3. Bases de données 1

Pierre Karpman

Lycée Champollion MP Tronc Commun

Avec des emprunts à Nicolas Pécheux et Marie Durand

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Table des matières

Qu'est-ce qu'une base de données

Un ensemble organisé d'informations avec un objectif commun.

- Peu importe le support (papier, fichiers,…)
- Organisé et *structuré* ?
- Grande (très grande) quantité d'information ?

Où trouve-t'on des bases de données

- (Presque) chaque fois qu'on traite des données informatiquement
  - sites Web commercial, cartographie, données statistiques, gestion RH, gestion de stock…

Problématiques

- Pouvoir être interrogée et mise à jour (efficacement) par une communauté d'utilisateurs (du « monde entier » : *distincte du/de la créatrice de la base*)
- Garantir une cohérence des données

Cohérence des données

Problèmes potentiels : *cf.* TD d'introduction

- Alias (« Poche » v. « poche »)
- Erreurs de saisies (« Nietzsche » v. « Nietsche »)
- Ambiguïtés (« Brin » v. « D. Brin » ou « F. Brin »)

Comment éviter ces problèmes

- À la création de la base ?
- Quand on la modifie ?

Début de réponse

- En limitant au plus la redondance des informations
  - une seule façon d'écrire « poche »
  - (aussi utile pour la performance)
- En imposant des contraintes sur les actions possibles (appliquées par le logiciel de BD), cf. plus loin

Conception d'une base de données

- Tâche complexe et essentielle
- Conditionne la pertinence, robustesse et durabilité de la base
  - Cf. TD d'introduction

Tout comme la conception d'un bâtiment commence sur une table à dessin, celle d'une base de données passe par l'élaboration d'un *Modèle Conceptuel des Données*.

- Au programme : BD basées sur un *modèle relationnel* (« le plus courant »)
- Autre aspect au programme : modèle entités-associations (cf. plus loin)

Table des matières

Modèle relationnel

Modèle ensembliste basé sur la théorie (naïve) des ensembles et la logique des prédicats

Double objectif

- Garantir un haut degré d'indépendance avec la représentation interne
- Fournir une base solide pour traiter le problème de la cohérence et de la redondance des données
- Le plus répandu dans le domaine

Présentation

Relation

- Les objets et leurs associations sont représentés par la notion de *relations*
- Base de données = ensemble de relations
- Relation = table (tableau à deux dimensions)

Stockage des relations

- Relation : notion abstraite : ne préjuge pas de la façon dont les informations sont stockées dans la machine
- Le système de gestion de bases de données est libre d'utiliser n'importe quelle technique de stockage

Un exemple

clef primaire

personne :	**id**	nom	prenom	tel	← attributs
	(entier)	(texte)	(texte)	(texte)	← domaines
	75	Dupont	Pierre	0645456789	← tuple
	34	Durand	Sylvie	0612253694	

clef étrangère

classe :	**no**	nom	effectif	prof_princ
	(entier)	(texte)	(entier)	(entier)
	12	MP2I	44	75
	5	MP*	42	34

Attributs et domaines

Attribut

Un *attribut* (ou *colonne*) est un nom décrivant une information stockée dans une base.

- Âge, nom, numéro de sécurité sociale

Domaine

Le *domaine* (ou *type*) d'un attribut est l'ensemble des valeurs qu'il peut prendre

- Au programme : entier, « réel », chaîne de caractères

Schéma relationnel

Schéma

On appelle *schéma relationnel* un n-uplet $S = (A_1; A_2; \ldots; A_n)$ d'attributs deux à deux distincts, avec $n > 0$.
Si $A_1; A_2; \ldots; A_n$ de domaines $\mathcal{D}_1; \mathcal{D}_2; \ldots; \mathcal{D}_n$, on note

$$S = (A_1 : \mathcal{D}_1; A_2 : \mathcal{D}_2; \ldots; A_n : \mathcal{D}_n)$$

- **personne** : $(id : entier, nom : texte, prenom : texte, tel : texte)$

Tuples

Définir un objet de la relation revient à donner une valeur à chaque attribut

- C'est définir un n-uplet

$$t = (t_1; t_2; \ldots; t_n) \in \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$$

où t_i est la valeur de l'attribut A_i

Terminologie

- tuple ou
- enregistrement ou
- ligne

Exemples

- $(5, \text{“}MP*\text{”}, 42, 34)$ est un tuple de la relation **classe**
- 42 correspond à l'attribut *effectif*

Relations

Relation

On appelle *relation* associée à un schéma $S = (A_1 : \mathcal{D}_1; A_2 : \mathcal{D}_2; \ldots; A_n : \mathcal{D}_n)$ une partie finie $\mathcal{R}(S) \subseteq \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$, c'est-à-dire un ensemble fini de tuples.

Remarques

- Le cardinal d'une relation se note $|\mathcal{R}(S)|$ ou $\#\mathcal{R}(S)$ ou …
- Une relation peut très bien être vide
- Une relation étant un **ensemble** de tuples, un tuple ne peut y apparaître qu'une seule fois
- Il ne peut donc y avoir deux lignes identiques dans une table

Base de donnée

Une *base de données relationnelle* est un ensemble de relations.

