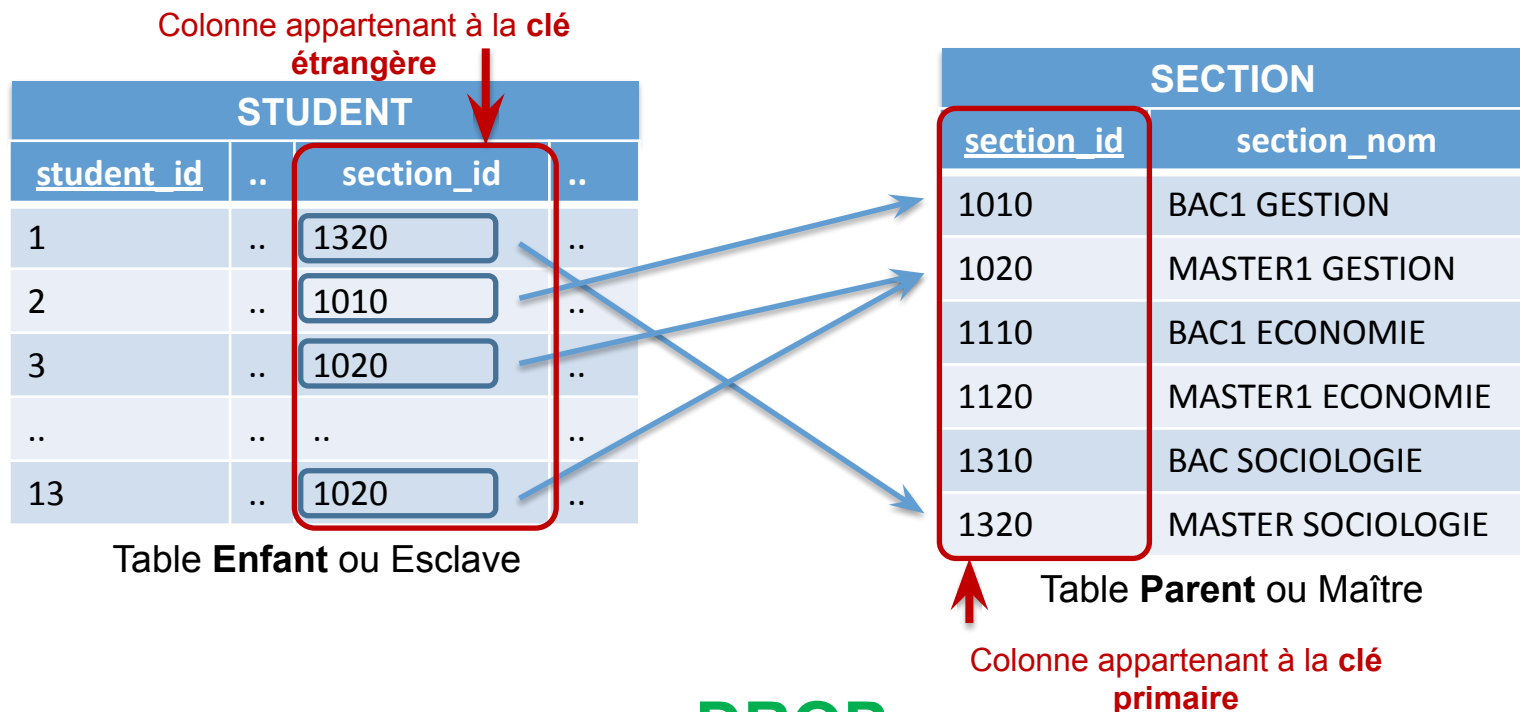
Ref	Nom	Prenom	Pseudo	DateNaiss
1	Albert	Camus	Bebert	07/11/1913
2	Jean-Paul	Sartre	Jy-Pé	21/06/1905



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

Contraintes : FOREIGN KEY

CREATE
INSERT



DROP
DELETE

Contraintes : CHECK

La contrainte **CHECK** d'une table permet de poser une condition spécifique sur les colonnes de la table, 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

$0 \leq \text{year_result} \leq 20$

**Birth_date >
01-01-1930**

Partie 2

DDL – DATA DEFINITION LANGUAGE

CREATE TABLE

AUTO-INCREMANTATION 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 timestamp,  
  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 **63 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 **DOUBLE PRECISION** (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 allant de -2.147.483.648 à +2.147.483.647
DECIMAL(x,y)	Variable	Valeurs numériques contenant « x » chiffres au total, dont « y » après la virgule (Par défaut: x = 10, y = 0)
VARCHAR(n)	0 à 126 caractères : Nbr. de caractères + 1 127 et plus caractères : Nbr. de caractères + 4	Chaîne de caractères d'une taille variable, « n » indiquant le nombre maximum de caractères.
DATE	4	Valeur chronologique allant de 4713 BC à 5874897 AD par pas de 1 jour.
BOOLEAN	1	Valeurs "true" ou "false".

CREATE TABLE

Il est conseillé d'utiliser la clause
« **GENERATED ALWAYS AS IDENTITY** »
pour activer l'auto-incrémentation à la place
des types « **SERIAL** » ou « **BIGSERIAL** ».

Types de données numériques

Nom	Taille sur le disque (octets)	Description
SMALLINT	2	Valeurs entières allant de -32.768 à 32.767
BIGINT	8	Valeurs entières allant de -9.223.372.036.854.775.808 à +9.223.372.036.854.775.807
NUMERIC(x,y)	Variable	(Idem DECIMAL)
SERIAL	4	Entier auto-incrémenté allant de 1 à 2.147.483.647
BIGSERIAL	8	Entier auto-incrémenté allant de 1 à 9.223.372.036.854.775.807

Types de données d'approximation

Nom	Taille sur le disque (octets)	Description
REAL	4	Valeurs réelles à 6 décimal
DOUBLE PRECISION	8	Valeurs réelles à 15 décimal

CREATE TABLE

Types de données textuelles

Nom	Taille sur le disque (octets)	Description
CHAR(n)	0 à 126 caractères : Nbr. de caractères + 1 127 et plus caractères : Nbr. de caractères + 4	Chaîne de caractères d'une taille fixe, « n » indiquant le nombre de caractères. Si la valeur fournie ne contient pas assez de caractères, les caractères manquant seront des "espaces blancs", à l'inverse, si la valeur contient trop de caractères, elle sera tronquée.
TEXT(n) DÉPRÉCIÉ	n (<= 65.535)	Chaîne de caractères d'une taille variable non limitée

CREATE TABLE

Types de données de dates et heures

Nom	Taille sur le disque (octets)	Description
TIMESTAMP [(n)]	8	Valeur chronologique allant de 4713 BC à 294276 AD par pas de 1 millisecond.
TIME [(n)]	8	Valeur chronologique allant de 00:00:00 à 24:00:00 par pas de 1 millisecond.
INTERVAL [<i>fields</i>] [(n)]	16	Valeur chronologique allant de -178.000.000 d'années à 178.000.000 d'années par pas de 1 millisecond.

Les fields correspondent à des périodes chronologique de taille fixe.

Voici les fields existant :

YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, YEAR TO MONTH, DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND, HOUR TO MINUTE, HOUR TO SECOND, MINUTE TO SECOND.

