Le langage SQL

Le langage SQL (*Structured Query Language*) est considéré comme le langage d'accès normalisé aux bases de données relationnelles et repose sur de solides fondements mathématiques. Contrairement aux langages courants *impératifs* (PYTHON, C) ou *fonctionnels* (CAML), c'est un langage dit *déclaratif*. L'utilisateur spécifie le résultat qu'il attend et non le moyen d'y parvenir, et c'est le SGBD qui trouve la manière d'aboutir à ce résultat le plus efficacement possible.

Dans les années 70, IBM travaille sur un prototype de bases de données relationnelles reposant sur la théorie de CODD. Le langage de programmation de ce système est appelé SEQUEL (*Structured English Query Language*) puis sera rebaptisé SQL. Le succès du langage SQL dans le monde des entreprises est en grande partie dû à sa normalisation. Phénomène rare dans le domaine de l'informatique, ce langage a été adopté par pratiquement tous les éditeurs de SGBD commerciaux.

1 Définir et modifier une base de données

1.1 Contraintes d'intégrité

Comme nous l'avons indiqué lors du cours d'introduction, l'un des principaux objectifs d'un SGBD est de garantir la cohérence des données lors de toute mise à jour de la base. Par exemple, il est nécessaire de garantir que toute valeur d'un tuple pour un attribut correspond bien au domaine de cet attribut. Nous avons également rencontré la contrainte d'*intégrité référentielle* : toute valeur pour une clé étrangère doit exister comme valeur pour la clé primaire dans la table référencée. Un dernier exemple : il faut aussi garantir qu'une clé primaire est bien une clé candidate et que deux tuples ne peuvent donc avoir la même valeur pour les attributs de cette clé dans la table. Il existe bien d'autres contraintes d'intégrité possibles. Le langage SQL doit permettre d'exprimer implicitement ou explicitement toutes ces contraintes afin que le SGBD puisse les interpréter et garantir la cohérence de la base en rejetant toute mise à jour qui les violerait.

1.2 Création d'une table

Comme toute cette partie n'est pas vraiment au programme des classes préparatoires, nous allons simplement donner un exemple. La commande suivante montre comment la relation **eleve** serait créée en SQL.

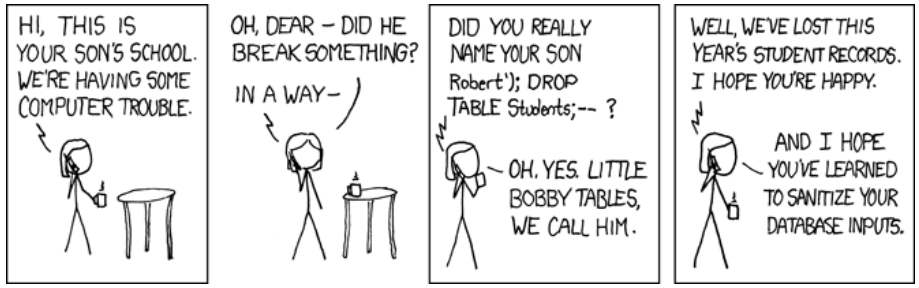
SQL

```sql
CREATE TABLE eleve (
  id INTEGER NOT NULL UNIQUE,
  ine INTEGER NOT NULL CHECK(1000<=ine AND ine<10000) UNIQUE,
  lycee_origine TEXT,
  no_classe INTEGER,
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES personne(id)
  FOREIGN KEY(no_classe) REFERENCES classe(no),
);
```

On peut remarquer :

- Des contraintes de domaine. Ainsi *id* doit être un entier et *lycee_origine* une chaîne de caractères.
- Des contraintes de non nullité. Ainsi l'identifiant et le numéro national d'étudiant doivent être définis et ne peuvent être *NULL*.
- Des contraintes d'unicité. Ainsi l'identifiant et le numéro national d'étudiant doivent être uniques. Ce sont alors deux clés candidates.
- Une contrainte de vérification (**CHECK**). Ainsi le numéro national d'étudiant doit être composé d'exactement quatre chiffres.
- Une contrainte de clé primaire. Ainsi l'identifiant est une clé primaire pour cette table.
- Des contraintes de clé étrangère. Ainsi l'identifiant est une clé étrangère qui référence l'identifiant de la table **personne** et le numéro de classe est une clé étrangère qui référence le numéro dans la table **classe**.

1.3 Suppression d'une table



https://xkcd.com/327/

1.4 Modification d'une table

On peut modifier une table avec la commande **ALTER TABLE**, insérer des lignes avec **INSERT INTO**, modifier des lignes avec **UPDATE** ou encore supprimer des lignes avec **DELETE**. Aucune de ces commandes n'est au programme.

1.5 Protection d'accès

Lorsqu'une base de donnée peut être consultée et modifiée par une communauté d'utilisateurs, il est absolument crucial de gérer correctement les permissions accordées aux utilisateurs, par exemple l'accès ou non à une table, ce que l'on peut faire avec les instructions **GRANT** et **REVOKE** par exemple. Ceci est complètement hors programme.

2 Interroger une base de données

Une *requête* est une combinaison d'opérations portant sur des tables (relations) et dont le résultat est lui-même une table, dont l'existence est éphémère (le temps de la requête).