Tables

Une relation est représentée sous la forme d'une table à deux dimensions dans laquelle les n attributs correspondent aux titres des n colonnes et les tuples aux lignes

classe :	**no**	nom	effectif	prof_princ
	12	MP2I	44	75
	5	MP*	42	34

Valeur d'un attribut

Notation

- La valeur de l'attribut A_i dans le tuple t se note $t.A_i$ ou $t[A_i]$
- On note $A \in S$ pour dire que A est l'un des attributs du schéma S
- Si $X = (A_{i_1}; A_{i_2}; \ldots; A_{i_p})$, on note $X \subseteq S$
- $t[X] = (t[A_{i_1}]; t[A_{i_2}]; \ldots t[A_{i_p}])$

Exemple

$t = (34, \text{“}Durand\text{”}, \text{“}Sylvie\text{”}, 0612253694) \in$ **personne**

- $t[nom] = \text{“}Durand\text{”}$
- $t[id, tel] = (34, 0612253694)$

Questions

- Est-ce que plusieurs personnes peuvent avoir le même prénom ?
- Une classe peut-elle avoir deux professeurs principaux ?

Clef

Principe

- Notion essentielle pour faire des rapprochements entre tables
- Une clef permet d'identifier à coup sûr un unique tuple
- Deux tuples distincts ne peuvent pas avoir la même valeur pour les attributs d'une clef

Clef candidate

Une *clef candidate* d'une relation $\mathcal{R}(S)$ est un ensemble minimal (au sens de l'inclusion) $K \subseteq S$ des attributs de la relation tel que $\forall t, t' \in \mathcal{R}(S), t[K] = t'[K] \Rightarrow t = t'$.

Proposition

Toute relation a au moins une clef candidate et peut en avoir plusieurs.

- Pourquoi ?

Clef primaire

Principe

- Parmi les clefs candidates, il convient d'en choisir une que l'on appelle *clef primaire*
- On la signale en soulignant ses attributs
- La clef primaire d'une relation est donc l'une de ses clefs candidates

Utilisation

- La donnée de $t[K]$ suffit à identifier t de façon certaine
- possible de faire référence à un tuple en le désignant par la valeur prise sur la clef
- On peut ainsi créer des liens entre plusieurs relations

Clef étrangère

Clef étrangère

Une *clef étrangère* dans une relation est formée d'un ou plusieurs attributs qui constituent une clef candidate (en général primaire) dans une autre relation.

Utilisation

- Une clef étrangère permet d'imposer une contrainte d'*intégrité référentielle* :
  - Un tuple ne peut exister que si sa clef étrangère est égale à la clef candidate en question d'un tuple dans l'autre relation
  - Au logiciel de gestion de la base de garantir cela !

Attention

Une clef étrangère d'une relation n'est pas nécessairement une clef candidate de cette même relation !

Utilisation d'une BD : SGBD

SGBD

- SGDB : *Système de gestion des bases de données*
- Idée : déléguer au SGDB la responsabilité de la gestion de la base
- Garantir la cohérence par des contraintes d'*intégrité*
  - Vérification des domaines
  - Contraintes des clefs primaires
  - Contraintes référentielles des clefs étrangères

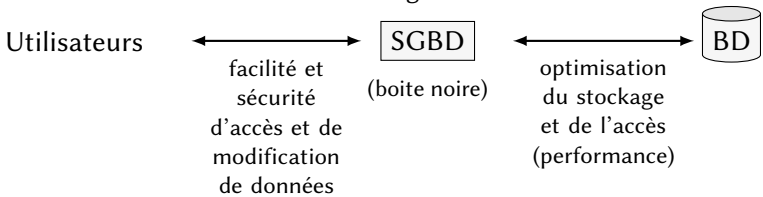


Table des matières

Modèle entité-association

Entité

Une *entité* est la représentation d'un objet (au sens large) dans le système d'information

- un livre, une personne, un théorème, une définition, une montagne, …

Association

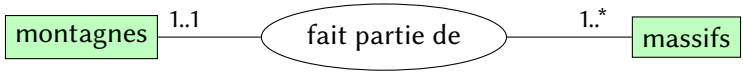
Une *association* est un lien entre différentes entités

- une montagne fait partie d'un massif
- une voie d'alpinisme se déroule sur une ou plusieurs montagnes
- une ou plusieurs personnes on ouvert une voie

Représentation graphique

Représentation sous forme de diagramme.

Situation : une montagne fait partie d'un massif ; un massif doit être constitué d'au moins une montagne, mais peut-être plus



Entités montagne, massif

Association fait partie de

Cardinalités ; types

Cardinalité

La *cardinalité* d'un lien est représentée par un couple de valeurs indiquant le minimum et le maximum de fois qu'une entité apparaît dans une association.

