# BD et SGBD

## Une **base de données**

est un ensemble d'informations, structurées de façon à pouvoir être consultées (extraites), modifiées, ajoutées, supprimées.

## Un **Système de Gestion d'une Bases de Données (SGBD)**

est un **logiciel** qui gère un ensemble de bases de données.

- Il permet
  - la mémorisation et la **représentation interne** des données.
  - la manipulation et la consultation de ces données par l'intermédiaire d'une **interface utilisateur** (langage d'interrogation, API...).
- Chaque SGBD est fondé sur un **modèle de représentation** des données.

# Modèles de données

## Les modèles

Un **modèle de données** est une façon de structurer les données.
C'est un paradigme, une logique d'organisation.
Il existe plusieurs modèles de données :

- relationnel
- hiérarchique
- réseau
- objet
- relationnel-objet
- ... différents modèles dits « NoSQL »

## Le cours de BDD

est consacré uniquement au **modèle relationnel**

# Bases relationnelles

## Le modèle relationnel

- formalisé par CODD, en 1970
- devenu le plus utilisé depuis les années 1980.
- les données sont structurées en un **ensemble**
  - de **relations** $\rightarrow$ selon une approche mathématique
  - de **tables** $\rightarrow$ selon une approche plus « graphique »

## Quelques SGBD relationnels

- Oracle Database
- DB2 (IBM)
- Microsoft SQL Server
- PostgreSQL
- MySQL
- Sqlite (non client-serveur)

# La **relation**, dans le modèle relationnel

## Le **schéma** d'une relation définit sa structure

Il comporte le **nom** de la relation et une suite d'attributs.
Chaque attribut est défini par un **nom** et un **domaine** (l'ensemble des valeurs admises pour cet attribut ; cf le type dans un langage de programmation)

## Exemple : la relation étudiants

possède 3 attributs :

$$(\textbf{nom} : \textbf{chaîne},\ \textbf{prénom} : \textbf{chaîne},\ \textbf{groupe} : \textbf{entier})$$

Graphiquement :

| | *chaîne* | *chaîne* | *entier* |
|---|---|---|---|
| **étudiants** : | **nom** | **prénom** | **groupe** |
| | | | |

# La **relation**, dans le modèle relationnel

## Une relation contient des **tuples**

Une relation est un **ensemble** de n-uplets appelés **tuples**.

- le tuple comporte une valeur pour chaque attribut.
- chaque valeur doit appartenir au domaine correspondant.

## **Exemple : la relation étudiants**

Chaque tuple de cette relation est un triplet (car 3 attributs) composé de 2 chaînes puis d'un entier.

$$\{(\textit{Dupont}, \textit{Alfred}, 2), (\textit{Smith}, \textit{John}, 1), (\textit{Durand}, \textit{Alfred}, 1)\}$$

Graphiquement :

`étudiants` :

| *chaîne* | *chaîne* | *entier* |
|---|---|---|
| **nom** | **prénom** | **groupe** |
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

# La **relation**, dans le modèle relationnel

- chaque tuple est une information que l'on souhaite représenter dans la base :
  Exemple : « un étudiant s'appelle Dupont Alfred et il est dans le groupe 2 »
- une relation est un ensemble : l'**ordre de ses éléments n'a pas d'importance.** Exemple :
  $\{(\mathit{Dupont}, \mathit{Alfred}, 2), (\mathit{Smith}, \mathit{John}, 1), (\mathit{Durand}, \mathit{Alfred}, 1)\} =$
  $\{(\mathit{Smith}, \mathit{John}, 1), (\mathit{Durand}, \mathit{Alfred}, 1), (\mathit{Dupont}, \mathit{Alfred}, 2)\}$
  Autrement dit : **l'ordre des lignes dans la table n'est pas significatif**.
- bien distinguer le schéma de la relation qui définit sa structure (nombre de colonnes, intitulés des colonnes et domaines) et la relation elle-même (les données : les tuples)

# Les relations

- **Schéma de la relation** : N-uple :
  $(Attribut_1 : Domaine_1,\ \ldots\ ,\ Attribut_N : Domaine_N)$

| $Domaine_1$ | ... | $Domaine_N$ |
|---|---|---|
| $Attribut_1$ | ... | $Attribut_N$ |
| | | |

  - **Attribut** $\equiv$ Nom de colonne
  - **Domaine** de définition $\equiv$ Type

- **Relation** : ensemble de N-uples (les « tuples »)
  $\{(a_1, ..., a_N), (b_1, ..., b_N), \ldots\}$

| $Domaine_1$ | ... | $Domaine_N$ |
|---|---|---|
| $Attribut_1$ | ... | $Attribut_N$ |
| $a_1$ | ... | $a_N$ |
| $b_1$ | ... | $b_N$ |
| ... | ... | ... |

## 5 opérations fondamentales constituent l'algèbre relationnel

- pour extraire de l'information d'une base de donnée, on utilise des **opérations** sur les tables.
- dans le modèle théorique, 5 opérations constituent l'algèbre relationnel :
  - **union** de 2 tables
  - **différence** de 2 tables
  - **projection** d'une table
  - **restriction** d'une table
  - **produit cartésien** de 2 tables

  **Le résultat de chaque opération est une table**
- on ajoutera l'opération de **jointure**, qui est en fait une combinaison de 2 autres, mais qui est l'une des plus couramment utilisée et mérite d'être étudiée en tant que telle.

