Transactions, concurrence et MVCC dans PostgreSQL

BDAV-2 – Bases de Données AVancées 2

THION Romuald

Licence Informatique UNC 2022

1 Introduction

1.1 Objectifs du cours

— présenter les notions ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*)
— connnaître et pratiquer (en TP) les niveaux d'isolations :
  — `Read committed`, `Repeatable read` et `Serializable`
— traverser de la vue **logique** (le SQL) à l'implantation **physique** (comment PostgreSQL implémente)
  — le modèle MVCC (*MultiVersion Concurrency Control*) dans le stockage physique de PostgreSQL
— préparer au TP transactions et niveaux d'isolation

1.2 Mise en place

Pour les exemples et pour le TP, créer une base avec un utilisateur propriétaire dédié et activer les accès via le réseau TCP comme indiqué dans le sujet de TP. Extensions supplémentaires à installer avec l'utilisateur `postgres` pour la fin du cour :

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

1.3 Références

— 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
— la partie I de *PostgreSQL 14 Internals* : https://postgrespro.com/community/books/internals

Quelques vidéos 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).

2 La gestion transactionnelle

La gestion transactionnelle et les niveaux d'isolation sont des **fonctionnalités clefs** 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** ;
  — la *cohérence* est une notion générale qui englobe celle d'*intégrité*, laquelle pourte uniquement sur les contraintes supportées dans les schémas.
— *Isolation* : les transactions n'agissent **pas les unes sur les autres**, sauf une fois définitivement validées;
— *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, ces contraintes sont souvent relaxées 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 finiront par être cohérentes.

Ce qui est assez fondamentalement opposé à *ACID* !

3 Les niveaux d'isolation et les transactions

Une transaction est un ensemble *atomique* d'opérations DML : `SELECT`, `INSERT` , `UPDATE` et `DELETE`. Pour PostgreSQL les opérations DDL comme `CREATE TABLE` ou `CREATE INDEX` sont aussi gérables dans les transactions.

Les principales commandes relatives aux transactions sont :

— `BEGIN [TRANSACTION]` pour ouvrir une transaction
— pour fermer une transcation, soit :
  — `COMMIT` pour valider, le *succès*
  — `ROLLBACK` pour annuler, l'*échec*

    - soit sous le contrôle du client, soit celui du serveur
- `SAVEPOINT` pour sauvegarder l'état d'une transactions à un point (pour une reprise partielle)

