Les transactions

Anne-Cécile Caron

Master MIAGE - BDA

1er trimestre 2011-2012

Pourquoi ?

L'état de la base doit toujours être cohérent

- même en cas d'accès concurrents à la base (cf TP).
- même en cas de panne logicielle ou matérielle.

Exemple

On considère le célèbre exemple de virement bancaire :

```
procedure virement(A,B,X) {
  A := A-X ;
  B := B+X ;
}
```

- A et B sont des "entités" de la base de donnée, X est la valeur du virement.
- A := A-X est une façon simplifiée d'écrire :
  `update COMPTE set solde = solde-X`
  `where refCompte = refA ;`
- Par la suite, un appel à cette procédure se fera à l'intérieur d'une *transaction*.

Lecture/Ecriture

On s'intéressera aux lectures et écritures faites par une transaction. La procédure de virement devient alors :

```
debut transaction
lire(A)
ecrire(A)
lire(B)
ecrire(B)
fin transaction
```

Premier problème

Un utilisateur exécute un virement bancaire :

```
debut transaction
lire(A)
ecrire(A)
PANNE SYSTEME
```

- Le rôle du système transactionnel est de garantir que la transaction se fait complètement ou pas du tout,
- il doit donc annuler la modification de A.
- Une transaction est *Atomique*.

Gestion de la concurrence

Deux utilisateurs exécutent des virements des mêmes comptes, à l'aide de deux transactions T1 et T2 :

```
T1 : virement(A,B,100)
T2 : virement(A,B,200)
```

Il faut différencier les actions de T1 de celles de T2, et considérer l'ordre dans lequel ces actions vont s'exécuter.

Ordonnancement

Un ordonnancement est une séquence d'actions de la forme (nomTransaction, opération, donnée).

Exemple d'ordonnancement O_1 de T1 et T2 :

```
(T1, lire, A)
(T1, ecrire, A)
(T2, lire, A)
(T2, ecrire, A)
(T1, lire, B)
(T1, ecrire, B)
(T2, lire, B)
(T2, ecrire, B)
```


Deuxième problème

Considérons l'ordonnancement O_2

```
(T1, lire, A)
(T2, lire, A)
(T1, ecrire, A)
(T1, lire, B)
(T1, ecrire, B)
(T2, ecrire, A)
(T2, lire, B)
(T2, ecrire, B)
```

Ici, on a perdu une instruction de A et la base est dans un état *inconsistant*. Les effets des transactions sont modifiés à cause de la *concurrence*.

Propriétés des ordonnancements

- Un ordonnancement est *sériel* s'il correspond à une exécution en série des transactions.
- Deux actions d'un ordonnancement sont *conflictuelles* si elles concernent la même entité et qu'au moins l'une des deux est une écriture.
- Deux ordonnancements O_1 et O_2 des mêmes transactions sont *équivalents* si pour toutes actions conflictuelles a et a', a est avant a' dans O_1 ssi a est avant a' dans O_2.
- Un ordonnancement est *sériable* s'il est équivalent à un ordonnancement sériel.
- Seuls les ordonnancements sériables sont corrects.

Exemples

- L'ordonnancement O_1 est sériable car équivalent à `T1 ; T2`.
- L'ordonnancement O_2 n'est pas sériable.

Synthèse : Concept de transaction

- Une transaction est un ensemble de modifications de la base qui forme une unité de traitement.
- Elle doit respecter les propriétés ACID
  - Atomicité : une transaction s'effectue entièrement ou pas du tout
  - Consistance : Une transaction qui prend la base dans un état cohérent doit la rendre dans un état cohérent.
  - Isolement : pas d'interférence avec les utilisateurs concurrents.
  - Durabilité : Les actions effectuées par une transaction terminée sont prise en compte dans la base de données.

Concept de transaction (2)

- Une transaction est donc un programme qui accède à la base.
- Les transactions sont gérées par un moniteur transactionnel.
- Une transaction peut être dans différents états :
  - Active : pendant le déroulement du programme, tant qu'aucun problème n'apparaît.
  - Partiellement validée : Lorsque la dernière instruction a été atteinte
  - Validée : Après une exécution totalement terminée (ordre `commit`)
  - Echouée : après un problème qui a interrompu la transaction

Comment garantir la sériabilité ?

- Verrouillage 2 phases.
- Estampillage
- Multi-versions.

Verrouillage 2-phases

- Verrous : une transaction ne peut accéder à une entité dans un certain mode (lecture/écriture) que si elle dispose du verrou correspondant
- Deux verrous conflictuels ne peuvent pas être accordés en même temps
- Deux phases : Une phase pour poser des verrous, une phase pour retirer les verrous. Quand une transaction a enlevé un verrou, elle ne peut plus en poser (phase 2).

