Les transactions

Olivier Caron

Polytech Lille
Avenue Paul Langevin
Cité Scientifique, Univ. Lille
59655 Villeneuve d'Ascq cedex

http://ocaron.polytech-lille.net
Olivier.Caron@polytech-lille.fr

Remerciements

Ce cours s'appuie essentiellement sur le cours d'Anne-Cécile Caron de l'Université de Lille.

Pourquoi des transactions ?

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

Pourquoi des transactions ?

- 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.

Pourquoi des transactions ?

- 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.
  - Même en cas de panne logicielle ou matérielle.

Pourquoi des transactions ?

- 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.
  - Même en cas de panne logicielle ou matérielle.
- Plan du cours:

Pourquoi des transactions ?

- 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.
  - Même en cas de panne logicielle ou matérielle.
- Plan du cours:
  - Présentation des problèmes rencontrés (si pas de gestion de transactions)

Pourquoi des transactions ?

- 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.
  - Même en cas de panne logicielle ou matérielle.
- Plan du cours:
  - Présentation des problèmes rencontrés (si pas de gestion de transactions)
  - Concept de transaction

Pourquoi des transactions ?

- 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.
  - Même en cas de panne logicielle ou matérielle.
- Plan du cours:
  - Présentation des problèmes rencontrés (si pas de gestion de transactions)
  - Concept de transaction
  - Fonctionnement des transactions

Pourquoi des transactions ?

- 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.
  - Même en cas de panne logicielle ou matérielle.
- Plan du cours:
  - Présentation des problèmes rencontrés (si pas de gestion de transactions)
  - Concept de transaction
  - Fonctionnement des transactions
  - Particularités du serveur Postgres

Exemple

- On considère le célèbre exemple de virement bancaire :

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

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.

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 ;
```

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

Pour mettre en évidence les problèmes, on s'intéressera aux lectures et écritures faites par une séquence d'instructions. La procédure de virement devient alors :

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

Premier problème

Un utilisateur exécute un virement bancaire :

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

- La base se retrouve dans un état incohérent (on a écrit A et pas B)

Premier problème

Un utilisateur exécute un virement bancaire :

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

- La base se retrouve dans un état incohérent (on a écrit A et pas B)
- Besoin d'un système transactionnel :

Premier problème

Un utilisateur exécute un virement bancaire :

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

- La base se retrouve dans un état incohérent (on a écrit A et pas B)
- Besoin d'un système transactionnel :
  - doit garantir que la transaction se fait complètement ou pas du tout,

Premier problème

Un utilisateur exécute un virement bancaire :

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

- La base se retrouve dans un état incohérent (on a écrit A et pas B)
- Besoin d'un système transactionnel :
  - doit garantir que la transaction se fait complètement ou pas du tout,
  - doit donc annuler la modification de A.

Premier problème

Un utilisateur exécute un virement bancaire :

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

- La base se retrouve dans un état incohérent (on a écrit A et pas B)
- Besoin d'un système transactionnel :
  - doit garantir que la transaction se fait complètement ou pas du tout,
  - doit donc annuler la modification de A.
  - Une séquence d'instructions dans une transaction est *Atomique*.

Accès concurrents

Deux utilisateurs exécutent des virements de mêmes comptes, à l'aide de deux séquences d'instructions S1 et S2 :

