Les bases de données sont présentes dans d'innombrables domaines : navigation sur Internet, fonctionnement des entreprises, recherche scientifique, etc.

Une base de données est un ensemble structuré de données, géré par un Système de Gestion de Bases de Données (SGBD).

Les objectifs de ce chapitre sont les suivants :

- Présenter le vocabulaire des bases de données, ainsi qu'un modèle pour les représenter.
- Présenter les bases du langage SQL, qui permet d'interroger une base de données.
- Apprendre à écrire des requêtes, sur papier et à l'aide d'un logiciel adapté.

Certains exemples seront illustrés sur la base de données `geographie.sqlite` qui pourra être interrogée

- soit à partir du script python `requetes.py`
- soit à partir d'un système de gestion de bases de données qui devra avoir été préalablement installé (par exemple DB Browser, ou tout autre choix convenable).

La base de donnée `geographie.sqlite` et le fichier `requetes.py` sont disponibles sur Cahier de Prépa.

1. Vocabulaire des bases de données

1.1. Tables

Imaginons que l'on souhaite organiser la gestion des notes de colles d'un établissement scolaire. On pourrait se contenter d'une feuille d'un tableur dans laquelle on écrirait en lignes les noms des élèves, et en colonnes les dates des colles. Chaque cellule contiendrait alors la note associée. Mais comment procéder si l'on veut de plus savoir quel colleur correspond à une colle donnée ? Et gérer toutes les matières, toutes les classes ?

On a rapidement besoin d'organiser les données de façon plus structurée. On va donc créer une base de données, constituée d'une ou plusieurs **tables**, aussi appelées **relations**. Mathématiquement, une table est un sous-ensemble d'un produit cartésien.

Par exemple, (une partie de) la table `Colleurs` pourrait être représentée de la façon suivante :

nom	*prénom*	*matière*	*annéeNaissance*	*affectation*	...
Riemann	Bernhard	Mathématiques	1826	Göttingen	...
Curie	Marie	Chimie	1867	Paris	
Einstein	Albert	Physique	1879	Princeton	
Kant	Emmanuel	Philosophie	1724	Königsberg	
Euler	Leonhard	Mathématiques	1707	Berlin	
Austen	Jane	Anglais	1775	Steventon	

• Chaque **colonne** est nommée par un **attribut** (les en-têtes dans l'exemple ci-dessus).

• À l'exception de l'en-tête, les différentes **lignes** sont aussi appelées les **enregistrements** de la table. Une ligne est donc un tuple. Chaque ligne a autant de composantes qu'il y a d'attributs.

• Chaque composante a une valeur, qui doit être d'un type élémentaire (fixé pour chaque attribut) : entier, flottant, chaîne de caractères, booléen. Pour représenter des horaires ou dates, on pourra utiliser des types numériques ou chaînes de caractères, pour lesquels la relation d'ordre coïncide avec l'écoulement du temps (par exemple '06102022' pour le 6 octobre 2022). Les valeurs des composantes ne peuvent pas être des listes, ensembles, vecteurs, dictionnaires.

• Le type élémentaire associé à un attribut est appelé son **domaine**.
Par exemple, le domaine de `nom` est « chaîne de caractère », celui de `annéeNaissance` est « entier » mais pourrait aussi être « chaîne de caractère » si l'on écrit `'1826'` pour représenter 1826.

• Une table étant un ensemble, l'ordre de ses éléments, *i.e.* l'ordre des lignes, n'a pas d'importance.

• Le **schéma** d'une table est la donnée de son nom et du tuple de ses attributs.
Par exemple le schéma de la table `Colleurs` peut être représenté sous la forme

```
Colleurs(nom, prénom, matière, annéeNaissance, affectation)
```

1.2. Clé primaire

Une **clé primaire** d'une table est un attribut ou tuple d'attributs qui permet d'identifier une ligne de façon unique, c'est-à-dire que deux lignes différentes ne peuvent pas avoir les mêmes valeurs pour la totalité des attributs qui constituent une clé primaire.

