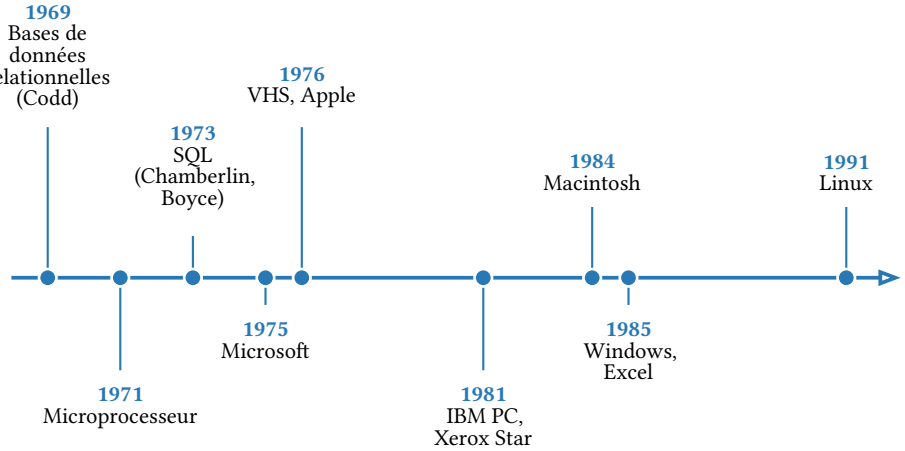
Bases de données et SQL

Contexte

- **Bases de données relationnelles**: Une façon d'organiser des données en *table*.
- **Langage de requête**: Un langage permettant d'*interroger* des bases de données.
- **SQL (Structured Query Language)**: Un exemple historique de langage de requête.

Contexte

- **Bases de données relationnelles**: Une façon d'organiser des données en *table*.
- **Langage de requête**: Un langage permettant d'*interroger* des bases de données.
- **SQL (Structured Query Language)**: Un exemple historique de langage de requête.



Vocabulaire

- **Table** (ou relation)
- **Colonne** (ou attribut) → domaine
- **Ligne** (ou enregistrement)

Mathématiquement

- **Domaine**: ensembles $A_1, ..., A_n$
- **Colonne**: indice $i \in [\![1, n]\!]$
- **Ligne**: $(a_1, ..., a_n) \in A_1 \times ... \times A_n$
- **Table**: $R \subset A_1 \times ... \times A_n$

Notation

On note une table `table(attribut1, ..., attributn)`.
Ici on a donc

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

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

Les superficies sont exprimées en km².

Syntaxe SQL

- Fonctionne par **mots-clefs**
- Mots-clefs non sensibles à la casse, sauts de ligne libres
- **Ordre des mots-clefs** imposé
- Pour la **lisibilité**:
  - Mots clefs en majuscules
  - Usage pertinent de l'indentation
  - On peut terminer la requête par un point-virgule `;`

```
SELECT expression1 AS nom1, expression2
FROM (table1 JOIN table2 ON condition_de_jointure)
WHERE condition_sur_les_lignes
GROUP BY attribut_de_groupe
HAVING condition_sur_les_groupes
ORDER BY expression
LIMIT n OFFSET k
```

Projections et sélections

Projection (formalisme)

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

Mathématiquement, étant donnée

$$\begin{aligned} p : A_1 \times \cdots \times A_n &\to B_1 \times \cdots \times B_m \\ (a_1, ..., a_n) &\mapsto (f_1(a_1, ..., a_n), ..., f_m(a_1, ..., a_n)) \end{aligned}$$

la projection est

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

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

Projection (exemple)

Dans l'exemple précédent, la projection sur «*nom*, *population*» revient à

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

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

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

Projection (exemple)

Dans l'exemple précédent, la projection sur «*nom*, *population*» revient à

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

→

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

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

Et la projection sur «*nom*, *population*, *population / superficie*» revient à

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

→

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

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

Projection (SQL)

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.

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

Projection (SQL)

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.

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

Remarques:

On peut aussi utiliser * pour sélectionner toutes les colonnes.

```
SELECT *
FROM communes
```

Projection (SQL)

Il suffit d'écrire les opérations sur les colonnes.

Par exemple pour calculer la densité

```
SELECT nom, population, population / superficie
FROM communes
```

et on peut aussi renommer une colonne avec le mot-clef `AS`:

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

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 une fonction

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

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 une fonction

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

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`.