Pour plus d'informations concernant les différents type :

<https://www.postgresql.org/docs/12/datatype.html>

AUTO INCREMENTATION 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 int GENERATED ALWAYS AS IDENTITY,  
  first_name varchar(50),  
  last_name varchar(50) DEFAULT 'Smith',  
  birth_date timestamp,  
  login varchar(50),  
  section_id int,  
  year_result int DEFAULT 0,  
  course_id varchar(6)  
);
```

AUTO INCREMENTATION et DEFAULT

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

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

ATTENTION : Sous PostgreSQL, il est possible (**en utilisant du code**) de rajouter l'auto-incrémentation d'une colonne lorsque la table est déjà créée, il continuera l'incrémentation là où elle s'arrêtait.

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

Contraintes

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  TYPE_CONTRAINTE (colonne_concernée)  
);
```

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

```
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 timestamp,  
    login varchar(50),  
    section_id int,  
    year_result int,  
    course_id varchar(6) NOT NULL  
);
```

*Les contraintes **NOT NULL** sont les seules contrainte qui **ne sont pas nommées**, elles sont incluses dans la définition de leur colonnes*

Contraintes : UNIQUE

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



Toute contrainte aura un nom dans le système. Ici, le nom de la première colonne spécifiée dans la création de l'index

Contraintes : PRIMARY KEY

Toute contrainte aura un nom dans le système. Ici , PRIMARY.

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

Version 1

Contraintes : PRIMARY KEY

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

Version 2

Contraintes : FOREIGN KEY

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  CONSTRAINT FK_nom_contrainte FOREIGN  
  KEY(colonne_concernée)  
    REFERENCES table_référencée (colonne_référencée)  
);
```

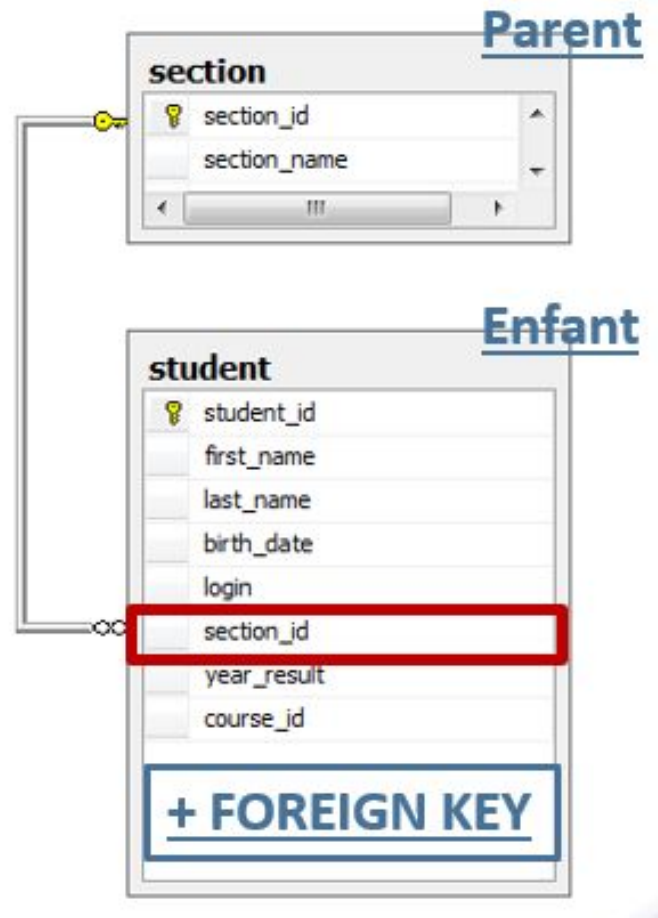
Le nom de la FK doit être unique !!!

Contraintes : FOREIGN KEY

```
CREATE TABLE student (  
    student_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    birth_date timestamp,  
    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)  
);
```

Première notation possible

Contraintes : FOREIGN KEY



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

Ex: une section disparaît, l'étudiant dans cette section disparaît /\ DANGER

- L'action « **SET NULL** » met à NULL les valeurs correspondantes dans la table enfant (possible uniquement si la colonne accepte les valeurs NULL...)

Ex: une section disparaît, le champ passe à null dans l'étudiant

- Si rien n'est spécifié, l'action par défaut est « **NO ACTION** » ou « **RESTRICT** » qui lève une erreur

```
CREATE TABLE nom_table (  
  nom_colonne1 TYPE,  
  nom_colonne2 TYPE,  
  CONSTRAINT nom_contrainte FOREIGN  
  KEY(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 timestamp,  
    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)  
    ON DELETE SET NULL ON UPDATE CASCADE  
);
```

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 timestamp,  
    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  
        (DATE_PART('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...*

Modifier une colonne

```
ALTER TABLE nom_table ALTER COLUMN nom_colonne  
NOUVEAU_TYPE ...;
```

```
ALTER TABLE nom_table RENAME COLUMN old_colonne TO  
new_colonne;
```

| # changer le type

| ALTER TABLE student ALTER COLUMN langue char(2);

| # renommer une colonne

| ALTER TABLE student RENAME COLUMN langue TO langage;

ALTER TABLE

Ajouter une colonne

```
ALTER TABLE nom_table ADD COLUMN nom_colonne TYPE  
contraintes;
```

```
ALTER TABLE student ADD COLUMN langue varchar(10);
```

Supprimer une colonne

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

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

ALTER TABLE

Ajouter une contrainte

```
ALTER TABLE nom_table ADD CONSTRAINT nom_contrainte FOREIGN  
KEY (colonne) ...;
```

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

Supprimer une contrainte

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

```
ALTER TABLE student DROP CONSTRAINT fk_student_section;
```


ALTER TABLE

Modifier une contrainte

```
ALTER TABLE nom_table DROP CONSTRAINT nom_contrainte;  
ALTER TABLE nom_table ADD CONSTRAINT nom_contrainte FOREIGN  
KEY (colonne) ...;
```

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

```
ALTER TABLE nom_table ALTER COLUMN nom_colone SET [NOT] NULL;
```

```
ALTER TABLE student ALTER COLUMN year_result SET NULL;
```

TRUNCATE TABLE

L'ordre DDL « **TRUNCATE TABLE** » permet de vider une table de son contenu. Par défaut, cet ordre ne réinitialise pas le compteur auto-incrémenté. Pour effectuer cette réinitialisation, il est nécessaire d'ajoutez l'option **RESTART IDENTITY**.

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 [ RESTART IDENTITY | CONTINUE IDENTITY ]
```

```
TRUNCATE TABLE student RESTART IDENTITY;
```

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",  
       birth_date "Date de naissance"  
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 PostgreSQL**, les alias doivent être accompagnés de guillemets doubles s'ils contiennent des espaces blancs, des caractères spéciaux tels que le '@' ou encore la volonté de garder les majuscules.

Prénom	Nom de famille	Section	Résultat annuel	Date de naissance
Georges	Lucas	1320	10	1944-05-17 00:00:00
Clint	Eastwood	1010	4	1930-05-31 00:00:00
Sean	Connery	1020	12	1930-08-25 00:00:00
Robert	De Niro	1110	2	1943-08-17 00:00:00

La clause « SELECT » : Opérations Arithmétiques

```
SELECT first_name, year_result,  
       (year_result/20.0)*100 AS "Nouveau Résultat"  
FROM student;
```

Opérateurs autorisés

Opérateur	Signification
/	Division
*	Multiplication
+	Addition
-	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
Andy	19	95

Attention : Division entière si le diviseur et dividende sont des entiers.

