Transactions

Fabien Coelho, Claire Medrala

Mines Paris – PSL, MESR

Janvier 2025

Concurrence des accès

Contradiction

intégrité des données
- cohérence des modifications, lecture et écriture...
- opérations liées vs en parallèle vs en conflit...

efficacité du système d'information
- ne pas bloquer les autres utilisateurs !

Transactions

commandes `BEGIN COMMIT PREPARE ROLLBACK`

propriétés ACID

implémentation locks MVCC WAL...

Transactions : requêtes cohérentes ensemble

Commandes SQL

`BEGIN` début de transaction

`COMMIT` fin de transaction, **validation**

`ROLLBACK` fin de transaction, **annulation**

sinon 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
```


Transactions

Opérations annulables ?

- presques toutes ! `TABLE ROLE FUNCTION...`
- **sauf** manipulations `DATABASE TABLESPACE...`
- **sauf** compteurs de séquences `NEXTVAL`

```
BEGIN;
CREATE USER "hobbes" WITH PASSWORD 'c@lvin';
CREATE TABLE foo(id SERIAL, data TEXT);
INSERT INTO foo(id,data) VALUES('comics');
COMMIT; -- ou ROLLBACK;
```

Vérification des contraintes décalée

- décalable : `DEFERRABLE` vs `NOT DEFERRABLE`
- si décalable : `INITIALY IMMEDIATE` vs `INITIALY DEFERRED`

```
CREATE TABLE Auteur
  (aid INTEGER PRIMARY KEY, nom TEXT);
CREATE TABLE Poeme
 (aid INTEGER NOT NULL REFERENCES Auteur
    DEFERRABLE INITIALLY DEFERRED,
  titre TEXT);
BEGIN;
INSERT INTO Poeme VALUES(1, 'La prose du Transibérien...');
-- attend le commit pour vérifier l'auteur
INSERT INTO Auteur VALUES(1, 'Blaise Cendrar');
COMMIT;
```


Propriété ACID des transactions

Atomicité séquence considéré comme **une seule** opération
- elle est complètement effectuée ou non effectuée

Consistance base doit être cohérence (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

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;
```


Exemple de verrouillage


```
-- 1
BEGIN;
SELECT * FROM fruit;
-- Pomme, Poire

-- 3
UPDATE fruit
SET nom='Cerise'
WHERE nom='Pomme';

-- 5
SELECT * FROM fruit;
-- Poire, Cerise
COMMIT;
```

```
-- 2
BEGIN;
SELECT * FROM fruit;
-- Pomme, Poire

-- 4
UPDATE fruit
SET nom='Figue'
WHERE nom='Pomme';
-- BLOCAGE !

-- 6
-- DÉBLOCAGE !
-- pas de modif: nom='Cerise'
```

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 ?*

Exemple Bancaire


```sql
-- quelques comptes bancaires...
CREATE TABLE comptes
  (nom TEXT UNIQUE NOT NULL,
   solde DECIMAL(10,2) NOT NULL);

INSERT INTO comptes VALUES
  ('Calvin', 100.00),
  ('Hobbes', 100.00),
  ('Daddy', 1000.00),
  ('Mummy', 1000.00);
```


Transactions (bancaire et base) simultannées


```sql
-- 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';

-- 6
COMMIT;
```

```sql
-- Hobbes -> Calvin
-- 2
BEGIN;
UPDATE comptes
SET solde=solde+1.00
WHERE nom='Hobbes';

-- 4
UPDATE comptes
SET solde=solde-1.00
WHERE nom='Calvin';

-- 5
-- ERROR: deadlock detected...
```


Verrouillage manuel d'une table

- contrôle fin du mode de verrouillage
- verrous applicatifs *advisory locks*
  usage systématique pour être efficace. . .

```sql
BEGIN;
LOCK TABLE fruit IN ACCESS SHARE MODE;

-- ...

-- déverrouillage à la fin de la transaction
COMMIT;
```

