VI. Accès à un SGBD distant

Bases de données relationnelles

Une base de données relationnelle est une base de données dans laquelle l'information est organisée dans des tableaux à deux dimensions (appelés tables ou relations). Les lignes de ces tables sont appelés N-uplets et les noms des colonnes sont appelés attributs. Les tables sont normalement construites de façon à éviter la redondance des données (principe de normalisation). Les relations entre les tables sont permises grâce aux principes de clé primaire (attribut qui identifie de façon unique chaque ligne d'une table) et clé étrangère (attribut d'une table qui référence la clé primaire d'une autre table). On appelle système de gestion de bases de données (SGBD) les logiciels chargés de créer, exploiter et maintenir des bases de données. Pratiquement tous les SGBD relationnels utilisent le langage SQL pour interagir avec les bases de données.

Quelques rappels sur le langage SQL

SQL permet de manipuler la structure de la BD (LDD : langage de définition de données), de manipuler les données présentes dans les différentes tables (LMD : langage de manipulation de données) et de contrôler l'accès aux données (LCD : langage de contrôle des données),

- LDD : **CREATE**, **DROP**, **ALTER**
- LMD : **INSERT**, **UPDATE**, **DELETE** (mise à jour), **SELECT** (interrogation)
- LCD : **GRANT**, **REVOKE** (droit d'accès aux données), **COMMIT**, **ROLLBACK** (transactions)

LMD : Requêtes de mise à jour

INSERT INTO nom_table (col_name_1 … col_name_n) values (val_1, …, val_n)

DELETE FROM table_name WHERE condition

UPDATE table_name set col_name_1=val_1, … col_name_n=val_n WHERE condition

LMD : Requêtes d'interrogation

```
SELECT col_name_1, …, col_name_n
FROM table_name_1, …, table_name_n
WHERE col_condition
GROUP BY col_name_1, …, col_name_n
HAVING group_condition
ORDER BY col_name_1, …, col_name_n
```

Remarque : en cas de requêtes multi-tables on trouvera dans le WHERE des tests d'égalité de type *clé primaire table X = clé étrangère table Y*

SELECT : spécifie des colonnes et/ou des résultats de calcul à afficher (5)
FROM : précise la/les tables à utiliser (1)
WHERE ; filtre les lignes selon une ou plusieurs conditions (2)
GROUP BY : forme des groupes de ligne à partir de valeurs communes sur un/des colonnes (3)
HAVING : filtre les groupes selon une ou plusieurs conditions (4)
ORDER BY : précise l'ordre d'apparition de données dans le résultat (6)

Requêtes d'interrogation (LMD)

SELECT est généralement suivi par le ou les noms de colonnes que l'on souhaite afficher (séparés par des **,**) ou par ***** si on souhaite afficher toutes les colonnes ; le mot clé **DISTINCT** (ou **UNIQUE** dans certains SGBD) permet d'éviter les doublons dans le résultat ; le mot-clé **AS** permet de modifier le nom de la colonne qui s'affichera dans le résultat (renommage temporaire) ; le SELECT peut aussi être suivi de fonction(s) telles que **AVG()**, **COUNT()**, **MAX()**, **MIN()**, **SUM()**) lorsqu'il est utilisé conjointement avec le GROUP BY

FROM est suivi du nom de la table ou des tables (les noms séparés par des **,**) à utiliser ; la encore il est possible de renommer temporairement le nom des tables grâce à **AS**)

WHERE est suivi d'une ou plusieurs conditions sur les colonnes qui peuvent s'exprimer avec des opérateurs de comparaisons, des opérateurs logiques … (**=**, **<>**, **!=**, **>**, **<**, **>=**, **<=**, **IN**, **BETWEEN**, **LIKE**, **IS NULL**, **IS NOT NULL**, **AND**, **OR)**

GROUP BY est suivi par le ou les noms de colonnes (séparés par **,**) utilisées pour faire le groupement

HAVING est utilisé conjointement au GROUP BY et est suivi de conditions sur les colonnes qui peuvent s'exprimer avec des fonctions (**COUNT()**, etc) et des opérateurs (**<**, etc)

ORDER BY est suivi par le ou les noms de colonnes utilisée(s) pour le tri, chacun pouvant être suivi par le mot-clé **ASC** ou **DESC** (ordre croissant resp. décroissant)

LDD

CREATE DATABASE database_name
DROP DATABASE database_name

CREATE TABLE table_name (
col_name_1 col_type,
...
col_name_1 col_type)

propriétés des colonnes éventuellement : PRIMARY KEY, FOREIGN KEY / UNIQUE / NOT NULL / DEFAULT / CHECK

