SGBD. SQL

I/ Introduction

Le modèle relationnel étudié précédemment est un modèle mathématique permettant de raisonner sur des données sous forme de tables. Il est mis en œuvre par un logiciel particulier, le ***S**ystème de **G**estion de **B**ases de **D**onnées* (*SGBD* en abrégé et *DBMS* en anglais pour ***D**ata **B**ase **M**anagement **S**ystem*).
Depuis les fin des années 1990, le modèle relationnel est plus répandu, environ les trois quarts des bases de données utilisent ce modèle.

Développé par IBM dans les années 1970, le **langage SQL** (***S**tructured **Q**uery **L**anguage* en anglais) est un langage de requêtes qui est rapidement devenu le standard des langages des bases de données relationnelles.
Il permet de manipuler la structure d'une base de données sous plusieurs aspects :

- **Création** / **Suppression** de tables.
- **Recherche** de données.
- **Ajout** / **Suppression** de données.

Ce langage s'articule autour d'une vingtaine d'instructions dont la syntaxe ressemble à celle de phrases ordinaires en anglais. SQL s'utilise souvent de manière interactive, ou à l'intérieur d'un langage hôte comme le Cobol, le C, Python, PHP etc.

Pour travailler sur les bases de données avec SQL, plusieurs outils sont nécessaires :

- Un **serveur** pour héberger la base.
- Un **SGBD** pour la manipuler comme *MySQL* ou *SQLite* par exemple.

A Noter :
Quelques correspondances de mots de vocabulaires entre le modèle relationnel et la base de données qui en résulte :

Modèle relationnel	Base de données
Relation	*Table*
Attribut	*Colonne / Champs*
Enregistrement	*Ligne*

II/ SQL : un langage de définition

1/ Création d'une table et insertion d'enregistrements

A titre d'exemple, on souhaite créer la table suivante :

Nom	Maths	Anglais	NSI
Joe	16	17	18
Jeanne	19	15	17

On supposera que l'attribut *Nom* est la clé primaire (ce qui est en général une mauvaise idée mais il s'agit seulement d'un exemple).

Pour créer la table, on utilise l'instruction **CREATE TABLE** :

```
CREATE TABLE IF NOT EXISTS Table_notes (
    Nom TEXT PRIMARY KEY,
    Maths INTEGER,
    Anglais INTEGER,
    NSI INTEGER );
```

Quelques remarques :

- L'instruction facultative *IF NOT EXISTS* permet de créer la table seulement si elle n'existe pas. Cela évite les erreurs d'éventuels doublons.
- Chaque attribut a un **nom** et un **type** qu'il faut indiquer.
- L'instruction *PRIMARY KEY* indique l'attribut qui sera la clé primaire.

Pour insérer les enregistrements de la table, on utilise la commande **INSERT** :

```
INSERT INTO Table_notes (Nom, Maths, Anglais, NSI)
VALUES ('Joe',16,17,18) , ('Jeanne',19,15,17);
```

Il faut bien préciser chaque **colonne** (*champ* dans le modèle relationnel) par son nom.
On peut utiliser la valeur NULL pour signifier qu'une colonne n'est pas renseignée (une absence à un contrôle ici par exemple).

2/ Recherche de données

La commande **SELECT** permet de sélectionner une ou plusieurs colonnes d'une ou plusieurs tables données en paramètres. L'étoile permet de sélectionner toutes les colonnes :

Voici deux exemples de requêtes et leur résultat :

```
SELECT * FROM Table_notes;
```

On obtient :

Nom	Maths	Anglais	NSI
Joe	16	17	18
Jeanne	19	15	17

```
SELECT Nom, Maths FROM Table_notes;
```

On obtient :

Nom	Maths
Joe	16
Jeanne	19

Important : le point-virgule permet d'indiquer la fin d'une requête. Il est obligatoire sinon il y aura des erreurs d'exécution.

III/ SQL : requêtes avancées

1/ Insertion de conditions

La commande **WHERE** permet de spécifier des critères de sélection. Elle peut s'utiliser avec les opérateurs logiques habituels (OR, AND notamment).

Par exemple, pour savoir quels sont les élèves ayant obtenu une note au moins égale à 17 en maths et en NSI, on peut écrire l'instruction suivante :

```
SELECT Nom
FROM Table_notes
WHERE Maths >= 17 AND NSI >= 17;
```

On obtient :

Nom
Jeanne

2/ Modifications de données

➢ La commande **UPDATE** permet de modifier des données.

Par exemple , pour donner la note de 17 en maths à Joe (un point oublié 😊), on peut écrire la requête suivante :

Requête de mise à jour :

```
UPDATE Table_Notes
SET Maths = 17
WHERE Nom = 'Joe';
```

La requête :

```
SELECT * FROM Table_notes;
```

donne maintenant pour résultat :

Nom	Maths	Anglais	NSI
Joe	17	17	18
Jeanne	19	15	17

➢ La commande **DELETE** permet de supprimer une ou plusieurs lignes.

Par exemple , pour supprimer lignes dont les notes en NSI sont strictement inférieures à 18 :

Requête de suppression :

```
DELETE FROM Table_Notes
WHERE NSI < 18;
```

La requête :

```
SELECT * FROM Table_notes;
```

donne maintenant pour résultat :

Nom	Maths	Anglais	NSI
Joe	17	17	18

➢ D'autres commandes permettent de modifier la structure de la table elle-même. Seulement, si la base de données est correcte, il n'y pas lieu d'utiliser ce type de commandes !

