

TP Gestion de données distribuées Intégration - Médiation : Extension JDBC et transaction globale

1. Schéma relationnel d'étude

Nous allons nous appuyer sur un schéma relationnel modélisant un univers bancaire très simplifié. Le schéma vous est donné ci-dessous :

- Client(**numSS**, nom, prénom, profession, adresse)
- AgenceBancaire(**numAgence**, nomAgence, nomBanque, ville)
- Compte(**numCompte**, typeCompte, solde, numClient, numAgence)

Le script de création des tables et d'insertion d'un jeu de tuples vous est donné dans le script ClientBanqueCompte.sql (voir http://www.lirmm.fr/~mougenot/Jdbc-Exemples_2007/Tp3_2007/). Attention, vous allez travailler sur des comptes Oracle définis en interne (comptes int1 à int40), veillez à renommer les tables et contraintes créées (par exemple en faisant précéder chaque nom par le compte user) pour ne pas vous retrouver en concurrence avec vos voisins.

2. Manipulation de procédures stockées dans la bd (écrites en PL/SQL) au travers de l'interface CallableStatement

L'interface *CallableStatement* va permettre d'exploiter les procédures et fonctions définies au sein de la base de données. L'intérêt est double :

- il s'agit, dans un premier temps, de s'appuyer sur les développements existants en terme de traitements de données
- il s'agit, dans un second temps, de s'assurer du bon déroulement de la transaction et notamment que rien n'est venu perturber le bon déroulement successif de deux ordres interdépendants (par exemple dans notre contexte, débit suivi d'un crédit dans le déroulement d'une transaction bancaire).

Un exemple de manipulation *CallableStatement* (classe CS.Procedure.java) vous est donné à partir du schéma de la relation *EMP* et d'une procédure nommée *AUGMENTATION* écrite au travers de la surcouche procédurale Oracle PL/SQL (augmentation.sql).

De la même manière que pour *Statement* ou *PreparedStatement*, une instance *CallableStatement* va être définie par appel de la méthode *prepareCall* sur un objet *Connection*. Dans sa signature la plus simple, *prepareCall* va admettre pour paramètre une chaîne de caractères qui va permettre de faire l'appel soit d'une procédure, soit d'une fonction. Les points d'interrogation indiquent alors les paramètres de la procédure ou de la fonction considérée.

Deux illustrations vous sont proposées :

```
Connection conn = null;

/* chaîne appel pour la procédure augmentation :
   qui admet 2 arguments IN et 3 arguments OUT */
CallableStatement cstmt =
    conn.prepareCall ("begin augmentation (?, ?, ?, ?, ?); end;");

/* chaîne appel pour la fonction augSalaire qui renvoie un argument
   et qui possède 2 arguments en entrée */
CallableStatement cstmt = conn.prepareCall (" {? = call augSalaire (?, ?)}");
```

A la manière de la méthode *prepareStatement*, la méthode *prepareCall* admet deux autres signatures qui vont permettre si

nécessaire de parcourir l'objet de type curseur (ResultSet) renvoyé ainsi que de le modifier. Il va falloir affecter des valeurs aux paramètres en entrée et enregistrer les paramètres que l'on veut manipuler en sortie.

```
// affecter la valeur 25712 au premier argument (num) de la procédure
cstmt.setString (1, 25712);
// enregistrer le même argument 1 pour pouvoir le manipuler en sortie
cstmt.registerOutParameter (1, Types.INTEGER);
```

2.1 Exercice demandé

A partir des exemples qui vous sont donnés (classes CS_Fonction.java et CS_Procedure.java, script de création de procédure augmentation.sql à l'adresse http://www.lirmm.fr/~mougenot/Jdbc-Exemples_2007), vous répondrez à la question suivante :

- créez une procédure PL/SQL nommée transfert qui permet de réaliser une opération de transfert d'argent pour un client donné, soit à partir de son compte épargne et à destination de son compte courant, soit ; à l'inverse, à partir de son compte courant et à destination de son compte épargne. Construisez une classe Java qui exploite cette procédure.

