

# Accès avancé aux bases de données relationnelles en Java (JDBC)

Emmanuel BRUNO  
Université de Toulon, LIS UMR CNRS 7020  
emmanuel.bruno@univ-tln.fr

2026-10-05

## Table des matières

|                                                                             |    |
|-----------------------------------------------------------------------------|----|
| 1 Optimisation des Connexions à la Base de Données .....                    | 1  |
| 2 La couche DAO (Data Access Object) en Java JDBC .....                     | 3  |
| 2.1 Interface générique DAO .....                                           | 4  |
| 2.2 Implantation concrète DAO .....                                         | 4  |
| 2.3 Création d'une base de données simple .....                             | 5  |
| 2.4 Utilisation de la DAO .....                                             | 6  |
| 3 Metadata .....                                                            | 7  |
| 3.1 Exemple de récupération des métadonnées de la base de données .         | 7  |
| 3.2 Exemple de récupération des métadonnées du <code>ResultSet</code> ..... | 8  |
| 3.3 Exemple de récupération des métadonnées des colonnes d'une table .....  | 9  |
| 4 <code>ResultSet</code> modifiables .....                                  | 10 |
| 5 Batch Update .....                                                        | 11 |

## 1 Optimisation des Connexions à la Base de Données

La gestion des connexions à la base de données est cruciale pour garantir des performances optimales et une utilisation efficace des ressources. Au lieu d'ouvrir et de fermer des connexions pour chaque requête, il est recom-

mandé de les réutiliser autant que possible. Voici deux approches courantes pour gérer les connexions :

1. **DataSource** : Utilisez le concept de `DataSource`, qui fournit une interface centralisée pour gérer les connexions. Les serveurs d'applications et les frameworks tels que Spring proposent des implémentations de `DataSource`. Cela vous permet de configurer les paramètres de connexion (URL, nom d'utilisateur, mot de passe) une seule fois et de réutiliser ces connexions dans toute votre application.
2. **Apache DBCP** et **C3P0** : Utilisez des bibliothèques de pooling de connexions comme Apache DBCP ou C3P0. Elles gèrent un pool de connexions réutilisables, ce qui évite les coûts de création et de fermeture des connexions à chaque requête. Elles offrent également des fonctionnalités telles que la surveillance des connexions, la configuration des limites de pool et la gestion des connexions inactives.

En adoptant l'une de ces approches, vous améliorerez la gestion des connexions à la base de données, réduirez la latence et optimiserez les performances de votre application.

L'utilisation est simple on définit une classe qui paramètre une datasource qui gère un pool de connexion et fournit des connexions :

```
%maven    org.apache.commons:commons-  
dbcp2:2.13.0
```

```
import org.apache.commons.dbcp2.BasicDataSource;  
import java.sql.Connection;  
import java.sql.SQLException;  
  
public class DBCPDataSource {  
  
    // Création d'un pool de connexions  
    avec BasicDataSource  
    private static BasicDataSource ds  
    = new BasicDataSource();  
  
    static {  
        // Configuration des paramètres  
        de connexion  
        ds.setUrl("jdbc:postgresql://  
docker/gtfs"); // URL de la base de  
        données  
        ds.setUsername(System.getenv("DB_USER")); // Nom  
        d'utilisateur (récupéré depuis  
        les variables d'environnement)  
        ds.setPassword(System.getenv("DB_PASSWORD")); // Mot  
        de passe (récupéré depuis les  
        variables d'environnement)  
  
        // Configuration du pool  
        de connexions  
        ds.setMinIdle(5); // Nombre  
        minimal de connexions inactives dans  
        le pool  
        ds.setMaxIdle(10); // Nombre  
        maximal de connexions inactives dans  
        le pool  
        ds.setMaxOpenPreparedStatements(100); // Nombre  
        maximal de requêtes préparées  
        ouvertes simultanément  
    }  
}
```

```
// Constructeur privé pour  
empêcher l'instanciation directe de  
la classe  
private DBCPDataSource() {  
}  
  
// Méthode statique pour obtenir  
une connexion à partir du pool  
public static Connection  
getConnection() throws SQLException {  
    return ds.getConnection();  
}  
}
```

