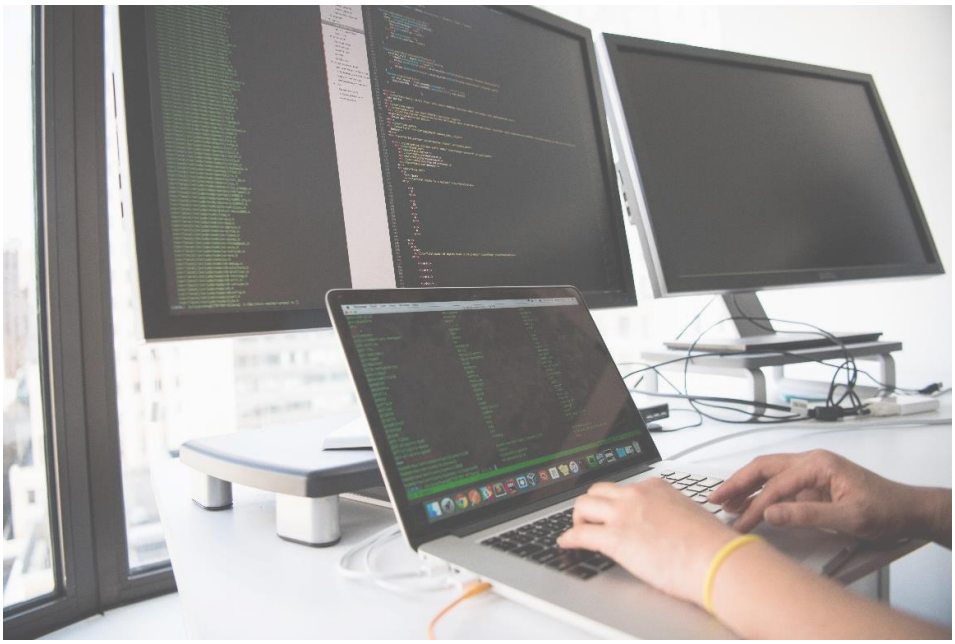
Formation

Base de données - SQL



Sébastien PHILIPPOT

Table des matières

Chapitre 1

Introduction

Les données sont au centre des systèmes d'information. Il est donc important de pouvoir les interroger, les stocker, les manipuler. C'est ce que proposent les bases de données. Elles offrent à la fois la possibilité de stocker l'information, mais aussi un langage pour manipuler ces données. Celles-ci constituent le métier traité par l'application.

Exemple de données : Facture, Produits, Clients, Utilisateurs, etc.

Ainsi une base de données offre une structure pour représenter le modèle métier.

1.1 Base de données orientée Clés/Valeurs

Ici on va représenter une donnée dans un dictionnaire

Exemple. *fact :1234 " date " 20/02/2000 " montant " 30000*

Ce type de base de données est très performante, car elles les données sont en mémoire.

1.2 Base de données orientée Document ou NoSql

Ici on va représenter un objet métier par un document au format JSON

Exemple.

```
1
2 {
3     "name":"Sue",
4     "age":26,
5     "status":A,
6     "groups":["NEWS","SPORT"]
7 }
```

Un langage permettant la manipulation des données est offert avec ce type de base de données
Exemple d'outil : MongoDb, Apache Cassandra, etc...

1.3 Base de données relationnelle

Ce type de base de données ce base sur des tableaux appelés des *tables* reliées entres elles par des clés. Nous allons étudier ce type de base de données.

Chapitre 2

Stocker des informations

2.1 Les opérations d'écriture

Nous allons voir comment on écrit des données dans une base de données à l'aide du langage SQL.

2.1.1 insérer des informations

Pour insérer des données SQl propose l'instruction *INSERT INTO*. La syntaxe est la suivante.

```
1 INSERT INTO <TABLE> (colonne1,colonne2,colonne3) VALUES (valeurColonne1,
  valeurColonne2...)
```

Toute colonne pas spécifiée prend pour valeur

Il est également possible de ne pas spécifier les colonnes, elles auront pour valeur *NULL* ou bien si la valeur par défaut si elle a été attribuée à la colonne au moment de la création de la table.

```
1 INSERT INTO <TABLE> VALUES (valeurColonne1,valeurColonne2...)
```

L'ordre des valeurs est celui de la déclaration des colonnes lors de la création de la table.

