



Concurrence : la grande contradiction

Intégrité des données

- cohérence des transformations
- opérations liées, opérations en parallèle...

efficacité du système d'information

- ne pas bloquer les autres utilisateurs

Id: transaction_cwk_2465 2011-05-11 09:32:05Z fabien

1

Transactions : requêtes cohérentes ensemble

BEGIN début de transaction

COMMIT fin de transaction, validation

ROLLBACK fin de transaction, annulation

si non requêtes implicitement dans un **BEGIN ... COMMIT**

BEGIN ;

INSERT INTO operations(libel,montant,src,dst)

VALUES ('pocket money', 10.0, 'Daddy', 'Calvin');

UPDATE comptes

SET solde = solde+10.0 **WHERE** nom='Calvin';

UPDATE comptes

SET solde = solde-10.0 **WHERE** nom='Daddy';

COMMIT; -- ou annulation avec ROLLBACK

2

Opérations annulables ?

— presques toutes ! **TABLE ROLE FUNCTION LANGUAGE...**

— **sauf** manipulations **DATABASE TABLESPACE...**

BEGIN ;

CREATE USER "hobbes" **WITH PASSWORD** 'calvin';

CREATE TABLE foo(id **SERIAL**, data **TEXT**);

INSERT INTO foo(id,data) **VALUES** ('comics');

COMMIT; -- ou ROLLBACK;

3

Propriété ACID des transactions

Atomicité séquence considéré comme **une seule** opération

elle est complètement effectuée ou non effectuée

Consistance base doit être cohérente (même si annulation)

vérification des contraintes d'intégrité déclarées...

pas nécessairement en cours de route...

Isolation indépendance mutuelles des transactions

gestion de la concurrence (parallélisme)

totale ou partielle, pour améliorer les performances...

Durabilité les mises à jour sont permanentes

sauvegarde sur disque garantie après validation

4

Consistance/isolation : comment ça marche ?

verrouillage des tables par les transactions (automatique)

mode de verrouillage selon les besoins

exclusion réciproque des verrous selon leur niveau

conséquence blocage si verrous incompatibles

assure la cohérence, mais pas la concurrence!

```
-- SELECT * FROM noms;
```

```
LOCK TABLE noms IN ACCESS SHARE MODE;
```

```
-- incompatible avec ACCESS EXCLUSIVE
```

```
-- ALTER TABLE noms ...
```

```
LOCK TABLE noms IN ACCESS EXCLUSIVE MODE;
```

5

Exemple de verrouillage

```
-- 1
```

```
BEGIN;
```

```
SELECT * FROM fruit;
```

```
-- Pomme, Poire
```

```
-- 3
```

```
UPDATE fruit
```

```
SET nom='Cerise'
```

```
WHERE nom='Pomme';
```

```
-- BLOCAGE !
```

```
SELECT * FROM fruit;
```

```
-- DÉBLOCAGE !
```

```
-- pas de modif: nom='Cerise'
```

6

Étreinte fatale... dead lock

blocage par verrouillage croisé de ressources

conséquence fatal à au moins une des transactions !

solutions système ou applicative

- détection automatique toujours nécessaire

- éviter ? *acquisition explicite ordonnée* de verrous ?

7

Exemple Bancaire

```
-- quelques comptes bancaires...
```

```
CREATE TABLE comptes
```

```
(nom TEXT UNIQUE NOT NULL,
```

```
solde DECIMAL(10,2) NOT NULL);
```

```
INSERT INTO comptes VALUES ('Calvin', 100.00);
```

```
INSERT INTO comptes VALUES ('Hobbes', 100.00);
```

```
INSERT INTO comptes VALUES ('Daddy', 1000.00);
```

```
INSERT INTO comptes VALUES ('Mummy', 1000.00);
```

8

Transactions (bancaire et base) simultanées

```

-- Calvin -> Hobbes
-- 1
BEGIN;
UPDATE comptes
SET solde=solde+10.00
WHERE nom='Calvin';
-- 3
UPDATE comptes
SET solde=solde-10.00
WHERE nom='Hobbes';
-- 4
UPDATE comptes
SET solde=solde-1.00
WHERE nom='Hobbes';
-- 5
-- 6
-- ERROR: deadlock detected...
COMMIT;