Il suffit ensuite de demander et de rendre une connexion (le close ne ferme pas forcément la connexion).

```
// Liste des identifiants d'arrêts à  
rechercher  
List<String> stopIds =  
Arrays.asList("TOCMAS", "TOCMAN");  
  
// Démonstration de l'utilisation du  
pool de connexions :  
// - Chaque itération demande/rend  
une connexion au pool  
// - Le pool évite la création/  
destruction coûteuse de connexions  
// - Les connexions sont réutilisées  
en interne par le pool  
for (String stopId : stopIds) {  
    // Utilisation de try-with-  
resources pour garantir la fermeture  
des ressources  
    try (  
        // Obtention d'une connexion  
        depuis le pool (rapide car connexion  
        déjà établie)  
        Connection connection =  
        DBCPDataSource.getConnection();  
    ) {  
        // Création d'un
```

```

PreparedStatement (optimisation et
sécurité) :
    // - Protection contre les
injections SQL
    // - Mise en cache du plan
d'exécution par la BD
    // - Réutilisation possible
de la requête compilée
    PreparedStatement pstmt =
connection.prepareStatement("SELECT
* FROM stops WHERE stop_id = ?")
    ) {
        // Paramétrage sécurisé de
la requête
        pstmt.setString(1, stopId); //
Les indices commencent à 1

        // Exécution de la requête et
récupération des résultats
        try (ResultSet rs =
pstmt.executeQuery()) {

System.out.printf("%nRésultats pour
stop_id = %s:%n", stopId);
            if (rs.next()) {
                // Affichage formaté
des résultats
                System.out.println("%s
(%s)".formatted(
rs.getString("stop_id"),
rs.getString("stop_name")));
            } else {
                System.out.printf("Aucun
arrêt trouvé avec l'ID %s:%n",
stopId);
            }
        }
    } catch (SQLException e) {
        // Gestion des erreurs avec
information contextuelle
        System.err.printf("Erreur lors
du traitement de l'ID %s: %s:%n",
stopId, e.getMessage());

```

```

    }
    // À la fin du bloc try,
les ressources sont automatiquement
fermées :
    // - Le ResultSet est fermé
    // - Le PreparedStatement est fermé
    // - La connexion est rendue au
pool (mais pas fermée physiquement)
}

```

Résultats pour stop\_id = TOCMAS:  
TOCMAS (Champ de Mars)

Résultats pour stop\_id = TOCMAN:  
TOCMAN (Champ de Mars)

## 2 La couche DAO (Data Access Object) en Java JDBC

Le pattern DAO (Data Access Object) est un composant fondamental de l'architecture des applications Java qui interagissent avec une base de données. Il répond à plusieurs problématiques essentielles du développement d'applications :

La couche DAO agit comme une interface entre l'application et la base de données. Elle permet d'isoler complètement la logique métier des détails techniques de la persistance des données. Cette séparation offre plusieurs avantages : - Maintenance facilitée du code - Possibilité de changer de système de persistance sans impacter le code métier - Centralisation de la gestion des erreurs liées à la base de données

L'implémentation d'une couche DAO repose généralement sur deux composants principaux :

1. **Interface générique** Cette interface définit un contrat standard pour les opérations CRUD (Create, Read, Update, Delete). L'utilisation des génériques Java permet

d'adapter l'interface à différents types d'entités tout en maintenant une structure commune.

2. **Classes d'implémentation** Chaque entité métier dispose de sa propre classe DAO qui implémente l'interface générique. Ces classes contiennent la logique spécifique d'accès aux données, notamment :

- Les requêtes SQL paramétrées
- La gestion des connexions
- La transformation des résultats en objets métier
- La gestion des exceptions

Pour une implémentation robuste de la couche DAO, il est recommandé de :

- Utiliser des requêtes préparées pour prévenir les injections SQL
- Gérer proprement les ressources (connexions, statements, resultsets)
- Transformer les exceptions SQL en exceptions métier
- Documenter clairement les comportements attendus
- Implémenter des tests unitaires pour chaque DAO

