



Transactions

Corentin Barloy, Anne-Cécile Caron, Julie
Jacques, Mikaël Monet

Pourquoi ?

Le concept de **transaction** va permettre de définir des processus garantissant que l'état de la base est toujours 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*.

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

Pour mettre en évidence les problèmes, 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 : panne système

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 (rollback).
- 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)`

Pour des raisons de performance, les actions des différentes transactions sont entrelacées.

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 d'un ensemble de transactions est une séquence d'actions de la forme (nomTransaction, opération, donnée) telle que, pour chaque transaction, l'ordre de ses opérations est respecté.

Exemple d'ordonnancement O_1 de T1 et T2 :

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

Deuxième problème : concurrence

T1 : virement(A,B,100)

T2 : virement(A,B,200)

Considérons l'ordonnancement O_2

(T1, lire, A)

(T2, lire, A)

(T1, écrire, A)

(T1, lire, B)

(T1, écrire, B)

(T2, écrire, A)

(T2, lire, B)

(T2, écrire, B)

Exercice : quelles sont les modifications faites par les transactions T1 et T2.

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*.

Sérialisabilité des ordonnancements

- Deux ordonnancements O_1, O_2 sont dits *équivalents* si pour toute base de données D , l'effet de O_1 sur D est le même que l'effet de O_2 sur D .
- Un ordonnancement est dit *sérialisable* s'il est équivalent à un ordonnancement en série de ses transactions.

Problème : vérifier si un ordonnancement est sérialisable est un problème NP-complet !

→ En pratique, on utilise la notion de *conflict-sérialisabilité* (voir plus tard), qui peut être testée et imposée efficacement

Quand on mélange les deux...

Les deux problèmes (panne/annulation d'une transaction et concurrence) peuvent se mélanger !

- Exemple : même si un ordonnancement O de deux transactions T_1, T_2 est sérialisable, si lors de son exécution T_1 est annulée (panne, ou parce qu'une contrainte a été violée), alors il faudrait potentiellement annuler T_2 si celle-ci a lu des données écrites par T_1 .

→ Solutions dans la suite des slides

Synthèse : Concept de transaction

- Une transaction est un programme qui modifie la base de données et 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)

- 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
- L'instruction `commit` permet de signaler que la transaction s'est bien terminée, les modifications qu'elle a effectuées sont rendues visibles (ou pas) aux autres transactions.
- Si une transaction échoue, le `rollback` permet d'annuler toutes les actions effectuées par cette transaction.

Conflit-sérialisabilité des ordonnancements

- Deux actions d'un ordonnancement sont *conflictuelles* si elles proviennent de deux transactions différentes, 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 *conflit-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 *conflit-sérialisable* s'il est conflit-équivalent à une exécution en série des transactions.
- Exercice : Montrer que l'ordonnancement O_1 est conflit-sérialisable.
- Exercice : Montrer que l'ordonnancement O_2 n'est pas conflit-sérialisable.

Proposition : Un ordonnancement conflit-sérialisable est sérialisable.

Preuve : au tableau. Idée : on passe de O à un ordonnancement en série des transactions en swappant itérativement deux actions non-conflictuelles adjacentes (de deux transactions différentes) judicieusement choisies.

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 sérialisable.

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 → interruption de l'une des transactions.
Détection :
 - timeout : si une transaction attend un verrou depuis un certain temps, on suppose qu'elle est bloquée par un deadlock
 - détection de cycle dans le graphe des attentes : les noeuds sont les transactions actives et il y a un arc de T_i vers T_j ssi T_i attend une ressource verrouillée par T_j .
- Comment choisir la transaction à interrompre ?
 - 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 interruptions *en cascade*.

Performances

- Le verrouillage utilise 2 techniques : le blocage et l'interruption des transactions. Ces 2 mécanismes pénalisent les performances.
 - Les transactions bloquées possèdent des verrous et entraînent le blocage d'autres transactions.
 - L'interruption suivie 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'interruptions.
→ 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).

Alternative au verrouillage

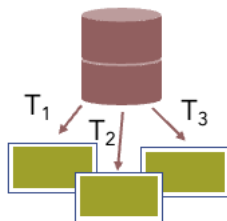
- Le verrouillage est une façon "pessimiste" de traiter la concurrence : on part du principe qu'il va y avoir des conflits et on gère des verrous. Si les conflits sont rares, la gestion des verrous fait perdre en performance.
- On peut aussi traiter le problème de façon optimiste, en partant du principe qu'il y aura très peu de conflits :
 - ① Lecture : la transaction s'exécute, lit des données dans la base et écrit dans un espace privé
 - ② Validation : Quand la transaction est terminée, le SGBD vérifie qu'elle peut être validée, i.e. qu'il n'y a pas eu de conflit avec une autre transaction pendant son exécution. En cas de conflit, la transaction est annulée, son espace privé est vidé, et la transaction est relancée
 - ③ Ecriture : Si la transaction termine sans conflit, les données écrites dans l'espace privé sont copiées dans la base.

Conclusion

- Une gestion pessimiste est intéressante s'il y a beaucoup de conflits, pénalisante sinon car on gère des verrous pour rien
- Une gestion optimiste est intéressante s'il y a peu de conflits, donc peu d'attente liée à la section critique des phases de validation/écriture.

