Notes de cours : bases de données distribuées et repliquées

Loïc Paulevé, Nassim Hadj-Rabia (2009), Pierre Levasseur (2008)

Licence professionnelle SIL de Nantes, 2009, version 1

Ces notes ont été élaborées à partir du cours "Distribution - Réplication" (2008) de Pierre Levasseur.

1 Préliminaires

1.1 Les objectifs

1. Conserver l'intégrité des données (comprend les relations + différentes contraintes).
2. Traitement rapide des requêtes (concurrence).
3. Souplesse d'utilisation, champs d'applications.

1.2 Les besoins

Dans les cours précédent, nous avons vu comment assurer l'intégrité d'une base de données au sein d'une seule transaction (contraintes oracles, triggers).

Nous avons ensuite élargi notre champs d'application en voulant autoriser plusieurs transactions en parallèle : des problèmes de concurrence d'accès se sont alors révélés, et nous avons vu comment les résoudre (vérrous).

Enfin, nous savons également interroger notre base de données depuis un programme externe (JDBC, application web par exemple), rendant notre base de données consultable et manipulable à distance.

La prochaine étape consiste à utiliser simulanément plusieurs bases de données. On peut imaginer divers scénarios :

- sauvegarde,
- répartir le traffic,
- optimiser la distance géographique entre les utilisateurs et la base de données (multi-sites),

– décentralisation des données,
– confidentialité/sécurité : un site n'a accès qu'aux données qui lui sont utiles,
– plusieurs sources de données (différents logiciels pré-existants, etc..)
– ...

1.3 Les termes

Deux principales approches :

1. Le traitement d'une requête SQL necessite de se connecter à plusieurs bases de données différentes. Il s'agit de la *distribution*.
2. Les données sont dupliquées sur plusieurs bases de données. Il s'agit de la *réplication*.

1.4 Identifier les composants d'un SGBD

De manière générale, on peut diviser un SGBD en trois couches illustrées par la figure 1 :

1. Les données, dans un format quelconque.
2. La couche SQL qui interprète et exécute les requêtes SQL. Cette couche doit s'assurer de l'intégrité de la couche données, ainsi que des contraintes du schémas. Elle gère (ou non) la concurrence d'accès. Pour assurer les contraintes, elle n'a besoin de lire que ses propres données (schématisé par un cycle sur la figure).
3. Enfin, une API est fournit, permettant au développeur d'accéder à la couche SQL.

L'actionneur (l'étudiant sur son clavier, la serveur web traitant un JSP, etc.) utilisera alors l'API pour manipuler et lire les données.

Dans le cas des bases de données distribuées, la couche SQL peut avoir besoin de lire des données sur une base de données distante comme le montre la figure 2. Pour cela elle aura besoin de se connecter à cette base de données au travers du réseau.

1.5 Les problématiques générales

– Fournir les 3 couches pour chaque base de données. Pour les données sans SGBD, JDBC ou ODBC sont capables d'attaquer un certains nombre de formats différents (à l'aide de drivers). Si le format est très spécifique, l'écriture d'un tel driver peut être requis.

