DÉVELOPPER DES COMPOSANTS D'ACCÈS AUX DONNÉES

Module 1

INTRODUCTION

RAPPEL SUR LE MODÈLE RELATIONNEL

- Une base de données est un ensemble cohérent d'informations mémorisées sur support informatique
- Ces informations sont accessibles à l'aide d'une application appelée **système de gestion de base de données** (SGBD)
- Si ce SGBD est basé sur le modèle relationnel de CODD, on dit qu'il s'agit d'un **système de gestion de base de données relationnel** (SGBDR)
- Pour dialoguer avec un SGBDR on utilise le langage SQL
- Ce langage permet de soumettre des requêtes (des questions) au SGBDR

RAPPEL SUR LE MODÈLE RELATIONNEL

- Le **modèle relationnel** est constitué d'un ensemble d'opérations formelles sur les relations
- Les données sont stockées dans des tables qu'on peut mettre en relation
- Une **relation** se fait entre des colonnes ayant des données qui correspondent

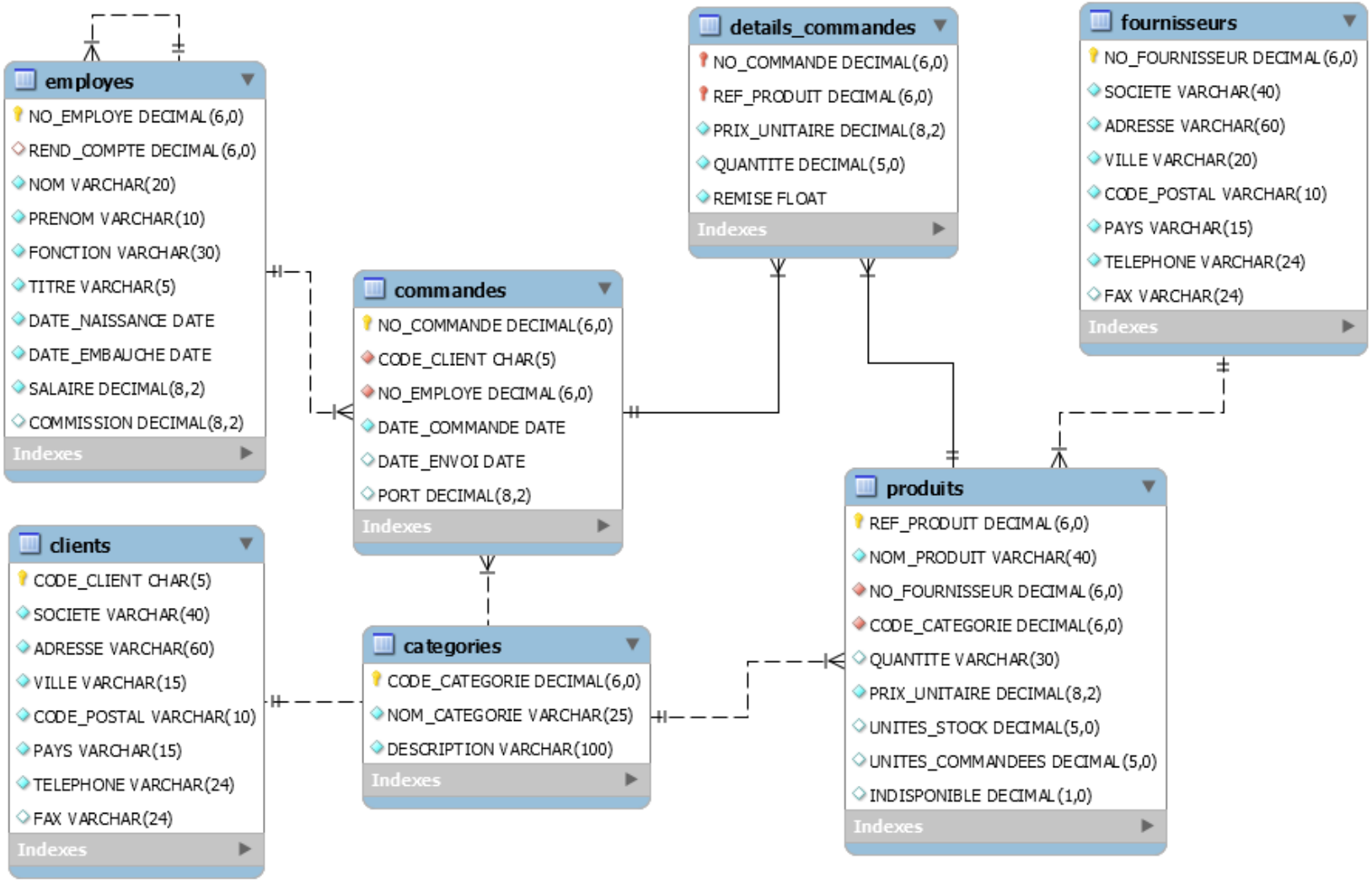
RAPPEL SUR LE MODÈLE RELATIONNEL

- La modélisation relationnelle permet de représenter les relations à l'aide de **tables** (à deux dimensions)
- Une **ligne** d'une table représente donc une entité
- Un **attribut** est le nom des colonnes qui constitue la définition d'une table : il comporte un typage de données
- On appelle **tuple** (ou n-uplet) une ligne de la table

RAPPEL SUR LE MODÈLE RELATIONNEL

- La **cardinalité** d'une relation est le nombre de tuples qui la composent
- La clé principale (ou primaire) d'une relation est l'attribut, ou l'ensemble d'attributs, permettant de désigner de façon unique un tuple.
- Une clé étrangère, par contre, est une clé faisant référence à une clé appartenant à une autre table.
- Le nom des colonnes à relier peut être différent, mais le type de données doit être le même.

LE MODÈLE DES DONNÉES DU COURS



RAPPEL SUR LE MODÈLE RELATIONNEL

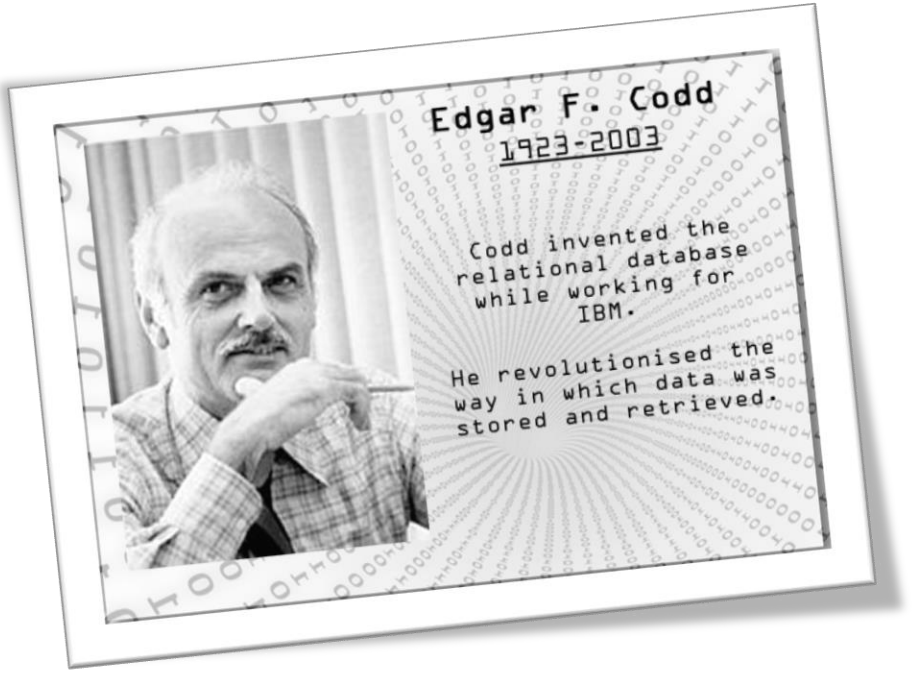
- **Indépendance physique** : le niveau physique peut être modifié indépendamment du niveau conceptuel
  - Cela signifie que tous les aspects matériels de la base de données n'apparaissent pas pour l'utilisateur, il s'agit simplement d'une structure transparente de représentation des informations
- **Indépendance logique** : le niveau conceptuel doit pouvoir être modifié sans remettre en cause le niveau physique
  - C'est-à-dire que l'administrateur de la base doit pouvoir la faire évoluer sans que cela gêne les utilisateurs
- **Manipulabilité** : des personnes ne connaissant pas la base de données doivent être capables de décrire leur requête sans faire référence à des éléments techniques de la base de données

RAPPEL SUR LE MODÈLE RELATIONNEL

- **Rapidité des accès** : le système doit pouvoir fournir les réponses aux requêtes le plus rapidement possible, cela implique des algorithmes de recherche rapides
- **Administration centralisée** : le SGBD doit permettre à l'administrateur de pouvoir manipuler les données, insérer des éléments, vérifier son intégrité de façon centralisée
- **Limitation de la redondance** : le SGBD doit pouvoir éviter dans la mesure du possible des informations redondantes, afin d'éviter d'une part un gaspillage d'espace mémoire mais aussi des erreurs
- **Vérification de l'intégrité** : les données doivent être cohérentes entre elles, de plus lorsque des éléments font référence à d'autres, ces derniers doivent être présents

RAPPEL SUR LE MODÈLE RELATIONNEL

- **Partageabilité des données** : le SGBD doit permettre l'accès simultané à la base de données par plusieurs utilisateurs
- **Sécurité des données** : le SGBD doit présenter des mécanismes permettant de gérer les droits d'accès aux données selon les utilisateurs



RAPPEL SUR LE MODÈLE RELATIONNEL

- Une **requête** est un ordre adressé à un SGBD
- Cet ordre peut consister à extraire, à ajouter, à modifier, à supprimer, à administrer les données de la base
- De façon générale, l'utilisateur comme l'administrateur, dialogue avec le SGBD en lui soumettant des requêtes (des questions) et en récupérant en retour des résultats (les réponses)

LANGAGE SQL : LES CARACTÉRISTIQUES

Le langage **SQL** est devenu le standard en matière d'interface relationnelle, ceci probablement parce que ce langage est :

- Issu de SEQUEL (interface de System-R), SQL a été développé chez IBM à San José en 1974 !
- Basé sur des mots clefs anglais explicites, il est relativement simple et facile à apprendre pour des utilisateurs non-informaticiens : Il illustre bien la tendance des langages formels à s'orienter vers un certain "langage naturel"
- Normalisé

LANGAGE SQL : LES CARACTÉRISTIQUES

- Le standard **ANSI** (American National Standards Institute) a valeur nominative, en principe seulement aux Etats-Unis
- L'équivalent français est la norme **AFNOR**
- La norme internationale de SQL est la norme **ISO** (International Standards Organisation) numéro 9075 de 1987
- Les normes sont accompagnées de niveaux qui indiquent le degré d'évolution de SQL

LANGAGE SQL : LES CARACTÉRISTIQUES

Nom	Appellation	Commentaires
ISO/CEI 9075:1986	SQL-86 ou SQL-87	Édité par l'ANSI puis adopté par l'ISO en 1987
ISO/CEI 9075:1989	SQL-89 ou SQL-1	Révision mineure
ISO/CEI 9075:1992	SQL-92 alias SQL2	Révision majeure
ISO/CEI 9075:1999	SQL-99 alias SQL3	Expressions rationnelles, requêtes récursives, déclencheurs, types non-scalaires et quelques fonctions orientées objet
ISO/CEI 9075:2003	SQL:2003	Introduction de fonctions pour la manipulation XML, ordres standardisés et colonnes avec valeurs auto-produites
ISO/CEI 9075:2008	SQL:2008	Ajout de quelques fonctions de fenêtrage, limitation du nombre de lignes (OFFSET/FETCH), curseurs, etc.
ISO/CEI 9075:2011	SQL:2011	Ajout du support des tables temporelles (historisation automatique)

LANGAGE SQL : LES CARACTÉRISTIQUES

La norme définit deux langages SQL :

- un **Langage de Manipulation de Données** et de modules (en anglais SQL-DML) pour déclarer les procédures d'exploitation et les appels à utiliser dans les programmes.
  On peut également rajouter une composante pour l'interrogation de la base : le **Langage d'Interrogation de Données** (en anglais SQL-DQL).
- un **Langage de Définition de Données** (en anglais SQL-DDL) à utiliser pour déclarer les structures logiques de données et leurs contraintes d'intégrité.
  On peut également rajouter une composante pour la gestion des accès aux données : le **Langage de Contrôle de Données** (en anglais SQL-DCL).

