

Transactions

Transactions

Le concept de requête ne permet pas de capturer les opérations atomiques nécessaire à la gestion d'une base de données

Définition

Une transaction est l'exécution d'un programme contenant **plusieurs requêtes** sur la base de données

Du point de vue du SGBD : c'est une **séquence d'opérations** de lecture et d'écriture sur les données

Exemple de transaction

1. Une valeur x est lue depuis la base de données
2. Un programme est exécuté en prenant x pour entrée et retourne y
3. La valeur y est insérée dans la base de donnée

Une opération (sous entendue avec un autre langage ?)

Un ensemble de requêtes

On peut mélanger plusieurs types de requêtes

Les transactions : les actions **cohérentes**

Ici, ajouter une carotte et MAJ le prix total est lié

Somme		Panier		Produit		
client	total	client	produit	produit	nom	prix
0	10	0	0	0	pommes	5
1	5 +3	0	1	1	poires	5
		1	0	2	carottes	3
		1	2			

La transaction pour “Le client 1 ajoute des carottes à son panier” est composée de **deux actions** inséparables :

1. ajouter la ligne correspondante dans Panier
2. ajouter le prix au total dans Somme

Cette transaction permet de vérifier la **contrainte implicite** sur le total

On le fait pour que tout soit cohérent : si on ajoute des produits, le prix total augmente

Principe de cohérence des transactions

Si une transaction s'exécute en l'absence d'autres transactions ou d'erreur de système sur une base de données cohérente, alors la base de données à la fin de l'exécution est aussi cohérente

Remarques

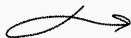
1. la base de données peut être incohérente au cours d'une transaction
2. une gestion des interactions entre transactions est nécessaire
3. une gestion des erreurs de système est nécessaire

Les transactions : les transformations valides

Les transactions définissent les transformations valides de la base de données

Exemple

Compte(num, solde)



On peut les nommer, et dire "On peut ajouter un produit"
→ Ajouter un produit ⇒ Ajouter dans la liste des produits ET
Mettre à Jour le panier

Transfert de 100€ entre le compte 123 et 456, une transaction en **deux requêtes inséparables** :

1. ajouter 100€ au compte 456

```
UPDATE Compte SET solde = solde + 100 WHERE num = 456;
```

2. retirer 100€ au compte 123

```
UPDATE Compte SET solde = solde - 100 WHERE num = 123;
```

Les deux requêtes forment un
objet "Transaction A → B"

Appliquer une seule des deux requêtes :

si j'ajoute de l'argent à Paul, mais n'en enlève pas à Lucas,
on a crée de l'argent

- laisse la base de données cohérente
- **créé ou supprime de l'argent** : ce n'est pas un transfert bancaire valide !

Vérification des contraintes d'intégrité

Par défaut, les contraintes sont vérifiées **après chaque requêtes**, ce qui n'est pas nécessaire pour vérifier le principe de cohérence des transactions

L'impasse mexicaine



On préfère que la vérification se fasse après un GROUPE de requêtes
→ On a transféré l'argent, on vérifie que c'est bien transmis APRÈS

Impasse(tireur, visée)

Une clé étrangère oblige que tous les personnes visées sont des tireurs.

Aucune des requêtes suivantes **ne peut être exécutée**, mais elles forment une transaction cohérente

- `INSERT INTO Impasse VALUES ('bon', 'mauvais');`
- `INSERT INTO Impasse VALUES ('mauvais', 'bon');`
- `INSERT INTO Impasse VALUES ('truand', 'mauvais');`



Les motivations à l'usage des transactions

-
- Un groupe de requêtes, ex : Transaction
- Des liens, des requêtes qui sont logiques entre elles
- définir des opérations, qui conservent la **cohérence des données**, même vis à vis des contraintes implicites
 - définir **des opérations**, qui valident vis à vis du domaine application
 - avoir une gestion de la cohérence plus souple en permettant des états incohérents des données **au cours des transactions**
 - travailler une version fixée des données isolées des autres transactions au cours de longues sessions (intervention humaines)
- Avoir des incohérences qui vont se résoudre APRÈS la transaction

