Chapitre 19

Bases de données

Principe de fonctionnement d'une base de données

Les bases de données sont devenues des éléments incontournables pour de nombreuses applications.

Elles se substituent à l'utilisation de fichiers.

Elles améliorent les performances des applications.

Elles facilitent le partage d'informations entre utilisateurs.

Base de données relationnelle

Une base de données relationnelle utilise des tables pour le stockage des informations.

Table : composant d'une base de données qui stocke les informations dans des enregistrements (lignes) et dans des champs (colonnes).

Exemple : la table des Clients, la table des Produits, la table des Commandes.

Enregistrement : ensemble des informations relatives à un élément d'une table.

Champ : un enregistrement est composé de plusieurs champs. Chaque champ contient une seule information sur l'enregistrement.

Exemple : un enregistrement Client peut contenir les champs CodeClient, Nom, Prenom, age

Clé primaire : une clé primaire est utilisée pour identifier de manière unique chaque ligne d'une table. Par exemple, le champ **`CodeClient`** est la clé primaire de la table **`Client`**. Il ne peut pas y avoir deux clients ayant le même code.

Clé étrangère : une clé étrangère représente un ou plusieurs champs d'une table qui font référence aux champs de la clé primaire d'une autre table. Les clés étrangères indiquent la manière dont les tables sont liées.

Relation : une relation est une association établie entre des champs communs dans deux tables. Une relation peut être de un à un, un à plusieurs ou plusieurs à plusieurs.

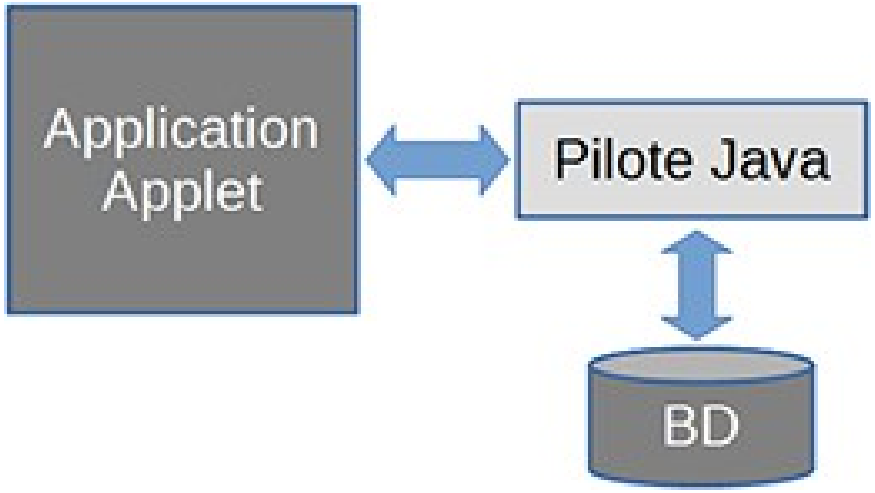
Grâce aux relations, les résultats de requêtes peuvent contenir des données issues de plusieurs tables.

Exemple : Une relation de un à plusieurs entre la table **`Client`** et la table **`Commande`** permet à une requête de renvoyer toutes les commandes correspondant à un client.

Accès à une base de données avec Java

JDBC (Java DataBase Connectivity) : bibliothèque Java permettant d'accéder à une base de données au moyen de pilotes, propres à chaque types de bases de données (Postgresql, Mysql, JavaDB, ...).

La bibliothèque JDBC contient principalement le package **java.sql**



JDBC permet de manipuler ces bases de données au moyen du langage SQL (*Structured Query Language*).
https://www.postgresql.org/docs/12/tutorial-sql.html
https://sql.sh/

Toutes les classes de JDBC sont dans le package **java.sql** qu'il faut donc importer dans tous les programmes utilisant JDBC :

```
import java.sql.*
```

Il y a 4 interfaces importantes :
`DriverManager`, **`Connection`**, **`Statement`** et **`ResultSet`**, chacune correspondant à une étape de l'accès aux données.

DriverManager : son rôle est de charger et configurer le pilote de la base de données. C'est par son intermédiaire que nous pouvons obtenir une connexion vers la base de données.

Connection : permet de réaliser la connexion et l'authentification à la base de données afin de transmettre des instructions vers la BDD.

