

## **Part 2**

# **BASES DE DONNÉES RELATIONNELLES**

## Introduction

## Stockage des données

- Plusieurs applications nécessitent le stockage des données pour de longues périodes.
- L'information peut être stockée:
  - dans des **fichiers**
    - Ne nécessite aucun logiciel supplémentaire
    - Facile à utiliser
    - Ne nécessite pas beaucoup d'espace disque
  - dans des **base de données**
    - Plus sécurisées
    - Les données sont mieux organisées et donc plus facile à manipuler
    - Possibilité de gérer les accès concurrents

## Inconvénients des fichiers

- Redondance des données et incohérence
  - même information dupliquée dans **plusieurs fichiers**
  - **mise à jour** dans un fichier et pas dans les autres
- **Difficulté** à accéder aux données
- **Isolation** des données: plusieurs fichiers, plusieurs formats
- Problèmes **d'intégrité**
  - contraintes d'intégrité codées dans des programmes
  - difficile d'en ajouter ou de les changer
- Problèmes **d'atomicité**
  - si crash du système durant l'exécution d'un programme, état incohérent
- Problèmes de **concurrency**
  - anomalies si accès concurrents
- Problèmes de **sécurité**

## Définitions

### ■ Base de Données (BD):

<b>Gros</b>	par rapport à la mémoire humaine!
<b>ensemble</b>	notion mathématique. . . formalisme
<b>persistant</b>	dans le temps, pannes
<b>de données</b>	informations, typées et multimédia
<b>structurées</b>	organisées, liées
<b>et cohérentes</b>	contraintes d'intégrité déclarées et forcées
<b>exploitable</b>	interrogation, modifications, évolution
<b>simultanément</b>	parallélisme des accès, partage

### ■ Système de Gestion Bases de Données (SGBD):

Logiciel assurant définition, structuration, stockage, maintenance, mise à jour et consultation de BDs

## Exemples de BD

- Gestion des personnels, étudiants, cours, inscriptions, ... De l'université
- Système de réservation de places d'avion chez Royal Air Maroc, de places de train à la ONCF
- Gestion des comptes clients de la banque
- Gestion des commandes chez Amazon.com
- Gestion d'une bibliothèque universitaire
- Gestion des pages Web chez google.com
- ...

## Avantages des BDs (1)

- **Indépendance physique :**

- disques, machines, méthodes d'accès non apparents

- **Indépendance logique :**

- vue partielle et présentation suivant les besoins utilisateur

- **Cohérence des données :**

- non-redondance et contraintes sur les valeurs possibles des données

- **Partage des données :**

- détection et contrôle des conflits d'accès

## Avantages des BDs (2)

- **Souplesse d'accès aux données :**

- faciliter l'interrogation (l'utilisateur doit dire ce qu'il veut, pas comment l'obtenir)

- **Performances :**

- minimiser les accès aux disques (volume/temps max.)

- **Sécurité, administration et contrôle global :**

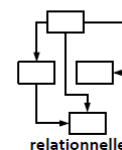
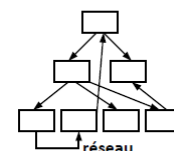
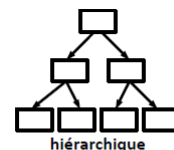
- autorisation et stratégie d'accès
- points de reprise, sauvegarde, contrôle de la cohérence, ...

## Utilisateurs de BD

- Plusieurs types d'utilisateur:
  - **Utilisateurs naïfs** : accèdent à la BD à travers une interface (ex. sur le web)
  - **Utilisateurs sophistiqués** : accèdent à la BD en écrivant des requêtes en Langage de Manipulation de Données (LMD)
  - **Programmeurs d'application** : accèdent à la BD en écrivant des programmes d'application
  - **Administrateur BD ("DBA")** : donne les droits d'accès aux utilisateurs, définit la structure de stockage et l'organisation physique de la BD, surveille les performances du système, etc.

## Différents types de bases de données

- **Base hiérarchique**
  - Contenu organisé dans une structure arborescente.
- **Base en réseau**
  - Hiérarchique mais avec des relations transverses.
- **Base relationnelle**
  - Informations organisées dans des matrices appelées relations ou tables.
- **Base XML**
  - S'appuie sur le modèle fourni par XML.
- **Base objet**
  - Informations groupées sous forme de collections d'objets. Chaque donnée est active et possède ses propres méthodes d'interrogation et d'affectation.



## Historique (1)

### ■ **Années 60:**

- ❑ Le 1er SGBD à caractère général et basé sur le **modèle réseau**
- ❑ IBM développe IMS qui est basé sur le **modèle hiérarchique**

### ■ **Années 70:**

- ❑ Edgar Codd définit le **modèle relationnel**
  - obtient en 1981 le prix “ACM Turing Award”
  - IBM lance le SGBD “System R” (prototype)
  - UC Berkeley lance le SGBD “Ingres” (prototype)
- ❑ Peter Chen définit le modèle entité-association

### ■ **Années 80:**

- ❑ Les SGBD relationnels prototypes deviennent des systèmes commercialisés, SQL devient un standard
- ❑ SGBD parallèles et distribués et SGBD orientés objet

## Historique (2)

### ■ **Années 90:**

- ❑ Applications de fouille de données (“data mining”)
- ❑ Grands entrepôts de données (“data warehouses”)
- ❑ Commerce électronique (“e-commerce”)

### ■ **Années 2000:**

- ❑ **Modèle** semi-structuré (**XML**, XQuery)
- ❑ Nouveaux types de données dans les BDR (image, texte, etc)
- ❑ Administration automatisée des BD

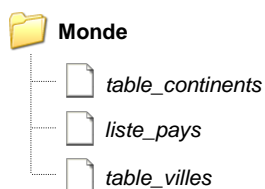
### ■ **Fin des années 2000:**

- ❑ **Modèle NoSQL (Not Only SQL)** basé sur le notion clé-valeur
- ❑ Technologie liée au Cloud Computing
- ❑ Les principaux site du Web utilisent une BD NoSQL: Facebook, Google, Yahoo et Amazon

# BD relationnelle : Terminologie

- Une **BD relationnelle** est un **ensemble de données organisées** sous forme de **tables** (appelée aussi **relation**) qui décrivent les types des données.
- En utilisant un **identifiant commun** (clé) entre les tables il est possible de les « **relier** » entre elles.
- Une relation est une table comportant des **colonnes** (dits aussi **attributs**) dont le nom et le type caractérise le contenu qui sera inséré dans la table.
- Les tables contiennent à leur tour des lignes (appelées aussi des **enregistrements** ou **tuples**) qui sont les vraies données.

# BD relationnelle



## Exemple : Base de données 'Monde'

- Table 1: 'table\_continents'  
Colonnes de la Table 1: 'continent', 'description'
- Table 2: 'liste\_pays'  
Colonnes de la Table 2: 'continent', 'pays', 'carte'
- Table 3: 'table\_villes'  
Colonnes de la Table 3: 'pays', 'ville', 'nb\_habitants'

## Exemple

- Imaginons que l'on veuille stocker dans notre base de données notre carnet d'adresses. On va donc créer la **relation** (table) **Personne** qui aura pour **attributs** : **nom**, **prénom**, **email**, **département**.

**Personnes**

Nom	Prénom	EMail	Département
El Hannani	Asmaa	elhannani.a@ucd.ac.ma	TRI
Tata	Titi	tata.t@ucd.ac.ma	Info

**Attribut** (pointing to 'Prénom')

**Valeurs de l'attribut prises à partir du domaine associé à l'attribut** (pointing to 'Asmaa' and 'Titi')

**Enregistrement : ensemble de valeurs correspondant à une occurrence particulière de l'objet représentée par la table** (pointing to the first row)

## Relation

- Chaque **relation** ou **table** a un nom unique dans la BD
- Une relation représente soit :
  - un **type d'objet** d'affaires spécifique sur lequel l'entreprise désire conserver de l'information  
ex: Coureur, Course
  - ou **une association** logique entre deux objets  
ex: Résultat – représente l'association logique entre les coureurs et les courses auxquelles ils ont participé
- **Synonymes**
  - relation, table, entité



## Enregistrement

- Chaque **ligne** correspond à un enregistrement.
- Chaque enregistrement est unique dans une relation.
- Un enregistrement est formé par l'ensemble des valeurs des attributs décrivant une occurrence particulière de l'objet d'affaire représenté par la relation.
- Chaque enregistrement contient exactement une valeur pour chaque attribut.
- L'ordre des enregistrements n'a pas d'importance.
- Le nombre des enregistrements est variable dans le temps.
- **Synonymes**
  - Enregistrement, tuple

## Attribut

- Chaque **colonne** d'une relation correspond à un attribut et à ses valeurs.
- L'ordre des colonnes dans une relation n'a pas d'importance.
- Le nom d'un attribut est unique dans une relation.
- Le nom d'attribut décrit un aspect de l'objet d'affaires.  
Ex: « nomcoureur » dans la relation « coureur »
- Un attribut prend des valeurs à partir d'un domaine donné.
- Chaque valeur d'un attribut est atomique.
- **Synonymes**
  - attribut, colonne, champ

# Langages de BD

- La plupart de SGBDR sont accessibles via le Langage structuré de requêtes (**SQL**):
  - **Langage de Définition de Données (LDD)**
    - Créer, modifier, et supprimer des tables
  - **Langage de Manipulation de Données (LMD)**
    - Insérer, modifier, supprimer des enregistrements
    - Extraire des données (requêtes)
  - **Langage de Contrôle de Données (LCD)**
    - Créer, modifier, supprimer des droits d'accès
- Un programme d'application accède à la BD par:
  - Un langage qui supporte du SQL imbriqué (ex: Pro\*C)
  - Une API (Application Program Interface) (ex: JDBC) qui permet d'envoyer des requêtes SQL à la BD d'une application Java

# Exemple de SGBDR

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>■ <b>Sharewares :</b><ul style="list-style-type: none"><li>□ Oracle</li><li>□ DB2 (IBM)</li><li>□ Ingres</li><li>□ Informix</li><li>□ Sybase</li><li>□ SQL Server (Microsoft)</li><li>□ O2</li></ul></li><li>■ <b>Sur micro :</b><ul style="list-style-type: none"><li>□ Access</li><li>□ Paradox</li><li>□ FoxPro</li><li>□ 4D</li><li>□ Windev</li></ul></li></ul> | <ul style="list-style-type: none"><li>■ <b>Freewares :</b><ul style="list-style-type: none"><li>□ MySQL</li><li>□ MSQL</li><li>□ Postgres</li><li>□ InstantDB</li></ul></li></ul> |
|--|---|

## Exemple de SGBD:

### ■ Oracle

#### □ Avantages :

- supporte les très grosses bases, pérennité (40% de part de marché), interface utilisateur très riche,
- procédures en PL/SQL ou JAVA (un + pour les équipes de développement),
- flashback query (retour à une date), système de vues, services web...