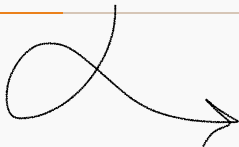
Exercices

Implémenter (du point de vue programmation, et du SGBD) les transactions suivantes en respectant le principe de cohérence

- Le client 0 ajoute des carottes à son panier
- le prix des pommes augmente de 1€
- supprimer les carottes des produits disponibles

Panne

Propriétés ACID



Atomicité, cohérence, isolation, durabilité

Atomicité : tout ou rien

Une transaction n'a que trois états :

- **active**, en cours d'exécution
- **validée** (**committed**), terminée par un succès
- **annulée** (**aborted**), terminée par un échec

On calcule les données

On a le résultat

erreur

L'annulation d'une transaction peut être provoquée par :

- un **plantage** (coupure de courant, disque plein, etc)
- le **système**, si une erreur survient au cours de l'exécution
- l'**utilisateur**, s'il souhaite annulé la transaction

Commandes SQL pour les transactions

SQL permet de définir un groupe de requêtes comme une transaction :

```
-- début de la transaction
-- certains SGBD, demande la commande BEGIN;
requete 1
requete 2
...
COMMIT; -- commande pour demander la validation de la transaction
```

Pour annuler la transaction, il suffit d'utiliser **ROLLBACK** au lieu de **COMMIT**

Autocommit

Avec l'autocommit, chaque requête envoyée est préalablement **entourée par BEGIN et COMMIT**.

Chaque requête est une transaction.



Revient au dernier commit

C'est le cas de Oracle (que l'on utilise en TP)



Atomicité : retour en arrière

Quand une transaction est annulée, il faut revenir en arrière et défaire les modifications réalisées : faire un **rollback**.

En TP on ne peut pas, c'est implicite

```
BEGIN;  
  
UPDATE Compte  
SET solde = solde + 100  
WHERE num = 456;
```

💣 le système plante entre les deux requêtes

```
UPDATE Compte  
SET solde = solde - 100  
WHERE num = 123;  
  
COMMIT;
```