La clause « SELECT » :

Concaténation

CONCAT - ||

```
SELECT CONCAT(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 » : DISTINCT

```
SELECT DISTINCT first_name  
FROM student;
```

Sans
DISTINCT

first_name
Alyssa
Andy
Bruce
Clint
David
Georges
Georges
Halle
Jennifer
Johnny
Julia

Avec
DISTINCT

first_name	1
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	▼ 1	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	▼ 1	year_result
Tom		4
Sophie		6
Shannen		2
Sean		12
Sarah		7
Sandra		2
Robert		3
Reese		7
Natalie		4

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 NOW() AS 'Date du jour', 'Vive le SQL';
```

Date du jour	Vive le SQL
2019-04-05	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
=	Est strictement égal
>	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	Garner	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
Julia	Roberts	1967-10-28 00:00:00
Georges	Clooney	1961-05-06 00:00:00
Tom	Cruise	1962-07-03 00:00:00
Sophie	Marceau	1966-11-17 00:00:00
Michael J.	Fox	1969-06-20 00:00:00
Sandra	Bullock	1964-07-26 00:00:00
Keanu	Reeves	1964-09-02 00:00:00
Halle	Berry	1966-08-14 00:00:00

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	Garner	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 A de l'importance**

student_id	first_name	last_name	year_result
13	Tom	Cruise	4
18	Jennifer	Garner	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	Garner	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
13	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	Garner	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 PostgreSQL, si aucune parenthèse n'est utilisée, l'ordre de précedence suivant est appliqué

Ordre	Associativité	Opérateurs évalués
1	Gauche	Exposant (^)
2	Gauche	Multiplication, Division, Modulo (* / %)
3	Gauche	Addition, Soustraction (+ -)
4		BETWEEN, IN, LIKE, SIMILAR
5		Opérateurs de comparaisons (= , > , < , != , ...)
6		IS , IS NULL, IS NOT NULL
7	Droite	NOT
8	Gauche	AND
9	Gauche	OR

<https://www.postgresql.org/docs/12/sql-syntax-lexical.html#SQL-PRECEDENCE>

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	▲ 1	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
13	Tom	Cruise		4

Limiter et ordonner : « ORDER BY »

```
SELECT section_id,  
       CONCAT(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	1	Nomcomplet	2
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	
1020		Reese Witherspoon	

Attention : Si le nom de l'alias comporte au minimum un espace blanc, un caractère spécial ou une majuscule :

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

- 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	DATE_PART, POSITION, CHAR_LENGTH, ABS, MOD
chaîne de caractères	SUBSTRING, UPPER, LOWER, OVERLAY, TRIM
timestamp	NOW

Les fonctions : **CAST** ou **::**

CAST (*valeur_à_convertir* **AS** NOUVEAU_TYPE)
valeur_à_convertir **::**NOUVEAU_TYPE

La fonction « **CAST** » et l'opérateur « **::** » attendent 2 paramètres en entrée et **renvoient une valeur correspondant au premier paramètre dans le type spécifié par le deuxième**

```
SELECT CAST(birth_date AS char(10)) AS "Date de naissance"  
FROM student;  
SELECT birth_date::char(10) AS "Date de naissance"  
FROM student;
```

Date de naissance

1944-05-17

1930-05-31

1930-08-25

1943-08-17

1958-07-08

1953-12-08

Les fonctions : TO_CHAR

TO_CHAR(valeur_à_convertir , FORMAT)

La fonction « **TO_CHAR** » attend 2 paramètres en entrée et **renvoie une valeur correspondant au premier paramètre dans le format spécifié par le deuxième.**

```
SELECT TO_CHAR(birth_date, 'DD-MM-YYYY') AS "Date de naissance"  
FROM student;
```

Format	Description
YYYY	Year as a numeric, 4-digit value
YY	Year as a numeric, 2-digit value
MM	Numeric month name (01 to 12)
MONTH	Month name in full (January to December)
DD	Day of the month as a numeric value (01 to 31)
D	Day of the week as a numeric value (1 to 7 – Sunday to Saturday)
DI	Day of the week as a numeric value (1 to 7 – Monday to Sunday)
HH	Hour (00 to 12)
HH12	Hour (00 to 12)
HH24	Hour (00 to 23)
MI	Minutes (00 to 59)
SS	Seconds (00 to 59)

Date de naissance

17-05-1944

31-05-1930

25-08-1930

17-08-1943

08 07 1958

Et bien d'autres formats sur :
<https://www.postgresql.org/docs/12/functions-formatting.html>

Les fonctions : **CURRENT_DATE/CURRENT_TIME**

```
SELECT TO_CHAR(CURRENT_TIMESTAMP, 'YYYY-MM-DD HH24:MI:SS') AS "Date du jour",  
       TO_CHAR(NOW(), 'Mon DD YYYY HH:MI:SS AM') AS "Date du jour formatée",  
       CURRENT_DATE AS "Date uniquement", CURRENT_TIME AS "Heure uniquement"
```

Sous PostgreSQL, la fonction « **NOW** » et la variable globale « **CURRENT_TIMESTAMP** » renvoie la date et l'heure actuelles, tandis que les variables globales « **CURRENT_DATE** » et « **CURRENT_TIME** » renvoient respectivement uniquement la date actuelle ou uniquement l'heure actuelle.

Type retourné : DATE , TIME, TIMESTAMP

Date du jour	Date du jour formatée	Date uniquement	Heure uniquement
2019-04-05 17:18:31	Apr 05 2019 05:18:31 PM	2019-04-05	17:18:31

Les fonctions : **POSITION**

POSITION (*chaine_de_caractères_recherchée* IN *valeur_à_évaluer*)

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

Type retourné : NOMBRE

```
SELECT POSITION('i' IN 'Kim Basinger') AS "Position du premier i", POSITION('08' IN  
'Basinger 08/12/1953') AS "Position du 08",  
POSITION('y' IN 'Kim Basinger') AS "pas de y",  
POSITION(' ' IN 'Kim Basinger') AS "espace"
```

Position du premier i	Position du 08	pas de y	espace
2	10	0	4

Les fonctions : **LENGTH**

CHAR_LENGTH

(*chaîne_de_caractères_à_mesurer*)

Les fonctions « **LENGTH** », « **CHAR_LENGTH** » et « **CHARACTER_LENGTH** » renvoient le nombre de lettres composant une chaîne de caractères donnée, espaces blancs compris.

Mais attention, « **BIT_LENGTH** » et « **OCTET_LENGTH** » renvoient respectivement la taille en bits ou en bytes. (si caractère accentué alors un caractère en plus)

Type retourné : NOMBRE

```
SELECT CHAR_LENGTH('Kim Basinger') AS "Longueur de la chaîne de caractères"
```

Longueur de la chaîne de caractères

12

Les fonctions : **ABS** / **@**

ABS

(*nombre*)

@ *nombre*

La fonction « **ABS** » et l'opérateur « **@** » renvoient la valeur absolue du nombre passé en paramètre.

ATTENTION : un espace blanc est nécessaire entre l'opérateur « **@** » et le nombre.

Type retourné : NOMBRE

```
SELECT ABS(-1.0) AS "val1", ABS(0.0) AS "val2", ABS(1.1) AS "val3";  
SELECT @ -1.0 AS "val1", @ 0.0 AS "val2", @ 1.1 AS "val3";
```

val1	val2	val3
1.0	0.0	1.1

Les fonctions : MOD / %

MOD(dividende,
diviseur)

dividende % diviseur

La fonction « **MOD** » et l'opérateur « % », 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 38.0 / 5 AS "Division", 38/5 AS "Division Entière", 38 % 5 AS "Reste";  
SELECT 38.0 / 5 AS "Division", 38/5 AS "Division Entière", MOD (38, 5) AS "Reste";
```

$$\begin{array}{r} 38,0 \\ 5 \overline{) 38,0} \\ \underline{35} \\ 30 \\ \underline{30} \\ 0 \end{array}$$

$$\begin{array}{r} 5 \\ \hline 7,6 \end{array}$$

	Division numeric	Division Entière... integer	Reste integer
1	7.6000000000000000	7	3

