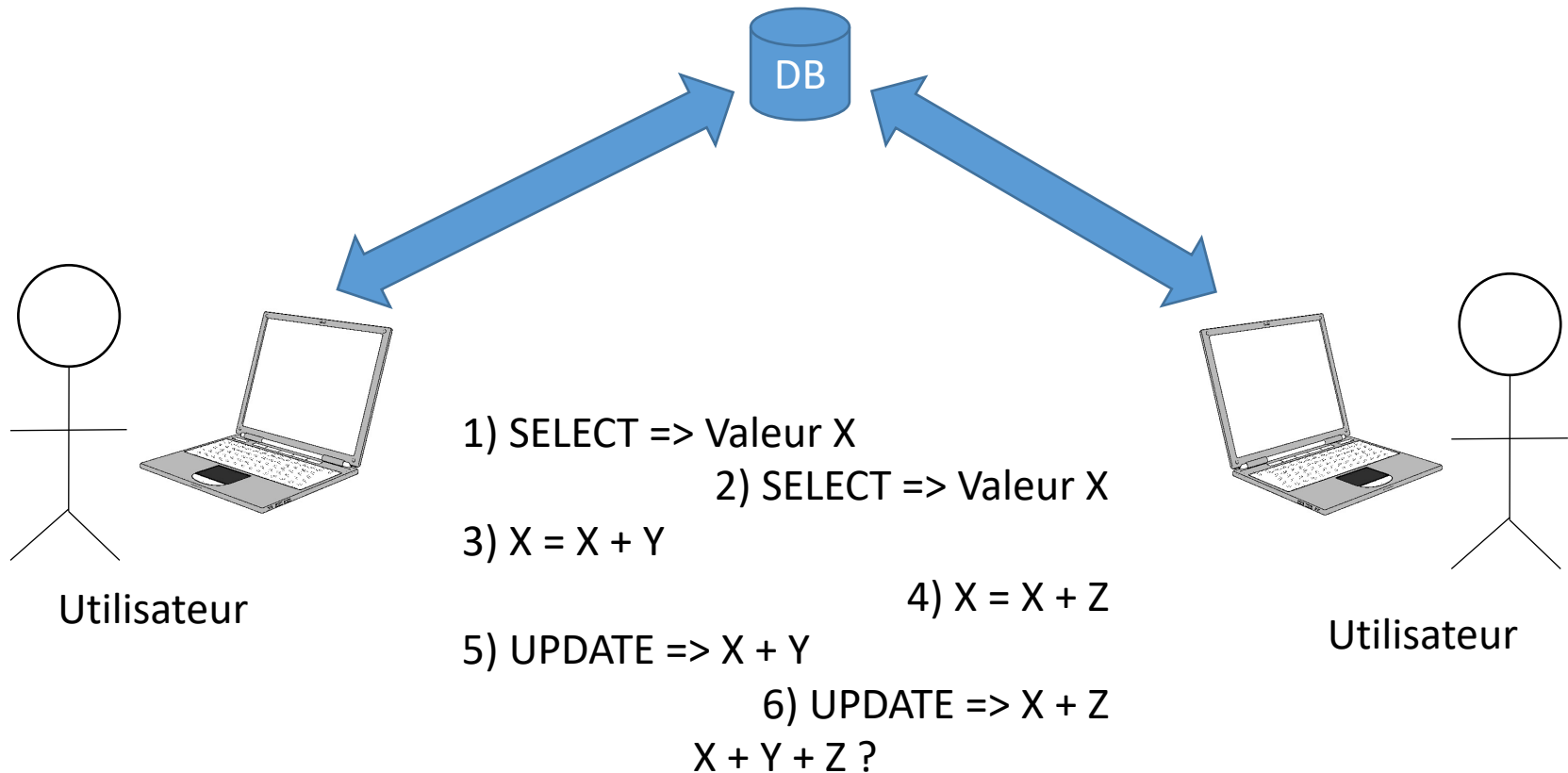


Bases de données (organisation générale)

Répétition 8

Les transactions

Les transactions: pourquoi?