```
S1 : virement(A,B,100)
S2 : virement(A,B,200)
```

Pour des raisons de performance, les actions des différentes séquences sont entrelacées.
Il faut différencier les actions de S1 de celles de S2, et considérer l'ordre dans lequel ces actions vont s'exécuter.
Pour cet exemple, les soldes de départ de A et B sont respectivement de 500 et 300 Euros.

Ordonnancement

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

Table: Exemple d'ordonnancement O_1 de S1 et S2

Ordonnancement	Action	Solde A	solde B
(S1, lire, A)	lecture A:500	500	300
(S1, ecrire, A)	écrire A:500-100	400	300
(S2, lire, A)	lire A:400	400	300
(S2, ecrire, A)	écrire A:400-200	200	300
(S1, lire, B)	lire B:300	200	300
(S1, ecrire, B)	écrire B:300+100	200	400
(S2, lire, B)	lire B:400	200	400
(S2, ecrire, B)	ecrire B:400+200	200	600

Deuxième problème

Table: Exemple d'ordonnancement O_2 de S1 et S2

Ordonnancement	Action	Solde A	solde B
(S1, lire, A)	lecture A:500	500	300
(S2, lire, A)	lire A:500	500	300
(S1, ecrire, A)	écrire A:500-100	400	300
(S1, lire, B)	lire B:300	400	300
(S1, ecrire, B)	écrire B:300+100	400	400
(S2, ecrire, A)	écrire A:500-200	300	400
(S2, lire, B)	lire B:400	300	400
(S2, ecrire, B)	ecrire B:400+200	**300**	600

Ici, on a perdu une instruction de A et la base est dans un état *inconsistant*.

Les effets des séquences sont modifiés à cause de la *concurrence*.

Autre problème : lectures non consistantes

Table: exemple lectures successives

Séquence 1		Séquence 2	
Action	**Résultat**	**Action**	**Résultat**
lire(A)	affiche A:500		
		écrire(A := 200)	A vaut 200
lire(A)	affiche A:200 !		

Ici, on lit 2 fois une même donnée A (séquence 1) et on obtient des résultats différents.

Propriétés des ordonnancements (1/2)

- Deux actions d'un ordonnancement sont *conflictuelles* si elles concernent la même entité et qu'au moins l'une des deux est une écriture.

Propriétés des ordonnancements (1/2)

- 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 séquences 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.

Propriétés des ordonnancements (2/2)

- Un ordonnancement est *sérialisable* s'il existe un ordre entre les séquences tel que toutes les opérations d'une séquence S qui sont en conflit avec les opérations d'une autre séquence S', sont exécutées après ces opérations.

Propriétés des ordonnancements (2/2)

- Un ordonnancement est *sérialisable* s'il existe un ordre entre les séquences tel que toutes les opérations d'une séquence S qui sont en conflit avec les opérations d'une autre séquence S', sont exécutées après ces opérations.
- Seuls les ordonnancements *sérialisables* sont corrects.

Propriétés des ordonnancements (2/2)

- Un ordonnancement est *sérialisable* s'il existe un ordre entre les séquences tel que toutes les opérations d'une séquence S qui sont en conflit avec les opérations d'une autre séquence S', sont exécutées après ces opérations.
- Seuls les ordonnancements *sérialisables* sont corrects.

Exemple

Propriétés des ordonnancements (2/2)

- Un ordonnancement est *sérialisable* s'il existe un ordre entre les séquences tel que toutes les opérations d'une séquence S qui sont en conflit avec les opérations d'une autre séquence S', sont exécutées après ces opérations.
- Seuls les ordonnancements *sérialisables* sont corrects.

Exemple

- L'ordonnancement O_1 est sérialisable car équivalent à `S1;S2`.

Propriétés des ordonnancements (2/2)

- Un ordonnancement est *sérialisable* s'il existe un ordre entre les séquences tel que toutes les opérations d'une séquence S qui sont en conflit avec les opérations d'une autre séquence S', sont exécutées après ces opérations.
- Seuls les ordonnancements *sérialisables* sont corrects.

Exemple

- L'ordonnancement O_1 est sérialisable car équivalent à `S1;S2`.
- L'ordonnancement O_2 n'est pas sérialisable.

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.
- Elle doit (devrait) respecter les propriétés **ACID**

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.
- Elle doit (devrait) respecter les propriétés **ACID**

  Atomicité Une transaction s'effectue entièrement ou pas du tout

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.
- Elle doit (devrait) 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.