#### □ Inconvénients :

- coût excessif,
- fort demandeur de ressources,
- gestion des rôles et des privilèges instable, ...

## Exemple de SGBD

### ■ MySQL

#### □ Avantages :

- très courant chez les hébergeurs, très bonne intégration Apache/PHP,
- OpenSource,
- version cluster (plusieurs serveurs reliés entre eux),
- facilité de prise en main.

#### □ Inconvénients :

- faible richesse fonctionnelle (contrôle d'intégrité),
- peu robuste avec des gros volumes,
- pas d'héritage de tables.

#### □ C'est le SGBD qu'on va utiliser dans le cadre de ce cours !

## Exemple de SGBD

### ■ PostgreSQL

#### □ Avantages :

- OpenSource,
- fiable et relativement performant,
- riche fonctionnellement,
- simple d'utilisation et d'administration,
- héritage de tables

#### □ Inconvénients :

- supporte les bases de moyenne importance,
- sauvegardes peu évoluées,
- gestion des permissions limitées

## Exemple de SGBD

### ■ SQLite

#### □ Avantages :

- OpenSource,
- le + petit SGBDR,
- simple d'utilisation et d'administration,
- installation aisée

#### □ Inconvénients :

- fonctionnalités minimales,
- pas d'intégrité référentielle,
- LDD très limité (à part ajouter une colonne)

## Outils à utiliser

- Nous utiliserons **phpMyAdmin** et/ou **MySQL Workbench**
  - **phpMyAdmin** est développé en PHP et offre une interface intuitive pour l'administration et la manipulation des bases de données du serveur MySQL.
  - **MySQL Workbench** est un outil graphique de conception et de modélisation de bases de données relationnelles pour accéder au serveur MySQL. Il simplifie la tâche de création et de modification de modèles de données.

## Structured Query Language (SQL)

# SQL

- Le **Structured Query Language** est un langage standardisé qui permet d'effectuer des opérations sur des bases de données.
- Il se compose de:
  1. **Langage de Définition de Données (LDD)** permettant de gérer les structures de la base
  2. **Langage de Manipulation de Données (LMD)** pour interagir avec les données.
  3. **Langage de Contrôle de Données (LCD)** pour Créer, modifier, supprimer des droits d'accès
- **Attention:** certaines syntaxes ou fonctions sont propres au système de gestion de base de données utilisé.

## SQL:

**Langage de Définition de Données**

**Langage de Manipulation de Données**

## Le langage de définition de données

- Les commandes principales sont :
  - Pour les bases : `create`, `alter`, `drop`
  - Pour les tables : `create`, `alter`, `rename`, `drop`
  - Pour les indexes : `create`, `drop`
  - La gestion des vues, des triggers, des events
- Ces fonctions sont assez peu utilisées car des outils tels que `phpMyAdmin` permettent d'effectuer directement ces opérations.

## Syntax: Identificateurs

- Les noms de bases, relations, attributs, index et alias sont constitués de caractères alphanumériques plus `_` et `$`.
- Un nom comporte au maximum 64 caractères.
- MySQL code le nom des bases et des tables directement dans le système de fichiers ! Attention donc à la casse ....
- Le point `"."` est un caractère réservé utilisé comme séparateur entre le nom d'une base, celui d'une relation et celui d'un attribut :

`Select base1.table2.attribut3 from base1.table2`

## Créer une BD (I)

- Il est possible de créer autant de bases de données que nécessaires.
- Généralement, une base regroupe toutes les données nécessaires pour un besoin fonctionnel précis.
- En clair, on peut résumer : une application, une base de données.
- L'interaction entre les bases de données est possible mais alourdit la syntaxe SQL.

## Créer une BD

- Création d'une base de données :

```
CREATE DATABASE [IF NOT EXISTS] Nom_base  
[create_specification]
```

- Les spécifications permettent notamment de définir le système d'encodage des caractères, et le moteur de la table (ENGINE).



## Suppression/Modification d'une BD

- **Suppression** d'une base de données :

`DROP DATABASE [IF EXISTS] db_name`

- **Modification** d'une base de données :

`ALTER DATABASE db_name alter_specification [, alter_specification] ...`

## Créer une table (I)

- La création d'une relation utilise la commande **CREATE TABLE** selon la syntaxe suivante :

```
CREATE [TEMPORARY] TABLE nom_relation [IF NOT EXISTS] (  
  nom_attribut TYPE_ATTRIBUT [OPTIONS]  
  ...  
)
```

- **TEMPORARY** donne pour durée de vie à la table : le temps de la connexion de l'utilisateur au serveur, après, elle sera détruite. En l'absence de cette option, la table sera permanente à moins d'être détruite par la commande **DROP TABLE**.
- L'option **IF NOT EXIST** permet de ne créer cette table que si une table de même nom n'existe pas encore.
- A l'intérieur des parenthèses, il sera listé tous les attributs, clés et indexes de la table.

## Créer une table (II)

- Exemple :

```
CREATE TABLE Personne (  
    nom VARCHAR(40),  
    prenom VARCHAR(40),  
    email VARCHAR(40),  
    laboratoire VARCHAR(30)  
)
```

## Types des attributs

- Les attributs peuvent avoir des **types** très différents :
  - Nombre entier signé ou non (numéro d'ordre, nombre d'objets)
  - Nombre à virgule (prix, température)
  - Date et heure (date de naissance, heure de l'insertion)
  - Énumération (un laboratoire parmi une liste)
  - Ensemble (une ou des compétences parmi une liste)
- Il s'agit de choisir le type le plus adapté aux besoins
- Les types requièrent une plus ou moins grande quantité de données à stocker : il vaut mieux choisir un type varchar pour stocker un nom qu'un type longtext.
- Voir [www.mysql.com](http://www.mysql.com) pour plus de détails

## Types des attributs (II) – entiers

nom	borne inférieure	borne supérieure
<b>TINYINT</b>	-128	127
<b>TINYINT UNSIGNED</b>	0	255
<b>SMALLINT</b>	-32768	32767
<b>SMALLINT UNSIGNED</b>	0	65535
<b>MEDIUMINT</b>	-8388608	8388607
<b>MEDIUMINT UNSIGNED</b>	0	16777215
<b>INT*</b>	-2147483648	2147483647
<b>INT* UNSIGNED</b>	0	4294967295
<b>BIGINT</b>	-9223372036854775808	9223372036854775807
<b>BIGINT UNSIGNED</b>	0	18446744073709551615

(\*) : **INTEGER** est un synonyme de **INT**.

**UNSIGNED** permet d'avoir un type non signé.

## Types des attributs (III) – flottants

- Type **DECIMAL(*precision*,*echelle*)**
  - Représente un nombre codé sur *precision* chiffres, avec *echelle* chiffres après la virgule.
  - *echelle* est optionnel et vaut 0 par défaut.
  - *precision* est optionnel si *echelle* n'est pas indiqué.
    - MySQL : valeur par défaut : 10
- Type **FLOAT(*precision*)**
  - Représente un nombre à virgule flottante.
  - *precision* est optionnel.
    - MySQL : précision en décimal, par défaut : 10
- Les types **INTEGER**, **INT**, **DOUBLE**, . . . sont des raccourcis pour des formes particulières de **DECIMAL** ou **FLOAT**

## Types des attributs (IV) – chaînes

nom	longueur
<b>CHAR(M)</b>	Chaîne de taille fixée à M, où $1 < M < 255$ , complétée avec des espaces si nécessaire.
<b>CHAR(M) BINARY</b>	Idem, mais insensible à la casse lors des tris et recherches.
<b>VARCHAR(M)</b>	Chaîne de taille variable, de taille maximum M, où $1 < M < 255$ , complété avec des espaces si nécessaire.
<b>VARCHAR(M) BINARY</b>	Idem, mais insensible à la casse lors des tris et recherches.
<b>TINYTEXT</b>	Longueur maximale de 255 caractères.
<b>TEXT</b>	Longueur maximale de 65535 caractères.
<b>MEDIUMTEXT</b>	Longueur maximale de 16777215 caractères.
<b>LONGTEXT</b>	Longueur maximale de 4294967295 caractères.

## Types des attributs (V) – chaînes

- Les types **TINYTEXT**, **TEXT**, **MEDIUMTEXT** et **LONGTEXT** peuvent être judicieusement remplacés respectivement par **TINYBLOB**, **BLOB**, **MEDIUMBLOB** et **LONGBLOB**.
- Ils ne diffèrent que par la sensibilité à la casse qui caractérise la famille des BLOB. Alors que la famille des TEXT sont insensibles à la casse lors des tris et recherches.
- Les **BLOB** peuvent être utilisés pour stocker des données binaires.
- Les **VARCHAR**, **TEXT** et **BLOB** sont de taille variable. Alors que les **CHAR** et **DECIMAL** sont de taille fixe.

## Types des attributs (VI) – dates et heures

nom	description
<b>DATE</b>	Date au format anglophone AAAA-MM-JJ.
<b>DATETIME</b>	Date et heure au format anglophone AAAA-MM-JJ HH:MM:SS.
<b>TIMESTAMP</b>	Affiche la date et l'heure sans séparateur : AAAAMMJJHHMMSS.
<b>TIMESTAMP(M)</b>	Idem mais M vaut un entier pair entre 2 et 14. Affiche les M premiers caractères de <b>TIMESTAMP</b> .
<b>TIME</b>	Heure au format HH:MM:SS.
<b>YEAR</b>	Année au format AAAA.

nom	description
<b>TIMESTAMP(2)</b>	AA
<b>TIMESTAMP(4)</b>	AAMM
<b>TIMESTAMP(6)</b>	AAMMJJ
<b>TIMESTAMP(8)</b>	AAAAMMJJ
<b>TIMESTAMP(10)</b>	AAMMJJHHMM
<b>TIMESTAMP(12)</b>	AAMMJJHHMMSS
<b>TIMESTAMP(14)</b>	AAAAMMJJHHMMSS

En cas d'insertion d'un enregistrement en laissant vide un attribut de type **TIMESTAMP**, celui-ci prendra automatiquement la date et heure de l'insertion. Contrairement à Unix (où le timestamp est le nombre de secondes écoulées depuis le 1er janvier 1970), en MySQL, il est une chaîne de format comme indiqué ci-contre.

## Types des attributs (VII) - énumérations

- Un attribut de type **ENUM** peut prendre des valeurs parmi celles définies lors de la création de la table. Ces valeurs sont exclusivement des chaînes de caractères (insensibles à la casse) et peuvent être au maximum au nombre 65535.
  - Nom\_attr **ENUM** ("valeur 1", "valeur 2'...)
  - Nom\_attr **ENUM** ("valeur 1", "valeur 2'...") **NULL**
- A chaque valeur est associée un index allant de 0 pour "" (chaîne nulle) à N si N valeurs ont été définies. L'index **NULL** est associé à la valeur **NULL**
- Si une sélection est faite dans un contexte numérique c'est l'index qui est renvoyé sinon c'est la valeur.