Les fonctions : **SUBSTRING**

SUBSTRING (*chaine_de_caractères* **FROM** *position_départ* **FOR** *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' FROM 4 FOR 3)
AS "Caractères 4, 5 et 6";
SELECT last_name, SUBSTRING(first_name FROM 1 FOR 1)
AS "Initiale du prénom"
FROM student;
```

Caractères 4, 5 et 6
ing

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

Les fonctions : **LEFT** et **RIGHT**

LEFT (*chaine_de_caractères, nombre_caractères*)
RIGHT (*chaine_de_caractères, nombre_caractères*)

La fonction « **LEFT** » renvoie une chaîne de caractère du nombre de caractère souhaité à partir de la gauche, et « **RIGHT** » à partir de la droite.

Type retourné : CHAÎNE DE CARACTÈRES

```
SELECT LEFT('Basinger', 4) AS "4 premiers caractères";  
SELECT last_name, RIGHT(first_name, 1)  
AS "Dernière lettre du prénom"  
FROM student;
```

4 premiers caractères
Basi

last_name	Dernière lettre du prénom
Lucas	s
Eastwood	t
Connery	n
De Niro	t
Bacon	n
Basinger	m
Denn	v

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

UPPER

(*chaine_de_caractères*)

LOWER

(*chaine_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) AS "Nom de famille",  
       LOWER(first_name) AS "Prénom"  
FROM student  
WHERE LOWER(first_name) LIKE LOWER('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 ', ' ', '') AS "Sans espace",  
REPLACE('1111000010101010', '1', '0') AS "Sans 1";
```

Sans espace	Sans 1
KimBasinger	0000000000000000

Les fonctions : **LTRIM**, **RTRIM** et **TRIM**

LTRIM (*chaine_de_caractères*)
TRIM(*chaine_de_caractères*)

RTRIM (*chaine_de_caractères*)

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

Type retourné : CHAÎNE DE CARACTÈRES

```
SELECT LTRIM(' Kim Basinger ') AS "LTRIM",  
RTRIM(' Kim Basinger ') AS "RTRIM",  
TRIM(' Kim Basinger ') AS "TRIM";
```

LTRIM

Kim Basinger

RTRIM

Kim Basinger

TRIM

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
COUNT	Nombre total de valeurs contenues dans la table/colonne
MAX	Valeur numérique la plus élevée
MIN	Plus petite valeur numérique disponible
SUM	Somme de l'ensemble des valeurs de la colonne
AVG	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 (BIGINT)

```
SELECT COUNT(*) AS "Total des lignes",  
COUNT(first_name) AS "Total des prénoms",  
COUNT(DISTINCT first_name) AS "Total des prénoms sans doublons"  
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) AS "Résultat le plus élevé", MIN(year_result*5) AS  
"Pourcentage le plus faible", MAX(CHAR_LENGTH(last_name)) AS "Taille du  
nom le plus long"  
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) AS "Somme des résultats annuels",  
SUM(year_result)::FLOAT/COUNT(*) AS "Moyenne générale"  
FROM student;
```

Somme des résultats annuels	Moyenne générale
219	8.7600

Les fonctions : **AVG**

AVG (colonne)

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

Type retourné : NOMBRE

Si AVG sur une valeur de type DOUBLE PRECISION, le résultat final est DOUBLE PRECISION,

~~pour une valeur INTEGER, le résultat sera NUMERIC.~~

```
SELECT AVG(year_result) AS "Moyenne générale",  
AVG(DATE_PART('YEAR',CURRENT_DATE)-DATE_PART('YEAR', birth_date))  
AS "Moyenne d'âge"  
FROM student;
```

Moyenne générale	Moyenne d'âge
8.7600	58.6400

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

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	.Johnnv	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 last_name, first_name,  
CASE section_id  
  WHEN 1010 THEN 'BSc Management'  
  WHEN 1320 THEN 'MA Sociology'  
  ELSE NULL  
END AS "Nom de section"  
FROM student;
```

last_name	first_name	Nom de section
Lucas	Georges	MA Sociology
Eastwood	Clint	BSc Management
Connery	Sean	NULL
De Niro	Robert	NULL
Bacon	Kevin	NULL
Basinger	Kim	NULL
Depp	Johnny	NULL
Roberts	Julia	NULL
Portman	Natalie	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) AS "Résultats  
sauf les 7/20"  
FROM student  
ORDER BY first_name
```

```
SELECT last_name, first_name, year_result,  
CASE year_result  
    WHEN 7 THEN NULL  
    ELSE year_result  
END AS "Résultats sauf les 7/20"  
FROM student  
ORDER BY first_name
```

last_name	first_name	year_result	Résultats sauf les 7/20
Milano	Alyssa	7	NULL
Garcia	Andy	19	19
Willis	Bruce	6	6
Eastwood	Clint	4	4
Morse	David	2	2
Lucas	Georges	10	10
Clooney	Georges	4	4
Berry	Halle	18	18
Garner	Jennifer	18	18

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
« BUDGET
S »

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 : Résumé

Concaténation

CONCAT	concaténer	CONCAT(nom_col1, ' ', nom_col2)
--------	------------	---------------------------------

Conversion

CONVERT	convertir	CONVERT (valeur_à_convertir, NOUVEAU_TYPE)
CAST	convertir	CAST(valeur_à_convertir AS NOUVEAU TYPE)

Date

CURDATE	recupérer la date	CURDATE()
CURTIME	recupérer l'heure	CURTIME()
DATE_FORMAT	extraire une partie d'une date	DATE_FORMAT (date_traitée, partie_de_date_à_extraire)
CURRENT_TIME STAMP	revoie la date et l'heure	CURRENT_TIMESTAMP()
DATEDIFF	soustraire des dates	DATEDIFF(startdate , enddate)
DAY	retourne le jour	DAY (date)
MONTH	retrouve le mois	MONTH (date)
YEAR	retourne l'année	YEAR(date)

Les fonctions : Résumé

Chaines de caractères

LOCATE	recupérer la position d'un car.	LOCATE (chaine_de_caractères_recherchée, valeur_à_évaluer)
CHAR_LENGTH	recupérer le nombre de car.	CHAR_LENGTH (chaine_de_caractères_à_mesurer)
LENGTH	recupérer le nombre de car. en bytes	LENGTH (chaine_de_caractères_à_mesurer)
SUBSTRING	couper une chaine	SUBSTRING (chaine_de_caractères, position_départ, nombre_caractères)
UPPER	mettre en MAJUSCULE	UPPER (chaine_de_caractères)
LOWER	mettre en minuscule	LOWER (chaine_de_caractères)
REPLACE	remplacer des car.	REPLACE (chaine_de_caractères_traitée, caract_à_remplacer, nouveau_caract)
LTRIM	retirer les espaces blancs avant	LTRIM (chaine_de_caractères)
RTRIM	retirer les espaces blancs après	RTRIM (chaine_de_caractères)
TRIM	retirer les espaces blancs avant et après	TRIM(chaine_de_caractères)
LEFT	recupérer le X car. à gauche	LEFT(nom_col, X)
RIGHT	recupérer le X car. à droite	RIGHT(nom_col, X)