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.
- Elle doit (devrait) 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.

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.
- Elle doit (devrait) 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 prises en compte dans la base de données.

Synthèse : concept de transaction (2/2)

- Les transactions sont gérées par un moniteur transactionnel.

Synthèse : concept de transaction (2/2)

- Les transactions sont gérées par un moniteur transactionnel.
- Une transaction peut être dans différents états :

Synthèse : concept de transaction (2/2)

- 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.

Synthèse : concept de transaction (2/2)

- 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

Synthèse : concept de transaction (2/2)

- 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

Synthèse : concept de transaction (2/2)

- 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
  - Echouée : après un problème qui a interrompu la transaction

Instructions SQL de base

- Débuter une transaction : `begin`, mémorisation de l'état supposé cohérent de la base.

Instructions SQL de base

- Débuter une transaction : `begin`, mémorisation de l'état supposé cohérent de la base.
- Valider une transaction : `commit`.
  Cette instruction permet de signaler que la transaction s'est bien terminée, les modifications qu'elle a effectuées sont rendues visibles aux autres transactions.

Instructions SQL de base

- Débuter une transaction : `begin`, mémorisation de l'état supposé cohérent de la base.
- Valider une transaction : `commit`.
  Cette instruction permet de signaler que la transaction s'est bien terminée, les modifications qu'elle a effectuées sont rendues visibles aux autres transactions.
- Annuler *explicitement* une transaction : `rollback` ou `abort`.
  Toutes les commandes depuis le début de la transaction sont annulées.

Instructions SQL de base

- Débuter une transaction : `begin`, mémorisation de l'état supposé cohérent de la base.
- Valider une transaction : `commit`.
  Cette instruction permet de signaler que la transaction s'est bien terminée, les modifications qu'elle a effectuées sont rendues visibles aux autres transactions.
- Annuler *explicitement* une transaction : `rollback` ou `abort`.
  Toutes les commandes depuis le début de la transaction sont annulées.
- Si une transaction échoue (erreur syntaxe SQL, droits, ...), un `rollback` *implicite* permet d'annuler toutes les actions effectuées par cette transaction. La transaction est automatiquement arrêtée.

Isolation des transactions (1/2)

- Le standard SQL définit 4 niveaux d'isolation des transactions.

Isolation des transactions (1/2)

- Le standard SQL définit 4 niveaux d'isolation des transactions.
- Les niveaux diffèrent par les **phénomènes** :

Isolation des transactions (1/2)

- Le standard SQL définit 4 niveaux d'isolation des transactions.
- Les niveaux diffèrent par les **phénomènes** :

Lecture sale Une transaction lit les données écrites par une transaction concurrente non validée (dirty read)

Isolation des transactions (1/2)

- Le standard SQL définit 4 niveaux d'isolation des transactions.
- Les niveaux diffèrent par les **phénomènes** :

Lecture sale Une transaction lit les données écrites par une transaction concurrente non validée (dirty read)

Lecture non reproductible Une transaction relit des données qu'elle a lues précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale) (non repeatable read).

Isolation des transactions (1/2)

- Le standard SQL définit 4 niveaux d'isolation des transactions.
- Les niveaux diffèrent par les **phénomènes** :

Lecture sale Une transaction lit les données écrites par une transaction concurrente non validée (dirty read)

Lecture non reproductible Une transaction relit des données qu'elle a lues précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale) (non repeatable read).

Lecture fantôme Une transaction ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée (phantom read).

Isolation des transactions (2/2)

- Plus le niveau est strict, moins les phénomènes sont possibles.

Isolation des transactions (2/2)

- Plus le niveau est strict, moins les phénomènes sont possibles.