# Opérations relationnelles : la **projection**

La **projection** d'une relation $R$ est une relation $R'$ obtenue en **ne conservant que certains attributs**.

## Sous-schéma

Si $s = (A_1, A_2, ...A_n)$, un sous-schéma de $s$ est un schéma $s' = (A_{i_1}, A_{i_2}, ...A_{i_m})$ où $i_k \in [1, n]$
Ex : $s' = (nom : chaine, groupe : entier)$

## **Projection** de R selon un sous-schéma

soit $R$ de schéma $(A_1, A_2, ...A_n)$ et $s' = (A_{i_1}, A_{i_2}, ...A_{i_m})$ un sous-schéma de $s$,

$$\pi_{s'}(R) = \{(x_1, ..x_m) | \exists (y_1, ...y_n) \in R, \forall k \in [1, m], y_{i_k} = x_k\}$$

# Opérations relationnelles : la **projection**

## Exemple de projection

Projection de la relation « étudiants » sur le sous-schéma *nom*, *groupe*) :

$$\pi_{nom,groupe}(etudiants) = \{(Dupont, 2), (Smith, 1), (Durand, 1)\}$$

| nom | prénom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

$\Rightarrow$

| nom | groupe |
|---|---|
| Dupont | 2 |
| Smith | 1 |
| Durand | 1 |

# Opérations relationnelles : la **restriction**

La **restriction** d'une relation $R$ est une relation $R'$ de même schéma obtenue en **ne conservant que certains tuples**.

## **Qualification** (ou condition)

Une qualification est une expression booléenne dans laquelle peuvent intervenir des constantes et des noms d'attributs. Ex :

- groupe $= 1$
- nom $=$ prenom
- prénom=Alfred et groupe $>1$

## Qualification d'un tuple

Un tuple est **qualifié** si l'évaluation de la qualification pour les valeurs du tuples a pour résultat VRAI. Par exemple pour la qualification « prénom=Alfred et groupe $>1$ », $(\mathit{Dupont}, \mathit{Alfred}, 2)$ est qualifié mais pas les 2 autres tuples.

# Opérations relationnelles : la **restriction**

**Restriction**

La restriction d'une relation $R$ par une qualification $q$ est une relation définie sur le même schéma que $R$.

$$\sigma_q(R) = \{t|\ t \in R \text{ et t qualifié pour } q\}$$

# Opérations relationnelles : la **restriction**

## Exemple

Restriction de la relation « étudiants » par la qualification « groupe=1 »

$$\sigma_{groupe=1}(etudiants) = \{(Smith, John, 1), (Durand, Alfred, 1)\}$$

| nom | prénom | groupe | |
|---|---|---|---|
| Dupont | Alfred | 2 | False |
| Smith | John | 1 | True |
| Durand | Alfred | 1 | True |

$\Rightarrow$

| nom | prénom | groupe |
|---|---|---|
| Smith | John | 1 |
| Durand | Alfred | 1 |

# Opérations relationnelles : la **restriction**

## Exemple

Restriction de la relation « étudiants » par la qualification « nom=prénom »

$$\sigma_{nom=prenom}(etudiants) = \emptyset$$

| nom | prénom | groupe | |
|---|---|---|---|
| Dupont | Alfred | 2 | False |
| Smith | John | 1 | False |
| Durand | Alfred | 1 | False |

$\Rightarrow$

| nom | prénom | groupe |
|---|---|---|

**NB : on obtient ici une relation vide. Obtenir une relation vide n'est pas une erreur.**

# Opérations relationnelles : **union** et **différence**

## Schémas

- Ces opérations font intervenir 2 tables qui doivent être **définies sur le même schéma**
- la table résultante est aussi sur le même schéma

## $R \cup R'$ : **union** de deux relations $R$ et $R'$

- tous les tuples de $R$ ou de $R'$

## $R - R'$ : **différence** de deux relations $R$ et $R'$

- tous les tuples de $R$ qui ne sont pas dans $R'$.

Exemple :
$R = \{(Dupont, Alfred, 2), (Smith, John, 1), (Durand, Alfred, 1)\}$
$R' = \{(Smith, John, 1), (Smith, Arthur, 3)\}$
$R - R' = \{(Dupont, Alfred, 2), (Durand, Alfred, 1)\}$

# Produit cartésien

**$T \times T'$ : produit cartésien de deux relations $T$ et $T'$**

- $T$ de schéma $s$ et $T'$ de schéma $s'$.
- $T \times T'$ défini sur le schéma $s.s'$ (concaténation).
- $T \times T'$ contient les tuples $(x_1, \ldots x_n, y_1, \ldots y_{n'})$ où $(x_1, \ldots x_n) \in R$ et $(y_1, \ldots y_{n'}) \in R'$

**remarque**

- le nombre d'attributs (« colonnes ») est la somme des nombres d'attributs des tables d'origine
- le nombre de tuples (« lignes ») est le produit des nombres de tuples des tables d'origine

# Produit cartésien

## Exemple de produit cartésien

étudiants

| nom | prénom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

$\times$

groupes

| numéro | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |

étudiants $\times$ groupes

| nom | prénom | groupe | numéro | salle |
|---|---|---|---|---|
| Dupont | Alfred | 2 | 1 | B-27 |
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |
| Smith | John | 1 | 2 | A-14 |
| Durand | Alfred | 1 | 2 | A-14 |

Quelle signification donner aux données de cette table ?