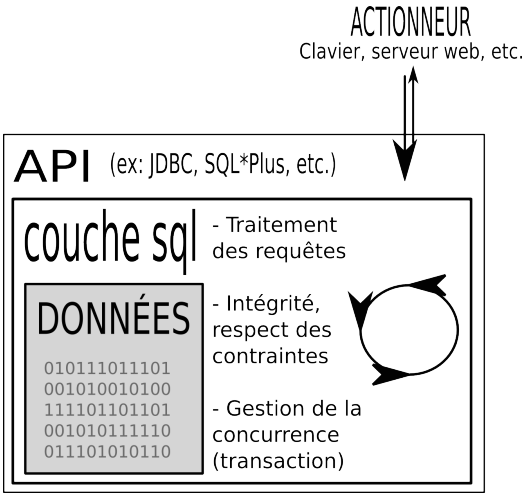

Fig. 1 – Schémas simplifié des couches fonctionnelles classiques dans un SGBD

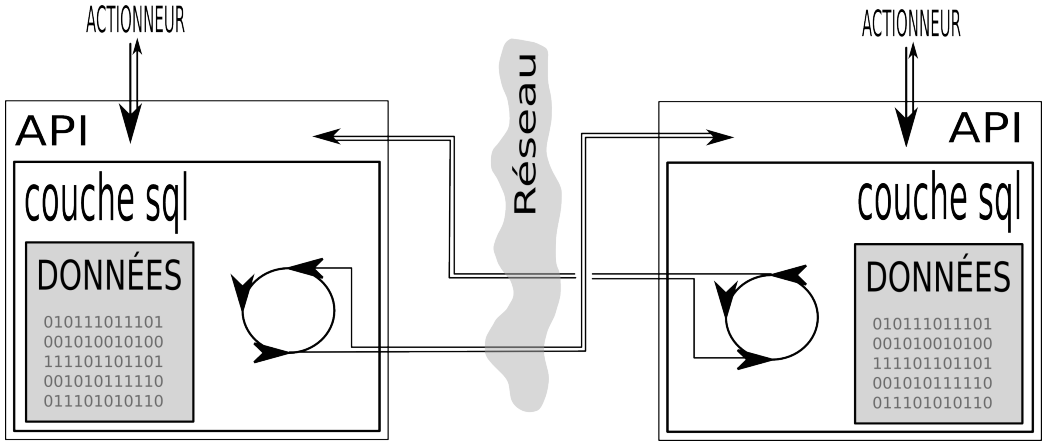

Fig. 2 – Deux SGBD en mode distribué : la couche SQL doit avoir accès à l'API de la base de données distante.

– Hétérogénéïté : la distribution/répartition peut se faire entre des logiciels de bases de données (SGBD) différents (oracle n'est pas utilisé partout... et on a rarement la possibilité de changer un sgbd existant). Il faut garantir un maximum de transparence dans le traitement des données distribuées entre plusieurs formats. Ceci n'est toutefois pas toujours évident à cause de contraintes inhérentes aux formats de fichiers (ex : un format où les accents ne sont pas autorisés).
– Aléas du réseau : échecs de connexions, temps de réponses longs, pas d'horloge globale (on ne peut pas se fier à la valeur absolue des horaires).
– Sécurité : firewall empêchant une connexion directe à une base de données distante.
– etc.

Il n'existe pas de solution toute prête pour résoudre chacun de ces problèmes. La réplication et distribution des données soulève souvent des questions qui sont au delà de la technique : sa mise en place nécessite le plus souvent une collaboration entre les différents composants d'une entreprise : informaticiens, commerciaux, partenaires, etc.

1.6 Un cas d'application

Pour illustrer les différentes parties de ce cours, nous allons nous baser sur un exemple tiré d'un cas d'application réel : Nous avons un catalogue de produits avec comme clé leur identifiant. Nous avons également un stock qui associe à chaque produit une quantité. Notre principale contrainte est de garantir que la quantité de chaque produit est supérieure ou égale à 0.

– `Produit` (**id**, intitulé)
– `LigneStock` (**id_produit**, quantite)

2 Distribution

Nous parlons de distribution à partir du moment ou une base de données doit se connecter à une base de données distante pour vérifier son schémas de données.

Par exemple, pour notre application, nous voulons mettre la table `Produit` et `LigneStock` sur des bases de données différentes, respectivement `db1` et `db2`.

Lors de l'insertion d'une ligne dans `LigneStock`, `db2` doit vérifier si l'identifiant du produit existe bien sur `db1`, en prenant soin à la concurrence d'accès. De même lors de la suppression d'une ligne dans `Produit`, `db1` doit vérifier qu'aucune ligne ne fait référence dans `db2` au produit supprimé.

2.1 Connexion distante

Pour lire des données sur une base de données distante, il faut au préalable s'y connecter :

- Il faut posséder un identifiant et un mot de passe valide sur la base de données distante.
- Il faut pouvoir techniquement se connecter à distance (problèmes typiques : firewall et/ou nat).

Oracle Il est possible de créer des *liens* ou *database link* vers des bases de données oracles distantes :

```
create database link nomlien connect to user identified by passwd
using 'configDB';
```

Pour accéder à une table distante :

```
select * from produit@db1
```

où `produit` est une table présente sur la base de données pointée par le lien `db1`.