```

9

Modes de verrouillage de tables : conflits progressifs

noms historiques, row bien au niveau table

```

Access Share accès en lecture simple, SELECT ANALYZE
Row Share SELECT FOR UPDATE
Row Exclusive UPDATE DELETE INSERT
Share Update Exclusive VACUUM
Share CREATE INDEX
Share Row Exclusive pas utilisé
Exclusive pas utilisé
Access Exclusive interdit tout, DROP TABLE

```

11

Liste des verrous : table système pg_locks

relation numéro (oid) de la table
 database numéro (oid) du catalogue
 transaction numéro de la transaction
 pid processus détenteur du verrou
 mode du verrou
 granted si le verrou est accordé

locktype	database	relation	transactionid	virtualtransaction	pid	mode	granted
object	0						
virtualid	125580	126386	27107	27107	24084	AccessShareLock	1
relation	125580	11000	27107	27107	24084	ShareLock	1
relation	125580	126386	27107	27107	24084	AccessShareLock	1
relation	125580	126386	27107	27107	24084	AccessExclusiveLock	1
transactionid			27107		24084	ExclusiveLock	1

13

Isolation : interactions entre transactions

dirty read données non validées visibles

non-repeatable read modifications en cours de transaction

données vues validées mais différentes entre le début et la fin

phantom read résultats d'une requête conditionnelle différents

données ajoutées/modifiées/effacées validées par d'autres

Conséquences : risques d'incohérences...

15

Verrouillage manuel d'une table

```

- contrôle fin du mode de verrouillage
BEGIN;
LOCK TABLE fruit IN ACCESS SHARE MODE;
-- ...
-- déverrouillage à la fin de la transaction
COMMIT;

```

10

Matrice des conflits entre modes

confit	AS	RS	RE	SUE	S	SRE	E	AE
AS								X
RS							X	X
RE					X	X	X	X
SUE					X	X	X	X
S			X	X		X	X	X
SRE			X	X	X		X	X
E		X	X	X	X	X		X
AE	X	X	X	X	X	X	X	

12

Niveaux de verrouillage

BASE Access, SQLite...
 TABLE selon opérations
 PAGE contenant plusieurs tuples
 TUPLE Postgresql

Contrôle du verrouillage

```

- automatique par les opérations INSERT UPDATE DELETE
- manuel pour des tables LOCK...
- manuel pour des tuples SELECT FOR UPDATE

```

14

Salade de fruits

```

CREATE TABLE fruit(id SERIAL, nom TEXT);
INSERT INTO fruit(nom) VALUES('Pomme');
INSERT INTO fruit(nom) VALUES('Poire');

```

Exemple non-repeatable read (update)

```

-- 1
BEGIN;
UPDATE fruit
SET nom='Cerise'
WHERE id=1;
-- 2
BEGIN;
SELECT * FROM fruit;
-- 1 Pomme, 2 Poire
-- 3
SELECT * FROM fruit;
-- 1 Cerise, 2 Poire
COMMIT;

```

16

Exemple phantom read (insert, delete)

```
-- 1
BEGIN;
INSERT INTO fruit(nom)
VALUES ('Figue');

DELETE FROM fruit
WHERE id=2;

-- 3
SELECT * FROM noms;
-- 1 Cerise, 2 Poire

-- 2
BEGIN;
SELECT * FROM noms;
-- 1 Cerise, 2 Poire

-- 4
SELECT * FROM noms;
-- 1 Cerise, 3 Figue

COMMIT;
```

17

Niveaux d'isolation des transactions

plus c'est sûr, plus c'est lent !

Isolation	Dirty/read	Non-repeatable read	Phantom read
Read uncommitted	Oui	Oui	Oui
Read committed	Non	Oui	Oui
Repeatable read	Non	Non	Oui
Serializable	Non	Non	Non

Niveaux disponibles avec PostgreSQL

read committed vision au début de chaque requête
serializable vision au début de la transaction !

18

Risques d'incohérence avec read committed

- dans des cas de mises à jours avec des conditions complexes
- en fait... pas si compliqué que ça !