- Le niveau le plus strict est `Serializable` qui doit garantir que l'exécution concurrente des transactions doit produire le même effet que l'exécution consécutive de chacune d'entre elles.

Niveau d'isolation	**Lecture sale**	**lecture non reproductible**	**lecture fantôme**
Uncommited Read : lecture de données non validées	possible pas sous Postgres	possible	possible
Commited Read : lecture de données validées	Impossible	possible	possible
Repeatable Read : lecture répétée	Impossible	Impossible	possible pas sous Postgres
Serializable : sérialisable	Impossible	Impossible	Impossible

Niveaux d'isolation 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`

Niveaux d'isolation 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`
- Le niveau `read commited` est le niveau par défaut.

Niveaux d'isolation 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`
- Le niveau `read commited` 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 `Uncommited read` et `Repeatable read`. Cela implique que les "dirty read" sont impossibles sous Postgres.

Illustration des phénomènes

Soit la table SQL `Emp` contenant une seule ligne au départ :

Table: table `Emp`

Id	nom
1	dupont

Exemple de lecture sale (”dirty read”) et lecture non reproductible

- Rappel: ce cas est impossible sous Postgres

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
set transaction isolation level read uncommited		set transaction isolation level read uncommited	
select * from emp	(1, ’dupont’)	insert into emp values(3,’durant’)	insert 1
select * from emp	(1, ’dupont’), (3,’durant’)		
		commit	
select * from emp	(1, ’dupont’), (3,’durant’)		
commit			

Exemple de lecture non reproductible

- Sous Postgres, la commande `set transaction` de cet exemple est facultatif (`Read commited` est le mode par défaut)

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
set transaction isolation		set transaction isolation	
level read commited		level read commited	
select * from emp	(1,'dupont'),(3,'durant')	insert into emp values(4,'dubois')	insert 1
select * from emp	(1,'dupont'),(3,'durant')		
		commit	
select * from emp	(1,'dupont'),(3,'durant'), (4,'dubois')		
commit			

Exemple de Lecture fantôme

- Rappel: ce cas est impossible sous Postgres avec `repeatable read`

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
set transaction isolation level repeatable read		set transaction isolation level repeatable read	
select * from emp where id between 1 and 3	(1,'dupont'),(3,'durant')	insert into emp values(2,'garcia')	insert 1
select * from emp where id between 1 and 3	(1,'dupont'),(3,'durant')	commit	
select * from emp between 1 and 3	(1,'dupont'),(2,'garcia'),(3,'durant')		
commit			

Implémentation des transactions

- Plusieurs solutions techniques permettent de gérer les transactions :

Implémentation des transactions

- Plusieurs solutions techniques permettent de gérer les transactions :

  Méthode pessimiste la méthode dite de *verrouillage* est une méthode préventive de conflits. Chaque instruction pose des verrous sur les données lues/écrites. Deux verrous conflictuels ne peuvent pas être accordés en même temps (transaction mise en attente dans ce cas).

Implémentation des transactions

- Plusieurs solutions techniques permettent de gérer les transactions :

  Méthode pessimiste la méthode dite de *verrouillage* est une méthode préventive de conflits. Chaque instruction pose des verrous sur les données lues/écrites. Deux verrous conflictuels ne peuvent pas être accordés en même temps (transaction mise en attente dans ce cas).

  Méthodes optimistes ces méthodes laissent les conflits se produire et les corrigent après. Plusieurs solutions existent telles que l'ordonnancement par estampille, la certification et les contrôles avec versions multiples.

Verrouillage vs méthodes optimistes (1/2)

Verrouillage

Verrouillage vs méthodes optimistes (1/2)

Verrouillage

- Le verrouillage utilise 2 techniques : le blocage et l'interruption des transactions.
  Ces 2 mécanismes pénalisent les performances.

Verrouillage vs méthodes optimistes (1/2)

Verrouillage

- 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.

Verrouillage vs méthodes optimistes (1/2)

Verrouillage

- 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.

Verrouillage vs méthodes optimistes (1/2)