Remarque:

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

Sélection (exemple)

Dans l'exemple précédent la sélection pour la condition «`region = 'Île-de-France'`» revient à

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

→

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

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

Sélection (exemple)

Dans l'exemple précédent la sélection pour la condition «`region = 'Île-de-France'`» revient à

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

→

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

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

Et la sélection pour la condition «`population` > 50000 et `superficie` < 150» revient à

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

→

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

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

Sélection (SQL)

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

Par exemple:

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

ou encore

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

Application

On travaille toujours sur la table `communes` précédente.

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

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.

Application

On travaille toujours sur la table `communes` précédente.

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

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.

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

Remarques:

- **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.

Unicité et clés

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.

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
- on déclare une clé comme **primaire** (choix arbitraire)

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
- on déclare une clé comme **primaire** (choix arbitraire)

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**.

Quel intérêt ?

Cela sera très utile pour bien réaliser des **jointures**.

Unicité (SQL)

D’un point de vue mathématique, une table étant un ensemble, les lignes sont distinctes par définition.

Unicité (SQL)

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 **distinctes**.

Cela peut être notamment le cas après une projection, où le nombre d'attributs de chaque ligne est réduit.

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

Unicité (SQL)

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 **distinctes**.

Cela peut être notamment le cas après une projection, où le nombre d'attributs de chaque ligne est réduit.

```
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
…	…

Unicité (SQL)

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 **distinctes**.

Cela peut être notamment le cas après une projection, où le nombre d'attributs de chaque ligne est réduit.

```
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
…	…

Pour obtenir des lignes distinctes, il faut utiliser le mot-clef `DISTINCT`:

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

nom	region
…	…
Beaulieu	Auvergne-Rhône-Alpes
…	…

Tris

Tri (SQL)

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.

- 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` permet de trier par ordre chronologique.
- Par défaut le tri est croissant `ASC`, mais on peut aussi trier de manière décroissante `DESC`.

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

Tri (SQL)

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.

- 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` permet de trier par ordre chronologique.
- Par défaut le tri est croissant `ASC`, mais on peut aussi trier de manière décroissante `DESC`.

```
SELECT nom, population
FROM communes
ORDER BY population DESC
LIMIT 10 OFFSET 2 -- Affiche les communes de 3 à 12 par population décroissante
```

On peut aussi

- Limiter le nombre de lignes retournées avec `LIMIT n`
- En partant d'une certaine ligne avec `OFFSET k`

Application

On travaille toujours sur la table `communes` précédente.

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

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

Application

On travaille toujours sur la table `communes` précédente.

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

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

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

Opérations ensemblistes

Opérations ensemblistes (SQL)

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**.

- 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
```

Remarque: après une opération ensembliste les lignes sont distinctes.

Application (anti-jointure)

On cherche à effectuer une **anti-jointure**, c'est-à-dire à sélectionner les lignes d'une table qui n'ont pas de correspondance dans une autre table.

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.

Application (anti-jointure)

On cherche à effectuer une **anti-jointure**, c'est-à-dire à sélectionner les lignes d'une table qui n'ont pas de correspondance dans une autre table.

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

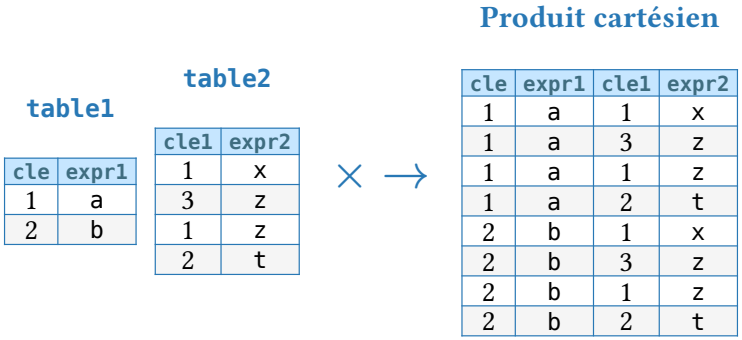
nom	commune	academie
Lycée Hoche	Versailles	Versailles
Lycée Jules Ferry	Versailles	Versailles
Lycée La Bruyère	Versailles	Versailles
Lycée Notre-Dame du Grandchamp	Versailles	Versailles
Lycée Saint Jean Hulst	Versailles	Versailles
Lycée Sainte-Geneviève	Versailles	Versailles
Lycée général et technologique 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
```

Produit cartésien

Mathématiquement le **produit cartésien** de deux tables est … leur produit cartésien.