Grâce au commit, en cas de crash on reviendra à l'état initial

1. situation initiale

num	solde
123	100
456	200

2. exécution de la première requête

num	solde
123	100
456	300

État qu'on veut pas en cas de crash

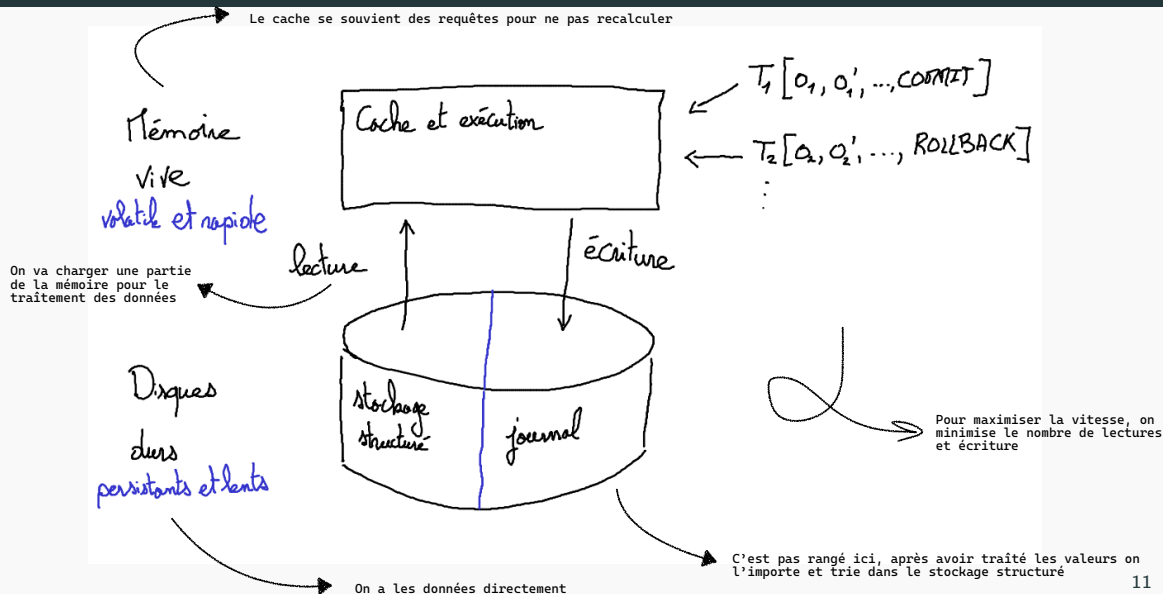
3. plantage, le système est éteint

4. au redémarrage, retour en arrière

num	solde
123	100
456	200

Le commit fait qu'on a toujours une table cohérente

Architecture de la mémoire



Révocabilité des opérations

COMMIT et ROLLBACK sont irrévocables

On ne peut pas revenir en arrière, en cas d'erreur

Opérations d'écriture

- Révocables tant que la transaction ne s'est pas terminée
- Irrévocables une fois la transaction validée (après le commit)

Mécanisme de révocation : rollback

Généralement basée sur le mécanisme de journalisation

- journalisation des données
- journalisation des images avant des données

Stocker les données avant les modifs, et les données après, pour stocker en cas d'erreur

Les effets des transactions validées sont persistants

Si pas d'erreur, on sait que les données seront écrites

- stockés sur le disque dur
- persistant même si une panne arrive 1ms après la validation

Reprise après panne

Elle est aussi garantie par le mécanisme de **journalisation**

- journal est un fichier sur disque
- le SGDB stocke dans le journal toutes les modifications faites par les transactions
- les ordres **COMMIT** sont aussi journalisés


Différer la vérification des contraintes

Vérifier plus tard

On peut différer la vérification des contraintes d'intégrité au **moment de la validation** de la transaction.

En SQL, on peut définir comment chaque contrainte peut être différée :


option	signification
NOT DEFERABLE	ne peut pas être différée (option par défaut)
DEFERABLE INITIALLY IMMEDIATE	différée, si précisé
DEFERABLE INITIALLY DEFERRED	différée par défaut

 **CREATE TABLE** Impasse(
 tireur **VARCHAR**(30),
 vise **VARCHAR**(30),
 -- on ajoute une clef etrangere nommee nom_fk de vise vers tireur
 CONSTRAINT nom_fk **FOREIGN KEY** vise **REFERENCES** Impasse(tireur) <option>;

!/ Il y a 2 R : DEFERRED

Pour résoudre certains problèmes comme :
A dépend de B
B dépend de A

On ne peut pas créer A ni B


 À partir de mtn on vérifiera après les transactions

Différer la vérification des contraintes (2)

On peut préciser les **contraintes que l'on souhaite différer** au début de la transaction :

```
-- différer la vérification de la contrainte nom_fk
SET CONSTRAINT nom_fk DEFERRED;
-- différer toutes les vérifications
SET CONSTRAINTS ALL DEFERRED;
```

On ne veut différer QU'UNE
contrainte (par sécurité)




NOT DEFERABLE

```
INSERT INTO Impasse VALUES ('bon', 'mauvais');
-- vérification -> échec
INSERT INTO Impasse VALUES ('mauvais', 'bon');
-- vérification -> échec
INSERT INTO Impasse VALUES ('truand', 'mauvais');
-- vérification -> échec
COMMIT;
-- pas de vérification et aucune modification
```

DEFERABLE INITIALLY IMMEDIATE

```
SET CONSTRAINT nom_fk DEFERRED;
INSERT INTO Impasse VALUES ('bon', 'mauvais');
-- pas vérification
INSERT INTO Impasse VALUES ('mauvais', 'bon');
-- pas vérification
INSERT INTO Impasse VALUES ('truand', 'mauvais');
-- pas vérification
COMMIT;
-- vérification et validation des 3 insertions
```