Une requête se présente généralement sous la forme :

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

- La clause **SELECT** permet de spécifier les attributs. Elle réalise donc la *projection*.
- La clause **FROM** permet de spécifier les tables sur lesquelles porte la requête. Ce pourra être un produit cartésien ou une jointure de tables.
- La clause **WHERE**, facultative, permet de spécifier des conditions que doivent respecter les lignes sélectionnées. Elle réalise donc la *sélection*.

Par convention on notera les mots-clés du langage en majuscules. Une requête SQL se termine toujours par un point-virgule.

2.1 Les opérateurs DISTINCT et ALL

Lorsque le SGDB construit la réponse d'une requête, il rapatrie toutes les lignes qui satisfont la requête, généralement dans l'ordre où il les trouve même si ces dernières sont en double. Supprimer les doublons est une opération qui peut être coûteuse et qui n'est pas effectuée par défaut. Le mot-clé **DISTINCT** permet de forcer le SGDB à éliminer les doublons.

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

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

2.2 Visualisation d'une table

On ne peut pas directement visualiser le contenu d'une table. En revanche, on peut faire une requête pour renvoyer tous les tuples d'une table. Le caractères $\star$ (*étoile*) permet d'indiquer que l'on souhaite tous les attributs de la table. Par exemple, pour renvoyer tous les tuples de la table **eleves**, on effectue la requête :

SQL
```
SELECT * FROM eleve;
```

3 Traduction des opérateurs relationnels

3.1 Traduction de l'opérateur de projection

Si $\mathcal{R}(S)$ est une relation et si $X = (A_1, A_2, \ldots, A_k) \subseteq S$ est une séquence d'attributs, l'opération de projection $\Pi_{A_1,A_2,\ldots,A_k}(\mathcal{R})$ se traduit simplement en SQL par :

SQL
```
SELECT DISTINCT A_1, A_2, ..., A_k FROM R;
```

Sans le mot-clé **DISTINCT**, les éventuels doublons ne sont pas éliminés, et on n'obtient donc pas rigoureusement parlant une relation, en général.

Par exemple, pour obtenir les prénoms et noms des personnes, sans doublons, on effectue la requête :

SQL
```
SELECT DISTINCT prenom, nom FROM personne;
```

3.2 Traduction de l'opérateur de sélection

Si $\mathcal{R}(S)$ est une relation et E une expression logique, l'opération de selection $\sigma_E(\mathcal{R})$ se traduit simplement en SQL par :

```sql
SELECT * FROM R WHERE E;
```

Par exemple, pour sélectionner les personnes dont le nom est *Martin* on écrit :

```sql
SELECT * FROM personne WHERE nom = 'Martin';
```

Pour délimiter une chaîne de caratères, on utilise les guillemets simples. On peut combiner deux expressions logiques avec les opérateurs logiques **AND** (et logique), **OR** (ou logique) et **NOT** (non logique). Par exemple, pour sélectionner les identifiants des professeurs de chaire supérieure ou bien avec au moins 15 ans d'ancienneté sans être simplement agrégés de classe normale on peut écrire :

```sql
SELECT id FROM prof
WHERE grade = 'Agr CS' OR
  (grade != 'Agr' AND anciennete >= 15);
```

Nous reviendrons plus tard sur ce que peuvent constituer les expressions logiques.

3.3 Renommage

L'opérateur **AS** permet de renommer une table, une colonne ou de renommer une colonne créée dans la requête.

Pour renommer une table on écrit `<nom_table> AS <nouveau_nom>`. Le mot-clé **AS** est facultatif. Les deux requêtes suivantes sont donc équivalentes :

```sql
SELECT * FROM personne AS p;
SELECT * FROM personne p;
```

Une application typique du renommage de table est de simplifier les noms de table trop longs lorsque nous en aurons besoin pour préfixer des noms d'attributs ambigus.

La syntaxe est la même pour renommer un attribut. Par exemple, la requête suivante permet de traduire l'opération $\rho_{anciennete \leftarrow anc, grade \leftarrow grd}(prof)$.

```sql
SELECT id, anciennete AS anc, grade AS grd FROM prof;
```

Ici aussi le mot-clé **AS** est facultatif, mais la requête me semble plus lisible en le conservant explicitement.

3.4 Opérations ensemblistes

Les opérations ensemblistes se traduisent assez naturellement en utilisant les mots-clés **UNION**, **INTERSECT** et **EXCEPT**. Attention, il ne faut pas mettre de parenthèses.

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

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

Par exemple, pour avoir l'ensemble des prénoms et des noms :

```sql
SELECT prenom FROM personne UNION SELECT nom FROM personne;
```

Ici c'est le nom de l'attribut à gauche qui est utilisé dans la table résultante. Pour conserver tous les doublons :

```sql
SELECT prenom FROM personne UNION ALL SELECT nom FROM personne;
```

Pour éliminer les doublons parmi les noms et prénoms respectivement, mais pour conserver ensuite les doublons éventuels lors de l'union :

SQL
```sql
SELECT DISTINCT prenom FROM personne
UNION ALL
SELECT DISTINCT nom FROM personne;
```