3. Verrouillage pessimiste

La syntaxe **SELECT ... FROM ... [WHERE ...] FOR UPDATE** permet de poser un verrou sur le ou les tuples consultés, le temps matériel d'effectuer une modification sur ces tuples après leur consultation, en évitant qu'une transaction concurrente vienne modifier les données dans l'intervalle situé entre l'ordre de consultation et l'ordre de modification. Il est d'usage de nommer ce type de verrouillage dès la consultation, en anticipant sur l'ordre de modification, du verrouillage pessimiste. Le parti pris est en effet de penser que différentes transactions peuvent fréquemment se trouver en concurrence et donc de prévenir en aval tout problème. Il est impératif de valider la transaction tout de suite après exécution de la modification sous peine de laisser les autres transactions en situation de famine.

3.1 Exercice demandé

- Vous allez travailler avec deux comptes utilisateurs (soit de manière individuelle, soit en binôme avec votre voisin). Vous exploitez les comptes utilisateurs de la base de données master (par exemple les comptes utilisateurs int10 et int20) et vous vous donnez des droits mutuels en lecture et modification sur la table Compte de chacun de ces comptes. Vous définirez une classe Java qui exploite le verrouillage pessimiste en posant des verrous dès la consultation des deux tables Comptes (des deux schémas utilisateurs) et qui débite le compte courant du véliplanchiste d'une des deux tables pour créditer le compte épargne du viticulteur de l'autre table. Vous pourrez tester l'effet du verrouillage pessimiste en ne validant pas, ou tout au moins en ne validant pas tout de suite les modifications et en essayant d'accéder aux tuples considérés (véliplanchiste et viticulteur) au travers d'une transaction concurrente.

4. Extension de JDBC et concepts associés

Les intérêts de l'extension de JDBC :
 packages javax.sql de manière générale
 et oracle.jdbc.pool dans le contexte d'Oracle
 reposent sur différentes notions parmi lesquelles :

- Le concept de **DataSource**. Celui-ci va permettre de désigner une base de données ou encore tout autre type de source de données et va également en simplifier l'accès et l'exploitation. ¹
- **Pool et cache de connexions (connection pooling et caching)** : les créations/fermetures de connexions physiques sur un SGBD, à travers un réseau, sont des opérations lourdes et donc longues. JDBC propose des extensions *Connection Pooling* et *Connection Caching* qui distinguent connexion logique et physique. Cette distinction entre connexion physique et logique et cette notion de cache vont donc améliorer les performances lorsque les applications vont nécessiter des accès multiples aux bases de données. Une connexion *Pooled Connection* est une connexion physique qui peut être réutilisée par de multiples connexions logiques. Quand un client JDBC obtient une connexion via une *Connection Pool*, il reçoit une connexion logique. Lorsqu'il ferme cette connexion, la connexion physique reste ouverte et peut être réutilisée pour une autre connexion logique. La connexion physique n'est fermée que lors de la fermeture de la *Connection Pool*.

¹Le concept de DataSource associé à JNDI (Java Naming and Directory Interface) va permettre d'attribuer aux bases de données des noms logiques qui en faciliteront l'exploitation et notamment la portabilité (localisation, nommage, indépendance et portabilité face à l'exploitation d'un driver ou d'une machine serveur en particulier, ...)

Pour améliorer ceci, la proposition de *Connection Caching* aide à la gestion d'ensemble de *Connection Pools*. On permet ainsi d'associer à chaque *Connection Cache* un ensemble de *Connection Pool*, chacune représentant une connexion physique sur une base de données.

- **Transactions distribuées** Cette notion va être à la base de mécanismes d'intégration entre sources de données. Il sera en effet possible de consulter/mettre à jour plusieurs bases de données en même temps.