Si l'on souhaite insérer des données d'une autre table, SQL propose la syntaxe suivante :

```
1 INSERT INTO <TABLE>
2 SELECT coloneTableACopier1, coloneTableACOpier2
3 FROM TableACopier
4 WHERE <restriction>
```

Exemple. *Supposons la table* villes *qui contient toutes les villes de toutes les régions de France, cette table est trop volumineuse et on décide alors d'avoir une table qui contient les villes par région. Ainsi on aura la table* villes_idf *qui contiendra les villes de la région* Iles de France. *On souhaite extraire les villes d'iles de France et les copier dans la table* ville_idf

```
1 INSERT INTO villes_idf
2 SELECT codeVille, nomVille
3 FROM villes
4 WHERE region="IDF"
```

2.1.2 modifier des informations

Pour modifier des données SQL propose la syntaxe suivante :

```
1 UPDATE <TABLE>  SET <colonne>=<valeur>
```

Avec l'instruction ci-dessus toutes les lignes sont affectées.

Exemple.

1	*DUPONT*	*PIERRE*	*36*
2	*DURAND*	*MARIE*	*21*
3	*SANCHEZ*	*MARION*	*24*

**Table client*

Si on applique la requête

```
1 UPDATE client>  SET age=18
```

Tous les clients auront 18 ans

1	*DUPONT*	*PIERRE*	*18*
2	*DURAND*	*MARIE*	*18*
3	*SANCHEZ*	*MARION*	*18*

**Table client modifiée*

Il est possible de faire une restriction sur une requête de modification

```
1 UPDATE table  SET <colonne>=<valeur> WHERE <condition>
```

Exemple.

```
1 UPDATE client  SET age=18 WHERE id=1
```

1	*DUPONT*	*PIERRE*	*18*
2	*DURAND*	*MARIE*	*21*
3	*SANCHEZ*	*MARION*	*24*

**Table client modifiée*

2.1.3 supprimer des informations

Pour supprimer des lignes d'une table nous pouvons utiliser la syntaxe suivante

```
1 DELETE FROM table  WHERE condition
```

Pour vider une table SQL propose la syntaxe suivante :

```
1 TRUNCATE table
```

Chapitre 3

rechercher des informations

3.1 Les opérations de lecture

3.1.1 La projection

L'opérateur de projection porte sur les attributs. Ces derniers sont représentés par des colonnes dans les bases de données relationnelles. La projection met en lumière certaines colonnes et permet de produire un sous-ensemble de la table d'origine.

Table : Utilisateurs

Nom	Prénom	Age
Durant	Jean	26
Dupont	Pierre	49

Π nom,age (Utilisateur)

Nom	Age
Durant	26
Dupont	49

En SQL la projection s'exprime

```
1 SELECT colonne1,colonne2 FROM table
```

Pour éliminer les doublons nous devons utiliser le mot-clé *DISTINCT*

```
1 SELECT DISTINCT colonne1,colonne2 FROM table
```

3.1.2 La restriction

L'opérateur de restriction permet de *restreindre* les lignes d'une table en fonction d'une condition.

Table : Utilisateurs

Nom	Prénom	Age
Durant	Jean	26
Dupont	Pierre	49

σ age>30 (Utilisateur)

Nom	Prénom	Age
Dupont	Pierre	49

En SQL la restriction s'exprime

```
1 SELECT colonne1,colonne2 FROM table WHERE condition
```

SQL donne la possibilité d'exprimer des conditions à l'aide d'opérateur
Pour illustrer les conditions en SQL nous allons nous baser sur la table suivante

1	DUPONT	PIERRE	18
2	DURAND	MARIE	21
3	SANCHEZ	MARION	24
4	BAE	DELPHINE	40
5	LEI	ANAIS	60

*Table client

Faire des comparaisons

Tableau 9.1. Opérateurs de comparaison

Opérateur	Description
<	inférieur à
>	supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à
=	égal à
<> ou !=	différent de

[1]