## 2.1 Interface générique DAO

```
public interface GenericDAO<T, ID> {
    void create(T entity) throws
        SQLException;
    T read(ID id) throws SQLException;
    void update(T entity) throws
        SQLException;
    void delete(ID id) throws
        SQLException;
    List<T> findAll() throws
        SQLException;
}
```

## 2.2 Implantation concrète DAO

```
// Record User pour représenter
l'entité utilisateur
public record User(Integer id, String
name, String email) {}
```

```
// Implémentation concrète de UserDao
public class UserDao implements
GenericDAO<User, Integer> {
    private Connection connection;
```

```
    public UserDao(Connection
connection) {
        this.connection = connection;
    }
```

```
    @Override
    public void create(User user)
throws SQLException {
        String sql = "INSERT INTO
users (name, email) VALUES (?, ?)";
        try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {
            pstmt.setString(1,
user.name());
            pstmt.setString(2,
user.email());
            pstmt.executeUpdate();
        }
    }
```

```
    @Override
    public User read(Integer id)
throws SQLException {
        String sql = "SELECT * FROM
users WHERE id = ?";
        try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            try (ResultSet rs =
pstmt.executeQuery()) {
                if (rs.next()) {
                    return new
```

```

        User(rs.getInt("id"),
            rs.getString("name"),
            rs.getString("email"));
    }
}
return null;
}

@Override
public void update(User user)
throws SQLException {
    String sql = "UPDATE users SET
name = ?, email = ? WHERE id = ?";
    try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {
        pstmt.setString(1,
user.name());
        pstmt.setString(2,
user.email());
        pstmt.setInt(3, user.id());
        pstmt.executeUpdate();
    }
}

@Override
public void delete(Integer id)
throws SQLException {
    String sql = "DELETE FROM
users WHERE id = ?";
    try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {
        pstmt.setInt(1, id);
        pstmt.executeUpdate();
    }
}

@Override
public List<User> findAll() throws
SQLException {
    List<User> users = new
ArrayList<>();
    String sql = "SELECT * FROM

```

```

users";
        try (Statement stmt =
connection.createStatement();
            ResultSet rs =
stmt.executeQuery(sql)) {
            while (rs.next()) {
                users.add(new
User(rs.getInt("id"),
rs.getString("name"),
rs.getString("email")));
            }
        }
        return users;
    }
}

```

## 2.3 Création d'une base de données simple

```

%maven com.h2database:h2:2.3.232
%maven
com.github.javafaker:javafaker:1.0.2
import com.github.javafaker.Faker;

```

```

private static final String DB_URL =
"jdbc:h2:mem:testdb";
private static final String
DB_USERNAME = "sa";
private static final String
DB_PASSWORD = "";

Class.forName("org.h2.Driver");

private static void
createTables(Connection connection)
throws SQLException {
    String createUsersTableSql =
"CREATE TABLE users (" +
        "id INT AUTO_INCREMENT
PRIMARY KEY," +
        "name VARCHAR(255) NOT
NULL," +

```

```

        "email VARCHAR(255)
NOT NULL)";
        try (Statement statement =
connection.createStatement()) {
            statement.execute(createUsersTableScript);
        }
    }

    private static
void populateTables(Connection
connection) throws SQLException {
        Faker faker = new Faker();
        String insertUsersSql = "INSERT
INTO users (name, email) VALUES
(?, ?)";

        try (PreparedStatement pstmt = connection.prepareStatement(insertUsersSql)) {
            IntStream.range(0,
10).forEach(i -> {
                try {
                    pstmt.setString(1,
faker.name().fullName());
                    pstmt.setString(2,
faker.internet().emailAddress());
                    pstmt.executeUpdate();
                } catch (SQLException e) {
                    throw new
RuntimeException(e);
                }
            });
        }
    }
}