DROP TABLE table_name

CREATE UNIQUE **INDEX index_name ON table_name (col_name_1, ,...col_name_n)**
DROP INDEX index_name ON table_name

ALTER TABLE table_name ADD col_name col_type
ALTER TABLE table_name DROP col_name

ALTER TABLE table_name ADD PRIMARY KEY (col_name)
ALTER TABLE table_name DROP PRIMARY KEY

ALTER TABLE table_name ADD INDEX index_name
ALTER TABLE table_name DROP INDEX index_name

ALTER TABLE table_name ancien_nom_table RENAME nouveau_nom_table

ALTER TABLE table_name RENAME COLUMN old_col_name TO new_col_name
ou ***ALTER TABLE*** **table_name** ***CHANGE old_col_name new_col_name***

ALTER TABLE nom_table ALTER COLUMN nom_colonne TYPE type_colonne
ou ***ALTER TABLE nom_table MODIFY nom_colonne type_colonne***

LCD

CREATE USER user_name … GRANT privilege_name_1, …, privilege_name_N

GRANT privilege_name_1, …, privilege_name_N
ON object_name
TO user_name

DENY privilege_name_1, …, privilege_name_N
ON object_name
TO user_name

REVOKE privilege_name_1, …, privilege_name_N
ON object_name
FROM user_name

privilège name : ALL PRIVILEGES / EXECUTE / SELECT / UPDATE / DELETE …

Object_name : TABLE, VIEW …

JDBC

JDBC (Java DataBase Connectivity) est une API Java créée par Sun (rachetée par Oracle) permettant à des programmes Java d'accéder à des bases de données relationnelles.

JDBC est dans le JDK depuis la version 1.1

Pour pouvoir utiliser JDBC il faut un pilote qui est spécifique à la BD à utiliser. Ce pilote permet l'indépendance de JDBC vis à vis des bases de données utilisées. Avec le JDK, un pilote est fourni qui permet l'accès aux bases de données via ODBC (pont JDBC-ODBC). Il existe des pilotes ODBC pour la quasi-totalité des BD.

Le package **java.sql** définit plusieurs classes et interfaces importantes :
- **Driver**, **DriverManager** : charger et configurer le pilote de la BD
- **Connection** : se connecter à la BD en s'identifiant si besoin
- **Statement** (et **PreparedStatement**) : établir la requête SQL et la transmettre à la BD pour exécution
- **CallableStatement** : déclencher l'exécution de procédures stockées dans la BD
- **ResultSet** (et **ResultSetMetaData**) : traiter les informations retournées par la requête (si c'est une requête d'interrogation) ;
- **DatabaseMetadata** : obtenir des informations sur la BD dans son ensemble

…

On peut aussi besoin du package **java.lang** qui définit notamment la classe **Class**

Chargement du pilote

Pour se connecter à la BD il faut charger un pilote (driver) JDBC (en pratique, il s'agit d'une classe implémentant l'interface **Driver**). Plusieurs façons de faire :

- Utiliser la méthode statique **Class.forName(…)** qui créé une instance de la classe du pilote et l'enregistre auprès du **DriverManager**.

  => Exemple de chargement d'un pilote JDBC-ODBC :

  ```
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  ```

  => Exemples de chargement d'un pilote spécifique à la BD (la documentation du pilote fournit généralement le nom de la classe à utiliser) :

  ```
  Class.forName("oracle.jdbc.driver.OracleDriver") ;
  /* ou "org.postgresql.Driver" ou "com.mysql.jdbc.Driver" etc. */
  ```

- Enregistrer directement une instance du pilote auprès du **DriverManager**

```
Driver monDriver = new com.mysql.jdbc.Driver();
DriverManager.registerDriver(monDriver);
```

- Fournir le pilote comme paramètre de la commande java :

```
java -Djdbc.drivers=org.postgresql.Driver …
```

L'interface DriverManager

```
static Connection getConnection(String url);
static Connection getConnection(String url, String user, String password);
/* Etablir une connexion à une BD ; l'url est sous la forme
protocol:sub_protocol:database_name */