- La valeur min est généralement soit 0 soit 1
- On écrit * lorsqu'on ne connaît pas la valeur max.
- La cardinalité est notée $a..b$ avec a la valeur min et b la valeur max.
  - Cas particulier : la cardinalité 0..* peut être notée *, la cardinalité 1..1 peut être notée 1.

Type

Le *type* d'une association est déterminé par les bornes maximums des deux côtés de l'association.

- Il indique comment traduire l'association dans le modèle relationnel

Type `0..1-1`

Chaque entité à gauche correspond à au plus une entité à droite ; une entité à droite correspond à exactement une entité à gauche

- Association de type « injection »

Exemples

- **montagnes** *est point culminant de* `0..1-1` **massif**

Représentation

Par exemple :

- **massifs** : (id : entier ; nom : texte; pointCulminant : entier) → pointCulminant est une clef étrangère pour **montagnes**, et est aussi une clef candidate pour **massifs**

Type 1 – 1

Chaque entité à gauche correspond à exactement une entité à droite, et vice-versa

- Association de type « bijection »
- Contrainte forte (pas si courant ?)

Exemples

- <Ici, votre bijection préférée>

Représentation

Avec une table pour chaque entité et en référençant mutuellement les clefs primaires, ou tout simplement en fusionnant (par *jointure*) les tables (dans ce cas : pas de redondance)

Type 1-*

Chaque entité à gauche correspond à exactement une entité à droite ; une entité à droite correspond à un nombre arbitraire d'entités à gauche

- Association de type « hiérarchique »

Exemples

- **montagnes** *appartient à* `1-*` **massifs**
- **exemplaire** *correspond à* `1-*` **livre**

Représentation

Par exemple par deux relations de schémas :

- **massif** : (id : entier; nom : texte)
- **montagne** : (id : entier; nom : texte; idMassif : entier) → idMassif est une clef étrangère

Type *-*

Chaque entité à gauche correspond à un nombre arbitraire d'entités à droite, et vice-versa

- Association de type « réflexif »

Exemples

- **voie** *parcourt* *-* **montagne** → une voie d'alpinisme peut parcourir une à plusieurs montagnes, qui peuvent être parcourues par zéro à plusieurs voies
- **personne** *est autrice de* *-* **livre**

Représentation

Par exemple par une table *de référence croisée* représentant l'association, dont les attributs sont (comme clefs étrangères) les clefs primaires des entités, et la seule clef candidate l'ensemble des deux attributs :

- **livres** : (idLi : entier, …)
- **personnes** : (idAu : entier, …)
- **autriceDe** : (idAu : entier, idLi : entier)

Type `*-*` (cont.) ; bilan entité-association

Représentation (cont.)

On peut conceptualiser la table de référence croisée comme représentant deux associations de type `1-*` (ou `0..1-*`):

- **personnes** *apparaît comme* `*-1` **autriceDe**
- **livre** *référencé dans* `1-*` **autriceDe**

Bilan

Le modèle entité-association peut guider la création de tables, et l'identification de clefs primaires, étrangères :

- `1-1` : fusion de tables possible
- `1-*` : utilisation de plusieurs tables, avec clef étrangère
- `*-*` : création d'une table de référence croisée spécifique pour représenter l'association

Table des matières

SQL : Structured Query Language

- Un langage normalisé **très** répandu dans les SGBDs (mais avec autant de variantes…)
- Pour BD relationnelles

Langage de requête

- Langage *déclaratif* : l'utilisateur/utilisatrice spécifie « simplement » le résultat attendu, et non le moyen d'y parvenir
- C'est au SGBD de déterminer comment répondre efficacement à la requête

Conception logique/ensembliste

- Adapté au modèle relationnel
- Utilisation de *prédicats*
  - pour seulement garder les données souhaitées
- Utilisation d'opérations ensemblistes (union, intersection, produit cartésien…)
  - pour extraire des informations réparties dans plusieurs tables

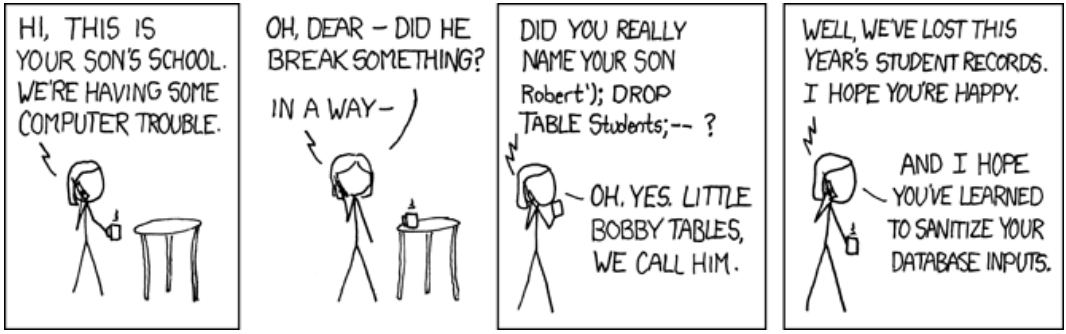
En ligne

https://sqliteonline.com/

SQL : pour quoi faire ?