```

```

// Utilisation de UserDao pour
effectuer des opérations CRUD
        UserDao userDao = new
UserDao(connection);

// Création d'utilisateurs
        userDao.create(new User(null,
"Alice", "alice@example.com"));
        userDao.create(new User(null,
"Bob", "bob@example.com"));

// Lecture d'un utilisateur
        User user = userDao.read(1);
        System.out.println("Read user: "
+ user);

// Mise à jour d'un utilisateur
        userDao.update(new
User(1, "Alice Updated",
"alice.updated@example.com"));

// Suppression d'un utilisateur
        userDao.delete(2);

// Lecture de tous les utilisateurs
        List<User> users =
userDao.findAll();
        users.forEach(System.out::println);

    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

## 2.4 Utilisation de la DAO

```

try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD)) {
    createTables(connection);
    populateTables(connection);
    System.out.println("Database
setup completed successfully.");
}

```

| Database                           | setup      | completed |
|------------------------------------|------------|-----------|
| successfully.                      |            |           |
| Read user:                         | User[id=1, |           |
| name=Angelica                      | Ryan       | Sr.,      |
| email=virginia.stokes@hotmail.com] |            |           |
| User[id=1,                         | name=Alice | Updated,  |
| email=alice.updated@example.com]   |            |           |
| User[id=3,                         | name=Miss  | Domonique |

```

Windler,
email=serita.hudson@hotmail.com]
User[id=4, name=Nella Morissette,
email=danny.koch@gmail.com]
User[id=5, name=Connie Gutmann I,
email=evan.mills@yahoo.com]
User[id=6, name=Antione McCullough,
email=larae.connelly@gmail.com]
User[id=7, name=Fermin Schmeler Jr., email=
User[id=8, name=Daine Kovacek,
email=rebeca.weber@hotmail.com]
User[id=9, name=Ivey Johnson Jr.,
email=dewayne.marquardt@hotmail.com]
User[id=10, name=Minh Zboncak,
email=luanna.rodriguez@yahoo.com]
User[id=11,
name=Alice, email=alice@example.com]
User[id=12,
name=Bob, email=bob@example.com]

```

### 3 Metadata

JDBC offre des fonctionnalités puissantes pour récupérer des informations à partir d'une base de données.

**ResultSetMetaData** : Cette interface permet d'obtenir des informations sur les résultats d'une requête SELECT. Vous pouvez récupérer des détails tels que le nombre de colonnes, les noms des colonnes, les types de données, les précisions, les échelles, etc. Cela facilite la manipulation des résultats et l'adaptation de votre code en fonction des données récupérées.

**DatabaseMetaData** : Cette interface fournit des informations sur la base de données elle-même, indépendamment des requêtes spécifiques. Vous pouvez obtenir des détails sur les tables, les colonnes, les index, les contraintes, les utilisateurs, les privilèges, etc. Les données accessibles via `DatabaseMetaData` varient en fonction du SGBD (Système de Gestion de Bases de Données) avec lequel vous

travaillez. Par exemple, certaines bases de données peuvent fournir plus d'informations que d'autres.

```

//| echo: false
//| output: false
String DB_URL="jdbc:postgresql://
docker/gtfs";
String DB_USERNAME=Optional.of(System.getenv("DB_USERNAME"))
>new IllegalArgumentException("DB_USERNAME
is required"));
String DB_PASSWORD=Optional.of(System.getenv("DB_PASSWORD"))
>new IllegalArgumentException("DB_PASSWORD
is required"));
%Maven
org.postgresql:postgresql:42.7.3

```

### 3.1 Exemple de récupération des métadonnées de la base de données

```

try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD)) {

    // Récupération des métadonnées
    de la base de données
    DatabaseMetaData metadata =
connection.getMetaData();

    // Affichage des informations
    sur le système de gestion de base de
    données
    System.out.printf("DBMS: %s %d.
%d%n",
metadata.getDatabaseProductName(),
metadata.getDatabaseMajorVersion(),
metadata.getDatabaseMinorVersion());

    // Affichage de la base de données
    et du schéma actuels
    System.out.printf("Current

```

```
Database: %s Schema: %s%n",
        connection.getCatalog(),
        connection.getSchema());
```