Verrouillage

- 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.
- Statistiquement, très peu de conflits interviennent.

Verrouillage vs méthodes optimistes (2/2)

Méthodes optimistes

Verrouillage vs méthodes optimistes (2/2)

Méthodes optimistes

- Les méthodes dites optimistes partent du principe qu'il y aura très peu de conflits :

Verrouillage vs méthodes optimistes (2/2)

Méthodes optimistes

- Les méthodes dites optimistes partent 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

Verrouillage vs méthodes optimistes (2/2)

Méthodes optimistes

- Les méthodes dites optimistes partent 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

Verrouillage vs méthodes optimistes (2/2)

Méthodes optimistes

- Les méthodes dites optimistes partent 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.

Les verrous sous Postgres

- Des verrous sont automatiquement posés lors de l'exécution des commandes SQL.

Les verrous sous Postgres

- Des verrous sont automatiquement posés lors de l'exécution des commandes SQL.
- les verrous ne sont libérés que lors de la fin de transaction.

Les verrous sous Postgres

- Des verrous sont automatiquement posés lors de l'exécution des commandes SQL.
- les verrous ne sont libérés que lors de la fin de transaction.
- Possibilité de poser explicitement des verrous (commande `lock`)

Les verrous sous Postgres

- Des verrous sont automatiquement posés lors de l'exécution des commandes SQL.
- les verrous ne sont libérés que lors de la fin de transaction.
- Possibilité de poser explicitement des verrous (commande `lock`)
- Il existe des verrous niveau ligne et des verrous niveau table

Les verrous niveau table de Postgres (V. 9.5) (1/2)

Access Share (AS) La commande `select` pose un verrou de ce mode sur les tables référencées.

Les verrous niveau table de Postgres (V. 9.5) (1/2)

Access Share (AS) La commande `select` pose un verrou de ce mode sur les tables référencées.

Row Share (RS) Les commandes `select for update` et `select for share` posent un verrou de ce mode avec la table cible (en plus des verrous AS).

Les verrous niveau table de Postgres (V. 9.5) (1/2)

Access Share (AS) La commande `select` pose un verrou de ce mode sur les tables référencées.

Row Share (RS) Les commandes `select for update` et `select for share` posent un verrou de ce mode avec la table cible (en plus des verrous AS).

Row Exclusive (RE) verrous posés sur les tables cibles lors de modifications via les commandes `update`, `delete` et `insert`.

Les verrous niveau table de Postgres (V. 9.5) (1/2)

Access Share (AS) La commande `select` pose un verrou de ce mode sur les tables référencées.

Row Share (RS) Les commandes `select for update` et `select for share` posent un verrou de ce mode avec la table cible (en plus des verrous AS).

Row Exclusive (RE) verrous posés sur les tables cibles lors de modifications via les commandes `update`, `delete` et `insert`.

Share Update Exclusive (SUE) verrous posés par des modifications du schémas (commande `alter table` notamment).

Les verrous niveau table de Postgres (V. 9.5) (2/2)

Share (S) verrou posé par la commande `create index`.

Les verrous niveau table de Postgres (V. 9.5) (2/2)

Share (S) verrou posé par la commande `create index`.

Share Row Exclusive (SRE) verrou posé par la commande `create trigger` et différentes formes de `alter table`.

Les verrous niveau table de Postgres (V. 9.5) (2/2)

Share (S) verrou posé par la commande `create index`.

Share Row Exclusive (SRE) verrou posé par la commande `create trigger` et différentes formes de `alter table`.

Exclusive (E) posé par `Refresh materialized view concurrently`

Les verrous niveau table de Postgres (V. 9.5) (2/2)

Share (S) verrou posé par la commande `create index`.

Share Row Exclusive (SRE) verrou posé par la commande `create trigger` et différentes formes de `alter table`.

Exclusive (E) posé par `Refresh materialized view concurrently`

Access Exclusive (AE) posé par plusieurs commandes de modification de schéma telles que `drop table` et `alter table`.

Les verrous conflictuels de niveau table

Verrou demandé	**Verrou déjà détenu**							
	AS	RS	RE	SUE	S	SRE	E	AE
Access **S**hare								x
Row **S**hare							x	x
Row **E**xclusive					x	x	x	x
