Tansactions, concurrence et MVCC dans PostgreSQL

BDAV – Bases de Données AVancées

M1 MIAGE UNC 2023-2024

1 Introduction

Notes de cours du lundi 7 février 2022.

1.1 Objectifs

- présenter les notions ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*)
- connnaître et pratiquer (en TP) les niveaux d'isolations en SQL qui garantissent des propriétés de plus en plus fortes
  - `Read uncommitted`, `Read committed`, `Repeatable read` et `Serializable`
- traverser de la vue **logique** (le SQL) à l'implantation **physique** (comment PostgreSQL le fait)
  - le modèle MVCC (MultiVersion Concurrency Control) dans le stockage physique de PostgreSQL

1.2 Mise en place

Pour le TP, créer une base avec un utilisateur propriétaire dédié et activer les accès via le réseau TCP, voir dans le sujet. Extensions à installer avec l'utilisateur `postgres`, dans la même base que le TP :

```
CREATE EXTENSION pgstattuple WITH SCHEMA public;
CREATE EXTENSION pageinspect WITH SCHEMA public;
```

1.3 Compléments

Quelques vidéos (de qualité assez médiocre) réalisées en 2021 sur ce sujet :

- *le MVCC dans PostgreSQL* https://www.youtube.com/watch?v=BXhFoWqxiF8&t=577s
- *différence entre READ COMMITTED et REPEATABLE READ* https://www.youtube.com/watch?v=7oPfT6ewrqw
- *différence entre REPEATABLE READ et SERIALIZABLE* https://www.youtube.com/watch?v=YCWAuF0afPE

Au surplus, l'excellente vidéo *PG Day France 2019 : Sécurisez vos transactions concurrentes* par Daniel Vérité, https://www.youtube.com/watch?v=phaS8obzcvo (ce cours et le TP en sont largement inspirés).

Voir aussi :

- les chapitres 5.2.4 et 6.7 de ~~*Not*~~ *Only SQL* https://framaclic.org/h/9r_3GLnsm_q
- les sections 2.2 et 6.6 du support de la formation *SQL pour PostgreSQL* https://dali.bo/devsqlpg_pdf;

2 La gestion transactionnelle

La gestion transactionnelle et les niveaux d'isolation sont des **fonctionnalités clefs de la concurrence** d'un moteur de SGBD(-R).

2.1 Les propriétés ACID

Les propriétés **ACID** sont les suivantes :

- *Atomicity* : une transaction est atomique, entière : tout échoue ou tout réussi ;
- *Consistency* : une transaction amène le système d'un état cohérent (c'est-à-dire qui respecte toutes les contraintes d'intégrité) à un autre (on ne doit pas enregistrer un état incohérent) ;
- *Isolation* : les transactions n'agissent pas les unes sur les autres (sauf une fois définitivement validée) ;
- *Durability* : une transaction validée provoque des changements permanents, persistants en base.

Ces propriétés sont au coeur des moteurs des SGBD-R. En revanche, dans les systèmes NoSQL, on utilise l'acronyme *BASE* pour :

- *Basic Availability* : le service est rendu, peut-être partiellement
- *Soft State* : la cohérence est le problème du programmeur, pas du SGBD
- *Eventual Consistency* : à un moment, les données finirons par être cohérentes (les modifications seront visibles de partout)

Ce qui est assez fondamentalement opposé à *ACID* !

3 Les niveaux d'isolation et les transaction

Une transaction est ensemble *atomique* d'opérations : `SELECT`, `INSERT` , `UPDATE` et `DELETE` mais aussi (pour PostgreSQL) d'opérations DDL comme `CREATE TABLE` ou `CREATE INDEX` (en revanche c'est *faux* pour Oracle). Les principales commandes sont :

- `BEGIN [TRANSACTION]` pour ouvrir une transaction
- Soit :
  - `COMMIT` pour valider, le succès
  - `ROLLBACK` pour annuler, l'échec. Soit sous le contrôle du client, soit celui du serveur en cas de défaillance (e.g., perte réseau)
- `SAVEPOINT` pour sauvegarder l'état d'une transactions à un point (pour une reprise partielle)