## Types des attributs (VIII) - ensembles

- Un attribut de type **SET** peut prendre pour valeur une chaîne vide, NULL ou une chaîne contenant une liste de valeurs qui doivent être déclarées au moment de la définition de l'attribut.
  - Par exemple, un attribut déclaré comme ci :
    - **SET ("voiture", "avion", "train") NOT NULL**  
Peut prendre les valeurs suivantes :  
""  
"voiture,avion"  
"train"  
Mais pas NULL ni "TGV "
- On ne peut définir que 64 éléments au maximum

## Options des attributs

- **PRIMARY KEY**
  - désigne l'attribut comme la clé primaire de la table
- **NOT NULL**
  - spécifie que l'attribut ne peut avoir de valeurs nulles
- **UNIQUE**
  - spécifie que les valeurs de cet attribut doivent être distinctes pour chaque couple de tuples.
- **AUTO\_INCREMENT**
  - Incrémentation automatique pour cet attribut si l'utilisateur ne spécifie pas de valeur pour cet attribut à l'insertion du tuple.
- **DEFAULT**
  - spécifie la valeur utilisée par défaut si l'utilisateur ne spécifie pas de valeur pour cet attribut à l'insertion du tuple.
- **REFERENCES**
  - spécifie que l'attribut correspond à la clé primaire d'une autre relation.

## Clé primaire (I)

- Dans une base de données relationnelle, **une clé primaire est une contrainte d'unicité** qui permet d'identifier de manière unique un enregistrement dans une table. Une clé primaire **peut être composée d'un ou de plusieurs attributs** de la table.
- Cet attribut devra ne jamais être vide, il faut donc préciser l'option **NOT NULL** pour le forcer à prendre une valeur de son domaine.
- Il devra aussi être unique, c'est-à-dire que deux enregistrements ne pourront pas avoir une valeur identique. Il faut alors faire la déclaration suivante : **UNIQUE (cle)** à la suite de la liste des attributs.
- Pour simplifier, on utilisera l'option **PRIMARY KEY** qui regroupe **NOT NULL** et **UNIQUE** en remplacement des deux dernières déclarations.
- Et pour finir, il faut signifier que cette valeur doit s'incrémenter automatiquement à chaque insertion d'un enregistrement grâce à l'option **AUTO\_INCREMENT**.

## Clé primaire (II)

- Dans notre exemple, la liste des inscrits ne devrait pas excéder plusieurs centaines de personnes. Ainsi un attribut de type **SMALLINT UNSIGNED** devrait faire l'affaire. Nous le nommerons : *id*
- Notre exemple devient :  

```
CREATE TABLE Personne (  
  id SMALLINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
  nom VARCHAR(40),  
  prenom VARCHAR(40),  
  adresse VARCHAR(40),  
  laboratoire VARCHAR(30)  
)
```
- La numérotation des clés primaires, **débute à 1 et pas à 0.**

## Clé primaire (III)

- Notre clé primaire **peut être associée simultanément à plusieurs attributs** mais selon une syntaxe différente.
- Si au lieu de créer un identifiant numérique unique, on souhaite simplement interdire d'avoir des doublon sur le couple (*nom,prenom*) et d'en interdire la nullité, on va créer une clé primaire sur ce couple.
- La connaissance des seuls nom et prénom suffit à identifier sans ambiguïté un et un seul enregistrement.

Bonne syntaxe :

Mauvaise syntaxe :

```
CREATE TABLE Personne (  
  nom VARCHAR(40) PRIMARY KEY,  
  prenom VARCHAR(40) PRIMARY KEY,  
  email VARCHAR(40) ,  
  laboratoire VARCHAR(30)  
)
```

```
CREATE TABLE Personne (  
  nom VARCHAR(40),  
  prenom VARCHAR(40),  
  email VARCHAR(40) ,  
  laboratoire VARCHAR(30) ,  
  CONSTRAINT pk_nomprenom  
  PRIMARY KEY (nom,prenom)  
)
```

## Attribut non nul

- Considérons que l'on souhaite que certains attributs aient obligatoirement une valeur. On utilisera l'option **NOT NULL**.
- Dans ce cas, si malgré tout, aucune valeur n'est fournie, la valeur par défaut (si elle est déclarée à la création de la relation ) sera automatiquement affectée à cet attribut dans l'enregistrement.
- Si aucune valeur par défaut n'est déclarée :
  - la chaîne vide "" sera affectée à l'attribut s'il est de type chaîne de caractères
  - la valeur zéro 0 s'il est de type nombre
  - la date nulle 0000-00-00 et/ou l'heure nulle 00:00:00 s'il est de type date, heure ou date et heure.
- Au contraire, on utilisera l'option **NULL** si on autorise l'absence de valeur.



## Valeur par défaut


- Pour donner une valeur par défaut à un attribut, on utilise l'option **DEFAULT**.
- Lors de l'ajout d'un enregistrement cette valeur sera affectée à l'attribut si aucune valeur n'est donnée.
- Exemple :  
*telephone* **DECIMAL(10,0) DEFAULT '0123456789'**
- Les attributs de type chaîne de caractères de la famille **TEXT** et **BLOB** ne peuvent pas avoir de valeur par défaut.

## Attribut sans doublon (I)

- Pour interdire l'apparition de doublon pour un attribut, on utilise l'option **UNIQUE**.
- Syntaxe :  
**UNIQUE** [*nom dela contrainte*](*liste des attributs*)
- Pour interdire les doublons sur l'attribut '*nom*' mais les interdire aussi sur '*prénom*', tout en les laissant indépendants :  
**UNIQUE('nom')**  
**UNIQUE('prénom')**

<i>nom</i>	<i>prénom</i>
Dupond	<b>Marc</b>
Ducret	Pierre
Martin	<b>Marc</b>

enregistrement interdit  
car 'Marc' est un doublon  
dans la colonne 'prénom'



## Attribut sans doublon (II)

- Pour interdire tout doublon à un ensemble d'attributs (tuple), on passe en paramètre à **UNIQUE** la liste des attributs concernés.
- Pour interdire tout doublon du couple ('nom', 'prénom') :  
**UNIQUE('nom', 'prénom')**

nom	prénom
Dupond	Marc
Dupont	Pierre
<b>Martin</b>	<b>Marc</b>
Martin	Pierre
<b>Martin</b>	<b>Marc</b>

enregistrement interdit car le couple ('Martin', 'Marc') est un doublon du couple ('nom', 'prénom')

## Index (I)

- Un index est un objet complémentaire (mais non indispensable) à la base de données permettant d'indexer certaines colonnes dans le but d'améliorer l'accès aux données par le SGBDR
  - Au même titre qu'un index dans un livre ne vous est pas indispensable mais vous permet souvent d'économiser du temps lorsque vous recherchez une partie spécifique de ce dernier...
- Un index permet au moteur d'accéder rapidement à la donnée recherchée.
  - Si vous recherchez un champs ayant une valeur donnée et qu'il n'y a pas d'index sur ce champ, le moteur devra parcourir toute la table, et à chaque fois, faire les comparaisons nécessaires pour extraire un résultat pertinent.

## Index (II)

- Les indexes doivent toutefois être utilisés avec parcimonie car ils pénalisent les temps d'insertion et de suppression des données dans la table.
  - La création d'index utilise de l'espace mémoire dans la base de données, et, un index est mis à jour à chaque modification de la table à laquelle il est rattaché,
- Une clé primaire est par définition un index unique sur un champ non null.
- Un index peut éventuellement être null.

## Création / Suppression d'un index

- Pour créer un index en utilise la syntaxe suivante:  
**CREATE [UNIQUE] INDEX index\_name ON tbl\_name (col\_name [(length)] [ASC | DESC],...)**
  - La taille peut être précisée pour des champs varchar par exemple.
  - Elle doit l'être obligatoirement pour les types blob.
- Pour supprimer un index en utilise la syntaxe suivante:  
**DROP INDEX index\_name ON tbl\_name**

## Optimisation

- Après la suppression de grandes parties d'une table contenant des index, les index des tuples supprimés sont conservés, rallongeant d'autant les sélections. Pour supprimer ces index obsolètes et vider les « trous », il faut l'optimiser.
- Syntaxe :  
**OPTIMIZE TABLE *Relation***
- Exemple :  
**OPTIMIZE TABLE *Personnes***

## Supprimer une relation

- La commande **DROP TABLE** prend en paramètre le nom de la table à supprimer. Toutes les données qu'elle contient sont supprimées et sa définition aussi.
- Syntaxe :  
**DROP TABLE *relation***
- Exemple :  
**DROP TABLE *Personnes***
- Si un beau jour on s'aperçoit qu'une relation a été mal définie au départ, plutôt que de la supprimer et de la reconstruire bien comme il faut, on peut la modifier très simplement. Cela évite de perdre les données qu'elle contient.

## Modifier une relation

- La création d'une relation par **CREATE TABLE** n'en rend pas définitives les spécifications. Il est possible d'en modifier la définition par la suite, à tout moment par la commande **ALTER TABLE**.
- Voici ce qu'il est possible de réaliser :
  - ❑ ajouter/supprimer un attribut
  - ❑ créer/supprimer une clé primaire
  - ❑ ajouter une contrainte d'unicité (interdire les doublons)
  - ❑ changer la valeur par défaut d'un attribut
  - ❑ changer totalement la définition d'un attribut
  - ❑ changer le nom de la relation
  - ❑ ajouter/supprimer un index

## Ajouter un attribut

- Syntaxe :  
**ALTER TABLE *relation* ADD *definition* [ FIRST | AFTER *attribut* ]**
- Ajoutons l'attribut *fax* qui est une chaîne représentant un nombre de 10 chiffres:  
**ALTER TABLE *Personnes* ADD *fax* DECIMAL(10,0)**
- Nous aurions pu forcer la place où doit apparaître cet attribut. Pour le mettre en tête de la liste des attributs de la relation, il faut ajouter l'option **FIRST** en fin de commande. Pour le mettre après l'attribut '*téléphone*', il aurait fallu ajouter **AFTER '*telephone*'**.
- **Note : il ne doit pas déjà avoir dans la relation un attribut du même nom !**

## Supprimer un attribut (I)

- Attention, supprimer un attribut implique la suppression des valeurs qui se trouvent dans la colonne qui correspond à cet attribut.

