Informatique tronc commun

Bases de données et SQL

Poly de cours à compléter — version enseignant

1 Bases de données et SQL

1.1 Contexte

- **Base de données relationnelle** :
  > une façon d'organiser des données en *tables*.
- **Langage de requête** :
  > un langage permettant d'*interroger* une base de données, c'est-à-dire d'en extraire des informations sans dire *comment* les calculer. On dit que le langage est **déclaratif** : on décrit le résultat voulu, le logiciel (le *système de gestion de base de données*, SGBD) choisit la manière de l'obtenir.
- **SQL** (*Structured Query Language*) :
  > l'exemple historique de langage de requête, encore utilisé partout aujourd'hui. Nous utiliserons le SGBD SQLite.

1.2 Vocabulaire

Le vocabulaire des bases de données est le suivant:

- **Table** (ou relation)
  > un ensemble de lignes, chacune décrivant un objet ou un événement, et chaque ligne étant décrite par un certain nombre de colonnes.
- **Colonne** (ou attribut),
  > à laquelle est associé un *domaine*, c'est-à-dire un type : entier, flottant, chaîne de caractères
- **Ligne** (ou enregistrement),
  > un ensemble de valeurs, une par colonne, décrivant un objet ou un événement.

Schématiquement on peut représenter une table par … un tableau:

communes

codeinsee	nom	region	population	superficie
78646	Versailles	Île-de-France	84 095	26,0
75056	Paris	Île-de-France	2 103 778	105,0
78686	Viroflay	Île-de-France	17 237	3,0
43157	Le Puy-en-Velay	Auvergne-Rhône-Alpes	18 540	17,0
69123	Lyon	Auvergne-Rhône-Alpes	519 127	48,0
13055	Marseille	Provence-Alpes-Côte d'Azur	886 040	238,0

Extrait de la table `communes` (superficies en km²) utilisée dans tout le poly.

et par exemple:

- la table est `communes` ;
- les colonnes sont `codeinsee`, `nom`, `region`, `population`, `superficie` de types `entier`, `chaîne de caractères`, `chaîne de caractères`, `entier`, `flottant`
- les lignes sont `(78686, Viroflay, Île-de-France, 17237, 3.0)`, `(75056, Paris, Île-de-France, 2103778, 105.0)`, `(13055, Marseille, Provence-Alpes-Côte d'Azur, 886040, 238.0)`, etc.

Une colonne

codeinsee	nom	region	population	superficie
78646	Versailles	Île-de-France	84 095	26,0
75056	Paris	Île-de-France	2 103 778	105,0
78686	Viroflay	Île-de-France	17 237	3,0

La colonne (ou attribut) `region`

Une ligne

codeinsee	nom	region	population	superficie
78646	Versailles	Île-de-France	84 095	26,0
75056	Paris	Île-de-France	2 103 778	105,0
78686	Viroflay	Île-de-France	17 237	3,0

La ligne (ou enregistrement) décrivant Paris

On peut formaliser mathématiquement ces notions:

- **Domaines** : on se donne des ensembles $A_1, ..., A_n$,
- **Colonne** : on appelle colonne un indice $i \in [\![1, n]\!]$;
- **Ligne** : on appelle ligne un n-uplet $(a_1, ..., a_n) \in A_1 \times ... \times A_n$;
- **Table** : on appelle table un ensemble de lignes c'est-à-dire $R \subset A_1 \times ... \times A_n$.

Ce formalisme n'est **pas officiellement au programme** mais peut vous aider à comprendre les notions de *projection*, *sélection*, *opérations ensemblistes*, *agrégation* et *jointure*.

Notation pratique: On décrit une table par son nom et la liste de ses colonnes : `table(attribut1, ..., attributn)`. Certains énoncés de concours utilisent cette notation, en soulignant parfois la clé primaire (voir la partie 3), ils donnent parfois le schéma seul, parfois avec des exemples graphiques de tables.

Par exemple on pourrait décrire la table précédente par:

```
communes(codeinsee, nom, region, population, superficie)
```

Si aucune représentation graphique n'est donnée, il ne faut pas hésiter à en faire une sur le **brouillon** si cela vous aide.

1.3 Aux concours

La syntaxe SQL au programme est un sous-ensemble de la syntaxe « usuelle » (en fait cette dernière dépend du moteur SQL utilisé). Il s'agit essentiellement d'un jeu de *création de requête*: on vous donne un schéma de tables, un résultat attendu et vous devez écrire une requête SQL qui produit ce résultat.

Ces questions de SQL sont **indépendantes** du reste du sujet (et le plus souvent entre elles) et une pratique régulière (~15min par semaine) permet de sécuriser sans difficulté les points associés.

En pratique:

- à **X-ENS**: depuis 2015, 5 sujets ont contenu du SQL, sous la forme de deux ou trois questions indépendantes du reste du sujet (c'était par exemple les **Q15** et **Q16** du sujet 2026), avec une exception: le sujet 2018 entièrement sur les bases de données,
- aux **Mines**: depuis 2015, chaque année deux-trois questions de SQL complètement indépendantes du reste du sujet (cf livret),
- à **Centrale**: pas d'ITC à l'écrit et pas de SQL à l'oral.

Par exemple sur la page suivante voici un extrait du sujet d'ITC X-ENS de 2019:

Partie V. Gestion des scores en SQL

On souhaite utiliser une base de données pour stocker les résultats obtenus par une communauté de joueurs. On suppose que l'on dispose d'une base de données comportant les tables `JOUEURS(id_j, nom, pays)` et `PARTIES(id_p, date, duree, score, id_joueur)` où :

- `id_j`, de type entier, est la clé primaire de la table `JOUEURS`,
- `nom` est une chaîne de caractères donnant le nom du joueur,
- `pays` est une chaîne de caractères donnant le pays du joueur,
- `id_p`, de type entier, est la clé primaire de la table `PARTIES`,
- `date` est la date (`AAAAMMJJ`) de la partie,
- `duree`, de type entier, est la durée en secondes de la partie,
- `score`, de type entier, est le nombre de points marqués au cours de la partie,
- `id_joueur` est un entier qui identifie le joueur de la partie.

Question 16. Étant donné une chaîne de caractères `cc` contenant le nom d'un joueur, écrire une requête SQL qui renvoie la date, la durée et le score de toutes les parties jouées par le joueur `cc`, listées par ordre chronologique (au choix, croissant ou décroissant).

Question 17. Étant donné un entier s (le score que vient de réaliser une joueuse nommée Alice), écrire une requête SQL qui renvoie la position qu'aura le score s dans le classement des parties par ordre de score (on suppose que la dernière partie d'Alice n'a pas encore été insérée dans la table des parties). En cas d'ex aequo pour le score s (une ou plusieurs parties déjà présentes ayant le score s), le rang sera le même que s'il n'y avait qu'une seule partie avec le score s.