Si un ordonnancement est à 2 phases (i.e. toutes ses transactions sont à 2 phases) alors il est sériable.

transaction "virement"

```
vl(A)
lire(A)
ve(A)
ecrire(A)
vl(B)
lire(B)
ve(B)
ecrire(B)
dl(A)
de(A)
dl(B)
de(B)
```


Inconvénients

- Dead-lock → avortement de l'une des transactions.
  Détection :
  - timeout
  - détection de cycle dans le graphe des attentes
- Comment choisir la transaction à avorter ?
  - laisser les transactions proches de la fin
  - laisser les transactions qui ont fait beaucoup de mise à jour (coût du rollback)
  - ne pas toujours tuer la même transaction
- En pratique, les verrous sont tous posés en début de transaction, pour éviter les *avortements en cascade*.

Performances

- Le verrouillage utilise 2 techniques : le blocage et l'avortement des transactions. Ces 2 mécanismes pénalisent les performances.
  - Les transactions bloquées possèdent des verrous et entraînent le blocage d'autres transactions.
  - L'avortement suivi du redémarrage d'une transaction est évidemment du temps perdu.
- En pratique, moins de 1% des transactions sont impliquées dans un deadlock, et il y a relativement peu d'avortements.
  $\rightarrow$ Les problèmes de performances viennent plutôt des attentes.
- Pour améliorer les performances :
  - Poser des verrous les plus petits possibles, pour diminuer le risque que 2 transactions aient besoin du même verrou
  - Réduire la durée des transactions, pour que les verrous ne soient pas conservés trop longtemps.
  - Eviter les ressources critiques (**hotspot** = objet fréquemment lu ou modifié).

Estampillage

- Chaque transaction est estampillée par la date de son début.
- En cas de conflit, on exécute la transaction de plus petite estampille (donc la plus vieille)
- Pour valider une transaction, il faut que les plus anciennes soient validées
- Lorsqu'une transaction veut disposer d'une ancienne valeur d'une entité, il faut l'avorter
- Bref, ça marche : un ordonnancement qui satisfait la règle d'estampillage (les actions conflictuelles s'exécutent dans l'ordre des estampilles) est sériable.

Multi-versions

- On conserve les versions successives d'une même donnée.
- Lorsqu'une transaction a besoin de lire une donnée, elle va chercher la version qui lui faut.
- Coût du stockage ; algo permettant de jeter les versions qui ne serviront plus.

Norme SQL2

La norme définit deux caractéristiques pour une transaction :

- Le mode, i.e. les opérations possibles,
  - `READ ONLY`
  - `READ WRITE` (par défaut)
- Le niveau d'isolement : Le TP illustre quelques problèmes que l'on peut rencontrer avec SQL utilisé de manière concurrente. La norme définit des niveaux d'isolement pour empêcher ces problèmes.

Norme SQL3

- *points de contrôle* dans les transactions.

```
begin
  insert into joueur
    values ('165789','Bisk','Otto');
  savepoint p1;
  insert into joueur
    values ('376487','Biss','Scott');
  rollback to p1;
  commit ;
end ;
```

  La première instruction `insert` est validée, pas la seconde.
- Le concept de transaction atomique est étendu à celui de **transactions imbriquées**
- Ces deux ajouts permettent de manipuler des transactions plus longues, mais étant composées de sous-transactions (courtes) qui peuvent être annulées indépendamment.
- Les points de contrôle sont juste un support pour la forme la plus simple d'imbrication (1 niveau)

Niveau d'isolement

- `READ UNCOMMITED` : aucun isolement des transactions. On peut avoir des lectures inconsistantes des tables.
- `READ COMMITED` : Evite les lectures inconsistantes. (Une instruction SQL est toujours consistante).
- `REPEATABLE READ` : Empêche le problème des lectures répétées qui ne donnent pas le même résultat.
- `SERIALIZABLE` : Garantit la sériabilité des ordonnancements. Evidemment ça pose des problèmes de performance.

Les transactions sous Oracle

- Une transaction commence à la connexion, ou à la fin de la transaction précédente.
- Une transaction se termine par l'instruction `commit` ou `rollback`.
- L'instruction `commit` valide toutes les modifications effectuées depuis le début de la transaction.
- L'instruction `rollback` annule toutes les modifications effectuées depuis le début de la transaction.
- Les instructions du DDL sont suivies d'un `commit` implicite, on ne peut donc pas faire d'annulation d'une telle instruction. Si la transaction courante contient des instructions DML, il y a d'abord un commit de ces instructions avant d'effectuer l'instruction DDL.
- La déconnexion entraîne aussi un `commit` de la transaction en cours.
- Imbrication de transactions (requête SQL sous SQL*Plus, imbrication en PL/SQL...).
- Points de contrôle en PL/SQL

Rollback au niveau instruction

- Si une instruction SQL cause une erreur à l'exécution (à cause d'une contrainte d'intégrité par exemple), alors cette instruction est annulée (`rollback`) mais pas celles qui l'ont précédée dans la transaction courante.
- Annuler une instruction signifie annuler tous les effets de cette instruction. Par exemple, en cas de trigger, ça peut entraîner l'annulation d'autres instructions.
- Une exception : les séquences ne participent pas aux transactions. Si une instruction utilisant une séquence échoue, la séquence ne revient pas dans son état avant exécution de l'instruction.