Modes de verrouillage de tables : conflits progressifs

Access Share	accès en lecture simple	`SELECT ANALYZE`
Row Share		`SELECT FOR UPDATE`
Row Exclusive		`UPDATE DELETE INSERT`
Share Update Exclusive		`VACUUM ANALYSE...`
Share		`CREATE INDEX`
Share Row Exclusive		`ALTER TABLE, CREATE TRIGGER`
Exclusive		`REFRESH MATERIALIZED VIEW CONCURRENTLY`
Access Exclusive	interdit tout	`DROP REINDEX...`

Aussi : verrous de niveau ligne *row-level locks*

Matrice des conflits entre modes

conflit	AS	RS	RE	SUE	S	SRE	E	AE
AS								x
RS							x	x
RE					x	x	x	x
SUE				x	x	x	x	x
S			x	x		x	x	x
SRE			x	x	x	x	x	x
E		x	x	x	x	x	x	x
AE	x	x	x	x	x	x	x	x

Liste des verrous : table système `pg_locks`

relation	num table	pid	processus tenant le verrou
database	num catalogue	mode	du verrou
transaction	num transaction	granted	si verrou accordé

locktype	*db*	*rel*	*tid*	*vtx*	*pid*	*mode*	*granted*
relation	406768	12073		19/317	755776	AccessShareLock	t
virtualxid				19/317	755776	ExclusiveLock	t
object	0			19/317	755776	AccessShareLock	t
object	406768			19/317	755776	AccessShareLock	t
relation	406768	407583		19/317	755776	AccessExclusiveLock	t
relation	406768	407582		19/317	755776	ShareLock	t
relation	406768	407582		19/317	755776	AccessExclusiveLock	t
relation	406768	407579		19/317	755776	AccessExclusiveLock	t
transactionid			101001	19/317	755776	ExclusiveLock	t

Verrouillage

Niveaux

BASE	Access, SQLite...	PAGE	contenant de tuples
TABLE	selon opérations	TUPLE	Postgres

Contrôle

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

Isolation : interactions entre transactions

dirty read données non validées visibles

non-repeatable read modifications en cours `UPDATE`
données vues validées mais différentes entre le début et la fin

phantom read résultats différents `INSERT` `DELETE`
données ajoutées/modifiées/effacées validées par d'autres

Conséquences : risques d'incohérences. . .

Salade de fruits


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

Exemple *non-repeatable read* (update)

```
-- 1
BEGIN;
UPDATE fruit
SET nom='Cerise'
WHERE id=1;

-- 3
COMMIT;
```

```
-- 2
BEGIN;
SELECT * FROM fruit;
-- 1 Pomme, 2 Poire

-- 4
SELECT * FROM fruit;
-- 1 Cerise, 2 Poire
```


Exemple *phantom read* (insert, delete)


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

DELETE FROM fruit
WHERE id=2;

-- 3
COMMIT;
```

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

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


Exemple *dirty read*


```
-- 2
BEGIN;
DELETE FROM fruit
  WHERE id = 1;
UPDATE fruit
  SET nom = 'Ananas'
  WHERE id = 2;
INSERT INTO fruit(id, nom)
  VALUES (7, 'Cerise');
SELECT * FROM fruit;

-- 4
COMMIT;
```

```
-- 1
CREATE EXTENSION pg_dirtyread;
BEGIN;
SELECT * FROM fruit;

-- 3
SELECT * FROM fruit;
SELECT *
  FROM pg_dirtyread('fruit')
  AS t(xmin XID, xmax XID,
       id INT, name TEXT);

-- 5 bis repetita...
COMMIT;
```

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 uncommitted non, ou via extension **pg dirtyread**

read committed vision au début de chaque **requête**

serializable vision au début de la **transaction** !

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

Échange de deux animaux

1. trouver les cages avec SELECT...
2. mettre à jour les animaux avec UPDATE

```
-- KO avec variables "psql"
\set x 'zebre'
\set y 'lion'
BEGIN;
SELECT cage AS n FROM zoo WHERE animal=:'x' \gset
SELECT cage AS m FROM zoo WHERE animal=:'y' \gset
UPDATE zoo SET animal=:'y' WHERE cage=:n;
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

Échange (suite)

```
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 verrou...
UPDATE zoo SET animal='lion'
 WHERE cage=2;
COMMIT;
```

cage	*animal*
1	zebre
2	lion
3	zebre

```
BEGIN; -- zebre/tigre
SELECT cage FROM zoo
 WHERE animal='zebre'; -- 2
SELECT cage FROM zoo
 WHERE animal='tigre'; -- 3
UPDATE zoo SET animal='tigre'
 WHERE cage=2; -- locked
UPDATE zoo SET animal='zebre'
 WHERE cage=3;
COMMIT;
```

cage	*animal*
1	lion
2	tigre
3	zebre

Solutions : verrouillage...