Les fonctions : Résumé

Mathématiques

ABS	récupérer la valeur absolue	ABS (nombre)
%	récupérer le modulo	dividende % diviseur
COUNT	renvoyer le nombre de champs	COUNT (*)
MAX	renvoyer le nombre max	MAX (colonne)
MIN	renvoyer le nombre min	MIN (colonne)
SUM	renvoyer la somme	SUM (colonne)
AVG	renvoyer la moyenne	AVG (colonne)

Les fonctions : Résumé

Structures conditionnelles

CASE	modifier l'affichage	CASE WHEN colonne_à_évaluer + condition THEN valeur1 ... ELSE valeur_par_défaut END
		CASE colonne_à_évaluer WHEN valeur_de_comparaison1 THEN valeur1 ... ELSE valeur_par_défaut END
NULLIF	mettre null si	NULLIF (colonne_considerée, valeur_à_mettre_à_NULL)
COALESCE	renvoyer 1 ^{er} valeur non NULL	COALESCE (colonne1, colonne2, ..., colonneN)

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 :

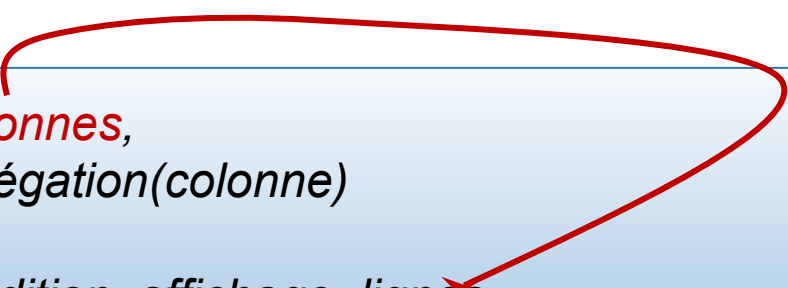
- **Parfait (P)** : vous avez parfaitement compris cette notion et vous vous sentez à votre aise
- **Satisfaisant (S)** : vous avez compris de quoi il s'agit mais la pratique vous manque
- **Vague (V)** : vous savez de quoi il s'agit, mais cela reste un peu vague dans votre esprit. Une explication supplémentaire 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) AS "Moyenne par section"  
FROM student  
GROUP BY section_id
```

section_id	Moyenne par section
1010	4.0000
1020	7.4286
1110	8.0000
1120	17.0000
1310	11.0000
1320	10.0000

Sans le « **GROUP BY** », le système affiche la moyenne générale mais avec la dernière section ?!?

section_id	Moyenne par section
1320	8.7600

GROUP BY : + WHERE

```
SELECT section_id, AVG(year_result) AS "Moyenne par section"  
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	2.0000
1020	6.6667
1110	7.0000
1120	16.0000
1310	19.0000
1320	10.0000

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) AS "Moyenne par section"
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	2.0000
1020	6.6667
1110	7.0000
1120	16.0000
1310	19.0000
1320	10.0000

WHERE + GROUP BY

section_id	Moyennes >= 10
1120	16.0000
1310	19.0000
1320	10.0000

WHERE + GROUP BY
+ HAVING

GROUP BY : colonnes multiples

```
SELECT section_id, course_id, AVG(year_result) AS "Moyenne"
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** ». PostgreSQL génère une erreur dans le cas contraire.*
- 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	2.0000
1010	EG2210	4.6667
1020	EG1020	7.0000
1020	EG2110	7.0000
1020	EG2210	10.0000

GROUP BY : ROLLUP et CUBE

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

```
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, SUM(year_result) AS "Somme"  
FROM student  
WHERE section_id IN(1010, 1020)  
GROUP BY ROLLUP (section_id, course_id);
```

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

Sans « ROLLUP »

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

Total section
1010

Total section
1020
Total général

Avec « ROLLUP »

GROUP BY : CUBE

```
SELECT section_id, course_id, SUM(year_result) AS "Somme"
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