Dans les différentes représentations visuelles, on distingue la clé primaire en soulignant le ou les attributs qui la constituent.

Dans la table des colleurs, il se peut que `nom` ne soit pas une clé primaire ; même si c'était le cas à un instant donné, cela ne serait pas un bon choix car une base de données est appelée à être modifiée.

À l'échelle d'un établissement, il est très probable que le couple `(nom, prénom)` soit une clé primaire. Néanmoins, il est parfois plus simple d'ajouter un attribut jouant le rôle d'**identifiant**, souvent numérique, qui agit comme un compteur de lignes et constitue une clé primaire.

1.3. Entités, associations, clés étrangères

Lors de la conception d'une base de données, on s'intéresse en fait à des **entités** (les colleurs, les étudiants, les notes de colles, les établissements, les classes, etc.) et l'on cherche à organiser les informations les concernant, de façon judicieuse, et notamment, en évitant les redondances. Concrètement, une fois la base de données modélisée, chaque entité donnera une table.

Les ensembles rassemblant les entités sont liés entre eux par des **associations** : un étudiant reçoit des notes, un colleur donne des notes, les colleurs exercent leur activité principale dans des établissements, etc.

On est alors amené à distinguer les associations selon leur **cardinalité** : par exemple, un étudiant reçoit plusieurs notes, plusieurs colleurs interviennent dans la même matière.

On distingue plusieurs cas :

— Association $1-1$: association univoque, une entité est liée à exactement une entité.
— Association $1-*$: une entité « du côté 1 » est liée à au moins deux « du côté $*$ », mais deux entités « du côté 1 » ne peuvent pas être liées à la même entité « du côté $*$ » ;
— Association $*-*$: une entité est liée à au moins deux, et ce dans les deux sens.

• Imaginons que la table `Colleurs` se poursuive ainsi :

nom	*prénom*	*matière*	*annéeNaissance*	*affectation*
Riemann	Bernhard	Mathématiques	1826	Göttingen
Curie	Marie	Chimie	1867	Paris
Einstein	Albert	Physique	1879	Princeton
Kant	Emmanuel	Philosophie	1724	Königsberg
Euler	Leonhard	Mathématiques	1707	Berlin
Austen	Jane	Anglais	1775	Steventon
Curie	Marie	Physique	1867	Paris

On est alors en présence d'une association $*-*$ entre *matière* et le reste des informations : plusieurs colleurs interviennent dans chaque matière, mais visiblement, les colleurs peuvent intervenir dans plusieurs matières.

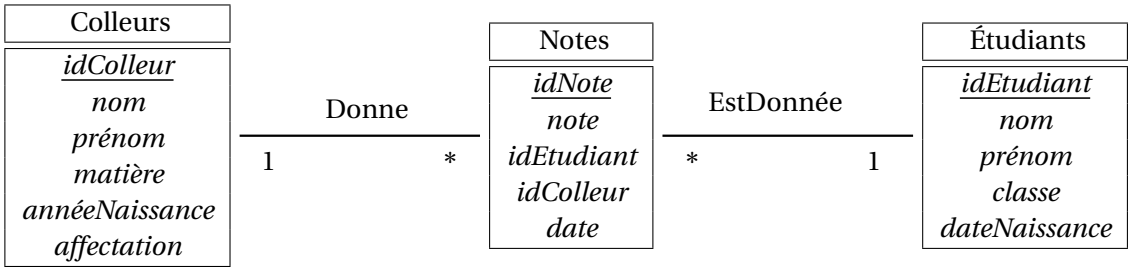
Dans la suite, on imagine que la base de données contient également les tables `Étudiants` et `Notes` ; on considérera que le schéma des différentes tables est

`Colleurs(idColleur, nom, prénom, matière, annéeNaissance, affectation)`

`Étudiants(idEtudiant, nom, prénom, classe, dateNaissance)`

`Notes(idNote, note, idEtudiant, idColleur, date)`

Ces tables sont liées entre elles, et l'on peut représenter ces liens de façon informelle (le programme n'impose aucun formalisme graphique), par exemple :



Attention ! Ci-dessus, les attributs d'une table sont placés dans une colonne pour plus de lisibilité, mais il ne faut pas confondre avec des lignes d'une table !