3.5 Produit cartésien

Le produit cartésien de deux relations se traduit simplement en séparant les tables par des virgules dans la clause **FROM**. Si $\mathcal{R}_1$ et $\mathcal{R}_2$ sont deux tables, l'opération de produit cartésien $\mathcal{R}_1$ et $\mathcal{R}_2$ se traduit en SQL par :

SQL
```sql
SELECT * FROM R_1, R_2;
```

Par exemple, pour répondre à la demande du secrétariat pour la journée des portes ouvertes, on traduit $\Pi_{personne.nom,prenom,classe.nom} \circ \sigma_{id \neq prof\_princ}(personne \times classe)$ en SQL par :

SQL
```sql
SELECT p.nom, p.prenom, c.nom FROM personne p, classe c
WHERE p.id != c.prof_princ;
```

Ici le renommage des tables est utile pour simplifier. On aurait pu écrire `prenom` au lieu de `p.prenom`, mais il me semble que c'est plus clair ainsi.

On peut bien sûr faire un produit cartésien de plusieurs relations en même temps. Par exemple, pour avoir le nom des professeurs, la matière et la classe de chaque enseignement possible, on peut écrire

SQL
```sql
SELECT p.nom, matiere, c.nom
FROM personne p, enseigne e, classe c
WHERE p.id = e.id_prof AND e.no_classe = c.no;
```

En y regardant de près, on s'aperçoit que l'on vient de réaliser ni plus ni moins qu'une jointure !

3.6 Jointure

Une jointure n'étant formellement rien d'autre qu'un produit cartésien suivi d'une sélection, on sait donc déjà traduire l'opérateur de jointure. On dispose cependant en SQL d'une [1] syntaxe explicite pour la jointure. Si $\mathcal{R}_1$ et $\mathcal{R}_2$ sont deux relations et si E est une expression, alors l'opération de jointure $\mathcal{R}_1 \underset{E}{\bowtie} \mathcal{R}_2$ se traduit par :

SQL
```sql
SELECT * FROM R_1 JOIN R_2 ON E;
```

Par exemple, pour avoir les noms et prénoms des professeurs de plus de 10 ans d'ancienneté on peut faire :

SQL
```sql
SELECT nom, prenom
FROM personne JOIN prof ON personne.id = prof.id
WHERE anciennete >= 10;
```

On peut bien sûr enchaîner les jointures. Pour traduire la dernière requête de la section précédente, on écrirait donc

SQL
```sql
SELECT p.nom, matiere, c.nom
FROM personne p JOIN enseigne e ON p.id = e.id_prof
                JOIN classe c ON e.no_classe = c.no;
```

Plusieurs arguments nous incitent à utiliser une syntaxe explicite pour la réalisation d'une jointure.

- L'opérateur de jointure est souvent l'opération naturelle pour recoller deux tables. Autant que cela apparaisse explicitement !

1. En réalité de nombreuses ! Nous verrons plus tard les jointures externes gauches et droites.

- L'utilisation de la clause **WHERE** pour spécifier la condition de jointure ne permet pas de faire la distinction logique entre ce qui relève de la sélection (dépendant fortement de la requête) et ce qui relève de la jointure (dépendant plutôt de la structure logique du découpage en tables de la base de données).
- La lisibilité des requêtes est ainsi plus grande en utilisant la syntaxe de l'opérateur **JOIN**.
- L'utilisation de l'opérateur **JOIN** permet plus simplement de cloisonner les conditions de jointures entre chaque couple de tables.
- A priori[2] un produit cartésien est plus coûteux à calculer : il faut considérer toutes les associations possibles. Inversement, les gestionnaires de bases disposent d'algorithmes efficaces et optimisés pour effectuer des jointures.

Règle : Utiliser la syntaxe avec **JOIN** ... **ON** dès que l'on réalise une jointure logique entre deux tables.

4 Fonctions et calculs sur une relation

4.1 Valeur NULL

Lorsque la valeur d'un attribut n'est pas renseignée, on lui attribue la valeur **NULL** en SQL. On dispose des opérateurs **IS NULL** et **IS NOT NULL** pour tester la valeur à **NULL**. Attention, ces opérateurs sont nécessaires, en effet **NULL** = **NULL** et **NULL** != **NULL** sont tous deux faux[3].

4.2 Calculs sur les attributs

Il est possible d'appliquer des fonctions aux attributs d'une table au sens où l'on applique la fonction aux valeurs des attributs dans chaque tuple. On peut également combiner différents attributs. On peut ensuite donner un nom à la colonne ainsi créée à l'aide du mot-clé **AS**, ce qui est impératif si on souhaite l'utiliser plus tard.

Par exemple pour obtenir le taux de remplissage des classes, en pourcentage :

SQL
```
SELECT nom, 100 * effectif / 36 AS taux_remplissage
FROM classe;
```

nom	taux_remplissage
MPSI 3	111
PCSI 2	83
MP*	61

Un autre exemple : pour obtenir le nom complet d'une personne, on peut combiner les attributs prénom et nom, avec l'opérateur de concaténation des chaînes de caractères ||.

SQL
```
SELECT prenom || ' ' || nom AS nom_complet, tel
FROM personne
WHERE nom = 'Martin' AND tel IS NOT NULL;
```

nom_complet	tel
Filou Martin	0612345678
Pierre Martin	0635147630

4.3 Fonctions et opérateurs mathématiques

On dispose des opérateurs mathématiques les plus courants : + (addition), - (soustraction), * (multiplication), / (division entière), % (modulo) et **ABS** (valeur absolue).