Exemple. `1 SELECT nom,prenom FROM client WHERE age>40`

le résultat de la requête est : LEI ANAIS car son âge est supérieur à 40 ans

Vérifier l'appartenance à une liste

L'appartenance s'exprime à l'aide de l'opérateur *IN* ou *NOT IN*

```
1 SELECT nom,prenom FROM client WHERE age IN (18,21,24)
```

le résultat de la requête est : DUPONT PIERRE, DURAND MARIE, SANCHEZ MARION

1. https ://docs.postgresql.fr/12/functions.html

Vérifier si une valeur est comprise dans un intervalle

```
1 SELECT nom,prenom FROM client WHERE age BETWEEN IN 20 and 50
```

le résultat de la requête est : BAE DELPHINE, car son âge (40 ans) est compris entre 20 et 50 ans

Vérifier la présence de certains caractères

Cette vérification se fait à l'aide de l'opérateur *LIKE* et d'un masque qui va décrire la forme de la valeur. Un masque est un filtre composé de

— _ désigne un caractère quelconque

— % désigne toutes suite de caractères

Exemple. *Supposons un client nommé DURAND et un second nommé DURANT , il existe également un client nommé DURANTE*

```
1 SELECT nom,prenom FROM client WHERE nom like (DURAN_)
```

le résultat de la requête est : DURAND et DURANT

```
1 SELECT nom,prenom FROM client WHERE nom like (DURAN%)
```

le résultat de la requête est : DURAND et DURANT et DURANTE

3.2 Les jointures

Une jointure permet d'obtenir des données issues de plusieurs tables. Il existe différentes manières de procéder

3.2.1 Le produit cartésien

Le produit cartésien aussi appelé *Produit relationnel* permet de récupérer toutes les lignes de chaque table le nombre de lignes serait le produit du nombre de lignes des tables jointes.

1	DUPONT	PIERRE	18
2	DURAND	MARIE	21
3	SANCHEZ	MARION	24
4	BAE	DELPHINE	40
5	LEI	ANAIS	60

*Table client

1	20/11/2015	3000	1
2	30/05/2016	6000	2
3	23/09/2016	1500	3
4	14/11/2006	320	4
5	17/02/2017	458	5

*Table commande

```
1 SELECT nom,prenom,montant,date FROM client , commande
```

Donnerait un résultat de 25 lignes. il s'agit de toutes les combinaisons possibles.

il est donc nécessaire de mettre une *condition de jointure* dans la clause *Where*. En effet, on peut voir que la table *client* et *commandes* sont reliées par l'identifiant du client (représenté en rouge dans les tableaux ci - dessus).

```
1 SELECT nom,prenom,montant,"date" FROM client, commande WHERE client.id=commande.
  idClient
```

Nous obtenons alors le résultat suivant :

DUPONT	PIERRE	3000	20/11/2015
DURAND	MARIE	6000	30/05/2016
SANCHEZ	MARION	1500	23/09/2016
BAE	DELPHINE	320	14/11/2006
LEI	ANAIS	458	17/02/2017

*Résultat de la jointure avec condition

3.2.2 Les sous requêtes

Il est possible de remplacer les jointures par des sous-requêtes, mais cela peut parfois complexifier la lecture.

Les deux requêtes ci-dessous sont équivalentes

```
1 SELECT *
2 FROM client, commande
3 WHERE client.id=commande.idClient
```

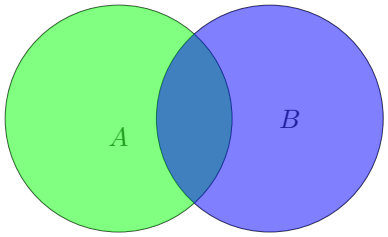
```
1
2 SELECT *
3 FROM commande
4 WHERE commande.idClient IN
5 (SELECT id FROM client)
```

En effet, dans la seconde requête on cherche à savoir si l'identifiant des clients présents dans la table commande est dans le résultat de la requête *SELECT idClient FROM commande*. Cette dernière renvoie bien la liste des identifiants des clients. C'est-à-dire 1,2,3,4,5 cela revient donc à faire

```
1
2 SELECT *
3 FROM client, commande
4 WHERE client.id IN (1,2,3,4,5)
```

On cherche les commandes attachées à un client présent dans la table client. Le résultat des deux requêtes est ainsi identique. Toutefois, l'emploi des sous-requêtes peut alourdir la lecture, on préférera donc utiliser des jointures.