# Jointure

## Jointure

- En pratique on utilise la jointure plutôt que le produit cartésien
- Une jointure est une combinaison d'un produit cartésien et d'une restriction
- La jointure fait intervenir 2 tables quelconques et une **condition de jointure**
- La **condition de jointure** doit porter sur des attributs de **chacune** des tables
- La condition de jointure la plus simple est l'**égalité entre un attribut de la table 1 et un attribut de la table 2**
- La condition de jointure peut être plus complexe mais nous commencerons par l'égalité simple

## Exemple de condition de jointure

étudiants.groupe = groupes.numéro

# Jointure

## Exemple de jointure

étudiants

| nom | prénom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

groupes

| numéro | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |

jointure sur étudiants.groupe = groupes.numéro

| nom | prénom | groupe | numéro | salle |
|---|---|---|---|---|
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |

Remarque sur la sémantique :

- les attributs étudiants.groupe et groupes.numéro portent sur la même "entité" (un numéro de groupe)
- on n'a conservé que les tuples significatifs

## **Simple Query Langage** (SQL)

- est un langage d'interrogation des SGBD relationnelles.
- Né dans les années 70-80, il a fait l'objet de plusieurs normes successives (86, 89, 92 [SQL2], 99 [SQL3], ..., [SQL :2011]).
- Par ailleurs les différents SGBD peuvent implémenter tout ou partie ... ou des extensions de SQL.

Fonctionnalités de SQL :

- Définition, modification ou suppression de schémas de tables (DDL) : `create, alter, drop`.
- Manipulation de données (DML) : `select, update, insert, delete`.
- Administration et gestion des droits (DCL) : `grant, revoke`.

# Select

**Select** : commande permettant de consulter (« extraire ») les données

- Sa syntaxe minimale est

  `select` *noms de colonnes* `from` *nom de table*

- Exemple : `select Nom,Groupe from R`
- Sous cette forme, c'est une **projection**. Les noms de colonnes forment le sous-schéma.
- Pour le sous-schéma, on peut utiliser la notation `*` qui désigne le schéma complet : l'expression `select * from R` désigne la projection « identité ».
- **Le résultat d'une requête « select » est une table**

## **Select** : premiers exemples

```
select * from etudiants
```

| nom | prenom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

```
select nom,groupe from etudiants
```

| nom | groupe |
|---|---|
| Dupont | 2 |
| Smith | 1 |
| Durand | 1 |

# Restriction(sélection) : `where`

## La clause **where**

- permet de réaliser une **restriction** du résultat (choix de lignes)
- optionnelle
- est suivie d'une qualification.

```
select * from etudiants
   where groupe<=1 and prenom='Alfred'
```

| nom | prenom | groupe |
|---|---|---|
| Durand | Alfred | 1 |

# Tris : `order by`

## L'ordre des lignes obtenues est **incertain**

Sauf si l'on indique explicitement un ordre de classement par la clause **order by** .

```
select * from etudiants order by groupe
```

| | | |
|---|---|---|
| Smith | John | 1 |
| Durand | Alfred | 1 |
| Dupont | Alfred | 2 |

*NB : ordre entre les lignes de même groupe non garanti*

```
select * from etudiants order by groupe desc
```

| | | |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

*NB : ordre entre les lignes de même groupe non garanti*

```
select * from etudiants order by groupe desc, nom asc
```

| | | |
|---|---|---|
| Dupont | Alfred | 2 |
| Durand | Alfred | 1 |
| Smith | John | 1 |

# Du modèle à la pratique...

## Différence par rapport au modèle ensembliste strict

Une table peut contenir des doublons.
La clause **distinct** permet de les éliminer du résultat, si besoin.