Statement : Les requêtes simples sont exécutées grâce à cette interface.

PreparedStatement : les requêtes avec paramètres sont exécutées grâce à cette interface.

ResultSet : les éventuels enregistrements sélectionnés par l'instruction SQL sont accessibles avec un élément de classe **Resultset**.

1) Chargement du pilote (« *driver* »)

La première étape indispensable est d'obtenir le pilote JDBC adapté à la base de données à laquelle on veut accéder (PostgreSQL, MySQL, Oracle, SQLite, JavaDB, ...).

En général, ce pilote est disponible en téléchargement sous la forme d'un fichier JAR sur le site du concepteur de la base de données.

https://jdbc.postgresql.org/download.html
https://bitbucket.org/xerial/sqlite-jdbc/downloads/

Ce fichier JAR doit être accessible via la variable CLASSPATH.

2) Etablir la connexion

On établit la connexion avec la méthode **`getConnection()`** de la classe **`DriverManager`** :

```
Connection connexion =
DriverManager.getConnection (url, user, password);
```

Paramètres :

url : chaîne de caractères représentant une URL indispensable pour établir la connexion, dont le contenu diffère d'un pilote à un autre.

user, **password** : chaînes de caractères contenant les éventuels login et mot de passe si la BDD est protégée par mot de passe.

Le début de cette chaîne de connexion est toujours :

`jdbc:nomDuProtocole`

Le nom du protocole change car il est propre à chaque pilote. C'est le nom du protocole qui permettra à la méthode **`getConnection()`** d'identifier le bon pilote.

Exemple :

```
String url = "jdbc:postgresql://localhost:5432/une_base"
```

Le reste de la chaîne est spécifique à chaque pilote. Il identifie le serveur et la base de données sur ce serveur vers laquelle la connexion doit être établie.

SQLite

https://www.sqlitetutorial.net/sqlite-java/

SQLite est un moteur de base de données relationnelle accessible par le langage SQL.
Contrairement aux serveurs de bases de données traditionnels comme MySQL ou PostgreSQL, il n'utilise pas le schéma habituel client-serveur mais il est directement intégré aux programmes. L'intégralité de la base de données (déclarations, tables, index et données) est stockée dans un fichier.

Accès à une base de données locale avec un chemin relatif :
`jdbc:sqlite:base.db`

Accès à une base de données locale avec un chemin absolu :
`jdbc:sqlite:C:/dossier/base.db`

Accès à une base de données en mémoire :
`jdbc:sqlite::memory:`

JavaDB

https://docs.oracle.com/javadb/index.html

Basé sur Apache Derby, JavaDB est un moteur de base de données relationnelle écrit en langage Java qui peut être embarqué dans des programmes écrits en Java. Il a pour avantage d'être de très petite taille (2MB).

Accès à une base de données locale avec un chemin relatif :
`jdbc:derby:base.db`

Accès à une base de données locale avec un chemin absolu :
`jdbc:derby:C:/dossier/base.db`

Accès à une base de données distante :
`jdbc:derby://localhost:1527/base.db`

3) Création d'une requête

Statement : instruction simple (requêtes statiques simples)

Un objet de la classe **`Statement`** permet d'envoyer des requêtes SQL à la base de données.

La création d'un objet **`Statement`** s'effectue à partir d'une instance de la classe **`Connection`** :

```
Statement statement = connexion.createStatement();
```

4) Exécution d'une requête

Deux méthodes de la classe **Statement** :

- Pour une requête d'interrogation (SELECT) : **`executeQuery()`**
- Pour des traitements de mise à jour : **`executeUpdate()`**

Requête d'interrogation :

```
ResultSet executeQuery (String ordre) ;
```

Retourne un objet de classe ResultSet contenant tous les résultats (les n-uplets résultants) de la requête.

Exemple :

```
Statement stmt = connexion.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Client");
```

Si l'on utilise **`executeQuery`** pour exécuter une requête SQL ne contenant pas d'ordre SELECT, alors une exception de type SQLException est levée.

Requête d'exécution :

```
int executeUpdate (String ordre) ;
```

Exécute un ordre de type INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE

```
String requete = "INSERT INTO client VALUES ('112233',
                 'Jean', 'Dupont', 50)";
try {
    Statement stmt = connexion.createStatement();
    int nbMaj = stmt.executeUpdate(requete);
    System.out.println("nb mises à jour = "+ nbMaj);
}
catch(SQLException ex)
{
    ex.printStackTrace();
}
```

La méthode **`executeUpdate()`** retourne le nombre d'enregistrements qui ont été mis à jour.

