Langage SQL

Ecrire des requêtes à l'aide du langage SQL

aelion

Sommaire

- ❏ Comprendre les bases des SGBDR
- ❏ Mettre en oeuvre les opérations de CRUD
- ❏ Maîtriser les requêtes fondamentales
- ❏ Découvrir les fonctions avancées
  - ❏ Agrégations
  - ❏ Fenêtrage

Concepts fondamentaux

Le modèle relationnel

- Relations => Tables
- Attributs => Colonnes
- Occurrences (Tuples / Enregistrements) => lignes
- Clés primaires
- Clés étrangères

id | lastname | firstname | salary

---

1 | Dupont | Robert | 20000

2 | Durand | Aline | 8500

La clé primaire

Une clé primaire (PK) est une colonne (ou un ensemble de colonnes) qui identifie une ligne parmi les n lignes de la table.
La clé primaire est donc **unique** et **non nulle**.

Dans le monde réel, une clé primaire peut être un numéro de plaque d'immatriculation, ou un numéro INSEE d'une personne.

id	lastname	firstname	salary
1	Dupont	Robert	20000
2	Durand	Aline	8500

aelion

Les clés étrangères

On définit la clé étrangère (FK) comme étant une colonne dans une table qui référence la valeur de la *clé primaire* (PK) d'une autre table.

Des contraintes peuvent être appliquées sur les clés étrangères afin de garantir l'intégrité de la base de données.

id	lastname	firstname	salary	***service_id***
1	Dupont	Robert	20000	1
2	Durand	Aline	8500	3

Table : service

id	name
1	Accounting
2	Marketing
3	Sales

La colonne "**service_id**" (FK) de la table "employee" référence la colonne "**id**" (PK) de la table "service".

aelion

Types de données standard

Numériques	INT, DECIMAL, FLOAT	Le séparateur décimal est le point “.”
Chaînes	VARCHAR, CHAR, TEXT	Toujours encadrés de “single quotes” (‘…’).
Dates	DATE, DATETIME, TIME	Le format des dates est “AAAA-MM-JJ”
Autres	BLOB	Destiné à stocker des données binaires
Spéciaux	JSON, POINT, …	Voir la documentation de votre SGBDR

Le CrUD (Create Read Update Delete)

Insérer des lignes

Le prototype de la requête permettant d'ajouter une ligne dans une table est le suivant :

INSERT INTO table_name (col1, col2, ..., coln) **VALUES** (val1, val2, ..., valn);

Par exemple :

INSERT INTO employee (id, lastname, firstname, salary) **VALUES** (3, 'Martin', 'Paul', 2500);

Insérer des lignes

Le langage SQL permet d'insérer en une seule requête plusieurs lignes :

INSERT INTO employee **VALUES** (4, 'Talut', 'Jean', 1950), (5, 'Lefort', 'Paul', 2100), (6, 'Dujardin', 'Martine', 2500);

Note : *ici, les colonnes ont été omises car les données respectent le nombre et l'ordre des colonnes de la table, ainsi que les éventuelles contraintes et types définis.*

Gérer l’apostrophe dans les chaînes

Attention, si dans une chaîne de caractères, vous devez insérer une apostrophe (single quote), il faudra penser à l’échapper avec le caractère ‘\’ ou à doubler l’apostrophe.

Cas spécial la valeur “null”

La valeur “null” est spéciale, elle désigne une valeur pour une colonne qui n’aurait jamais été définie.
Il ne s’agit pas d’une valeur vide :

INSERT INTO employee (id, lastname, firstname, salary) **VALUES** (3, ‘Martin’, ‘’, 2500);
N’est pas identique à :
INSERT INTO employee (id, lastname, firstname, salary) **VALUES** (3, ‘Martin’, NULL, 2500);

Mettre à jour des lignes

Le prototype de la requête permettant mettre à jour des lignes dans une table est le suivant :

UPDATE nom_table **SET** col1 = new_value, col2 = new_value [**WHERE** id = value];

Par exemple :

UPDATE employee **SET** lastname = ‘Durand’, salary = 2350 **WHERE** id = 1;

Mettre à jour des lignes

Attention : sans clause WHERE dans une requête de mise à jour, c'est l'ensemble des lignes de la table qui est mis à jour :

UPDATE employee **SET** lastname = 'Durand', salary = 2350;

Cette requête mettra à jour la table "employee" en remplaçant **TOUS** les noms par Durand et les salaires à 2350 !

L'utilisation de la clause **WHERE** est fortement recommandée.

Mettre à jour des lignes

Il est tout à fait possible d'utiliser, dans les requêtes SQL, des calculs. Par exemple, si vous souhaitez augmenter TOUS vos salariés de 3%, vous pouvez écrire la requête suivante :

UPDATE employee **SET** salary = salary * 1.03;

Supprimer des lignes

Le prototype de la requête permettant la suppression de lignes dans une table est le suivant :

DELETE FROM nom_table [**WHERE** id = value];

Par exemple :

DELETE FROM employee **WHERE** id = 1;

Attention, la requête **DELETE** devrait toujours être accompagnée d'une clause WHERE pour éviter la suppression de TOUTES les lignes.

Supprimer des lignes

Considérations spéciales :

Une requête **DELETE** peut échouer si les contraintes d'intégrités référentielles ne sont pas respectées.

Le mot-clé **CASCADE** peut être utilisé pour supprimer à la fois les lignes d'une table ainsi que toutes celles qui sont référencées dans les autres tables.

Il est préférable d'utililser une requête **DELETE** au sein d'une transaction.