3.2.3 L'union

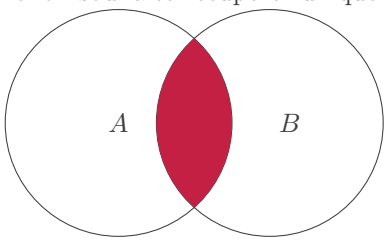


Si on a des lignes dans une table A et des lignes dans une table B, l'union des deux tables aura les lignes de A et de B sans inclure les doublons. C'est pratique lorsque l'on a les mêmes données séparées en plusieurs tables. Par exemple une table entreprise_idf et entreprise_occitanie. L'union des deux donnera l'ensemble des entreprises.

```
1
2 SELECT * FROM entreprise_idf
3 UNION
4 SELECT * FROM entreprise_occitanie
```

3.2.4 L'intersection

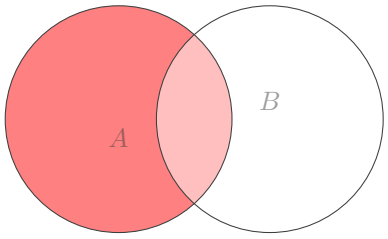
Ici on souhaite récupérer uniquement les clients qui ont une commande sur le site



SQL propose alors la syntaxe suivante :

```
1 SELECT nom,prenom FROM Client INNER JOIN commande on client.id=commande.idClient
```

3.2.5 Les jointures à gauche



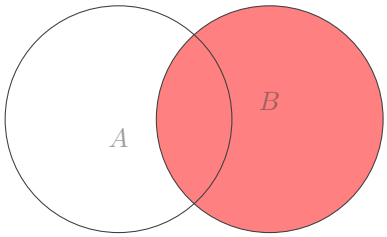
dans ce type de jointure on va prendre tous les éléments de l'ensemble A y compris ceux de l'intersection avec l'ensemble B.

la syntaxe est la suivante

```
1 SELECT nom,prenom,montant FROM Client LEFT JOIN commande on id=idClient
```

L'élément à gauche est celui donné par la clause *FROM*. Le résultat donne l'ensemble des clients même ceux qui n'ont pas de commandes. La valeur de *montant* sera *NULL*

3.2.6 Les jointures à droite



dans ce type de jointure on va prendre tous les éléments de l'ensemble B y compris ceux de l'intersection avec l'ensemble A.

la syntaxe est la suivante

```
1 SELECT nom, prenom, montant FROM Client RIGHT JOIN commande on id=idClient
```

L'élément à gauche est celui donné par la clause *FROM*. Le résultat donne l'ensemble des commandes même celles qui n'ont pas de clients affectés. La valeur de *nom* et *prenom* sera *NULL*

Chapitre 4

Les fonctions

SQL offre de nombreuses fonctions, nous allons voir les principales.

4.1 Les fonctions de chaînes de caractères

— lower(ch) et upper(ch) : construit une chaîne formée des caractères de ch transformés respectivement en minuscules et majuscules *SELECT UPPER(nom) FROM client*

— char_length(ch) : renvoie le nombre de caractères de la chaîne *ch SELECT char_length(nom) FROM client*

— substring(ch from debut for fin) : renvoie les caractères compris entre *debut* et *fin* de la chaîne *ch SELECT substring(nom from 1 for 2) FROM client (renvoie les deux premiers caractères*

4.2 Les fonctions de convertions

cast(ch AS type) *SELECT cast(âge as TEXT) FROM client*
on peut également utiliser la syntaxe suivante
SELECT âge : :TEXT FROM client

4.3 Fonctions de sélections

Ces fonctions renvoient une valeur choisie selon un cas particulier, c'est l'équivalent des expressions IF..ELSEIF

```
case
 when c1 then exp1
 when c2 then exp2
  ...
 else expn
 end
renvoie

```

Exemple.
```
SELECT  nom,
        case
        WHEN age >=18 THEN 'OUI'
        WHEN age <18 THEN 'NON'
        END as "majeur ?"
FROM client
```

On peut ajouter également l'instruction *ELSE* qui est la valeur par défaut si aucun cas n'est trouvé

4.4 Fonction d'agrégation

Il existe également des fonctions prédéfinies qui donnent une valeur agrégée calculée pour les lignes sélectionnées.