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

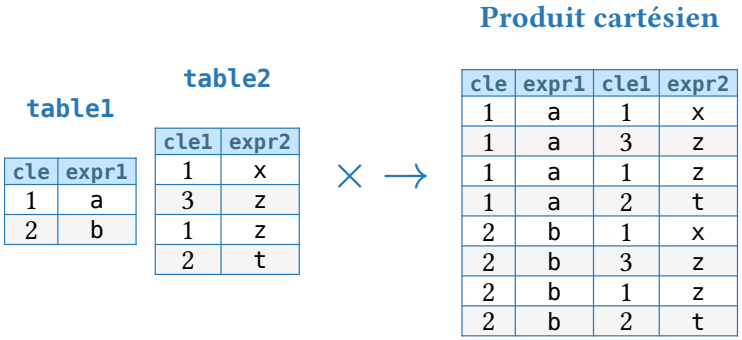
```
SELECT *
FROM table1, table2
```

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).

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

Produit cartésien

Mathématiquement le **produit cartésien** de deux tables est … leur produit cartésien.



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).

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

Quel intérêt ?: Cela sera essentiel pour bien comprendre les **jointures**.

Agrégat

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`

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

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` →

`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(*)` →

`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

Remarque: les colonnes de la table agrégée doivent impérativement **ne dépendre que** de la (ou les) **colonne(s) d'agrégation**, ou des **fonctions d'agrégat**.

Agrégat (SQL)

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)`

Ainsi l'exemple précédent s'écrit en SQL:

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

Agrégat (SQL)

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)`

Ainsi l'exemple précédent s'écrit en SQL:

```
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**. Par exemple pour calculer la moyenne de la population de toutes les communes

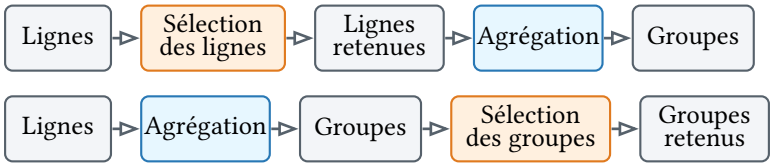
```
SELECT AVG(population)
FROM communes
```

Cela peut être utile pour effectuer des **sous-requêtes scalaires** (cf la suite).

Sélection et agrégat

Il y a deux types de sélections avec des agrégats:

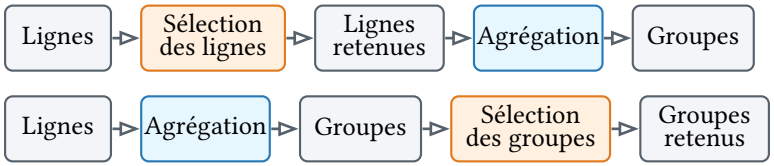
- la sélection sur les lignes **avant** l'agrégation,
- la sélection sur les groupes **après** l'agrégation.



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:

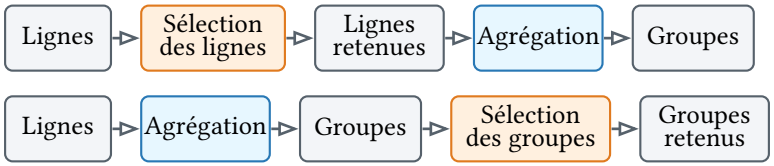
- déterminer la moyenne des populations des communes d'au moins 1000 habitants, région par région,
- déterminer les régions dont le nombre moyen d'habitants est supérieur ou égal à 1000

ou encore

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.

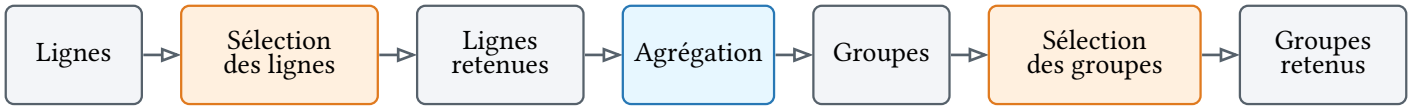


On peut par exemple vouloir:

- déterminer la moyenne des populations des communes d'au moins 1000 habitants, région par région,
- déterminer les régions dont le nombre moyen d'habitants est supérieur ou égal à 1000

ou encore

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



Sélection et agrégat (SQL)

- 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
```