- Syntaxe :

**ALTER TABLE** *relation* **DROP** *attribut*

- Exemple :

**ALTER TABLE** *Personnes* **DROP** 'prénom'

## Supprimer un attribut (II)

- La suppression d'un attribut peut incidemment provoquer des erreurs sur les contraintes clé primaire (**PRIMARY KEY**) et unique (**UNIQUE**).

```
CREATE TABLE Personne (  
  id SMALLINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
  nom VARCHAR(40), prenom VARCHAR(40),  
  adresse TINYTEXT, telephone DECIMAL(10,0), UNIQUE(nom,prenom)  
)
```

**ALTER TABLE** *Personnes* **DROP** 'prénom'

<i>nom</i>	<i>prénom</i>
Dupond	Marc
Martin	Marc
Martin	Pierre

<i>nom</i>
Dupond
<b>Martin</b>
<b>Martin</b>

Refus d'opérer la suppression,  
car cela contredirait la  
contrainte d'unicité qui  
resterait sur l'attribut *nom*.

## Créer une clé primaire

- La création d'une clé primaire n'est possible qu'en l'absence de clé primaire dans la relation.
- Syntaxe :  
**ALTER TABLE *relation* ADD PRIMARY KEY (*attribut*)**
- Exemple :  
**ALTER TABLE *Personnes* ADD PRIMARY KEY (*nom,prenom*)**

## Supprimer une clé primaire

- Comme une clé primaire est unique, il n'y a aucune ambiguïté lors de la suppression.
- Syntaxe :  
**ALTER TABLE *relation* DROP PRIMARY KEY**
- Exemple :  
**ALTER TABLE *Personnes* DROP PRIMARY KEY**
- S'il n'y a aucune clé primaire lorsque cette commande est exécutée, aucun message d'erreur ne sera généré, la commande sera simplement ignorée.

## Ajout d'une contrainte d'unicité

- Cette contrainte peut s'appliquer à plusieurs attributs.
- Il est possible (facultatif) de donner un nom à la contrainte.
- Si les valeurs déjà présentes dans la relation sont en contradiction avec cette nouvelle contrainte, alors cette dernière ne sera pas appliquée et une erreur sera générée.

- Syntaxe :

**ALTER TABLE *relation* ADD UNIQUE [*contrainte*] (*attributs*)**

- Exemple pour interdire tout doublon sur l'attribut *fax* de la relation ***Personnes*** :

**ALTER TABLE *Personnes* ADD UNIQUE *u\_fax* (*fax*)**

- Autre exemple fictif :

**ALTER TABLE *Moto* ADD UNIQUE *u\_coul\_vitre* (*couleur,vitre*)**

## Changer la valeur par défaut d'un attribut

- Pour changer ou supprimer la valeur par défaut d'un attribut.
- Attention aux types qui n'acceptent pas de valeur par défaut (les familles **BLOB** et **TEXT**).

- Syntaxe :

**ALTER TABLE *relation* ALTER *attribut* { SET DEFAULT valeur | DROP DEFAULT }**

- Changer sa valeur par défaut :

**ALTER TABLE *Personnes* ALTER 'telephone' SET DEFAULT '999999999'**

- Supprimer sa valeur par défaut :

**ALTER TABLE *Personnes* ALTER 'telephone' DROP DEFAULT**

- Le changement ou la suppression n'affecte en rien les enregistrements qui ont eu recours à cette valeur lors de leur insertion.



## Changer la définition d'un attribut

- Pour changer la définition de l'attribut sans le renommer :  
**ALTER TABLE *relation* MODIFY *attribut* *definition\_relative***
  - **Exemple 1 :**  
**ALTER TABLE *Personnes* MODIFY *fax* VARCHAR(14)**
- Pour changer sa définition en le renommant :  
**ALTER TABLE *relation* CHANGE *attribut* *definition\_absolue***
  - **Exemple 2 :**  
**ALTER TABLE *Personnes* CHANGE *fax* *num\_fax* VARCHAR(14)**
- **Attention:** si le nouveau type appliqué à l'attribut est incompatible avec les valeurs des enregistrements déjà présents dans la relation, alors elles risquent d'être modifiées ou remises à zéro !

## Changer le nom de la relation

- Syntaxe :  
**ALTER TABLE *relation* RENAME *nouveau\_nom***
- Exemple :  
**ALTER TABLE *Personnes* RENAME *Carnet***
- Cela consiste à renommer la table, et donc le fichier qui la stocke.

## SQL:

### Langage de Définition de Données

### Langage de Manipulation de Données

- ❖ Ajouter les données (INSERT)
- ❖ Modifier les données (UPDATE)
- ❖ Supprimer les données (DELETE)
- ❖ Consulter les données (SELECT)

## Ajouter un enregistrement

- Pour ajouter un enregistrement à une relation, il faudra préciser la valeur pour chacun des attributs.
  - Si certaines valeurs sont omises, alors les valeurs par défaut définies lors de la création de la relation seront utilisées. Si on ne dispose pas non plus de ces valeurs par défaut, alors MySQL mettra 0 pour un nombre, "" pour une chaîne, 0000-00-00 pour une date, 00:00:00 pour une heure, 0000000000000000 pour un timestamp.
- **Syntaxe :**  
**INSERT INTO** *relation*(*liste des attributs*) **VALUES**(*liste des valeurs*)
- **Exemple :**  
**INSERT INTO** *Personnes*(*nom, prenom*) **VALUES**('Martin','Jean')
- **REPLACE** est un synonyme de **INSERT**, mais sans doublon. Pratique pour respecter les contraintes d'unicité (**UNIQUE**, **PRIMARY KEY**).

## Modifier un enregistrement (I)

- Pour modifier un ou des enregistrement(s) d'une relation, il faut préciser un critère de sélection des enregistrement à modifier (clause **WHERE**), il faut aussi dire quels sont les attributs dont on va modifier la valeur et quelles sont ces nouvelles valeurs (clause **SET**).
- Syntaxe :  
**UPDATE** [ **LOW\_PRIORITY** ] *relation* **SET** *attribut=valeur*, ... [ **WHERE condition** ] [ **LIMIT n** ]
- Exemple : modifier le numéro de téléphone de Martin Pierre:  
**UPDATE TABLE Personnes SET telephone='0156281469' WHERE nom='Martin' AND prenom = 'Pierre'**
- **LOW\_PRIORITY** est une option un peu spéciale qui permet de n'appliquer la ou les modification(s) qu'une fois que plus personne n'est en train de lire dans la relation.

## Modifier un enregistrement (II)

- Il est possible de modifier les valeurs d'autant d'attributs que la relation en contient.
  - Exemple pour modifier plusieurs attributs :  
**UPDATE TABLE Personnes SET telephone='0156281469', fax='0156281812' WHERE id = 102**
- Pour appliquer la modification à tous les enregistrements de la relation, il suffit de ne pas mettre de clause **WHERE**.
- **LIMIT n** permet de n'appliquer la commande qu'aux **n** premiers enregistrements satisfaisant la condition définie par **WHERE**.
- Il est possible de modifier la valeur d'un attribut relativement à sa valeur déjà existante.
  - Exemple : **UPDATE TABLE Enfants SET age=age+1**

## Supprimer un enregistrement

- Attention, la suppression est définitive !
- Syntaxe :  
**DELETE [LOW\_PRIORITY] FROM *relation* [WHERE *condition*] [LIMIT *a*]**
  - Exemple :  
**DELETE FROM *Personnes* WHERE *nom*='Martin' AND *prenom*='Marc'**
- Pour vider une table de tous ces éléments, ne pas mettre de clause WHERE. Cela efface et recrée la table, au lieu de supprimer un à un chacun des tuples de la table (ce qui serait très long).
  - Exemple :  
**DELETE FROM *Personnes***

## Contraintes d'intégrité

- Il s'agit d'une méthode déclarative pour assurer le respect des règles de gestion dans la base de données.
- Les règles de gestion sont définies avec la table correspondante grâce aux commandes **CREATE TABLE** ou **ALTER TABLE** en utilisant la clause **CONSTRAINT**.
- Ces règles sont validées quand les contraintes sont définies, et à chaque instruction **INSERT**, **UPDATE** ou **DELETE**.

## Exemples: clé primaire et unicité

- Exemple 1:

```
CREATE TABLE personnes (  
    id INT NOT NULL PRIMARY KEY,  
    nom VARCHAR(50),  
    prenom VARCHAR(50),  
    avs INT UNIQUE,  
    language VARCHAR(50),  
    CONSTRAINT u_nom_prenom UNIQUE (nom, prenom)  
)
```

- Exemple 2:

```
ALTER TABLE etudiant  
ADD CONSTRAINT etudiant_pk  
PRIMARY KEY (id_etudiant);
```

## Contraintes de domaine

- Une contrainte qui permet de valider la valeur de la colonne.
- A chaque fois que l'on va ajouter un élément dans cette colonne, cette condition va se vérifier et l'enregistrement ne se fera qu'en cas de passage de la validation.
  - Néanmoins, ces validations peuvent se révéler lourde, il ne faut donc pas en abuser.

## Contraintes de domaine

- Exemple 1:

```
CREATE TABLE tests (  
    id INT NOT NULL PRIMARY KEY,  
    nom VARCHAR(20),  
    note INT CHECK (VALUE BETWEEN 1 AND 6)  
)
```

- Exemple 2:

```
ALTER TABLE etudiant ADD  
CONSTRAINT sexe_ck  
CHECK (sexe_etudiant IN ('F', 'M'))
```

## Contraintes d'intégrité référentielle

### Rappel :

- Dans un modèle physique de données, les tables sont liées entre elles par le biais d'une clé étrangère.
- La clé étrangère d'une table permet de lier la table à la clé primaire de l'autre table.
- Lors de la sélection des données, l'association entre les deux tables est effectuée grâce à une jointure entre ces deux tables.

## Contraintes d'intégrité référentielle

Exemple :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

Ici, chaque référence  
à une adresse est respectée.

## Contraintes d'intégrité référentielle

Exemple :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

Si on supprime  
de la table adresse la ligne 4,  
il devient impossible de retrouver l'adresse de Marie Durand.

## Contraintes d'intégrité référentielle

- Comment garantir qu'une clé trouve toujours la donnée référencée dans la table ?
  - En empêchant la suppression de la table si une référence vers cette donnée existe.
  - En supprimant la référence lors de la suppression de la donnée référencée.
  - En supprimant automatiquement les données référencées.

## Contraintes d'intégrité référentielle

- L'**intégrité référentielle** permet de garantir que toute clé étrangère correspond bien à une clé primaire à laquelle elle fait référence.
  - C'est un contrôle ajouté à une clé étrangère.
- Lors de la création des tables, il est possible de confier ce contrôle au moteur de la base de données.
- **Note :** Sur MySQL, cette fonctionnalité n'est disponible qu'avec le moteur des tables : InnoDB.



