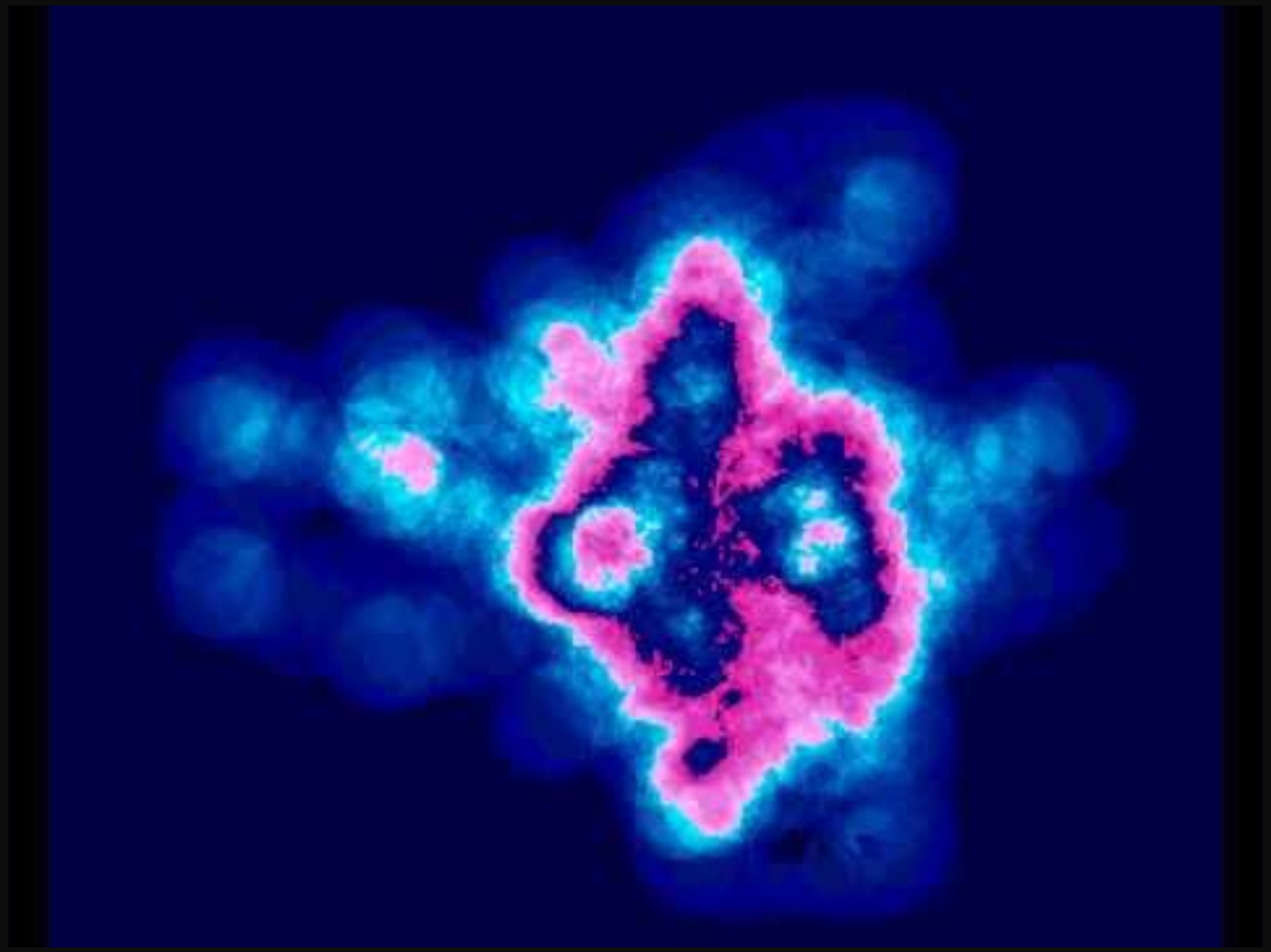
PGDay France 2026

Comprendre les Niveaux d’Isolation Transactionnelle

frederic.delacourt@data-bene.io

Frédéric Delacourt

Demomaker
on Amiga 500



Data Bene

Audit – Consulting – Conception

Technical Assistance

Support – Managed Services

Training

data-bene.io

We are hiring!

Mad or regular (sane) scientists are welcome.

recrutement@data-bene.fr

Agenda

- Les phénomènes du standard SQL
- MVCC
- Read UnCommitted
- Read Committed
- Repeatable Read
- Serializable

De nombreux détails manquent mais je promets de vous transmettre les fondamentaux.

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


UPDATE t SET i=i+1;

COMMIT;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


COMMIT;
```

Où en êtes-vous sur le sujet ? Panique Totale ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


UPDATE t SET i=i+1;

COMMIT;
```

Session B

```
Quelle transaction a BEGIN en premier ?

BEGIN ISOLATION LEVEL READ COMMITTED;


UPDATE t SET i=i+1;


COMMIT;
```

Où en êtes-vous sur le sujet ? Panique Totale ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;



UPDATE t SET i=i+1;

COMMIT;
```

Session B

Quelle transaction a BEGIN en premier ?

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

Ah ouais mais chaque transaction a sa propre vision donc i=1.