Par exemple, la requête renverra 1 (le score d'Alice est "1$^{\text{er}}$") si aucun score n'est meilleur que s. Autre exemple, si la base de données contient 6 parties dont les scores sont $87, 75, 75, 63, 60, 60$, alors le rang de $s = 75$ sera 2, le rang de $s = 70$ sera 4 et le rang de $s = 60$ sera 5.

Question 18. Écrire une requête SQL qui renvoie le record de France de Tetris couleur, c'est-à-dire le meilleur score réalisé par un joueur dont le pays est la France.

Question 19. Étant donné une chaîne de caractères `cc` contenant le nom d'un joueur (ayant déjà joué au moins une partie de Tetris couleur), écrire une requête SQL qui renvoie le rang du joueur `cc`, c'est-à-dire sa position dans le classement des joueurs par ordre de leur meilleur score dans une partie de Tetris couleur (on traitera les ex aequo de la même manière qu'à la question 17).

F I N

* *
*

extrait concours X-ENS 2019 16

1.4 Syntaxe SQL

Le langage SQL fonctionne par **mots-clefs**, dont l'**ordre** est imposé, ces mots-clefs ne sont pas sensibles à la casse et les sauts de ligne sont libres.

Ainsi

```
SELECT C.region, COUNT(*) AS nb_lycees
FROM lycees AS L JOIN communes AS C ON L.commune = C.nom
WHERE C.population < 100000
GROUP BY C.region
HAVING COUNT(*) >= 2
ORDER BY nb_lycees DESC
```

et

```
select C.region, count(*) as nb_lycees from lycees as L join communes as C on
L.commune = C.nom where C.population < 100000 group by C.region having count(*)
>= 2  order by nb_lycees desc
```

sont équivalentes.

Néanmoins pour la **lisibilité** on impose les règles suivantes:

- les mots-clefs sont en majuscules,
- on indente de manière à rendre le code lisible (conseil: un mot-clef par ligne)

Dans le cas de plusieurs requêtes dans un même fichier (hors programme) on termine chaque requête par un point virgule `;`.

Le squelette général d'une requête ressemble donc à ceci:

Seuls `SELECT` et `FROM` sont obligatoires. Le résultat d'une requête est toujours une **table** : c'est ce qui permet d'enchaîner des requêtes (sous-requêtes, opérations ensemblistes, cf la suite).

2 Projections et sélections

2.1 Projection

Une **projection** revient à choisir certaines **colonnes** particulières, ou des fonctions des colonnes.

Mathématiquement, étant donnée

$$p : A_1 \times \cdots \times A_n \to B_1 \times \cdots \times B_m, \quad (a_1, ..., a_n) \mapsto (f_1(a_1, ..., a_n), ..., f_m(a_1, ..., a_n)),$$

la projection est

$$\pi_p : \mathcal{P}(A_1 \times ... \times A_n) \to \mathcal{P}(B_1 \times ... \times B_m), \quad R \mapsto \{p(a_1, ..., a_n) \mid (a_1, ..., a_n) \in R\}.$$

En pratique, on se contente souvent de projeter sur certaines colonnes.

Dans l'exemple précédent projection de `communes` sur «`nom`, `population`», puis sur «`nom`, `population`, `population / superficie`» :

$\pi_{\texttt{nom, population}}(\texttt{communes})$ $\pi_{\texttt{nom, population, population / superficie}}(\texttt{communes})$

`communes` →

nom	population
Versailles	84 095
Paris	2 103 778
Viroflay	17 237
Le Puy-en-Velay	18 540
Lyon	519 127
Marseille	886 040

nom	population	population / superficie
Versailles	84 095	3 234,4
Paris	2 103 778	20 036,0
Viroflay	17 237	5 745,7
Le Puy-en-Velay	18 540	1 090,6
Lyon	519 127	10 815,1
Marseille	886 040	3 722,9

En SQL la projection, c'est-à-dire le choix des colonnes (obligatoire), s'écrit avec le mot-clef `SELECT` suivi de la liste des colonnes à afficher, puis `FROM` et le nom de la table.

Question. Afficher le nom et la population de toutes les communes.

```
SELECT nom, population
FROM communes
```

Remarque. On peut utiliser l'opérateur *wildcard* * pour sélectionner toutes les colonnes :

```
SELECT * FROM communes
```

Il suffit d'écrire les opérations sur les colonnes, et on peut renommer une colonne avec le mot-clef `AS`.

Question. Afficher le nom (sous le nom `ville`), la population et la densité (sous le nom `densite`) de chaque commune.

```
SELECT nom AS ville, population, population / superficie AS densite
FROM communes
```

2.2 Sélection

Une **sélection** revient à choisir certaines **lignes** particulières d'une table, selon une condition.

Mathématiquement, étant donné un prédicat $\mathcal{C}(a_1, ..., a_n)$, l'opération de sélection est la fonction

$$\sigma_{\mathcal{C}} : \mathcal{P}(A_1 \times ... \times A_n) \to \mathcal{P}(A_1 \times ... \times A_n), \quad R \mapsto \{(a_1, ..., a_n) \in R \mid \mathcal{C}(a_1, ..., a_n)\}.$$

Les conditions que l'on écrira seront des prédicats utilisant :
- les opérations sur les types numériques `+`, `-`, `*`, `/` ;
- les opérateurs de comparaison `=`, `<>`, `<`, `>`, `<=`, `>=` ;
- les opérateurs booléens `AND`, `OR`, `NOT`.

Attention. Le test d'égalité est `=` et non `==`, la différence est `<>` et non `!=`.

Par exemple sélection de `communes` selon «`region` = `'Île-de-France'`», puis selon «`population` > 50000 et `superficie` < 150» :

$\sigma_{\texttt{region = 'Île-de-France'}}(\texttt{communes})$

`communes` →

codeinsee	nom	region	population	superficie
78646	Versailles	Île-de-France	84 095	26,0
75056	Paris	Île-de-France	2 103 778	105,0
78686	Viroflay	Île-de-France	17 237	3,0

$$\sigma_{\texttt{population > 50 000 et superficie < 150}}(\texttt{communes})$$

codeinsee	nom	region	population	superficie
78646	Versailles	Île-de-France	84 095	26,0
75056	Paris	Île-de-France	2 103 778	105,0
69123	Lyon	Auvergne-Rhône-Alpes	519 127	48,0

En SQL la sélection, c'est-à-dire le choix des lignes (optionnel), s'écrit avec le mot-clef `WHERE` suivi de la condition à vérifier, placé après le `FROM`.