Chaque transaction (et donc session) est isolée à un certain point :

- elle ne voit pas les opérations des autres avant `COMMIT`

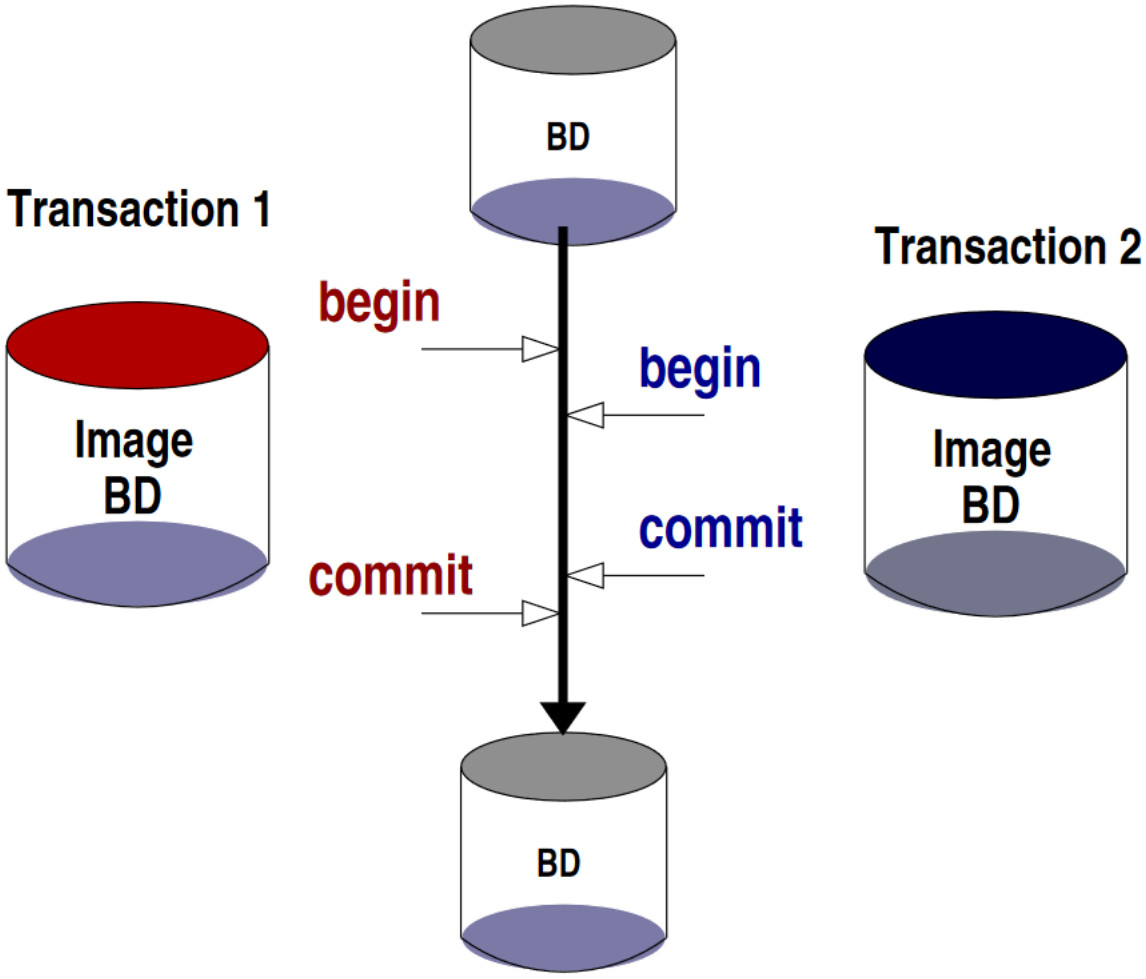

Figure 1: Principe générale de l'isolation des données, extrait de ~~*Not*~~ *Only SQL*

- elle s'exécute indépendamment des autres
- il y a une transaction implicite quand on envoie