Supprimer des lignes

Le mot-clé **TRUNCATE** peut aussi être utilisé pour supprimer des lignes dans une ou plusieurs tables.

TRUNCATE [**CASCADE**] table_name;

TRUNCATE vide complètement une table, il n'est pas possible d'y ajouter une clause **WHERE**, toutes les lignes seront supprimées ; c'est l'équivalent de :

DELETE [**CASCADE**] **FROM** table_name;

TRUNCATE ne peut pas être restauré avec un **ROLLBACK**.

Lire des données

SELECT

La requête **SELECT** permet de lire des lignes d'un ensemble de données (tables, vues, ...).

SELECT [**DISTINCT**] col1, col2, ..., coln
FROM ensembles
WHERE conditional_expression
GROUP BY (agregats)
HAVING (conditional_expression)
ORDER BY col1, col2 [**DESC**]
LIMIT row_limitations

SELECT

La ***projection*** identifie les informations, séparées par une virgule, que l'on souhaite récupérer et lister. La ***projection*** se situe immédiatement après le **SELECT** et avant le **FROM** :

SELECT * **FROM** employees;

Cas spécifique "*" : le caractère "*" permet de définir *TOUTES* les colonnes de l'ensemble à "projeter", en l'occurrence, dans l'exemple, toutes les colonnes de la table "employees", cette requête est donc identique à :

SELECT id, lastname, firstname, salary, service_id **FROM** employees;

SELECT : alias de colonne

Les “alias” de colonnes peuvent être utilisées pour différentes raison, la première est de rendre plus lisible un résultat :

SELECT lastname nom, firstname prénom **FROM** employees;

nom	prénom
Dupont	Robert
Durand	Aline

Le mot-clé “AS” peut être utilisé, mais n’est plus requis pour définir un alias, seul un espace entre la colonne de la projection et son alias suffit.

aelion

SELECT : alias de colonne

Les alias sont souvent utilisés pour les colonnes calculées dans la projection.

SELECT lastname, firstname, (salary * (1 - 40 / 100)) “net salary”
FROM employees;

lastname	**firstname**	**net salary**
Dupont	Robert	1800
Durand	Aline	2100

Il est recommandé d’utiliser les alias de colonnes lors de la création de vues (voir plus loin).

SELECT : alias de table

Il est recommandé d'utiliser les alias dans les **jointures** (voir plus loin) pour éviter les ambiguïtés sur les noms de colonnes et/ou faciliter la lecture :

```
SELECT e.name, e.firstname, s.name
FROM employees e JOIN service s ON e.service_id = s.id;
```

L'alias "**e**" identifie la table "employees" ; l'alias "**s**" identifie la table "service".
On utilise généralement les initiales des tables en tant qu'alias.
Une fois la table aliasée, il est **obligatoire** d'utiliser l'alias pour préfixer les colonnes.

SELECT : Trier le résultat

SQL offre la clause de tri ORDER BY pour trier le résultat selon une ou plusieurs colonnes, avec la possibilité de tri ascendant (par défaut) ou descendant (DESC)

```
SELECT e.name, e.firstname
FROM employees e ORDER BY e.name;
```

lastname	firstname
Dujardin	Paul
Dupont	Robert
Durand	Aline

Le résultat sera trié dans l'ordre alphabétique croissant.
Le mot clé ASC n'est pas requis.

aelion

SELECT : Trier le résultat

Le mot clé **DESC** utiliser dans le **ORDER BY** permet de changer l'ordre de tri de manière descendante (du plus grand au plus petit) :

```
SELECT e.name, e.firstname
FROM employees e ORDER BY e.name DESC;
```

lastname	**firstname**
Durand	Aline
Dupont	Robert
Dujardin	Paul

SELECT : Trier le résultat

La clause de tri permet de trier sur de multiples colonnes (présentes ou non dans la projection). La première colonne sera le premier ordre, les suivantes seront dépendantes de l'ordre immédiatement précédent.

SELECT e.name, e.firstname, e.birthdate
FROM employees e **ORDER BY** e.name, e.birthdate DESC;

lastname	**firstname**	**birthdate**
Durand	***Aline***	***1998-12-03***
Durand	***Aline***	***1987-05-26***
Dupont	Robert	1975-02-16
Dujardin	Paul	1987-10-23

Les lignes sont triées en premier lieu sur le nom, puis, les deux premières sont triées selon la date de naissance.

aelion

Exercices : aliases, tris

A partir de la table “communes” :

- Lister les noms des communes ainsi que leur superficie en triant le résultat dans l’ordre décroissant des superficies,
- Lister les noms des communes, la superficie, le nombre d’habitants et la densité, en ajoutant des alias de colonnes pour la lisibilité, en triant les communes dans l’ordre croissant des densités,
- Lister les noms des communes, le nombre d’habitants, la superficie en triant le résultat dans l’ordre décroissant du nombre d’habitants, puis dans l’ordre croissant des superficies.

SELECT : Filtrer le résultat

Le filtrage des lignes est géré par la clause **WHERE** dans une requête. **WHERE** sert à restreindre le nombre de lignes retournées, en fonction de l'expression "*booléenne*" fournie.

```
SELECT e.name, e.firstname, e.birthdate
FROM employees e WHERE e.name = 'Durand';
```

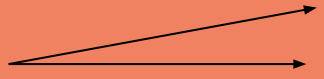
lastname	firstname	birthdate
Durand	*Aline*	*1998-12-03*
Durand	*Aline*	*1987-05-26*

aelion

SELECT : Filtrer le résultat

Expression “*booléenne*”
Une expression booléenne est une expression qui doit retourner une valeur “**vraie**” ou “**fausse**”.
Si l’expression est vraie, la ligne du **SELECT** est conservée dans le résultat.