Question. Afficher toutes les informations des communes d'Île-de-France.

```sql
SELECT *
FROM communes
WHERE region = 'Île-de-France'
```

Question. Afficher le code INSEE et le nom des communes de plus de 50000 habitants et de moins de 150 km^2.

```sql
SELECT codeinsee, nom
FROM communes
WHERE population > 50000 AND superficie < 150
```

Question. Déterminer le code INSEE et la région de toutes les communes dont la densité est entre 5000 et 10000 habitants par kilomètre carré, ou qui se trouvent en région Auvergne-Rhône-Alpes.

```sql
SELECT codeinsee, region
FROM communes
WHERE (population / superficie >= 5000 AND population / superficie <= 10000)
      OR region = 'Auvergne-Rhône-Alpes'
```

Attention.
- **Pas de chaînage** d'opérateurs : `a < b < c` ne donne pas le résultat attendu, il faut écrire `a < b AND b < c`.
- Ne pas hésiter à utiliser des **parenthèses**, notamment en cas de doute sur la priorité des opérateurs logiques (`AND` est prioritaire sur `OR`).

3 Unicité et clés

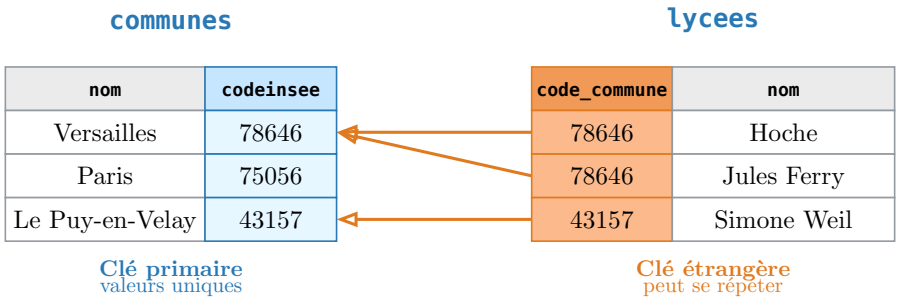
3.1 Clé

On appelle **clé** (ou clé candidate) un ensemble de colonnes dont la valeur permet de déterminer de manière unique une ligne dans la table.

Par exemple dans la table `communes` :
- la colonne `region` n'est pas une clé ;
- la colonne `codeinsee` est une clé ;
- la combinaison des colonnes `nom` et `region` est une clé **s'il** n'existe pas deux communes de même nom dans la même région.

En pratique, plutôt que d'*espérer* qu'un ensemble de colonnes soit une clé :
- on attribue un **identifiant** unique à chaque ligne (souvent un entier `id`) ;
- on déclare une clé comme **primaire** (choix arbitraire parmi les clés).

De plus, étant données deux tables, une clé primaire de l'une aussi présente comme colonne de l'autre (non nécessairement en tant que clé) est appelée **clé étrangère**. Elle *référence* une ligne de la première table.



Convention de notation. Dans la notation `table(attribut1, ...)`, la clé primaire est parfois soulignée. Très souvent les clés sont des identifiants (numériques).

Les clés ont deux intérêts majeurs:
- assurer l'unicité après une projection (ci-dessous),
- permettre d'effectuer une **équi-jointure** de la forme `table1.cle_primaire = table2.cle_étrangère` (voir la partie 7).

3.2 Remarques sur l'unicité

D'un point de vue mathématique, une table étant un ensemble, les lignes sont distinctes par définition. Pour des raisons historiques, en SQL par défaut les lignes ne sont **pas** nécessairement distinctes. C'est notamment le cas après une projection, où le nombre d'attributs de chaque ligne est réduit.

```sql
SELECT nom, region
FROM communes
```

nom	region
...	...
Beaulieu	Auvergne-Rhône-Alpes
Beaulieu	Auvergne-Rhône-Alpes
Beaulieu	Auvergne-Rhône-Alpes
Beaulieu	Auvergne-Rhône-Alpes
Beaulieu	Auvergne-Rhône-Alpes
...	...

```sql
SELECT DISTINCT nom, region
FROM communes
```

nom	region
...	...
Beaulieu	Auvergne-Rhône-Alpes
...	...

Pour obtenir des lignes distinctes, il faut utiliser le mot-clef `DISTINCT` juste après `SELECT`.

Dans un énoncé, `DISTINCT` traduit un « **au moins un(e)** » : une région apparaît dans le résultat dès qu'une de ses communes vérifie la condition, mais une seule fois quel que soit leur nombre. Sans `DISTINCT`, elle apparaîtrait autant de fois qu'elle a de communes vérifiant la condition.

Question. Déterminer les régions ayant au moins une commune de plus de 100000 habitants.

```sql
SELECT DISTINCT region
FROM communes
WHERE population > 100000
```

4 Tris

A priori les lignes d'une table n'ont pas d'ordre particulier. En SQL on peut néanmoins trier le résultat d'une requête avec le mot-clef `ORDER BY`, qui permet d'ordonner les lignes selon une (ou plusieurs) colonne(s) ou expression(s).
- Les entiers et flottants sont ordonnés par l'ordre usuel.
- Les chaînes de caractères sont ordonnées par ordre lexicographique.
- Notamment pour les **dates**, un format `AAAA-MM-JJ` (chaîne) ou `AAAAMMJJ` (entier) permet de trier par ordre chronologique.
- Par défaut le tri est croissant (`ASC`), mais on peut aussi trier de manière décroissante (`DESC`).

- Avec plusieurs colonnes, `ORDER BY a, b DESC` trie selon `a`, puis départage les ex æquo selon `b` (chaque colonne a son propre sens de tri).

Question. Afficher les communes par région (ordre alphabétique) puis, dans chaque région, par population décroissante.

```
SELECT region, nom, population
FROM communes
ORDER BY region, population DESC
```

On peut aussi limiter le nombre de lignes retournées avec `LIMIT n`, en partant d'une certaine ligne avec `OFFSET k` (on saute les k premières lignes).

Résultat trié

	rang	nom
×	1	Paris
×	2	Marseille
▸	3	**Lyon**
▸	4	**Versailles**
▸	5	**Le Puy-en-Velay**
	6	Viroflay

× `OFFSET 2` saute les 2 premières lignes.

↓

▸ `LIMIT 3` renvoie les 3 lignes suivantes.

Question. Afficher les communes de la 3e à la 12e par population décroissante.

```
SELECT nom, population
FROM communes
ORDER BY population DESC
LIMIT 10 OFFSET 2
```

On peut notamment utiliser `LIMIT` et `OFFSET` pour déterminer la **n$^{\text{e}}$ plus grande** valeur d'une colonne (cf requêtes scalaires plus loin).