```
UPDATE t SET i=i+1;



COMMIT;
```

Où en êtes-vous sur le sujet ? Panique Totale ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;



UPDATE t SET i=i+1;

COMMIT;
```

Session B

Quelle transaction a BEGIN en premier ?

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

Ah ouais mais chaque transaction a sa propre vision donc i=1.

```
UPDATE t SET i=i+1;
```

Ah non attends, au moins une transaction va être ROLLBACK parce qu'il y a un conflit.

```
COMMIT;
```

Où en êtes-vous sur le sujet ? Panique Totale ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;



UPDATE t SET i=i+1;

COMMIT;
```

Session B

Quelle transaction a BEGIN en premier ?

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

Ah ouais mais chaque transaction a sa propre vision donc i=1.

```
UPDATE t SET i=i+1;
```

Ah non attends, au moins une transaction va être ROLLBACK parce qu'il y a un conflit. Non non, en fait i = 2 à la fin, parce qu'on COMMIT en dernier. Arg, j'sais plus…

```
COMMIT;
```

Le standard SQL

La concurrence génère des situations appelées « Phénomènes »

Les phénomènes du Standard SQL

Les phénomènes décrivent
des **situations interdites**
à certains niveaux d'isolation.

Les phénomènes du Standard SQL

Dirty Read Phenomenon

Une transaction lit les modifications
opérées par une autre transaction non validée.

Les phénomènes du Standard SQL

Non Repeatable Read Phenomenon

Une transaction lit des données qu'elle a déjà lu et voit les modifications opérées par une autre transaction (qui a été validée entre temps).

Les phénomènes du Standard SQL

Phantom Read Phenomenon

Une transaction ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée.

Les phénomènes du Standard SQL

Serialization Anomaly Phenomenon

Le résultat de la validation d’un groupe de transactions dépend de l’ordre d’exécution de chacune des transactions.

Les niveaux d’isolation transactionnelle

Isolation Level	Dirty Read	Non Repeatable Read	Phantom Read	Serialization Anomaly
Read Uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read Committed	Not possible	Possible	Possible	Possible
Repeatable Read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

PostgreSQL

Gestion des accès concurrents

MVCC

MultiVersion Concurrency Control

MultiVersion Concurrency Control

- MultiVersion de **QUOI** ?

MultiVersion Concurrency Control

- MultiVersion de **QUOI** ?
- PostgreSQL manipule des **versions de tuples**.

- Une **version de tuple** naît à **xmin** et meurt à **xmax**
- **xmin** et **xmax** sont des **temps transactionnels**.

MultiVersion Concurrency Control

- MultiVersion de **QUOI** ?
- PostgreSQL manipule des **versions de tuples**.

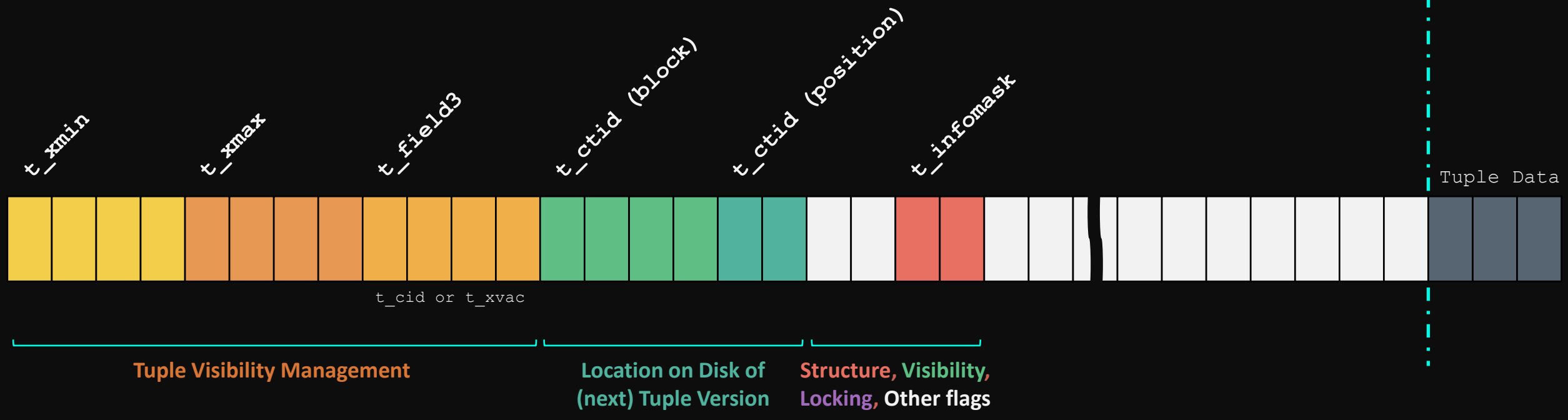
- Une **version de tuple** naît à **xmin** et meurt à **xmax**
- **xmin** et **xmax** sont des **temps transactionnels**.

- INSERT produit une nouvelle **version de tuple** (**xmin**)
- DELETE modifie la **version de tuple** à supprimer (**xmax**).
- UPDATE
  - masque la **version de tuple** à modifier (**xmax**)
  - produit une nouvelle **version de tuple** (**xmin**)

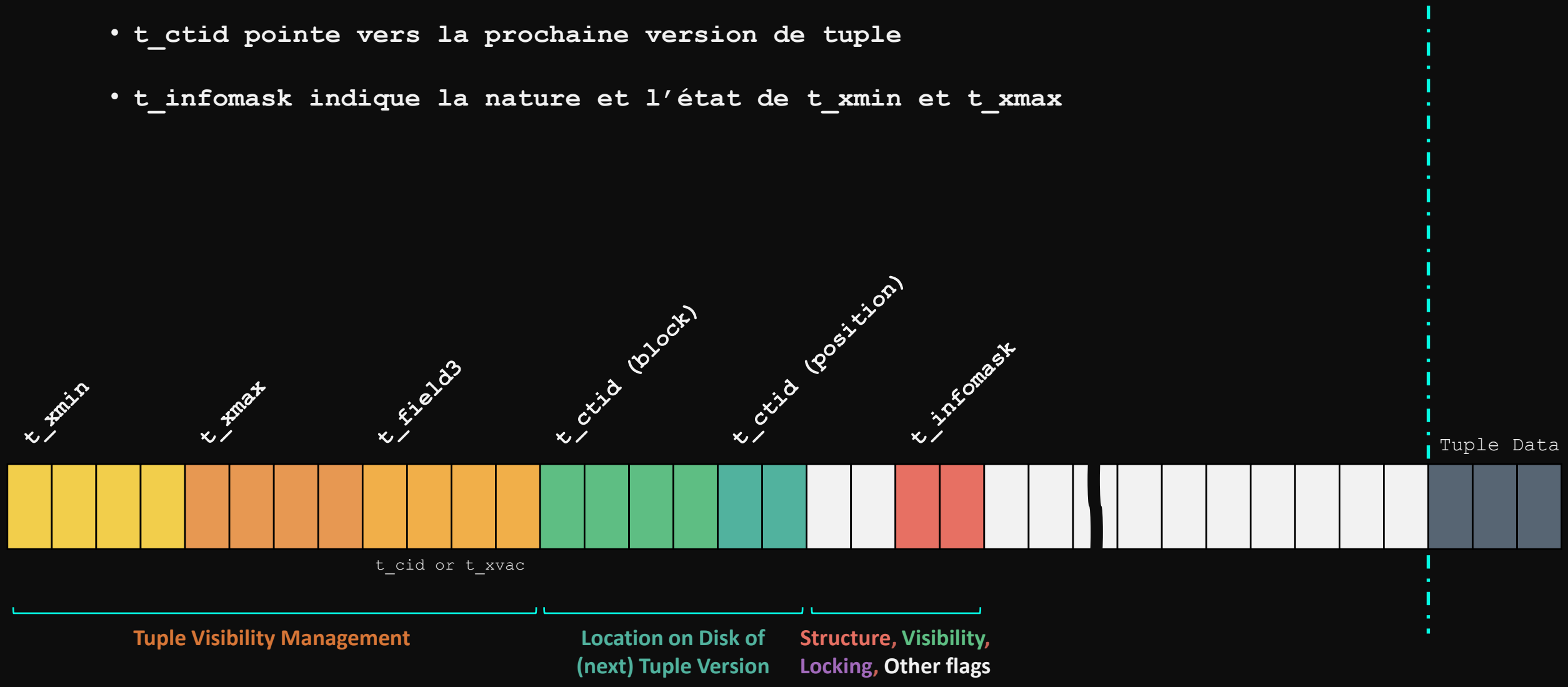
Tuple Header

Structure – Visibilité – RowLocking