Exemple : échange d'un attribut

```
Zoo(cage INTEGER PRIMARY KEY,
    animal TEXT NOT NULL);

cage animal
1 lion
2 zebre
3 tigre
```

19

Échange de deux animaux

1. trouver les cages avec **SELECT**...
 2. mettre à jour les animaux avec **UPDATE**
- ```
-- en pseudo-SQL...
BEGIN;
$m = SELECT cage FROM zoo WHERE animal=$x;
$y = SELECT cage FROM zoo WHERE animal=$y;
UPDATE zoo SET animal=$y WHERE cage=$m;
UPDATE zoo SET animal=$x WHERE cage=$m;
COMMIT;
```

**Un animal est-il toujours dans une cage ? NON !**

– modification des données entre **SELECT** et **UPDATE**

20

```
BEGIN; -- lion/zebre
SELECT cage FROM zoo
WHERE animal='lion'; -- 1
SELECT cage FROM zoo
WHERE animal='zebre'; -- 2
UPDATE zoo SET animal='zebre'
WHERE cage=1; -- locked

-- attente vertou...
UPDATE zoo SET animal='lion'
WHERE cage=2;
COMMIT;
```

| cage | animal |
|------|--------|
| 1    | zebre  |
| 2    | lion   |
| 3    | zebre  |

21

**Solutions : verrouillage...**

**SERIALIZABLE** changement du mode de transaction...  
**ROLLBACK** vérification applicative de la modification...  
**FOR UPDATE** données verrouillées dès la consultation !  
**-- en pseudo-SQL...**

```
BEGIN;
$m = SELECT cage FROM zoo WHERE animal=$x FOR UPDATE;
UPDATE zoo SET animal=$y WHERE cage=$m;
UPDATE zoo SET animal=$x WHERE cage=$m;
COMMIT;
```

22

**Besoins de reprises avec serializable**

- abandon en cours de transaction si impossible !
- gestion reprise nécessaire par l'application

```
CREATE TABLE logiciel(nom TEXT);
INSERT INTO logiciel(nom) VALUES('apache');
INSERT INTO logiciel(nom) VALUES('perl');
```

23

**Transactions sérialisées...**

24

```

-- 1
-- 2
BEGIN;
-- 3
UPDATE logiciel
SET nom='ruby'
WHERE nom='perl';
-- 4
COMMIT;
-- 1
BEGIN TRANSACTION ISOLATION
LEVEL SERIALIZABLE;
-- 5
SELECT * FROM logiciel;
-- apache, perl
-- 6
UPDATE logiciel
SET nom='python'
WHERE nom='perl';
-- ERROR: could not serialize
-- access due to
-- concurrent update

```

25

### MVCC : Multi-View Concurrency Control

- objectif** permettre la concurrence entre transactions moins d'interactions entre verrous
- vues** différentes simultanées d'une même table mises à jour, ajouts, effacements...

26

### Vue concurrentes distinctes

```

-- 1
BEGIN;
SELECT * FROM fruit;
-- Pomme, Poire
-- 3
INSERT INTO fruit
VALUES('Banane');
-- 5
SELECT * FROM fruit;
-- Pomme, Poire, Banane
-- 2
BEGIN;
SELECT * FROM fruit;
-- Pomme, Poire
-- 4
INSERT INTO fruit
VALUES('Pêche');
-- 6
SELECT * FROM fruit;
-- Pomme, Poire, Pêche

```

27

### Technique pour MVCC

- xmin, xmax** attributs supplémentaires des tables début et fin de validité d'un tuple...
- tuples non réellement effacés avant **VACUUM!**
- xid** numéro de transaction unique croissant
  - transactions en cours
  - transactions passées toutes validées
  - transactions récentes validées ou annulées stockées dans pg\_xlog?