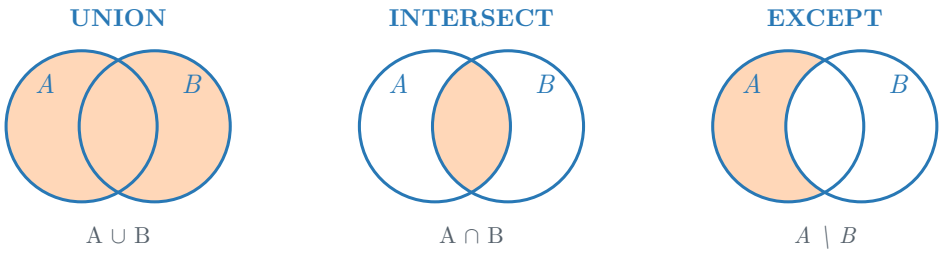
Question. Déterminer la deuxième commune la plus peuplée de la région Auvergne-Rhône-Alpes.

```
SELECT nom, population
FROM communes
WHERE region = 'Auvergne-Rhône-Alpes'
ORDER BY population DESC
LIMIT 1 OFFSET 1
```

5 Opérations ensemblistes

On peut effectuer des opérations ensemblistes sur des tables, à condition que les tables aient le **même nombre de colonnes** et que les colonnes correspondantes aient le **même type**. Il s'agit des opérations usuelles sur les ensembles.

- L'union $\cup$ s'écrit avec le mot-clef `UNION` ;
- l'intersection $\cap$ s'écrit avec le mot-clef `INTERSECT` ;
- la différence ensembliste $\setminus$ s'écrit avec le mot-clef `EXCEPT`.



Par exemple, si `(attribut1a, attribut1b)` et `(attribut2a, attribut2b)` sont de même type :

```
SELECT attribut1a, attribut1b
FROM table1
UNION
SELECT attribut2a, attribut2b
FROM table2
```

Attention.
- après une opération ensembliste, les lignes sont **distinctes** (équivalent à `DISTINCT`),
- une condition nécessaire et suffisante pour pouvoir effectuer une opération ensembliste est que les tables aient le **même nombre de colonnes** et que les colonnes correspondantes aient le **même type**, le nom des colonnes n'a pas d'importance.

Un cas courant d'opération ensembliste est l'**anti-jointure**, c'est-à-dire sélectionner les lignes d'une table qui n'ont pas de correspondance dans une autre table.

Par exemple, on dispose de la table `communes` et d'une table `lycees`.

communes

codeinsee	nom	region	population	superficie
78646	Versailles	Île-de-France	84 095	26,0
75056	Paris	Île-de-France	2 103 778	105,0
78686	Viroflay	Île-de-France	17 237	3,0
43157	Le Puy-en-Velay	Auvergne-Rhône-Alpes	18 540	17,0
69123	Lyon	Auvergne-Rhône-Alpes	519 127	48,0
13055	Marseille	Provence-Alpes-Côte d'Azur	886 040	238,0

lycees

nom	commune	academie
Lycée Hoche	Versailles	Versailles
Lycée Jules Ferry	Versailles	Versailles
Lycée La Bruyère	Versailles	Versailles
Lycée Notre-Dame du Grandchamp	Versailles	Versailles
Lycée Saint Jean Hulst	Versailles	Versailles
Lycée Sainte-Geneviève	Versailles	Versailles
Lycée général et technologique Simone Weil	Le Puy-en-Velay	Clermont-Ferrand
Lycée polyvalent privé Saint-Jacques de Compostelle	Le Puy-en-Velay	Clermont-Ferrand
Lycée Blaise Pascal	Orsay	Versailles

Question. Déterminer la liste des communes sans lycée.

```
SELECT nom FROM communes
EXCEPT
SELECT commune FROM lycees
```

Enfin on peut effectuer le **produit cartésien** de deux tables, c'est-à-dire toutes les combinaisons possibles de lignes des deux tables. Mathématiquement, si $R \subset A_1 \times ... \times A_n$ et $S \subset B_1 \times ... \times B_m$, alors le produit cartésien est

$$R \times S = \{(a_1, ..., a_n, b_1, ..., b_m) \mid (a_1, ..., a_n) \in R \text{ et } (b_1, ..., b_m) \in S\} \subset A_1 \times ... \times A_n \times B_1 \times ... \times B_m.$$

Produit cartésien

table1

cle	expr1
1	a
2	b

table2

cle1	expr2
1	x
3	z
1	z
2	t

$\times \longrightarrow$

cle	expr1	cle1	expr2
1	a	1	x
1	a	3	z
1	a	1	z
1	a	2	t
2	b	1	x
2	b	3	z
2	b	1	z
2	b	2	t

En SQL il suffit d'écrire les deux tables séparées par une virgule dans le `FROM` :

```
SELECT *
FROM table1, table2
```

On peut si besoin renommer les tables avec un **alias** pour simplifier l'écriture des colonnes, notamment en cas d'**ambiguïté** qu'il faut lever (deux colonnes `nom`) :

```
SELECT C.nom, L.nom
FROM communes AS C, lycees AS L
```

Attention. En cas de colonnes ayant le même nom il est essentiel d'utiliser des alias, ou bien d'écrire `table1.attribut` et `table2.attribut` pour lever l'ambiguïté.

Si le produit cartésien tel quel est rarement utilisé, il permettra de mieux comprendre les **jointures** (voir la partie 7).

6 Agrégat

6.1 Agrégat

Formellement, **agréger** des lignes revient à les regrouper selon des valeurs partagées. Par exemple agréger par la valeur de la colonne `region` regroupe les lignes de la table `communes` selon leur région.