Sélection et agrégat (SQL)

- 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
```

- pour déterminer la moyenne des populations des communes d'au moins 1000 habitants, région par région:

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

- pour déterminer les régions dont le nombre moyen d'habitants est supérieur ou égal à 1000

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

Application

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

Question. Déterminer les régions dont le nombre de communes de superficie inférieure à 30 km² est au moins 2.

Application

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

Question. Déterminer les régions dont le nombre de communes de superficie inférieure à 30 km² est au moins 2.

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

Explication:

- `WHERE superficie < 30` sélectionne les communes de superficie inférieure à 30 km² **avant** l'agrégation,
- `GROUP BY region` agrège les communes par région, et uniquement celles de superficie inférieure à 30 km²,
- `HAVING COUNT(*) >= 2` sélectionne les régions ayant au moins 2 communes de superficie inférieure à 30 km² **après** l'agrégation.

Application (sous-requêtes scalaires)

On peut aussi utiliser des sous-requêtes scalaires, pour effectuer des sélections.

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

Application (sous-requêtes scalaires)

On peut aussi utiliser des sous-requêtes scalaires, pour effectuer des sélections.

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

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

Application (sous-requêtes scalaires)

On peut aussi utiliser des sous-requêtes scalaires, pour effectuer des sélections.

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

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

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

Application (sous-requêtes scalaires)

On peut aussi utiliser des sous-requêtes scalaires, pour effectuer des sélections.

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

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

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

```
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
)
```

Application (sous-requêtes scalaires)

On peut aussi utiliser des sous-requêtes scalaires, pour effectuer des sélections.

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

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

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

```
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 obtenir le même résultat sans sous-requête scalaire mais avec une **auto-jointure** (cf suite).

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 aequo partagent le même rang : ici Bob, à égalité avec Dan, est 3^{e}.

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 aequo partagent le même rang : ici Bob, à égalité avec Dan, est 3e.

```
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))
```

Jointures

Associations et 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

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

Associations et 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'*équijointure*, c'est à dire que la condition de jointure $\mathcal{C}$ est une **conjonction d'égalités**.

Associations et 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'*équijointure*, c'est à dire que la condition de jointure $\mathcal{C}$ est une **conjonction d'égalités**.

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.