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé est donc un *compromis* entre :

- le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) ;
- les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

3.1 Illustration

Table d'exemple :

```
CREATE TABLE list (x int PRIMARY KEY);
INSERT INTO list (SELECT i FROM generate_series(1,5) AS g(i));
```

On va exécuter en parallèle et **pas à pas** les deux transactions suivantes :

```
BEGIN ISOLATION LEVEL READ COMMITTED;
    SELECT * FROM list ;
    INSERT INTO list VALUES (42);
    SELECT * FROM list ;
COMMIT;
SELECT * FROM list ;

BEGIN ISOLATION LEVEL READ COMMITTED;
    SELECT * FROM list ;
    INSERT INTO list VALUES (43);
    SELECT * FROM list ;
COMMIT;
SELECT * FROM list ;
```

On remarque que chaque transacation ne voit que ses propres modifications.

On reprend ensuite avec la même valeur insérée dans chaque transaction : on voit que la seconde transaction est bloquée jusqu'à ce que la première soit résolue. Des mécanismes de vérrous à la granularité variable, qu'on ne détaillera pas, permettent de contrôler les accès concurrents. On peut consulter la liste des vérrous en cours avec la requête suivante :

```
SELECT locktype, relname, pid, mode
FROM pg_locks l
     JOIN pg_class t ON l.relation = t.oid
WHERE t.relkind = 'r'
      AND t.relname = 'list';
```

3.2 Les différents niveaux d'isolation

Le niveau demandé se spécifie soit à la création de la transaction, pour toute la session ou par défaut globalement. Les niveaux SQL sont, dans l'ordre d'importance des garanties :

- `READ UNCOMMITTED`
  - n'existe pas en PostgreSQL
  - à ce niveau, on peut voir des modifications concurrentes non validés
- `READ COMMITTED`
  - **par défaut en PostgreSQL**

  - garanti l'absence des anomalies du niveau inférieur, mais dans une même transaction on peut lire des valeurs différentes sur la même donnée (si une transaction concurrente valide entre les deux lecture)
- `REPEATABLE READ`
  - pas de lecture fantomes
  - garanti l'absence des anomalies du niveau inférieur, toutes les lectures successives doivent donner le même résultat
- `SERIALIZABLE` :
  - garanti que le résultat de **toute** exécution concurrente est celui **d'une** exécution où les transactions seraient séquentielles

Si le SGBD n'arrive pas à garantir les propriétés du niveau demandé, alors les transactions à problèmes sont annulées, ce qui provoque une erreur/exception chez le client. Voir :

- https://www.postgresql.org/docs/current/tutorial-transactions.html le tutoriel sur les transactions
- https://www.postgresql.org/docs/current/sql-begin.html et https://www.postgresql.org/docs/current/sql-end.html les syntaxes du `BEGIN` et du `END`
- https://www.postgresql.org/docs/current/transaction-iso.html : la définition standard des niveaux d'isolation
- https://www.postgresql.org/docs/current/sql-set-transaction.html : définir le niveau d'isolation demandé dans une transaction en cours

3.3 Exemple `READ COMMITTED` versus `REPEATABLE READ`

On va exécuter en parallèle les deux transactions suivantes :

```
BEGIN ISOLATION LEVEL READ COMMITTED;
    UPDATE list SET x = x - 1;
COMMIT;

BEGIN ISOLATION LEVEL READ COMMITTED;
    DELETE FROM list
    WHERE x = (SELECT max(x) FROM list);
COMMIT;
```

On arrive à `DELETE 0` : quand bien même une valeur maximum existe toujours dans une table non vide, rien n'est supprimé. En effet, entre la lecture de la sous-requête `SELECT max(x) FROM list` (qui ne connait pas encore l'effet du `UPDATE`) et l'exécution (bloquante) du `DELETE`, la valeur lue *a disparue*.

On augmente le niveau d'isolation pour passer à `REPEATABLE READ` et on répète la même expérience.