```
    // Récupération de la liste des
    tables pour vérifier l'existence de
    "SIMPLEJDBC_PERSON"
    ResultSet tables = metadata.getTables(connection.getSchema(),
        "SIMPLEJDBC_PERSON", null);

    if (tables.next()) {
        // Si la table existe déjà,
        affichage d'un message
        System.out.println("Table "
            + tables.getString("TABLE_NAME") + "
            already exists.");
    } else {
        // Si la table
        n'existe pas, création de la table
        "SIMPLEJDBC_PERSON"
        String createTableSql = "CREATE
        TABLE \"SIMPLEJDBC_PERSON\"(" +
            "id INT PRIMARY KEY
        NOT NULL, " +
            "firstname
        VARCHAR(100))";
        try (Statement statement =
            connection.createStatement()) {
            statement.execute(createTableSql);
            System.out.println("Table
            SIMPLEJDBC_PERSON created.");
        }
    }
} catch (SQLException e) {
    // Gestion des exceptions SQL
    e.printStackTrace();
}
```

```
DBMS: PostgreSQL 17.2
Current Database: gtfs Schema: public
```

```
Table SIMPLEJDBC_PERSON already
exists.
```

### 3.2 Exemple de récupération des métadonnées du ResultSet

```
// Connexion à la base de données
et récupération des métadonnées du
ResultSet
try (Connection connection =
    DriverManager.getConnection(DB_URL,
    DB_USERNAME, DB_PASSWORD);
    ResultSet rs = connection.createStatement().executeQuery(
        "SELECT * FROM stops")) {

    // Récupération de l'objet
    ResultSetMetaData pour obtenir des
    informations sur les colonnes
    ResultSetMetaData rsmd =
        rs.getMetaData();
    int nbColonnes =
        rsmd.getColumnCount();

    // Parcours des colonnes et
    affichage des informations
    for (int i = 1; i <= nbColonnes;
        i++) {
        String typeColonne =
            rsmd.getColumnTypeName(i); // Type de
            la colonne
        String nomColonne =
            rsmd.getColumnName(i); // Nom de
            la colonne
        System.out.printf("Colonne %d:
        nom = %s, type = %s%n", i, nomColonne,
        typeColonne);
    }
} catch (SQLException e) {
    // Gestion des exceptions SQL
    e.printStackTrace();
}
```

```

Colonne 1: nom = stop_id, type = text
Colonne 2: nom = stop_code, type = text
Colonne 3: nom = stop_name, type = text
Colonne 4: nom = stop_desc, type = text
Colonne 5: nom = stop_loc, type = geography
Colonne 6: nom = zone_id, type = text
Colonne 7: nom = stop_url, type = text
Colonne 8: nom = location_type, type = location_type_val
Colonne 9: nom = parent_station, type = text
Colonne 10: nom = stop_timezone, type = text
Colonne 11: nom = wheelchair_boarding_val
Colonne 12: nom = level_id, type = text
Colonne 13: nom = platform_code, type = text

```

### 3.3 Exemple de récupération des métadonnées des colonnes d'une table

```

import java.sql.*;
import java.util.stream.IntStream;

// Connexion à la base de données
// et récupération des métadonnées du ResultSet
try (Connection connection =
    DriverManager.getConnection(DB_URL,
    DB_USERNAME, DB_PASSWORD);
    ResultSet rs = connection.createStatement(
    * FROM stops LIMIT 5")) {

    // Récupération de l'objet
    ResultSetMetaData pour obtenir des

```

```

informations sur les colonnes
    ResultSetMetaData rsmd =
rs.getMetaData();
    int nbColonnes =
rsmd.getColumnCount();

    // Affichage des noms de colonnes
    IntStream.range(1, nbColonnes + 1)
        .mapToObj(i -> {
            try {
                return
rsmd.getColumn(i)+"
"+rsmd.getColumnTypeName(i);
            } catch (SQLException
e) {
                throw
new RuntimeException(e);
            }
        })
        .forEach(nomColonne
-> System.out.printf("%-20s",
nomColonne));
    System.out.println();