À chaque commit on vérifie mtn, ça marche mieux pour certaines erreurs

Exemple de lecture impropre

Transfert de 100€

Personnes imposables

num	solde
123	120
456	50

```
UPDATE Compte SET solde = solde + 100  
WHERE num = 456;
```

num	solde
123	120
456	150

`SELECT num FROM Compte WHERE solde > 100`
les deux comptes sont considérés
comme imposables!!

```
UPDATE Compte SET solde = solde - 100  
WHERE num = 123;  
COMMIT;
```

num	solde
123	20
456	150

Comme on lit pendant l'écriture, on a pendant quelques millisecondes un état où on a les deux imposables

On doit différer donc

Exemple de lecture impropre (2)

Transfert de 100€

Personnes imposables

num	solde
123	80
456	50

```
UPDATE Compte SET solde = solde + 100  
WHERE num = 456;
```

num	solde
123	80
456	150

`SELECT num FROM Compte WHERE solde > 100`
le compte n°456 est considéré
comme imposable


```
UPDATE Compte SET solde = solde - 100  
WHERE num = 123; ❌  
ROLLBACK; -- provoqué par l'erreur
```

Le num n'existe pas, on a une erreur

num	solde
123	80
456	50

Lectures impropres

On lit quand une transaction n'est pas terminée \Rightarrow Données erronées



Un **lecture impropre** (dirty read) est la lecture d'une donnée écrite par une transaction, qui n'est pas encore validée.

Risques

La transaction qui a écrit la donnée peut :

- être annulée
- modifier à nouveau la donnée

Plus de cohérence



Une autre transaction ayant lu cette donnée impropre peut prendre des décisions, qui ne sont pas cohérentes avec un état valide des données.

Exemple de lecture non reproductible

Pas moi :(

Transfert de 100€

Robin des bois

transfert de 100€ de 123 vers 456

num	solde
123	120
456	50

Qui est le plus riche ?
le compte n°123

num	solde
123	20
456	150

Qui est le plus pauvre ?
le compte n°123 **encore !**


redistribution
du plus riche vers le plus pauvre
inutile

les transactions doivent modifier la base comme si elles étaient chacune exécutée seule

En pratique

- une transaction **ne doit pas révéler ses modifications** aux autres transactions tant qu'elle n'a pas été validée
- les transactions exécutées en parallèle peuvent **engendrer des problèmes** de cohérence
- des conflits entre les transactions peuvent entraîner l'**annulation de certaines transactions**
- il n'y pas de **problème d'isolation**, si toutes les transaction lisent uniquement les données

Gestion de la concurrence

- maintenir un degré de concurrence élevé  donc de bonnes performances (faible temps de réponse)
- gestion transparente du point de vue de celui qui envoie la transaction

On ne fait que lire, donc pas de modifications donc pas de problème

Niveau d'isolation d'une transaction

Pour des questions de performance, on peut considérer plusieurs niveau d'isolation pour une transaction :

Niveau d'isolation	lectures impropres	lectures non reproductibles
READ COMMITTED	impossible	possible
SERIALIZABLE	impossible	impossible

En pratique


On peut définir le niveau d'une transaction de la manière suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Dans Oracle, le niveau d'isolation par défaut est READ COMMITTED.

Propriétés ACID

Les transactions doivent vérifier les propriétés **ACID** :

- 
- **Atomicité** chaque transaction s'exécute entièrement ou pas du tout
 - **Cohérence** quelque soit l'exécution des transactions, la base de données reste cohérente
 - **Isolation** les transactions doivent modifier la base comme si chacune était exécutée seule
 - **Durabilité** les modifications d'une transaction sur la base de données ne sont jamais perdues

La prochaine fois ...

- Ordonnancement des opérations
- Sériailisation des transactions