LANGAGE SQL : LES CARACTÉRISTIQUES

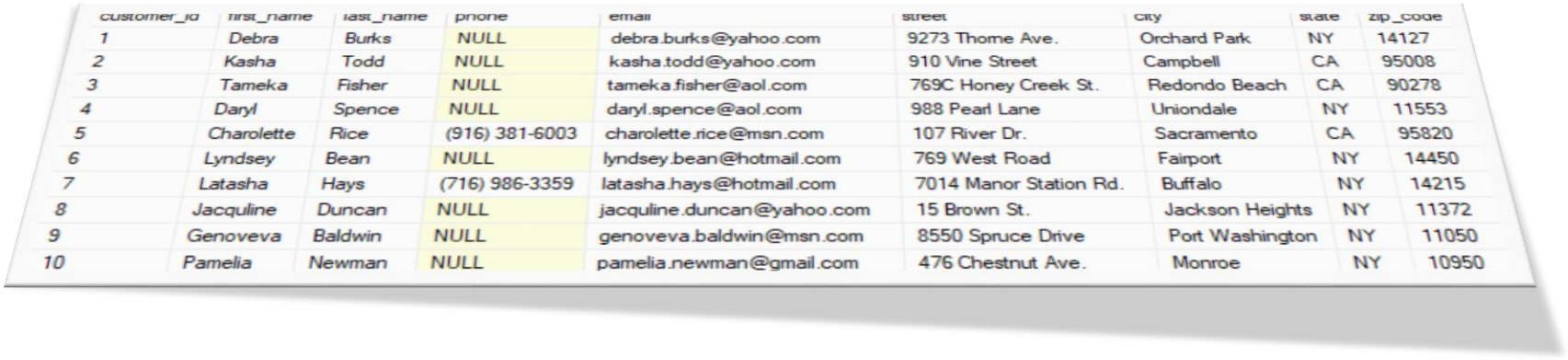
Types of SQL Commands			
DDL	**DML**	**DCL**	**TCL**
CREATE ALTER DROP TRUNCATE RENAME	SELECT INSERT UPDATE DELETE MERGE	GRANT REVOKE	COMMIT ROLLBACK SAVEPOINT

LA SÉLECTION DES DONNÉES (LID)

Module 2

LA COMMANDE SELECT

- L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données
- Cela s'effectue grâce à la commande **SELECT,** qui retourne des enregistrements dans un tableau de résultat
- Elle permet de sélectionner une ou plusieurs colonnes d'une ou plusieurs tables



customer_id	first_name	last_name	phone	email	street	city	state	zip_code
1	Debra	Burks	NULL	debra.burks@yahoo.com	9273 Thorne Ave.	Orchard Park	NY	14127
2	Kasha	Todd	NULL	kasha.todd@yahoo.com	910 Vine Street	Campbell	CA	95008
3	Tameka	Fisher	NULL	tameka.fisher@aol.com	769C Honey Creek St.	Redondo Beach	CA	90278
4	Daryl	Spence	NULL	daryl.spence@aol.com	988 Pearl Lane	Uniondale	NY	11553
5	Charolette	Rice	(916) 381-6003	charolette.rice@msn.com	107 River Dr.	Sacramento	CA	95820
6	Lyndsey	Bean	NULL	lyndsey.bean@hotmail.com	769 West Road	Fairport	NY	14450
7	Latasha	Hays	(716) 986-3359	latasha.hays@hotmail.com	7014 Manor Station Rd.	Buffalo	NY	14215
8	Jacquline	Duncan	NULL	jacquline.duncan@yahoo.com	15 Brown St.	Jackson Heights	NY	11372
9	Genoveva	Baldwin	NULL	genoveva.baldwin@msn.com	8550 Spruce Drive	Port Washington	NY	11050
10	Pamelia	Newman	NULL	pamelia.newman@gmail.com	476 Chestnut Ave.	Monroe	NY	10950

LA COMMANDE SELECT

Associées au **SELECT,** il existe plusieurs clauses qui permettent de mieux gérer les données que l'ont souhaite lire :

- Joindre un autre tableau aux résultats avec **JOIN** dans la clause **FROM**
- Filtrer pour ne sélectionner que certains enregistrements avec **WHERE**
- Grouper les résultats pour faire uniquement des statistiques (CA mensuel, salaire le plus élevé, etc.) avec **GROUP BY** et **HAVING**
- Classer les résultats avec **ORDER BY**

LA COMMANDE SELECT

- La requête **SELECT** suivante possède toutes les clauses possibles :

```
SELECT [DISTINCT|ALL] * ou colonnes/expressions
```
Spécification des colonnes à afficher dans le résultat
```
FROM tables/vues
```
Spécification des tables/vues sur lesquelles porte le SELECT
```
[WHERE conditions]
```
Filtre portant sur les données (conditions à remplir pour afficher le résultat)
```
[GROUP BY expressions]
```
Définition du sous-ensemble
```
[HAVING conditions]
```
Conditions de regroupement des lignes
```
[ORDER BY expressions]
```
Tri des données du résultat final

LA COMMANDE SELECT

Syntaxe :

```
SELECT nom, prenom
FROM client;
```

Résultat :

NOM	PRENOM
TALON	Achille
CASTAFIORE	Bianca
TSUNO	Yoko
LAGAFFE	Gaston
LAGAFFE	Jeanne

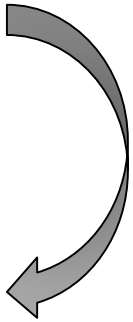


Table CLIENT

ID	PRENOM	NOM	VILLE
1	Achille	TALON	Paris
2	Bianca	CASTAFIORE	Milan
3	Yoko	TSUNO	Tokyo
4	Gaston	LAGAFFE	Bruxelles
5	Jeanne	LAGAFFE	

LA COMMANDE SELECT

- Le caractère « * » permet de récupérer toutes les colonnes de la table interrogée :

```
SELECT *
FROM client;
```

ID	PRENOM	NOM	VILLE
1	Achille	TALON	Paris
2	Bianca	CASTAFIORE	Milan
3	Yoko	TSUNO	Tokyo
4	Gaston	LAGAFFE	Bruxelles
5	Jeanne	LAGAFFE	

- Le mot-clé **DISTINCT** permet d'éliminer les doublons dans le résultat :

```
SELECT DISTINCT nom
FROM client;
```

NOM
TALON
CASTAFIORE
TSUNO
LAGAFFE

LA COMMANDE SELECT

- L'opérateur « || » (Oracle), le signe « + » (SQL Server) ou la fonction **CONCAT** permettent de concaténer des champs de type caractères :

```
SELECT prenom + ' ' + nom, prenom || ' ' || nom,
CONCAT(prenom, ' ', nom) FROM client;
```

PRENOM + ' ' + NOM	PRENOM || ' ' || NOM	CONCAT(PRENOM, ' ', NOM)
Achille TALON	Achille TALON	Achille TALON
Bianca CASTAFIORE	Bianca CASTAFIORE	Bianca CASTAFIORE
Yoko TSUNO	Yoko TSUNO	Yoko TSUNO
Gaston LAGAFFE	Gaston LAGAFFE	Gaston LAGAFFE
Jeanne LAGAFFE	Jeanne LAGAFFE	Jeanne LAGAFFE

- Les opérateurs arithmétiques de base suivants peuvent également être utilisés pour combiner différentes colonnes : + - * /

LA COMMANDE SELECT

- Le mot-clé **AS** permet de donner un alias aux colonnes renvoyées par la requête :

```
SELECT prenom + ' ' + nom AS Oracle, prenom || ' ' || nom
AS "SQL Server", CONCAT(prenom, ' ', nom) FROM client AS
MySQL;
```

Oracle	SQL Server	MySQL
Achille TALON	Achille TALON	Achille TALON
Bianca CASTAFIORE	Bianca CASTAFIORE	Bianca CASTAFIORE
Yoko TSUNO	Yoko TSUNO	Yoko TSUNO
Gaston LAGAFFE	Gaston LAGAFFE	Gaston LAGAFFE
Jeanne LAGAFFE	Jeanne LAGAFFE	Jeanne LAGAFFE

LA GESTION DE NULL

Dans le langage SQL il peut s'avérer utile de traiter des résultats qui possèdent des données vierges (ou **NULL**)

Pour cela on utilise une fonction qui porte des noms différents selon le système de gestion de base de données :

- **COALESCE**(expression_testée, expression_de_remplacement)
  - Si l'argument **expression_testée** n'est pas NULL, la fonction retournera la valeur de l'argument **expression_testée,** sinon elle retournera la valeur de l'argument **expression_de_remplacement**
- `SELECT COALESCE(ville, 'inconnue') FROM client;`
- La fonction **COALESCE** est supportée par tous les systèmes de gestion de base de données

LA GESTION DE NULL

- La gestion de NULL sous SQL Server :
  ISNULL(expression testée, expression de remplacement)
- La gestion de NULL sous Oracle :
  NVL(expression testée, expression de remplacement)
- La gestion de NULL sous MySQL :
  IFNULL(expression testée, expression de remplacement)
-
```
SELECT prenom, ISNULL(ville, 'inconnue') AS "SQL Server",
NVL(ville, 'inconnue') AS Oracle, IFNULL(ville, 'inconnue')
AS MySQL FROM client;
```

PRENOM	SQL Server	Oracle	MySQL
Achille	Paris	Paris	Paris
Bianca	Milan	Milan	Milan
…	…	…	…
Jeanne	inconnue	inconnue	inconnue

TRAVAUX PRATIQUES 2-1

1. Affichez tous les employés de la société
2. Affichez toutes les catégories de produits
3. Affichez le nom, le prénom, la date de naissance et la commission (mettre à 0 si pas de commission) de tous les employées de la société
4. Affichez la liste des fonctions exercées par les employés de la société
5. Affichez la liste des pays de résidence des clients
6. Affichez la liste des villes dans lesquelles résident les fournisseurs

TRAVAUX PRATIQUES 2-2

1. Affichez le nom des produits ainsi que leur valeur en stock (prix unitaire * unités en stock)
2. Affichez le nom, le prénom, l'âge et l'ancienneté des employés (fonction DATEDIFF)
3. Écrivez la requête qui permet d'afficher pour tous les employés le résultat suivant :

```
Employé     a un    gain annuel     sur 12 mois
-------     -----   -----------     -----------
Davolio     gagne   37620           par an
Fuller      gagne   120000          par an
…           …       …               …
Buchanan    gagne   96000           par an
Ford        gagne   60000           par an
```

LA CLAUSE WHERE

- La clause **WHERE** dans une requête SQL permet d'extraire les lignes d'une table qui respectent une ou plusieurs conditions :
- **Syntaxe :**
  ```
  SELECT nom_colonnes FROM nom_table WHERE condition
  ```
- **Exemples :**
  ```
  SELECT prenom, nom, ville FROM client WHERE ville = 'Paris';
  ```

PRENOM	NOM	VILLE
Achille	TALON	Paris

```
SELECT id, prenom, nom FROM client WHERE id > 3;
```

ID	PRENOM	NOM
4	Gaston	LAGAFFE
5	Jeanne	LAGAFFE

LA CLAUSE WHERE

- Il existe plusieurs opérateurs de comparaison dont voici les plus couramment utilisés :

Opérateur	Description
=	Égal
<> Ou !=	Différent de
> / <	Supérieur strictement à / Inférieur strictement à
>= / <=	Supérieur ou égale à / Inférieur ou égale à
IN	Une liste de plusieurs valeurs possibles
BETWEEN	Comprise dans un intervalle donné
LIKE	Recherche en spécifiant le début, milieu ou fin d'un mot
IS NULL / IS NOT NULL	Valeur est nulle / Valeur n'est pas nulle

LA CLAUSE WHERE - AND / OR

Si une requête SQL peut être restreinte à l'aide de la condition **WHERE,** les opérateurs logiques **AND** et **OR** permettent de combiner des conditions

- ```
  SELECT nom_colonnes
  FROM nom_table
  WHERE condition1 AND condition2
  ```
- ```
  SELECT nom_colonnes
  FROM nom_table
  WHERE condition1 OR condition2
  ```
- ```
  SELECT nom_colonnes
  FROM nom_table
  WHERE condition1 AND (condition2 OR condition3)
  ```
  ```

LA CLAUSE WHERE - IN

L'opérateur logique **IN** permet de vérifier si une colonne est égale à une des valeurs comprise dans un jeu de valeurs déterminés : il évite alors d'avoir à utiliser de multiple fois l'opérateur **OR**

- ```
  SELECT nom_colonne
  FROM table
  WHERE nom_colonne IN (valeur1, valeur2, valeur3, …)
  -- OU –
  WHERE nom_colonne IN (SELECT … FROM …)
  ```

- **Exemple :**
  ```
  SELECT prenom, nom FROM client
  WHERE ville IN ('Lyon', 'Tokyo', 'Paris');
  ```

PRENOM	NOM	VILLE
Achille	TALON	Paris
Yoko	TSUNO	Tokyo
  ```

LA CLAUSE WHERE - BETWEEN

L'opérateur logique **BETWEEN** permet de sélectionner un intervalle de données pouvant être constitué de chaînes de caractères, de nombres ou de dates dans une requête utilisant **WHERE**

- ```
  SELECT *
  FROM table
  WHERE nom_colonne BETWEEN valeur1 AND valeur2
  ```

- **Exemple :**

```
SELECT prenom, nom FROM client
WHERE prenom BETWEEN 'E' AND 'K';
```

PRENOM	NOM
Gaston	LAGAFFE
Jeanne	LAGAFFE
```

LA CLAUSE WHERE - LIKE

L'opérateur **LIKE** permet d'effectuer une recherche sur un modèle (ou pattern) particulier : on peut ainsi rechercher des enregistrements dont la valeur d'une colonne commence par telle ou telle lettre

- ```
  SELECT *
  FROM table
  WHERE colonne LIKE modele
  ```

le modèle utilise très généralement les jokers suivants :

- le caractère « % » est un caractère joker qui remplace à n caractères
- le caractère « _ » (underscore) peut être remplacé par un et un seul caractère uniquement
  ```