• Dans le cas d'associations $1-1$, on peut regrouper les données dans un même type d'entités, comme ce serait par exemple le cas avec deux types d'entités « Pays » et « Capitales ». On peut alors prendre comme clé primaire pour le nouveau type d'entités n'importe laquelle des clés primaires des entités d'origine.

• Une association $1-*$ entre deux tables se traduit par la présence d'une **clé étrangère**. Une clé étrangère est un attribut (ou tuple d'attributs) d'une table qui fait référence à la clé primaire d'une autre table.
Par exemple, *idEtudiant* est la clé étrangère pour l'assocation entre les tables `Étudiants` et `Notes`. Elle assure notamment :

— qu'on ne pourra pas ajouter une ligne dans la table `Notes` pour un étudiant qui n'apparaît pas dans la table `Étudiants` ;

— qu'on ne pourra pas supprimer une ligne de la table `Étudiants` pour un étudiant qui a au moins une note enregistrée dans `Notes`.

De même, *idColleur* est la clé étrangère pour l'association entre les tables `Colleurs` et `Notes`.

• Une association $*-*$ sera séparée en deux associations $1-*$ par l'ajout d'une autre table (table de jonction). Par exemple, pour la musique, en présence de deux types d'entités « Chansons » et « Artiste », on créera deux tables `Chansons` et `Artistes` ainsi qu'une table `Interprète` dont les enregistrements seront des couples

`(identifiantChanson, identifiantArtiste)`

et dont l'association avec chacune des deux tables `Chansons` et `Artistes` est du type $1-*$.

2. Requêtes en langage SQL

Dans tout ce qui suit, sauf mention explicite, on se réfère à la base de données `geographie.sqlite`. Elle contient trois tables nommées `communes`, `departements` et `regions` qui contiennent les attributs suivants :

Table `communes`

Attributs	**Description / Type**
`num_departement`	Chaîne de caractères
`nom`	Chaîne de caractères
`canton`	Numéro de canton (chaîne de caractères)
`population_2010`	Population en 2010 (entier)
`population_1999`	Population en 1999 (entier)
`surface`	Surface en km^2 (flottant)
`longitude`	Longitude en milligrades (chaîne de caractères)
`latitude`	Latitude en milligrades (chaîne de caractères)
`zmin`	Altitude minimale en mètres (entier)
`zmax`	Altitude maximale en mètres (entier)

Table `departements`

Attributs	**Description / Type**
`num_departement`	Chaîne de caractères
`num_region`	Chaîne de caractères
`nom`	Chaîne de caractères

Table `regions`

Attributs	**Description / Type**
`num_region`	Chaîne de caractères
`nom`	Chaîne de caractères

Il est recommandé de tester tous les exemples donnés qui portent sur cette base.

2.1. Langage SQL

• Le langage SQL (Structured Query Langage) est un langage qui permet d'interroger une base de données à l'aide de **requêtes**.

• Une requête interroge la base de données et produit une nouvelle table.

• Une requête comporte plusieurs **clauses** (SELECT, FROM, WHERE, etc.)

• L'indentation, les retours à la ligne, n'ont pas d'importance en SQL, mais on respecte certaines habitudes de clarté.

• La casse (minuscules et majuscules) n'a pas d'importance, sauf dans les chaînes de caractères bien sûr.

• Une requête SQL se termine par un point-virgule.

2.2. SELECT

• La clause `SELECT` permet la **sélection** des lignes souhaitées. La requête

```
SELECT *
FROM departements;
```

va afficher toutes les lignes de la table `departements`.