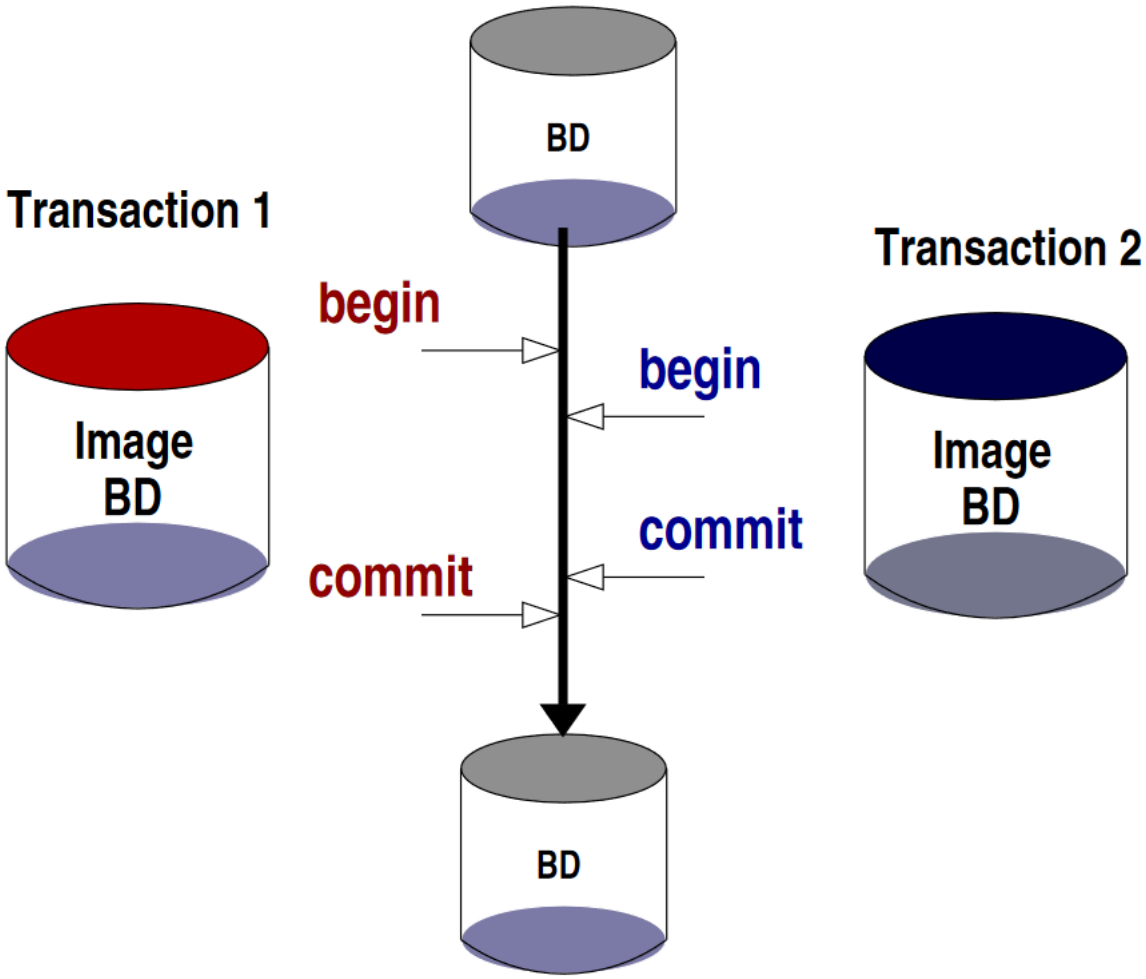

FIGURE 1 – Principe général de l'isolation des données, extrait de ~~*Not*~~ *Only SQL*

Chaque transaction est isolée à un certain point :

- elle ne voit **pas** les opérations des autres avant `COMMIT`
- elle s'exécute **indépendamment** des autres

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau d'isolation est un compromis entre :

- les performances
    - plus le niveau est élevé, plus le coût de la vérification l'est
- le besoin de cohérence applicatif
    - le coût des erreurs de cohérences

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 transaction 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 trop, permettent de contrôler les accès concurrents. On peut consulter la liste 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 PostgreSQL

Le niveau demandé se spécifie à la création de la transaction où à travers les valeurs par défaut via `SHOW default_transaction_isolation;`. Les niveaux SQL sont, dans l'ordre d'importance des garanties :

- `READ UNCOMMITTED`
  - n'existe pas en PostgreSQL, c'est le suivant qui s'applique automatiquement
  - à ce niveau, on peut voir des modifications concurrentes non validées
- `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 lectures)
- `REPEATABLE READ`
  - garanti l'absence des anomalies du niveau inférieur, toutes les lectures doivent donner le même résultat : pas de lecture fantomes
- `SERIALIZABLE` :
  - garanti que le résultat de **toute** exécution concurrente est celui d'une exécution ou les transactions seraient séquentielles

Si le SGBD n'arrive pas à garantir les propriétés du niveau demandé, alors les transactions à problèmes **sont annulées** et il faut les terminer par `ROLLBACK`, 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

On synthétise ici les incohérences acceptées selon le niveau :

level	lost update	dirty read	non-repeatable read	phantom read	other anomalies
Read Uncommitted	–	yes	yes	yes	yes
Read Committed	–	–	yes	yes	yes
Repeatable Read	–	–	–	yes	yes
Serializable	–	–	–	–	–