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
    UPDATE list SET x = x - 1;
COMMIT;

BEGIN ISOLATION LEVEL REPEATABLE READ;
    DELETE FROM list
    WHERE x = (SELECT max(x) FROM list);
COMMIT;
```

La transaction qui supprime est maintenant *refusée* avec le message `ERROR: 40001: could not serialize access due to concurrent update` car la lecture n'est pas répétable : le maximum a changé entre le temps de la lecture et celui de la suppression.

3.4 Exemple `REPEATABLE READ` versus `SERIALIZABLE`

On fait un exemple similaire mais avec le niveau `SERIALIZABLE`, plus strict encore que `REPEATABLE READ`.

- https://wiki.postgresql.org/wiki/SSI
- https://wiki.postgresql.org/wiki/Serializable

```
CREATE TABLE dots(id int PRIMARY KEY, color text NOT NULL);
INSERT INTO dots
    SELECT id, CASE WHEN id % 2 = 1 THEN 'black' ELSE 'white' END
    FROM generate_series(1,10) AS g(id);
```

```
-- transaction tx1
BEGIN ISOLATION LEVEL REPEATABLE READ;
    SELECT * FROM dots ORDER BY id;

    -- le BLANC passe en NOIR
    UPDATE dots SET color = 'black'
    WHERE color = 'white';

    SELECT * FROM dots ORDER BY id;
COMMIT;
```

```
-- transaction tx2
BEGIN ISOLATION LEVEL REPEATABLE READ;
    SELECT * FROM dots ORDER BY id;

    -- le NOIR passe en BLANC
    UPDATE dots SET color = 'white'
    WHERE color = 'black';

    SELECT * FROM dots ORDER BY id;
COMMIT;
```

On obtient un état final *où* ***tx1*** *et* ***tx2*** *ont été exécutée simultanément* qu'on ne peut **pas** obtenir en exécutant soit `tx1` puis `tx2` soit `tx2` puis `tx1`. En augmentant le niveau d'isolation à `SERIALIZABLE` l'erreur suivante est levée `ERROR: 40001: could not serialize access due to read/write dependencies among transactions. DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt.`

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
    SELECT * FROM dots ORDER BY id;

    UPDATE dots SET color = 'black'
    WHERE color = 'white';

    SELECT * FROM dots ORDER BY id;
COMMIT;
```

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
    SELECT * FROM dots ORDER BY id;

    UPDATE dots SET color = 'white'
```

```
    WHERE color = 'black';

    SELECT * FROM dots ORDER BY id;
COMMIT;
```

Le problème est détecté car on ne peut pas entrelacer ces transactions et avoir un résultat qui soit celui d'une exécution séquentielle. Il faut donc reéxecuter la transaction en erreur et ainsi *choisir* une sérialisation (`tx1` puis `tx2` si `tx2` échoue).

4 L'implantation physique : le modèle MVCC

On va regarder comment fait PostgreSQL pour gérer ces versions multiples avec des fenêtres (xmin, xmax).

- https://www.postgresql.org/docs/current/mvcc-intro.html : le modèle MVCC
- https://www.postgresql.org/docs/current/functions-info.html : fonction système, dont `pg_current_xact_id()` et `pg_current_xact_id_if_assigned()`
- https://www.postgresql.org/docs/current/storage-page-layout.html : la représentation physique en pages de 8192 octets
- https://www.postgresql.org/docs/current/pageinspect.html : contenu des tables physiques (`postgres` seulement)
- https://www.postgresql.org/docs/current/pgstattuple.html : informations détaillée sur les tables

Attention les valeurs, en particulier les identifiants des transactions, seront *différentes* lors d'une autre exécution.

4.1 La représentation physique des données

Attention pour utiliser les extensions `pgstattuple` et `pageinspect` il faut disposer de privilèges élevés (`pg_stat_scan_tables` et `SUPERUSER` respectivement).