## exemple

```
select prenom
from etudiants
```

| prenom |
|---|
| Alfred |
| Alfred |
| John |

```
select distinct prenom
    from etudiants
```

| prenom |
|---|
| Alfred |
| John |

# Du modèle à la pratique...

## Absence de valeur

- sauf indication contraire, la valeur d'un attribut peut ne pas être définie (absence de valeur)
- l'absence de valeur est un état particulier appelé **NULL**
- testé par le prédicat `IS NULL` (jamais =`NULL`, puisque ce n'est pas une valeur)
- négation : `IS NOT NULL`

NB : on verra ultérieurement que, dans la définition d'une relation, on peut interdire l'absence de valeur

# Les spécifications d'attributs

## « Renommage » des attributs

- le mot clé **as** placé après la spécification d'un attribut définit un nom

## Renommage

```
select nom as lastname,
       prenom as firstname
    from etudiants
```

| lastname | firstname |
|---|---|
| Dupont | Alfred |
| Smith | John |
| Durand | Alfred |

# Les spécifications d'attributs

## Une spécification d'attribut est une expression

- faisant intervenir **noms d'attributs**, **constantes**, **opérateurs**, **appels de fonction**
- les chaînes sont délimitées par des **simples** quotes
- les noms d'attributs peuvent être délimités par des **doubles** quotes
- opérateurs
  - arithmétiques (usuels)
  - de comparaison (usuels)
  - de concatenation : `||`
  - test nullité : `IS NULL`, `IS NOT NULL`
- appels de fonctions
  - voir doc PostgreSQL (ou du SGBD utilisé)

## Il est fortement recommandé d'utiliser le renommage (« as ») quand la spécification est une expression complexe

# Les spécifications d'attributs

## Exemple

```
select prenom || ' ' || nom  as "Nom complet"
     from etudiants
```

| Nom complet |
| --- |
| Alfred Dupont |
| John Smith |
| Alfred Durand |

## Délimitation des noms d'attributs par " "

Obligatoire si le nom comporte espaces ou majuscules.

# Op. ensemblistes : `union`, `except`, `intersect`

## combiner 2 relations (ou plus)

union : **UNION**, différence : **EXCEPT**, intersection : **INTERSECT**

## condition

- les relations doivent avoir le même nombre de colonnes
- les domaines (types) des colonnes doivent être compatibles (par conversion de type)

(voir le cours sur l'algèbre relationnelle)

- les noms d'attributs du select (« premier » select) déterminent ceux du résultat
- les doublons sont éliminés par défaut, sauf si le mot clé **ALL** est ajouté (ex : `UNION ALL`)
- une clause « `order by` » finale peut être utilisée

# Op. ensemblistes : `union`, `except`, `intersect`

## Exemple

```
select prenom as mix
 from etudiants
 union
  select nom from etudiants
order by mix
```

| mix |
|---|
| Alfred |
| Dupont |
| Durand |
| John |
| Smith |

```
select prenom as mix
 from etudiants
 union all
  select nom from etudiants
```

| mix |
|---|
| Alfred |
| John |
| Alfred |
| Durand |
| Smith |
| Dupont |

# Jointure

## Syntaxe SQL

**select** *spécification_attributs*
**from** *table1* **join** *table2* **on** *condition*

## Exemple

```
select * from etudiants
join groupes on etudiants.groupe=groupes.numero
```

étudiants

| nom | prenom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Durand | Alfred | 1 |

$\bowtie_{groupe=numero}$

groupes

| numero | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |

| nom | prenom | groupe | numero | salle |
|---|---|---|---|---|
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |

# Jointure

## Les alias de nom de table

Dans une requête select, on peut attribuer un alias à chaque table (nom local valide uniquement dans la requête)

- pour simplifier ou clarifier l'écriture
- pour réaliser une auto-jointure (jointure d'une table avec elle-même)

## Exemple

```
select t1.nom, t1.prenom, t1.groupe, t2.salle
 from etudiants as t1
 join groupes as t2
    on t1.groupe=t2.numero
```

| nom | prenom | groupe | salle |
|---|---|---|---|
| Smith | John | 1 | B-27 |
| Durand | Alfred | 1 | B-27 |
| Dupont | Alfred | 2 | A-14 |

# Les différents types de jointure.

## Jointure interne : `inner join`

- seules les lignes de chaque table qui répondent à la condition de jointure figurent dans le résultat
- c'est la jointure par défaut (le mot `inner` est optionnel)

## Jointures externes : `left join`, `right join`, `full join`

- **`left join`**
  - contient tous les tuples de la jointure interne
  - de plus, les tuples de la 1ère table (« gauche ») qui ne remplissent jamais la condition de jointure sont conservés (avec état NULL pour les attributs correspondant à la deuxième table)
- **`right join`** : idem, symétriquement
- **`full join`** : idem, dans les 2 sens.

## Jointure interne (`join` ou `inner join` )

étudiants

| nom | prenom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Cheval | Aline | 4 |
| Durand | Alfred | 1 |

groupes

| numero | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |
| 3 | A-4 |

```
select * from etudiants
join groupes on etudiants.groupe=groupes.numero
```

| nom | prenom | groupe | numero | salle |
|---|---|---|---|---|
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |

## Jointure externe gauche (`left join` ou `left outer join` )

étudiants

| nom | prenom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Cheval | Aline | 4 |
| Durand | Alfred | 1 |

groupes

| numero | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |
| 3 | A-4 |

```
select * from etudiants
left join groupes on etudiants.groupe=groupes.numero
```

| nom | prenom | groupe | numero | salle |
|---|---|---|---|---|
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |
| Cheval | Aline | 4 | *NULL* | *NULL* |