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;
    -- ici, l'exécution est suspendue
    DELETE FROM list
    WHERE x = (SELECT max(x) FROM list);
COMMIT;
```

On arrive à `DELETE 0` : alors qu'on 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* : c'est **une lecture fantome**.

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).

3.5 Conlusion

Extrait de *PostgreSQL 14 Internals, part I* section 2.4.

> 2.4. Which Isolation Level to Use?
>
> `Read Committed` is the default isolation level in Postgre, and apparently it is this level that is used in the vast majority of applications. This level can be convenient because it allows aborting transactions *only* in case of a failure; **it does not abort any transactions to preserve data consistency**. In other words, serialization failures cannot occur, so you do **not** have to take care of transaction retries.
>
> The downside of this level is **a large number of possible anomalies**, which have been discussed in detail above. A developer has to keep them in mind all the time and write the code in a way that prevents their occurrence. If it is impossible to define all the needed actions in a single statement, then you have to resort to explicit locking. The toughest part is that *the code is hard to test for errors related to data inconsistency;* such errors can appear in unpredictable and barely reproducible ways, so they are very hard to fix too.
>
> The `Repeatable Read` isolation level eliminates some of the inconsistency problems, but alas, not all of them. Therefore, you must not only remember about the remaining anomalies, but also modify the application to **correctly handle serialization failures**, which is certainly inconvenient. However, for *read-only transactions* this level is a perfect complement to the `Read Committed` level; it can be very useful for cases like building reports that involve multiple queries.
>
> And finally, the `Serializable` isolation level allows you not to worry about data consistency at all, which simplifies writing the code to a great extent. The only thing required from the application is *the ability to retry any transaction that is aborted with a serialization failure*. However, the number of aborted transactions and associated overhead can significantly reduce system throughput. You should also keep in mind that the `Serializable` level is not supported on replicas and cannot be combined with other isolation levels.

4 Le modèle MVCC

On va regarder comment fait PostgreSQL pour gérer les versions multiples avec des fenêtres.

- 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

PostgreSQL simplémente le *Multi-Version Concurrency Control* (MVCC), où les tuples peuvent avoir **différentes versions**. Dans une transaction donnée, *une seule version d'un tuple est visible*, mais différentes transacations peuvent voir *différentes versions*.

Attention les valeurs, en particulier les identifiants des transactions, seront *différentes* lors d'une autre exécution. **Attention** pour utiliser les extensions `pgstattuple` et `pageinspect` il faut disposer de privilèges élevés (`pg_stat_scan_tables` et `SUPERUSER` respectivement).

4.1 La représentation physique des données

On comment par voir comment les tuple sont réprésentés *physiquement*. Un tuple contient des méta-données, dont notamment `xmin` et `xmax` qui définissent l'espace de visibilité du tuple dans les transactions.

```
-- 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, avec les droits admin
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 |   4785 |      0 | (0,1)  | \x010000000556
  2 |   8128 |     30 |   4785 |      0 | (0,2)  | \x02000000054c
  3 |   8096 |     30 |   4785 |      0 | (0,3)  | \x030000000546
  4 |   8064 |     30 |   4785 |      0 | (0,4)  | \x040000000558
  5 |   8032 |     30 |   4785 |      0 | (0,5)  | \x050000000547
  6 |   8000 |     30 |   4785 |      0 | (0,6)  | \x060000000556
  7 |   7968 |     30 |   4785 |      0 | (0,7)  | \x070000000555
  8 |   7936 |     30 |   4785 |      0 | (0,8)  | \x080000000552
  9 |   7904 |     30 |   4785 |      0 | (0,9)  | \x090000000551
 10 |   7872 |     30 |   4785 |      0 | (0,10) | \x0a000000054a
```

Depuis la ligne de commande Linux, la commande suivante permet d'avoir la page brute :

```
psql -d bdav -tA -c "select encode(get_raw_page::bytea, 'hex') from get_raw_page('trans',0)" | xxd -p -r
```

4.1.1 Remarque

L'explication de la taille $30 = 24 + 4 + 1 + 1$. Chaque tuple contient une en-tête de 24 octets en plus du contenu binaire. Ici sous Linux x86 64bits, les données sont stockées en *little endian*.

```
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
- https://postgrespro.com/blog/pgsql/5967892 avec des exemple d'utilisation de `pageinspect`.