## Contraintes d'intégrité référentielle

- Pour déclarer une référence il faut ajouter une clause lors d'un « **CREATE TABLE** » ou avec un « **ALTER TABLE** »:

```
CONSTRAINT nom_contrainte  
FOREIGN KEY (champ1, [..., champN])  
REFERENCES table(champ1, [..., champN])  
[ON UPDATE action]  
[ON DELETE action]
```

## Contraintes d'intégrité référentielle

- Les actions possibles sont :
  - ❑ **RESTRICT** : si une référence est trouvée, la suppression ou la modification sera interdite.
  - ❑ **SET NULL** : si une référence est trouvée, la suppression ou la modification aura pour effet en plus de l'action, de mettre à jour la référence avec la valeur NULL.
  - ❑ **CASCADE** : si une référence est trouvée, la suppression ou la modification d'un élément aura pour effet d'effectuer la même opération sur les éléments qui le référence.
  - ❑ **NO ACTION** : pas de contrôle d'intégrité référentielle.

## Contraintes d'intégrité référentielle

Exemple avec **ON UPDATE RESTRICT** :

Personne			
Id	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
Id	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

**Ne fonctionnent pas :**

Update **adresse** set id = 8 where id =3

Update **personne** set id\_adresse = 5 where id = 1

Insert **personne** values (5, 'Yu','Van',5)

**Fonctionnent :**

Update **personne** set id\_adresse=1 where id=3

Update **personne** set id\_adresse=null where id=1

## Contraintes d'intégrité référentielle

Exemple avec **ON UPDATE CASCADE** :

Personne			
Id	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	5
3	Dubois	Jean	NULL
4	Simpson	Lisa	5

Adresse			
Id	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
5	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

Action :

update **adresse** set id=5 where id=3

## Contraintes d'intégrité référentielle

Exemple avec **ON UPDATE CASCADE** :

Personne			
Id	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
Id	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

**Ne fonctionnent pas :**

Insert personne values (5, 'Yu','Van',5)

Update personne set id\_adresse = 5 where id = 1

**Fonctionnent :**

Update personne set id\_adresse=1 where id=3

Update adresse set id = 8 where id=3

Update personne set id\_adresse=null where id=1

## Contraintes d'intégrité référentielle

Exemple avec **ON UPDATE SET NULL** :

Personne			
Id	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	NULL
3	Dubois	Jean	NULL
4	Simpson	Lisa	NULL

Adresse			
Id	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
5	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

Action :

update **adresse** set id=5 where id=3

## Contraintes d'intégrité référentielle

Exemple avec **ON UPDATE SET NULL** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

**Ne fonctionnent pas :**

Insert personne values (5, 'Yu','Van',5)

Update personne set id\_adresse = 5 where id = 1

**Fonctionnent :**

Update personne set id\_adresse=1 where id=3

Update adresse set id = 8 where id =3

Update personne set id\_adresse=null where id=1

## Contraintes d'intégrité référentielle

Exemple avec **ON UPDATE NO ACTION** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

**Fonctionnent :**

Insert personne values (5, 'Yu','Van',5)

Update personne set id\_adresse = 5 where id = 1

Update personne set id\_adresse=1 where id=3

Update adresse set id = 8 where id =3

Update personne set id\_adresse=null where id=1

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE RESTRICT** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

**Ne fonctionnent pas :**

Delete **adresse** where id =3

**Fonctionnent :**

Delete **personne** where id = 1

Delete **adresse** where id = 2

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE CASCADE** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

Action :

Delete adresse where id=3

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE CASCADE** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
3	Dubois	Jean	NULL

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
4	Chemin perdu	66000	Perpignan

Action :

Delete adresse where id=3

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE CASCADE** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

**Fonctionnent :**

Delete **personne** where id=3

Delete **adresse** where id =3

Delete **adresse** where id =2

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE SET NULL** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

Action :

Delete **adresse** where id=3

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE SET NULL** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	NULL
3	Dubois	Jean	NULL
4	Simpson	Lisa	NULL

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
4	Chemin perdu	66000	Perpignan

Action :

Delete **adresse** where id=3

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE SET NULL** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

### Fonctionnent :

Delete personne where id=3

Delete adresse where id =3

Delete adresse where id =2

## Contraintes d'intégrité référentielle

Exemple avec **ON DELETE NO ACTION** :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

### Fonctionnent :

Delete personne where id=3

Delete adresse where id =3

Delete adresse where id =2



## Supprimer des contraintes

- Syntaxe :

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

- Exemple :

```
ALTER TABLE etudiant DROP CONSTRAINT sexe_ck;
```

- Une façon plus simple de supprimer des contraintes est d'utiliser la syntaxe suivante, où l'option CASCADE supprime toutes les contraintes de clé étrangère.

```
ALTER TABLE nom_table DROP PRIMARY KEY CASCADE;
```

## Activation/Désactivation de contraintes

- Une contrainte peut être temporairement désactivée puis réactivée ultérieurement en utilisant la syntaxe

```
ALTER TABLE nom_table DISABLE  
CONSTRAINT nom_contrainte
```

```
ALTER TABLE nom_table ENABLE  
CONSTRAINT nom_contrainte
```

## SQL:

### Langage de Définition de Données

### Langage de manipulation de données

- ❖ Ajouter les données (INSERT)
- ❖ Modifier les données (UPDATE)
- ❖ Supprimer les données (DELETE)
- ❖ **Consulter les données (SELECT)**

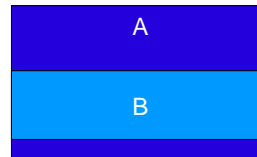
## Algèbre relationnelle

- C'est l'ensemble des opérations possible sur les relations. On distingue :
  - ❑ La sélection
  - ❑ La projection
  - ❑ L'intersection
  - ❑ L'union
  - ❑ La différence
  - ❑ Le produit cartésien
  - ❑ La jointure
  - ❑ La division cartésienne
- Cette algèbre est facilement possible avec la syntaxe SQL :

**SELECT** attributs **FROM** relations **WHERE** criteres

## La sélection

A partir d'un ensemble A, obtenir les données B correspondant à des critères donnés.

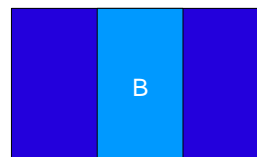


Opérateur unaire.

ID	Nom	Prénom
1	Simpson	Marge
2	Simpson	Homer
3	Simpson	Bart
4	Simpson	Lisa

## La projection

A partir d'un ensemble A, obtenir uniquement les données pertinentes B.

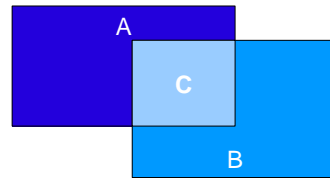


Opérateur unaire.

ID	Nom	Prénom
1	Simpson	Marge
2	Simpson	Homer
3	Simpson	Bart
4	Simpson	Lisa

## L'intersection

A partir des ensembles A et B, obtenir les données C qui existent à la fois dans A et Dans B.



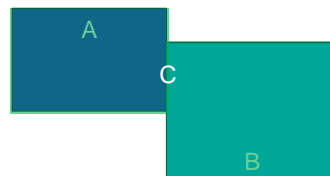
Opérateur binaire.

ID	Prénom
1	Bart
2	Lisa
3	Marge

ID	Prénom
5	Marie
6	Lisa
7	Jacques

## L'union

A partir des ensembles A et B, obtenir les données C qui existent soit dans A soit dans B.



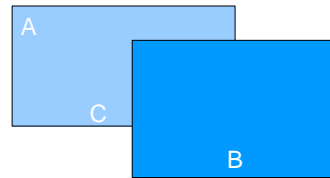
Opérateur binaire.

ID	Prénom
1	Bart
2	Lisa
3	Marge

ID	Prénom
5	Marie
6	Lisa
7	Jacques

## La différence

A partir des ensembles A et B, obtenir les données C qui existent dans A mais pas dans B.



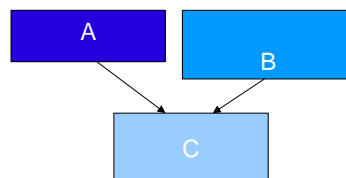
Opérateur binaire.

ID	Prénom
1	Marge
2	Lisa
3	Homer

ID	Prénom
5	Marie
6	Lisa
7	Jacques

## Le produit cartésien

A partir des ensembles A et B, obtenir toutes les combinaisons possibles C.



Opérateur N-aire.

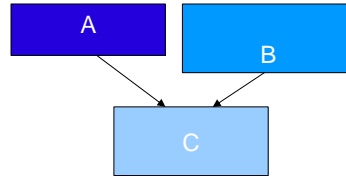
ID	Prénom
1	Bart
2	Lisa

ID	Nom
5	Simpson

Prénom	Nom
Bart	Simpson
Lisa	Simpson

## La jointure

A partir des ensembles A et B, obtenir le produit cartésien C **limité à une valeur commune**.



Opérateur N-aire.

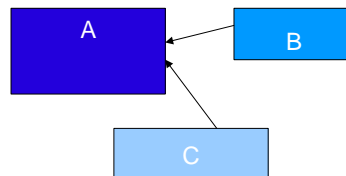
ID	Prénom	ID	Nom
1	Marge	1	Simpson
2	Homer		

Prénom	Nom
Marge	Simpson

## Le division cartésienne

A partir des ensembles A et B, obtenir les données C qui combinées à l'ensemble B donnent l'ensemble A.



Opérateur N-aire.

Prénom	Nom	Nom
Lisa	Simpson	Simpson
Bart	Simpson	

Prénom
Lisa
Bart

## Sélectionner des enregistrements (I)

- Pour extraire de votre base de données des informations, comme la liste des personnes de votre carnet d'adresse qui vivent à Paris.