## Jointure externe droite (`right join` ou `right outer join` )

étudiants

| nom | prenom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Cheval | Aline | 4 |
| Durand | Alfred | 1 |

groupes

| numero | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |
| 3 | A-4 |

```
select * from etudiants
right join groupes on etudiants.groupe=groupes.numero
```

| nom | prenom | groupe | numero | salle |
|---|---|---|---|---|
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |
| *NULL* | *NULL* | *NULL* | 3 | A-4 |

## Jointure externe complète (`full join` ou `full outer join` )

étudiants

| nom | prenom | groupe |
|---|---|---|
| Dupont | Alfred | 2 |
| Smith | John | 1 |
| Cheval | Aline | 4 |
| Durand | Alfred | 1 |

groupes

| numero | salle |
|---|---|
| 1 | B-27 |
| 2 | A-14 |
| 3 | A-4 |

```
select * from etudiants
full join groupes on etudiants.groupe=groupes.numero
```

| nom | prenom | groupe | numero | salle |
|---|---|---|---|---|
| Smith | John | 1 | 1 | B-27 |
| Durand | Alfred | 1 | 1 | B-27 |
| Dupont | Alfred | 2 | 2 | A-14 |
| Cheval | Aline | 4 | *NULL* | *NULL* |
| *NULL* | *NULL* | *NULL* | 3 | A-4 |

# SQL : Les fonctions de regroupement (ou agrégation)

## Exemple

Soit une table de relevé téléphonique (`decompte`)

| date | numero | cout |
|---|---|---|
| 2016-1-5 | +33320434343 | 0.40 |
| 2016-2-29 | +33320434343 | 0.10 |
| 2016-1-5 | +33320434343 | 0.05 |
| 2016-1-5 | +33328778551 | 0.12 |
| 2016-2-29 | +33320434343 | 0.60 |

Comment obtenir la somme des coûts ?

```
select sum(cout) from decompte
```

Le résultat comporte une seule ligne :

| sum(cout) |
|---|
| 1.27 |

# SQL : Les fonctions de regroupement

- les expressions rencontrées jusqu'à maintenant renvoyaient un calcul pour chaque ligne.
- au contraire, la fonction **sum()** a fourni un seul résultat à partir de **plusieurs lignes** (un groupe de lignes)
- dans notre exemple toutes les lignes sont réunies en un **unique groupe** et la fonction renvoie **un** résultat pour ce groupe.

## Exemple

```
select sum(cout) from decompte
```

| date | numero | cout |
|---|---|---|
| 2016-1-5 | +33320434343 | 0.40 |
| 2016-2-29 | +33320434343 | 0.10 |
| 2016-1-5 | +33320434343 | 0.05 |
| 2016-1-5 | +33328778551 | 0.12 |
| 2016-2-29 | +33320434343 | 0.60 |

| sum(cout) |
|---|
| 1.27 |

# SQL : Les fonctions de regroupement

## Fonctions de regroupement prédéfinies

| nom | rôle | argument |
|---|---|---|
| COUNT | dénombrement | quelconque |
| AVG | moyenne | numérique |
| SUM | somme | numérique |
| MAX | maximum | quelconque |
| MIN | minimum | quelconque |

NB :
`count(*)` compte le nombre de lignes
`count(` *attribut* `)` compte le nombre de lignes dont l'attribut **n'est pas NULL**

PostgreSQL propose d'autres fonctions de regroupement, qui n'existent pas nécessairement dans un autre SGBD

# SQL : Les fonctions de regroupement

## Exemple

```
select avg(cout) as moyenne, count(*) as nombre
 from decompte
```

| date | numero | cout |
|---|---|---|
| 2016-1-5 | +33320434343 | 0.40 |
| 2016-2-29 | +33320434343 | 0.10 |
| 2016-1-5 | +33320434343 | 0.05 |
| 2016-1-5 | +33328778551 | 0.12 |
| 2016-2-29 | +33320434343 | 0.60 |

| moyenne | nombre |
|---|---|
| 0.254 | 5 |

# SQL : Les fonctions de regroupement : group by

## la clause group by

- précise le critère de constitution des groupes
- toutes les lignes ayant le même résultat vis à vis du critère sont dans un même groupe

## Exemple

```
select sum(cout) from decompte group by numero
```

| date | numero | cout | numero | sum(cout) |
|---|---|---|---|---|
| 2016-1-5 | +33320434343 | 0.40 | } | 1.15 |
| 2016-2-29 | +33320434343 | 0.10 | | |
| 2016-1-5 | +33320434343 | 0.05 | | |
| 2016-2-29 | +33320434343 | 0.60 | | |
| 2016-1-5 | +33328778551 | 0.12 | } | 0.12 |

# SQL : Les fonctions de regroupement : group by

## Exemple

```
select sum(cout) from decompte group by date,numero
```

| date | numero | cout | date,numero | sum(cout) |
|---|---|---|---|---|
| 2016-1-5 | +33320434343 | 0.40 | } | 0.45 |
| 2016-1-5 | +33320434343 | 0.05 | } | |
| 2016-2-29 | +33320434343 | 0.60 | } | 0.70 |
| 2016-2-29 | +33320434343 | 0.10 | } | |
| 2016-1-5 | +33328778551 | 0.12 | } | 0.12 |