5) Traitement des résultats

Le résultat d'une requête d'interrogation est transféré en mémoire dans un **`ResultSet`** par le driver JDBC. On peut se déplacer dans les lignes du résultat au moyen d'un curseur.

Méthodes de l'interface **`ResultSet`** :

```
boolean next()
```

Fait avancer le curseur sur la ligne suivante. Retourne **true** si le déplacement a été fait, **false** s'il n'y a pas d'autres lignes.

```
while(rs.next())
{
    // Traitement de chaque ligne
}
```

`boolean previous()`

Recule d'une ligne. Retourne **true** si le déplacement a été fait, **false** s'il n'y a pas de ligne précédente.

`first()`

Positionne le curseur sur la première ligne du résultat.

`last()`

Positionne le curseur sur la dernière ligne du résultat.

`close()`

Ferme le ResultSet.

Analyse du ResultSet : par indice

Accès aux données dans une ligne en fournissant **le numéro de la colonne** par les méthodes :

```
int getInt(int numcol)
String getString(int numcol)
float getFloat(int numcol)
boolean getBoolean(int numcol)
...
```

Retourne le contenu de la colonne dont l'élément est d'un type compatible avec un type Java, pouvant être String, int, float, boolean, …

Exemple :

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Client");
while(rs.next())
{
    System.out.println(rs.getString(1));
}
```

Analyse du ResultSet : par nom

Accès aux données dans une ligne en fournissant **le nom de la colonne** par les méthodes :

```
int getInt(String nomcol)
String getString(String nomcol)
float getFloat(String nomcol)
boolean getBoolean(String nomcol)
...
```

Retourne le contenu de la colonne dont l'élément est d'un type compatible avec un type Java, pouvant être String, int, float, boolean, …

Exemple :

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Client");
while(rs.next())
{
    System.out.println(rs.getString("NOM"));
}
```

6) Fermeture de la connexion

Fermer les connexions ouvertes permet de libérer les ressources mobilisées.

Chaque objet possède une méthode **`close()`** :

```
resultset.close()
statement.close()
connection.close()
```

Exceptions

Si une requête ne peut s'exécuter ou qu'une erreur de syntaxe SQL a été détectée, l'exception SQLException (erreurs SQL) est générée.

Toutes les méthodes présentées précédemment sont concernées.

Opérations possibles sur cette exception :

`int getErrorCode()`
Retourne le code de l'erreur renvoyé par le SGBD (et dépendant du type du SGBD).

`SQLException getNextException()`
Si plusieurs exceptions sont chaînées entre elles, retourne la suivante ou null s'il n'y en a pas.

`String getSQLState()`
Retourne « l'état SQL » associé à l'exception.

Exemple

```
import java.sql.*;

public class testdb
{
    public static void main(String[] args)
    {
        Connection conn = null;
        Statement statement = null;
        ResultSet result = null;
        String databaseURL = "jdbc:derby:bibliotheque.db;create=true";

        try
        {
            conn = DriverManager.getConnection(databaseURL);
            statement = conn.createStatement();

            String sql = "CREATE TABLE livres (id int primary key, titre varchar(128))";
            statement.execute(sql);

            sql = "INSERT INTO livres VALUES (1, 'Premier livre'), (2, 'Second livre')";
            statement.execute(sql);
```

```
            sql = "SELECT * FROM livres";
            result = statement.executeQuery(sql);

            while (result.next())
            {
                System.out.println(result.getInt("id") + ":" + result.getString("titre"));
            }
        }
        catch (SQLException ex)
        {
            ex.printStackTrace();
        }
        finally
        {
            try { result.close(); } catch (Exception e) {}
            try { statement.close(); } catch (Exception e) {}
            try { conn.close(); } catch (Exception e) {}
        }
    }
}
```

Netbeans

L'IDE Netbeans offre la possibilité de se connecter à des bases de données à toutes sortes de formats (il est possible d'installer de nouveaux pilotes) et d'effectuer des opérations manuellement ou à l'aide d'instructions SQL : création de tables, ajout/modification/suppression de données, etc.