Chaque table contient des colonnes implicitement présentes comme

- `ctid` : identifiant du tuple au sein de la page,
- `xmin` : le petit id de transcation qui peut voir le tuple
- `xmax` : le plus grand id de transcation qui peut voir le tuple, qui vaut 0 si le tuple n'est pas encore supprimé/modifié.

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() mais qui a
-- le défaut de d'incrémenter le txid à chaque appel
-- SELECT pg_current_xact();

-- 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)  | 4785 |    0 |    0 |    0 |  1 | V
 (0,2)  | 4785 |    0 |    0 |    0 |  2 | L
 (0,3)  | 4785 |    0 |    0 |    0 |  3 | F
 (0,4)  | 4785 |    0 |    0 |    0 |  4 | X
 (0,5)  | 4785 |    0 |    0 |    0 |  5 | G
 (0,6)  | 4785 |    0 |    0 |    0 |  6 | V
 (0,7)  | 4785 |    0 |    0 |    0 |  7 | U
 (0,8)  | 4785 |    0 |    0 |    0 |  8 | R
 (0,9)  | 4785 |    0 |    0 |    0 |  9 | Q
 (0,10) | 4785 |    0 |    0 |    0 | 10 | J
```

Les tuples sont ajoutés **sur le tas** (*heap* en anglais) et en versions multiples. Ainsi, un `UPDATE` indique que l'ancienne version est périmée et qu'une nouvelle existe. On voit que de nouveaux `ctid` on été créés

```
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)  | 4785 |    0 |    0 |    0 |  1 | V
 (0,2)  | 4785 |    0 |    0 |    0 |  2 | L
 (0,3)  | 4785 |    0 |    0 |    0 |  3 | F
 (0,4)  | 4785 |    0 |    0 |    0 |  4 | X
 (0,5)  | 4785 |    0 |    0 |    0 |  5 | G
 (0,11) | 4786 |    0 |    0 |    0 |  6 | v
 (0,12) | 4786 |    0 |    0 |    0 |  7 | u
```

```
 (0,13) | 4786 |    0 |    0 |    0 |  8 | r
 (0,14) | 4786 |    0 |    0 |    0 |  9 | q
 (0,15) | 4786 |    0 |    0 |    0 | 10 | j
```

Avec l'extension `pageinspect` on voit qu'il n'y a pas de suppression mais des ajouts des tuples modifiés : les anciennes versions sont toujours disponibles, pour une session qui aurait commencé **avant** l'`UPDATE`. Une modification consiste ainsi à remplir `t_xmax` avec la valeur courante et `t_ctid` avec la nouvelle version du tuple (ce qui différencie un `UPDATE` d'un `DELETE`).

```
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 |   4785 |      0 | (0,1)  | \x010000000556
  2 |   8128 |     30 |   4785 |      0 | (0,2)  | \x02000000054c
  3 |   8096 |     30 |   4785 |      0 | (0,3)  | \x030000000546
  4 |   8064 |     30 |   4785 |      0 | (0,4)  | \x040000000558
  5 |   8032 |     30 |   4785 |      0 | (0,5)  | \x050000000547
  6 |   8000 |     30 |   4785 |   4786 | (0,11) | \x060000000556
  7 |   7968 |     30 |   4785 |   4786 | (0,12) | \x070000000555
  8 |   7936 |     30 |   4785 |   4786 | (0,13) | \x080000000552
  9 |   7904 |     30 |   4785 |   4786 | (0,14) | \x090000000551
 10 |   7872 |     30 |   4785 |   4786 | (0,15) | \x0a000000054a
 11 |   7840 |     30 |   4786 |      0 | (0,11) | \x060000000576
 12 |   7808 |     30 |   4786 |      0 | (0,12) | \x070000000575
 13 |   7776 |     30 |   4786 |      0 | (0,13) | \x080000000572
 14 |   7744 |     30 |   4786 |      0 | (0,14) | \x090000000571
 15 |   7712 |     30 |   4786 |      0 | (0,15) | \x0a000000056a
```

4.3 Suppression des tuples définitivement innaccessibles

On voit qu'un `DELETE` ne supprime **pas** physiquement les données. Toutefois, si *aucune transaction ne peut accéder à une version ancienne* car il n'existe plus aucune transaction inférieure à `xmax`, alors on peut élaguer la table, c'est l'opération `VACUUM`. Par exemple, si on lance `VACUUM trans;`, les anciennes version des tuples 6, 7, 8, 9 et 10 sont purgées et ne prennent plus de place.