WHERE e.name = ‘Durand’



lastname	**firstname**	**birthdate**
Durand	***Aline***	***1998-12-03***
Durand	***Aline***	***1987-05-26***
Dupont	Robert	1975-02-16
Dujardin	Paul	1987-10-23

SELECT : Filtrer le résultat

Opérateurs booléens : les opérateurs classiques

=	Egalité stricte : **e.name = 'Durand'**
>	Supériorité stricte : **e.birthdate > '1980-01-01'**
<	Infériorité stricte : **e.name < 'T'**
>=	Supériorité inclusive : **e.salary >= 1500**
<=	Infériorité inclusive : **e.birthdate <= '1980-01-01'**
<> ou !=	Différent strictement : **e.nom <> 'Dupont'**

SELECT : Filtrer le résultat

Algèbre de Boole : OR, AND et XOR

Expr 1	Expr 2	AND	OR	XOR
FAUX	FAUX	**FAUX**	**FAUX**	**FAUX**
FAUX	VRAI	**FAUX**	**VRAI**	**VRAI**
VRAI	FAUX	**FAUX**	**VRAI**	**VRAI**
VRAI	VRAI	**VRAI**	**VRAI**	**FAUX**

SELECT : Filtrer le résultat

Opérateur SQL : IN

L'opérateur **IN** permet de déterminer l'appartenance d'une valeur à un ensemble. On utilise souvent **IN** avec une sous-requête :

```
SELECT e.lastname, e.firstname
FROM employees e
WHERE e.service_id IN (SELECT id FROM services WHERE name
= 'Accounting' OR name = 'Sales');
```

SELECT : Filtrer le résultat

Opérateur SQL : BETWEEN
L'opérateur **BETWEEN** permet de déterminer l'appartenance d'une valeur dans un intervalle. On utilise **BETWEEN** souvent avec les dates et les nombres :

```
SELECT e.lastname, e.firstname
FROM employees e
WHERE e.birthdate BETWEEN '1980-01-01' AND '1990-12-31;
```

***BETWEEN** est inclusif, les bornes sont incluses dans le résultat*

SELECT : Filtrer le résultat

Opérateur SQL : LIKE

L'opérateur **LIKE** permet de déterminer l'appartenance d'une valeur à un ensemble de valeurs "approchantes".

e.lastname LIKE 'Dupon%'	*Commence par* 'Dupon' quelque soit la chaîne qui suit
e.lastname LIKE '%nd'	*Termine par* 'nd' quelque soit la chaîne qui précède
e.lastname LIKE '%up%'	*Contient* 'up' quelque soit la chaîne avant et après
e.lastname LIKE 'Dupon_'	Commence par 'Dupon' quelque soit le caractère qui suit

SELECT : Filtrer le résultat

Opérateur SQL : LIKE
Il est nécessaire de vérifier le comportement de l'opérateur **LIKE** en fonction du SGBD utilisé.

MySQL et *MariaDB* sont insensibles à la casse :

LIKE 'DuPo%' sera accepté pour toutes les formes de casse.

PostgreSQL étant sensible à la casse, il est nécessaire de lire la documentation pour obtenir le même résultat.

SELECT : Filtrer le résultat

Gérer les valeurs nulles dans les filtres
La valeur NULL est particulière en SQL. Il ne s'agit pas d'une valeur à proprement parler, il est donc nécessaire d'utiliser d'autres opérateurs : **IS [NOT] NULL**

SELECT e.lastname, e.firstname **WHERE** salary **IS NULL**;
SELECT e.lastname, e.firstname **WHERE** salary **IS NOT NULL**;

Cette syntaxe est beaucoup utilisée dans les jointures externes (voir plus loin).

Exercices : filtres

A partir de la table “communes” :

- Lister le nom, la population, des communes dont la population est supérieure à 500 000,
- Lister les communes dont le département est dans la liste : 03, 63, 43, 15,
- Lister le nom des communes dont la superficie est comprise entre 30 et 50 km carré,
- Lister les noms des villes dont la densité est inférieure à 10 personnes par km carré,
- Lister les villes dont le nom commence par ‘Saint’ et dont la population est inférieure à 10 000

SELECT : Jointures

Les jointures en SQL permettent de générer de nouveaux ensembles à partir de deux ou plusieurs autres ensembles.

La **jointure interne** s'exprime basiquement de la manière suivante :

```
SELECT e1.col1, e1.col2, ..., e2.coln
FROM
    ensemble1 e1 [INNER | LEFT | RIGHT | OUTER ] JOIN ensemble2 e2
ON
    join_condition;
```

SELECT : Jointures internes

Par exemple, la jointure interne entre service et employees :

```
SELECT e.lastname, e.firstname, s.name
FROM
    service s JOIN employee e ON s.id = e.service_id;
```

Une **jointure interne** inclut uniquement les lignes dans lesquelles une valeur pour le champ-clé est commune à toutes les tables d'entrée. Cela signifie que les lignes sans correspondance ne sont pas incluses dans le jeu de données de sortie.

Dans l'exemple, seules les lignes pour lesquelles l'égalité entre **s.id** et **e.service_id** seront retournées dans l'ensemble résultat.