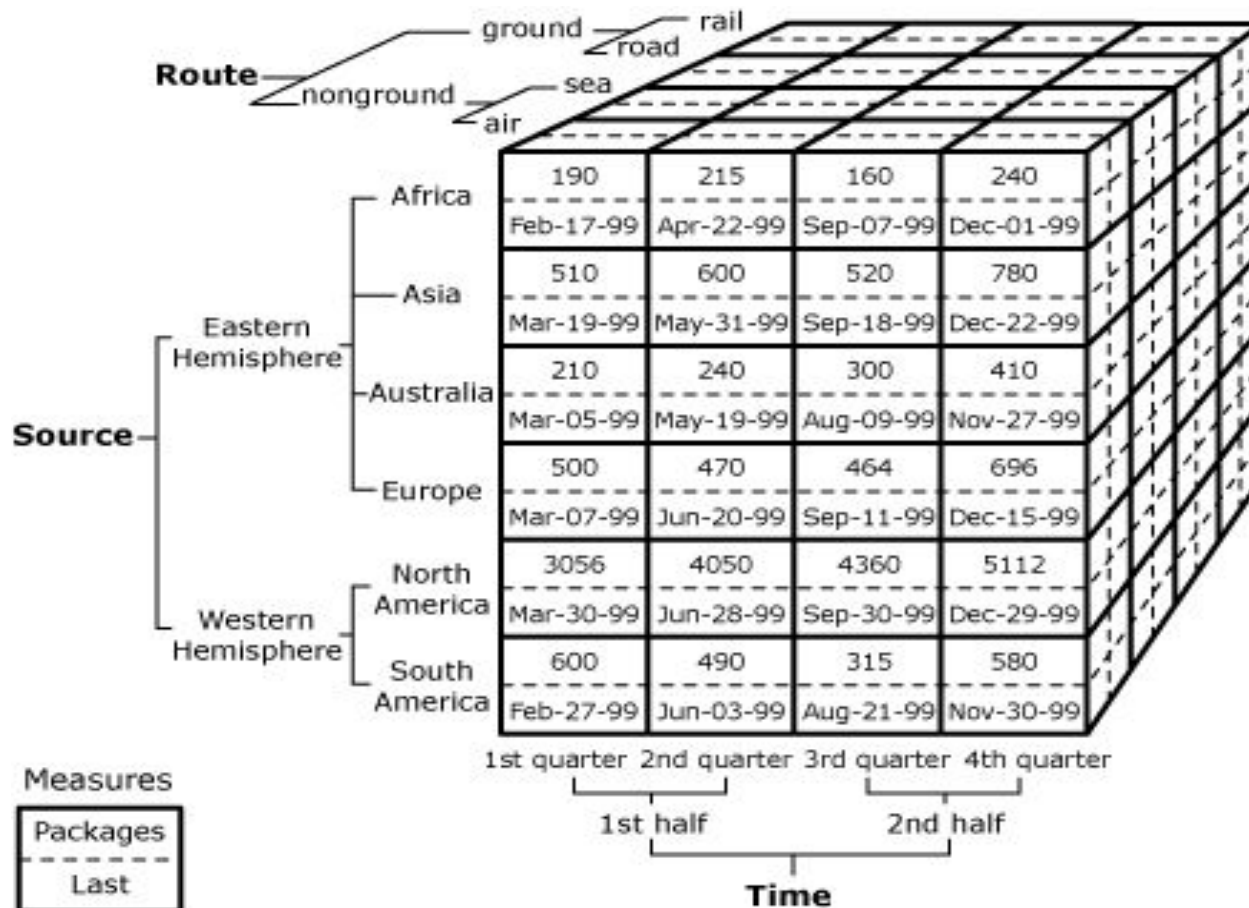
Total section
1010

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
```

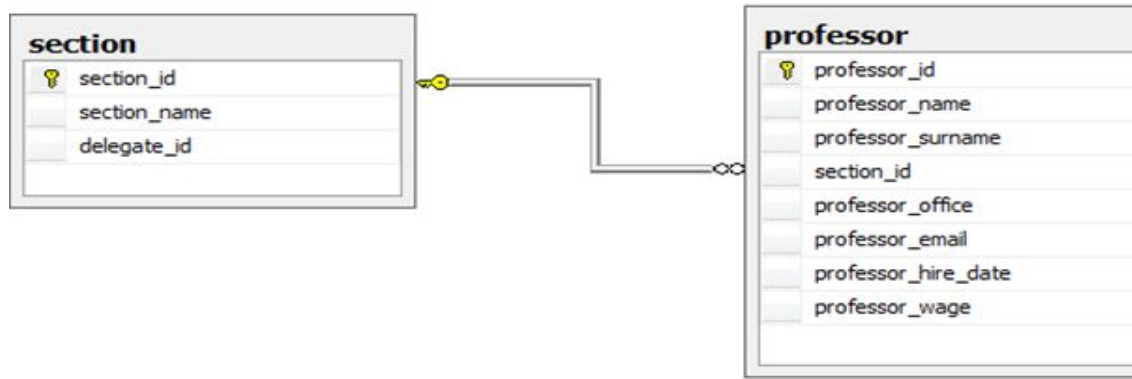
Comparison des colonnes des tables entre elles ...

Jointures verticales

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

Comparison du résultat de deux requêtes entre eux

Jointures horizontales

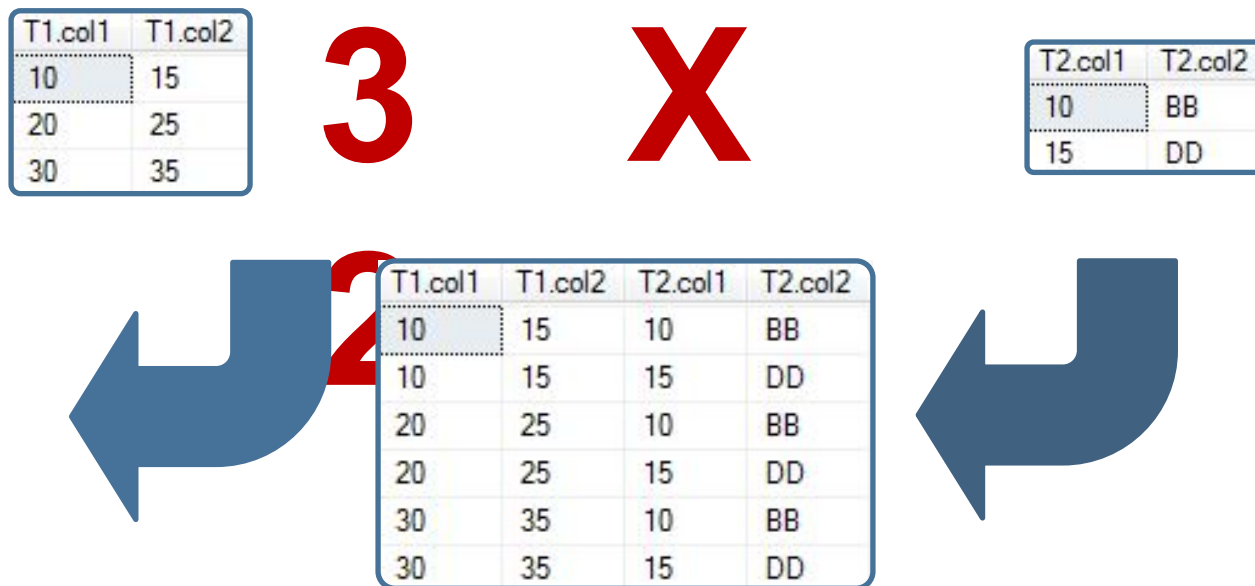


- La jointure horizontale compare **deux colonnes** entre elles et affiche les colonnes souhaitées pour chaque concordance trouvée
- La condition de la jointure (c'est-à-dire la comparaison à faire) utilisera souvent les **clés primaires et étrangères** liant les tables (mais ce n'est pas obligatoire)
- 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 travaille 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

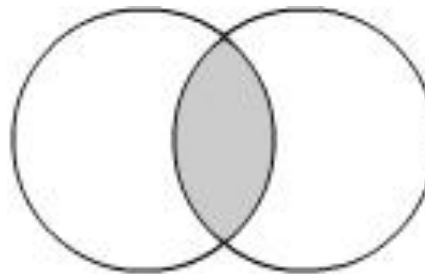


Jointures : **INNER JOIN**

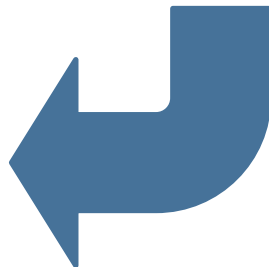
```
SELECT *  
FROM table1 T1 JOIN table2 T2 ON T1.col1 = T2.col1
```

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**)

T1.col1	T1.col2
10	15
20	25
30	35



T2.col1	T2.col2
10	BB
15	DD



T1.col1	T1.col2	T2.col1	T2.col2
10	15	10	BB



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	6

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

Equivalence

```
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 :

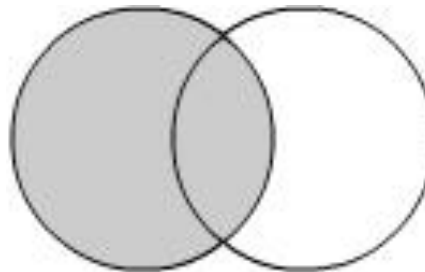
```
ERROR: ERREUR: la référence à la colonne « section_id » est ambigu
LINE 1: SELECT section_id, s.section_name, p.professor_name
                ^
```

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**)

T1.col1	T1.col2
10	15
20	25
30	35



T2.col1	T2.col2
10	BB
15	DD



T1.col1	T1.col2	T2.col1	T2.col2
10	15	10	BB
20	25	NULL	NULL
30	35	NULL	NULL



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

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

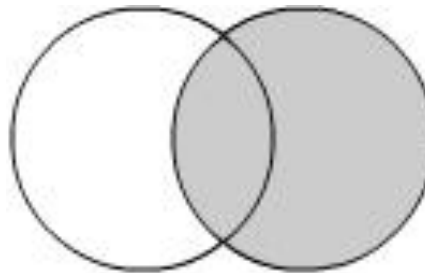
Liste de toutes les sections avec les professeurs qui y sont inscrits, s'il y en a

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**)