# SQL : Les fonctions de regroupement : group by

Les attributs qui servent de critère de regroupement peuvent faire partie du schéma

## Exemple (correct)

```
select numero, date, sum(cout) as total
 from decompte
 group by date,numero
```

| date | numero | sum(cout) |
|---|---|---|
| 2016-1-5 | +33320434343 | 0.45 |
| 2016-2-29 | +33320434343 | 0.70 |
| 2016-1-5 | +33328778551 | 0.12 |

## Exemple (incorrect) ⇒ **erreur SQL**

```
select numero, date, sum(cout) as total
 from decompte
 group by numero
```

## la clause **having**

permet de faire une sélection portant sur
**le résultat d'une fonction de regroupement**.
La clause having est liée à une clause « group by ».

```
select date,numero,sum(cout) from decompte
    group by date,numero
    having sum(cout)>0.5
```

| date | numero | sum(cout) |
|---|---|---|
| 2016-2-29 | +33320434343 | 0.7 |

```
select numero,sum(cout) from decompte
    group by numero
    having count(*)=1;
```

| numero | sum(cout) |
|---|---|
| +33328778551 | 0.12 |

# Les fonctions de regroupement

## `where` vs `having`

- **`where`** est une qualification des lignes de la table source, donc s'applique **avant** regroupement

- **`having`** est une qualification portant sur le résultat d'une fonction de regroupement, donc s'applique **après** regroupement

# Ajout de données : commande `insert into`

**insert** permet d'ajouter des tuples dans une table

- Forme simple :

  **insert into *nomTable* values (*liste valeurs*)**

  ```
  insert into concerts values ('Zénith','Chose')
  ```

  Fournir **toutes** les valeurs, **dans l'ordre** des attributs.

- On peut spécifier les attributs :

  **insert into *nomTable* (*liste attrs*) values (*liste valeurs*)**

  ```
  insert into concerts (salle,artiste)
         values ('Zénith','Chose')
  insert into concerts (artiste,salle)
         values ('Chose','Zénith')
  ```

  **On peut omettre certains attributs (leur état sera NULL).**

# Ajout de données : `insert ... select`

## Les valeurs à ajouter peuvent résulter d'un select.

**insert into *nom de table* (*liste d'attributs*)**
**select *suite du select***

- Le select doit renvoyer autant de colonnes qu'il y a d'attributs dans la liste.
- Les types des données doivent être compatibles
- Cette forme de commande insert permet d'ajouter plusieurs tuples d'un coup (chaque ligne résultant du select)
- Permet de faire simplement une recopie des données d'une table.

# Suppression de données : commande `delete`

## supprimer une ou des lignes

**delete from *nom de table* where *qualification***

(la clause where est optionnelle).
Supprime de la table tous les tuples satisfaisant la qualification.
Exemples

- `delete from concerts where salle='Splendid'`

  détruit toutes les lignes correspondantes.

- `delete from concerts`

  vide la table...

**Cette opération est irréversible !**

# Modification de données : commande `update`

**update *nom de table* set *attribut* = *expression* where *qualification***

(la clause where est optionnelle).
Exemple : remplacement du coût TTC par le coût HT

```
update decompte set cout=cout/1.2
```

Là aussi, l'opération est irréversible.

```
update decompte set cout=0
```

# Les principaux types de données (variable selon SGBD)

| Numériques | |
|---|---|
| INTEGER | Entier signé, sur 32 bits |
| SMALLINT | Entier signé, sur 16 bits |
| BIGINT | Entier signé, sur 64 bits |
| NUMERIC(n,d) | *n* chiffres **dont** *d* après la virgule |
| REAL | Flottant |
| **Textuels** | |
| CHAR(n) | Chaîne de *n* caractères exactement. |
| VARCHAR(n) | Chaîne de *n* caractères maximum. |
| TEXT | Texte. |
| **Dates** | |
| DATE | date `A-M-J` |
| TIME | temps `H:MN:S` |
| TIMESTAMP | temps (estampille) |
| **Binaires** | |
| BYTEA | donnée binaire (NB :dénomination Postgresql) |

# Création de table

## commande `create table`

Syntaxe simplifiée :

`create table` ***nom de table*** `(` ***attribut type, ...*** `)`

## Exemple

```
create table etudiants
  ( nom varchar(20),
    prenom varchar(30),
    groupe smallint,
    naissance date
  )
```

# Destruction de table ou modification de structure

## commande `drop table`

destruction (irréversible) de la table.

ex :

```
drop table concerts
```

## commande `alter table`

modification de la structure de le table ou des contraintes associées à la table.

ex : ajout d'un attribut `email`

```
alter table artistes add email varchar(30)
```

Voir la documentation pour plus de détails.