• La clause `SELECT` permet la **projection**, c'est-à-dire le choix de colonnes :

```
SELECT nom
FROM communes;
```

```
SELECT nom, surface
FROM communes;
```

• Il est possible de **renommer** les attributs et les tables avec `AS` (cela ne modifie pas la base de données sur laquelle on travaille) :

```
SELECT nom, population_2010 AS P
FROM communes;
```

```
SELECT *
FROM communes AS C;
```

• On peut limiter le nombre de lignes avec `LIMIT` :

```
SELECT *
FROM communes
LIMIT 100;
```

• Lorsque l'on utilise `LIMIT`, on peut aussi commencer à la ligne $n+1$ avec `OFFSET n` :

```
SELECT *
FROM communes
LIMIT 10 OFFSET 5;
```

• On peut supprimer les répétitions avec `DISTINCT` :

```
SELECT DISTINCT nom
FROM communes;
```

Le mot-clé `DISTINCT` peut être suivi de plusieurs attributs.

• On peut classer les résultats avec `ORDER BY attribut ASC` (par ordre croissant des valeurs de l'attribut `attribut`, par exemple) ou `ORDER BY attribut DESC` (ordre décroissant).
Par défaut, c'est l'ordre croissant qui est utilisé (on peut donc écrire `ORDER BY attribut`).

```
SELECT *
FROM communes
ORDER BY population_2010;
```

```
SELECT *
FROM communes
ORDER BY population_2010 ASC;
```

```
SELECT *
FROM communes
ORDER BY population_2010 DESC;
```

```
SELECT * FROM communes
ORDER BY zmax DESC
LIMIT 10 OFFSET 2;
```

Il est possible d'utiliser deux critères de tri :

```
SELECT *
FROM communes
ORDER BY surface, population_2010;
```

Les lignes seront triées (de façon croissante) par surface de la commune, et en cas d'égalité de surface, par population en 2010.

• On peut faire des calculs avec les opérateurs $+, -, *, /$:

```
SELECT nom, population_2010 / surface AS densite
FROM communes;
```

```
SELECT *
FROM communes
ORDER BY population_2010 - population_1999;
```

ou même

```
SELECT nom, population_2010 - population_1999 AS variation
FROM communes
ORDER BY variation;
```

2.3. WHERE

• La clause `WHERE` permet de sélectionner les lignes en précisant des critères :

```
SELECT *
FROM communes
WHERE nom = 'Brest';
```

• On peut faire des comparaisons avec =, <>, <, <=, >, >=, par exemple

```
SELECT *
FROM communes
WHERE population_2010 < 100;
```

Dans le cas des chaînes de caractères, < et > comparent pour l'ordre alphabétique.

• On peut utiliser les opérateurs logiques `AND`, `OR`, `NOT` :

```
SELECT *
FROM communes
WHERE 1e5 < population_2010 AND population_2010 < 2e5;
```

Remarque – Le programme ne le mentionne pas, mais le mot-clé `LIKE` permet de faire un test sur une chaîne de caractères avec des caractères non spécifiés : le caractère `%` représente un nombre quelconque de caractères. Les minuscules et majuscules n'ont ici pas d'importance. Par exemple :

• Les départements dont le nom commence par la lettre d s'obtiennent avec

```
SELECT nom
FROM departements
WHERE nom LIKE 'd%';
```

• Les communes dont le nom commence par la lettre a et finit par la lettre z s'obtiennent avec

```
SELECT nom
FROM communes
WHERE nom LIKE 'a%z';
```

2.4. Opérations ensemblistes UNION, INTERSECT, EXCEPT

On peut traduire en SQL les opérations ensemblistes $A \cup B$, $A \cap B$, $A \cap^c B$.

Imaginons deux tables, de schémas

```
Chanteurs(nom, prenom, anneeNaissance, tessiture, agent, label)
Acteurs(nom, prenom, anneeNaissance, agent)
```

On peut alors obtenir :

• La table des artistes qui figurent dans au moins une des tables, avec

```
(SELECT nom, prenom FROM Chanteurs)
    UNION
(SELECT nom, prenom FROM Acteurs);
```

• La table des artistes qui figurent dans les deux tables, avec

```
(SELECT nom, prenom FROM Chanteurs)
    INTERSECT
(SELECT nom, prenom FROM Acteurs);
```

• La table des chanteurs qui ne figurent pas dans la table `Acteurs`, avec

```
(SELECT nom, prenom FROM Chanteurs)
    EXCEPT
(SELECT nom, prenom FROM Acteurs);
```

On peut bien sûr ajouter des conditions avec `WHERE`, etc.