Ajout d'une colonne

```
ALTER TABLE Table_notes
ADD COLUMN Classe TEXT;
```

(On ajoute une colonne « Classe » de type TEXT ici).

Renommage d'une table

```
ALTER TABLE Table_notes
RENAME TO notes;
```

(La table s'appelle « notes » désormais).

Suppression d'une table

```
DROP TABLE Table_notes;
```

(Suppression de la table -dans le respect de l'intégrité de la base de données-).

3/ Agrégation, tri de données

Le **langage SQL** permet d'opérer **certaines opérations mathématiques** (statistiques notamment) et de **trier** les lignes

En voici quelques-unes, elles sont à utiliser après la commande **SELECT** :

- AVG(*nom_colonne*) : calcule la moyenne des valeurs de *nom_colonne*.
- SUM(*nom_colonne*) : calcule la somme des valeurs de *nom_colonne*.
- MIN(*nom_colonne*) , MAX(*nom_colonne*) : donne le minimum -respectivement le maximum- des valeurs de *nom_colonne*.
- COUNT(*) : donne le nombre de lignes selon une condition.

Quelques fonctions de tri / détection de doublons.

- ORDER BY *nom_col1*, *nom_col2 etc.* ASC / DESC : permet de trier les lignes selon les valeurs de *nom_col1* (et en cas d'égalité de *nom_col2 etc.*) dans l'ordre croissant (ASC) ou décroissant (DESC).
- GROUP BY *nom_colonne* : permet d'éviter des doublons selon les valeurs de *nom_colonne*. Le mot clé DISTINCT (à placer après la commande SELECT le permet aussi).

Exemple d'utilisation

Prenons en considération une table « achat » qui résume les ventes d'une boutique :

id	client	tarif	date
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03

Ce tableau contient une colonne qui sert d'identifiant pour chaque ligne, une autre qui contient le nom du client, le coût de la vente et la date d'achat.

Pour obtenir le coût total de chaque client en regroupant les commandes des mêmes clients, il faut utiliser la requête suivante :

```
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
```

La fonction SUM() permet d'additionner la valeur de chaque tarif pour un même client. Le résultat sera donc le suivant :

client	SUM(tarif)
Pierre	262
Simon	47
Marie	38

Remarque : ne pas hésiter à se reporter sur le fichier Commandes_SQL.pdf pour plus de détails.

3/ Jointures

La puissance d'une base de données relationnelle réside dans le fait qu'elle permette de relier plusieurs tables à l'aide du système clé primaire / clé étrangère.
Pour cela, la commande **INNER JOIN** (ou **JOIN**) permet d'effectuer des recherches en croisant les données d'au moins deux tables.

Pour utiliser ce type de jointure il convient d'utiliser une requête SQL avec cette syntaxe :

```
SELECT *
FROM table1
INNER JOIN table2 ON table1.id = table2.fk_id
```

La syntaxe ci-dessus stipule qu'il faut sélectionner les enregistrements des tables *table1* et *table2* lorsque les données de la colonne « id » de *table1* est égal aux données de la colonne *fk_id* de table2.

La jointure SQL peut aussi être écrite de la façon suivante :

```
SELECT *
FROM table1
INNER JOIN table2
WHERE table1.id = table2.fk_id
```

La syntaxe avec la condition WHERE est une manière alternative de faire la jointure mais qui possède l'inconvénient d'être moins facile à lire s'il y a déjà plusieurs conditions dans le WHERE.

Exemple

Imaginons une application qui possède une table utilisateur ainsi qu'une table commande qui contient toutes les commandes effectuées par les utilisateurs.

Table utilisateur :

id	prenom	nom	email	ville
1	Aimée	Marechal	aime.marechal@example.com	Paris
2	Esmée	Lefort	esmee.lefort@example.com	Lyon
3	Marine	Prevost	m.prevost@example.com	Lille
4	Luc	Rolland	lucrolland@example.com	Marseille

Table commande :

utilisateur_id	date_achat	num_facture	prix_total
1	2013-01-23	A00103	203.14
1	2013-02-14	A00104	124.00
2	2013-02-17	A00105	149.45
2	2013-02-21	A00106	235.35
5	2013-03-02	A00107	47.58

Dans cette base de données, *id* est la clé primaire de la table *utilisateur* et *utilisateur_id* est une clé étrangère (reliée à *id*) de la table *commande*.

Pour afficher toutes les commandes associées aux utilisateurs, il est possible d'utiliser la requête suivante :

```
SELECT id, prenom, nom, date_achat, num_facture, prix_total
FROM utilisateur
INNER JOIN commande ON utilisateur.id = commande.utilisateur_id
```

Résultats :

id	prenom	nom	date_achat	num_facture	prix_total
1	Aimée	Marechal	2013-01-23	A00103	203.14
1	Aimée	Marechal	2013-02-14	A00104	124.00
2	Esmée	Lefort	2013-02-17	A00105	149.45
2	Esmée	Lefort	2013-02-21	A00106	235.35

Remarque : Le résultat de la requête montre parfaite la jointure entre les 2 tables. Les utilisateurs 3 et 4 ne sont pas affichés puisqu'il n'y a pas de commandes associés à ces utilisateurs.

On peut également la commande **AS** qui permet d'utiliser des alias.

```
SELECT id, prenom, nom, date_achat, num_facture, prix_total
FROM utilisateur AS users
JOIN commande AS command ON users.id = command.utilisateur_id;
```