Les verrous explicites

- Verrous sur les tables :
 - *LOCK TABLE <table> READ* : Verrouille l'entièreté de la table en lecture. Tout le monde peut lire, mais personne ne peut écrire.
 - *LOCK TABLE <table> WRITE* : Verrouille l'entièreté de la table en écriture. La session qui obtient le verrou peut lire et écrire, mais les autres sessions doivent attendre, même pour lire.
 - *UNLOCK TABLES* : Enlève l'ensemble des verrous posés (toutes les tables)
- Verrous sur les tuples. Dans une transaction (où autocommit = 0)
 - *SELECT ... LOCK IN SHARE MODE* : Permet la lecture et bloque les autres sessions s'ils essaient d'écrire sur les tuples sélectionnés
 - *SELECT ... FOR UPDATE* : Idem, mais un deuxième appel est, cette-fois, bloquant (évite le problème de la modification simultanée)
 - Déblocage avec *COMMIT* (ou *ROLLBACK*)

Exercice 1

Une base de données bancaire contient les relations suivantes :

- *client*(*ID*,*NOM*,*PRENOM*,*ADRESSE*) contenant la liste des clients ;
- *compte*(*IBAN*, *SOLDE*, *#ID*) reprenant la liste des comptes bancaires et leur titulaire ;
- *transaction*(*NUM*,*#IBANFROM*, *#IBANTO*, *MONTANT*, *DATE*) reprenant la liste des transactions effectuées ;

1) En utilisant LOCK TABLES, écrivez un script pour une transaction qui effectuerait un transfert d'un compte vers un autre.

Exercice 1

LOCK TABLE compte **WRITE**;

SELECT solde FROM compte WHERE IBAN = <compte1>;

Si solde < montant à transférer => **UNLOCK TABLES**;

Sinon

LOCK TABLE transaction **WRITE**;

UPDATE compte SET solde = solde + x WHERE IBAN = <compte2>;

UPDATE compte SET solde = solde - x WHERE IBAN = <compte1>;

INSERT INTO transaction VALUES ((SELECT DISTINCT MAX(NUM)+1 from transaction),<compte1>,<compte2>,montant,NOW());

UNLOCK TABLES;

1 seule transaction à la fois, même si des transactions concurrentes ne travailleraient pas sur les mêmes tuples.

Exercice 1

- Pourquoi bloquer la table en écriture pour faire la première lecture de la table compte?
 - Je pourrais obtenir un lock READ puis un lock WRITE si tout va bien, mais:
 - Je suis quoi qu'il arrive bloqué pendant qu'un autre a le lock READ
 - Je dois refaire un SELECT pour vérifier qu'entre-temps, la valeur n'a pas été modifiée
- Est-ce que je ne risque pas d'avoir un deadlock?
 - Non, car les tables sont, dans cet exemple, verrouillées toujours dans le même ordre.

Exercice 1

Une base de données bancaire contient les relations suivantes :

- *client*(*ID*,*NOM*,*PRENOM*,*ADRESSE*) contenant la liste des clients ;
- *compte*(*IBAN*, *SOLDE*, *#ID*) reprenant la liste des comptes bancaires et leur titulaire ;
- *transaction*(*NUM*,*#IBANFROM*, *#IBANTO*, *MONTANT*, *DATE*) reprenant la liste des transactions effectuées ;

2) Améliorez votre script pour que plusieurs transactions puissent travailler simultanément sur la base de données.

Exercice 1

Je vais utiliser FOR UPDATE pour ne bloquer que les tuples sur lesquels je vais travailler.

START TRANSACTION;


SELECT solde, IBAN FROM compte WHERE IBAN = <compte1> OR IBAN = <compte2> **FOR UPDATE;**

*Si solde de compte1 < montant à transférer => **ROLLBACK;***

Sinon

UPDATE compte SET solde = solde + x WHERE IBAN = <compte2>;

UPDATE compte SET solde = solde - x WHERE IBAN = <compte1>;

 SELECT DISTINCT MAX(NUM)+1 FROM transaction; => *NewID*

SELECT * FROM transaction WHERE NUM = <NewID> **FOR UPDATE;**

Tant que le dernier SELECT renvoie un tuple, recommencer à 

INSERT INTO transaction VALUES (<NewID>, <compte1>, <compte2>, montant, NOW());

Si deadlock, recommencer à 

COMMIT;

Exercice 1

- Pourquoi le premier SELECT n'est pas LOCK IN SHARE MODE plutôt que FOR UPDATE
 - Idem que précédemment. Je dois de toutes façons modifier après.
- Pourquoi bloquer les deux comptes d'entrée de jeu?
 - Parce que si on tente de prendre deux verrous séparément, on pourrait se retrouver dans un deadlock
- Pourquoi la boucle?
 - SELECT... FOR UPDATE donne un verrou sur les tuples sélectionnés
 - Mais nous voulons bloquer un tuple qui n'existe pas encore!
 - Dès lors, il faut trouver un valeur pour laquelle aucun tuple n'existe, et obtenir le verrou pour cette valeur
 - Cela peut prendre plusieurs essais.
 - Et échouer dans le pire des cas (deadlock détecté), il faut alors relancer la transaction.
 - Dans ce cas particulier où l'ID est unique et pour lequel on attribue une valeur croissante, il est préférable d'utiliser un champ AUTO_INCREMENT