Share **U**pdate **E**xclusive				x	x	x	x	x
Share			x	x		x	x	x
Share **R**ow **E**xclusive			x	x	x	x	x	x
Exclusive		x	x	x	x	x	x	x
Access **E**xclusive	x	x	x	x	x	x	x	x

Illustrations verrous table

- Exemple 1:

Illustrations verrous table

- Exemple 1:
  1. Une transaction $t1$ réalise une commande `select` sur une table X, un verrou `Access share` est donc positionné sur X.

Illustrations verrous table

- Exemple 1:
  1. Une transaction $t1$ réalise une commande `select` sur une table X, un verrou `Access share` est donc positionné sur X.
  2. Une transaction concurrente $t2$ essaye de supprimer la table X avec la commande `drop table`. Le verrou demandé est donc `Access Exclusive` qui rentre en conflit avec le verrou `Access share` déjà détenu. La commande `drop table` est bloquée en attente de libération du verrou `Access share` (à la fin de la transaction $t1$).

Illustrations verrous table

- Exemple 1:
  1. Une transaction $t1$ réalise une commande `select` sur une table X, un verrou `Access share` est donc positionné sur X.
  2. Une transaction concurrente $t2$ essaye de supprimer la table X avec la commande `drop table`. Le verrou demandé est donc `Access Exclusive` qui rentre en conflit avec le verrou `Access share` déjà détenu. La commande `drop table` est bloquée en attente de libération du verrou `Access share` (à la fin de la transaction $t1$).
- Exemple 2:

Illustrations verrous table

- Exemple 1:
  1. Une transaction $t1$ réalise une commande `select` sur une table X, un verrou `Access share` est donc positionné sur X.
  2. Une transaction concurrente $t2$ essaye de supprimer la table X avec la commande `drop table`. Le verrou demandé est donc `Access Exclusive` qui rentre en conflit avec le verrou `Access share` déjà détenu. La commande `drop table` est bloquée en attente de libération du verrou `Access share` (à la fin de la transaction $t1$).
- Exemple 2:
  1. Une transaction $t1$ réalise une commande `update` sur une ligne de la table X, un verrou `Row Exclusive` est donc positionné sur X.

Illustrations verrous table

- Exemple 1:
  1. Une transaction $t1$ réalise une commande `select` sur une table X, un verrou `Access share` est donc positionné sur X.
  2. Une transaction concurrente $t2$ essaye de supprimer la table X avec la commande `drop table`. Le verrou demandé est donc `Access Exclusive` qui rentre en conflit avec le verrou `Access share` déjà détenu. La commande `drop table` est bloquée en attente de libération du verrou `Access share` (à la fin de la transaction $t1$).
- Exemple 2:
  1. Une transaction $t1$ réalise une commande `update` sur une ligne de la table X, un verrou `Row Exclusive` est donc positionné sur X.
  2. Une transaction concurrente $t2$ essaye de réaliser exactement la même opération. Le verrou demandé est donc `Row Exclusive` qui n'est pas un verrou conflictuel avec `Row Exclusive` d'après le tableau des conflits de verrou de table !!

Retour sur Exemple 2

- Des modifications concurrentes sur des lignes **différentes** d'une même table ne posent pas de problème $\rightarrow$ ce qui explique que le verrou de table `Row Exclusive` n'est pas conflictuel avec lui-même.

Retour sur Exemple 2

- Des modifications concurrentes sur des lignes **différentes** d'une même table ne posent pas de problème $\rightarrow$ ce qui explique que le verrou de table `Row Exclusive` n'est pas conflictuel avec lui-même.
- Deux modifications concurrentes sur des mêmes valeurs dans deux transactions posent par contre problème (cas de l'exemple 2).

Retour sur Exemple 2

- Des modifications concurrentes sur des lignes **différentes** d'une même table ne posent pas de problème $\rightarrow$ ce qui explique que le verrou de table `Row Exclusive` n'est pas conflictuel avec lui-même.
- Deux modifications concurrentes sur des mêmes valeurs dans deux transactions posent par contre problème (cas de l'exemple 2).
- **Question :** Par quel moyen, Postgres bloque le second `update` de l'exemple 2 ?

Retour sur Exemple 2

- Des modifications concurrentes sur des lignes **différentes** d'une même table ne posent pas de problème $\rightarrow$ ce qui explique que le verrou de table `Row Exclusive` n'est pas conflictuel avec lui-même.