```
 lp | lp_off | lp_len | t_xmin | t_xmax | t_ctid |     t_data
----+--------+--------+--------+--------+--------+----------------
  1 |   8160 |     30 |   4785 |      0 | (0,1)  | \x010000000556
  2 |   8128 |     30 |   4785 |      0 | (0,2)  | \x02000000054c
  3 |   8096 |     30 |   4785 |      0 | (0,3)  | \x030000000546
  4 |   8064 |     30 |   4785 |      0 | (0,4)  | \x040000000558
  5 |   8032 |     30 |   4785 |      0 | (0,5)  | \x050000000547
  6 |     11 |      0 |   NULL |   NULL | NULL   | NULL
  7 |     12 |      0 |   NULL |   NULL | NULL   | NULL
  8 |     13 |      0 |   NULL |   NULL | NULL   | NULL
  9 |     14 |      0 |   NULL |   NULL | NULL   | NULL
 10 |     15 |      0 |   NULL |   NULL | NULL   | NULL
 11 |   8000 |     30 |   4786 |      0 | (0,11) | \x060000000576
 12 |   7968 |     30 |   4786 |      0 | (0,12) | \x070000000575
 13 |   7936 |     30 |   4786 |      0 | (0,13) | \x080000000572
 14 |   7904 |     30 |   4786 |      0 | (0,14) | \x090000000571
 15 |   7872 |     30 |   4786 |      0 | (0,15) | \x0a000000056a
```

4.4 Synthèse

A gros grain, on peut voir chaque paire `(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 appellée *snapshots*, comme dans le schéma ci dessous où les `t_x` sont des tuples et les `i_y` des transactions :

```
      i0  i1  i2  i3  i4  i5  i6  i7  i8  i9
=============================================
t1 : [XXXXXXX[------------------------------
t2 : [XXXXXXXXXXXXXXX[----------------------
t3 : --------[XXXXXXXXXXXXXXX[--------------
t4 : [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX>
t5 : -------------[XXXXXXXXXXXXXXXXXXXXXXXX>
```

- Par exemple, la transaction `i3` voit les tuples `t2`, `t3`, `t4` et `t5` qu'elle vient de créer.
- En revanche, la transaction `i3` ne voit plus `t1` qui a été supprimé en `i1`.

En présence de transactions, c'est un peu plus compliqué car il faut avoir l'état des transactions qui ont `INSERT`, `DELETE` ou `UPDATE`. Des *flags* dans l'en-tête l'indiquent et sont pris en compte dans le calcul des *snapshots*.

Voir par exemple la fonction `heap_page` heap_page.sql qui s'exécute avec `SELECT * FROM heap_page('trans',0);` et rend les bits des transactions plus lisibles.

```
  ctid   | state  |  xmin  |  xmax
--------+--------+--------+--------
 (0,1)  | normal | 4785 c | 0 a
 (0,2)  | normal | 4785 c | 0 a
 (0,3)  | normal | 4785 c | 0 a
 (0,4)  | normal | 4785 c | 0 a
 (0,5)  | normal | 4785 c | 0 a
 (0,6)  | normal | 4785 c | 4786 c
 (0,7)  | normal | 4785 c | 4786 c
 (0,8)  | normal | 4785 c | 4786 c
 (0,9)  | normal | 4785 c | 4786 c
 (0,10) | normal | 4785 c | 4786 c
 (0,11) | normal | 4786 c | 0 a
 (0,12) | normal | 4786 c | 0 a
 (0,13) | normal | 4786 c | 0 a
 (0,14) | normal | 4786 c | 0 a
 (0,15) | normal | 4786 c | 0 a
```

La modélisation complète dépasse le cadre de cette introduction, car y a de nombreuses fonctionnalités qui intéragissent et rendent l'ensemble du mécanisme assez commplexe.