- Syntaxe générale :

```
SELECT [ DISTINCT ] attributs  
      [ INTO OUTFILE fichier ]  
      [ FROM relation ]  
      [ WHERE condition ]  
      [ GROUP BY attributs [ ASC | DESC ] ]  
      [ HAVING condition ]  
      [ ORDER BY attributs [ ASC | DESC ] ]  
      [ LIMIT [ a, ] b ]  
      ...
```

## Sélectionner des enregistrements (II)

Nom	Description
<b>SELECT</b>	Spécifie les attributs dont on souhaite connaître les valeurs.
<b>DISTINCT</b>	Permet d'ignorer les doublons de ligne de résultat.
<b>INTO OUTFILE</b>	Spécifie le fichier sur lequel effectuer la sélection.
<b>FROM</b>	Spécifie le ou les relations sur lesquelles effectuer la sélection.
<b>WHERE</b>	Définit le ou les critères de sélection sur des attributs.
<b>GROUP BY</b>	Permet de grouper les lignes de résultats selon un ou des attributs.
<b>HAVING</b>	Définit un ou des critères de sélection sur des ensembles de valeurs d'attributs après groupement.
<b>ORDER BY</b>	Permet de définir l'ordre ( <b>ASC</b> endant par défaut ou <b>DESC</b> endant) dans l'envoi des résultats.
<b>LIMIT</b>	Permet de limiter le nombre de lignes du résultats

## Exemple(1)

**Personnes**

Nom	Prénom	E-Mail	Laboratoire
Charnay	Daniel	charnay@in2p3.fr	Centre de Calcul
Macchi	Pierre-Etienne	macchi@in2p3.fr	Centre de Calcul
Boutherin	Bernard	Boutherin@isn.in2p3.fr	ISN

On ne veut sélectionner que les tuples dont l'attribut laboratoire est égal à ISN

```
SELECT * FROM Personnes WHERE Laboratoire="ISN"
```

Nom	Prénom	E-Mail	Laboratoire
Boutherin	Bernard	Boutherin@isn.in2p3.fr	ISN

## Exemple(2)

*Personnes*

Nom	Prenom	E-Mail	Laboratoire
Charnay	Daniel	charnay@in2p3.fr	Centre de Calcul
Macchi	Pierre-Etienne	macchi@in2p3.fr	Centre de Calcul

On projette la table *Personnes* sur les colonnes nom, prenom

```
SELECT nom,prenom FROM Personnes
```

Nom	Prenom
Charnay	Daniel
Macchi	Pierre-Etienne



## Sélectionner des enregistrements (III)

### ■ Procédons par étapes :

1. Pour sélectionner tous les enregistrements d'une relation :  
**SELECT \* FROM relation**
2. Pour sélectionner toutes les valeurs d'un seul attribut :  
**SELECT attribut FROM relation**
3. Pour éliminer les doublons :  
**SELECT DISTINCT attribut FROM relation**
4. Pour trier les valeurs en ordre croissant :  
**SELECT DISTINCT attribut FROM relation ORDER BY attribut ASC**
5. Pour se limiter aux **n** premiers résultats :  
**SELECT DISTINCT attribut FROM relation ORDER BY attribut ASC LIMIT n**
6. Pour ne sélectionner que ceux qui satisfont à une condition :  
**SELECT DISTINCT attribut FROM relation WHERE condition ORDER BY attribut ASC LIMIT n**

## Sélectionner des enregistrements (IV)

**SELECT \* FROM Gens**

1

<b>Gens</b>		
<i>Nom</i>	<i>Prenom</i>	<i>Age</i>
Dupond	Pierre	24
Martin	Marc	48
Dupont	Jean	51
Martin	Paul	36
Dupond	Lionel	68
Chirac	Jacques	70

**SELECT Nom FROM Gens**

2

<b>Gens</b>
<i>Nom</i>
Dupond
Martin
Dupont
Martin
Dupond
Chirac

**SELECT DISTINCT Nom FROM Gens**

3

<b>Gens</b>
<i>Nom</i>
Dupond
Martin
Dupont
Chirac

## Sélectionner des enregistrements (V)

4

<i>Gens</i>
<i>Nom</i>
Chirac
Dupond
Dupont
Martin

- `SELECT DISTINCT Nom FROM Gens ORDER BY Nom ASC`

5

<i>Gens</i>
<i>Nom</i>
Chirac
Dupond

- `SELECT DISTINCT Nom FROM Gens ORDER BY Nom ASC LIMIT 2`

6

<i>Gens</i>
<i>Nom</i>
Dupond
Dupont

- `SELECT DISTINCT Nom FROM Gens WHERE Nom <> 'Chirac' ORDER BY Nom ASC LIMIT 2`

## Alias (I)

- Pendant la sélection on peut renommer les attributs résultants en utilisant des alias:

`SELECT exp1 [AS alias1], exp2 [AS alias2], ... FROM table1`

- `AS` permet de renommer l'expression. C'est un alias qui est alors utilisé comme nom de colonne pour la table résultat.
- La syntaxe d'un alias suit les mêmes règles qu'un nom de base, de table, de colonne ou d'index.

- Exemple:

```
SELECT nom as cadre
FROM Personnes
WHERE salaire >=3000
```

## Alias (II)

- On peut aussi utiliser les alias sur les tables.
  - Permet d'attribuer un autre nom à une table dans une requête SQL.
- Cela peut aider à avoir des noms plus court, plus simple et plus facilement compréhensible. Ceci est particulièrement vrai lorsqu'il y a des jointures.
- Exemple:  

```
Select * from Personne as P, Adresse as A  
where P.Id_adresse = A.Id  
and P.Nom = 'Durand'
```

## Les fonctions MySQL

- Ces fonctions sont à ajouter à vos requêtes dans un **SELECT**, **WHERE**, **GROUP BY** ou encore **HAVING**.
- D'abord sachez que vous avez à votre disposition :
  - les parenthèses **()**,
  - les opérateurs arithmétiques **(+, -, \*, /, %)**,
  - les opérateurs logiques qui retournent **0** (faux) ou **1** (vrai) (**AND**, **OR**, **NOT**, **BETWEEN**, **IN**),
  - les opérateurs relationnels **(<, <=, =, >, >=)**.
- Les opérateurs et les fonctions peuvent être composés entre eux pour donner des expressions très complexes.

## Quelques exemples

- Liste du nom des produits dont le prix est inférieur ou égale à 105 DHs.  
**SELECT nom FROM produits WHERE prix <= 105**
- Liste des nom et prénom des élèves dont l'âge est compris entre 12 et 16 ans.  
**SELECT nom,prenom FROM eleves WHERE age BETWEEN 12 AND 16**
- Liste des articles dont le prix unitaire est > 150 et dont la quantité est < 100  
**SELECT \* FROM Article WHERE (Prix > 150) AND (Qté<100)**
- Liste des noms et adresses des clients de Rabat ou Salé :  
**SELECT NomCl, AdrCl FROM Client WHERE (Ville="Rabat") OR (Ville="Salé")**
- Liste des modèles de voiture dont la couleur est dans la liste : rouge, blanc, noir.  
**SELECT modèle FROM voitures WHERE couleur IN ('rouge', 'blanc', 'noir')**
- Liste des modèles de voiture dont la couleur n'est pas dans la liste : rose, violet.  
**SELECT modèle FROM voitures WHERE couleur NOT IN ('rose', 'violet')**

## Fonctions de comparaison de chaînes

- Le mot clé **LIKE** permet de comparer deux chaînes.
  - Le caractère '%' est spécial et signifie : 0 ou plusieurs caractères.
  - Le caractère '\_' est spécial et signifie : 1 seul caractère, n'importe lequel.
- L'exemple suivant permet de rechercher tous les clients dont le prénom commence par 'Jean', cela peut être 'Jean-Pierre', etc... :  
**SELECT nom FROM clients WHERE prénom LIKE 'Jean%'**
- Pour utiliser les caractères spéciaux ci-haut en leur enlevant leur fonction spéciale, il faut les faire précéder de l'antislash : '\'.  
**SELECT \* FROM produit WHERE code LIKE '\\_XE%'**

## Fonctions mathématiques

Fonction	Description
ABS(x)	Valeur absolue de X.
SIGN(x)	Signe de X, retourne -1, 0 ou 1.
FLOOR(x)	Arrondi à l'entier inférieur.
CEILING(x)	Arrondi à l'entier supérieur.
ROUND(x)	Arrondi à l'entier le plus proche.
EXP(x), LOG(x), SIN(x), COS(x), TAN(x), PI()	Bon, là c'est les fonctions de maths de base...
POW(x,y)	Retourne X à la puissance Y.
RAND(), RAND(x)	Retourne un nombre aléatoire entre 0 et 1.0 Si x est spécifié, entre 0 et X
TRUNCATE(x,y)	Tronque le nombre X à la Yème décimale.

```
SELECT nom
FROM filiales
WHERE SIGN(ca) = -1
```

Cet exemple affiche dans un ordre aléatoire le nom des filiales dont le chiffre d'affaire est négatif.  
A noter que :  $SIGN(ca) = -1 \Leftrightarrow ca < 0$

## Fonctions de chaînes

Fonction	Description
TRIM(x)	Supprime les espaces de début et de fin de chaîne.
LOWER(x)	Converti en minuscules.
UPPER(x)	Converti en majuscules.
LONGUEUR(x)	Retourne la taille de la chaîne.
LOCATE(x,y)	Retourne la position de la dernière occurrence de x dans y. Retourne 0 si x n'est pas trouvé dans y.
CONCAT(x,y,...)	Concatène ses arguments.
SUBSTRING(s,i,n)	Retourne les n derniers caractères de s en commençant à partir de la position i.
SOUNDEX(x)	Retourne une représentation phonétique de x.

```
SELECT UPPER(nom)
FROM clients
WHERE SOUNDEX(nom) = SOUNDEX('Dupond')
```

On affiche en majuscules le nom de tous les clients dont le nom ressemble à 'Dupond'.

## Fonctions de dates et heures

Fonction	Description
NOW()	Retourne la date et heure du jour.
TO_DAYS(x)	Conversion de la date X en nombre de jours depuis le 1er janvier 1970.
DAYOFWEEK(x)	Retourne le jour de la semaine de la date x sous la forme d'un index qui commence à 1 (1=dimanche, 2=lundi...)
DAYOFMONTH(x)	Retourne le jour du mois (entre 1 et 31).
DAYOFYEAR(x)	Retourne le jour de l'année (entre 1 et 366).
SECOND(x), MINUTE(x), HOUR(x), MONTH(x), YEAR(x), WEEK(x)	Retournent respectivement les secondes, minutes, heures, mois, année et semaine de la date.

```
SELECT titre
FROM article
WHERE (TO_DAYS(NOW()) - TO_DAYS(parution)) < 30
```

Cet exemple affiche le titre des articles parus il y a moins de 30 jours.

Prof. Asmaa El Hannani

2TTE-S1

167

## Fonctions d'agregation

Fonction	Description
COUNT([DISTINCT]x,y,...)	Décompte des tuples du résultat par projection sur le ou les attributs spécifiés (ou tous avec '*'). L'option DISTINCT élimine les doublons.
MIN(x), MAX(x), AVG(x), SUM(x)	Calculent respectivement le minimum, le maximum, la moyenne et la somme des valeurs de l'attribut X.