LA CLAUSE WHERE - LIKE

- **prenom LIKE '%o'** : renvoie tous les prénoms qui se terminent par un "a", comme "Yoko"
- **prenom LIKE 'A%'** : renvoie tous les prénoms qui commencent par un "a", comme "Achille"
- **prenom LIKE '%a%'** : renvoie tous les prénoms qui contiennent au moins une fois le caractère "a" , comme "Jeanne" ou "Gaston"
- **mot LIKE 'pa%on'** : renvoie les chaines qui commence par "pa" et qui se terminent par "on", comme "pantalon" ou "pardon"
- **mot LIKE 'a_c'** : renvoie les mots de trois lettres commençant par "a" et finissant par "c", comme "arc"

LA CLAUSE WHERE - IS NULL

L'opérateur **IS** permettant de filtrer les résultats qui ne contiennent rien est indispensable car **NULL** est une valeur inconnue et ne peut donc pas être filtrée par les opérateurs de comparaison

- ```
  SELECT *
  FROM 'table'
  WHERE nom_colonne IS [NOT] NULL
  ```

- **Exemple :**
  ```
  SELECT prenom, nom, ville FROM client
  WHERE ville IS NULL;
  ```

PRENOM	NOM	VILLE
Jeanne	LAGAFFE	
  ```

TRAVAUX PRATIQUES 2-3

1. Affichez le nom de la société et le pays des clients qui habitent à Toulouse
2. Affichez le nom, le prénom et la fonction des employés dirigés par l'employé numéro 2
3. Affichez le nom, le prénom et la fonction des employés qui ne sont pas des représentants
4. Affichez le nom, le prénom et la fonction des employés qui ont un salaire inférieur à 3500
5. Affichez le nom, la ville et le pays des clients qui n'ont pas de fax
6. Affichez le nom, le prénom et la fonction des employés qui n'ont pas de supérieur

LA CLAUSE ORDER BY

- La clause **ORDER BY** permet de trier les lignes du résultat d'une requête SQL : on peut trier les données sur une/plusieurs colonnes et par ordre ascendant/descendant
- **Syntaxe :**

```
SELECT colonne1, colonne2
FROM table
ORDER BY colonne1
```

- Par défaut les résultats sont classés par ordre ascendant, toutefois il est possible d'inverser l'ordre en utilisant le mot-clé **DESC** après le nom de la colonne
- **Syntaxe :**

```
SELECT colonne1, colonne2, colonne3
FROM table
ORDER BY colonne1 DESC, colonne2 ASC
```

LA CLAUSE ORDER BY

- ```
  SELECT nom, prenom
  FROM client
  ORDER BY nom, prenom DESC;
  ```

NOM	PRENOM
CASTAFIORE	Bianca
LAGAFFE	Jeanne
LAGAFFE	Gaston
TALON	Achille
TSUNO	Yoko

- ```
  SELECT id, prenom, nom
  FROM client
  ORDER BY id DESC;
  ```

ID	PRENOM	NOM
5	Jeanne	LAGAFFE
4	Gaston	LAGAFFE
3	Yoko	TSUNO
2	Bianca	CASTAFIORE
1	Achille	TALON

TRAVAUX PRATIQUES 2-4

1. Affichez le nom et le prénom des employés triés par nom en ordre décroissant
2. Affichez le nom des sociétés ainsi que la ville des clients triés par pays
3. Affichez le nom des sociétés, le pays et la ville des clients triés par pays croissant et par ville décroissant
4. Affichez le nom et la commission des employés triés par commission

LES JOINTURES

- Les **jointures** en SQL permettent d'associer plusieurs tables dans une même requête
- Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace
- En général, les jointures consistent à **associer des lignes de 2 tables** en associant **l'égalité** des valeurs d'une **colonne d'une première table** par rapport à la valeur d'une **colonne d'une seconde table**
- Il existe également d'autres cas de jointures incluant des jointures sur la même table ou des jointures d'inégalité

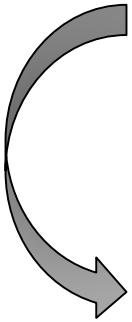
LES JOINTURES

Table SOCIETE : 3 lignes

SIRET	RAISON
11	Casterman
22	Marvel
33	Dupuis

Table CLIENT : 5 lignes

ID	PRENOM	NOM	VILLE	SIRET
1	Achille	TALON	Paris	
2	Bianca	CASTAFIORE	Milan	11
3	Yoko	TSUNO	Tokyo	33
4	Gaston	LAGAFFE	Bruxelles	33
5	Jeanne	LAGAFFE		33



```
SELECT * FROM societe, client; -- 15 lignes
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	1	Achille	TALON	Paris	44
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
11	Casterman	3	Yoko	TSUNO	Tokyo	33
11	Casterman	4	Gaston	LAGAFFE	Bruxelles	33
11	Casterman	5	Jeanne	LAGAFFE		33
…	…	…	…	…	…	…
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33

LES JOINTURES

- Dans la requête précédente, une nouvelle table est construite avec pour colonnes l'ensemble des colonnes des deux tables et pour lignes le **produit cartésien** des deux tables : *3 lignes x 5 lignes = 15 lignes*
- En ajoutant une **condition de jointure** dans la clause **WHERE**, on ne ramène plus que les lignes pour lesquelles la (ou les) condition(s) de jointure sont respectée(s) et uniquement celles-ci :

```
SELECT * FROM societe s, client c
WHERE a.siret = c.siret; -- 4 lignes
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33

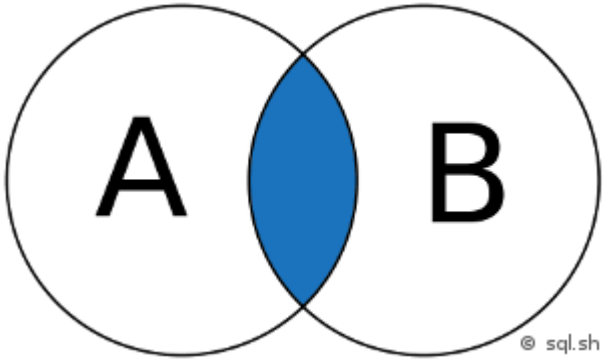
LES JOINTURES

- Il y a plusieurs méthodes pour associer 2 tables ensemble :
  - **INNER JOIN** : jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables - jointures les plus courantes
  - **NATURAL JOIN** : jointure naturelle entre 2 tables s'il y a au moins une colonne qui porte le même nom entre les 2 tables
  - **LEFT JOIN** (ou LEFT OUTER JOIN) : jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifié dans l'autre table
  - **RIGHT JOIN** (ou RIGHT OUTER JOIN) : jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifié dans l'autre table
  - **FULL JOIN** (ou FULL OUTER JOIN) : jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables
  - **CROSS JOIN** : jointure croisée permettant de faire sciemment le produit cartésien entre 2 tables

LES JOINTURES - INNER JOIN

- **Syntaxe :**

```
SELECT *
FROM A [INNER] JOIN B
ON A.key = B.key;
```



- **Exemple :**

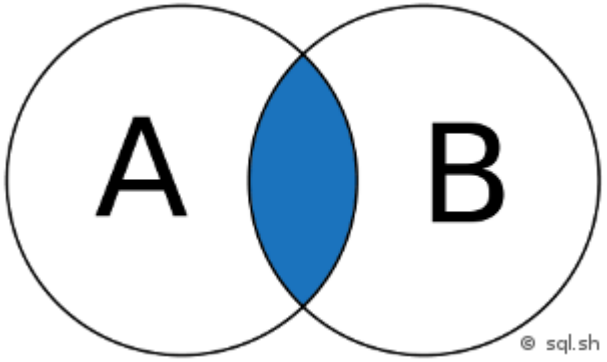
```
SELECT * FROM societe s INNER JOIN client c
ON s.siret = c.siret;
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33

LES JOINTURES - JOIN ... USING

- **Syntaxe :**

```
SELECT *
FROM A [INNER] JOIN B
USING(key);
```



- **Exemple :**

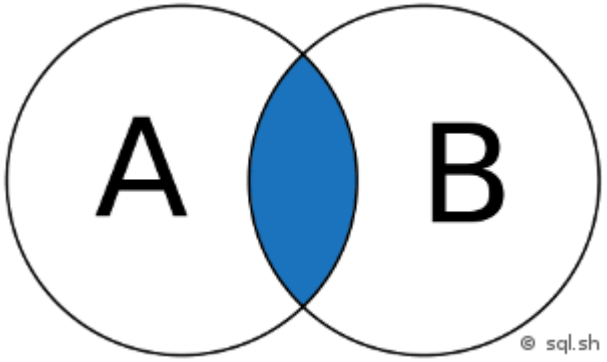
```
SELECT * FROM societe s INNER JOIN client c
USING(siret);
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33

LES JOINTURES - NATURAL JOIN

- **Syntaxe :**

```
SELECT *
FROM A NATURAL JOIN B
```



- **Exemple :**

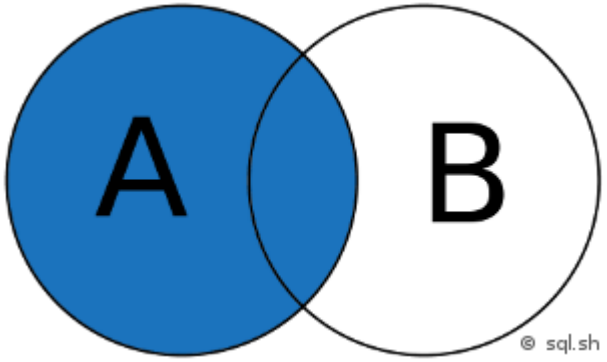
```
SELECT * FROM societe s NATURAL JOIN client c;
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33

LES JOINTURES - LEFT JOIN

- **Syntaxe :**

```
SELECT *
FROM A LEFT [OUTER] JOIN B
ON A.key = B.key;
```



- **Exemple :**

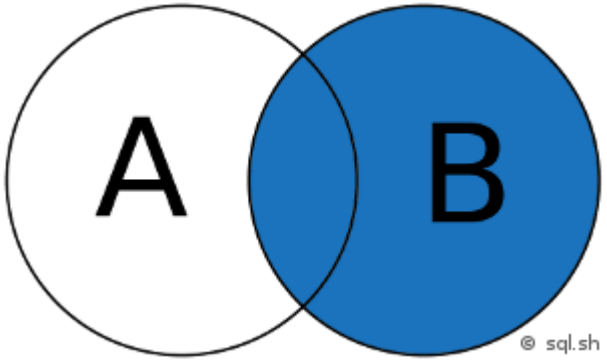
```
SELECT * FROM societe s LEFT OUTER JOIN client c
ON s.siret = c.siret;
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33
22	Marvel					

LES JOINTURES - RIGHT JOIN

- **Syntaxe :**

```
SELECT *
FROM A RIGHT [OUTER] JOIN B
ON A.key = B.key;
```



- **Exemple :**

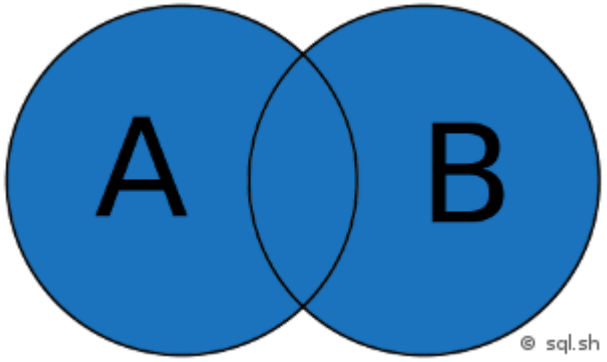
```
SELECT * FROM societe s RIGHT OUTER JOIN client c
ON s.siret = c.siret;
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33
		1	Achille	TALON	Paris	

LES JOINTURES - FULL JOIN

- **Syntaxe :**

```
SELECT *
FROM A FULL [OUTER] JOIN B
ON A.key = B.key;
```



- **Exemple :**

```
SELECT * FROM societe s FULL OUTER JOIN client c
ON s.siret = c.siret;
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
33	Dupuis	3	Yoko	TSUNO	Tokyo	33
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33
22	Marvel					
		1	Achille	TALON	Paris	

LES JOINTURES - CROSS JOIN

- **Syntaxe :**