Pour les chaînes de caractères, en plus de l'opérateur de concaténation ||, on peut citer la fonction **LENGTH** qui renvoie le nombre de caractères d'une chaîne de caractères et les fonctions **LOWER** et **UPPER** qui permettent de convertir une chaîne de caractères en minuscule et majuscule, respectivement.

Enfin, citons l'opérateur de comparaison **LIKE** pour lequel un tiret bas _ remplace n'importe quel caractère et un pourcentage % remplace un nombre quelconque de caractères. Par exemple :

- `LIKE 'N%'` : commence par 'N' ;
- `LIKE '%a%'` : contient au moins un 'a' ;
- `LIKE 'pa%on'` : commence par 'pa', finit par 'on' (paon, pantalon, passion, etc.).

Par exemple, si l'on souhaite le prénom accolé aux deux derniers chiffres[4] du numéro de téléphone comme mot de passe[5] des personnes dont le prénom est de longueur au moins 6 et a pour deuxième lettre un 'i', on peut établir la requête suivante.

2. Cela dépend de l'optimisation des requêtes, qui sait souvent reconnaître une jointure derrière un produit cartésien et faire la recherche efficacement.
3. En réalité, ils sont tous les deux évalués à **NULL**.
4. Les optionnaires informatique devraient être scandalisés ici.
5. Ici, tout le monde devrait être scandalisé.

SQL
```
SELECT prenom || (tel % 100) AS mot_de_passe
FROM personne
WHERE LENGTH(prenom) >= 6 AND prenom LIKE '_i%'
  AND tel IS NOT NULL;
```

login
Pierre89
Pierre30

4.4 Fonctions d'agrégation

Contrairement aux opérations précédentes qui agissent individuellement sur toutes les valeurs d'une ou plusieurs colonnes, on dispose de fonctions qui utilisent toutes les valeurs des lignes pour renvoyer un seul résultat. Ce sont les fonctions d'*agrégation*. Attention, l'utilisation de ces fonctions aboutit à une relation avec un seul tuple ou dit autrement *une seule ligne*. Les fonctions d'agrégation au programme sont **COUNT**, **MIN**, **MAX**, **SUM** et **AVG**, et sont donc à connaître parfaitement.

Par exemple pour avoir la somme des effectifs des classes :

SQL
```
SELECT SUM(effectif) AS effectif_total FROM classe;
```

effectif_total
92

Pour avoir le minimum, la moyenne et le maximum des anciennetés des professeurs, on peut faire la requête suivante qui contient le seul tuple $(5, 11.67, 20)$.

SQL
```
SELECT MIN(anciennete), AVG(anciennete), MAX(anciennete)
FROM prof;
```

Pour ne pas prendre en compte les doublons, on utilise **DISTINCT**. Attention, le **DISTINCT** se fait avant agrégation, donc dans la parenthèse. Il ne peut y avoir de doublons en sortie puisqu'il n'y a qu'un seul tuple ! Par exemple, pour compter le nombre de professeurs principaux :

SQL
```
SELECT COUNT(DISTINCT prof_princ) FROM classe;
```

Si l'on souhaite compter le nombre de tuples dans une relation (*i.e.* sa taille), on peut utiliser **COUNT** sur n'importe quel attribut (qui ne prend pas la valeur **NULL**). Mieux vaut dans ce cas utiliser la syntaxe spécifique **COUNT**`(*)`. Par exemple pour avoir le nombre d'élèves :

SQL
```
SELECT COUNT(*) FROM eleve;
```

⚠ Toutes ces requêtes commencent toujours par **SELECT** !

5 Sous-requêtes

Une *sous-requête* ou *requête interne* est une requête SQL, entre parenthèses, imbriquée dans une autre requête SQL, qualifiée de requête *externe* ou *englobante*.

5.1 Sous-requêtes dans une clause FROM

Les tables mentionnées dans la clause **FROM** peuvent très bien correspondre à des tables résultant d'une sous-requête plutôt qu'à des tables existantes dans la base de données. Le mot-clé **AS** prend alors tout son sens.

Par exemple, pour compter le nombre de couples de noms et prénoms attribués on peut faire :

SQL
```
SELECT COUNT(*)
FROM (SELECT DISTINCT nom, prenom FROM personne);
```

5.2 Sous-requêtes dans une clause `WHERE`

On peut utiliser les résultats de sous-requêtes dans la condition d'une sélection. Les attributs, les tables et les alias de la requête englobante sont disponibles dans la sous-requête.

5.2.1 Sous-requête avec un seul résultat

Lorsqu'une sous-requête ne renvoie qu'un seul résultat, en général celui d'une fonction d'agrégation, on peut directement utiliser ce résultat avec les opérateurs de comparaison.

Par exemple, pour trouver les noms des classes qui réalisent le plus petit effectif, on peut écrire :

SQL
```
SELECT nom FROM classe
WHERE effectif = (SELECT MIN(effectif) FROM classe);
```

Pour avoir les professeurs qui ont enseigné plus longtemps que la moyenne des numéros de classe (quelle question !) :

SQL
```
SELECT * FROM prof
WHERE anciennete >= (SELECT AVG(no) FROM classe);
```

5.2.2 Sous-requête avec plusieurs résultats

On utilise alors `IN` ($\in$) ou `NOT IN` ($\notin$). Le résultat de la sous-requête ne doit comporter qu'une seule colonne.