- Une fonction d'agrégation permet de calculer une seule valeur numérique à partir des valeurs de plusieurs tuples pour un attribut donné.  
**Par exemple:** la somme des budgets des projets
- Une fonction d'agrégation prend un attribut en paramètre
- **Important** : Les fonctions d'agrégat ne peuvent en aucun cas être ailleurs que suite au **SELECT** (pas dans la clause **WHERE** par exemple)

Prof. Asmaa El Hannani

2TTE-S1

168

## Exemples

Emp[Eno, Ename, Title, City]  
Pay[Title, Salary]

Project[Pno, Pname, Budget, City]  
Works[Eno#, Pno#, Resp, Dur]

- Budget max des projets de Paris?  
`SELECT MAX (Budget)`  
`FROM Project`  
`WHERE City = 'Paris'`
- Affichage du nombre d'employés  
`SELECT COUNT(*) FROM Emp`
- Noms des projets dont le budget est supérieur au budget moyen?  
`SELECT Pname`  
`FROM Project`  
`WHERE Budget > (SELECT AVG (Budget) FROM Project)`
- Nombre de villes où il y a un projet avec l'employé E4?  
`SELECT COUNT (DISTINCT City)`  
`FROM Project, Works`  
`WHERE Project.Pno = Works.Pno`  
`AND Works.Eno= 'E4'`



JOINTURE

## Les jointures

- Les jointures permettent de **sélectionner les données se trouvant dans plusieurs tables.**
- Il est indispensable de savoir les utiliser car elles permettent de préciser les données sur lesquelles travailler lors d'un :
  - Select (lecture)
  - Update (mise à jour)
  - Delete (suppression)

# Les jointures

## Principe :

- Une jointure a lieu **entre deux tables**. Elle exprime une **correspondance entre** deux clés par **un critère d'égalité**.
- Si les données à traiter se trouvent **dans trois tables**, la **correspondance entre** les trois tables s'exprime par **deux égalités**.

# Exemple

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

On veut afficher les nom , prénom et ville des personnes

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

```
SELECT Personne.Nom, Personne.Prenom, Adresse.Ville FROM  
Personne, Adresse WHERE Personne.Id_adresse=Adresse.Id
```

Nom	Prenom	Ville
Durand	Marie	Perpignan
Simpson	Bart	Springfield
Simpson	Lisa	Springfield



## Les jointures

### Notes :

- Même si l'association des tables est explicité dans la base de données par le biais d'une clé étrangère, il faut toujours indiquer au moteur comment associer les deux tables.
- Il est impossible d'effectuer une jointure entre deux tables si les champs utilisés ne sont pas d'un type compatible.
- Si je dois toutefois effectuer cette jointure, il faut convertir l'un des deux champs.

## Les jointures

- Dans la clause WHERE se mélangent les associations entre les tables et les conditions de sélection des données.

```
Select * from Personne, Adresse  
where Personne.Id_adresse = Adresse.Id  
and Personne.Nom = 'Durand'
```

- Dans cette requête, `Personne.Id_adresse = Adresse.Id` permet d'associer les 2 tables.
- `Personne.nom = 'Durand'` permet d'ajouter un critère de sélection.
- L'ordre n'a pas d'importance.

## Les jointures

- **Problèmes de la jointure par = :**

- Mélange des critères de sélection et des jointures
- Mise en relation des données **uniquement** quand les deux attributs sont remplis (jointure **fermée**)

- Exemple :

- On désire lire toutes les personnes et optionnellement donner leur adresse **si celle-ci est connue**.

- La requête :

```
SELECT Nom,Prenom,Ville FROM Personne, Adresse  
WHERE Personne.Id_adresse = Adresse.Id
```

**ne retournera pas les personnes n'ayant pas d'adresse référencée.**

## Les jointures fermées et ouvertes

- Il existe **une autre syntaxe** permettant de séparer les clauses dédiées à la jointure des critères de sélection.

- Il y a trois type d'associations :

- **INNER JOIN** : jointure fermée, les données doivent être à la fois dans les 2 tables
- **LEFT JOIN** : jointure ouverte, on lit les données de la table de gauche en y associant éventuellement celle de la table de droite.
- **RIGHT JOIN** : jointure ouverte, on lit les données de la table de droite en y associant éventuellement celle de la table de gauche.

## Les jointures fermées

- L'association se fait directement entre les tables en précisant les colonnes concernées.

```
SELECT liste_colonnes FROM table1 INNER JOIN table2 ON  
condition_jointure1 INNER JOIN table3 ON condition_jointure2  
.... WHERE conditions_where
```

Exemple :

```
SELECT * FROM Personne, Adresse WHERE Personne.Id_adresse  
= Adresse.Id and Personne.Nom = 'Durand'
```

```
SELECT * FROM Personne INNER JOIN Adresse ON  
Personne.Id_adresse = Adresse.Id WHERE Personne.Nom =  
'Durand'
```

## Les jointures ouvertes

- Si l'on cherche à préciser optionnellement des données, il faudra utiliser une **jointure ouverte**.

- Exemple:

Pour lire toutes les personnes et **accessoirement** donner leur adresse il faudra écrire :

```
SELECT * from Personne LEFT JOIN Adresse ...
```

Les personnes pour lesquelles l'adresse n'est pas connue auront les champs de la table **adresse** à **NULL**.

## Les jointures ouvertes

- Les mots clés **LEFT JOIN** et **RIGHT JOIN** diffèrent uniquement dans le sens de lecture.

- Les 2 requêtes :

**SELECT \* from Personne LEFT JOIN Adresse ...**

et

**SELECT \* from Adresse RIGHT JOIN Personne ...**

sont parfaitement identiques.

## Exemple avec critère

- Lire toutes les personnes qui s'appellent Durand et accessoirement donner leur adresse si celle-ci est connue. :

**SELECT \* from Personne LEFT JOIN Adresse**  
**ON Personne.Id\_adresse = Adresse.Id**  
**WHERE personne.nom = 'Durand'**

## Jointure naturelle

- La jointure se fait sur la colonne qui porte le même nom dans les deux tables

SELECT ...

FROM <table gauche>

**NATURAL JOIN** <table droite>

Personne			
Id	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
Id_adresse	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

## Exemple

- Avec NATURAL JOIN

SELECT \* FROM Personne **NATURAL JOIN** Adresse WHERE  
personne.nom = 'Durand'

- Avec INNER JOIN

SELECT \* FROM Personne **INNER JOIN** Adresse **ON**  
Personne.Id\_adresse = Adresse.Id\_adresse WHERE personne.nom  
= 'Durand'

Personne			
Id	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
Id_adresse	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

## Produit Cartésien

- Le produit cartésien est la combinaison de toutes les lignes d'une table avec toutes les lignes d'une autre table sans tenir aucun compte du « sens » associé aux données.
- **C'est une opération qui n'a guère d'intérêt en pratique !**
- En SQL, cette opération s'écrit simplement.  
`SELECT * FROM table1, ..., tablen`

## Union, Intersection et Différence

**UNION:** `SELECT liste_attributs FROM table1`  
**UNION**  
`SELECT liste_attributs FROM table2 ;`

**INTERSECTION:** `SELECT liste_attributs FROM table1`  
**WHERE** attribut1 **IN**  
`(SELECT attribut1 FROM table2);`

**DIFFERENCE:** `SELECT liste_attributs FROM table1`  
**MINUS**  
`SELECT liste_attributs FROM table2 ;`

## Union, Intersection et Différence

### ■ Exemples:

- ❑ Donnez la liste des clients vivant soit à Rabat soit à Salé  
(SELECT \* FROM Client WHERE Client.Ville="Rabat")  
**UNION**  
(SELECT \* FROM Client WHERE Client.Ville="Salé")
- ❑ Nom des sommets de plus de 8500 m situés au Népal mais pas sur la frontière avec la Chine ?  
(SELECT nom FROM sommet WHERE altitude > 8500)  
**WHERE nom IN**  
(SELECT nom\_sommet FROM localisation WHERE pays = "Népal")  
**MINUS**  
(SELECT nom\_sommet FROM localisation WHERE pays = "Chine");

## Les valeurs nulles

- Les valeurs de certains attributs sont inconnus. Ils sont mis à **NULL**
- Les comparateurs **IS NULL** et **IS NOT NULL** permettent de tester si une valeur est nulle
- Toute comparaison avec **NULL** donne un résultat qui n'est ni vrai ni faux mais une troisième valeur **UNKNOWN**
- **NULL** est un mot clé et non pas une constante
  - ❑ TRUE 1
  - ❑ UNKNOWN ½
  - ❑ FALSE 0

## Le groupement

- Avec les fonctions d'agrégation, SQL permet de grouper des lignes de données ayant des valeurs communes ;
  - ainsi, on peut formuler par exemple, une requête qui liste le nombre de clients par ville

- Cette possibilité est explicité par la clause **GROUP BY**

- Ainsi pour compter le nombre de fois où une valeur est utilisée par groupe on écrira :

```
SELECT champ2, COUNT(champ1)
FROM table 1
GROUP BY champ2
```

## Le groupement et la clause HAVING

- Si l'on désire créer des critères de sélection qui portent sur un ensemble de données, il faudra associer la clause **GROUP BY** à la clause **HAVING** (ou **NOT HAVING**).

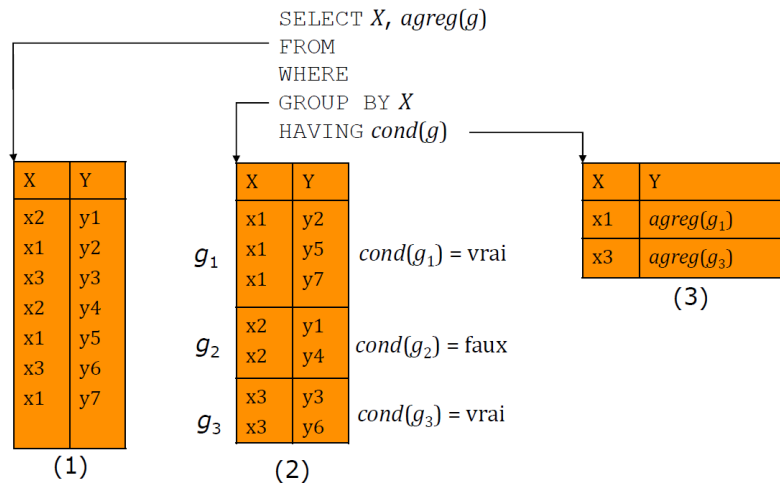
- Exemple, on désire sélectionner les données dont le nombre de répétition est supérieur à N.

```
SELECT * FROM table1
GROUP BY champ2
HAVING COUNT(champ2) > N
```

- On rappelle que la condition de la clause **WHERE** ne peut en aucun cas inclure des fonctions d'agrégats.