```
-- une table d'exemple tirée au hasard
CREATE TABLE trans(id integer PRIMARY KEY, b text);
INSERT INTO trans(SELECT i, chr((floor(random()*26)+65)::integer) FROM generate_series(1,1E1) AS g(i));

-- le contenu de la table vue depuis la page brute
SELECT lp, lp_off, lp_len, t_xmin, t_xmax, t_ctid, t_data FROM heap_page_items(get_raw_page('trans', 0))
--  lp | lp_off | lp_len | t_xmin | t_xmax | t_ctid |     t_data
-- ----+--------+--------+--------+--------+--------+----------------
--   1 |   8160 |     30 |   1876 |      0 | (0,1)  | \x010000000542
--   2 |   8128 |     30 |   1876 |      0 | (0,2)  | \x02000000054a
--   3 |   8096 |     30 |   1876 |      0 | (0,3)  | \x030000000554
--   4 |   8064 |     30 |   1876 |      0 | (0,4)  | \x040000000556
--   5 |   8032 |     30 |   1876 |      0 | (0,5)  | \x050000000543
--   6 |   8000 |     30 |   1876 |      0 | (0,6)  | \x060000000554
--   7 |   7968 |     30 |   1876 |      0 | (0,7)  | \x070000000544
--   8 |   7936 |     30 |   1876 |      0 | (0,8)  | \x080000000558
--   9 |   7904 |     30 |   1876 |      0 | (0,9)  | \x090000000549
--  10 |   7872 |     30 |   1876 |      0 | (0,10) | \x0a0000000546


-- depuis la ligne de commande
-- psql -d bdav -tA -c "select encode(get_raw_page::bytea, 'hex') from get_raw_page('trans',0)" | xxd -
```

L'explication de la taille $30 = 24 + 4 + 1 + 1$

```
SELECT pg_column_size(row());                          -- 24 : la taille incompressible des headers
SELECT pg_column_size(row(0::integer));                -- 28 : int32
SELECT pg_column_size(row(0::integer, 'Z'::text));     -- 30 : 1o taille/metadata (varlena) + 1o de char
SELECT pg_column_size(row(0::integer, 'ZZ'::text));    -- 31 : +1o de char
```

4.2 Le maintien des xmin et xmax

- https://www.postgresql.org/docs/current/ddl-system-columns.html : les colonnes implicitement présentes dans toutes les tables, comme `ctid` (identifiant du tuple au sein de la page), `xmin` (plus petit id de transcation qui peut voir le tuple) et `xmax` (plus grand id de transcation qui peut voir le tuple, 0 si le tuple n'est pas supprimé et qu'il est visible de toutes).
- https://postgrespro.com/blog/pgsql/5967892 avec des exemple d'utilisation de `pageinspect`.

Savoir dans quelle transaction on est :

```
-- pas de transaction en cours
SELECT pg_current_xact_id_if_assigned();

-- on peut aussi utiliser pg_current_xact_id() mais qui a
-- le défaut de d'incrémenter le txid à chaque appel
-- SELECT pg_current_xact_id();

-- un bloc de transaction
BEGIN;
-- pas de numéro de transaction affecté, car on a encore rien fait
SELECT pg_current_xact_id_if_assigned();
DELETE FROM trans;
-- maintenant on en a un
SELECT pg_current_xact_id_if_assigned();
INSERT INTO trans VALUES(0, 'Z');
-- pas d'incrémentation du xact_id
SELECT pg_current_xact_id_if_assigned();
SELECT pg_current_xact_id();
COMMIT;
```

On affiche les colonnes implicites pour voir le snapshot actuellement accessible par la transaction en cours.

```
SELECT ctid, xmin, cmin, xmax, cmax, t.* FROM trans t;

-- ctid | xmin | cmin | xmax | cmax | id | b
-- -------+------+------+------+------+----+---
-- (0,1) | 1704 | 0 | 0 | 0 | 1 | V
-- (0,2) | 1704 | 0 | 0 | 0 | 2 | J
-- (0,3) | 1704 | 0 | 0 | 0 | 3 | A
-- (0,4) | 1704 | 0 | 0 | 0 | 4 | B
-- (0,5) | 1704 | 0 | 0 | 0 | 5 | I
-- (0,6) | 1704 | 0 | 0 | 0 | 6 | T
-- (0,7) | 1704 | 0 | 0 | 0 | 7 | X
-- (0,8) | 1704 | 0 | 0 | 0 | 8 | K
-- (0,9) | 1704 | 0 | 0 | 0 | 9 | W
-- (0,10) | 1704 | 0 | 0 | 0 | 10 | H