# Clé primaire d'une relation (table)

## Définition

**Attribut ou ensemble d'attributs pouvant servir d'identifiant au tuple.**

- une clé primaire ne peut jamais être dans l'état NULL
- la table contient au maximum une occurrence de chaque valeur de clé primaire

## Choix de la clé primaire

Attention à la permanence dans le temps. La clé choisie doit rester unique pour tous les jeux de données.

### Ex : une table d'étudiants avec nom , prénom et nip

- le couple (nom, prénom) n'est pas une bonne idée (on peut un jour trouver des homonymes)
- le nip peut être une bonne clé primaire : il attribué à une et une seule personne au moment de l'inscription.

# Clé primaire d'une relation (table)

## Pourquoi une clé primaire ?

- une table devrait toujours comporter une clé primaire.
- le SGBD structure la représentation (interne) de la relation en l'indexant sur la clé primaire.
- amélioration de l'efficacité des requêtes.

## Et si aucun ensemble d'attributs ne peut constituer une clé primaire ?

**ajouter un attribut « artificiel » : une numérotation des tuples.**

Les SGBD proposent un mécanisme simple pour cela : un entier « auto-incrémenté ». Un numéro est attribué « automatiquement » à chaque insertion
$\rightarrow$ en postgresql : pseudo type `serial` (valeur de type `integer` générée de façon incrémentale). NB : syntaxe varie selon les SGBD

# Conception : définition de **contraintes** et intégrité

- une contrainte porte sur les **valeurs** contenues dans la base.
- le SGBD déclenche une **erreur** en cas de non respect.

- **contrainte au niveau tuple** :
  - **restriction du type d'un attribut** : limitation des valeurs autorisées (ex : valeur maxi, valeur mini ...).
  - **interdiction de l'état NULL**
- **contrainte au niveau relation : unicité**
  une valeur ne peut figurer plus d'une fois dans la relation
  NB : la définition d'une clé primaire entraîne une contrainte d'unicité.
- **contrainte entre relations : intégrité référentielle**
  seules les valeurs présente dans une table « de référence » sont autorisées.

# Conception : définition de **contraintes** et intégrité

## exemples de contraintes

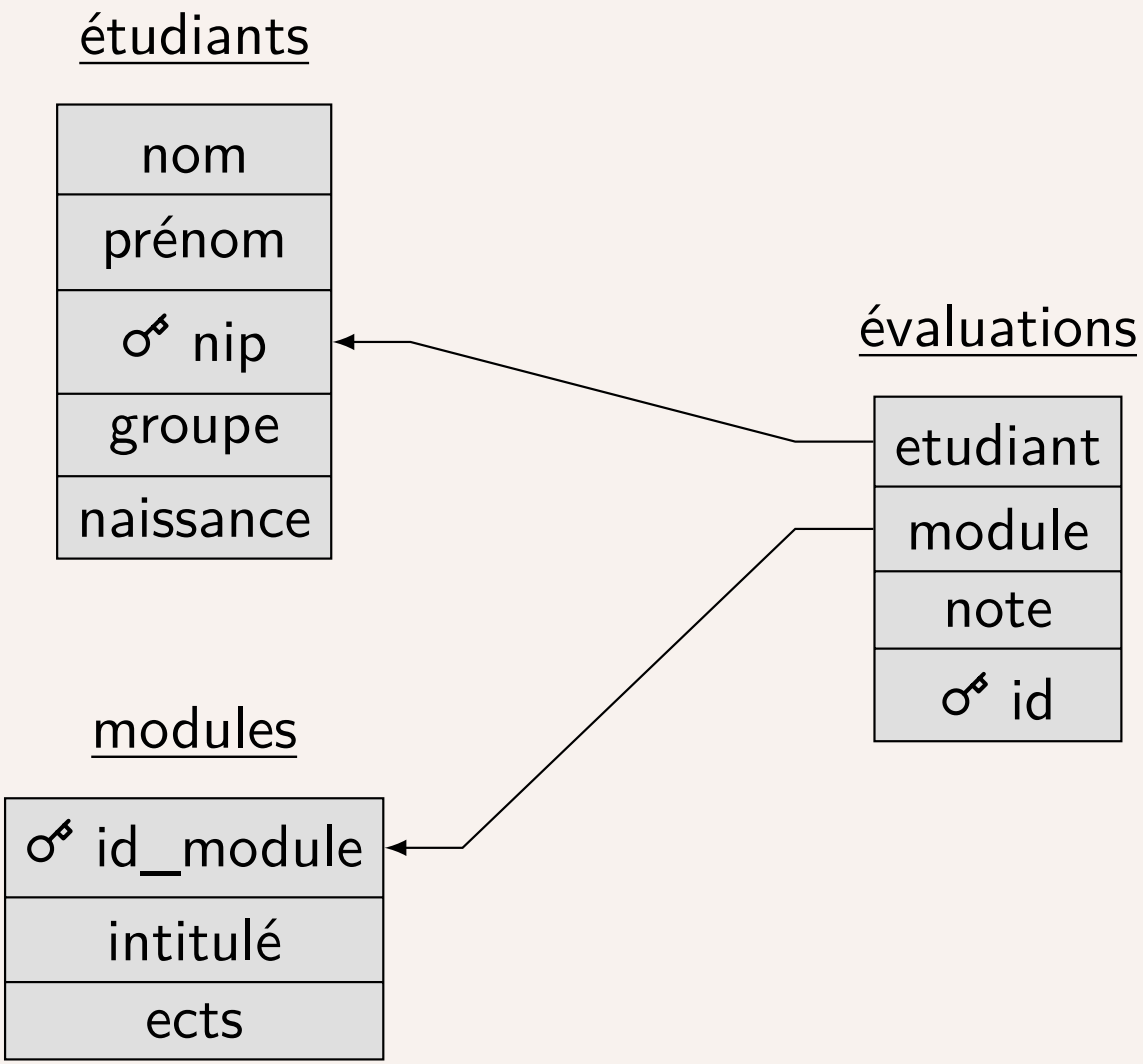
```
create table prixdate (
    ref character(5) NOT NULL,
    prix numeric(6,2) NOT NULL,
    date date NOT NULL,
    check (prix >0),
    check (extract(year from date)=2004)
);
```