    // Affichage des données des lignes
    while (rs.next()) {
        IntStream.range(1, nbColonnes
+ 1)
            .mapToObj(i -> {
                try {
                    return
rs.getString(i);
                } catch
(SQLException e) {
                    throw
new RuntimeException(e);
                }
            })
            .forEach(valeur ->
System.out.printf("%-20s", valeur));
        System.out.println();
    }

```

```

} catch (SQLException e) {
    // Gestion des exceptions SQL
    e.printStackTrace();
}

```

```

stop_id text      stop_code text
stop_name text    stop_desc text
stop_loc geography zone_id text
stop_url text      location_type
location_type_valparent_station text
stop_timezone
text              wheelchair_boarding
wheelchair_boarding_vallevel_id text
platform_code text
BNBACE           null          Bassin
Castigneau null          0101000020E610
null            stop          null
null            accessible
null            null
BNBACO           null          Bassin
Castigneau null          0101000020E610
null            stop          null
null            accessible
null            null
BNBVIE           null          Base
de Vie Ouest null          0101000020E610
null            stop          null
null            accessible
null            null
BNBVIO           null          Base
de Vie Ouest null          0101000020E610
null            stop          null
null            accessible
null            null
BNCASN           null          Porte
Castigneau null          0101000020E610
null            stop          null
null            accessible
null            null

```

## 4 ResultSet modifiables

Il est aussi possible de paramétrer un ResultSet pour qu'il soit modifiable pendant la consultation et/ou que l'on puisse revenir en arrière.

```

try (Connection connection =
    DriverManager.getConnection(DB_URL,
        DB_USERNAME, DB_PASSWORD);
    ResultSet rs = connection.createStatement(ResultSet.
        ResultSet.CONCUR_UPDATABLE)
    .executeQuery("SELECT * FROM
        agency")) {

    // Affichage des données initiales
    System.out.println("Données
        initiales:");

    while (rs.next()) {
        System.out.printf("ID: %s, Nom:
            %s\n", rs.getString("agency_id"),
                rs.getString("agency_name"));
    }

    // Mise à jour d'un enregistrement
    rs.beforeFirst(); // Revenir au
    début du ResultSet
    while (rs.next()) {
        if (rs.getString("agency_name").equalsIgnoreCase("
            Mistral")) {
            rs.updateString("agency_name",
                "Agence Université Updated");
            rs.updateRow();

        }
    }

    // Ajout d'un nouvel
    enregistrement
    rs.moveToInsertRow();
    rs.updateString("agency_id",
        "9999");
    rs.updateString("agency_name",
        "New Agency");
    rs.updateString("agency_url",
        "http://newagency.com");
}

```

```

rs.updateString("agency_timezone",
"Europe/Paris");
    rs.insertRow();

    // Suppression d'un enregistrement
    rs.beforeFirst(); // Revenir au
début du ResultSet
    while (rs.next()) {
        if (rs.getString("agency_name").equals(
{
            rs.deleteRow();
        }
    }

    // Affichage des données mises
à jour
    rs.beforeFirst(); // Revenir au
début du ResultSet
    System.out.println("Données mises
à jour:");
    while (rs.next()) {
        System.out.printf("ID: %s, Nom:
%s\n",    rs.getString("agency_id"),
rs.getString("agency_name"));
    }
} catch (SQLException e) {
    // Gestion des exceptions SQL
    e.printStackTrace();
}

```

```

Données initiales:
ID: 3, Nom: Neverland
ID: 9999, Nom: New Agency
ID: 1, Nom: Agence Université Updated

```

```

org.postgresql.util.PSQLException:
ERROR: duplicate key value violates
unique constraint "agency_pkey"
Detail: Key (agency_id)=(9999)
already exists.

```