## Le groupement



## Le groupement

### ■ Exemple :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

- On désire connaître le nombre de personne vivant à chaque adresse :  
**SELECT** **personne.id\_adresse**, **adresse.voie**, **adresse.cp**, **adresse.ville**,  
**COUNT(\*)** **FROM** **personne**, **adresse** **WHERE** **personne.id\_adresse** =  
**adresse.id** **GROUP BY** **id\_adresse**

## Le groupement

### ■ Exemple :

Personne			
<u>Id</u>	Nom	Prenom	Id_adresse
1	Durand	Marie	4
2	Simpson	Bart	3
3	Dubois	Jean	NULL
4	Simpson	Lisa	3

Adresse			
<u>Id</u>	Voie	CP	Ville
1	1, rue Ici	75002	Paris
2	12, rue labas	75015	Paris
3	742 Evergreen Terrasse		Springfield
4	Chemin perdu	66000	Perpignan

- On désire sélectionner les adresses ayant au moins 2 occupants :

```
SELECT personne.id_adresse, adresse.voie, adresse.cp, adresse.ville  
FROM personne , adresse WHERE personne.id_adresse = adresse.id  
GROUP BY id_adresse HAVING count(*) >= 2
```

## Optimisation de requêtes

- Lorsque l'on écrit une requête, il faut penser aux conséquences calculatoires (en terme de calcul et de ressources mémoires)
- En effet, chaque sous-requête, chaque jointure, chaque **ORDER BY**, chaque **GROUP BY**, chaque **HAVING** a pour conséquence de créer une table temporaire : si le nombre de données est très important, cela prend nécessairement du temps (le SGBD n'est pas magique)
- Il faut donc que vous pensiez au coût de vos requêtes : entre 2 requêtes, pour un même résultat, laquelle des 2 créera le moins de tables temporaires

## Sous-requêtes dans la clause FROM

- Les sous-requêtes sont valides dans la clause FROM d'une commande SELECT. Voici une syntaxe que vous allez rencontrer :

**SELECT ... FROM (<subquery>) AS <name> ...**

- La clause AS <name> est obligatoire, car les tables de la clause FROM doivent avoir un nom.
- Toutes les colonnes de la sous-requête <subquery> doivent avoir des noms distincts

- Exemple:

```
SELECT MAX(sum_column1)
FROM (SELECT SUM(column1) AS sum_column1
        FROM t1 GROUP BY column2) AS t2;
```

## Condition de sous-requête (1)

- SQL permet de comparer une expression ou une colonne au résultat d'une autre requête SELECT. Cette condition est dite **condition de sous-requête** et les 2 requêtes sont dites **requête imbriquées**

- Bien-sur, une sous-requête peut faire appel à une autre sous-requête etc.

- Les requêtes imbriquées dans SQL peuvent être utilisés pour
  - **comparer** une expression ou une liste d'expression au résultat d'une autre requête SELECT,
  - **déterminer l'appartenance** d'une expression ou l'existence d'une ligne dans le résultat d'une requête SELECT etc.

## Condition sous-requête (2)

- Une condition de sous-requête peut être formulé selon l'une des possibilité suivante :
  - ... WHERE Exp **Opérateur\_de\_comparaison** {**ALL** | **ANY** | **SOME**} (Requête\_SELECT)
  - ... WHERE Exp [NOT] **IN** (Requête\_SELECT)
  - ... WHERE [NOT] **EXISTS** (Requête\_SELECT)
- L'évaluation des sous-requêtes peut renvoyer plusieurs valeurs qui sont interprété comme suit :
  - **ALL** : la condition est vrai, si la comparaison est vrai pour chacune des valeurs retournées (si l'expression de condition est de type numérique et si l'opérateur de comparaison est > alors la sous-requête sera équivalente à l'extraction de la valeur maximale car si la condition est vrai pour le maximum, elle est aussi vrai pour toutes les autres...)

## Condition sous-requête (3)

- **ANY** : la condition est vrai si la comparaison est vrai pour au moins une des valeurs retournée (si l'opérateur de comparaison est > alors la sous-requête sera équivalente à l'extraction de la valeur minimale ....)
- **SOME = ANY**
- **IN** : la condition est vrai si la comparaison est vrai pour une des valeurs retournées par la sous-requête
- **EXISTS** : il est un peu différent des autres opérateurs, il renvoie un booléen (vrai ou faux) selon le résultat de la sous-requête. Si l'évaluation de la sous-requête donne lieu a une ou plusieurs ligne, la valeur retourné est vrai. Cette valeur sera fausse dans le cas contraire.

## Exemples de sous-requêtes

- Lister tous les articles dont la quantité en stock est > à toutes quantité commandé du même article
  - ❑ `SELECT IdArticle, Designation, QtéStock from Article WHERE QtéStock > ALL (SELECT QtéComm FROM ligne_commande WHERE Article.IdArticle = ligne_commande.IdArticle)`
  - ❑ `SELECT IdArticle, Designation, QtéStock FROM Article WHERE QtéStock > (SELECT MAX(QtéCommandé) FROM ligne_commande WHERE Article.IdArticle = ligne_commande.IdArticle)`
- Lister tous les articles dont la quantité est > a au moins une quantité commandé au même article
  - ❑ `... > ANY (SELECT QtéCommandé ...)`
  - ❑ `ou > SELECT Min(QtéCommandé)`

## Exemples de sous-requêtes

- Lister tous les clients parisien qui ont passé une commande entre le 1er janvier 2013 et aujourd'hui et dont la quantité commandé est égale à la quantité en stock
  - ❑ `SELECT DISTINCT IdClient FROM Commande WHERE IdClient IN (SELECT IdClient FROM Client WHERE ville="Paris") AND DateComm BETWEEN "01-Jan-13" AND SYSDATE)`
- Mémo technique
  - ❑ (`<=` ou `=`) 1 seul valeur
  - ❑ (`IN`, `ALL`, `ANY`) une liste de valeur
  - ❑ `EXISTS`, un ensemble de valeur

## Exercice 1

- Soit le schéma de la BD représentant les achats des clients d'une entreprise informatique:
  - **Client** [IDC, Nom, Prénom, Ville, AnnéeN]
  - **Produit** [IDP, Description, Prix]
  - **Achat** [IDA, RefC#, RefP#, Quantité, Date]
- Ecrivez les requêtes SQL permettant de donner:
  - a. Les noms et les prénoms des clients nés après 1971.
  - b. Les produits dont le prix est entre 3000 et 7000Dh, triés par ordre décroissant.
  - c. Le numéro et le nom des clients dont le nom commence par 'M'.
  - d. Le nombre des clients habitants à El Jadida.
  - e. Le produit le plus cher.
  - f. Les produits qui n'ont pas été vendus.
  - g. Les noms des clients de Rabat qui ont acheté des produits.
  - h. Les produits coûtant plus de 1000 Dh, achetés par Ahmed Chaouki.
  - i. Le coût total des achats de chaque client.

## Solution

- Ecrivez les requêtes SQL permettant de donner:
  - a. Les noms et les prénoms des clients nés après 1971.  
`Select nom, prénom from Client where AnnéeN > 1971`
  - b. Les produits dont le prix est entre 3000 et 7000Dh, triés par ordre décroissant.  
`Select * from Produit where prix between 3000 and 7000 order by prix desc`
  - c. Le numéro et le nom des clients dont le nom commence par 'M'.  
`Select IDC, nom from Client where Nom like "M%"`
  - d. Le nombre des clients habitants à El Jadida.  
`Select count(*) from Client where Ville="ElJadida"`
  - e. Le produit le plus cher.  
`Select * from Produit where prix = (select max(prix) from Produit)`
  - f. Les produits qui n'ont pas été vendus.  
`Select IDP from Produit where IDP not in (select RefP from Achat)`

## Solution-suite

**Client** [IDC, Nom, Prénom, Ville, AnnéeN]  
**Produit** [IDP, Description, Prix]  
**Achat** [IDA, RefC#, RefP#, Quantité, Date]

- Ecrivez les requêtes SQL permettant de donner:

- g. Les noms des clients de Rabat qui ont acheté des produits.  

```
Select nom from Client inner join Achat on Client.IDC = Achat.RefC  
where ville ="Rabat"
```

Ou

```
Select nom from Client where IDC in (select RefC from Achat) and  
ville ="Rabat"
```
- h. Les produits coûtant plus de 1000 Dh, achetés par Ahmed Chaouki.  

```
Select Description from Produit inner join Achat on Produit.IDP =  
Achat.refP inner join Client on Achat.RefC = Client.IDC where nom  
= "Chaouki " and prénom ="Ahmed" and prix > 1000
```
- i. Le coût total des achats de chaque client.  

```
Select IDC, nom, prénom, sum(prix*quantité) as sommePrix from  
Produit inner join Achat on Produit.IDP = Achat.refP inner join  
Client on achat.RefC = Client.IDC group by IDC, nom, prénom
```

## Les vues

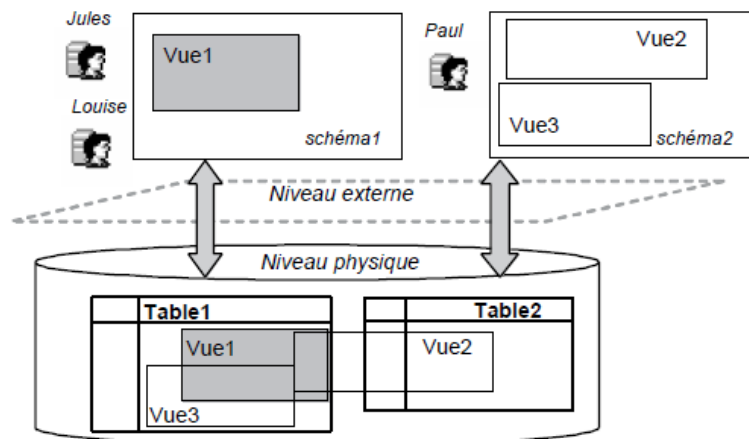
### ■ Définitions

- Une vue est une table virtuelle
- Elle n'existe pas dans la base
- Elle est construite à partir du résultat d'un SELECT.
- La vue sera vu par l'utilisateur comme une table réelle.

### ■ Les vues permettent

- des accès simplifiés aux données
- l'indépendance des données
- la confidentialité des données : restreint les droits d'accès à certaines colonnes ou à certains n-uplets.

## Les vues



## Les vues

- Création d'une vue :

Syntaxe:

```
CREATE VIEW view_name [(column_list)] AS select_statement  
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Exemple:

```
CREATE VIEW ClientParis (NumCl, NomCl, TélCl) AS SELECT  
NumCl, NomCl, TélCl FROM Client WHERE ville="Paris"
```

- Suppression d'une vue : **DROP VIEW** view-name



## Requêtes et vues

- Pour récupérer les données de vues, on procédera comme si l'on était en face d'une table classique
- `SELECT * FROM ClientParis WHERE NomCl LIKE "Dup%"`
- En réalité, cette table est virtuelle et est reconstruite à chaque appel de la vue `ClientParis` par exécution du `SELECT` constituant la définition de la vue.

## Protection par les vues

- Les vues jouent un rôle important pour la confidentialité en permettant de spécifier de façon très fine les données auxquels un utilisateur a le droit d'accéder.
- On pourra spécifier un accès à l'affectation des employés dans les départements en définissant la vue suivante :  
`CREATE VIEW affectation(nom_emp, nom_dept) AS`  
`SELECT e.nom_emp, d.nom_dept`  
`FROM employe AS e, departement AS d`  
`WHERE e.nom_dept = d.nom_dept;`
- Un utilisateur dont les droits d'accès se limitent à la vue `affectation` ne pourra connaître ni le numéro, ni le salaire d'un employé, ni le numéro et le directeur d'un département.