Les actions autorisées sont : `select`, `insert`, `update`, `delete`, `lock table`.

2.2 Les contraintes de références

Oracle Exemples de trigger pour assurer l'intégrité de nos bases de données distribuées.

Note : on préfère les triggers groupés en fin d'action pour des raisons de performance : la connexion réseau ne sera utilisé qu'une seule fois par action au lieu d'une fois par ligne modifiée/insérée/supprimée.

Sur `db2`, on s'assure qu'une ligne de stock fait toujours référence à un produit existant.

```
create trigger produit_existe after insert or update on LigneStock
declare
  res LigneStock%rowtype;
begin
   lock table Produit@db1 in share mode;
   select * into res from LigneStock where id_produit not in
      (select id from Produit@db1);
   raise_application_error(-20202, 'integrity error');
exception
when no_data_found then null;
when too_many_rows then raise_application_error(-20202, 'integrity error');
end;
```

Sur `db1`, on s'assure qu'on ne supprime pas un produit présent dans une ligne de stock.

```
create trigger produit_orphelin after delete on Produit
declare
  res LigneStock@db2%rowtype;
begin
   lock table LigneStock@db2 in share mode;
   lock table Produit in share mode;
   select * into res from LigneStock@db2 where id_produit not in
       (select id from Produit);
   raise_application_error(-20202, 'integrity error')
exception
when no_data_found then null;
when too_many_rows then raise_application_error(-20202, 'integrity error');
end;
```

2.3 Le commit distribué

L'opération de commit des modifications sur plusieurs bases de données se complique.

Imaginons la situation suivante :

1. On ouvre une transaction sur `db2`,
2. exécution de `insert into Produit@db1 (id, libelle) values (9, 'nouveau produit') ;` (ouverture d'une transaction sur `db1`),
3. exécution de `insert into LigneStock values (9, 3) ;`,
4. commit des modifications sur `db2`, OK,

5. commit des modifications sur `db1`, **erreur**, rollback sur `db1`.

L'intégrité de nos données vient d'être perdue, les données sur `db2` étant désormais invalides (la ligne de stock fait référence à un produit non existant.

Une solution : demander au préalable à toutes les transactions ouvertes si les modifications sont intègres. Si et seulement si tout le monde répond oui, alors appliquer le commit réel sur toutes les transactions.

Toutefois, la gestion des aléas du réseau restent problématiques et demande l'utilisation de nombreux artifices (journaux, mises en attente, etc.)

Oracle La commande `commit` implante le commit distribué, il n'y a rien de particulier à faire.

2.4 Conclusion

Avantage :
- Facile à mettre en place.

Inconvénients :
- Attention aux commits distribués.
- Soumis aux aléas du réseau : ralentissement, voire erreurs (fragilité).
- Impossible si l'accès à la base de données distante est verrouillée par un firewall.

Conclusion : facile à mettre en place, mais semble être une mauvaise idée si l'accès à une table distante doit se faire trop souvent.

3 Réplication

On veut limiter l'utilisation du réseau. Une idée est de travailler sur une copie locale d'une table distante.

Pour notre application, on voudrait que `db2` contienne une copie de la table `Produit` de `db1`.

3.1 Réplique en lecture seule

La partie "facile" : le contenu de la table à répliquer n'est modifiable que par une seule base de données, dite maître. Les bases de données (dites esclaves) récupèreront périodiquement les données modifiées.

Attention toutefois lorsque l'on réplique deux tables ayant une relation de parenté : les copies doivent être mises à jour simultanément pour garantir l'intégrité des données à tout moment sur les bases esclaves.

Une première méthode est de copier périodiquement la totalité du contenu de la table à répliquer. Ceci n'est pas satisfaisant pour les gros volumes.

Une seconde méthode consiste à ne récupérer que les modifications effectuées depuis le dernier rafraîchissement. Pour ce faire, l'idéal est de garder une trace (journalisation) de toutes les modifications apportées. Chacune de ces modification est horodatée. Lors d'une mise à jour, on récupère toutes les modifications plus récentes que la dernière modification répliquée.

Une méthode intermédiaire serait, à l'image des fichiers, d'attacher à chaque ligne sa date de dernière modification. Attention toutefois à la suppression : il faut conserver cette opération (avec sa date).