String databaseUrl = "jdbc:odbc:maBase";
Connection con = DriverManager.getConnection(databaseUrl);
```

L'interface Connection

- **void setAutoCommit(boolean autoCommit);** /* Si le mode auto-commit est activé (true), chaque requête SQL est considérée comme une transaction individuelle et est commitée; sinon (false), les requêtes sont regroupées et aboutissent lors d'un appel de commit() ou rollback() ; par défaut, le mode auto-commit est activé */
- **void commit();** /* Valide tous les ordres émis depuis le dernier COMMIT /ROLLBACK et relâche les verrous (utilisable si le mode auto-commit est désactivé) */
- **void rollback();** /* Annule tous les ordres émis depuis le dernier COMMIT/ROLLBACK et relâche les verrous (utilisable si le mode auto-commit est désactivé) */

- **DatabaseMetaData getMetaData();** /* Retourne un objet qui permettra de récupérer des informations à propos de la BD (nom des tables, des colonnes …) */
- **Statement createStatement();** /* Retourne un objet qui permettra d'envoyer des requêtes SQL à la BD */
- **PreparedStatement prepareStatement(String sql);** /* Retourne un objet qui permettra d'envoyer des requêtes SQL paramétrées à la BD */
- **CallableStatement prepareCall(String sql);** /* Retourne un objet qui permettra de déclencher des procédures stockées dans la BD */

- **void close();** /* Ferme la connexion */

L'interface Statement

- **ResultSet executeQuery (String sql);** /* Exécute une requête SQL de type interrogation (SELECT) et retourne le résultat dans un objet de type ResultSet */
- **int executeUpdate(String sql);** /* Exécute une requête SQL de type mise à jour (INSERT, UPDATE ou DELETE) et retourne le nombre de lignes impactées par la requête */
- **boolean execute (String sql);** /* Exécute une requête SQL d'interrogation ou de mise à jour ; dans le cas classique où il n'y a qu'un seul résultat, la méthode retourne true si ce résultat est de type ResultSet (requête d'interrogation) et false s'il s'agit d'un entier (requête de mise à jour) ou qu'il n'y a pas de résultats */
- **ResultSet getResultSet();** /* Récupère le résultat après une requête d'interrogation lancée par un execute(…) */
- **int getUpdateCount();** /* Récupère le résultat après une requête de mise à jour lancée par un execute(…) */
- **void close();** referme la requête (relâche l'objet Statement) et libère les ressources associées (sinon, cela sera fait automatiquement).

- **void addBatch(String sql);** /* Ajoute une requête SQL à un batch */
- **int[] executeBatch();** /* Exécute le batch ; la méthode renvoie un tableau d'entiers indiquant le nombre de mises à jour de chaque requête */
- **void clearBatch();** /* Supprime toutes les requêtes ajoutées au batch */

Remarque : la mise à jour de masse n'est pas forcément supportée par le pilote ; la méthode *supportsBatchUpdates()* de la classe *DatabaseMetaData* permet de le savoir.

Exemple avec executeQuery(…) :

```
try{
    …
    String requete = "SELECT * FROM client";
    Statement stmt = con.createStatement();
    ResultSet resultats = stmt.executeQuery(requete);
    …
}
catch (SQLException e) { …}
```

Exemple avec executeUpdate(…) (requête SQL de mise à jour) :

```
try{
    …
    String requete = "INSERT INTO client VALUES (3,'client 3','prenom 3')";
    Statement stmt = con.createStatement();
    int nbMaj = stmt.executeUpdate(requete);
    …
}
catch (SQLException e) {…}
```

Exemple avec execute (requête SQL de n'importe quel type) :

```
try{
    …
    ResultSet resultats = null ;  int nbMaj ; String requete = … ;
    Statement stmt = con.createStatement() ;
    if (stmt.execute(sql)) { resultats = stmt.getResultSet(); }
    else { nbMaj = stmt.getUpdateCount(); }
    …
}
catch (SQLException e) { …}
```

L'interface PreparedStatement (hérite de Statement)

- **void setInt(int paramIndex, int value);** /* Met à jour le paramètre d'indice paramIndex avec la valeur entière fournie (la numérotation des paramètres commence à 1) */
- Autres : **setFloat(…)**, **setBigDecimal(...)**, **setDate(...)**, **setString(…)**, **setObject(...)**, **setShort(...)**, **setLong(...)**, **setDouble(...)**, **setNull(...)**, etc.

- **boolean execute();** /* Même principe que pour Statement */
- **ResultSet executeQuery();** /* Même principe que pour Statement */
- **int executeUpdate();** /* Même principe que pour Statement */

Exemple :