SELECT : Jointures produit cartésien

Attention au **produit cartésien**. Si on omet de définir la condition de la jointure, le résultat sera perçu comme aberrant :

```
SELECT e.lastname, e.firstname, s.name
FROM
    service s JOIN employee e;
```

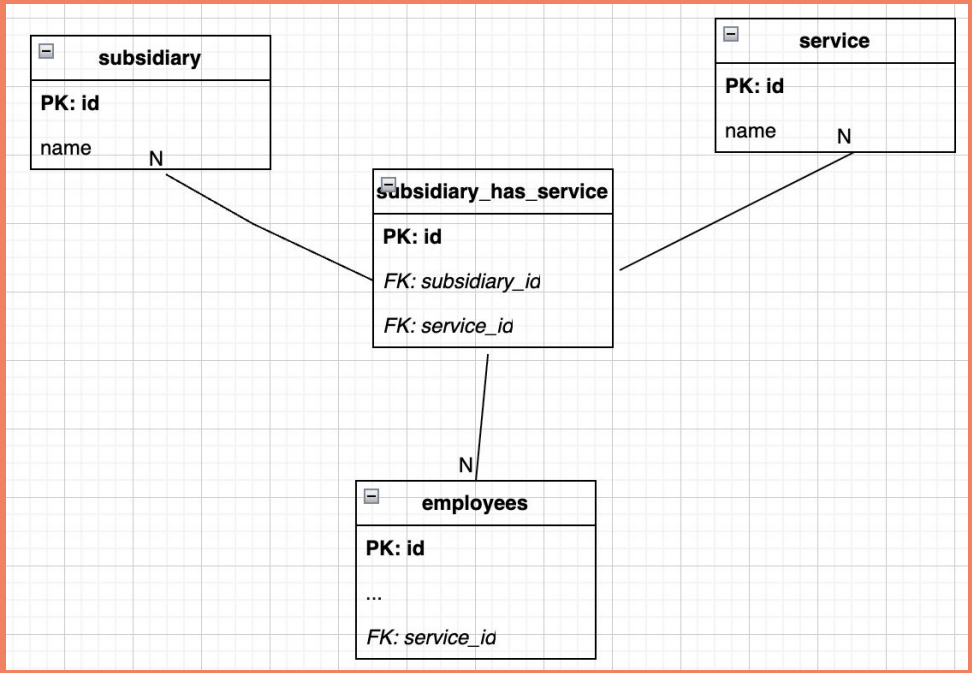
e.**lastname**	e.**firstname**	s.**name**
Durand	***Aline***	***Accouting***
Durand	***Aline***	***Sales***
Durand	***Aline***	***Marketing***

Contenu de la table employees

lastname	firstname	service_id
Durand	*Aline*	*1*

aelion

SELECT : Jointures multiples



Dans ce modèle, les filiales (subsidiary) dispose d'un ou plusieurs services. Cette relation se matérialise par la table d'association "subsidiary_has_service", les employés font partie d'un des services d'une filiale.

aelion

SELECT : Jointures multiples

Les jointures suivantes s'appliquent au modèle précédent et permettent de de récupérer les salariés ainsi que le service et la filiale dans laquelle ils opèrent :

```
SELECT e.lastname, e.firstname, sv.name, s.name
FROM
    subsidiary s
    JOIN subsidirary_has_service shs ON s.id = shs.subsidiary_id
    JOIN service sv ON shs.service_id = sv.id
    JOIN employees e ON e.service_id = shs.id;
```

Exercices : Jointures

A partir de la base de données Recensement :

- Lister le nom de la commune ainsi que le nom de son département, en triant le résultat sur le nom du département,
- Lister le nom des départements ainsi que le nom de la région, en triant le résultat par nom de région puis nom du département,
- Lister le nom de la commune, le nom du département, pour les communes dont la population dépasse 150 000 habitants,
- Lister les noms des villes, le nom du département, le nom de la région en triant le résultat par numéro de région, puis département, puis commune.

SELECT : Jointures externes

Les jointures externes permettent de récupérer les résultats d'un ensemble, qu'ils aient ou non une donnée associée dans un autre ensemble :

```
SELECT a.col1, a.col2, ..., b.col1
FROM
    ensembleA a LEFT | RIGHT JOIN ensembleB b
    ON a.id = b.ensembleA.id;
```

Le choix de **LEFT** ou **RIGHT** dépend de l'ensemble à partir duquel vous souhaitez conserver les données.

SELECT : Jointures externes

L'exemple suivant listera TOUS les services, qu'il y ait ou non des salariés :

```
SELECT s.name, e.lastname,
e.firstname
FROM
    service s LEFT JOIN employees e
    ON s.id = e.service_id;
```

s.name	e.lastname	e.firstname
Accounting	*Durand*	*Aline*
Sales	*Dupont*	*Paul*
Marketing	*NULL*	*NULL*

SELECT : Jointures externes

```
SELECT s.name, e.lastname,
e.firstname
FROM
    service s LEFT JOIN employees e
    ON s.id = e.service_id;
```

Ici, on dit que la jointure est à gauche (**LEFT**) car on souhaite conserver les lignes de l'ensemble "service", donc, de l'ensemble qui se situe à gauche de la jointure !

SELECT : Jointures externes

```
SELECT s.name, e.lastname, e.firstname
FROM
    employees e RIGHT JOIN service s
    ON s.id = e.service_id;
```

Ici, on dit que la jointure est à droite (**RIGHT**) car on souhaite conserver les lignes de l'ensemble "service", donc, de l'ensemble qui se situe à droite de la jointure !
Cette requête est strictement équivalente à la précédente...