- Créer des tables
- Modifier des tables
- Interroger des tables ← seule partie au programme
- Supprimer des tables

Suppression d'une table



https://xkcd.com/327/

Interroger une base de données

Requête

Une *requête* est une combinaison d'opérations portant sur des tables (relations) et dont le résultat *est lui-même une table*

- La table résultat a une existence éphémère, le temps de la requête seulement
- Mais c'est une table ! Elle peut donc faire l'objet de requêtes !
  - Notion de requêtes imbriquées (pas pour aujourd'hui)

Requête SQL

Une requête se présente généralement sous la forme :

```
-- Commentaire
SELECT <attributs> FROM <tables>
WHERE <conditions>;
```

- `SELECT` permet de spécifier les attributs (*projection*)
- `FROM` permet de spécifier les tables
- `WHERE` permet de spécifier des conditions que doivent respecter les lignes sélectionnées (*sélection* ; correspond à un prédicat)
- Mots-clés (généralement) en majuscules (convention, qu'on peut aussi choisir d'ignorer)
- Une requête se termine toujours par un point-virgule

DISTINCT et ALL

Lorsque le SGDB construit la réponse d'une requête :

- Il rapatrie toutes les lignes qui satisfont la requête
- Généralement dans l'ordre où il les trouve (par ex., pas forcément trié)
- Même si ces dernières sont en double
  - Autrement dit, la table renvoyée *n'est pas forcément une relation*

Supprimer les doublons est une opération qui peut être coûteuse et n'est pas effectuée par défaut → utilisation de `DISTINCT` si désiré

```
SELECT DISTINCT <attributs> FROM <tables>
WHERE <conditions>;
```

Sinon, le mot-clé par défaut est `ALL`, qui est donc facultatif

Renommage

- L'opérateur `AS` permet de renommer une table ou une colonne
- Pour renommer une table on écrit `<nom_table> AS <nouveau_nom>`
  - Le mot-clé `AS` est facultatif
- Même syntaxe pour renommer une colonne/attribut

```
SELECT id, nom FROM montagnes AS mnt;
SELECT id, nom FROM montagnes mnt;
SELECT nom, altitude AS alt FROM montagnes;
```

- Utile pour simplifier les noms de table/colonnes trop longs dans de longues requêtes; simplifie la gestion des alias

Visualisation d'une table ; projection

Toute la table

On ne peut pas accéder directement au contenu d'une table, mais on peut faire une requête pour renvoyer tous les tuples

```
SELECT * FROM montagnes;
```

- Le caractère `*` : tous les attributs
- Peut-on avoir des doublons dans ce cas ?

Projection

Pour garder seulement certains attributs, il suffit de les mentionner explicitement

```
SELECT nom, massif FROM montagnes
SELECT DISTINCT massif FROM montagnes
```

- `DISTINCT` non nécessaire si une clef candidate est incluse dans les attributs demandés, mais possiblement utile sinon

Sélection : WHERE

Pour sélectionner les tuples satisfaisant un prédicat ϕ :

```
SELECT * FROM t WHERE phi
```

- `*`, ou en couplant avec une projection !

Pour sélectionner les montagnes des Écrins ; — pas des Écrins; — d'altitude supérieure ou égale à 4000m :

```
SELECT * FROM montagnes WHERE massif = 'Écrins';
SELECT * FROM montagnes WHERE massif <> 'Écrins';
SELECT * FROM montagnes WHERE altitude >= 4000
```

- Utilisation de « quotes » simples pour délimiter les chaînes de caractères
- (On suppose pour l'instant, pour simplifier que les massifs sont stockés de façon redondante dans la table **montagnes**, mais cela ne devrait pas être le cas puisqu'on a une association de type `1-*` !)

Sélection (cont.)

On peut combiner logiquement des conditions avec AND, OR, NOT

```
SELECT * FROM montagnes WHERE massif = 'Belledonne' OR
        (massif = 'Chartreuse' AND altitude >= 1600);
SELECT * FROM montagnes
WHERE NOT (massif = 'Mont Blanc' OR massif = 'Mercantour');
```

Exercice rapide

Écrire une requête permettant de trouver les noms de montagnes de moins de 2000m se trouvant dans le massif du Dévoluy ou du Vercors, et toutes les montagnes du massif du Taillefer

Exercice rapide (correction)

Écrivez une requête permettant de trouver les noms montagnes de moins de 2000m se trouvant dans le massif du Dévoluy ou du Vercors, et toutes les montagnes du massif du Taillefer

```
SELECT nom FROM montagnes WHERE
(altitude <= 2000 AND (massif = 'Dévoluy' OR massif = 'Vercors')) OR
(massif = 'Taillefer');
```

Opérations ensemblistes

Permettent de combiner plusieurs requêtes (sur une ou plusieurs tables)

```
SELECT ... UNION SELECT ...;
SELECT ... INTERSECT SELECT ...;
SELECT ... EXCEPT SELECT ...;
```