Multi-versions

- Utilisation d'instantanés (SNAPSHOTS)
- On conserve les versions successives d'une même donnée, ce qui revient à donner une estampille aux objets, pas seulement aux transactions.
- Chaque transaction travaille sur une version à un instant T de la base (T = début de la transaction)



La norme définit deux caractéristiques pour une transaction :

- Le mode, i.e. les opérations possibles,
 - READ ONLY *transaction-level read consistency*
 - READ WRITE (par défaut) *statement-level read consistency*
- 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.

Niveau d'isolement de la norme SQL

- **READ UNCOMMITTED** : aucun isolement des transactions. On peut avoir des lectures inconsistantes (dirty read) des tables, i.e. lire une valeur modifiée par une autre transaction mais non validée.
- **READ COMMITTED** : évite les lectures inconsistantes. On ne lit que des données dont les modifications ont été validées.
- **REPEATABLE READ** : empêche le problème des lectures répétées tq entre 2 lectures, certaines lignes ont disparu ou ont été modifiées. N'empêche pas le problème des lignes "fantômes", ajoutées entre une lecture et la suivante.
- **SERIALIZABLE** : Garantit la sériabilité des ordonnancements. Évidemment ça pose des problèmes de performance.

Les transactions sous Postgres

- Une transaction commence par l'instruction `BEGIN TRANSACTION`; ou juste `BEGIN`;
- 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.
- Une erreur (par exemple contrainte non satisfaite) entraîne un `rollback` implicite.
- Une instruction SQL qui n'est pas dans une transaction est automatiquement validée par un `commit` implicite; sauf si elle provoque une erreur, elle est alors annulée par un `rollback` implicite. Elle constitue donc à elle seule une transaction.
- Postgres suit la norme concernant les modes et les niveaux d'isolement possibles.

Niveaux d'isolement des transactions sous Postgres

- La commande `set transaction` permet de fixer le niveau au sein d'une transaction (après instruction `begin`)
Syntaxe Postgres : `set transaction isolation level niveau`
`niveau` peut correspondre à : `SERIALIZABLE`, `REPEATABLE READ`, `READ COMMITTED` ou `READ UNCOMMITTED`
- Comme dans beaucoup de SGBD, le niveau `read committed` est le niveau par défaut.
- Le standard SQL accepte qu'un niveau soit plus strict que la norme, c'est le cas pour Postgres pour les niveaux `Uncommitted read` et `Repeatable read`. Cela implique que les lectures inconsistantes sont impossibles.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Comment fait Postgres ?

- Postgres utilise à la fois du multi-versions et du verrouillage pour que les lectures n'entrent pas en conflit avec les écritures, i.e. une lecture ne bloque pas une écriture et réciproquement une écriture ne bloque pas une lecture.
- A chaque instruction SQL utilisant une table, un verrou est posé sur cette table (le type de verrou dépend de l'instruction exécutée) et éventuellement sur certaines lignes (e.g. celles modifiées). Selon le verrou, d'autres instructions vont pouvoir accéder ou pas à cette table.
- Postgres gère 8 niveaux de verrouillage des tables, et 4 niveaux de verrouillage des lignes. Pour simplifier :
 - Une transaction pose un verrou en lecture quand elle exécute un `SELECT` ;
 - Une transaction pose un verrou en écriture quand elle exécute un `INSERT`, `DELETE`, `UPDATE` ou `SELECT ... FOR UPDATE` ;
 - Les instructions du DDL entraînent aussi la pose de verrous plus restrictifs vis à vis des accès concurrents.
- Si le comportement par défaut ne convient pas, l'utilisateur peut explicitement poser des verrous avec la commande `LOCK TABLE`.
- Le verrouillage dure le temps de la transaction : il prend fin au premier commit ou rollback (commande explicite ou implicite).

Reprise après panne

Il existe un gestionnaire de reprise après panne, qui est responsable de 2 propriétés :

- ① La durabilité : les effets des actions effectuées par les transactions validées au moment de la panne doivent être enregistrées dans la base.
- ② L'atomicité : une transaction se fait entièrement ou pas du tout. Il faut donc défaire le travail des transactions non validées qui étaient actives au moment de la panne.

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 : remettre la base dans l'état qu'elle avait avant la panne,
- Problème physique (panne serveur, crash disque ...) : utiliser des sauvegardes des fichiers.

En fait, le gestionnaire de reprise après panne est aussi responsable de l'exécution des rollbacks.

→ Les procédés mis en oeuvre pour annuler une transaction sont aussi utilisés pendant une reprise après une panne système.

Algorithme ARIES

- ARIES = base des algo de reprise après panne des SGBD actuels.
- Cet algorithme utilise la **journalisation**
- Toutes les actions effectuées par le SGBD sont tracées dans un journal (fichier de log).
- En cas de panne, le journal est utilisé pour remettre la base dans l'état où elle se trouvait au moment de la panne.
- Cet algorithme fonctionne parce que le journal est sauvegardé sur disque à chaque commit, et surtout AVANT que les pages modifiées (données) ne soient écrites sur disque.