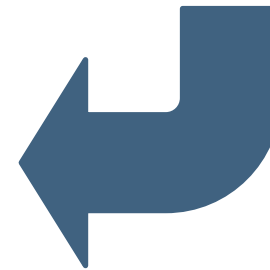
T1.col1	T1.col2
10	15
20	25
30	35



T2.col1	T2.col2
10	BB
15	DD



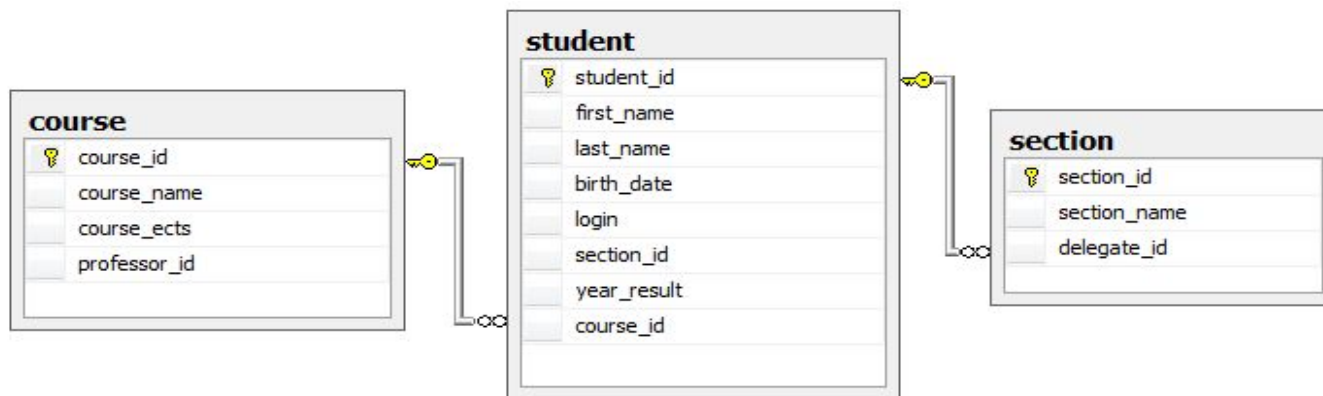
T1.col1	T1.col2	T2.col1	T2.col2
10	15	10	BB
NULL	NULL	15	DD



Jointures : RIGHT OUTER JOIN

```
SELECT CONCAT(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 CONCAT(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

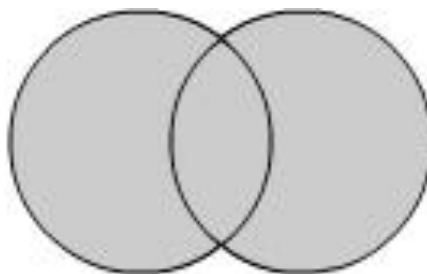
Certains étudiants ne sont pas délégué de section, certains sont délégués de 2 sections, certains étudiants ne sont inscrits dans aucun cours, mais la liste de tous les étudiants doit apparaître quoiqu'il en soit

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**)

T1.col1	T1.col2
10	15
20	25
30	35



T2.col1	T2.col2
10	BB
15	DD



T1.col1	T1.col2	T2.col1	T2.col2
10	15	10	BB
20	25	NULL	NULL
30	35	NULL	NULL
NULL	NULL	15	DD



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
ECGE2183	Financial Management
ECGE2184	Marketing management
EING2234	Derivatives
EING2283	Marketing engineering
EING2383	Supply chain management et e-business

Table
« COURSE »

professor_id
3
2
3
1
5

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

Table
« PROFESSEUR »

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

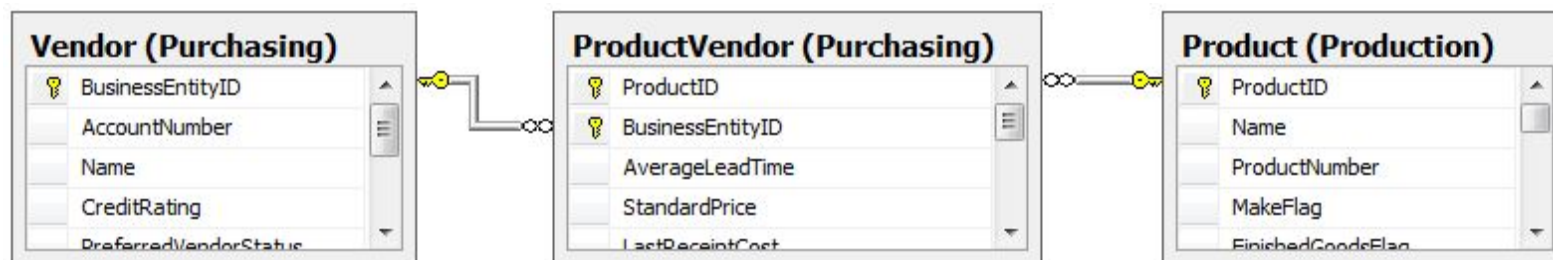


last_name	year_result	Grade
Basinger	19	E
Garcia	19	E
Gamer	18	E
Berry	18	E
Lucas	10	F
Depp	11	F
Reeves	10	F
Hanks	8	I
Eastwood	4	IG
De Niro	3	IG
Portman	4	IG
Clooney	4	IG

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 »

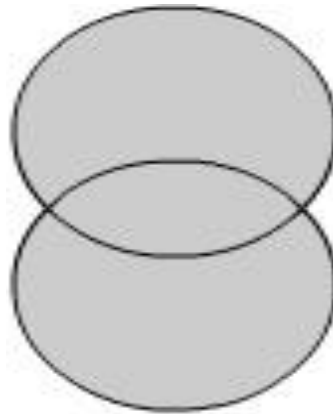
Jointures verticales

```
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

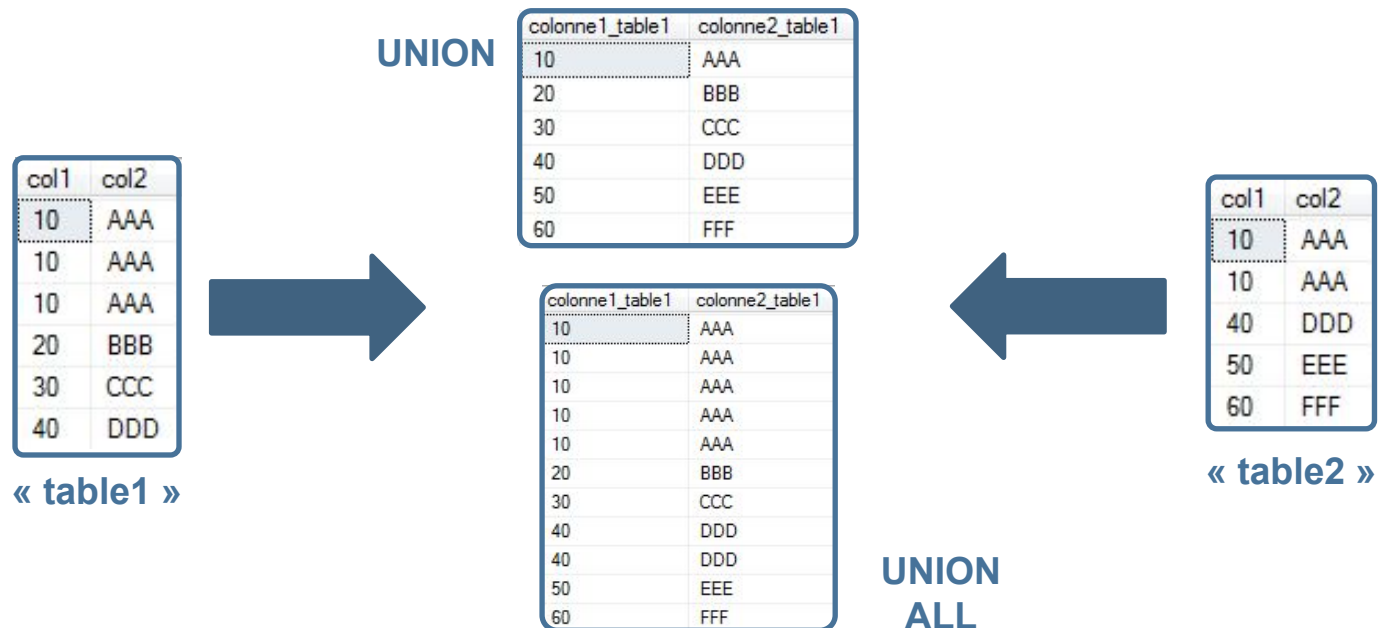
Jointures : **UNION [ALL]**

- 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



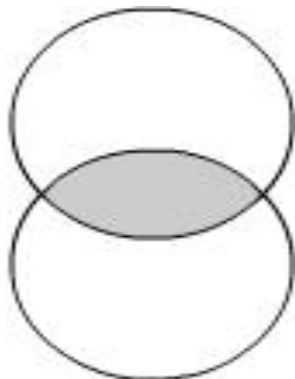
Jointures : UNION [ALL]

```
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"
```