```
SELECT *
FROM A
CROSS JOIN B;
```

- **Exemple :**

```
SELECT * FROM societe s CROSS JOIN client c;
```

SIRET	RAISON	ID	PRENOM	NOM	VILLE	SIRET
11	Casterman	1	Achille	TALON	Paris	44
11	Casterman	2	Bianca	CASTAFIORE	Milan	11
11	Casterman	3	Yoko	TSUNO	Tokyo	33
…	…	…	…	…	…	…
33	Dupuis	4	Gaston	LAGAFFE	Bruxelles	33
33	Dupuis	5	Jeanne	LAGAFFE		33

TRAVAUX PRATIQUES 2-5

1. Affichez le nom, le prénom, la fonction et le salaire des employés qui ont un salaire compris entre 2500 et 3500
2. Affichez le nom du produit, la raison sociale du fournisseur, le nom de la catégorie et les quantités de produits qui ne sont pas d'une des catégories 1, 3, 5 ou 7
3. Affichez le nom du produit, la raison sociale du fournisseur, le nom de la catégorie et les quantités de produits qui ont code fournisseur compris entre 1 et 3 ou un code catégorie compris entre 1 et 3 et pour lesquelles les quantités sont données en boîte(s) ou en carton(s)

TRAVAUX PRATIQUES 2-5

4. Écrivez la requête qui permet d'afficher le nom des employés qui ont effectué au moins une vente pour un client parisien
5. Affichez le nom des produits et la raison sociale des fournisseurs pour les produits des catégories 1, 4 et 7
6. Affichez le nom des employés ainsi que le nom de leur supérieur hiérarchique

LES FONCTIONS

Module 3

LES FONCTIONS NUMÉRIQUES

- Dans le langage SQL il existe de nombreuses **fonctions mathématiques** pour effectuer des calculs ou des statistiques concernant les données contenus dans une base de données

Fonction	Description
CEIL(expression) CEILING(expression) – SQL Server	Renvoie la valeur entière supérieure d'un nombre
FLOOR(expression)	Renvoie la valeur entière inférieure d'un nombre
ROUND(expression, précision)	Permet d'arrondir un résultat numérique
TRUNC(expression, précision) TRUNCATE(expression, précision) – MySQL ROUND(expression, précision,1) – SQL Server	Tronque une valeur numérique en un nombre avec les décimales souhaitées (sans arrondir)

LES FONCTIONS NUMÉRIQUES

Fonction	Description
ABS(expression)	Renvoie la valeur absolue d'un nombre
MOD(expression)	Renvoie le modulo d'un nombre (le reste d'une division)
SIGN(expression)	Indique si l'argument est supérieur, inférieur ou égal à zéro
SQRT(expression)	Calcule la racine carrée d'un nombre
EXP(expression)	Calcule l'exponentiel d'un nombre
LN(expression)	Renvoie le logarithme népérien de l'argument
LOG(expression)	Renvoie le logarithme décimal de l'argument
POWER(expression, valeur)	Renvoie l'argument à la puissance spécifiée
SIN(expression)	Renvoie le sinus d'un nombre
COS(expression)	Renvoie le cosinus d'un nombre
TAN(expression)	Renvoie la tangente d'un nombre
PI() ASIN(1)*2 – Oracle	Renvoie la valeur de PI (3,14159)
EXP(expression)	Calcule l'exponentiel d'un nombre

LES FONCTIONS DATE ET HEURE

- En SQL il existe une multitude de fonctions qui concerne les **éléments temporels** pour pouvoir lire ou écrire plus facilement des données à une date précise ou à un intervalle de date

Fonction	Description
CURRENT_DATE() GETDATE() – SQL Server SYSDATE – Oracle	Renvoie la date courante du système
DAY(date) / MONTH(date) / YEAR(date)	Extrait une partie d'une date
DATE_ADD(date) / DATE_SUB(date)	Ajoute/soustrait une valeur à une date
DATEADD (day, 1, interval) – SQL Server	Ajoute des jours, des mois, des années à une date

LES FONCTIONS DATE ET HEURE

Fonction	Description
date + INTERVAL '1' interval	Ajoute des jours, des mois, des années à une date
ADD_MONTHS(date, nb_mois) – Oracle	Ajoute des mois à une date
NEXT_DAY (date, 'jour') – Oracle	Date postérieure à la date pour le jour indiqué
LAST_DAY(date)	Indique le dernier jour du mois de la date indiquée
MONTHS_BETWEEN(date1, date2) – Oracle	Nombre de mois entre date1 et date2 : la partie décimale est obtenue en divisant le nombre de jours par 31
DATEDIFF(date1, date2) AGE(date1, date2) – PostgreSQL	Renvoie l'écart en jours entre deux dates
DATE_PART(date, interval) DATEPART(date, interval) – SQL Server	Extrait une partie de la date indiquée (heure, jour, mois…)
EXTRACT(interval FROM date)	Renvoie la valeur du jour, mois ou année

LES FONCTIONS CHAÎNES DE CARACTÈRES

- Les fonctions SQL sur les **chaînes de caractères** permettent d'ajouter de nombreuses fonctionnalités aux requêtes SQL : elles sont mono-lignes et ne s'appliquent donc qu'à une seule ligne en même temps

Fonction	Description
LOWER(chaine)	Transforme la chaîne pour tout retourner en minuscule
UPPER(chaine)	Transforme la chaîne pour tout retourner en majuscule
INITCAP(chaine) – Oracle, PostegreSQL	Transforme la chaîne en lettres capitales
SUBSTRING(chaine, debut, longueur) SUBSTR(chaine, debut, longueur) – Oracle	Retourne une sous-chaîne en partant de la position "debut" sur la longueur définie par "longueur"
REPLACE(chaine, cherche, remplace)	Remplace les caractères d'une chaine par d'autres caractères
CONCAT(chaine1, chaine2)	Concatène plusieurs chaînes de caractères en une seule
REVERSE(chaine)	Inverse l'ordre des caractères d'une chaîne

LES FONCTIONS CHAÎNES DE CARACTÈRES

Fonction	Description
LPAD(chaine, nbcar, car) – MySQL, Oracle	Ajoute un caractère spécifié au début d'une chaîne jusqu'à atteindre la longueur souhaitée
RPAD(chaine, nbcar, car) – MySQL, Oracle	Ajoute un caractère spécifié en fin d'une chaîne jusqu'à atteindre la longueur souhaitée
LTRIM(chaine)	Supprime les caractères vides au début de chaîne
RTRIM(chaine)	Supprime les caractères vides en fin de chaîne
INSTR(chaine, 'texte cherché') – Oracle	Renvoie la position d'une occurrence dans une chaîne de caractères
PATINDEX('%texte cherché%', chaine) – SQL Server	Renvoie la position d'une occurrence dans une chaîne de caractères
POSITION('texte cherché' in chaine) – PostgreSQL	Renvoie la position d'une occurrence dans une chaîne de caractères
LENGTH(chaine) LEN(chaine) – SQL Server	Renvoie la longueur d'une chaîne
SOUNDEX(chaine)	Renvoie la version SOUNDEX de la chaîne

LES FONCTIONS DE CHIFFREMENT

- Les **fonctions de chiffrement** sont particulièrement utile pour chiffrer ou crypter des données avant de les enregistrer : elles permettent d'améliorer la sécurité en évitant d'enregistrer en "clair" des données sensibles

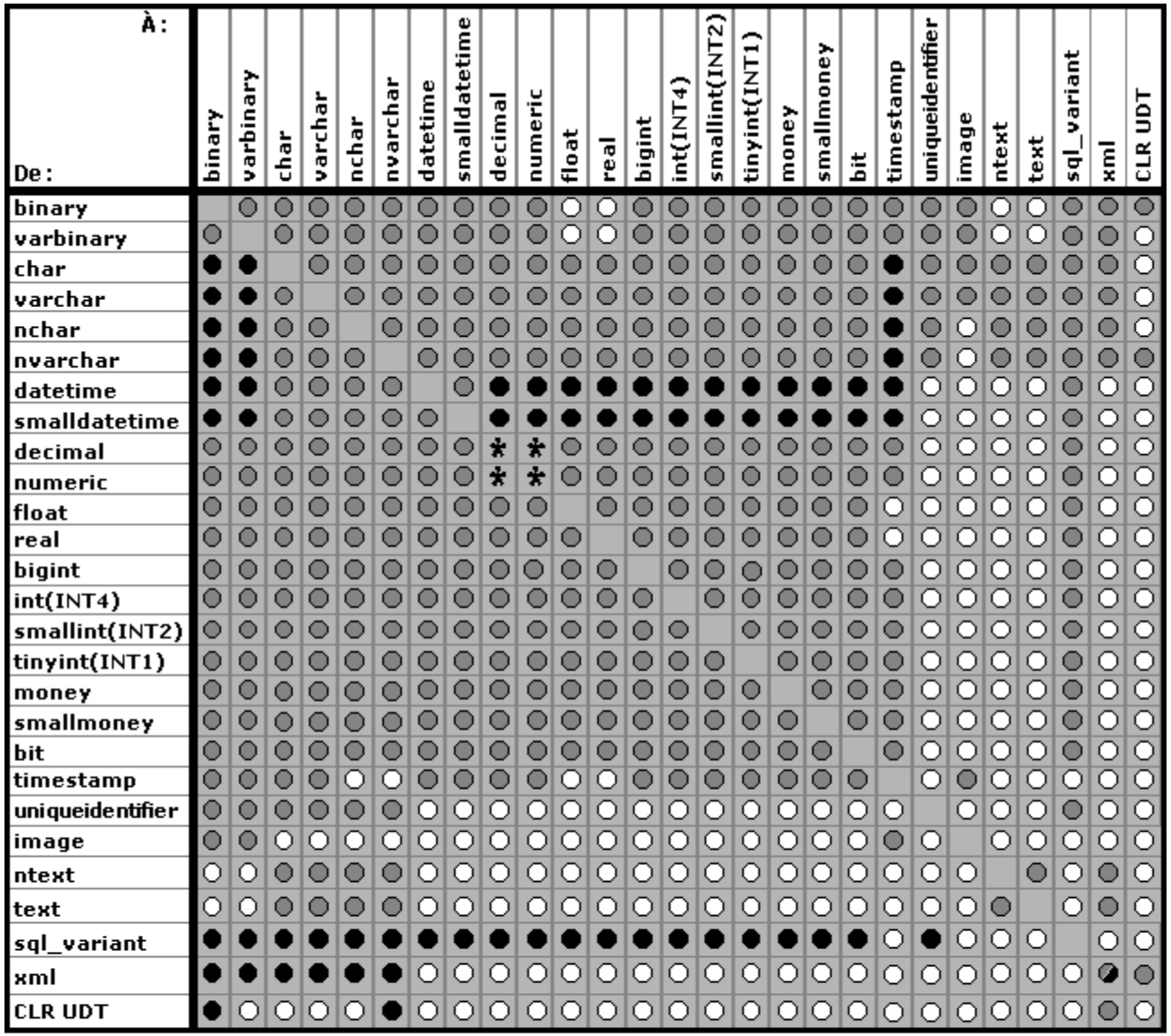
Fonction	Description
ENCODE(chaine, pass)	Chiffre des données en utilisant une clé spécifique
DECODE(chaine, pass)	déchiffre des données en utilisant une clé spécifique
MD5(chaine)	Calcule le MD5 d'une chaîne de caractère et renvoie un entier hexadécimal de 32 caractères (fonction souvent utilisée pour établir une clé de hashage)
SHA1(chaine)	Calcule le SHA1 d'une chaîne de caractères et renvoie un entier hexadécimal de 40 caractères (fonction de cryptographie sûre pour stocker un mot de passe)

LES FONCTIONS DE TRANSTYPAGE

- Les fonctions de **transtypage** permettent de convertir une donnée d'un type en un autre : on peut par exemple transformer une date du format DATETIME en DATE, ou l'inverse

Fonction	Description
CAST(expression AS type)	Transforme "expression" en un type de données qui peut être l'une des valeurs suivantes : BINARY, CHAR, NUMERIC, DATE, DATETIME ou TIME
CONVERT(expression, type) – MySQL, SQL Server	Le paramètre "expression" correspond à la donnée qui doit subir le transtypage : le type de données peut être BINARY, CHAR, DATE, DATETIME, INTEGER ou TIME
TO_CHAR(expression, 'format') – Oracle, PostgreSQL	Transforme "expression" en une date sous forme littérale
TO_DATE(expression, 'format') – Oracle, PostgreSQL	Convertit une chaîne de caractère en date
TO_NUMBER(expression, 'format') – Oracle, PostgreSQL	Transforme "expression" en un nombre sous forme littérale

TABLE DE CONVERSION



● Conversion explicite

● Conversion implicite

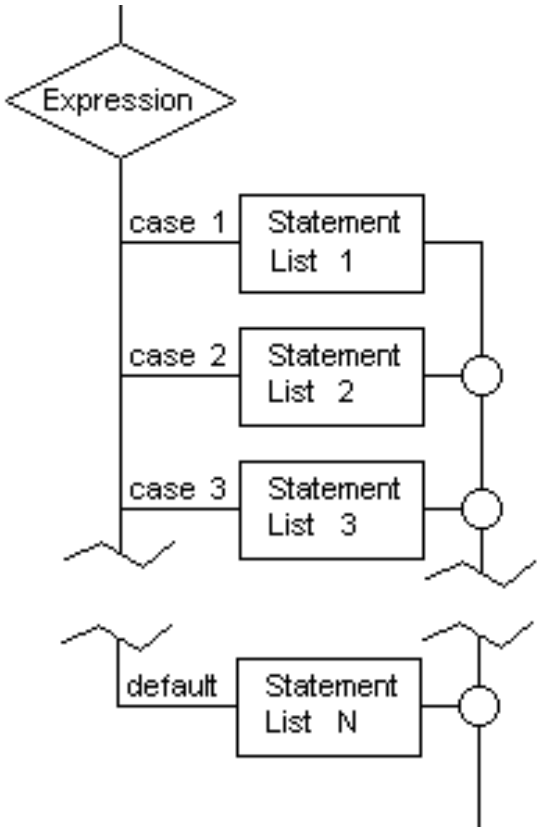
○ Conversion non autorisée

* Nécessite une fonction CAST explicite afin d'éviter la perte de précision ou d'échelle susceptible de se produire dans une conversion implicite.

◒ Les conversions implicites entre des types de données XML sont prises en charge uniquement si la source ou la cible est du code XML non typé. Sinon, les conversions doivent être explicites.

L'INSTRUCTION CASE

- L'instruction **CASE…WHEN** permet d'utiliser une structure conditionnelle du type IF similaire à celle d'un langage de programmation pour retourner un résultat disponible entre plusieurs possibilités
- Cette instruction peut être utilisée dans n'importe quelle clause, telle que SELECT, UPDATE, DELETE, WHERE, ORDER BY ou HAVING



L'INSTRUCTION CASE

- On peut comparer une colonne/expression à un jeu de résultat possible :