28

### Exemple MVCC

```

CREATE TABLE legume (nom TEXT);
INSERT INTO legume (nom) VALUES('carotte');
-- 1 - xact 2301
BEGIN;
SELECT xmin, xmax, nom
FROM legume;
-- 298 | 0 | carotte
-- 3
SELECT xmin, xmax, nom
FROM legume;
-- 298 | 2302 | carotte
-- 2 - xact 2302
BEGIN;
DELETE FROM legume
WHERE nom='carotte';
-- 4
ROLLBACK;

```

29

### Durabilité et Atomicité

- écritures** sur disque
  - accès aléatoire lents...
  - coordination des écritures multiples?
- WAL (Write Ahead Log)**
  - fichier séparé qui annonce les modifications des pages
  - séparation sur un disque différent?
  - accès contigus, regroupement de transactions simultanées essentiel pour les performances en charge (vs MySQL)
  - mise à jour différée des données (logwriter)

30

### Réplication de données

- scénario : du crash système à l'incendie...
- sauvegardes journalières? attention
- partage de charge? découpage des données possible?
- disques** miroir, à distance via ethernet...
- base** réplication asynchrone ou synchrone? niveau? comment?
- application** modifications parallèles vs sérialisation

### Double validation two-phase commit 2PC

- nécessaire** aux transactions distribuées (et réplication synchrone?)
- préparation** PREPARE TRANSACTION ...
- nommage de la transaction pour reprise éventuelle
- confirmation** COMMIT PREPARED ou ROLLBACK PREPARED
- attention : état intermédiaire bloquant (verrous)
- BEGIN;**
- INSERT INTO vivement(sid,did,montant,libelle)
- VALUES (12, 32, 100.00, 'retrait distributeur');
- PREPARE TRANSACTION 'vivement 64312';
- ...
- COMMIT PREPARED 'vivement 64312';
- OU BIEN
- ROLLBACK PREPARED 'vivement 64312';

31

32

### Transactions préparées en cours ?

- description `pg_prepared_xacts : num trans, ident, date, catalogue`
- surveillance périodique par l'application
- rapprochement manuel éventuel

| transaction | gpl | prepared | owner | database |
|-------------|-----|----------|-------|----------|
|-------------|-----|----------|-------|----------|

33

### Conclusion sur les transactions

- préservation de la cohérence : **ACID**
- Atomique** WAL, MVCC
- Consistant** verrous, 2PC pour le distribué
- Isolée** niveaux, verrous, MVCC
- Durable** WAL (SSD ?)
- transactions concurrentes complexes : doivent être programmées avec finesse !
- risques d'abandon en cours de route ! reprise...
- pas toujours besoins !
- perte de données non essentielles
- réplications plusieurs mémoires/machines

35

### Point de sauvegarde *savepoint*

- état intermédiaire d'une transaction complexe
- permet des reprises partielles

```

BEGIN;
INSERT INTO ...;
SAVEPOINT etat1;
-- un essai, plout
ROLLBACK TO SAVEPOINT etat1;
-- un autre essai
SAVEPOINT etat2;
...
COMMIT;

```

34

### List of Slides

- 1 Concurrency : la grande contradiction
- 2 Transactions : requêtes cohérentes ensemble
- 3 Opérations annulables ?
- 4 Propriétés ACID des transactions
- 5 Consistance/isolation : comment ça marche ?
- 6 Exemple de verrouillage
- 7 Etreinte fatale... *dead/lock*
- 8 Exemple Bancaire
- 9 Transactions (bancaire et base) simultanées
- 10 Verrouillage manuel d'une table
- 11 Modes de verrouillage de tables : conflits progressifs

12 Matrice des conflits entre modes

13 Liste des verrous : table `system.pg_locks`

14 Niveaux de verrouillage

14 Contrôle du verrouillage

15 Isolation : interactions entre transactions

15 Conséquences : risques d'incohérences...

16 Salade de fruits

16 Exemple *non-repeatable read* (update)17 Exemple *phantom read* (insert, delete)

18 Niveaux d'isolation des transactions

18 Niveaux disponibles avec PostgreSQL

19 Risques d'incohérence avec *read committed*

19 Exemple : échange d'un attribut

20 Echange de deux animaux

20 Un animal est-il toujours dans une cage ? NON !

22 Solutions : verrouillage...

23 Besoins de reprises avec *serializable*

24 Transactions sérialisées...

26 MVCC : Multi-View Concurrency Control

27 Vue concurrentes distinctes

28 Technique pour MVCC

29 Exemple MVCC

30 Durabilité et Atomicité

31 Réplication de données

32 Double validation *two-phase commit* 2PC

33 Transactions préparées en cours ?