`communes`

region	nom	population	superficie
Île-de-France	Versailles	84 095	26,0
Île-de-France	Paris	2 103 778	105,0
Île-de-France	Viroflay	17 237	3,0
Auvergne-Rhône-Alpes	Le Puy-en-Velay	18 540	17,0
Auvergne-Rhône-Alpes	Lyon	519 127	48,0
Provence-Alpes-Côte d'Azur	Marseille	886 040	238,0

regrouper selon `region` $\rightarrow$

Lignes regroupées par `region`

region	nom	population	superficie
Île-de-France	Versailles	84 095	26,0
	Paris	2 103 778	105,0
	Viroflay	17 237	3,0
Auvergne-Rhône-Alpes	Le Puy-en-Velay	18 540	17,0
	Lyon	519 127	48,0
Provence-Alpes-Côte d'Azur	Marseille	886 040	238,0

Une fois les lignes agrégées, on peut leur appliquer des **fonctions d'agrégat**, c'est-à-dire des fonctions prenant une liste de valeurs et renvoyant une seule valeur. Par exemple pour obtenir la plus grande population par région, la somme des superficies par région, et le nombre de communes par région :

Lignes regroupées par `region`

region	nom	population	superficie
Île-de-France	Versailles	84 095	26,0
	Paris	2 103 778	105,0
	Viroflay	17 237	3,0
Auvergne-Rhône-Alpes	Le Puy-en-Velay	18 540	17,0
	Lyon	519 127	48,0
Provence-Alpes-Côte d'Azur	Marseille	886 040	238,0

`MAX(population)`
`SUM(superficie)`
`COUNT(*)`
$\rightarrow$

Une ligne par groupe

region	MAX(population)	SUM(superficie)	COUNT(*)
Île-de-France	2 103 778	134,0	3
Auvergne-Rhône-Alpes	519 127	65,0	2
Provence-Alpes-Côte d'Azur	886 040	238,0	1

Attention. Les colonnes de la table agrégée doivent impérativement **ne dépendre que** de la (ou des) **colonne(s) d'agrégation**, ou des **fonctions d'agrégat** : une ligne du résultat représente tout un groupe, une valeur propre à une commune n'y a pas de sens.

Le mot-clef `GROUP BY` permet d'agréger les lignes selon une ou plusieurs colonnes. Les fonctions d'agrégat sont :

- le nombre de lignes `COUNT(*)`, ou de valeurs d'une colonne `COUNT(attribut)` ;
- la somme `SUM(attribut)` ;
- le minimum `MIN(attribut)` et le maximum `MAX(attribut)` ;
- la moyenne `AVG(attribut)`.

Question. Écrire en SQL l'exemple précédent (plus grande population, somme des superficies et nombre de communes, par région).

```
SELECT region, MAX(population), SUM(superficie), COUNT(*)
FROM communes
GROUP BY region
```

S'il n'y a pas de `GROUP BY`, alors toutes les lignes sont agrégées en un seul groupe, et le résultat obtenu est **scalaire** (une table d'une ligne et une colonne).

Question. Calculer la moyenne de la population de toutes les communes.

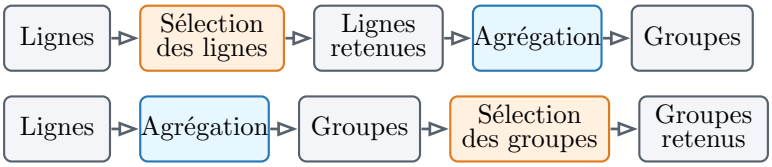
```
SELECT AVG(population)
FROM communes
```

Cela sera utile pour effectuer des **sous-requêtes scalaires** (voir plus loin).

6.2 Sélection et agrégat

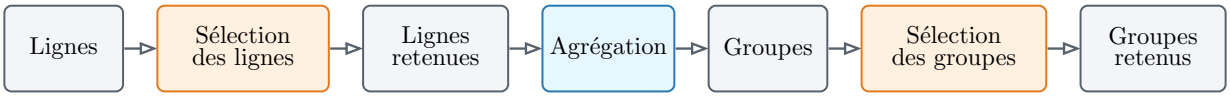
Il y a deux types de sélections avec des agrégats :

- la sélection sur les lignes avant l'agrégation ;
- la sélection sur les groupes après l'agrégation.



On peut par exemple vouloir :

- déterminer la moyenne des populations des communes d'au moins 1000 habitants, région par région (sélection **avant**) ;
- déterminer les régions dont le nombre moyen d'habitants est supérieur ou égal à 1000 (sélection **après**) ;
- ou encore les deux à la fois : déterminer les régions dont le nombre de communes de superficie inférieure à 30 km^2 est au moins 2.



- Le mot-clef pour la sélection **avant** l'agrégation est `WHERE`,
- le mot-clef pour la sélection **après** l'agrégation est `HAVING`.

L'**ordre** est le suivant :

```
SELECT … FROM …
WHERE condition_sur_les_lignes_avant_agrégation
GROUP BY colonne_d_agrégation
HAVING condition_sur_les_groupes_après_agrégation
```

Schéma

`WHERE` → `GROUP BY` → `HAVING`

Question. Déterminer la moyenne des populations des communes d'au moins 1000 habitants, région par région.

```sql
SELECT region, AVG(population)
FROM communes
WHERE population >= 1000
GROUP BY region
```

Question. Déterminer les régions dont le nombre moyen d'habitants est supérieur ou égal à 1000.

```sql
SELECT region, AVG(population)
FROM communes
GROUP BY region
HAVING AVG(population) >= 1000
```

Question. Déterminer les régions dont le nombre de communes de superficie inférieure à 30 km^2 est au moins 2.

```sql
SELECT region
FROM communes
WHERE superficie < 30
GROUP BY region
HAVING COUNT(*) >= 2
```

6.3 Sous-requêtes scalaires

Une requête dont le résultat une seule ligne (aussi appelée scalaire) peut être utilisée, entre parenthèses, comme une valeur dans une condition.

Question. Déterminer toutes les communes de densité supérieure à la densité moyenne de toutes les communes.

```sql
SELECT nom, population / superficie AS densite
FROM communes
WHERE population / superficie >= (SELECT AVG(population / superficie) FROM
communes)
```

La sous-requête peut faire référence à la ligne courante de la requête principale (sous-requête *corrélée*) : elle est alors recalculée pour chaque ligne.

Question. Déterminer toutes les communes de densité supérieure à la densité moyenne de leur région.

```sql
SELECT nom, region, population / superficie AS densite
FROM communes AS C1
WHERE population / superficie >=
(
  SELECT AVG(population / superficie)
  FROM communes AS C2
  WHERE C1.region = C2.region
)
```

Remarque : on peut souvent obtenir le même résultat sans sous-requête scalaire mais avec une **jointure** ou **auto-jointure** (voir la partie sur les jointures).

6.4 Application (X 2019)

Une base de données stocke les résultats obtenus par une communauté de joueurs, dans deux tables `JOUEURS(id_j, nom, pays)` et `PARTIES(id_p, date, duree, score, id_joueur)`, où `id_j` et `id_p` sont les clés primaires et `id_joueur` identifie le joueur d'une partie.

JOUEURS

id_j	nom	pays
1	Alice	France
2	Bob	France
3	Carla	Italie
4	Dan	France

PARTIES

id_p	date	duree	score	id_joueur
1	20190401	312	87	3
2	20190402	208	76	3
3	20190404	275	80	1
4	20190408	240	75	2
5	20190407	190	63	2
6	20190410	230	75	4

Question. Étant donné une chaîne de caractères `cc` contenant le nom d'un joueur, déterminer le **rang** de ce joueur, c'est-à-dire sa position dans le classement des joueurs par ordre de leur meilleur score.