```
CASE a
   WHEN 1 THEN 'un'
   WHEN 2 THEN 'deux'
   WHEN 3 THEN 'trois'
   ELSE 'autre'
END
```

- On peut aussi élaborer une série de conditions booléennes pour déterminer un résultat :

```
CASE
   WHEN a=b THEN 'A égal à B'
   WHEN a>b THEN 'A supérieur à B'
   ELSE 'A inférieur à B'
END
```

TRAVAUX PRATIQUES 3-1

1. Affichez les employés et leur salaire journalier (salaire/20) arrondi à l'entier supérieur
2. Affichez les employés et leur revenu annuel (salaire*12 + commission) arrondi à la centaine près
3. Affichez les produits commercialisés, la valeur du stock (unités en stock fois prix unitaire) arrondie à la dizaine inférieure
4. Affichez le nom, la date de fin de période d'essai (3 mois) et leur ancienneté à ce jour exprimé en mois pour tous les employés
5. Affichez le nom des employés et le jour de leur première paie (dernier jour du mois de leur embauche)

LA CLAUSE GROUP BY

- La clause **GROUP BY** permet par **regroupement** de supprimer les doublons dans les lignes ramenées et de réaliser des calculs statistiques sur les colonnes sélectionnées

- **Syntaxe :**

```
SELECT colonne1, fonction(colonne2)
FROM table
GROUP BY colonne1
```

- **Exemple :**

```
SELECT nom, COUNT(id) AS nb
FROM client
GROUP BY nom;
```

NOM	NB
LAGAFFE	2
TSUNO	1
CASTAFIORE	1
TALON	1

LA CLAUSE GROUP BY

Plusieurs fonctions peuvent être utilisées pour manipuler les enregistrements, il s'agit des **fonctions d'agrégations statistiques** dont les principales sont :

- **COUNT()** : pour compter le nombre de lignes concernées
  (Ex.: nombre d'articles achetés par client)
- **SUM()** : pour calculer la somme de plusieurs lignes
  (Ex.: total de tous les achats d'un client)
- **AVG()** : pour calculer la moyenne d'un jeu de valeur
  (Ex.: prix du panier moyen pour de chaque client)
- **MAX()** : pour récupérer la plus grande valeur
  (Ex.: déterminer l'achat le plus cher)
- **MIN()** : pour récupérer la plus petite valeur
  (Ex.: trouver la date du premier achat d'un client)

LA CLAUSE GROUP BY

- La clause **GROUP BY** est donc nécessaire dès que l'on utilise des fonctions de calculs statistiques avec des données brutes
- Cette clause regroupe les lignes en se basant sur la valeur de colonnes spécifiées et renvoie une seule ligne par groupe.

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

SELECT *DeptID, AVG(Salary)*
FROM *Employee*
GROUP BY *DeptID;*

GROUP BY Employee Table using DeptID

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

LA CLAUSE HAVING

- La clause **HAVING** remplace le WHERE sur les opérations résultant des regroupements
- **Syntaxe :**

```
SELECT colonne1, SUM(colonne2)
FROM nom_table
GROUP BY colonne1
HAVING fonction(colonne2) operateur valeur
```

- **Exemple :**

```
SELECT nom, COUNT(id) AS nb
FROM client
GROUP BY nom
HAVING COUNT(id) > 2;
```

NOM	NB
LAGAFFE	2

LA CLAUSE HAVING

- La clause **HAVING** est très souvent utilisé en même temps que **GROUP BY** bien que ce ne soit pas obligatoire

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

SELECT *DeptID, AVG(Salary)*
FROM *Employee*
GROUP BY *DeptID;*

GROUP BY Employee Table using DeptID

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

SELECT *DeptID, AVG(Salary)*
FROM *Employee*
GROUP BY *DeptID*
HAVING *AVG(Salary) > 3000;*

HAVING

DeptID	AVG(Salary)
2	4000.00
3	4250.00

TRAVAUX PRATIQUES 3-2

1. Affichez la somme des salaires et des commissions des employés (cumulez les 2 en une seule colonne)
2. Affichez la moyenne des salaires et la moyenne des commissions des employés
3. Affichez le salaire maximum et la plus petite commission des employés
4. Affichez le nombre distinct de fonction

TRAVAUX PRATIQUES 3-2

5. Écrivez la requête qui permet d'afficher la masse salariale des employés par fonction
6. Affichez le montant de chaque commande, en ne conservant que les commandes qui comportent plus de 5 références de produit
7. Afficher la valeur des produits en stock et la valeur des produits commandés par fournisseur, pour les fournisseurs qui ont un numéro compris entre 3 et 6

LES OPÉRATEURS ENSEMBLISTES

Module 4

LES OPÉRATEURS ENSEMBLISTES

Les **opérations ensemblistes** en SQL sont celles définies dans l'algèbre relationnelle

Elles sont réalisées grâce aux opérateurs :

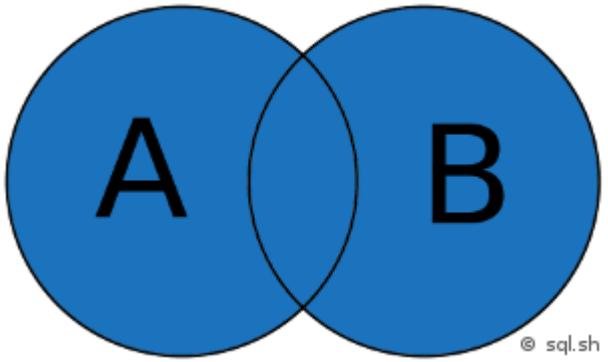
- **UNION**
- **INTERSECT** (n'est pas implémenté dans tous les SGBD)
- **EXCEPT** ou **MINUS** (n'est pas implémenté dans tous les SGBD)

Ces opérateurs s'utilisent entre deux clauses SELECT

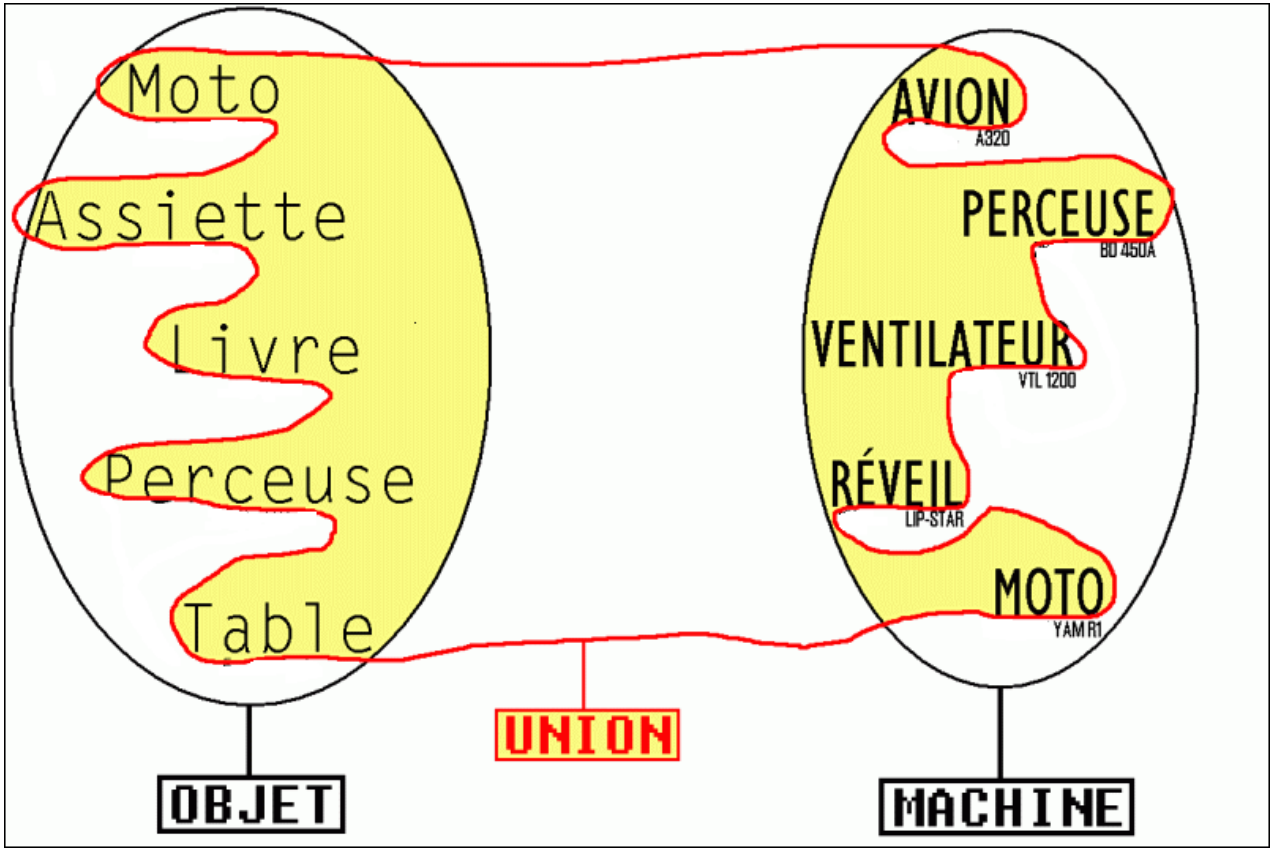
L'OPÉRATEUR UNION

- L'opérateur **UNION** permet de mettre bout-à-bout les résultats de plusieurs requêtes SELECT
- Pour l'utiliser il est nécessaire que chacune des requêtes à concaténer retourne le même nombre de colonnes, avec les mêmes types de données et dans le même ordre
- Par défaut, les doublons seront éliminés : pour effectuer une union dans laquelle même les lignes dupliquées sont affichées il faut plutôt utiliser l'instruction **UNION ALL**
- **Syntaxe :**

```
SELECT * FROM table1
UNION
SELECT * FROM table2
```



L'OPÉRATEUR UNION



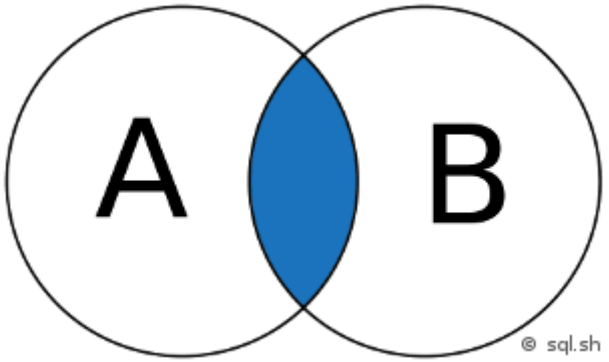
L'OPÉRATEUR INTERSECT

- L'opérateur **INTERSECT** permet de récupérer les enregistrements communs à plusieurs requêtes SELECT
- Pour l'utiliser convenablement il faut que les requêtes retournent le même nombre de colonnes, avec les mêmes types et dans le même ordre
- **Syntaxe :**

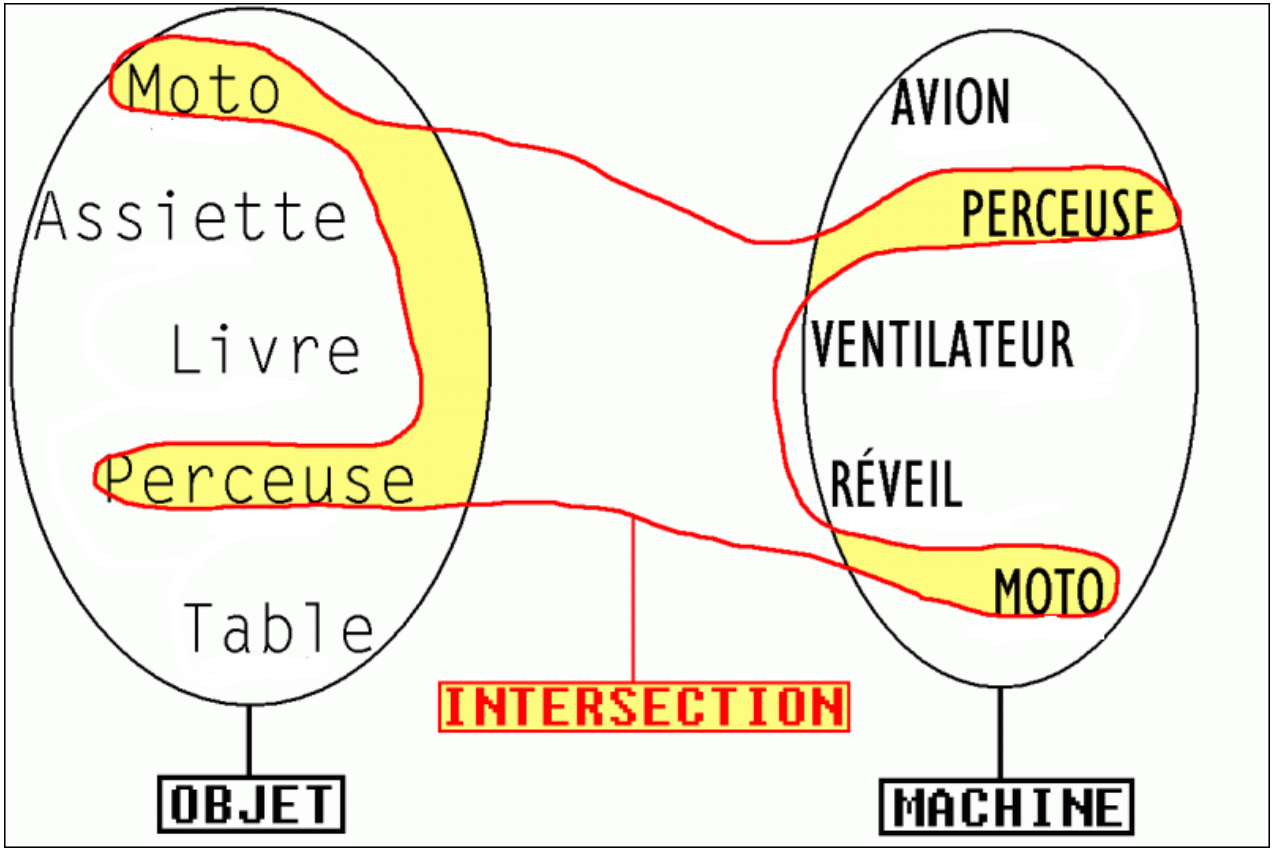
```
SELECT * FROM table1
INTERSECT
SELECT * FROM table2
```

- **Syntaxe alternative :**