Exercices : Jointures externes

Retrouvez de manière ludique un QCM relatifs aux jointures, internes, externes et la gestion des valeurs nulle :

Jointures externes et valeurs nulles

SELECT : Fonctions d'agrégation

Le langage SQL fournit 5 fonctions d'agrégations :

COUNT(x) : dénombre les lignes retournées par une requête SELECT,
SUM(exp) : cumule l'expression (exp) à partir des lignes retournées,
AVG(exp) : calcule la moyenne de de l'expression exp à partir des lignes retournées,
MIN(exp) : détermine la plus petite valeur exp à partir des lignes **retournées,**
MAX(exp) : détermine la plus grande valeur exp à partir des lignes

SELECT : Fonctions d'agrégation

Les fonctions dites d'agrégation ne retournent, dans une requête SQL simple, qu'une seule ligne. L'exemple suivant est donc incorrect :

SELECT e.lastname, **COUNT(*) FROM** employees e;

En effet, la machine ne saurait pas renvoyer (de manière simple), TOUS les noms de famille des salariés ainsi que le nombre de salariés ! Une erreur sera donc levée indiquant que vous ne pouvez pas utiliser une fonction d'agrégation dans ce contexte.

SELECT : Fonctions d'agrégation

La fonction **COUNT(exp)** retourne donc le nombre de lignes impactées par une requête SQL :

SELECT COUNT(*) nb_salaries **FROM** employees;

Cette requête va donc retourner le nombre de lignes total (*) de la table "employees".

SELECT COUNT(salary**)** nb_salaried_employees **FROM** employees;

Cette requête retournera le nombre de lignes de la table "employees" pour lesquelles la valeur de la colonne "salary" est NON NULLE.

SELECT COUNT(*) FROM employees **WHERE** service_id = 1; **<= ?**

aelion

SELECT : Fonctions d'agrégation

La fonction **SUM(exp)** retourne le cumul (la somme) de l'expression définie (exp) dépendant des lignes impactées :

SELECT SUM(salary**)** salary_mass **FROM** employees;

Retourne la masse salariale totale.

L'expression peut être un calcul :

SELECT SUM(salary * (1 - 40 / 100)**)** net_salary_mass **FROM** employees;

Retourne donc la masse salariale nette à payer.

SELECT : Fonctions d'agrégation

La fonction **AVG(exp)** retourne la moyenne de l'expression définie (exp) dépendant des lignes impactées :

SELECT AVG(salary**)** average_salary **FROM** employees;

Retourne donc la moyenne des salaires de la table "employees".

SELECT AVG(salary**)** salary_mass **FROM** employees **WHERE** salary >= 2000;

Retourne la moyenne des salaires des employés gagnant plus de 2000 €

SELECT : Fonctions d'agrégation

La fonction **MAX(exp)** retourne la valeur maximale de l'expression définie (exp) dépendant des lignes impactées :

SELECT MAX(salary**)** average_salary **FROM** employees;

Retourne le salaire le plus élevé de la table "employees".

SELECT MAX(salary**)** salary_mass **FROM** employees **WHERE** salary < 2000;

Retourne le salaire maximum des employés gagnant moins de 2000 €

aelion

SELECT : Fonctions d'agrégation

La fonction **MIN(exp)** retourne la valeur minimale de l'expression définie (exp) dépendant des lignes impactées :

SELECT MIN(salary**)** average_salary **FROM** employees;

Retourne le salaire le moins élevé de la table "employees".

SELECT MIN(salary**)** salary_mass **FROM** employees **WHERE** salary **>** 2000 **AND** service_id = 2;

Retourne le salaire minimum des employés gagnant plus de 2000 € dans le service dont le clé primaire vaut 2.

aelion

Exercices : Agrégations

A partir de la base Recensement :

- ❑ Afficher la plus petite population des communes de la base de données,
- ❑ Afficher la somme totale des populations de la base de données,
- ❑ Afficher la population moyenne de la base de données,
- ❑ Afficher la plus haute et la plus petite densité des communes.

SELECT : les regroupements

Il est nécessaire parfois d'agréger des lignes (les regrouper) pour disposer d'informations plus synthétiques, que les lignes brutes.
La clause **GROUP BY** permet de regrouper des lignes selon un ou plusieurs critères. Le **GROUP BY** est souvent utilisé avec les fonctions d'agrégations.
Le **GROUP BY** arrive toujours après le **WHERE** si appliqué :

SELECT col
FROM ensemble
[**WHERE** condition]
[**GROUP BY** group_exp];

SELECT : les regroupements

L'exemple suivant illustre un regroupement simple ; il s'agit de lister le nombre de salariés répartis par services, il met en oeuvre des *jointures* pour accéder au nom du service, la fonction **COUNT()** permettra de dénombrer les lignes :

```
SELECT sv.name service, COUNT(*) nb_employees
FROM
    service sv
    JOIN subsidiarie_has_service shs ON shs.service_id = sv.id
    JOIN employees e ON e.service_id = shs.id
GROUP BY sv.id;
```

SELECT : les regroupements

On peut filtrer les résultats après regroupement à l'aide de la clause **HAVING**

SELECT agregation_col ac
FROM ensemble
[**WHERE** cond_exp]
GROUP BY sv.id
HAVING group_cond_exp;

group_cond_exp sera l'expression utilisée pour filtrer les résultats **APRÈS** regroupement.