- aucun attribut ne peut être indéfini (NULL)
- l'attribut `prix` doit être $> 0$.
- l'atrtribut `date` doit être en 2004.

# Intégrité référentielle

- une relation sert de référence à une entité : par exemple une relation `étudiants` sert de référence à l'entité « étudiant » et contient ses informations intrinsèques (nip, nom, prénom, date de naissance ...).
- cette relation possède une clé primaire qui permet d'identifier chaque entité « étudiant » (par exemple `nip`)
- tout étudiant qui apparaît dans la base **doit** faire partie de la relation `étudiants`.
- tout autre relation où intervient un étudiant doit le désigner par sa clé primaire dans la relation `étudiants` : elle **fait référence à** la relation `étudiants`.
- exemple : une relation `évaluations` (étudiant, module, note), l'attribut `étudiant` doit être le numéro `nip` et donc faire référence à l'attribut nip de la relation `étudiants`.

# Exemple


étudiants
nom
prénom
nip
groupe
naissance
évaluations
etudiant
module
note
id
modules
id_module
intitulé
ects

# Contraintes en SQL

## s'ajoutent à la définition de la table

- **`check ( ... )`**
  contraintes niveau tuple (sauf not null)
- **`unique ( ... )`**
  **`primary key ( ... )`**
  contraintes niveau relation
- **`foreign key ( ... ) references`** *nom_table*`(...)`
  intégrité référentielle

NB : **`not null`** est une mention qui suit le type de la donnée.

## on peut donner un nom à chaque contrainte

exemples

`constraint` *nom_de_contrainte* `check (...)`

`constraint` *nom_de_contrainte* `primary key (...)`

NB : postgresql attribue un nom par défaut à chaque contrainte.

# Contrainte simple en SQL

La clause **check** *condition* peut figurer dans la liste des définitions de colonne. Il peut y avoir plusieurs clauses check.
La clause **not null** peut compléter une définition de colonne.

```
create table etudiants
  ( nom varchar(20) not null,
    prenom varchar(30) not null,
    nip char(8),
    groupe smallint,
    naissance date,
    check (groupe between 1 and 4),
    check (naissance < '2004-1-1'),
    primary key (nip)
  )
```

# Clé primaire en SQL

Pour une clé primaire simple, on peut aussi la mentionner dans la définition de l'attribut

```
create table modules
  ( id_module varchar(20) primary key,
    intitule varchar(50) ,
    ects integer not null,
    check (ects >=0)
  )
```

# Clé étrangère en SQL

De la même manière une clé étrangère est définie par
**foreign key** (*liste d'attributs* )
**references** *nom de table*(*liste d'attributs* )

```
create table evaluations
 ( etudiant char(8),
   module varchar(12),
   note smallint,
   check (note between 0 and 20),
   id serial primary key,
   foreign key (etudiant)
      references etudiants(nip),
   foreign key (module)
      references modules(id_module),
 )
```

# Sous-requêtes

## Sous-requêtes à la place d'une table

- une nom de table (dans `FROM`) peut-être remplacé par une sous-requête
- la sous-requête doit figurer entre parenthèses
- le résultat doit recevoir un nom (par **as**)

# Sous-requêtes

## Exemple

```
select id_reg, nom
  from regions
   join (
     select id_reg from regions
     except
     select reg from vaccins
   ) as manquantes
  on regions.id_reg = manquantes.id_reg
```

# Sous-requêtes

**Exemple**

```
select avg(pop_totale)
  from  (
     select sum(pop) as pop_totale
     from vaccins
     group by reg
   ) as t
```

# Sous-requêtes

## Sous-requête à résultat scalaire

- requête qui renvoie **un seul** tuple **et un seul** attribut
- peut-être utilisée à la place d'une valeur simple
- la sous-requête doit figurer entre parenthèses

# Sous-requêtes

**Exemple**

```
select reg,  classe, 100*complet/pop as taux
  from  vaccins
  where taux > (
     select avg(100*complet/pop)
       from vaccin
  )
```