```
SELECT a,b FROM table1
WHERE EXISTS (
   SELECT c,d
   FROM table2
   WHERE a=c AND b=d );
```



L'OPÉRATEUR INTERSECT



L'OPÉRATEUR EXCEPT/MINUS

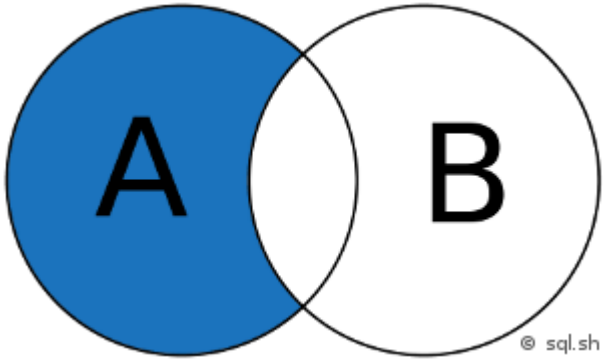
- L'opérateur **MINUS** (Oracle) ou **EXCEPT** (SQL Server) permet d'effectuer une différence entre les enregistrements sélectionnés par deux requêtes SELECT : c'est-à-dire sélectionner les enregistrements de la première table n'appartenant pas à la seconde

- **Syntaxe :**

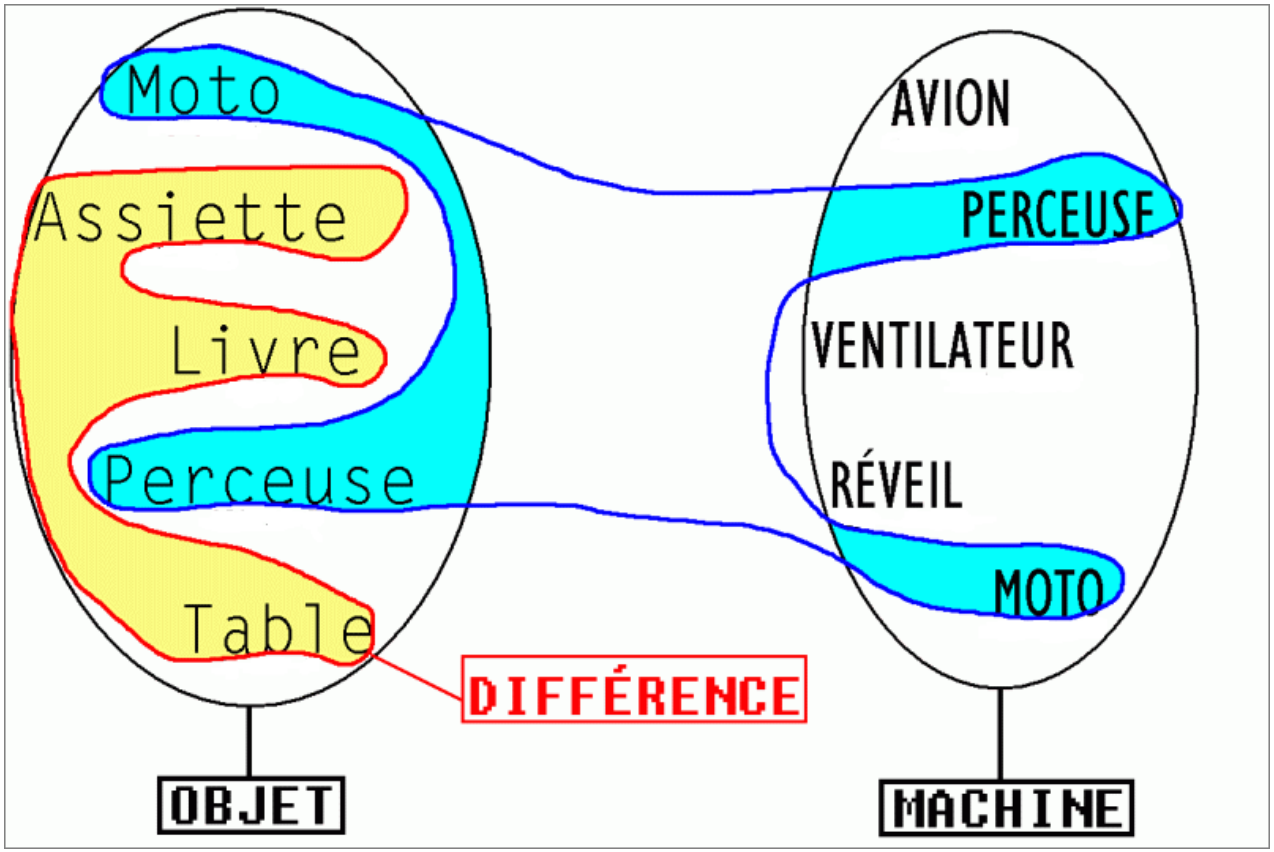
```
SELECT * FROM table1
EXCEPT/MINUS
SELECT * FROM table2
```

- **Syntaxe alternative :**

```
SELECT a,b FROM table1
WHERE NOT EXISTS (
   SELECT c,d
   FROM table2
   WHERE a=c AND b=d );
```



L'OPÉRATEUR EXCEPT/MINUS



TRAVAUX PRATIQUES 4

1. Affichez le nom des sociétés, adresses et villes de résidence pour tous les tiers de l'entreprise (clients et fournisseurs)
2. Affichez toutes les commandes qui comportent en même temps des produits de catégorie 1 du fournisseur 1 et des produits de catégorie 2 du fournisseur 2
3. Affichez la liste des produits que les clients parisiens ne commandent pas

LES SOUS-REQUÊTES

Module 5

SOUS-REQUÊTE DANS LA CLAUSE WHERE

- Dans le langage SQL une sous-requête, également appelée “requête imbriquée” ou “requête en cascade”, consiste à exécuter une requête à l’intérieur d’une autre requête
- Une requête imbriquée est souvent utilisée au sein d’une clause **WHERE** ou de **HAVING** pou remplacer une ou plusieurs constantes
- Il y a plusieurs syntaxes possibles pour utiliser des requêtes dans des requêtes :
  - Requête imbriquée qui retourne un seul résultat (soit une valeur)
  - Requête imbriquée qui retourne une colonne (soit un ensemble de valeurs)

SOUS-REQUÊTE DANS LA CLAUSE WHERE

Requête imbriquée qui retourne un seul résultat

- La syntaxe ci-dessous est typique d'une sous-requête qui retourne un seul résultat à la requête principale :

```
SELECT *
FROM table1
WHERE col1 = (
    SELECT col2
    FROM table2
    WHERE col_pk = val
)
```

- Il est possible d'utiliser n'importe quel opérateur de comparaison tel que =, >, <, >=, <=, != ou <>

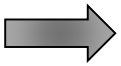
SOUS-REQUÊTE DANS LA CLAUSE WHERE

Requête imbriquée qui retourne un seul résultat

- L'exemple ci-dessous porte sur notre table CLIENT :

```
SELECT id, nom, prenom
FROM client
WHERE id = (SELECT id FROM client WHERE ville = 'Milan')
```

ID	PRENOM	NOM	VILLE
1	Achille	TALON	Paris
2	Bianca	CASTAFIORE	Milan
3	Yoko	TSUNO	Tokyo
4	Gaston	LAGAFFE	Bruxelles
5	Jeanne	LAGAFFE	



ID	NOM	PRENOM
2	CASTAFIORE	Bianca

SOUS-REQUÊTE DANS LA CLAUSE WHERE

Requête imbriquée qui retourne une colonne

- Une requête imbriquée peut aussi retourner une colonne entière :

```
SELECT *
FROM table1
WHERE col1 IN (
    SELECT col2
    FROM table2
    WHERE col_fk = val
)
```

- La requête externe peut alors utiliser l'opérateur **IN** pour filtrer les lignes qui possèdent une des valeurs retournées par la requête interne

SOUS-REQUÊTE DANS LA CLAUSE WHERE

Requête imbriquée qui retourne une colonne

- Une requête imbriquée peut ramener plusieurs lignes à condition que l'opérateur de comparaison admette à sa droite un ensemble de valeurs
- Ajouter **ANY/SOME** ou **ALL** devant les opérateurs classiques =, <>, <, >, <=, ou >= permet de comparer une valeur à un ensemble de valeurs
  - **ANY/SOME** : la comparaison sera vraie si elle est vraie pour au moins un élément de l'ensemble (elle est donc fausse si l'ensemble est vide). On pourra remplacer ANY par la fonction MIN dans la sous-requête
  - **ALL** : la comparaison sera vraie si elle est vraie pour tous les éléments de l'ensemble (elle est vraie si l'ensemble est vide). On pourra remplacer ALL par la fonction MAX dans la sous-requête

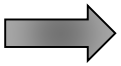
SOUS-REQUÊTE DANS LA CLAUSE WHERE

Requête imbriquée qui retourne une colonne

- L'exemple ci-dessous porte toujours sur notre table CLIENT :

```
SELECT nom, prenom
FROM client
WHERE nom IN ('LAGAFFE', 'TALON')
```

ID	PRENOM	NOM	VILLE
1	Achille	TALON	Paris
2	Bianca	CASTAFIORE	Milan
3	Yoko	TSUNO	Tokyo
4	Gaston	LAGAFFE	Bruxelles
5	Jeanne	LAGAFFE	



NOM	PRENOM
TALON	Achille
LAGAFFE	Gaston
LAGAFFE	Jeanne

SOUS-REQUÊTE DANS LA CLAUSE FROM

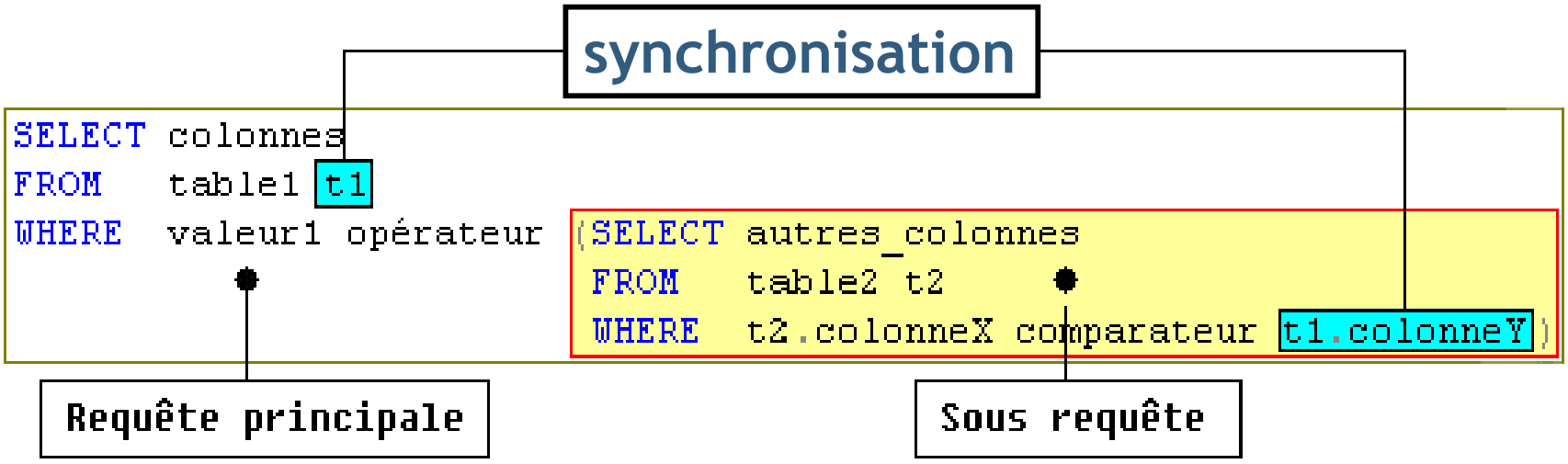
- Une requête renvoyant une table peut être imbriqué dans la clause **FROM** d'une autre requête : il convient donc de lui donner systématiquement un **nom de corrélation** pour pouvoir y faire référence
- **Syntaxe :**
- ```
  SELECT t1.x, t1.y, t2.z
  FROM table1 t1, (SELECT x, z FROM table2) t2
  WHERE t1.x = t2.x;
  ```
- **Ou :**
- ```
  SELECT t1.x, t1.y, t2.z
  FROM table1 t1
  JOIN (SELECT x, z FROM table2) t2 ON t1.x = t2.x;
  ```

SOUS-REQUÊTE DANS LA CLAUSE FROM

- **Exemple :**
  Part du salaire de chaque employé par rapport à la masse salariale :
- ```
  SELECT e.nom, e.salaire, ROUND(e.salaire/ms.masse*100)
  FROM employes e,
      (SELECT SUM(salaire) masse
      FROM employes) ms;
  ```
- **Ou :**
- ```
  SELECT nom, salaire, salaire/SUM(salaire) OVER()*100
  FROM employes;
  ```
- La fonction de fenêtrage **OVER**() ne marche pas sous MySQL

SOUS-REQUÊTE SYNCHRONISÉE

- Une **sous-requête synchronisée** est une sous-requête qui s'exécute pour chaque ligne de la requête principale et non une fois pour toute
- Pour arriver à ce résultat, il suffit de faire varier une condition en rappelant dans la sous-requête la valeur de la ou des colonnes de la requête principale qui doit servir de condition



TRAVAUX PRATIQUES 5

1. Affichez tous les produits pour lesquels la quantité en stock est inférieure à la moyenne des quantités en stock
2. Affichez toutes les commandes pour lesquelles les frais de ports dépassent la moyenne des frais de ports pour ce client
3. Affichez les produits pour lesquels la quantité en stock est supérieure à la quantité en stock de chaque produit de catégorie 3

LA MANIPULATION DES DONNÉES (LMD)

Module 6

LE LANGAGE DE MANIPULATION DE DONNÉES (LMD)

- Le **langage de manipulation de données** (LMD) est le langage SQL qui permet de modifier les valeurs contenues dans la base de données
- Il existe trois clauses permettant d'effectuer la modification des données :
  - **INSERT INTO** pour ajouter des lignes
  - **UPDATE** pour modifier des lignes
  - **DELETE** pour supprimer des lignes
- Ces trois clauses peuvent être effectuées dans le cadre d'une **transaction**
- En mode transactionnel, on peut valider les modifications avec **COMMIT** ou les annuler avec **ROLLBACK**

L'INSERTION DE DONNÉES

- L'insertion de données dans une table s'effectue à l'aide de la commande **INSERT INTO** qui permet d'ajouter une seule ligne à la fois dans la table existante ou plusieurs lignes d'un seul coup dans la même table
- **Syntaxe (une ligne) :**