SELECT : les regroupements

Dans l'exemple suivant on souhaite juste récupérer les services dans lesquels il y a plus de 5 salariés :

```
SELECT sv.name service, COUNT(*) nb_employees
FROM
    service sv
    JOIN subsidiarie_has_service shs ON shs.service_id = sv.id
    JOIN employees e ON e.service_id = shs.id
GROUP BY sv.id
HAVING (nb_employees > 5); – On peut réutiliser l'alias à ce moment là
```

SELECT : les regroupements

Attention avec la fonction de regroupement COUNT() avec les jointures externes :

```
SELECT sv.name, COUNT(*) nb_employees
FROM
        service sv LEFT JOIN employees e
GROUP BY sv.id;
```

Cette requête listera TOUS les services qu'ils aient ou pas un employé rattaché.
Si un service ne dispose pas d'employé, la ligne sera :

sv.name | nb_employees
Marketing | 3
Sales | 1

SELECT : les regroupements

Attention avec la fonction de regroupement **COUNT()** avec les jointures externes :

```
SELECT
    sv.name,
    COUNT(employee_id) nb_employees
FROM
    service sv
        LEFT JOIN employees e
GROUP BY sv.id;
```

Cette requête listera uniquement les lignes pour lesquelles **employee_id** est non nul.
Si un service ne dispose pas d'employé, la ligne sera :

sv.name | nb_employees
Marketing | 3
Sales | 0

aelion

Exercices : Regroupement et filtres

A partir de la base Recensement :

- ❏ Afficher le département, la population totale de chaque département, trier les résultats du plus au moins peuplé,
- ❏ Afficher le département, la population totale, pour les départements dont la population totale dépasse 600 000,
- ❏ Afficher le département, la densité moyenne, pour les départements de la région Occitanie, trier le résultat par ordre croissant des densités,
- ❏ Afficher le département le plus peuplé de chaque région.

SELECT : UNION / INTERSECTION

SELECT : UNION

L'union est le concept ensembliste qui consiste à obtenir tous les éléments qui correspondent à la fois à l'ensemble A ou à l'ensemble B.
Concrètement, les ensembles mis en oeuvre doivent avoir le même nombre de colonnes, avec le même type et dans le même ordre.

```
SELECT col1, col2 col3 FROM ensemble1
UNION
SELECT col1, col2, col3 FROM ensemble2;
```

Par défaut, les lignes exactement identiques ne sont pas répétées dans le résultat.

SELECT : INTERSECT

L'intersection est le concept ensembliste qui consiste à récupérer les lignes communes entre plusieurs ensembles.
Concrètement, les ensembles mis en oeuvre doivent avoir le même nombre de colonnes, avec le même type et dans le même ordre.

```
SELECT col1, col2 col3 FROM ensemble1
INTERSECT
SELECT col1, col2, col3 FROM ensemble2;
```

Exercices : Regroupement et filtres

A partir de la base Recensement :

- ❏ Afficher le département, la population totale de chaque département, trier les résultats du plus au moins peuplé,
- ❏ Afficher le département, la population totale, pour les départements dont la population totale dépasse 600 000,
- ❏ Afficher le département, la densité moyenne, pour les départements de la région Occitanie, trier le résultat par ordre croissant des densités,
- ❏ Afficher le département le plus peuplé de chaque région.

SELECT : Fenêtrage

Les fonctions de fenêtrage ont été introduites en 2003 dans le langage SQL.
Un peu à la manière des fonctions d'agrégation, elles opèrent des calculs sur l'ensemble retourné, mais à l'inverse de l'agrégation, les lignes sont conservées dans le résultat.
Le mot-clé **OVER** désignera la fonction de regroupement comme fonction de "fenêtrage".

SELECT : Fenêtrage

L'exemple suivant illustre la manière d'utiliser une fonction comme fonction de fenêtrage :

```
SELECT
        lastname,
        firstname,
        salary,
        SUM(salary) OVER(ORDER BY lastname) salary_mass
FROM
        employees;
```

aelion

SELECT : Fenêtrage

```
SELECT
        lastname,
        firstname,
        salary,
        SUM(salary) OVER(ORDER BY lastname) salary_mass
FROM
        employees;
```

On peut lire cette requête comme : effectue la somme des salaires sur l'ensemble des employés, ainsi que les noms, prénoms et salaires triés dans l'ordre de leur nom de famille

SELECT : Fenêtrage

Il est possible d'utiliser **PARTITION BY** dans la fonction de fenêtrage **OVER** afin d'opérer un regroupement sur une des colonnes.

```
SELECT
    e.lastname,
    e.firstname,
    sv.name,
    e.salary,
    SUM(salary) OVER(PARTITION BY sv.id ORDER BY sv.name) s_salary_mass
FROM
    service sv JOIN employees e ON sv.id = e.service_id;
```

aelion

SELECT : Fenêtrage

Le résultat de la requête précédente donnera un résultat tel que le suivant :

Ici la fonction **SUM** avec fenêtrage 

e.lastname	e.firstname	sv.name	e.salary	s_salary_mass
Durant	Delphine	Marketing	2000	35700
Dupont	Jean	Marketing	1770	35700
Martin	Paul	Accounting	2500	2500
...	...	...	...	...