```
try{
    …
    String sql = "UPDATE Stocks SET prix = ?, quantite = ? WHERE nom = ?";
    PreparedStatement stmt = con.prepareStatement(sql);
    stmt.setBigDecimal(1, 15.6);
    stmt.setInt(2, 256);
    stmt.setString(3, "café");
    nbMaj=stmt.executeUpdate();
    System.out.println(nbMaj+" mises à jour concernant  le café");
   …
}
catch (SQLException e) {…}
```

L'interface CallableStatement (hérite de PreparedStatement)

```
String sql = "{? = call maProcedure[(?, ?, ...)]}";
CallableStatement statement = connection.prepareCall(sql);
```

/* Prototype d'un appel de procédure stockée **? =** si fonction mais pas si procédure **[(?, ?, ...)]** sont les éventuels paramètres de la fonction/procédure (ces paramètres peuvent être de type **IN**, **OUT** ou **INOUT**) */

- **void setInt(String paramName, int value);** /* Met à jour le paramètre de nom paramName avec la valeur entière fournie */
- Autres : **setFloat(…)**, **setBigDecimal(...)**, **setDate(...)**, **setString (…)**, **setObject(...)**, **setShort(...)**, **setLong(...)**, **setDouble(...)**, **setNull(...)**, etc.

- **int getInt(int paramIndex);** /* Retourne la valeur java (int) du paramètre d'indice paramIndex (lequel doit être de type JDBC INTEGER) */
- **int getInt(String paramName);** /* Même chose mais à partir du nom du paramètre */
- Même surcharge pour les autres types : **getFloat(…)**, **getBigDecimal(...)**, **getDate(...)**, **getString (…)**, **getObject(...)**, **getShort(...)**, **getLong(...)**, **getDouble(...)**, etc.

- **void registerOutParameter(int paramIndex, int sqlType);** /* Enregistre le paramètre de sortie d'indice paramIndex dans le type JDBC désigné */
- **void registerOutParameter(String paramName, int sqlType);** /* Même chose mais à partir du nom du paramètre */

void registerOutParameter(int index, int sqlType, String typeName);
void registerOutParameter(String paramName, int sqlType, String typeName);
/* Pour les types construits par le programmeur (JAVA_OBJECT, STRUCT …) */

void registerOutParameter(int index, int sqlType, int scale);
void registerOutParameter(String parameterName, int sqlType, int scale);
/* Pour les types JDBC NUMERIC ou DECIMAL ; scale indique le nombre de chiffres voulus après la virgule */

Tous les paramètres de type OUT doivent être enregistrés avant que la procédure soit exécutée

Il faut faire attention aux correspondances entre les types SQL (voir **java.sql.Types**) et les types Java.

Exemple

```
try{
    …
    String sql = "{call nombreAbonnes(?)}";
    CallableStatement stmt = con.prepareCall(sql);
    stmt.registerOutParameter("nb", java.sql.Types.INTEGER);
    stmt.execute();
    int resultat = stmt.getInt("nb");
    System.out.println ("nombre abonnés : "+resultat);
    …
}
catch (SQLException e) {…}
```

Type SQL	Méthode ResultSet	Type Java
ARRAY	getArray	java.sql.Array
BIGINT	getLong	long
BINARY	getBytes	byte[]
BIT	getBoolean	boolean
BLOB	getBlob	java.sql.Blob
CHAR	getString	java.lang.String
CLOB	getClob	java.sql.Clob
DATE	getDate	java.sql.Date
DECIMAL	getBigDecimal	java.math.BigDecimal
DINSTINCT	get*TypeDeBase*	*typeDeBase*
DOUBLE	getDouble	double
FLOAT	getDouble	double
INTEGER	getInt	int
JAVA_OBJECT	(*type*)getObject	*type*
LONGVARBINARY	getBytes	byte[]
LONGVARCHAR	getString	java.lang.String
NUMERIC	getBigDecimal	java.math.BigDecimal
OTHER	getObject	java.lang.Object
REAL	getFloat	float
REF	getRef	java.sql.Ref
SMALLINT	getShort	short
STRUCT	(*type*)getObject	*type*
TIME	getTime	java.sql.Time
TIMESTAMP	getTimestamp	java.sql.Timestamp
TINYINT	getByte	byte
VARBINARY	getBytes	byte[]
VARCHAR	getString	java.lang.String

L'interface ResultSet

- **boolean first();** /* Place le curseur sur le premier enregistrement */
- **boolean next();** /* Place le curseur sur l'enregistrement suivant */
- **boolean last();** /* Place le curseur sur le dernier enregistrement */

- **int getInt(int indexCol)**; /* Retourne l'entier (de l'enregistrement courant) contenu à la colonne d'indice indexCol */
- **int getInt(String nomCol)** : /* Même chose mais à partir du nom de colonne */
- autres : **getBoolean(...)**, **getShort(...)**, **getLong(...)**, **getString(...)**, etc.

Remarque 1 : la méthode **getString(...)** permet d'obtenir la valeur d'un champ de n'importe quel type.

Remarque 2 : si le champ attendu n'est pas renseigné dans la table (valeur null), la méthode getBoolean(…) retourne false, getInt(…) retourne 0, getString(…) retourne null etc. ; en général, on préfère utiliser la méthode **boolean wasNull()** (de la même interface) qui indique si la dernière colonne lue vaut null ou non.

- **ResultSetMetaData getMetaData();** /* Retourne un objet qui permettra d'obtenir des informations à propos du ResultSet courant */
- **void close();** /* Ferme le ResultSet */