2.5. Produit cartésien

Il est possible de combiner les informations de plusieurs tables.

• Le produit cartésien de plusieurs tables s'obtient simplement avec

```
...
FROM table1, table2, ..., tableN
...;
```

Dans le cas de deux tables par exemple, chacune des lignes de la première sera accolée à chacune des lignes de la deuxième. Si les deux tables d'origine ont respectivement n et p lignes, le produit cartésien aura $n \times p$ lignes.

• Il est possible que deux colonnes issus de deux tables différentes aient le même nom. Dans ce cas, pour lever toute ambiguïté, on met en préfixe le nom de la table, par exemple `table.A`

On peut le faire même lorsque ce n'est pas indispensable :

La requête

```
SELECT communes.num_departement, departements.nom AS N
FROM communes, departements
WHERE communes.nom = N;
```

permet de connaître les noms qui sont portés à la fois par une commune et par un département, ainsi que le numéro de département de la commune. On remarquera l'utilisation du renommage pour éviter une répétition.

• Il est possible de faire un produit cartésien d'une table avec elle-même, sur le modèle

```
...
FROM table AS T1, table AS T2
...;
```

Par exemple,

```
SELECT C1.num_departement AS N1, C2.num_departement AS N2, C1.nom
FROM communes AS C1, communes AS C2
WHERE C1.nom = C2.nom
   AND N1 < N2;
```

permet de voir les noms de communes identiques pour deux communes situées dans deux départements différents, ainsi que le numéro des deux départements concernés, tout en évitant les doublons. On voit sur cet exemple qu'un alias peut être utilisé « avant » d'avoir effectivement fait le renommage.

2.6. Jointure (JOIN)

Lorsque l'on réalise un produit cartésien de deux tables par exemple, cela n'a pas toujours de sens que toute ligne de la première table soit accolée à toutes les lignes de la deuxième.

On précise souvent un ou plusieurs critères de cohérence, en imposant l'égalité des valeurs de un ou plusieurs attributs. Par exemple pour obtenir la table des communes avec le nom (et pas le numéro) du département dans lequel elle se situent :

```
SELECT communes.nom, departements.nom
FROM communes, departements
WHERE communes.num_departement = departements.num_departement;
```

Ceci revient à faire une **jointure** (interne) des deux tables, ce qui s'écrit, de façon plus lisible

```
SELECT communes.nom, departements.nom
FROM communes JOIN departements
   ON communes.num_departement = departements.num_departement;
```

Ci-dessus, on n'a pas renommé les tables pour se concentrer sur la syntaxe de la jointure, mais on peut bien sûr le faire.

Le programme ne le mentionne pas, mais, dans le cas où le critère (ce qui suit `ON`) est un nom d'attribut identique, comme dans l'exemple ci-dessus, on peut aussi écrire

```
SELECT communes.nom, departements.nom
FROM communes JOIN departements
   USING(num_departement);
```

De la même façon qu'il est possible de faire un produit cartésien d'une table avec elle-même, il est possible de faire une **auto-jointure**, *i.e.*, la jointure d'une table avec elle-même.

Il est possible de faire une jointure sur plus de deux tables, sur le modèle

```
...
FROM T1 JOIN T2 ... JOIN T_n
ON E1 AND ... AND Em
...;
```

où chaque `Ej` est une égalité.

3. Agrégats

Il est très fréquent de vouloir faire des calculs sur des ensembles de lignes d'une table. Les **fonctions d'agrégation** permettant cela en SQL sont :

- `COUNT` : pour compter les lignes ;
- `MIN` : pour calculer un minimum ;
- `MAX` : pour calculer un maximum ;
- `SUM` : pour calculer une somme ;
- `AVG` : pour calculer une moyenne.