- Attention, il ne faut pas mettre de parenthèses

Logique ensembliste

Le comportement par défaut de ces opérateurs ensemblistes est `DISTINCT`. Pour conserver les doublons il faut explicitement utiliser `ALL`

Exemple UNION

Pour avoir *dans une colonne* la liste des noms de massifs et de montagnes

```
SELECT nom AS nom-massif FROM montagnes UNION
SELECT massif FROM montagnes
```

- Attention : on doit sélectionner le même nombre d'attributs dans les deux requêtes
- C'est le nom des colonnes de la première requête qui seront ceux de la table résultat :

```
nom-massif
----------
...
Chamechaude
Chapotet
...
Dévoluy
Dolomites
...
```

Exemple INTERSECT

Pour avoir *dans une colonne* la liste des noms de montagnes qui sont aussi un nom de massif

```
SELECT nom AS montagne-massif FROM montagnes INTERSECT
SELECT massif FROM montagnes
```

- Même remarques que pour UNION

```
montagne-massif
----------
...
Grand Paradis
Mont Blanc
Taillefer
...
```

Exemple EXCEPT

Pour avoir *dans une colonne* la liste des noms de montagnes qui ne sont pas aussi un nom de massif

```
SELECT nom  FROM montagnes EXCEPT
SELECT massif FROM montagnes
```

- Attention : Rappel : la différence ensembliste n'est pas symétrique !

Produit cartésien

Le produit cartésien de deux tables (pas nécessairement distinctes) s'obtient simplement en les séparant par une virgule dans la clause `FROM`

```
SELECT * FROM montagnes, voies;
SELECT * FROM montagnes, montagnes AS mtn;
```

- Coûteux ! Si t_1, t_2 tuples dans les deux tables, $t_1 \times t_2$ dans la table résultat !
- Généralement peu pertinent : quel sens cela a-t'il d'accoler le nom d'une voie à celui d'une montagne qu'elle ne parcourt pas ?
  - À combiner avec une sélection $\rightarrow$ notion de jointure

Jointure

Une jointure est très souvent utilisée pour reconstituer une information stockée sur plusieurs tables en faisant correspondre les clef étrangères aux clefs primaires associées. Supposons (comme cela devrait être le cas !!) que les massifs sont en fait stockés dans une table séparée avec un identifiant numérique comme clef primaire :

```
-- On préfixe les attributs par le nom de la table pour éviter
-- les ambiguïtés
SELECT montagnes.nom, massif.nom FROM montagnes, massifs
WHERE montagnes.idMassif = massif.id;
```

Alternative : syntaxe `JOIN ... ON` dédiée :

```
SELECT montagnes.nom, massif.nom FROM montagnes
JOIN massif on montagnes.idMassif = massif.id;
```

Jointure ou produit cartésien ?

- Jointure : opération naturelle pour recoller deux tables. Autant que ce soit explicite !
- Distinction entre ce qui relève de la sélection (dépendant de la recherche) et de ce qui relève de la jointure (dépendant de la structure)
- Permet de cloisonner les conditions de jointures entre chaque couple de tables (en cas de jointures multiples)
- A priori un produit cartésien est plus coûteux à calculer : il faut considérer toutes les associations possibles. Les gestionnaires de bases disposent d'algorithmes efficaces et optimisés pour effectuer des jointures

Utilisez la syntaxe avec `JOIN ... ON` dès que l'on réalise une jointure logique entre deux tables.

Quelques exemples

Avec des tables :

- **montagnes** : (id : entier ; nom : texte ; …)
- **voies** : (id : entier ; nom : texte ; …)
- **parcourt** : (idVoie : entier ; idMtn : entier)

Jointure multiple : les noms de voies passant par la Meije

```
SELECT voies.nom FROM parcourt p
JOIN voies ON p.idVoie = voies.id
JOIN montagnes ON p.idMtn = montagnes.id
WHERE montagnes.nom = 'La Meije';
```

⇝

```
Le Z
Directe Face N
Traversée
Les grimpeurs se cachent pour ouvrir
```

Quelques ~~exemples~~ exercices

Avec aussi une table :

- **massifs** : (id : entier ; nom : texte ; pointCulminant : entier)

Écrivez une requête permettant de trouver les noms des voies se déroulant dans le massif des Écrins

Quelques ~~exemples~~ exercices (correction)

Avec aussi une table :

- **massif** : (*id* : entier ; nom : texte ; pointCulminant : entier)

Écrivez une requête permettant de trouver les noms des voies se déroulant dans le massif des Écrins

```
SELECT voies.nom FROM parcourt p
JOIN voies ON p.idVoie = voies.id
JOIN montagnes mtn ON p.idMtn = mtn.id
JOIN massifs msf ON mtn.idMassif = msf.id
WHERE msf.nom = 'Écrins';
```

Quelques exercices 2

Écrivez une requête permettant de trouver les noms des montagnes parcourues par une voie d'alpinisme

Quelques exercices 2 (correction)

Écrivez une requête permettant de trouver les noms (uniques) des montagnes parcourues par une voie d'alpinisme

```
SELECT DISTINCT mtn.nom FROM montagnes mtn
JOIN parcourt p ON p.idMtn = mtn.id;
```

ORDER BY

`ORDER BY` permet de trier l'ordre des lignes de la table construite par la requête

- Par ordre croissant : `ASC` (facultatif)
- Par ordre décroissant : `DESC`
- La clause `ORDER BY` vient après la clause `WHERE` ; syntaxe complète (pour les clauses vues jusqu'à présent) :

```
SELECT ... FROM ... JOIN . ON ... WHERE ...
ORDER BY {attributs} ... DESC;
```

Quel ordre ?