Exercice 1

Une base de données bancaire contient les relations suivantes :

- *client*(*ID,NOM,PRENOM,ADRESSE*) contenant la liste des clients ;
- *compte*(*IBAN, SOLDE, #ID*) reprenant la liste des comptes bancaires et leur titulaire ;
- *transaction*(*NUM,#IBANFROM, #IBANTO, MONTANT, DATE*) reprenant la liste des transactions effectuées ;

3) Ouvrez un compte épargne gratuit a 500 clients, tirés au hasard. Travaillez avec 5 transactions en parallèle pour vous répartir la charge (le code est identique aux 5 transactions). Attention a ne pas ouvrir plusieurs comptes gratuits pour le même client.

Exercice 1

A prendre en considération:

- Je ne dois pas ouvrir plusieurs comptes gratuits à un même client
- Le compte gratuit n'est pas différenciable des autres comptes
- Rien, dans ma base de données, ne me permet de stocker l'information sur le fait que j'ai ouvert un compte gratuit pour un client donné

→ J'ai besoin d'une nouvelle table temporaire pour stocker cette information

Remplir la table temporaire

- Premier essai:
 - A chaque fois que je crée un compte, j'inscris le numéro du client dans cette table
 - Avant de créer un compte, je consulte la table pour voir si je n'ai pas déjà ouvert de compte pour ce client.
 - Je protège cette table:
 - LOCK TABLES : Ne me permettra pas de travailler en réelle concurrence
 - SELECT ... FOR UPDATE : Ne marche pas très bien pour les insertions.

Exercice 1

- Deuxième essai:

Créez une table temporaire avec deux attributs : ID et IBAN, remplie de 500 numéros IBAN uniques et avec des ID vides (appelons-la Temp).

```
SELECT ID FROM client ORDER BY RAND() LIMIT 500;
```

Remplir la table Temp avec les ID

Chacune des 5 transactions :

START TRANSACTION;

```
SELECT ID, IBAN FROM Temp LIMIT 100 OFFSET <trans.ID>*100; %trans.ID dans [0,4]
```

Pour chaque ligne retournée

```
INSERT INTO compte VALUES(<IBAN>,0,<ID>);
```

COMMIT;

En réalité, je n'ai même plus besoin de protéger mes données, puisque chaque transaction fonctionne sur des données distinctes

Les transactions ne sont plus *parfaitement* identiques (numéro de transaction pour les différencier)

Exercice 1

Une base de données bancaire contient les relations suivantes :

- *client*(*ID*,*NOM*,*PRENOM*,*ADRESSE*) contenant la liste des clients ;
- *compte*(*IBAN*, *SOLDE*, *#ID*) reprenant la liste des comptes bancaires et leur titulaire ;
- *transaction*(*NUM*,*#IBANFROM*, *#IBANTO*, *MONTANT*, *DATE*) reprenant la liste des transactions effectuées ;

4) Ecrivez un script vous permettant d'encoder un nouveau client, sachant que l'ID est unique, va croissant pour les clients, et que si N est le numéro maximum pour ID, alors tous les numéros strictement inférieurs à N et plus grands que 0 sont utilisés dans la bases de données. La politique de la maison est d'annoncer à la personne son futur numéro client au début de l'entretien. On propose une tasse de café au nouveau client pour l'accueillir.

Exercice 1 : Mauvaise méthode

START TRANSACTION;

Choisir le prochain numéro (voir transparent 8)

SELECT * FROM client WHERE ID = <ID> **FOR UPDATE;**

Prendre les infos du client. Lui laisser boire sa tasse de café

INSERT INTO client VALUES (<ID>,<Nom>,<Prénom>,<Adresse>)

COMMIT;

Il est toujours possible qu'une autre transaction tente de prendre le même numéro ID (deadlock détecté, on doit recommencer la transaction).

Exercice 1 : Mauvaise méthode (2)

LOCK TABLE client **WRITE**;

SELECT MAX(ID) FROM client;

Prendre les infos du client. Lui laisser boire sa tasse de café

INSERT INTO client VALUES
(<ID+1>,<Nom>,<Prénom>, <Adresse>)

UNLOCK TABLES;

La table est verrouillée pendant que le client prend son café.

Exercice 1 : Bonne méthode

```
LOCK TABLE client READ;  
SELECT MAX(ID) FROM client;  
UNLOCK TABLES;
```

Le client boit son café.

```
LOCK TABLE client WRITE;  
SELECT MAX(ID) FROM client;  
INSERT INTO client VALUES (<ID+1>,<Nom>,<Prénom>, <Adresse>)  
UNLOCK TABLES;
```

On sépare la phase d'acquisition de la phase de traitement utilisateur. On refait une petite transaction après.