```
INSERT INTO table1 (col1, col2, …)
VALUES (val1, val2, …);
```

- **Syntaxe (plusieurs lignes) :**

```
INSERT INTO table1 (col1, col2, …)
VALUES
  (val1, val2, …)
  (val11, val12, …;
```

- Lorsque la colonne à remplir est de type **texte** ou **date**, il faut mettre la valeur entre simple quote. En revanche, lorsque la colonne est **numérique** il suffit juste d'indiquer le nombre

L'INSERTION DE DONNÉES

- **Syntaxe (avec SELECT) :**

```
INSERT INTO table1 (col1, col2, …)
SELECT col11, col12, …
FROM table2;
```

- La liste des colonnes (**col1, col2, …**) est optionnelle
- Si elle est omise alors le moteur prendra par défaut l'ensemble des colonnes de la table **dans l'ordre** où elles ont été données lors de la création de la table
- Si une liste de colonnes est spécifiée alors les colonnes ne figurant pas dans la liste auront la valeur **NULL**

L'INSERTION DE DONNÉES

- **Exemple :**

```
INSERT INTO client (id, prenom, nom, ville, age)
VALUES
    (6, 'Benoît', 'BRISEFER', 'Vivejoie-la-Grande', 11) ,
    (7, 'Guy', 'LEFRANC', 'Bruxelles', 31) ,
    (8, 'Adèle', 'BLANC-SEC', 'Meudon', 24) ;
```

ID	PRENOM	NOM	VILLE	AGE
1	Achille	TALON	Paris	35
…	…	…	…	
5	Jeanne	LAGAFFE		27
6	Benoît	BRISEFER	Vivejoie-la-Grande	11
7	Guy	LEFRANC	Bruxelles	31
8				

LA MODIFICATION DE DONNÉES

- La clause **UPDATE** permet d'effectuer des modifications sur des lignes existantes dans la table : elle est souvent utilisée avec la clause **WHERE** pour spécifier sur quelles lignes doivent porter les modifications
- **Syntaxe :**

```
UPDATE table1
SET col1 = val1, col2 = val2, …
WHERE predicat;
```

- **Ou :**

```
UPDATE table1
SET (col1, col2, …) = (SELECT col11, col12, … FROM table2)
WHERE predicat;
```

- Les valeurs des colonnes sont mises à jour dans toutes les lignes satisfaisant le prédicat
- La clause **WHERE** est facultative : si elle est omise alors toutes les lignes seront mises à jour

LA MODIFICATION DE DONNÉES

- **Exemple 1 :**
  changer l'âge et la ville de résidence de Gaston LAGAFFE

```
UPDATE client
SET age = 28, ville = 'Namur'
WHERE id = 4;
```

- **Exemple 2:**
  Donner à tous les clients sans âge la moyenne d'âge des clients

```
UPDATE client
SET age = (SELECT FLOOR(AVG(age)) FROM client)
WHERE age IS NULL;
```

LA SUPPRESSION DE DONNÉES

- La commande **DELETE** en SQL permet de supprimer des lignes dans une table
- En utilisant cette commande associé à **WHERE** il est possible de sélectionner les lignes concernées qui seront supprimées
- **Syntaxe :**

```
DELETE FROM table
WHERE predicat;
```

- **Attention !** La clause WHERE est facultative : si elle est omise alors toutes les lignes de la table seront supprimées

LA SUPPRESSION DE DONNÉES

- **Exemple 1:**
  Supprime les clients agés de moins de 18 ans
  ```
  DELETE FROM client
  WHERE age < 18;
  ```
- **Exemple 2 :**
  Supprime tous les clients
  ```
  DELETE FROM client;
  ```

TRAVAUX PRATIQUES 6

1. Insérez une nouvelle catégorie de produits nommée « Produits bios » ayant comme description « Bio c'est bon ! » et respectant les contraintes
2. Créez un nouveau fournisseur « Grandma » (NO_FOURNISSEUR = 30) avec exactement les mêmes coordonnées que le fournisseur « Grandma Kelly's Homestead »
3. Attribuer les produits de « Grandma Kelly's Homestead » au nouveau fournisseur précédemment créé
4. Supprimez l'ancien fournisseur « Grandma Kelly's Homestead »

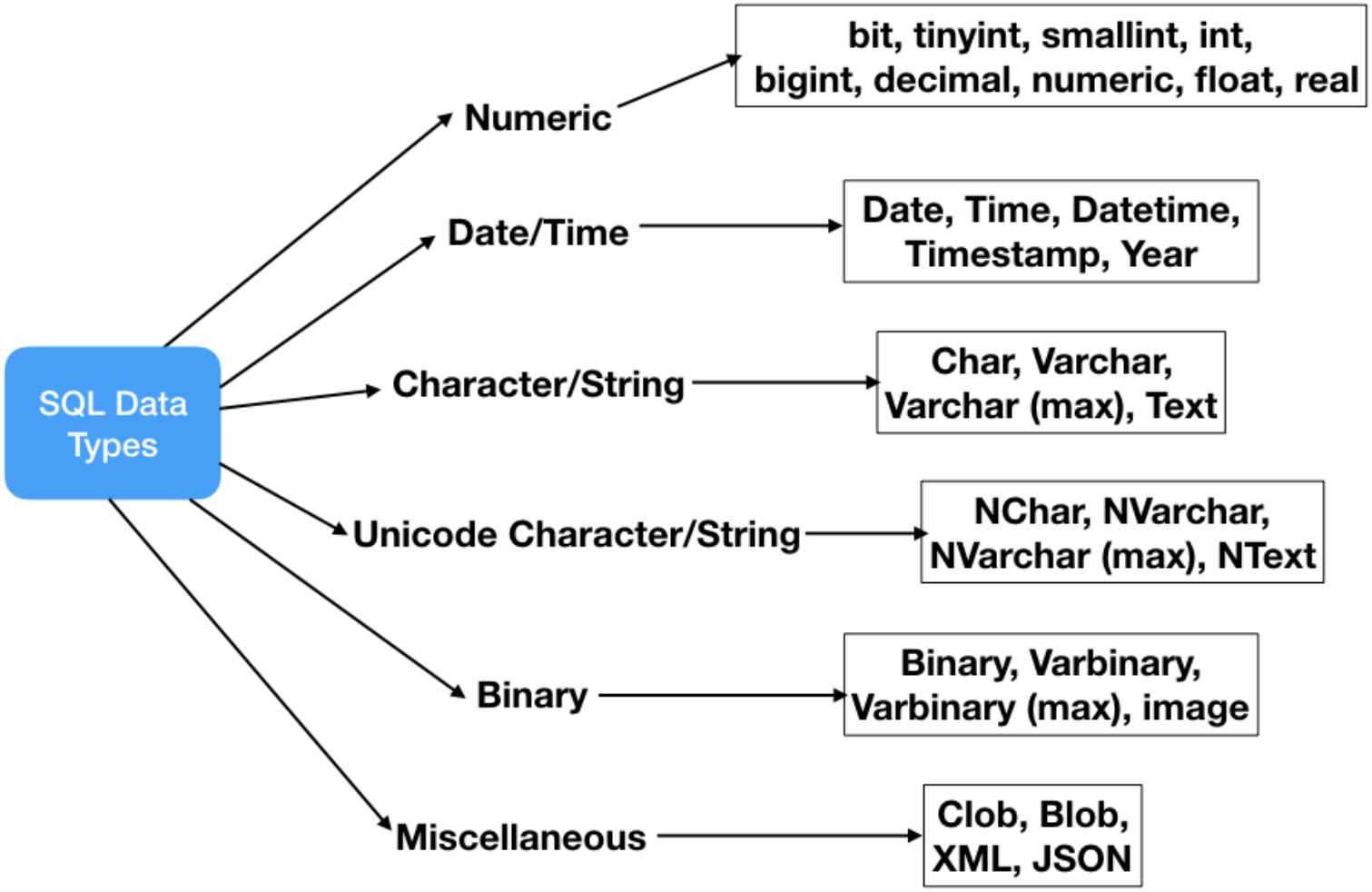
LA DÉFINITION DES DONNÉES (LDD)

Module 7

LES TYPES DE DONNÉES

- En SQL, comme dans la plupart des langages informatiques, les données sont séparées en plusieurs **types**
- Lorsque l'on crée une colonne dans une table, on lui affecte un type : toute valeur stockée dans cette colonne devra alors correspondre au type de la colonne
- Les types de données SQL sont principalement classés en **6 catégories** :
  1. Numériques (entier et réels)
  2. Date et heure
  3. Chaînes de caractères
  4. Caractères Unicode
  5. Binaire
  6. Autres

LES TYPES DE DONNÉES

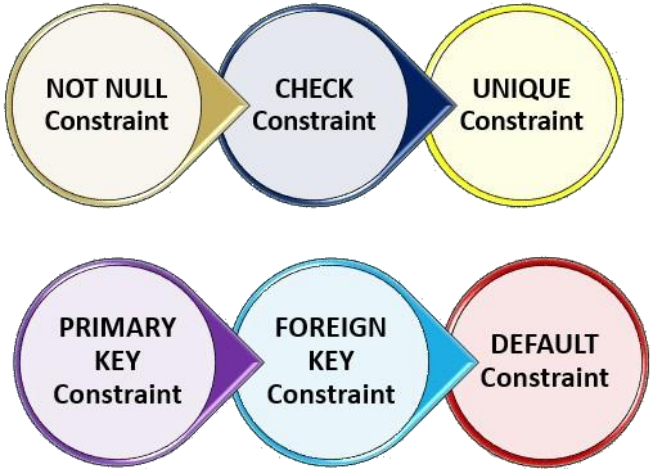


LES CONTRAINTES

- Les **contraintes** permettent de limiter le type de données à insérer dans une table : elles peuvent être définies lors de la création/modification d'une table par l'intermédiaire de l'instruction **CONSTRAINT**

- Une colonne peut recevoir les contraintes suivantes :
  - **NULL/NOT NULL** : précise si une valeur est obligatoire dans la colonne ou pas
  - **UNIQUE** : impose que toute valeur de la colonne doit être unique – *Génère un index*
  - **CHECK** : permet de préciser un prédicat qui acceptera la valeur si elle est vérifiée
  - **PRIMARY KEY** : précise si la colonne est la clé primaire de la table – *Génère un index*
  - **FOREIGN KEY** : permet, pour les valeurs de la colonne, de faire référence à des valeurs existantes dans une colonne clé primaire d'une autre table : ce mécanisme s'appelle intégrité référentielle
  - **DEFAULT** : valeur par défaut placée dans la colonne lors des insertions et de certaines opérations particulières, lorsque l'on n'a pas donné de valeur explicite à la colonne

LES CONTRAINTES



Il existe deux types de contraintes :

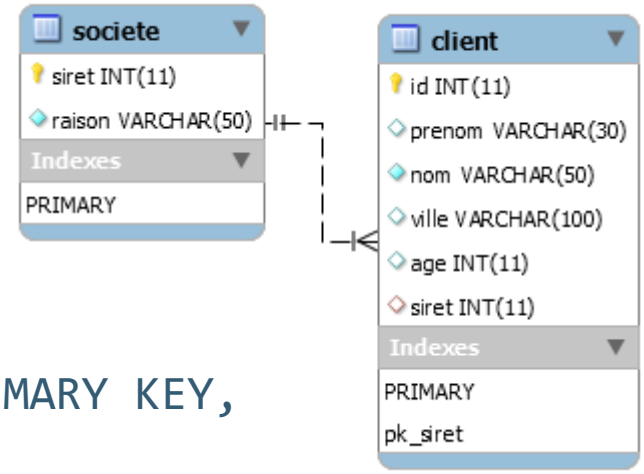
- **Contraintes de colonnes** (porte sur une seule colonne) :
  - [NOT] NULL
  - UNIQUE
  - PRIMARY KEY
  - CHECK (prédicat_de_colonne)
  - FOREIGN KEY [colonne] REFERENCES table (colonne)
- **Contraintes de table** (porte sur une ou plusieurs colonnes) :
  - CONSTRAINT nom_contrainte UNIQUE
  - PRIMARY KEY (liste_colonnes)
  - CHECK (prédicat_de_table)
  - FOREIGN KEY liste_colonnes REFERENCES nom_table (liste_colonnes)

CRÉATION DE TABLE

- Une **table** est un objet qui est contenu dans le schéma d'une base de données pour y stocker des valeurs rangées dans des colonnes
- La création d'une table permet de définir les colonnes, le type des données qui seront contenus dans chacune des colonnes (entier, chaîne de caractères, date, valeur binaire, …) et leurs contraintes
- **Syntaxe :**