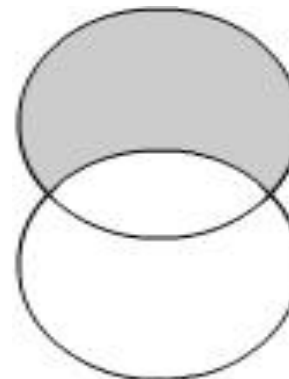


Jointures : **INTERSECT** et **EXCEPT**

- 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

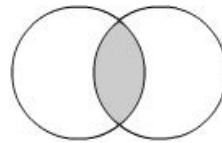


col1	col2
10	AAA
10	AAA
40	DDD
50	EEE
60	FFF

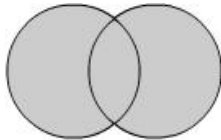
« table2 »

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

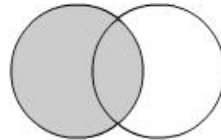
Jointures : Résumé



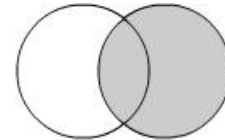
INNER JOIN



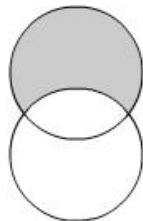
FULL OUTER JOIN



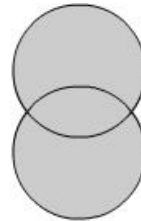
LEFT OUTER JOIN



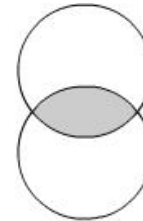
RIGHT OUTER JOIN



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')
```

WHERE year_result >= 16

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

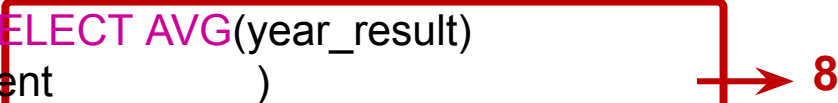
← Valeur renvoyée par la sous-requête

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 )
```



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
```



6
12
19
18
19
18

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

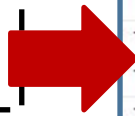
Sous-Requêtes : WHERE et HAVING

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
```



6
12
19
18
19
18

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

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

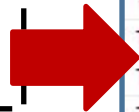
Sous-Requêtes : WHERE et HAVING

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

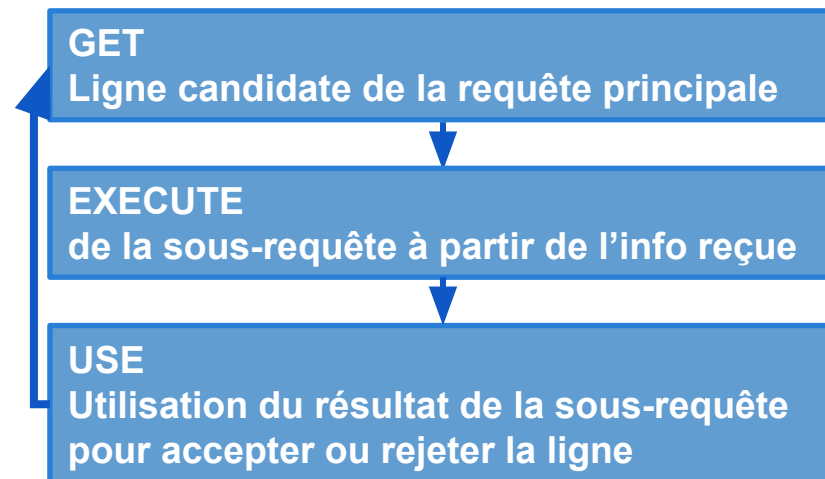
Si le résultat annuel de l'étudiant est supérieur ou égal à toutes les valeurs

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

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

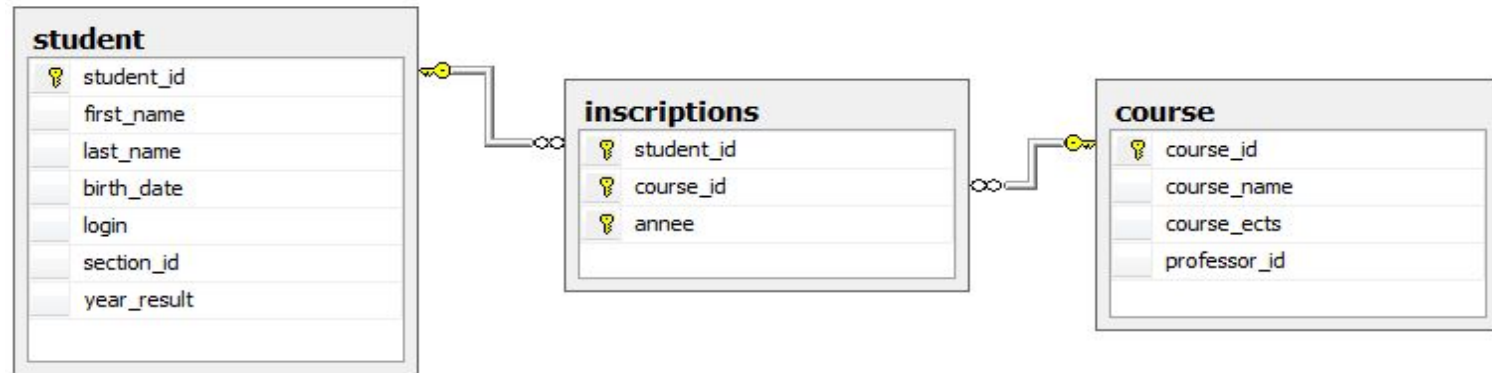


Table
« STUDENT »

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 / "UNIQUEMENT" disponible MySQL >= version 8

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 ttemp JOIN section AS s ON s.section_id = ttemp.section_id
WHERE nbr > 5
```



```
WITH ttemp
AS(
    SELECT section_id, COUNT(*) AS 'Nbr'
    FROM student
    GROUP BY section_id
)
SELECT section_name AS 'Section', nbr AS "Nombre d'étudiants"
FROM ttemp JOIN section AS s ON s.section_id = ttemp.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
« MYEMPLOYEE
S »

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)
```

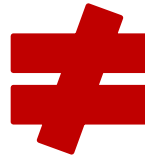


```
SELECT DISTINCT course_name  
FROM course c  
JOIN inscriptions i ON c.course_id = i.course_id
```

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) VALUES
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 section_id::CHAR(4) 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 = ( SELECT student_id
                    FROM student
                    WHERE last_name LIKE 'Cruise'),
section_name = 'SQL Déclaratif' WHERE section_id LIKE '19%'
```

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
```


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				

Références

- **PostgreSQL 12.3 Documentation**, site de PostgreSQL : [postgresql.org/docs/current/](https://www.postgresql.org/docs/current/)

Basé sur le cours de « SQL déclaratif du standard à la pratique » orienté SQLServer :

- 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
- **Oracle Database Online Documentation 12c Release (12.1)**, site d'Oracle : docs.oracle.com