Exemple avec Oracle :

```
create table produit_mtime (
  id number unique,
  horodatage date default sysdate,
  isdelete number(1) default 0
);

create trigger produit_insert on produit after insert for each row
begin
   delete from produit_mtime where id = :new.id;
   insert into produit_mtime (id) values (:new.id);
end;

create trigger produit_update after update for each row on produit
begin
```

```
   update produit_mtime set horodatage = sysdate
      where id = :new.id;
end;

create trigger produit_delete after delete for each row on produit
begin
   update produit_mtime set horodatage = sysdate,
      isdelete = 1
      where id = :new.id;
end;

---
lock table produit in share mode;
- données à mettre à jour
select * from produit p, produit_mtime m where p.id = m.id
   and m.horodatage > date '2009-03-01';
- lignes à supprimer
select id from produit_mtime m where isdelete = 1
   and m.horodatage > date '2009-03-01';
```

Oracle Les vues *matérialisées* sont des vues dont le contenu est physiquement présent (contrairement aux vues classiques qui sont des select à retardement). Le contenu doit être régulièrement rafraîchi. Plusieurs stratégies sont implantées.

1. Rafraîchissement complet : `create materialized view` *`replique`* `refresh complete as select * from` *`table@lien`*
   La requête sera exécutée sur la table distante et le résultat remplacera le contenu actuel. Peut être très gourmand en ressource et en temps pour les tables volumineuses.
2. Rafraîchissement incrémental : `create materialized view` *`replique`* `refresh fast as select * from` *`table@lien`*
   !!! Il faut au préalablea activer la journalisation des opérations sur la base maître : `create materialized view log on` *`table`* `with primary key ;` (créera une table `MLOG$_`*`table`*).
   Le maître enregistre les opérations effectuées sur la table ; lors d'un rafraîchissement de la réplique seules les modifications depuis le dernier rafraîchissement seront téléchargées et appliquées localement.

Par defaut, les vues sont rafraîchies à la demande, c'est à dire par un appel manuel de la procédure : `execute dbms_mview('`*`replique`*`', '`*`mode`*`', NULL)` où mode est `F` ou `C`, respectivement pour *fast* ou *complete*.

On peut demander à oracle de rafraîchir une vue matérialisée à une date précise : `create materialized view` *`replique`* `refresh fast start with sysdate next sysdate +` *`intevalle`* `as select * from` *`table`*`@`*`nomlien`* `;`. Oracle effectuera le premier rafraîchissement à la date indiquée par le `start with` et programmera le suivant à la date indiquée par `next`. Exemple de valeurs pour *intervalle* dans l'exemple donné :

1. `1` : tous les jours,
2. `1/24` : toutes les heures,
3. `5/1440` : toutes les 5 minutes, etc.

Oracle offre également la possibilité de grouper les mises à jour pour limiter une décohérence temporaire des répliques. Voir le récapitulatif des commandes oracles pour la syntaxe.

Pour la syntaxe complète et toutes les options des vues matérialisées en oracle, voir `http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_6002.htm`.

Pour plus d'information sur la manipulation des dates : `http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/sql_elements001.htm#sthref116`

3.2 Hétérogénéïté et transfert des donnnées

Dans le cas hétérogène, pour rafraîchir une réplique, deux principales façons de faire :

1. Se connecter directement à la base de données maître (via JDBC par exemple).
2. Demander au maître de générer un fichier contenant les données à mettre à jour. Ce fichier peut ensuite être envoyé directement via un service web, ou déposé sur un FTP par exemple. Le format XML est souvent préféré, la majorité des langages implantant les méthodes pour le manipuler.

3.3 Réplique en lecture/écriture

On pourrait imaginer appliquer les méthodes présentées précedemment pour faire remonter les modifications apportées sur les tables répliquées chez les esclaves vers le maître.

Attention toutefois aux problèmes inhérents à une réplique en lecture/écriture :

– Conflits de clés primaires : chaque site ajoute une ligne avec une clé primaire identique. Pas de solution universelle.

– Conflits de mises à jours : chaque site modifie la même ligne : quelle doit être la modification à garder ? Pas de solution universelle.
– Conflits de mises à jours avec suppression : un site modifie, l'autre supprime. Même problème.
– Charge réseau : si la table est répliquée en lecture/écriture sur n bases de données, il faut structurer les mises à jours pour limiter le traffic (il faut garder une hiérarchie dans l'ordre des mises à jours).