```
try{
    …
    String requete = "SELECT * FROM client";
    ResultSet résultats = stmt.executeQuery(requete);
    while (resultats.next())
       System.out.println("nom/prénom:"+resultats.getString(2)+"/"+resultats.getString(3));
    …
}
catch (SQLException e) {…}
```

L'interface ResultSetMetaData

- **int getColumnCount();** /* Retourne le nombre de colonnes du ResultSet */
- **String getTableName(int indexCol);** /* Retourne le nom de la table à partir d'un indice de colonne (du ResultSet) */
- **String getColumnName(int indexCol);** /* Retourne le nom de la colonne à partir de son indice */
- **String getColumnLabel(int indexCol)**; /* Retourne le libellé de la colonne à partir de son indice */
- **int getColumnType(int indexCol);** /* Retourne le type de la colonne à partir de son indice */

Exemple :

```
try{
    …
    String requete = "SELECT * FROM client";
    ResultSet résultats = stmt.executeQuery(requete);
    ResultSetMetaData resultatsMetaDonnees = resultats.getMetaData();
    int nbCols = resultatsMetaDonnees.getColumnCount();
    System.out.println("nombre de colonnes dans le résultat : "+nbCols);
    …
}
catch (SQLException e) {…}
```

L'interface DatabaseMetaData

- **String getURL();** /* Retourne l'URL de la base de données à laquelle on est connecté */

- **String getDriverName();** /* Retourne le nom du pilote utilisé */

- **ResultSet getCatalogs();** /* Retourne la liste du catalogue d'informations (liste de BD en général) dans un ResultSet à une colonne de type String */

- **ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types);** /* Retourne la description des tables du catalogue catalog dans un ResultSet (catalog peut souvent être laissé à null). schemaPattern : généralement laissé à null. tableNamePattern : filtre (utilisation possible des caractères '%' et '_') sur les noms de tables (mettre '%' si aucun filtre, soit toutes les tables). types : tableau de chaînes indiquant les types de tables ("TABLE", "VIEW", "SYSTEM TABLE" …) à retrouver (null pour toutes). La méthode retourne un ResultSet contenant les colonnes suivantes :
1. TABLE_CAT : String => nom du catalogue qui contient la table (éventuellement null)
2. TABLE_SCHEM : String => schéma de la table (éventuellement null)
3. TABLE_NAME : String => nom de la table
4. TABLE_TYPE : String => type de la table ("TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM").
5. REMARKS … **6. TYPE_CAT** … **7. TYPE_SCHEM** … **8. TYPE_NAME** …
9. SELF_REFERENCING_COL_NAME … **10. REF_GENERATION** … */

- **ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern);** /* Même principe que précédemment mais pour une description des colonnes. Le ResultSet retourné contient :

1. TABLE_CAT : String => nom du catalogue
2. TABLE_SCHEM : String => schéma de la table
3. TABLE_NAME : String => nom de la table
4. COLUMN_NAME : String => nom de la colonne
5. DATA_TYPE : int => type SQL (tous les types SQL sont décrits dans java.sql.Types).
6. TYPE_NAME … **... 23. IS_GENERATEDCOLUMN** … */

Exemple :

```
try{
    …
    DatabaseMetaData metadonneesBD =con.getMetaData();
    String[] types = new String[1]; types[0] = "TABLE";
    ResultSet resultats = metadonneesBD.getTables(null, null, "%", types);
    ResultSetMetaData resultatsMetaDonnees = résultats.getMetaData();
    int nbCols = resultatsMetaDonnees.getColumnCount();
    while (resultats.next()) {
        for (i = 1; i <= nbCols; i++)
            System.out.print(resultats.getString(i)+" ");
        System.out.println();
    }
    …
}
catch (SQLException e) {…}
```