Exemple de jointure

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

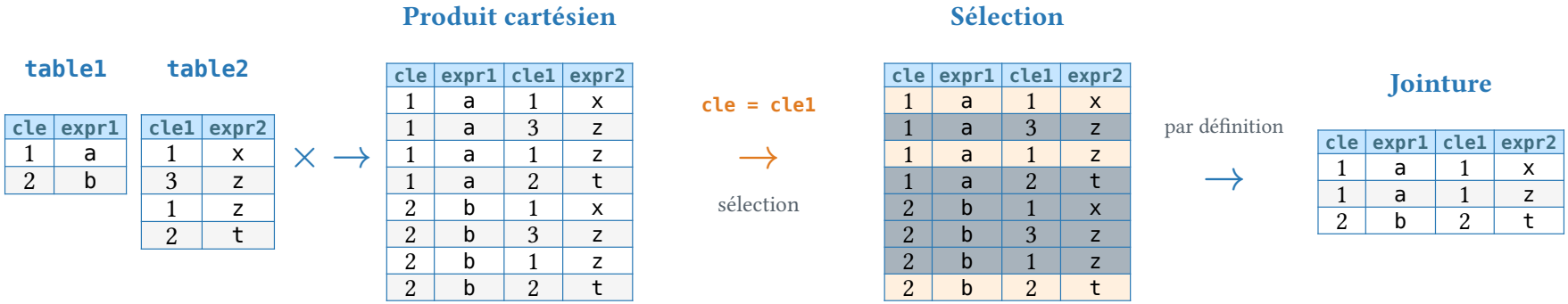


table1

cle	expr1
1	a
2	b

table2

cle1	expr2
1	x
3	z
1	z
2	t

Produit cartésien

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

Sélection

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

Jointure

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

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

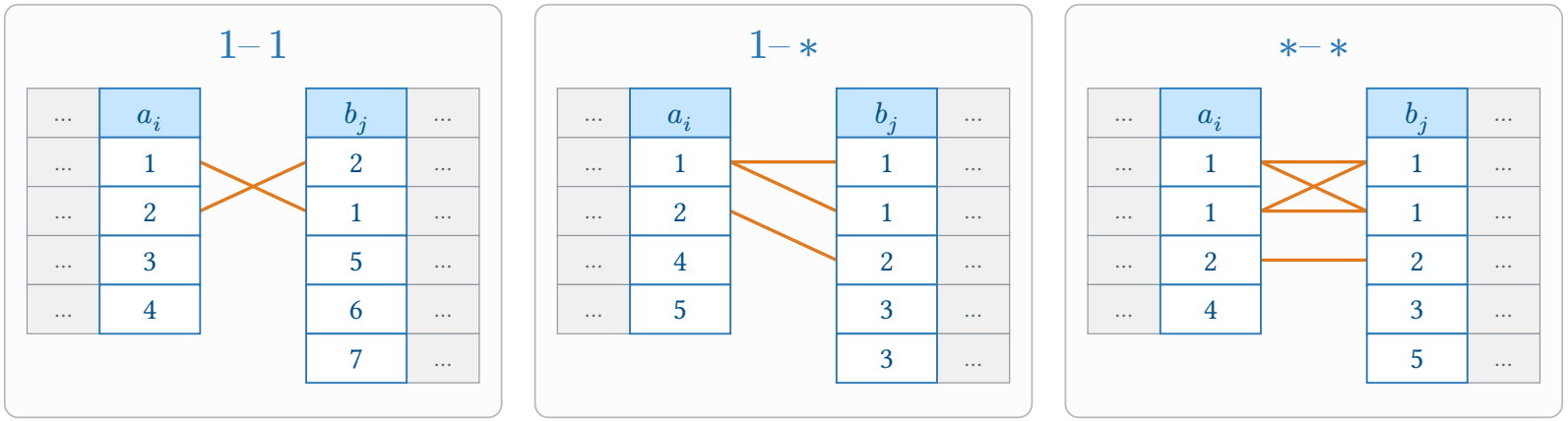
Types d'associations

Étant donnée une jointure de la forme

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

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.



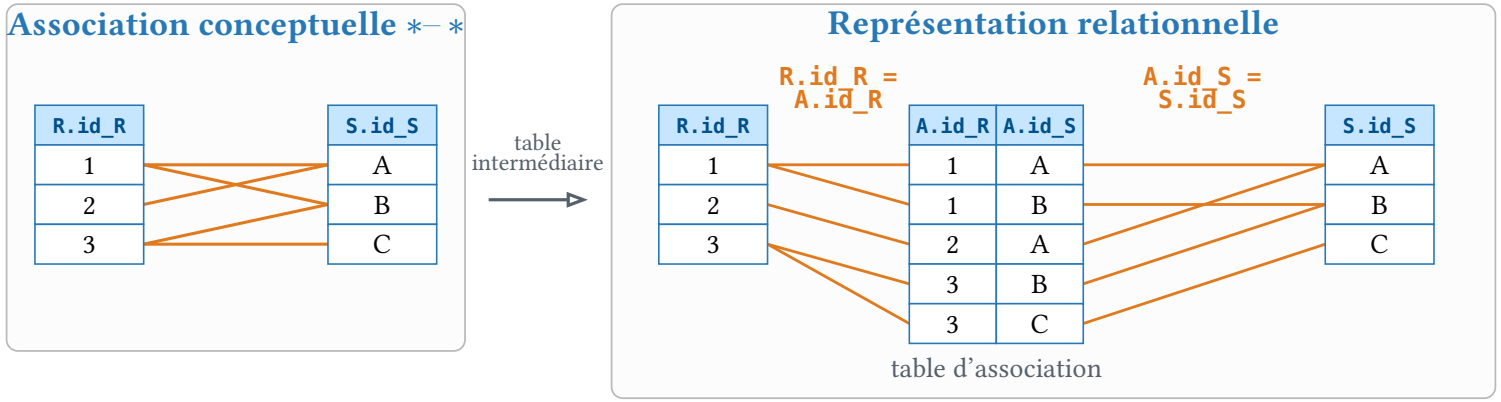
Jointures et clés

En pratique on effectue souvent des conditions de jointure sous la forme **clé primaire = clé étrangère**, et donc de type $1 - *$.