Exemple : dans notre application, nous voudrions partager la table `LigneStock` sur `db1` et `db2`. Comment gérer la décrémentation du stock ? Si on ne veut pas utiliser de distribution, la contrainte de stock positif doit être relachée. La réponse est généralement logistique et/ou commerciale.

3.4 Conclusion

Avantages :
– Permet de camoufler les aléas du réseau.
– Rapidité dans le traitement des données : pas de connexion à la base de données distante pendant les transactions, seulement périodiquement par le gestionnaire de base de données.
– Gestion des bases de données non accessibles à distances.

Inconvénients :
– Délais dans la mise à jour, attention aux synchronismes entre les vues.
– La base maître doit généralement s'abtenir de supprimer des entrées ou modifier les clés des tables répliquées.
– Pas de réponse universelle à la réplique en lecture/écriture.
– Mise en place difficile.

Conclusion : au prix d'une mise en place difficile, la réplication apporte performance et souplesse dans la gestion de bases de données multiples et hétérogènes.

Récapitulatif des commandes oracles abordées

Accès distant

`create database link` *`nomlien`* `connect to` *`user`* `identified by` *`passwd`* `using` *`'configDB'`* `;`

Crée un lien nommé *nomlien* vers la base de données *configDB*, en se connectant avec l'utilisateur *user* et le mot de passe *passwd*. *configDB* doit être configuré dans le fichier `tnsnames.ora`. Pour vos TD, utilisez `db01_lnk` et `db02_lnk` pour les bases de données `db01` et `db02` respectivement.

`table@nomlien`

Accès à la table *table* présente sur la base de données pointée par le lien *nomlien*. Les actions autorisées sont : `select`, `insert`, `update`, `delete`, `lock table`.

`drop database link` *`nomlien`* `;`

Supprime le lien *nomlien*.

Partage en lecture seule

Sur la base maître

`create materialized view log on` *`table`* `;`

Enregistre toutes les modifications apportées à *table*.

`drop materialized view log on` *`table`* `;`

Arrête l'enregistrement des modifications. Les bases esclaves ne peuvent plus utiliser la méthode “fast”.

Sur la base esclave

`create materialized view` *`nomvue`* `refresh fast start with sysdate next sysdate +` *`intevalle`* `as select * from` *`table@nomlien`* `;`

Crée la vue *nomvue* qui sera actualisée toutes les *intervalle*.

`alter materialized view` *`nomvue`* `[refresh ...] [start ...] [next ...] ;`
Mise à jour des propriétées de la vue *nomvue* (même syntaxe que pour le `create`).

`drop materialized view` *`nomvue`* `;`

Supprime la vue *nomvue*.

```
begin
   dbms_refresh.subtract(name => 'nomvue1', list => 'nomvue1');
   dbms_refresh.subtract(name => 'nomvue2', list => 'nomvue2');
   dbms_refresh.make(name => 'nomvue1_et_2',
      list => 'nomvue1, nomvue2',
      next_date => sysdate,
      interval => 'sysdate + 5/1440',
      implicit_destroy => TRUE);
end;
```

Construction d'un groupe d'actualisation : on commence par désactiver l'actualisation de *nomvue1* et *nomvue2*, puis on crée une actualisation groupée : *nomvue1* et *nomvue2* seront actualisées en même temps.

Pour aller plus loin

Différents pointeurs vers des cours, tutoriels, API utiles à la réplication et la distribution de vos bases de données.

- Cours :
  - `http://libd.isnetne.ch/cours/DistributionReplication/distribution/distribution_sup.pdf`
  - `http://libd.isnetne.ch/cours/DistributionReplication/02-replication/replication_sup.pdf`
- ODBC :
  - `http://www.commentcamarche.net/contents/odbc/odbcintro.php3` : Tutoriel ODBC
  - `http://www.unixodbc.org/` : ODBC sous Unix.
- JDBC (ODBC à la Java) : `http://java.sun.com/javase/technologies/database/`
- Oracle : `http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/toc.htm` : Documentation générale.