Le numéro de séquence peut changer entre temps, mais la table ne reste pas verrouillée.

Exercice 1 : Autre bonne méthode

```
LOCK TABLE client WRITE;  
SELECT MAX(ID) FROM client; (=> NewID)  
INSERT INTO client VALUES (NewID,"","");  
UNLOCK TABLES;
```

Le client boit son café.

```
LOCK TABLE client WRITE;  
UPDATE client SET Nom=<Nom>, Prenom=<Prénom>, Adresse=<Adresse> WHERE  
ID = NewID  
UNLOCK TABLES;
```

Ici, on réserve le numéro de client pour plus tard.

Si le client ne veut plus s'inscrire, il faut supprimer le numéro et changer le SELECT pour obtenir le premier numéro non utilisé.

Les niveaux d'isolation des transactions

- Plutôt que de jouer avec le blocage par SELECT, il est possible d'utiliser le blocage par transaction, en spécifiant le niveau d'isolation (SET TRANSACTION ISOLATION LEVEL).
- 4 valeurs possibles:
 - READ UNCOMMITTED : En fait, aucune protection
 - READ COMMITTED : Ne lira les données que si elles sont validées (pas de lecture pendant une transaction)
 - REPEATABLE READ : Ne lit que des données validées, et des lectures répétées donnent le même résultat
 - SERIALIZABLE : Les transactions peuvent être sérialisées et s'exécuter dans n'importe quel ordre, la transaction apparaît comme atomique.

Exercice 2

Une base de données contient les valeurs suivantes :

COL
1
2
3

Imaginez les scenarios suivants :

1. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;
(Transaction 2) : ROLLBACK;

2. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

3. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : INSERT INTO TABLE VALUES (4) ;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Donnez, pour chaque scenario, le résultat affiché par le dernier SELECT de la transaction, selon que le niveau d'isolation des transactions est, respectivement :

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Read uncommitted

1. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;
(Transaction 2) : ROLLBACK;

La transaction 1 n'attend pas
la transaction 2 → Affiche -3

2. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Idem, Affiche -3

3. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : INSERT INTO TABLE VALUES (4) ;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Idem, Affiche -4

Read uncommitted → aucune protection de quoi que ce soit.
Lectures "sales" possibles

Read committed

1. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;
(Transaction 2) : ROLLBACK;

La transaction 1 ne tient pas compte des modifications apportées par la transaction 2
→ Affiche 0

2. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Cette-fois, l'update a fait son effet, car le verrou n'est pas mis en place sur toute la transaction → Affiche -3

3. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : INSERT INTO TABLE VALUES (4) ;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Pas de verrou en insertion → Affiche -4

Read committed → Ne tient pas compte des données non validées
Anomalies de répétition possibles

Repeatable read

1. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;
(Transaction 2) : ROLLBACK;

La transaction 1 ne tient pas compte des modifications apportées par la transaction 2
→ Affiche 0

2. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Le verrou est mis en place sur toute la transaction → Affiche 0

3. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : INSERT INTO TABLE VALUES (4) ;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Pas de verrou en insertion → Affiche -4

Repeatable read → Place un verrou pour la durée de la transaction
Lectures "fantômes" toujours possibles

Serializable

1. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;
(Transaction 2) : ROLLBACK;

La transaction 1 ne tient pas compte des modifications apportées par la transaction 2
➔ Affiche 0

2. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : UPDATE TABLE SET COL = COL+1;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

Le verrou est mis en place sur toute la transaction ➔ Affiche 0

3. (Transaction 1) : SELECT SUM(COL) FROM TABLE; => variable INT
(Transaction 1) : Wait 20 seconds
(Transaction 2) : INSERT INTO TABLE VALUES (4) ;
(Transaction 2) : Wait 20 seconds
(Transaction 2) : COMMIT;
(Transaction 1) : SELECT INT-SUM(COL) FROM TABLE;

La transaction veille à la cohérence des données ➔ Affiche 0

Serializable ➔ Pas d'effet de bord entre transactions
Deadlocks possibles

LOCK TABLE vs FOR UPDATE vs ISOLATION LEVEL

- ISOLATION LEVEL : Permet de réfléchir aux besoins d'isolation en termes de transactions, pas par sélection. Deadlocks possibles si SERIALIZABLE (mode SNAPSHOT possible)
- FOR UPDATE : Meilleur contrôle sur les selections bloquantes. Risques d'erreurs.
- LOCK TABLE : (Presque) aucun risque de deadlock. Blocage jusqu'au déblocage explicite. Toute la table verrouillée.