— count(*) donne le nombre de lignes trouvées,

— count(nom-colonne) donne le nombre de valeurs de la colonne

— avg(nom-colonne) donne la moyenne des valeurs de la colonne,

— sum(nom-colonne) donne la somme des valeurs de la colonne,

— min(nom-colonne) donne le minimum des valeurs de la colonne,

— max(nom-colonne) donne le maximum des valeurs de la colonne.

```
select avg(age) from client;
```

Donne l'âge moyen des clients
Vous trouverez l'ensemble des fonctions fournies par postgresql
https ://www.postgresql.org/docs/9.1/functions.htmlici

4.5 Créer une fonction

SQL offre la possibilité de créer ses propres fonctions avec la syntaxe suivante.

```

CREATE OR REPLACE FUNCTION "nom de la fonction"()
RETURNS integer
LANGUAGE 'sql'
AS $$
 code de la fonction
$$;
```

Exemple.
```
    CREATE OR REPLACE FUNCTION "totalClientMajeur"()
   RETURNS integer
   LANGUAGE 'sql'
    AS $$
      select count(*) from client where age>=18
   $$;
```

Cette fonction renvoie le nombre de clients majeurs. On peut l'utiliser dans nos requêtes ou bien l'exécuter de manière isolée.

```
1 SELECT "totalClientMajeur"();
```

Chapitre 5

Les procédures stockées

Une procédure est un ensemble d'instructions qui peut-être exécutée par un programme ou un trigger. L'avantage d'une procédure est qu'elle est stockée en base de données, elle est donc commune pour toutes les applications. Lorsqu'il y a un traitement métier commun entre différentes applications, une procédure peut-être une bonne option.

SQL propose la syntaxe suivante pour créer une procédure.

Exemple.

```
1    CREATE PROCEDURE
2        inert_data(nom text, age integer)
3    LANGUAGE SQL
4    AS $$
5      INSERT INTO client(nom,age) VALUES (nom,age);
6    $$;
7
```

On commence par nommer la procédure ici insert_data on lui donne comme paramètre d'entré le nom et l'âge. On doit spécifier également le langage ici SQL permet d'écrire une procédure portable entre la plupart des SGBD relationnels. Il excite également le langage PL/PGSQL qui est spécifique à Postgresql par exemple, vous trouverez plus d'informations sur ce langage https ://-docs.postgresql.fr/9.6/plpgsql.htmlici. Ce qui se trouve dans le bloc *AS* est le corps de la procédure. Il est entouré d'un délimiteur *$$* qui permet de dire explicitement au SGBD que ce code SQL est celui de la procédure.

On peut exécuter cette procédure par l'instruction suivante.

```
1    CALL insert_data("ALIBERT", 42);
```

On peut également ajouter des contrôles avec des instructions IF.. THEN, des SWITCH ..CASE, loop .... vous trouverez plus d'informations https ://www.postgresql.org/docs/current/plpgsql-control-structures.htmlici

Exemple.

```
1      CREATE PROCEDURE
2        inert_data3(nom text, age integer)
3    LANGUAGE plpgsql
4    AS $$
5BEGIN
6      SELECT * FROM client;
```

```
 7       IF NOT FOUND THEN
 8           INSERT INTO client(nom, age) VALUES (nom, age);
 9       END IF;
10END;
11    $$;
12
```

Dans ce cas là le langage doit être plsql et les instructions doivent être dans un bloc *BEGIN*.

On peut donc faire des conditions, des boucles, des opérations on peut finalement reproduire un traitement métier. Il est donc nécessaire de s'interroger si le traitement métier doit être dans une procédure afin de le rendre commun pour différentes applications ou encore pour optimiser des performances ou bien s'il doit être au niveau applicatif afin de bénéficier d'un langage de haut niveau et de centraliser le métier.