Par exemple, pour avoir l'ancienneté des professeurs qui ne sont pas professeurs principaux d'une classe, on peut écrire :

SQL
```
SELECT anciennete FROM prof
WHERE id NOT IN (SELECT prof_princ FROM classe);
```

Remarquons que l'on peut aussi écrire cette requête avec une jointure et une différence ensembliste.

SQL
```
SELECT anciennete FROM prof
EXCEPT
SELECT anciennete FROM prof JOIN classe on id = prof_princ;
```

☞ **Remarque 1**

On peut aussi utiliser `IN` pour tester l'appartenance à une séquence. Par exemple pour trouver les numéros des enseignants de mathématiques, informatique ou philosophie, on peut écrire :

SQL
```
SELECT id_prof FROM enseigne
WHERE matiere IN ('Maths', 'Info', 'Philo');
```

5.2.3 Existence

On peut également tester l'existence avec les opérateurs `EXISTS` ($\exists$) et `NOT EXISTS` ($\nexists$). Pour `EXISTS` par exemple, le résultat est vrai si la sous-requête contient au moins une ligne. La sous-requête peut faire référence à des variables de la requête englobante, qui agissent comme des constantes à chaque évaluation de la sous-requête.

Par exemple, pour avoir l'ancienneté des professeurs qui ne sont pas professeurs principaux d'une classe, on peut aussi écrire :

SQL
```
SELECT anciennete FROM prof p
WHERE NOT EXISTS (SELECT * FROM classe
                  WHERE prof_princ = p.id);
```

☞ **Remarque 2**

Pour `EXISTS (...)`, on peut aussi utiliser `0 != (SELECT COUNT(*) ...)`.

Étudions différentes manières de chercher le nom des élèves n'étant pas les seuls à être issus d'un lycée.

- Avec jointure :

```sql
SELECT nom FROM personne p
  JOIN eleve e1 ON p.id = e1.id
  JOIN eleve e2 ON e1.lycee_origine = e2.lycee_origine
WHERE e1.id != e2.id;
```

- Avec produit cartésien :

```sql
SELECT nom FROM personne p, eleve e1, eleve e2
WHERE p.id = e1.id AND e1.lycee_origine = e2.lycee_origine
  AND e1.id != e2.id;
```

- Avec **EXISTS** :

```sql
SELECT nom FROM personne p JOIN eleve e ON p.id = e.id
WHERE EXISTS (SELECT * FROM eleve
              WHERE lycee_origine = e.lycee_origine
                AND id != e.id);
```

- Avec **COUNT** :

```sql
SELECT nom FROM personne p JOIN eleve e ON p.id = e.id
WHERE 0 != (SELECT COUNT(*) FROM eleve
            WHERE lycee_origine = e.lycee_origine
              AND id != e.id);
```

- Avec **IN** :

```sql
SELECT nom FROM personne p JOIN eleve e ON p.id = e.id
WHERE e.lycee_origine IN (SELECT lycee_origine FROM eleve
                          WHERE id != e.id);
```

6 ORDER BY, LIMIT ET OFFSET

La clause **ORDER BY** permet de trier les résultats par ordre croissant (**ASC**) ou décroissant (**DESC**). Le mot-clé **ASC** est facultatif. La clause **ORDER BY** vient après la clause **WHERE**. Par exemple, pour avoir les classes triées par effectif décroissant :

```sql
SELECT * FROM classe ORDER BY effectif DESC;
```

On peut trier successivement selon plusieurs attributs. Le tri se fait d'abord selon les premiers, puis selon les suivants en cas d'égalité des premiers. Par exemple pour avoir les identifiants des personnes, sauf celles de nom *Galois*, triés suivant le nom de manière croissante, puis suivant le prénom de manière décroissante on peut écrire :

```sql
SELECT * FROM personne
WHERE nom != 'Galois'
ORDER BY nom ASC, prenom DESC;
```

La clause **LIMIT** n permet de se limiter aux n premiers tuples.

La clause **LIMIT** n **OFFSET** m permet de se limiter au premiers n tuples après un décalage de m, c'est-à-dire après élimination des m premiers, ou encore autrement dit ceux entre le $(m+1)^e$ et le $(m+n)^e$. Cette clause vient après un **ORDER BY**.

Par exemple, pour avoir la personne avec le deuxième plus petit identifiant on peut faire :

```sql
SELECT * FROM personne ORDER BY id LIMIT 1 OFFSET 1;
```

7 Groupement avant agrégation

Quelles sont les classes dans lesquelles enseignent le plus de professeurs ? Pour répondre à cette question il suffirait de compter le nombre de professeurs qui enseignent devant une classe donnée, ce que l'on peut faire avec la fonction d'agrégation **COUNT**. Puis, on peut chercher le maximum de ces nombres avec la fonction d'agrégation **MAX**. Mais comment obtenir le nombre de professeurs pour *chaque* classe ? Il faudrait répéter l'opération pour chaque classe, puisque les fonctions d'agrégation s'appliquent sur *toutes* les lignes. De la même manière, comment obtenir le nombre de classes *par* professeur, et ce pour *chaque* professeur ?