- Deux modifications concurrentes sur des mêmes valeurs dans deux transactions posent par contre problème (cas de l'exemple 2).
- **Question :** Par quel moyen, Postgres bloque le second `update` de l'exemple 2 ?
- **Réponse :** à l'aide des verrous de **niveau ligne**

Verrous niveau ligne Postgres (1/2)

For Update verrouille pour modification les lignes récupérées par l'instruction `select for update` et les lignes modifiées par les instructions `delete` et `update`.

Verrous niveau ligne Postgres (1/2)

For Update verrouille pour modification les lignes récupérées par l'instruction `select for update` et les lignes modifiées par les instructions `delete` et `update`.

For No Key Update ce verrou est posé par la commande `select for no key update`. Ce verrou similaire à `for update` est moins strict car il ne bloque pas les commandes `select for key share`.

Verrous niveau ligne Postgres (2/2)

For Share ce verrou est posé par la commande `select for share`. il a un comportement similaire à `For No Key Update`, sauf qu'il obtient un verrou partagé plutôt qu'un verrou exclusif sur chaque ligne récupérée. Un verrou partagé bloque les autres transactions réalisant des `update, delete, select for update` et `select for no key update` sur ces lignes mais il n'empêche pas les `select for share` et `select for key share`.

Verrous niveau ligne Postgres (2/2)

For Share ce verrou est posé par la commande `select for share`. il a un comportement similaire à `For No Key Update`, sauf qu'il obtient un verrou partagé plutôt qu'un verrou exclusif sur chaque ligne récupérée. Un verrou partagé bloque les autres transactions réalisant des `update, delete, select for update` et `select for no key update` sur ces lignes mais il n'empêche pas les `select for share` et `select for key share`.

For Key Share verrou plus faible que `For Share` : `select for update` est bloqué alors que `select for no key update` ne l'est pas.

Les verrous conflictuels de niveau ligne

Verrou demandé	**Verrou déjà détenu**			
	For Key Share	For Share	For No Key Update	For Update
For Key Share				x
For Share			x	x
For No Key Update		x	x	x
For Update	x	x	x	x

1. Illustration: une transaction $t1$ réalise une commande `update` sur une ligne i de la table X, un verrou ligne `For Update` est donc positionné sur la ligne concernée i de X.

Les verrous conflictuels de niveau ligne

Verrou demandé	**Verrou déjà détenu**			
	For Key Share	For Share	For No Key Update	For Update
For Key Share				x
For Share			x	x
For No Key Update		x	x	x
For Update	x	x	x	x

1. Illustration: une transaction $t1$ réalise une commande `update` sur une ligne i de la table X, un verrou ligne `For Update` est donc positionné sur la ligne concernée i de X.
2. Une transaction concurrente $t2$ essaye de réaliser exactement la même opération sur la même ligne. Le verrou demandé est donc `For Update` qui entre en conflit avec le `For Update` de $t1$. La transaction est bloquée dans l'attente de la fin de $t1$

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.
- Les solutions d'implémentation diffèrent d'un éditeur de SGBD à un autre.

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.
- Les solutions d'implémentation diffèrent d'un éditeur de SGBD à un autre.
- Tout n'est pas standardisé (ex: `select for no key update`).

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.
- Les solutions d'implémentation diffèrent d'un éditeur de SGBD à un autre.
- Tout n'est pas standardisé (ex: `select for no key update`).
- La version sûre "Serializable" n'est pas la plus performante.

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.
- Les solutions d'implémentation diffèrent d'un éditeur de SGBD à un autre.
- Tout n'est pas standardisé (ex: `select for no key update`).
- La version sûre "Serializable" n'est pas la plus performante.
- Besoin de trouver le bon compromis fiabilité/performances

Conclusion

- Bien d'autres possibilités (ex: verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.
- Les solutions d'implémentation diffèrent d'un éditeur de SGBD à un autre.
- Tout n'est pas standardisé (ex: `select for no key update`).
- La version sûre "Serializable" n'est pas la plus performante.
- Besoin de trouver le bon compromis fiabilité/performances
- Nécessite de comprendre finement les rouages du SGBD