SELECT : Fenêtrage

La fonction **ROW_NUMBER()** peut être utilisée pour afficher le numéro d'une ligne donnée. Utilisée conjointement avec un **PARTITION BY** la numérotation reprendra à 1 à chaque rupture.

```
SELECT
    service_id,
    lastname,
    ROW_NUMBER() OVER(ORDER BY service_id) num_row
FROM
    employees;
```

SELECT : Fenêtrage

La fonction **RANK()** à l'inverse de **ROW_NUMBER()** qui donne des numéros incrémentaux, fournit le rang (donc possiblement le même rang pour des lignes différentes ayant la même valeur).

```
SELECT
    service_id,
    lastname,
    salary,
    RANK() OVER(PARTITION BY service_id ORDER BY salary) rank
FROM
    employees;
```

SELECT : Fenêtrage

La fonction **DENSE_RANK()** identique à **RANK()** fournit le rang d'une valeur mais au contraire de **RANK()** cette fonction ne sautera pas de rang en cas de valeurs identiques.

La fonction **NTILE(slice_nb)** permet d'afficher le numéro de la "tranche" de regroupement dans laquelle la ligne se situe.

La fonction **LAG(col, distance), LEAD(col, distance)** permettent d'afficher la colonne "col" à la distance "distance" soit en arrière (LAG) soit en avant (LEAD), ce qui peut être pratique pour comparer une valeur à une autre dans une requête.

SELECT : Fenêtrage

- **Fonctions de fenêtrage** : Elles permettent d'effectuer des calculs sur un ensemble de lignes liées à la ligne courante sans regrouper les résultats en une seule ligne.
- **Syntaxe de base** : Utilisez **OVER** pour désigner une fonction de fenêtrage et **PARTITION BY** pour diviser les données en groupes.
- **Agrégats courants** : **SUM**, **COUNT**, et **AVG** peuvent être utilisés avec des fonctions de fenêtrage.
- **Numérotation des lignes** : **ROW_NUMBER()**, **RANK()**, et **DENSE_RANK()** permettent de numéroter les lignes selon un ordre spécifique.
- **Subdivision des données** : **NTILE** permet de diviser les données en quartiles, quintiles, percentiles, etc.
- **Comparaison de lignes** : **LAG** et **LEAD** permettent de comparer des lignes avec des lignes précédentes ou suivantes.
- **Alias de fenêtre** : Utilisez **WINDOW** pour définir un alias de fenêtre et simplifier les requêtes avec plusieurs fonctions de fenêtrage.

Source : https://iana-data.org/cest-a-vous/tutoriels/866-tutoriel-sur-les-fonctions-de-fenetrage-sql.html

aelion

Exercices : Fenêtrage

A partir de la base Recensement :

- ❏ Afficher le ratio de population des départements de la région Occitanie (rapport entre la population du département et celle globale de la région),
- ❏ Afficher le top 10 des villes les plus peuplées par régions,
- ❏

Travailler avec des vues

Les vues

- **Agissent comme une façade** : Une vue peut être considérée comme une façade pour masquer la complexité d'un modèle de données.
- **Syntaxe de base** : Utilisez **CREATE VIEW** pour définir une vue à partir d'une requête complexe, souvent créée à partir de jointures multiples,
- **Règles courantes** : Utilisez des requêtes les plus vastes possibles (évitez les restrictions WHERE dans les vues, sauf si nécessaire) ; utilisez les alias pour les colonnes afin de faciliter les requêtes ultérieures et éviter les conflits de nommage.
- **Les vues sont dynamiques** : Lors de l'utilisation d'une vue, la requête originale est "rejouée", les données récoltées sont donc bien celles qui existent dans les tables d'origine.
- **Les mises à jour sont interdites** : les vues ne permettent pas l'ajout / modification / suppression de données.

Créer une vue

```
CREATE VIEW nom_vue AS (

      table_query_based

);
```

Par exemple :

```
CREATE VIEW whole_population AS (
      SELECT c.id id_commune, c.nom nom_commune, population, d.id id_departement, d.nom
nom_departement, r.id id_region, r.nom nom_region
      FROM region r
            JOIN departement d ON r.id = d.region_id
            JOIN commune c ON d.id = c.departement_id
      );
```

aelion

Utiliser une vue

Une vue s'utilise comme une table dans un SELECT :

```
SELECT nom_commune, population FROM whole_population;
```

Vous pouvez utiliser la vue comme un ensemble à part entière et donc l'inclure dans une jointure, une union, une intersection.

Une vue peut être supprimée : `DROP VIEW whole_population;`

La suppression de la vue ne supprimera donc pas les données des tables ayant servi à la création de la vue.

Une vue peut être modifiée : `ALTER VIEW whole_population AS (new_query);`

Utiliser des transactions

Le système transactionnel

- **Une transaction garantit l'intégrité des données** : Les requêtes d'une transaction sont toutes exécutées s'il n'y a aucune erreur, sinon, l'ensemble n'est pas exécuté, garantissant ainsi la cohérence de votre base de données.
- **Ecriture des données** : un **COMMIT** écrira les données réellement dans la base si la transaction a réussi ; vous pouvez annuler une transaction (par exemple pour vérifier le résultat d'une transaction) avec un **ROLLBACK**,
- **Règles courantes** : Pensez à terminer vos transactions assez rapidement, pour éviter le blocage de votre base de données.
- **Veillez à insérer des validations intermédiaires** : Sur des transactions volumineuses, des points de validations intermédiaires sont bienvenus, ainsi il est possible de revenir en arrière plus facilement.