Paramètrage des transactions

- Conformément à la norme SQL2, on peut définir le mode de la transaction par l'instruction : `SET TRANSACTION { READ ONLY | READ WRITE } ;`
- On peut aussi définir les niveaux d'isolement suivants :
  `SET TRANSACTION ISOLATION LEVEL`
  `{ SERIALIZABLE | READ COMMITTED } ;`
- L'instruction `SET TRANSACTION` doit être la première de la transaction.
- Par défaut (si l'on ne précise pas le mode), une transaction est `READ WRITE` et son niveau d'isolement est `READ COMMITED`.

Verrouillage

- Une transaction pose un verrou quand elle exécute une instruction SQL. Les verrous sont relachés à la fin du commit ou rollback de la transaction.
- Oracle utilise deux types de verrou :
  1. Verrou sur une ligne
  2. Verrou sur une table
- Si le comportement par défaut ne convient pas, l'utilisateur peut explicitement poser des verrous. Ce verrouillage dure le temps de la transaction : il prend fin au premier commit ou rollback (explicite ou implicite). Il se fait par la commande `LOCK TABLE`, ou par la commande `select ... for update`

Transactions "Autonomes"

- Une transaction autonome est une transaction indépendante, qui peut être appelée à l'intérieur d'une autre transaction.
- Une fois invoquée, une transaction autonome T_2 est complétement indépendante de la transaction T_1 qui l'a appelée : T_2 ne voit pas les modifications faite par T_1 si elle ne sont pas validées avant l'invocation de T_2. T_1 et T_2 ne partagent pas de ressources ou de verrous, il peut donc y avoir un dead lock entre T_1 et T_2.
- Une transaction autonome peut en invoquer une autre, il n'y a pas de limite au nombre d'imbrications.
- Pour définir une transaction autonome, on utilise la directive :

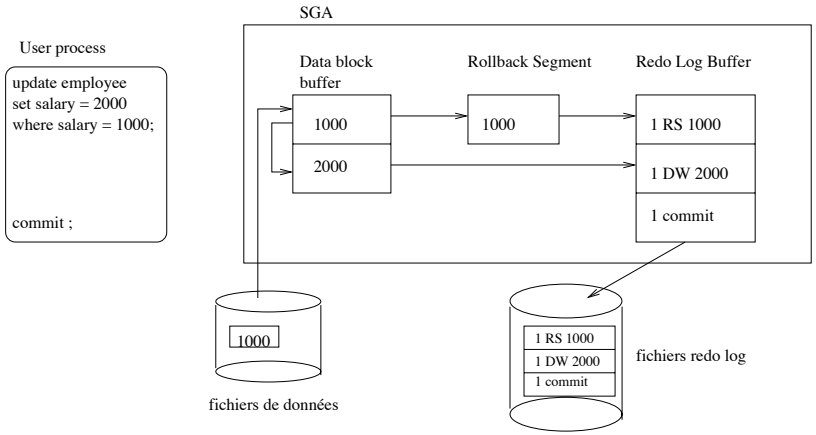
  `pragma AUTONOMOUS_TRANSACTION`

Le serveur Oracle

Le serveur gère trois types d'éléments :

- une zone d'échanges appelée `System Global Area`
- un ensemble de processus (systèmes et utilisateurs)
- des fichiers contenant les informations (données, journal, paramètres du serveur, ...)

Exécution d'une transaction Oracle



Exécution d'une transaction Oracle (2)

- On lit le bloc de donnée à partir des fichiers (si nécessaire)
- On sauve l'ancienne valeur dans le rollback segment
- On inscrit cette action dans le journal
- On change la valeur 1000 en 2000
- On inscrit cette action dans le journal
- Au commit : on écrit le journal dans les fichiers REDO LOG
- On avertit le process à l'origine de la transaction que le commit s'est bien déroulé.