7.1 GROUP BY

Un groupe est un sous-ensemble de lignes d'une table avec la même valeur pour un ou plusieurs attributs donnés. Par exemple, si l'on choisit l'attribut *nom* de la table **personne**, on forme, pour chaque nom de famille, un groupe avec toutes les personnes qui partagent ce nom de famille :

id	nom	prenom	tel
54	Beauval	Elise	0678951354
33	Galois	Evariste	0699999999
75	Dupont	Pierre	0645456789
34	Durand	Sylvie	0612253694
81	Martin	Pierre	0635147630
17	Martin	Pierre	NULL
44	Martin	Filou	0612345678

On peut ensuite, par exemple, projeter suivant le nom d'une personne, ou suivant un ou plusieurs attributs qui doivent alors avoir la même valeur au sein de chaque groupe. Cela ne fait pas sens de demander la valeur du prénom pour chaque groupe par exemple. Dans la table résultante, chaque regroupement va se traduire par une ligne.

En SQL, la clause **GROUP BY**, permet d'effectuer ce regroupement :

SQL
```
SELECT nom
FROM personne
GROUP BY nom;
```

nom
Beauval
Dupond
Durand
Galois
Martin

Ici, on regroupe les lignes suivant le nom puis on projette, pour chaque groupe, ce nom. On obtient donc une colonne avec une ligne pour chaque groupe : tous les noms, sans doublons. Cela revient ici à simplement utiliser **DISTINCT**.

Cela devient particulièrement intéressant lorsque l'on fait intervenir les fonctions d'agrégations, qui s'appliquent alors sur les sous-lignes du groupe uniquement. On peut alors par exemple compter le nombre de cours d'un enseignant, par enseignant :

SQL
```
SELECT id_prof, COUNT(*) as nb_cours
FROM enseigne GROUP BY id_prof;
```

id_prof	nb_cours
34	1
75	1
81	2

Par exemple, pour obtenir, par professeur, le nombre de classes dans lesquelles il intervient au moins une fois :

SQL
```
SELECT id_prof, COUNT(DISTINCT no_classe) AS nb_classes
FROM enseigne
GROUP BY id_prof;
```

On peut faire un regroupement suivant plusieurs attributs. Dans ce cas, le regroupement est effectué pour tous les tuples qui partagent la même valeur pour tous les attributs spécifiés. Par exemple, pour avoir les couples de noms et prénoms avec leur nombre d'attribution, on peut faire :

SQL
```
SELECT nom, prenom, COUNT(*) AS nb
FROM personne
WHERE nom LIKE '%n%'
GROUP BY nom, prenom;
```

nom	prenom	nb
Dupont	Pierre	1
Durand	Sylvie	1
Martin	Pierre	2
Martin	Filou	1

On a ajouté une sélection des noms qui contiennent la lettre 'n', ce qui permet de voir que la clause **GROUP BY** doit se placer après la clause **WHERE**.

7.2 HAVING

On souhaite l'ensemble des couples noms prénoms qui sont attribués au moins deux fois. On peut utiliser la sous-requête précédente puis faire une sélection :

SQL
```sql
SELECT nom, prenom FROM
  (SELECT nom, prenom, COUNT(*) AS nb
  FROM personne
  WHERE nom LIKE '%n%'
  GROUP BY nom, prenom)
WHERE nb >= 2;
```

La clause **HAVING** permet plus directement de sélectionner certains groupes. Elle agit comme la clause **WHERE**, mais après le groupement. Elle doit se placer après la clause **GROUP BY**.

Par exemple, pour la requête précédente, on peut écrire :

SQL
```sql
SELECT nom, prenom
FROM personne
WHERE nom LIKE '%n%'
GROUP BY nom, prenom
HAVING COUNT(*) >= 2;
```

Insistons, la sélection de la clause **WHERE** s'effectue *avant* le regroupement et celle de la clause **HAVING** *après* le regroupement. Le prédicat dans la clause **HAVING**, tout comme la projection après regroupement, ne peut porter que sur des caractéristiques du groupe :

- Soit des expressions sur des attributs figurant dans la clause **GROUP BY** ou dont les valeurs sont identiques pour tous les tuples du groupes ;
- Soit des fonctions d'agrégation.

7.3 Agrégats de double niveau

On peut maintenant répondre à la question initiale : quelles sont les classes dans lesquelles enseignent le plus de professeurs ?

On commence par associer, pour chaque classe, le nombre de professeurs (différents !) qui y enseignent.