Deux ex æquo partagent le même rang : ici Bob, à égalité avec Dan, est 3e. Le rang est 1 + le nombre de joueurs ayant un meilleur score strictement supérieur.

```sql
SELECT COUNT(DISTINCT id_joueur) + 1
FROM PARTIES
WHERE score > (SELECT MAX(score) FROM PARTIES
               WHERE id_joueur = (SELECT id_j FROM JOUEURS WHERE nom = cc))
```

7 Jointures

7.1 Jointures

Associer, ou **joindre** deux tables revient à combiner les lignes de ces deux tables selon une condition.

Formellement, étant R et S deux tables, et $\mathcal{C}(a_1, ..., a_n, b_1, ..., b_m)$ un prédicat sur les lignes de $R \times S$, la **jointure selon $\mathcal{C}$** est

$$R \underset{\mathcal{C}}{\bowtie} S = \{(a_1, ..., a_n, b_1, ..., b_m) \in R \times S \mid \mathcal{C}(a_1, ..., a_n, b_1, ..., b_m)\}.$$

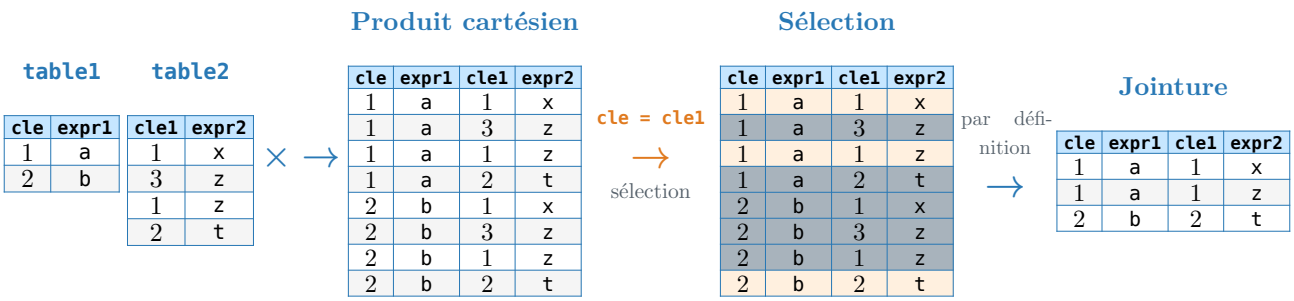
Ainsi une jointure est une sélection sur le produit cartésien : les colonnes sont les colonnes de R et de S, et les lignes sont celles du produit cartésien qui vérifient la condition de jointure. Plus formellement,

$$R \underset{\mathcal{C}}{\bowtie} S = \sigma_{\mathcal{C}}(R \times S).$$

Au programme on se contente d'**équijointures**, c'est-à-dire que la condition de jointure $\mathcal{C}$ est une **conjonction d'égalités**.

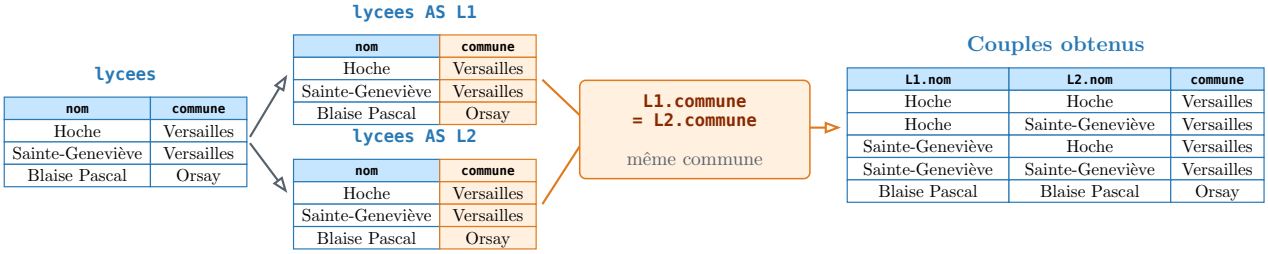
Ces conditions sont souvent de la forme **clé primaire = clé étrangère**, mais pas nécessairement.

La jointure selon `cle = cle1` s'obtient en ne conservant que les lignes du produit cartésien qui vérifient cette égalité.



$$\text{table1} \underset{\text{cle} = \text{cle1}}{\bowtie} \text{table2} = \sigma_{\text{cle} = \text{cle1}}(\text{table1} \times \text{table2})$$

On peut aussi joindre une table avec elle-même, on parle alors d'**auto-jointure**. Cela revient à faire la jointure de *deux copies distinctes* de la table, que l'on renomme avec des alias pour lever l'ambiguïté. Par exemple, une auto-jointure permet d'associer les lycées situés dans la même commune :



Cette équijointure produit des couples avec soi-même et les deux ordres d'une même paire. Pour les supprimer, on peut ajouter `WHERE L1.nom < L2.nom`.