Cause de l'échec d'une transaction

- Problème logique (erreur logicielle, contrainte non satisfaite ...) : l'opération `rollback` permet de remettre la base dans l'état qu'elle avait avant le début de la transaction.
- Panne du serveur ou de l'OS : utiliser le journal pour remettre la base dans l'état qu'elle avait avant la panne,
- Problème physique (panne serveur, crash disque ...) : utiliser des sauvegardes des fichiers de données et des journaux.

Recouvrement en cas d'erreur logique

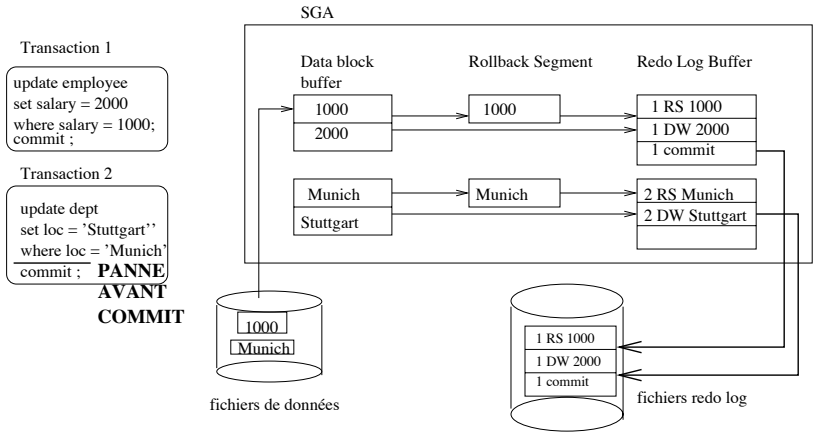
Process recovery

- Le processus PMON utilise le Roll back segment pour remettre le data block buffer dans l'état qu'il avait avant le début de la transaction.
- Ce processus enlève les verrous posés par le processus utilisateur terminé anormalement.

Recouvrement en cas d'erreur système

- Anomalie du SGBD ou de l'OS.
- L'utilisation du journal est nécessaire : "rejouer" les transactions validées (mais sans sauvegarde dans les fichiers de données) et "défaire" les transactions non validées.

Recouvrement en cas d'erreur système (2)



Rejouer les transactions validées

La transaction T1 a été validée, mais la mise à jour n'est pas répercutée dans les fichiers de données. A partir du fichier REDO LOG, le processus SMON va donc :

- écrire 1000 dans le rollback segment
- écrire 2000 dans le fichier de données
- effectuer le commit : effacer 1000 du rollback segment

La transaction T1 est terminée

Annuler les transactions non validées

Pour la transaction T2, le processus SMON poursuit la lecture du journal :

- il écrit *Munich* dans le rollback segment
- il écrit *Stuttgart* dans le fichier de données
- comme il n'y a pas de commit pour T2, il effectue un rollback : remplace *Stuttgart* par *Munich* dans le fichier de données.

Checkpoints

Combien de temps dure cette opération, combien de lignes du REDO LOG file doit-on traiter ?

- Il faut savoir à quelle moment la base s'est trouvée dans un état cohérent.
- Consistency Checkpoint : Moment où l'on écrit dans les fichiers de données.
- Ces points se produisent
  - Quand un Redo Log file est plein et qu'il faut basculer sur un autre fichier
  - Quand un certain nombre de blocs ont été écrits dans le Redo Log file (paramètre `LOG_CHECKPOINT_INTERVAL`, défini par l'administrateur).

Gestion des "Undo Tablespace"

- Avant Oracle 9i : toutes les données non validées (avant commit) d'une transaction sont dans des rollback segments. L'instruction commit ou rollback a pour effet de supprimer ces données des rollback segments.
- Les rollback segments sont aussi utilisés pour garantir des lectures consistantes (on a une version à un instant t)
- Depuis Oracle 9i, on peut choisir de ranger les données temporaires dans des `Undo Tablespaces` (à la place des rollback segments). Ca permet de pouvoir les mémoriser plus longtemps, et en cas de transaction longue, de pouvoir garantir plus sûrement les lectures consistantes.
(*moins de messages* `snapshot too old`*)*

Problème matériel : panne disque

Oracle fonctionne sous 2 modes :

- mode `NOARCHIVELOG` :
  - Arrêter le serveur régulièrement pour faire des sauvegardes (data files, redo log files, control files).
  - On pourra restaurer la base mais les informations depuis la dernière sauvegarde sont perdues.
- mode `ARCHIVELOG` :
  - Il n'est pas nécessaire d'arrêter le serveur pour faire des sauvegardes des fichiers de données ou de contrôle.
  - Les fichiers RedoLog sont archivés automatiquement (process `ARCH`).
  - Si un fichier de données est perdu, on restaure le fichier de données (à partir des sauvegardes) puis on utilise les redo log files archivés pour rejouer les transactions qui ont eu lieu depuis la sauvegarde.