- Ordre naturel pour les domaines numériques
- Ordre lexicographique pour les chaînes de caractère
- Ordre lexicographique sur les attributs, si plusieurs attributs

Exercice

Écrivez une requête listant (dans cet ordre) les noms de massifs, noms de leur point culminant, altitude du point culminant, trié par altitude de point culminant décroissante. Un résultat possible est par exemple :

```
Écrins     | Barre des Écrins         | 4102
Vanoise    | Grande Casse             | 3855
Belledonne | Grand Pic de Belledonne | 2977
Vosges     | Grand Ballon             | 1424
```

Exercice (correction)

Écrivez une requête listant (dans cet ordre) les noms de massifs, noms de leur point culminant, altitude du point culminant, trié par altitude de point culminant décroissante. Un résultat possible est par exemple :

```
Écrins     | Barre des Écrins         | 4102
Vanoise    | Grande Casse             | 3855
Belledonne | Grand Pic de Belledonne | 2977
Vosges     | Grand Ballon             | 1424
```

```sql
SELECT msf.nom, mtn.nom, mtn.altitude FROM massifs AS msf
JOIN montagnes AS mtn ON msf.pointCulminant = mtn.id
ORDER BY mtn.altitude DESC;
```

LIMIT

Si l'on souhaite garder seulement certains tuples du résultat d'une requête :
`...` `LIMIT` n avec *n* le nombre de tuples à garder

- A surtout du sens pour des requêtes ordonnées avec `ORDER BY`

```
SELECT msf.nom, mtn.nom, mtn.altitude FROM massifs AS msf
JOIN montagnes AS mtn ON msf.pointCulminant = mtn.id
ORDER BY mtn.altitude DESC LIMIT 2;
```

donne :

```
Écrins      | Barre des Écrins        | 4102
Vanoise     | Grande Casse            | 3855
```

OFFSET

En conjonction avec `LIMIT` : `... LIMIT n OFFSET m` si l'on veut écarter les m premiers résultats :

```
SELECT msf.nom, mtn.nom, mtn.altitude FROM massifs AS msf
JOIN montagnes AS mtn ON msf.pointCulminant = mtn.id
ORDER BY mtn.altitude DESC LIMIT 1 OFFSET 2;
```

donne :

```
Belledonne | Grand Pic de Belledonne | 2977
```

Exercice : vers un MAX

Écrivez une requête donnant l'altitude de la plus haute montagne connue

Exercice : vers un MAX

Écrivez une requête donnant l'altitude de la plus haute montagne connue

```
SELECT altitude FROM montagnes ORDER BY altitude DESC LIMIT 1;
```

Fonctions d'agrégation 1 : MIN, MAX

La requête précédente peut se réécrire plus simplement (et plus efficacement) comme :

```
SELECT MAX(altitude) FROM montagnes;
```

- MAX est une fonction d'agrégation qui renvoie *une* ligne avec un attribut de valeur maximum pour son attribut argument

Attention

On ne **peut pas** sélectionner d'autre attribut en plus de la fonction d'agrégation (même si c'est accepté en SQLite…). Par exemple :

```
-- INTERDIT
SELECT nom, MIN(altitude) FROM montagnes; -- INTERDIT
-- INTERDIT
```

Comment faire pour trouver le(s) nom(s) de montagne(s) d'altitude minimum ?

Agrégation : exemple de requête imbriquée

Comment faire pour trouver le(s) nom(s) de montagne(s) d'altitude minimum ?

- S'il n'y en a qu'une : (à nouveau) `ORDER BY` + `LIMIT`
- Sinon, on peut utiliser une *requête imbriquée* :

```
-- version 1
SELECT nom, altitude alt FROM montagnes
WHERE alt = (SELECT MIN(altitude) FROM montagnes);
-- version 2
SELECT nom, altitude alt FROM montagnes
JOIN (SELECT MIN(altitude) ma FROM montagnes)
ON alt = ma;
```

donne :

```
Hartmannswillerkopf | 957
Mont Ascutney       | 957
```

Principe d'une requête imbriquée (ou sous-requête)

Requête imbriquée

- Une requête renvoie ses résultats comme une table
- On peut utiliser ces résultats comme (n'importe quelle autre) table dans un `FROM` d'une requête
- Si la table renvoyée contient une unique valeur, on peut l'utiliser dans un `WHERE`
  - et en fait aussi dans le cas de plusieurs valeurs, mais pas au programme