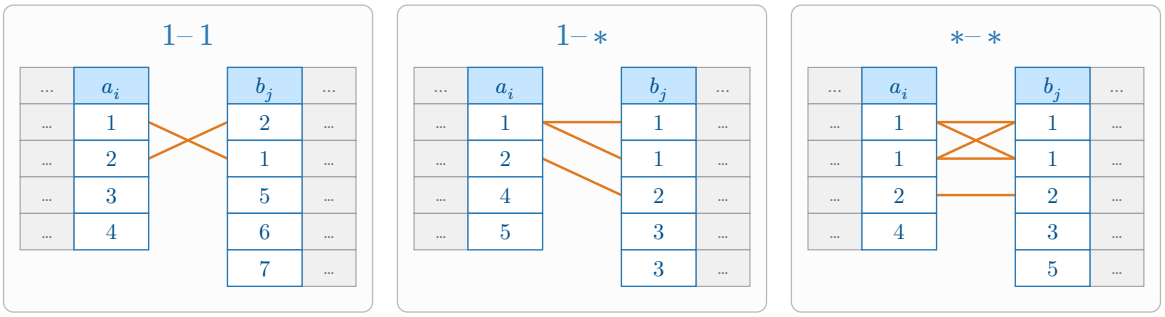
7.2 Types d'associations

Étant donnée une jointure de la forme

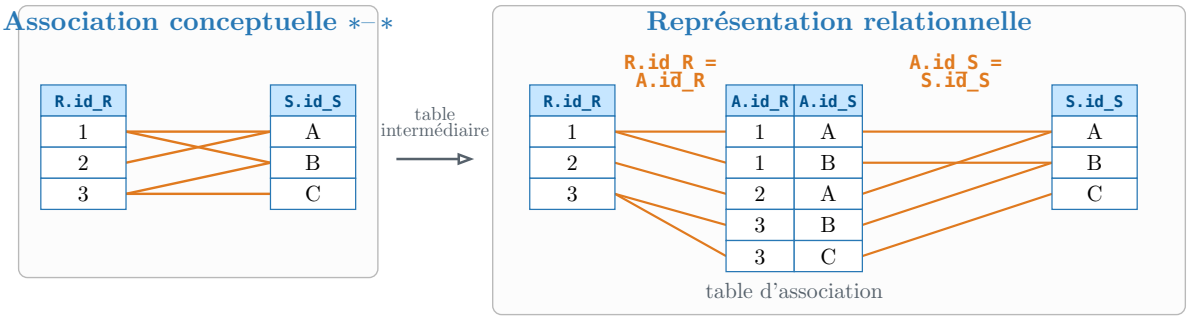
$$R \underset{a_i = b_j}{\bowtie} S = \left\{(a_1, ..., a_n, b_1, ..., b_m) \in R \times S \mid a_i = b_j\right\}$$

on dit qu'elle est

- **1 — 1** si a_i est une clé de R et b_j est une clé de S ;
- **1 — ∗** si a_i est une clé de R (et b_j pas nécessairement) ;
- **∗ — ∗** sinon.



Une **association ∗ — ∗** entre deux types d'objets (un point appartient à plusieurs ensembles, un ensemble contient plusieurs points) se représente dans le modèle relationnel par une **table d'association**, dont chaque ligne est un couple de clés étrangères. On obtient alors deux relations $1 - *$, et on utilise deux équijointures **clé primaire = clé étrangère**, que l'on enchaîne :



7.3 Jointure en pratique

Le mot-clef pour la jointure est `JOIN` et pour la condition de jointure `ON`. Ainsi pour joindre deux tables `table1` et `table2` selon la condition `condition_de_jointure` on écrit :

```
SELECT expression
FROM table1 JOIN table2 ON condition_de_jointure
```

ce qui revient à écrire (jointure = sélection sur le produit cartésien) :

```
SELECT expression
FROM table1, table2
WHERE condition_de_jointure
```

Pour une **auto-jointure**, ou dès que des noms de colonnes identiques apparaissent, il faut utiliser des alias pour lever l'ambiguïté :

```
SELECT expression
FROM table1 AS T1 JOIN table1 AS T2 ON T1.colonne = T2.colonne
```

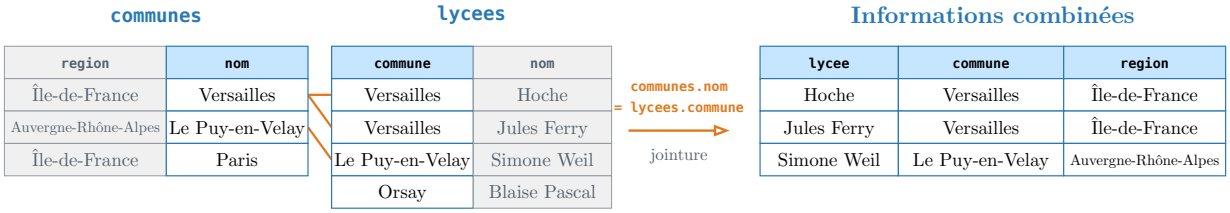
On peut aussi utiliser des parenthèses pour clarifier

```
SELECT expression
FROM (table1 AS T1 JOIN table1 AS T2 ON T1.colonne = T2.colonne)
```

C'est le cas notamment pour les jointures en chaîne, où l'on peut avoir plusieurs tables et plusieurs conditions de jointure.

```
SELECT expression
FROM table1 AS T1 JOIN table2 AS T2 ON T1.colonne1 = T2.colonne1
JOIN table3 AS T3 ON T2.colonne2 = T3.colonne2
```

Les jointures permettent de **combiner des informations** provenant de plusieurs tables (ou de la même table). Si on reprend l'exemple des lycées et des communes :



Sans correspondance, Paris et le lycée Blaise Pascal n'apparaissent pas.

Question. Afficher le nom de chaque lycée avec le nom et la région de sa commune.

```
SELECT L.nom, C.nom, C.region
FROM lycees AS L JOIN communes AS C ON L.commune = C.nom
```

Votre résumé des jointures:

En pratique une question combine **jointure**, **sélection**, **agrégat** et **projection**. On reprend les tables `lycees(nom, commune, academie)` et `communes(codeinsee, nom, region, population, superficie)`, décrites par leur schéma seul comme dans un énoncé.

Question. Pour chaque région, déterminer le nombre de lycées situés dans une commune de moins de 100000 habitants, en ne gardant que les régions qui en ont au moins 2, par ordre décroissant.

```
SELECT C.region, COUNT(*) AS nb_lycees                    -- 4. projection
FROM lycees AS L JOIN communes AS C ON L.commune = C.nom  -- 1. jointure
WHERE C.population < 100000                               -- 2. lignes
GROUP BY C.region                                         -- 3. agrégation
HAVING COUNT(*) >= 2                                      -- 3'. groupes
ORDER BY nb_lycees DESC                                   -- 5. tri
```

Attention. On **écrit** `SELECT … FROM ( … JOIN … ON …) WHERE … GROUP BY … HAVING … ORDER BY`, mais on **construit** la requête (et le logiciel l'exécute) dans l'ordre suivant :

$$\text{JOIN} \to \text{WHERE} \to \text{GROUP BY} \to \text{HAVING} \to \text{SELECT} \to \text{ORDER BY}$$

Une bonne méthode au brouillon consiste à suivre cet ordre : « de quelles tables ai-je besoin ? quelles lignes ? regroupées comment ? quelles colonnes m'intéressent finalement ? dans quelle présentation ? »

7.4 Application (X 2019)

On reprend les tables `JOUEURS(id_j, nom, pays)` et `PARTIES(id_p, date, duree, score, id_joueur)` du sujet X 2019, où `id_joueur` est une clé étrangère référençant `id_j`.

JOUEURS

id_j	nom	pays
1	Alice	France
2	Bob	France
3	Carla	Italie
4	Dan	France

PARTIES

id_p	date	duree	score	id_joueur
1	20190401	312	87	3
2	20190402	208	76	3
3	20190404	275	80	1
4	20190408	240	75	2
5	20190407	190	63	2
6	20190410	230	75	4

Question. Déterminer le **record de France**, c'est-à-dire le meilleur score réalisé par un joueur dont le pays est la France.