`SERIALIZABLE` changement du mode de transaction...

`ROLLBACK` vérification applicative de la modification...

`FOR UPDATE` données verrouillées dès la consultation !

```
-- OK avec variables "psql"
\set x 'zebre'
\set y 'lion'
BEGIN;
SELECT cage AS n FROM zoo WHERE animal=:'x' FOR UPDATE \gset
SELECT cage AS m FROM zoo WHERE animal=:'y' FOR UPDATE \gset
UPDATE zoo SET animal=:'y' WHERE cage=:n;
UPDATE zoo SET animal=:'x' WHERE cage=:m;
COMMIT;
```


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'), ('perl');
```


Transactions sérialisées...


```
-- 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
```


MVCC : Multi-View Concurrency Control

objectif permettre la concurrence entre transactions
moins d'interactions entre verrous

vues différentes simultannées d'une même table
mises à jour, ajouts, effacements...

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('Peche');

-- 6
SELECT * FROM fruit;
-- Pomme, Poire, Peche
```


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 : `txid_current()`
- transactions passées toutes validées
- transactions récentes validées ou annulées stockées dans le répertoire `pg_xlog` ?

Exemple MVCC


```
CREATE TABLE legume(nom TEXT);
INSERT INTO legume(nom) VALUES('carotte');

-- 1 - xact 2301
BEGIN;
SELECT xmin, xmax, nom
FROM legume;
-- 2298 |    0 | carotte

-- 3
SELECT xmin, xmax, nom
FROM legume;
-- 2298 | 2302 | carotte
```

```
-- 2 - xact 2302
BEGIN;
DELETE FROM legume
WHERE nom='carotte';

-- 4
ROLLBACK;
```


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 simultannées
  essentiel pour les performances en charge (vs MySQL)
- mise à jour différée des données *(bgwriter)*

Réplication de données

- scénario : du crash système à l'incendie...
- sauvegardes journalières ? attention !
- partage de charge ? découpage des données possible ?

disques mirroir, à distance via ethernet...
base réplication asynchrone, synchrone ?
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 'virement 64312';
-- ...
COMMIT PREPARED 'virement 64312';
-- OU BIEN
ROLLBACK PREPARED 'virement 64312';
```


Transactions préparées en cours ?

- description `pg_prepared_xacts` :
  numéro de transaction, identifiant, date, catalogue
- surveillance périodique par l'application
- rapprochement manuel éventuel

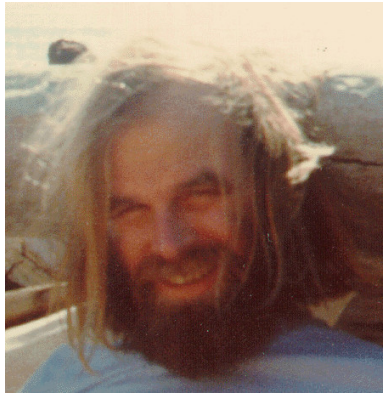
Point de sauvegarde *savepoint*

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

```
BEGIN;
INSERT INTO ...;
SAVEPOINT etat1;
  -- un essai, ERROR!
ROLLBACK TO SAVEPOINT etat1;
  -- un second essai, ok!
SAVEPOINT etat2;
...
COMMIT;
```

James (Jim) Gray 1944-2007 (2012)

- Berkeley, IBM (System R)
  Tandem, DEC, MS
- recherches DB
  - verrous
  - ACID
  - 2PC
  - CUBE
  - gestion de caches
- Turing Award 1998



Conclusion sur les transactions

- préservation de la cohérence : **ACID**
  - Atomique WAL, MVCC
  - Consistante verrous, 2PC pour le distribué
  - Isolée niveaux, verrous, MVCC
  - Durable WAL
- 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