Exercice

Écrivez une requête utilisant des requêtes imbriquées pour donner les noms des montagnes atteignant les altitudes maximales et minimales, par ordre d'altitude décroissant

Exercice (correction)

```
SELECT nom, altitude alt FROM montagnes
WHERE (alt = (SELECT MIN(altitude) FROM montagnes)) OR
      (alt = (SELECT MAX(altitude) FROM montagnes))
ORDER BY alt DESC;
```

ce qui donne :

```
Barre des Écrins     | 4102
Hartmannswillerkopf | 957
Mont Ascutney        | 957
```

Fonctions d'agrégation 2 : SUM, AVG

SUM, AVG

- Calculent la somme, moyenne des valeurs prises par un attribut

Exemple SUM

```
SELECT SUM(altitude) FROM montagnes;
```

Exemple AVG

```
SELECT AVG(altitude) FROM montagnes mtn
JOIN massifs msf ON mtn.idMassif = msf.id
WHERE msf.nom = 'Belledonne';
```

- Que fait cette requête ?
- Comment faire pour obtenir la même chose pour tous les massifs, *en une requête* ?
  - Réponse bientôt

Exercice : altitude moyenne des montagnes parcourues par des voies d'alpinisme

Écrivez une requête donnant l'altitude moyenne des montagnes parcourues par au moins une voie d'alpinisme

- Indice : utilisez une requête imbriquée dans un `FROM`

Exercice : altitude moyenne des montagnes parcourues par des voies d'alpinisme

Écrivez une requête donnant l'altitude moyenne des montagnes parcourues par au moins une voie d'alpinisme

- Indice : utilisez une requête imbriquée dans un `FROM`

```sql
SELECT AVG(alt) FROM (
  SELECT DISTINCT id, altitude alt
      -- DISTINCT et id : pas de doublons
      -- alt : doit être sélectionné pour être disponible
      -- dans la requête *englobante*
  FROM montagnes mtn
  JOIN parcourt p ON mtn.id = p.idMtn
  );
```

Fonctions d'agrégation 3 : COUNT

COUNT

- Compte le nombre (éventuellement `DISTINCT`) d'enregistrements (lignes) d'une table (satisfaisant une condition)

```sql
-- Nombre d'enregistrement dans montagnes
SELECT COUNT(*) FROM montagnes; -- ou n'importe quel attribut

-- Nombre de montagnes du massif 3
SELECT COUNT(*) FROM montagnes WHERE idMassif = 3;

-- Nombre de massifs pour lesquels au moins une montagne est connue
SELECT COUNT(DISTINCT idMassif) FROM montagnes;
```

Exercice : Nombre de montagnes de plus de 4000m

Fonctions d'agrégation 3 : COUNT

COUNT

- Compte le nombre (éventuellement `DISTINCT`) d'enregistrements (lignes) d'une table (satisfaisant une condition)

```sql
-- Nombre d'enregistrement dans montagnes
SELECT COUNT(*) FROM montagnes; -- ou n'importe quel attribut

-- Nombre de montagnes du massif 3
SELECT COUNT(*) FROM montagnes WHERE idMassif = 3;

-- Nombre de massifs pour lesquels au moins une montagne est connue
SELECT COUNT(DISTINCT idMassif) FROM montagnes;
```

Exercice : Nombre de montagnes de plus de 4000m

```sql
SELECT COUNT(*) FROM montagnes WHERE altitude >= 4000;
```

Exercice : nombre de voies parcourant La Meije

Écrivez une requête donnant le nombre de voies parcourant La Meije

- Indice : utilisez une requête imbriquée dans un FROM
  - Écrivez d'abord la requête imbriquée, puis la requête avec COUNT

Exercice : nombre de voies parcourant La Meije

Écrivez une requête donnant le nombre de voies parcourant La Meije

- Indice : utilisez une requête imbriquée dans un `FROM`
  - Écrivez d'abord la requête imbriquée, puis la requête avec `COUNT`

```
SELECT COUNT(*) FROM
  ( SELECT * FROM voies v
    JOIN parcourt p ON v.id = p.idVoie
    JOIN montagnes mtn ON mtn.id = p.idMtn
    WHERE mtn.nom = 'La Meije'
  );
```

Regroupement

Un *groupe* est un sous-ensemble de lignes d'une table partageant la même valeur pour un ou plusieurs attributs donnés

Principe du regroupement

- Regrouper suivant un ensemble d'attributs : créer une ligne pour chaque groupe (associé à ces attributs)
- Projeter (avec `SELECT`) sur des attributs *qui ne sont pas constants pour tout un groupe* **n'a pas de sens**
  - Raison pour laquelle `SELECT nom, MIN(altitude) FROM...` n'est pas valide : `MIN` crée implicitement un groupe : celui de toutes les montagnes d'altitude minimum, et leur nom n'est pas forcément le même
- Utilisation majeure : en combinaison avec les fonctions d'agrégation
  - Une fonction d'agrégation **renverra un résultat par groupe**