```
CREATE TABLE nom_de_la_table (
   col1 type_donnees [contrainte_de_colonne],
   col2 type_donnees [contrainte_de_colonne],
   col3 type_donnees [contrainte_de_colonne],
   [contrainte_de_table]
);
```

CRÉATION DE TABLE



- **Exemple :**

```
CREATE TABLE societe (
   siret INTEGER AUTO_INCREMENT PRIMARY KEY,
   raison VARCHAR(50) NOT NULL
);


CREATE TABLE client (
   id INTEGER AUTO_INCREMENT PRIMARY KEY,
   prenom VARCHAR(30),
   nom VARCHAR(50) NOT NULL,
   ville VARCHAR(100) CHECK (ville <> ''),
   age INT DEFAULT 0,
   siret INTEGER,
   CONSTRAINT ck_age CHECK (age BETWEEN 18 AND 65),
   CONSTRAINT pk_siret FOREIGN KEY (siret)
      REFERENCES societe(siret)
);
```

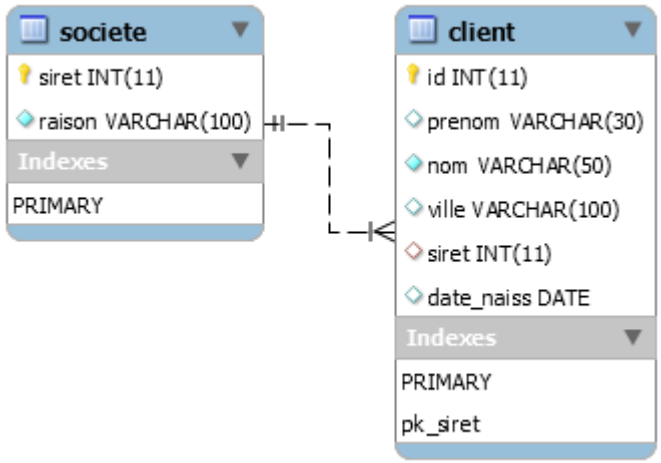
MODIFICATION DE TABLE

- Le langage SQL permet également de modifier une table existante pour y ajouter, supprimer ou modifier une colonne existante
- **Ajout de colonne :**
```
ALTER TABLE nom_table
ADD col1 type_donnees [contrainte_de_colonne];
```
- **Modification de colonne - Oracle et MySQL :**
```
ALTER TABLE nom_table MODIFY col1 type_donnees;
```
- **Modification de colonne - Oracle et MySQL :**
```
ALTER TABLE nom_table ALTER COLUMN col1 type_donnees;
```
- **Suppression de colonne :**
```
ALTER TABLE nom_table DROP COLUMN col1;
```
- **Attention !** La colonne à supprimer ne doit pas être référencée par une clé étrangère ou être utilisée par un index

MODIFICATION DE TABLE

- Des contraintes d'intégrité peuvent être ajoutées, modifiées ou supprimées par la clause **ALTER TABLE** : on ne peut ajouter que des contraintes de table
- **Ajout de contrainte :**
```
ALTER TABLE nom_table ADD CONSTRAINT const1;
```
- **Modification de contrainte :**
```
ALTER TABLE nom_table
MODIFY CONSTRAINT const1 ENABLE|DISABLE;
```
- **Modification de contrainte :**
```
ALTER TABLE nom_table
RENAME CONSTRAINT const1 TO const2;
```
- **Suppression de contrainte :**
```
ALTER TABLE nom_table DROP CONSTRAINT const1;
```

MODIFICATION DE TABLE



- **Exemple 1 :**

```
ALTER TABLE client
ADD date_naiss DATE;
```

- **Exemple 2 :**

```
ALTER TABLE societe
MODIFY raison VARCHAR(100) NOT NULL;
```

- **Exemple 3 :**

```
ALTER TABLE client
ADD CONSTRAINT ck_date_naiss CHECK (date_naiss BETWEEN
'2000-01-01' AND '2020-12-31');
```

- **Exemple 4 :**

```
ALTER TABLE client
DROP COLUMN age;
```

SUPPRESSION DE TABLE

- SQL permet enfin de supprimer définitivement une table d'une base de données en supprimant en même temps les éventuels index, trigger, contraintes et permissions associés à cette table
- **Attention !** Utiliser cette ordre avec attention car une fois supprimée, les données sont perdues définitivement
- **Syntaxe :**
  ```
  DROP TABLE nom_table;
  ```
- S'il y a une **dépendance** avec une autre table, il est conseillé de la supprimer avant : c'est le cas par exemple s'il y a des clés étrangères

CRÉATION DE VUE

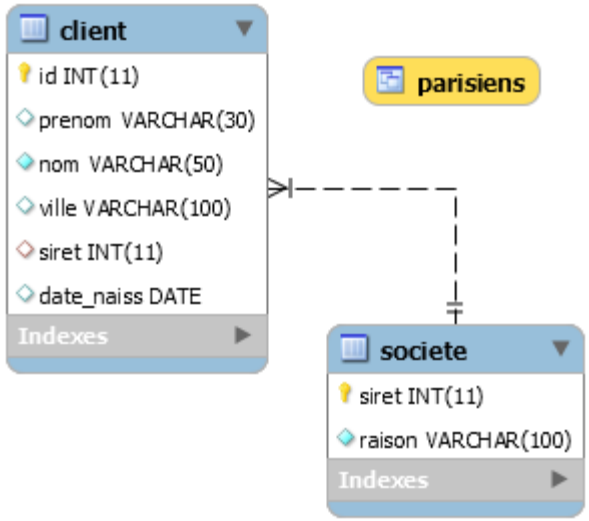
- Une **vue** peut être considérée comme une **table virtuelle**
  - Une table contient un jeu de définitions et est destinée à stocker physiquement les données
  - Une vue contient aussi un jeu de définitions - créé au-dessus des tables ou d'autres vues - mais elle ne stocke pas physiquement les données

- **Syntaxe :**

```
CREATE VIEW nom_vue AS
    SELECT …
    FROM …
    [WHERE …]
```

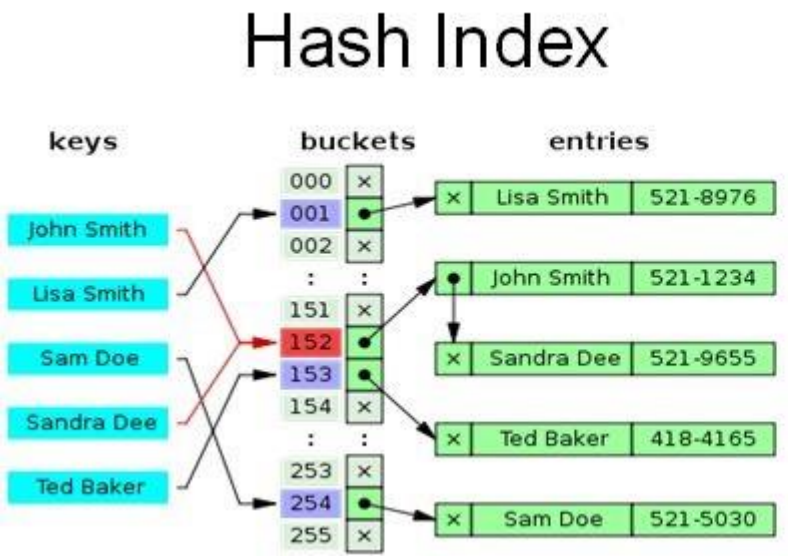
- **Exemple :**

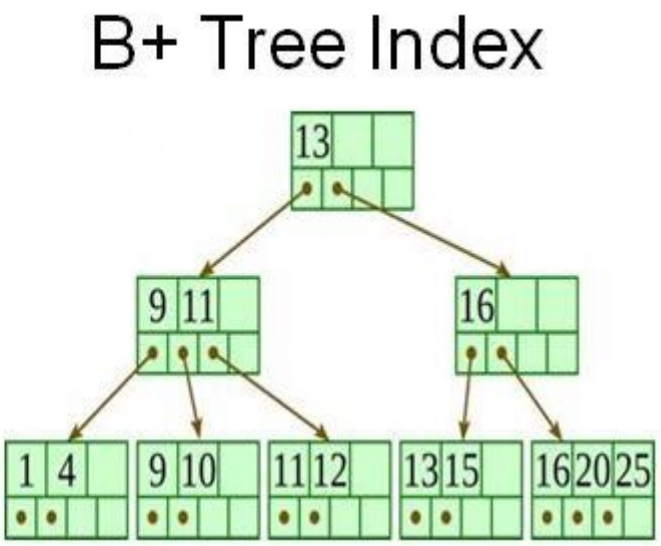
```
CREATE VIEW parisiens AS
    SELECT *
    FROM client
    WHERE ville = 'Paris'
;
```



CRÉATION D'INDEX

- Un **index** est utile pour accélérer l'exécution d'une requête SQL qui lit des données dans une table et ainsi améliorer les performances d'une application utilisant une base de données





CRÉATION/SUPPRESSION D'INDEX

- Lors de la création d'un index, on peut spécifier par l'option UNIQUE que chaque valeur d'index doit être unique dans la table
- **Syntaxe – Création :**

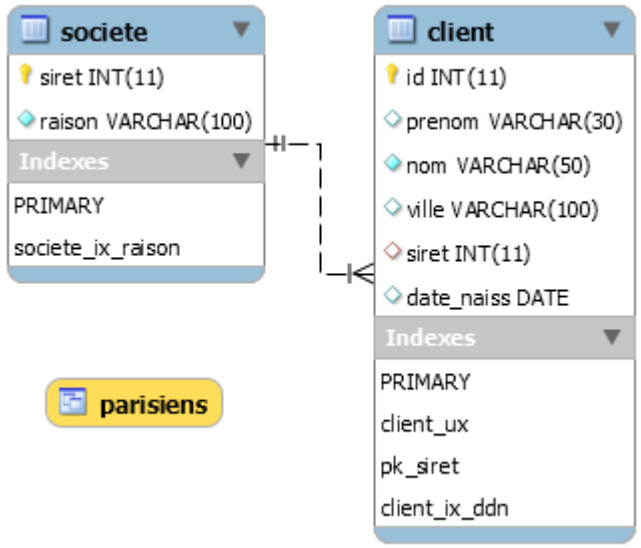
```
CREATE [UNIQUE] INDEX nom_index
ON nom_table (col1, col2, …);
```

- Lors de la suppression d'un index, le nom de la table ne doit être précisé que si l'on veut supprimer l'index d'une table d'un autre schéma alors que l'on possède un index du même nom
- **Syntaxe – Suppression :**

```
DROP INDEX nom_index [ON nom_table];
```

CRÉATION/SUPPRESSION D'INDEX

- **Exemple 1 :**
```
CREATE UNIQUE INDEX client_ux
ON client(nom, prenom);
```
- **Exemple 2 :**
```
CREATE INDEX client_ix_ddn
ON client(date_naiss);
```
- **Exemple 3 :**
```
CREATE INDEX societe_ix_raison
ON societe(raison);
```
- **Exemple 4 :**
```
DROP INDEX societe_ix_raison
ON societe;
```



TRAVAUX PRATIQUES 7

1. Créez une table PAYS avec 2 champs :
   a. code pays (4 caractères et clé primaire)
   b. nom pays (40 caractères maximum)
2. Ajoutez une colonne COURRIEL (60 caractères) à la table CLIENTS
   a. Modifiez-la pour la passer à 75 caractères
   b. Pour finir, supprimez cette colonne
3. Créez une vue PROVINCE qui affiche le nom de la société, l'adresse, le téléphone et la ville des clients domiciliés en province