Exemple

Imaginons qu'on doive insérer un nouvel employé dans la base, ainsi que le service auquel il est rattaché :

```
START TRANSACTION;
SAVEPOINT insert_service;
INSERT INTO service (id, name) VALUES (10, 'Research and Development');
COMMIT;
SAVEPOINT insert_employee;
INSERT INTO employee (lastname, firstname, salary, service_id) VALUES ('Ghost', 'Kasper', 3500,
(SELECT id FROM service WHERE name = 'Research and Development');
ROLLBACK TO SAVEPOINT insert_service;
COMMIT;
```

Utiliser des procédures stockées

Procédures stockées

- **Une procédure stockée permet d'isoler un traitement** : A l'instar des procédures / fonctions des langages de programmation, une procédure stockée exécute un série d'instructions destinée à produire un résultat.
- **Paramètres** : Une procédure stockée peut accepter des paramètres en entrée,,
- **Valeur de retour** : Il est possible de retourner une valeur depuis une procédure stockée.
- **CREATE / DROP / ALTER** : Permettent de gérer les procédures stockées.

Exemple

Imaginons que nous souhaitions connaître rapidement l'id du service Accounting, plutôt que de systématiquement écrire la requête, il est possible de déplacer la logique vers une procédure stockée :

```
DELIMITER //
CREATE PROCEDURE GetAccountingServiceId()
BEGIN
	SELECT id
	FROM service
	WHERE name = 'accounting';
END //
DELIMITER;
```

Et utiliser ensuite cette procédure de la manière suivante :

```
CALL getAccountingServiceId();
```

aelion

Exemple

La procédure précédente est trop limitative, pour l'améliorer, on peut définir un paramètre dans la procédure, pour ensuite pouvoir transmettre la valeur à ce paramètre et modifier le comportement de la procédure :

```
DELIMITER //
CREATE PROCEDURE getServiceByName(IN service_name  VARCHAR(100))
BEGIN
	SELECT id
	FROM service
	WHERE name = service_name;
END //
DELIMITER;
```

Et utiliser ensuite cette procédure de la manière suivante :

```
CALL getServiceByName('accounting');
CALL getServiceByName('Research and Development');
```

aelion

Exemple

Pour utiliser encore plus efficacement les procédures stockées, il est possible d'ajouter un paramètre dit de sortie

```
DELIMITER //
CREATE PROCEDURE getServiceIdByName(IN service_name  VARCHAR(100), OUT servcice_id INT)
BEGIN
	SELECT id INTO service_id
	FROM service
	WHERE name = service_name;
END //
DELIMITER;
```

Et utiliser ensuite cette procédure de la manière suivante :

```
SET @serviceId = 0;
CALL getServiceIdByName('accounting', @serviceId);
INSERT INTO employee (lastname, firstname, service_id) VALUES ('Doe', 'John',
@serviceId);
```

aelion

Utiliser des déclencheurs

Triggers (Déclencheurs)

- **Il existe 6 types de triggers** : Les triggers se déclenchent selon un cycle de vie, il existe 6 crochets (hooks) sur lesquels vous pouvez déclencher des actions : BEFORE / AFTER INSERT, BEFORE / AFTER UPDATE, BEFORE/AFTER DELETE,
- **Un trigger déclenche une action de manière transparente** : Une fois défini, le trigger se déclenche de manière transparente, en tâche de fond après le cycle de vie classique. Il est donc important de maintenir correctement les déclencheurs,
- **OLD/NEW**: Les mots clés OLD et NEW permettent d'accéder aux anciennes (OLD) ou aux nouvelles (NEW) valeurs de l'objet impacté.
- **CREATE / DROP / ALTER** : Permettent de gérer les triggers.

Exemple : soft delete

A l'aide d'un déclencheur, il est possible de mettre en place un "soft delete", ou, plus précisément, conserver les données supprimées, pour éventuellement revenir en arrière en cas de mauvaise manipulation.

D'abord, on crée la table qui va recevoir l'archive :

```
CREATE TABLE deleted_employee (
      id INT,
      lastname VARCHAR(100),
      firstname VARCHAR(100),
      service_id INT,
      💥deletion_date TIMESTAMP
DEFAULT CURRENT_TIMESTAMP

);
```

Puis le trigger lui même :

```
DELIMITER //
CREATE TRIGGER after_employee_delete
AFTER DELETE ON employee
FOR EACH ROW
BEGIN
    -- Insertion des données
supprimées dans la table d'archive
    INSERT INTO deleted_employee (id,
lastname, firstname, service_id)
    VALUES (OLD.id, OLD.lastname,
OLD.firstname, OLD.service_id);
END //

  DELIMITER ;
```

aelion

Exemple : Salary mass

De la même manière on pourrait envisager la création d'une table qui stocke la masse salariale sur chaque ajout/modification/suppression de salariés :

```
DELIMITER //
CREATE TRIGGER after_employee_delete
AFTER INSERT ON employee
FOR EACH ROW
BEGIN
    -- Insertion des données
supprimées dans la table d'archive
    UPDATE salary_mass SET mass =
mass + NEW salary
END //

DELIMITER ;
```

```
DELIMITER //
CREATE TRIGGER after_employee_delete
AFTER UPDATE ON employee
FOR EACH ROW
BEGIN
      DECLARE old_salary DECIMAL(15, 2) DEFAULT 0;
      DECLARE new_salary DECIMAL(15, 2) DEFAULT 0;

      IF OLD.salary != NEW.salary THEN

            UPDATE salary_mass SET mass = mass - OLD.salary +
NEW.salary;
      END IF;

END //

DELIMITER ;
```

Bon courage !

Vous venez de voir l'essentiel de la syntaxe et des possibilités offertes par **le langage SQL**.
Rien ne remplace la pratique et n'oubliez pas de vous armer de patience.
En outre, pensez à vérifier les résultats obtenus, à l'aide d'experts métiers.