Regroupement : avec GROUP BY

Syntaxe

```
SELECT ... FROM ... WHERE ... GROUP BY {attributs};
```

Exemple : altitude moyenne *par massif*

```
SELECT msf.nom, AVG(altitude)
  FROM montagnes mtn
  JOIN massifs msf ON mtn.idMassif = msf.id
  GROUP BY msf.nom;
```

Groupes suivant plusieurs attributs

Les groupes sont constitués pour tous les ensembles de tuples égaux sur ces attributs

Exercice 1

Nombre de montagnes par massif

Écrivez une requête qui indique pour chaque massif le nombre de montagnes s'y trouvant, donnant par exemple :

```
Écrins           | 3
Vanoise          | 2
Belledonne       | 3
Vosges           | 2
Montagnes Vertes | 1
```

Exercice 1

Nombre de montagnes par massif

Écrivez une requête qui indique pour chaque massif le nombre de montagnes s'y trouvant, donnant par exemple :

```
Écrins           | 3
Vanoise          | 2
Belledonne       | 3
Vosges           | 2
Montagnes Vertes | 1
```

```sql
SELECT msf.nom, COUNT(*) nombre_de_montagnes
  FROM montagnes mtn
  JOIN massifs msf ON mtn.idMassif = msf.id
  GROUP BY mtn.idMassif;
```

Exercice 2

Nombre de voies par montagne

Écrivez une requête qui indique pour chaque montagne parcourue par des voies d'alpinisme le nombre de voies la parcourant, donnant par exemple :

```
Bosse 3343                | 1
La Meije                  | 4
Pointe de la Grande Glière | 1
```

Exercice 2

Nombre de voies par montagne

Écrivez une requête qui indique pour chaque montagne parcourue par des voies d'alpinisme le nombre de voies la parcourant, donnant par exemple :

```
Bosse 3343                  | 1
La Meije                    | 4
Pointe de la Grande Glière  | 1
```

```sql
SELECT mtn.nom, COUNT(*) as nombre_de_voies
  FROM voies v
  JOIN parcourt p on v.id = p.idVoie
  JOIN montagnes mtn ON mtn.id = p.idMtn
  GROUP BY mtn.nom;
```

Filtrage des groupes : HAVING

La clause HAVING permet de ne garder que des groupes vérifiant certaines conditions

Syntaxe

```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING *conditions sur les groupes*;
```

- Les conditions HAVING portent sur les « résultats » des groupes
  - Attributs présents dans la clause GROUP BY, ou dont les valeurs sont identiques pour tous les tuples **de chaque groupe**
  - Résultats de fonction d'agrégation appliquées aux groupes
- Différent des conditions WHERE, qui portent sur la table **avant** construction des groupes

Exemples de (non) filtrage

`WHERE` ou `HAVING` pour renvoyer la liste des altitudes moyennes par massifs :

- d'altitude moyenne supérieure à 2000m ;
- dont le point culminant est supérieur à 3000m ?

Exemples de (non) filtrage

`WHERE` ou `HAVING` pour renvoyer la liste des altitudes moyennes par massifs :

- d'altitude moyenne supérieure à 2000m ;
- dont le point culminant est supérieur à 3000m ?

```
-- Altitude moyenne >= 2000
SELECT msf.nom, AVG(mtn.altitude) altitude_moyenne
  FROM montagnes mtn JOIN massifs msf ON mtn.idMassif = msf.id
  GROUP BY msf.nom HAVING altitude_moyenne >= 2000;

-- Altitude moyenne de massifs de point culminant >= 3000
SELECT msf.nom, AVG(mtn.altitude) altitude_moyenne
  FROM montagnes mtn
  JOIN massifs msf ON mtn.idMassif = msf.id
  JOIN montagnes mtn2 ON msf.pointCulminant = mtn2.id
  WHERE mtn2.altitude >= 3000 -- ou HAVING ci-dessous
  GROUP BY msf.nom; -- HAVING mtn2.altitude >= 3000
```

Exercice : altitude moyenne de massifs de plus d'une montagne

Écrivez une requête donnant son altitude moyenne pour chaque massif, à condition qu'on lui connaisse (strictement) plus d'une montagne

- Indice : comment fait-on pour compter un nombre de lignes ? Sur quoi ces comptes portent-ils en cas de regroupement ?

Exercice : altitude moyenne de massifs de plus d'une montagne

Écrivez une requête donnant son altitude moyenne pour chaque massif, à condition qu'on lui connaisse (strictement) plus d'une montagne

- Indice : comment fait-on pour compter un nombre de lignes ? Sur quoi ces comptes portent-ils en cas de regroupement ?

```sql
SELECT msf.nom, AVG(mtn.altitude) altitude_moyenne
  FROM montagnes mtn JOIN massifs msf ON mtn.idMassif = msf.id
  GROUP BY msf.nom HAVING COUNT(*) > 1;
```

Résumé