Ici le record absolu est 87 (Carla, italienne), mais le record de France est 80 (Alice).

```
SELECT MAX(P.score)
FROM PARTIES AS P JOIN JOUEURS AS J ON P.id_joueur = J.id_j
WHERE J.pays = 'France'
```

Ici, le pays n'est pas dans `PARTIES`, il faut donc **joindre** (clé étrangère `id_joueur` = clé primaire `id_j`) **avant** de sélectionner puis d'agréger. Une sous-requête scalaire ne suffit pas : il y a **plusieurs** joueurs français.

7.5 Application (X 2016)

Une base de données d'un réseau social contient deux tables `INDIVIDUS(id, nom, prenom)` et `LIENS(id1, id2)` où `id` est la clé primaire, et `LIENS` répertorie les liens d'amitié. On suppose que si (x, y) est dans `LIENS`, alors (y, x) l'est aussi.

INDIVIDUS

id	nom	prenom
1	Potter	Harry
2	Granger	Hermione
3	Weasley	Ron
4	Malefoy	Drago

LIENS

id1	id2
1	2
2	1
2	3
3	2
3	4
4	3

Question. Étant donné un entier x, déterminer les (noms, prénoms) des amis de l'individu d'identifiant x.

```
SELECT I.nom, I.prenom
FROM LIENS AS L JOIN INDIVIDUS AS I ON L.id2 = I.id
WHERE L.id1 = x
```

> On sélectionne dans `LIENS` les lignes dont `id1` vaut x, puis la jointure sur `id2` va chercher le nom et le prénom correspondants dans `INDIVIDUS`.

Question. Étant donné un entier x, déterminer les identifiants des individus qui sont amis avec au moins un ami de l'individu d'identifiant x.

```
SELECT DISTINCT L2.id2
FROM LIENS AS L1 JOIN LIENS AS L2 ON L1.id2 = L2.id1
WHERE L1.id1 = x
```

- C'est une **auto-jointure** : `L1` donne les amis de x, `L2` les amis de ces amis. Les alias sont **indispensables** pour distinguer les deux copies de `LIENS`.
- `DISTINCT` évite les doublons (deux chemins vers un même individu).

7.6 Exemple complet: auto-jointure et sous-requête corrélée

On a vu (partie 6.3) que les communes de densité supérieure à la moyenne de leur région s'obtiennent par une sous-requête corrélée. Une auto-jointure donne le même résultat : on associe chaque commune `C1` à toutes les communes `C2` de sa région, on regroupe par commune, et on compare sa densité à la moyenne du groupe.

Question. Déterminer, par auto-jointure, toutes les communes de densité supérieure à la densité moyenne de leur région.

```
SELECT C1.nom, C1.region, C1.population / C1.superficie AS densite
FROM communes AS C1 JOIN communes AS C2 ON C1.region = C2.region
GROUP BY C1.codeinsee, C1.nom, C1.region, C1.population, C1.superficie
HAVING C1.population / C1.superficie >= AVG(C2.population / C2.superficie)
```

7.7 Application (X 2017)

Des points du plan sont stockés dans une table `POINTS(id, x, y)` (`id` clé primaire, deux points distincts n'ont pas les mêmes coordonnées). L'appartenance d'un point à un ensemble est représentée par la table d'association `MEMBRE(idpoint, idensemble)`.

POINTS

id	x	y
1	0	0
2	1	0
3	0	1
4	1	1

MEMBRE

idpoint	idensemble
1	1
2	1
3	1
2	2
4	2
3	3

Question. Étant donnés deux entiers i et j, déterminer les coordonnées des points qui appartiennent à l'intersection des ensembles d'identifiants i et j.

```
SELECT P.x, P.y
FROM POINTS AS P JOIN MEMBRE AS M1 ON M1.idpoint = P.id
                 JOIN MEMBRE AS M2 ON M2.idpoint = P.id
WHERE M1.idensemble = i AND M2.idensemble = j
```

Variante : une jointure par ensemble, puis l'intersection des deux résultats avec `INTERSECT`.

```
SELECT P.x, P.y FROM POINTS AS P JOIN MEMBRE AS M ON M.idpoint = P.id WHERE
M.idensemble = i
INTERSECT
SELECT P.x, P.y FROM POINTS AS P JOIN MEMBRE AS M ON M.idpoint = P.id WHERE
M.idensemble = j
```

8 Quelques points qui reviennent souvent dans les rapports de jury

8.1 Agrégats

- `COUNT(*)` compte les **lignes** (du groupe), `COUNT(attribut)` les **valeurs** de la colonne, `COUNT(DISTINCT attribut)` les valeurs **distinctes**.

```sql
SELECT COUNT(*), COUNT(DISTINCT region)   -- 6 et 3
FROM communes
```

- `SELECT DISTINCT` traduit un « au moins un(e) » de l'énoncé : « les régions ayant au moins une commune de plus de 100000 habitants ».
- Une colonne non agrégée **n'a pas de sens** à côté d'un agrégat (même si certains logiciels l'acceptent) :

```sql
SELECT nom, MAX(population) FROM communes                  -- INCORRECT
SELECT nom FROM communes ORDER BY population DESC LIMIT 1  -- correct
SELECT nom FROM communes                                   -- correct, et garde
WHERE population = (SELECT MAX(population) FROM communes)  -- les ex æquo
```

- Un agrégat **ne peut pas** apparaître dans `WHERE` (il n'est pas encore calculé) : c'est le rôle de `HAVING`.
- Une sous-requête scalaire `= (SELECT …)` doit renvoyer **une seule** valeur. Si plusieurs lignes peuvent correspondre, il faut une **jointure**.

8.2 Ordre et syntaxe

- **Ordre d'exécution** réel d'une requête :

$$\texttt{FROM} \to \texttt{WHERE} \to \texttt{GROUP BY} \to \texttt{HAVING} \to \texttt{SELECT} \to \texttt{ORDER BY} \to \texttt{LIMIT}$$

  Un alias défini dans le `SELECT` est donc utilisable dans `ORDER BY`, mais **pas** dans `WHERE` :

```sql
SELECT nom, population / superficie AS densite
FROM communes
WHERE population / superficie > 5000   -- et non : WHERE densite > 5000
ORDER BY densite DESC
```

- `LIMIT` sans `ORDER BY` renvoie des lignes **non déterminées** ; `LIMIT 1` écarte les **ex æquo**.
- On écrit `table.attribut` (ou `alias.attribut`), jamais `attribut.table`.
- Les chaînes sont entre **apostrophes** et la casse compte : `'France'` $\neq$ `'france'`.

Et enfin on verra en TP:

- La division de deux entiers peut être **entière** selon le logiciel : écrire `1.0 * a / b` pour forcer un flottant.