SQL
```sql
SELECT no_classe, COUNT(DISTINCT id_prof) AS nb_profs
FROM enseigne
GROUP BY no_classe;
```

On utilise cette requête comme sous-requête pour chercher le nombre maximal de professeurs qui enseignent dans une même classe :

SQL
```sql
SELECT MAX(nb_profs) FROM (
  SELECT no_classe, COUNT(DISTINCT id_prof) AS nb_profs
  FROM enseigne
  GROUP BY no_classe);
```

On cherche ensuite les classes qui réalisent ce maximum :

SQL
```sql
SELECT no_classe FROM enseigne
GROUP BY no_classe
HAVING COUNT(DISTINCT id_prof) = (
  SELECT MAX(nb_profs) FROM (
    SELECT no_classe, COUNT(DISTINCT id_prof) AS nb_profs
    FROM enseigne
    GROUP BY no_classe));
```

7.4 Exercices

1. Déterminer le nombre moyen d'élèves de chaque professeur qui en a.
2. Déterminer le nombre moyen de chaque professeur, qu'il ait ou non des élèves.
3. Donner le nom de chaque professeur qui enseigne dans au moins deux classes.
4. Donner la moyenne du nombre total d'élèves par professeur qui a des élèves, sans compter bien sûr plusieurs fois la même classe si le professeur enseigne plusieurs matières dans une même classe.

8 Division cartésienne en SQL

La division cartésienne ne fait actuellement ni partie de la norme SQL, ni de la plupart des implémentations de SGDB. Pourtant, nous avons vu qu'un tel opérateur est très utile pour répondre à une question comme *quels sont les noms qui sont associés à tous les prénom ?* Nous avions traduit cela dans le langage de l'algèbre relationnelle par $\Pi_{nom,prenom}(personne) \div \Pi_{prenom}(personne)$.

Notons $\cdot$ la concaténation d'attributs ou de tuples. Soit $\mathcal{R}$ une relation de schéma $S = X \cdot Y$ et $\mathcal{R}'$ une relation (non vide) de schéma Y. On a

$$\mathcal{R} \div \mathcal{R}' = \{t \in \Pi_X(\mathcal{R}) \mid \forall t' \in \mathcal{R}', t \cdot t' \in \mathcal{R}\} \tag{1}$$

L'expression (1) nous permet de reformuler le problème de la manière suivante : *quels sont les noms qui vérifient que quel que soit le prénom, le nom est associé avec ce prénom ?*

Le quantificateur existentiel $\forall$ n'existe pas non plus en SQL. En revanche, on dispose du quantificateur existentiel $\exists$. La logique des prédicats nous donne l'équivalence $\forall x, P(x) \iff \nexists\, x \neg P(x)$. On a donc

$$\mathcal{R} \div \mathcal{R}' = \{t \in \Pi_X(\mathcal{R}) \mid \nexists\, t' \in \mathcal{R}', t \cdot t' \notin \mathcal{R}\} \tag{2}$$

On peut donc reformuler le problème avec (2) de la manière suivante : *quels sont les noms pour lesquels il n'existe pas de prénom qui n'y soit pas associé ?*

8.1 Traduction avec `NOT EXISTS` et `NOT IN`

Remarquons que pour $t \in \Pi_X(\mathcal{R})$ et $t' \in \mathcal{R}'$, $t \cdot t' \in \mathcal{R} \iff t' \in \Pi_Y \circ \sigma_{X=t}(\mathcal{R})$. On a donc

$$\mathcal{R} \div \mathcal{R}' = \{t \in \Pi_X(\mathcal{R}) \mid \nexists\, t' \in \mathcal{R}', t' \notin \Pi_Y \circ \sigma_{X=t}(\mathcal{R})\} \tag{3}$$

L'expression (3) se traduit par *quels sont les noms pour lesquels il n'existe pas de prénom qui n'appartient pas aux prénoms associés à ce nom ?*

Cela, on peut le traduire directement en SQL !

SQL

```
SELECT DISTINCT p1.nom FROM personne p1 WHERE NOT EXISTS
  (SELECT * FROM personne p2 WHERE p2.prenom NOT IN
    (SELECT p3.prenom FROM personne p3 WHERE p1.nom = p3.nom));
```

8.2 Traduction avec double `NOT EXISTS`

Pour $t \in \Pi_X(\mathcal{R})$ et $t' \in \mathcal{R}'$ on a aussi $t \cdot t' \in \mathcal{R} \iff \exists t'' \in \mathcal{R}, t'' = t \cdot t'$. On a donc

$$\mathcal{R} \div \mathcal{R}' = \{t \in \Pi_X(\mathcal{R}) \mid \nexists\, t' \in \mathcal{R}', \nexists\, t'' \in \mathcal{R}, t'' = t \cdot t'\} \tag{4}$$

L'expression (4) se traduit par *quels sont les noms pour lesquels il n'existe pas de prénom pour lequel il n'existe pas d'association qui lie ce nom à ce prénom ?* En SQL :

SQL

```
SELECT DISTINCT p1.nom FROM personne p1 WHERE NOT EXISTS
  (SELECT * FROM personne p2 WHERE NOT EXISTS
    (SELECT * FROM personne p3 WHERE p1.nom = p3.nom AND
                                     p2.prenom = p3.prenom));
```

8.3 Traduction avec `COUNT`

Le quantificateur existentiel $\forall$ n'existe pas en SQL, mais on peut remarquer que $\forall x \in \mathcal{X}, P(x) \iff |\mathcal{X}| = |\{x \in \mathcal{X} \mid P(x)\}|$. Autrement dit pour vérifier que tous les éléments d'un ensemble vérifient un propriété, on peut vérifier que le nombre des éléments qui la vérifient est le cardinal de l'ensemble lui-même. On a donc

$$\mathcal{R} \div \mathcal{R}' = \{t \in \Pi_X(\mathcal{R}) \mid |\mathcal{R}'| = |\{t' \in \mathcal{R}' \mid t \cdot t' \in \mathcal{R}\}|\} \tag{5}$$