```

at org.postgresql.core.v3.QueryExecutorImpl.receiveE
at org.postgresql.core.v3.QueryExecutorImpl.processRe
at org.postgresql.core.v3.QueryExecutorImpl.execute(P
at org.postgresql.jdbc.PgStatement.executeInternal(P
at org.postgresql.jdbc.PgStatement.execute(PgStatemen
at org.postgresql.jdbc.PgPreparedStatement.executeWi
at org.postgresql.jdbc.PgPreparedStatement.executeUp
at org.postgresql.jdbc.PgResultSet.insertRow(PgResul
at REPL.$JShell$168.do_it$($JShell$168.java:78)
at java.base/jdk.internal.reflect.DirectMethodHandle
at java.base/java.lang.reflect.Method.invoke(Method.
at io.github.spencerpark.ijava.execution.IJavaExecut
at java.base/java.util.concurrent.FutureTask.run(Fut
at java.base/java.util.concurrent.ThreadPoolExecutor
at java.base/java.util.concurrent.ThreadPoolExecutor
at java.base/java.lang.Thread.run(Thread.java:1583)

```

## 5 Batch Update

**Principe et avantages** - Optimisation des performances pour les opérations multiples - Réduction des échanges avec la base de données - Amélioration significative des temps de traitement

**Points de vigilance** - Gestion explicite des transactions requise (auto-commit désactivé) - Risque de saturation mémoire avec de trop grands lots - Nécessité de prévoir un mécanisme de rollback

**Bonnes pratiques** - Utiliser des PreparedStatement pour les requêtes - Définir une taille de lot adaptée (100-1000 éléments) - Exécuter périodiquement pour les grands volumes de données - Implémenter une gestion d'erreurs robuste

```

private                                void
populateTablesWithBatch(Connection
connection, int batchSize) throws
SQLException {
    Faker faker = new Faker();
    String insertUsersSql = "INSERT

```

```

    INTO users (name, email) VALUES
    (?, ?)";

    // Désactivation de l'auto-commit
    pour gérer la transaction
    boolean initialAutoCommit =
    connection.getAutoCommit();
    connection.setAutoCommit(false);

    try (PreparedStatement pstmt = connection.prepareStatement(
    {
        // Création et exécution
        des batches
        for (int i = 0; i < batchSize;
        i++) {
            pstmt.setString(1,
            faker.name().fullName());
            pstmt.setString(2,
            faker.internet().emailAddress());
            pstmt.addBatch();

            // Exécution tous les
            100 enregistrements
            if ((i + 1) % 100 == 0) {
                int[] results
                = pstmt.executeBatch();
                System.out.printf("Lot
                de %d insertions traité%n",
                results.length);
                pstmt.clearBatch();
            }

            // Exécution du dernier
            lot partiel
            int[] remainingResults =
            pstmt.executeBatch();
            System.out.printf("Dernier
            lot de %d insertions traité%n",
            remainingResults.length);

            // Validation de la transaction
            connection.commit();

            } catch (SQLException e) {
                // Annulation en cas d'erreur
                System.err.println("Erreur
                pendant le batch insert: " +
                e.getMessage());
                connection.rollback();
                throw e;
            } finally {
                // Restauration de l'état
                initial de l'auto-commit
                connection.setAutoCommit(initialAutoCommit);
            }
        }
    }

    private static final String DB_URL =
    "jdbc:h2:mem:testdb_batch";
    private static final String
    DB_USERNAME = "sa";
    private static final String
    DB_PASSWORD = "";

    try (Connection connection =
    DriverManager.getConnection(DB_URL,
    DB_USERNAME, DB_PASSWORD)) {
        // Création de la table
        createTables(connection);

        // Mesure du temps pour l'insertion
        en batch
        long startTime =
        System.currentTimeMillis();

        populateTablesWithBatch(connection,
        1000);

        long endTime =
        System.currentTimeMillis();
        System.out.printf("Insertion
        en batch: %d ms%n", endTime -
        startTime);

        // Vérification du nombre
        d'enregistrements

```

```

        try (Statement stmt =
connection.createStatement());
            ResultSet rs
= stmt.executeQuery("SELECT COUNT(*)
FROM users")) {
            if (rs.next()) {
                System.out.printf("Nombre
total d'utilisateurs: %d%n",
rs.getInt(1));
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

```

Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Lot de 100 insertions traité
Dernier lot de 0 insertions traité
Insertion en batch: 1640 ms
Nombre total d'utilisateurs: 1000

```