UPDATE trans SET b = lower(b) WHERE id > 5;
```

```
SELECT ctid, xmin, cmin, xmax, cmax, t.\* FROM trans t;
-- ctid | xmin | cmin | xmax | cmax | id | b
-- -------+------+------+------+------+----+---
-- (0,1) | 1704 | 0 | 0 | 0 | 1 | V
-- (0,2) | 1704 | 0 | 0 | 0 | 2 | J
-- (0,3) | 1704 | 0 | 0 | 0 | 3 | A
-- (0,4) | 1704 | 0 | 0 | 0 | 4 | B
-- (0,5) | 1704 | 0 | 0 | 0 | 5 | I
-- (0,11) | 1709 | 0 | 0 | 0 | 6 | t
-- (0,12) | 1709 | 0 | 0 | 0 | 7 | x
-- (0,13) | 1709 | 0 | 0 | 0 | 8 | k
-- (0,14) | 1709 | 0 | 0 | 0 | 9 | w
-- (0,15) | 1709 | 0 | 0 | 0 | 10 | h
```

Avec l'extension `pageinspect` on voit qu'il n'y a pas de suppression mais des ajouts "sur le tas" des tuples modifiés.

```
SELECT lp, lp_off, lp_len, t_xmin, t_xmax, t_ctid, t_data FROM heap_page_items(get_raw_page('trans', 0))
-- lp | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_data
-- ----+--------+--------+--------+--------+--------+----------------
-- 1 | 8160 | 30 | 1704 | 0 | (0,1) | \x010000000556
-- 2 | 8128 | 30 | 1704 | 0 | (0,2) | \x02000000054a
-- 3 | 8096 | 30 | 1704 | 0 | (0,3) | \x030000000541
-- 4 | 8064 | 30 | 1704 | 0 | (0,4) | \x040000000542
-- 5 | 8032 | 30 | 1704 | 0 | (0,5) | \x050000000549
-- 6 | 8000 | 30 | 1704 | 1709 | (0,11) | \x060000000554
-- 7 | 7968 | 30 | 1704 | 1709 | (0,12) | \x070000000558
-- 8 | 7936 | 30 | 1704 | 1709 | (0,13) | \x08000000054b
-- 9 | 7904 | 30 | 1704 | 1709 | (0,14) | \x090000000557
-- 10 | 7872 | 30 | 1704 | 1709 | (0,15) | \x0a0000000548
-- 11 | 7840 | 30 | 1709 | 0 | (0,11) | \x060000000574
-- 12 | 7808 | 30 | 1709 | 0 | (0,12) | \x070000000578
-- 13 | 7776 | 30 | 1709 | 0 | (0,13) | \x08000000056b
-- 14 | 7744 | 30 | 1709 | 0 | (0,14) | \x090000000577
-- 15 | 7712 | 30 | 1709 | 0 | (0,15) | \x0a0000000568
```

On peut voir chaque couple `(xmin, xmax)` comme une fenêtre de visibilité du tuple. Chaque transaction a un *instant associé*, son `xact_id`, qui définit ainsi **une tranche** de visibilité comme dans le schéma ci dessous :

```
      i0  i1  i2  i3  i4  i5  i6  i7  i8  i9
=============================================
t1 : [XXXXXXXX[----------------------------
t2 : [XXXXXXXXXXXXXXXX[--------------------
t3 : --------[XXXXXXXXXXXXXXXX[-------------
t4 : [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX>
t5 : -------------[XXXXXXXXXXXXXXXXXXXXXXX>
```

La transaction d'identifiant `i3` voir les tuples `t2`, `t3`, `t4` et `t5` qu'elle vient de créer. En revanche `i3`, elle ne voit plus `t1` qui a été supprimé en `i2`.