Quelques exemples :

```
SELECT COUNT(*) FROM communes;
```

```
SELECT COUNT(DISTINCT nom) FROM communes;
```

```
SELECT MAX(population_2010) FROM communes;
```

```
SELECT MIN(latitude) FROM communes;

SELECT SUM(population_2010) FROM communes;

SELECT AVG(surface) FROM communes;
```

• Dans les exemples précédents, la requête a produit un seul agrégat, et donc une table constituée d'une seule ligne.

Il est possible de faire plusieurs agrégats avec `GROUP BY` suivi d'un critère de groupement.

Par exemple, pour obtenir la population totale en 2010 de chaque département :

```
SELECT SUM(population_2010)
FROM communes
GROUP BY num_departement;
```

• Il est essentiel de ne demander avec SELECT que des données qui sont compatibles avec les agrégats que l'on a formés. Dans la requête

```
SELECT nom, SUM(population_2010)
FROM communes
GROUP BY num_departement;
```

le nom est sans rapport avec l'agrégation effectuée.

• Il est bien sûr possible de combiner une agrégation avec des conditions spécifiées par `WHERE` :

```
SELECT COUNT(*)
FROM communes
WHERE population_2010 >= 1e4
GROUP BY num_departement;
```

donne, pour chaque numéro de département, le nombre de communes d'au moins 10000 habitants en 2010.

• On peut filtrer les agrégats avec la clause `HAVING` :

```
SELECT num_departement
FROM communes
GROUP BY num_departement
HAVING SUM(population_2010) <= 3*1e5;
```

donne le numéro des départements dont la population totale en 2010 était au plus 300000.

La requête

```
SELECT num_departement, SUM(surface), max(population_2010) AS maxi
FROM communes
GROUP BY num_departement
HAVING maxi >= 1e5;
```

donne, parmi les départements qui avaient au moins une commune de plus de 100 000 habitants en 2010, le numéro de département, la surface totale, et la population de la commune la plus peuplée du département en 2010.

Il importe de bien comprendre la différence entre `WHERE` et `HAVING` dans une agrégation. La bonne question à se poser est : a-t-on besoin de filtrer pour faire le calcul (`WHERE`, sélection en **amont**) ou a-t-on besoin du résultat du calcul pour filtrer (`HAVING`, sélection en **aval**) ?

• On a parfois besoin de faire à la fois une sélection en amont et une autre en aval :

```
SELECT num_departement
FROM communes
WHERE zmin > 1000
GROUP BY num_departement
HAVING MAX(population_2010) > 10000;
```

donne le numéro des départements qui ont au moins une commune en altitude (au moins 1000 mètres) dont la population en 2010 dépassait 10000.

• Attention ! Pour obtenir un maximum ainsi qu'un élément en lequel il est atteint, une erreur classique est d'écrire, par exemple pour obtenir la commune la plus peuplée en 2010,

```
SELECT nom, max(population_2010)
FROM communes;
```

On peut parfois avoir l'impression que « cela fonctionne », mais ce n'est pas la bonne syntaxe, on pourrait tout à fait avoir un nom de commune quelconque, et pas nécessairement la plus peuplée.

La requête suivante convient :

```
SELECT nom, population_2010
FROM communes
ORDER BY population_2010 DESC
LIMIT 1;
```

ou la suivante, qui montre au passage la possibilité de faire des **sous-requêtes** :

```
SELECT nom, population_2010
FROM communes
WHERE population_2010 =
    (SELECT MAX(population_2010) FROM communes);
```

4. Exercices

On travaille sur la base de données `geographie.sqlite`. Pour chaque question, écrire une requête SQL permettant d'obtenir la ligne, ou les lignes, correspondant à la description.

Pour les requêtes du type « les départements tels que ... » ou « les régions telles que ... », on demande les **noms** des départements ou régions concernés (pas leur numéro).