L'expression (5) se traduit par *quels sont les noms pour lesquels le nombre de prénoms associés à ce nom est exactement le nombre de tous les prénoms ?* En SQL :

SQL

```
SELECT DISTINCT p1.nom FROM personne p1 WHERE
   (SELECT COUNT(DISTINCT prenom) FROM personne) =
   (SELECT COUNT(DISTINCT p2.prenom ) FROM personne p2
    WHERE p1.nom = p2.nom);
```

8.4 Traduction avec `GROUP BY` et `HAVING`

On peut aussi grouper les prénoms par nom, puis ne sélectionner que les groupes pour lesquels le nombre de prénoms est le nombre total de prénoms possibles.

SQL

```
SELECT nom FROM personne
GROUP BY nom
HAVING COUNT(DISTINCT prenom) =
            (SELECT COUNT(DISTINCT prenom) FROM personne);
```

C'est peut-être la manière la plus simple de réaliser la division cartésienne.

8.5 Traduction avec les autres opérateurs

On a

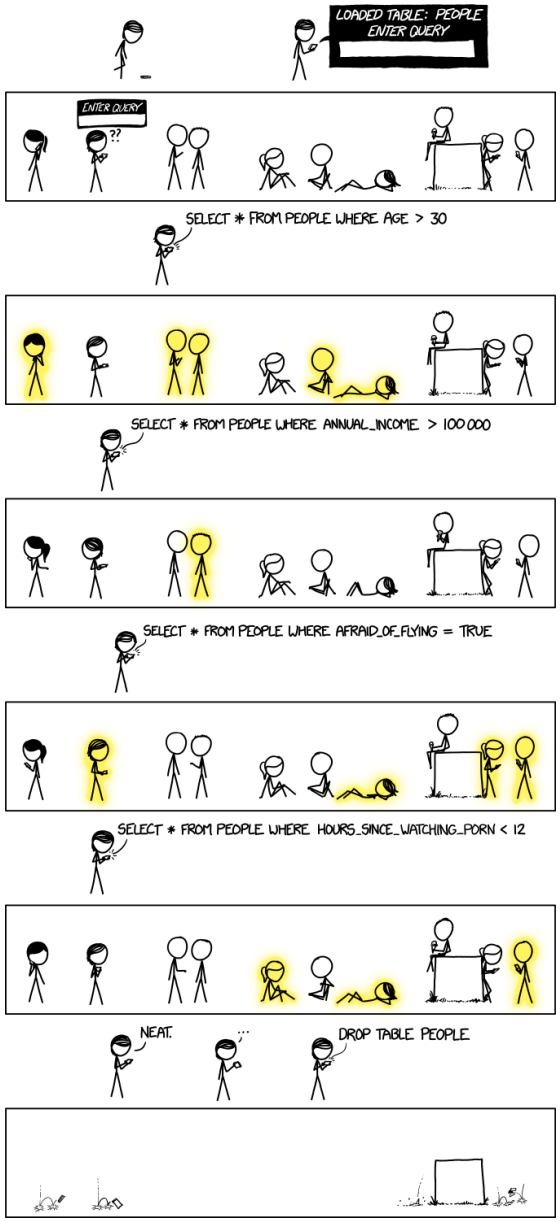
$$\mathcal{R} \div \mathcal{R}' = \{t \in \Pi_X(\mathcal{R}) \mid \nexists\, t' \in \mathcal{R}', t \cdot t' \notin \mathcal{R}\} \tag{6}$$
$$= \Pi_X(\mathcal{R}) \setminus \{t \in \Pi_X(\mathcal{R}) \mid \exists t' \in \mathcal{R}', t \cdot t' \notin \mathcal{R}\} \tag{7}$$
$$= \Pi_X(\mathcal{R}) \setminus \{t \in \Pi_X(\mathcal{R}) \mid \exists t' \in \mathcal{R}', t \cdot t' \in \Pi_X(\mathcal{R}) \times \mathcal{R}' \setminus \mathcal{R}\} \tag{8}$$
$$= \Pi_X(\mathcal{R}) \setminus \Pi_X(\Pi_X(\mathcal{R}) \times \mathcal{R}' \setminus \mathcal{R}) \tag{9}$$

Une division cartésienne s'exprime donc en combinant des projections, des différences ensemblistes et un produit cartésien. Pour l'exemple ci-dessus, l'expression (9) se comprend comme suit. Pour avoir les noms qui sont associés à tous les prénoms, on commence par associer tous les noms et tous les prénoms possibles. Puis on retranche les associations entre un nom et un prénom présentes dans la relation. Les noms qui apparaissent alors, ce que l'on obtient par projection, sont donc tous les noms qui ne sont pas associés à tous les prénoms. Il nous reste à ne considérer que les autres.

SQL

```
SELECT nom FROM personne
EXCEPT
 SELECT nom FROM
  (SELECT * FROM
        (SELECT nom FROM personne),
        (SELECT prenom FROM personne)
   EXCEPT
   SELECT nom, prenom FROM personne);
```

C'est souvent assez compliqué à réaliser.



https://xkcd.com/1409/