Jointures et clés

En pratique on effectue souvent des conditions de jointure sous la forme **clé primaire = clé étrangère**, et donc de type $1 - *$.

Une **association $* - *$** entre deux types d'objets se représente dans le modèle relationnel par une table d'association. On obtient alors deux relations $1 - *$, et on utilise deux équijointures **clé primaire = clé étrangère**, que l'on enchaîne:



Jointure (SQL)

- 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

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

Jointure (SQL)

- 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

```
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 condition_de_jointure
```

Application: à quoi sert une jointure

Les jointures permettent de **combiner des informations** provenant de plusieurs tables (ou de la même table).

Application: à quoi sert une jointure

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,

communes

region	nom
Île-de-France	Versailles
Auvergne-Rhône-Alpes	Le Puy-en-Velay
Île-de-France	Paris

lycees

commune	nom
Versailles	Hoche
Versailles	Jules Ferry
Le Puy-en-Velay	Simone Weil
Orsay	Blaise Pascal

`communes.nom = lycees.commune`

jointure

Informations combinées

lycee	commune	region
Hoche	Versailles	Île-de-France
Jules Ferry	Versailles	Île-de-France
Simone Weil	Le Puy-en-Velay	Auvergne-Rhône-Alpes

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

Application: à quoi sert une jointure

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,

communes

region	nom
Île-de-France	Versailles
Auvergne-Rhône-Alpes	Le Puy-en-Velay
Île-de-France	Paris

lycees

commune	nom
Versailles	Hoche
Versailles	Jules Ferry
Le Puy-en-Velay	Simone Weil
Orsay	Blaise Pascal

`communes.nom = lycees.commune` → jointure

Informations combinées

lycee	commune	region
Hoche	Versailles	Île-de-France
Jules Ferry	Versailles	Île-de-France
Simone Weil	Le Puy-en-Velay	Auvergne-Rhône-Alpes

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

Ce qui s'écrit en SQL

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

Requête complète (SQL)

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)`.

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.

Requête complète (SQL)

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)`.

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.

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

Ordre: 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:

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

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).

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'
```

Remarque: 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.

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.

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
```

Remarque: 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`.

Application (X 2016, auto-jointure)

On reprend les tables `INDIVIDUS(id, nom, prenom)` et `LIENS(id1, id2)` précédentes.

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 identifiants des individus qui sont amis avec au moins un ami de l'individu d'identifiant x.

Application (X 2016, auto-jointure)

On reprend les tables `INDIVIDUS(id, nom, prenom)` et `LIENS(id1, id2)` précédentes.

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 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
```

Remarques:

- 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).

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.

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
```

Remarque: un « et » entre deux appartenances se traduit en joignant **deux fois** la table d'association `MEMBRE` (une copie par ensemble). Pour $i = 1$ et $j = 2$ on obtient $(1, 0)$.

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
```

Remarque: un « et » entre deux appartenances se traduit en joignant **deux fois** la table d'association `MEMBRE` (une copie par ensemble). Pour $i = 1$ et $j = 2$ on obtient $(1, 0)$.

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
```

Points d’attention

Attention (SQL): 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».

Attention (SQL): 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,
WHERE population = (SELECT MAX(population) FROM communes)    -- garde les ex æquo
```

Attention (SQL): 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,
WHERE population = (SELECT MAX(population) FROM communes)    -- garde 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**.

Attention (SQL): ordre et syntaxe

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

$$\text{FROM} \rightarrow \text{WHERE} \rightarrow \text{GROUP BY} \rightarrow \text{HAVING} \rightarrow \text{SELECT} \rightarrow \text{ORDER BY} \rightarrow \text{LIMIT/OFFSET}$$

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
```

Attention (SQL): ordre et syntaxe

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

$$\text{FROM} \to \text{WHERE} \to \text{GROUP BY} \to \text{HAVING} \to \text{SELECT} \to \text{ORDER BY} \to \text{LIMIT/OFFSET}$$

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