4.1. Requêtes portant sur une seule table

1. Tous les renseignements concernant la commune de Brest, issus de la table `communes`.
2. Le numéro de département de la Savoie.
3. Le nombre de lignes dans la table `communes`.
4. La population totale en 2010.
5. La liste des noms de communes, sans répétition, classés par ordre alphabétique.
6. Le nombre de noms de communes distincts.
7. Les dix communes ayant les points culminants les plus hauts, et l'altitude correspondante, classées par altitude décroissante.
8. Le nombre de communes d'au plus 20 000 habitants en 2010.
9. Les numéros de régions et, pour chaque région, le nombre de départements qui la composent.
10. Les numéros des cantons, et du département associé, et pour chaque canton, la surface moyenne des communes qui le composent.
11. Les numéros de départements, et pour chacun, la population de la commune la plus peuplée en 2010.
12. Les numéros de départements, et pour chacun, la population de la commune la plus peuplée en 2010, parmi les communes d'au plus 50 000 habitants.
13. Les numéros des départements dont la commune la plus peuplée en 2010 comptait au plus 50 000 habitants, et la population de cette commune.
14. Les six communes dont l'augmentation de population entre 1999 et 2010 a été la plus forte, ainsi que cette augmentation, et l'augmentation relative associée.

4.2. Requêtes avec jointures

15. Les départements de la région Occitanie.

16. Les communes qui portent le nom du département dans lequel elles se situent.

17. La superficie de la région Bretagne.

18. Les départements, classés par nombre de communes décroissant, ainsi que ce nombre de communes.

19. Les régions constituées d'au moins trois départements, classées par ordre alphabétique.

20. Les départements dont la population en 2010 était d'au moins 1 000 000 d'habitants, et leur population, classés par ordre décroissant de population.

21. Les départements, et pour chacun, la population de la commune la plus peuplée en 2010 (voir la différence avec la question **11**).

4.3. Requêtes imbriquées

22. La commune ayant la plus grande superficie, ainsi que cette superficie.

23. Le nom et la longitude des communes dont la latitude est la même que celle de Brest.

24. Le pourcentage de communes dont la superficie est d'au moins 10 km^2.

25. Lorsqu'un nom de commune est porté par plusieurs communes, combien de communes en moyenne le portent ?

26. L'espérance et la variance de la variable aléatoire associée à l'attribut `population_2010`.

27. Les départements, et pour chacun, le nom et la population de la commune la plus peuplée en 2010 (voir la différence avec la question **21**).

Nom	Effet	SQL
Projection	Choix de colonnes	`SELECT A1,...,An FROM T`
Sélection	Choix de lignes	`SELECT * FROM T WHERE condition`
Renommage	`Ai` est renommé `Bi`	`SELECT A1,...,An AS B1,...,Bn FROM T`
Union		`(SELECT * FROM T1) UNION (SELECT * FROM T2)`

Intersection		`(SELECT * FROM T1)`
		`INTERSECT`
		`(SELECT * FROM T2)`

Différence		`(SELECT * FROM T1) EXCEPT (SELECT * FROM T2)`
Produit cartésien		`SELECT * FROM T1, T2`

Jointure		`SELECT * FROM T1 JOIN T2 ... JOIN T_n`
		`ON E1 AND ... AND Em`
		où chaque `Ej` est une égalité,
		ou, lorsque les conditions portent sur un même nom d'attribut,
		`SELECT * FROM T1 JOIN T2 ... JOIN T_n`
		`USING(A)`

Fonctions d'agrégation	Regroupement
	+
	Comptage, min., max., somme, moyenne
		`SELECT A1,...,An, f1(B1),...,fm(Bm)`
		`FROM T`
		`GROUP BY A1,...,An`
		
		Fonctions d'agrégation : `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`

Remarque – Dans les syntaxes ci-dessus, `SELECT *` n'est qu'un exemple, `SELECT` n'est bien sûr pas nécessairement suivi de $\star$.