L'objectif des sections proposées ci-dessous, est de montrer les ouvertures possibles en matière d'exploitation conjointe de plusieurs schémas utilisateurs voire de plusieurs bases de données.

5. Simplifier l'accès à une base de données via `OracleDataSource`

Un exemple est donné, un objet `DataSource` (plutôt `OracleDataSource`) est créé (nous n'avons pas recours à JNDI ici) et va nous faciliter l'accès à la base de données master. Cet objet `DataSource` va nous permettre d'instancier aisément des objets `Connexions` vers différents comptes utilisateurs (ici `scott/tiger` et `int1/int1`) pouvant être définis au sein de différentes bases de données hébergées au sein du même serveur Oracle (bases de données *lic* et *master*). Chaque connexion est une connexion physique *classique*.

5.1 Exercice demandé

- L'exemple proposé affiche l'ensemble des tuples contenus dans les tables *Emp* des deux utilisateurs. Vous reprendrez cet exemple pour réaliser des opérations de type transaction bancaire (débits suivis de crédits) à partir de deux comptes utilisateurs appartenant pour l'un à la base de données *lic* et pour l'autre à la base de données *master*.

6. Pool et cache de connexions

L'importance est donnée aux objets `ConnectionPoolDataSource` et `PooledConnection`. `ConnectionPoolDataSource` a les mêmes fonctionnalités que `DataSource` et est d'ailleurs une extension de `DataSource` mais va retourner des instances de `PooledConnection` au lieu d'instances de `Connection`, vues jusqu'à maintenant. `PooledConnection` est une connexion physique à la base de données et va servir un ensemble de connexions logiques (`Connection` du paquetage `java.sql`). La fermeture d'une connexion logique n'imposera pas la fermeture d'une connexion physique. Les performances d'accès et d'exploitation vont donc pouvoir être meilleures.

Le cache de connexions permet de maintenir l'ouverture d'un ensemble de connexions physiques. Plusieurs illustrations sont données (la plupart issues du site de documentation Oracle), une traite de l'exploitation d'objets `PooledConnection` et `OracleConnectionPoolDataSource`, deux traitent des performances du cache de connexions au travers d'objets `OracleConnectionPoolDataSource` et `OracleConnectionCacheImpl` (cette dernière classe est aujourd'hui obsolète).

6.1 Exercice demandé

- Vous reprendrez l'exercice précédent et vous le traiterez de manière à faire partager un ensemble de connexions (au travers d'un pool de connexions) à différents utilisateurs.

7. Transactions distribuées

Une transaction distribuée (ou globale) est un ensemble de deux transactions dites locales ou plus qui doivent être coordonnées. L'idée est de garantir qu'un ensemble de transactions locales soient validées de manière globale ou bien annulées toujours dans leur globalité, si d'aventure un problème survient (accès impossible à une source de données, requête mal formulée, etc). Un des exemples les plus fréquemment cités pour souligner l'importance d'un tel concept est l'exemple des transactions bancaires (un crédit sur un compte est toujours associé à un débit sur un autre compte et il faut donc passer simultanément les deux ordres).

Les concepts `XADataSource` et `XAConnection` respectent le standard XA (eXtended Architecture) et sont très proches de `pooledConnection` et de `ConnectionPoolDataSource`. Pour chaque instance de base de données, un objet `XADataSource` est créé qui va à son tour donner vie à un objet `XAConnection` (application de la méthode `getXAConnection()`). `XAConnection` va donner lieu à une `XAResource` (intermédiaire du gestionnaire de transactions et de la méthode `getXAResource()`) ainsi qu'à une instance de connexion JDBC. Chaque instance de `XAResource` est associée à un identifiant de transaction et peut réaliser (préparer, annuler, valider) les opérations de transaction qui lui sont associées.

7.1 Exercice demandé

- Un exemple qui exploite potentiellement deux bases de données, vous est donné, vous le reprendrez dans le contexte des transactions bancaires.