```
#define HEAP_HASNULL            0x0001    /* has null attribute(s)            */
#define HEAP_HASVARWIDTH        0x0002    /* has variable-width attribute(s)  */
#define HEAP_HASEXTERNAL        0x0004    /* has external stored attribute(s) */
#define HEAP_HASOID             0x0008    /* has an object-id field           */
#define HEAP_XMAX_KEYSHR_LOCK   0x0010    /* xmax is a key-shared locker      */
#define HEAP_COMBOCID           0x0020    /* t_cid is a combo cid             */
#define HEAP_XMAX_EXCL_LOCK     0x0040    /* xmax is exclusive locker         */
#define HEAP_XMAX_LOCK_ONLY     0x0080    /* xmax, if valid, is only a locker */
#define HEAP_XMIN_COMMITTED     0x0100    /* t_xmin committed                 */
#define HEAP_XMIN_INVALID       0x0200    /* t_xmin invalid/aborted           */
#define HEAP_XMIN_FROZEN        (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
#define HEAP_XMAX_COMMITTED     0x0400    /* t_xmax committed                 */
#define HEAP_XMAX_INVALID       0x0800    /* t_xmax invalid/aborted           */
#define HEAP_XMAX_IS_MULTI      0x1000    /* t_xmax is a MultiXactId          */
#define HEAP_UPDATED            0x2000    /* this is UPDATEd version of row   */
```

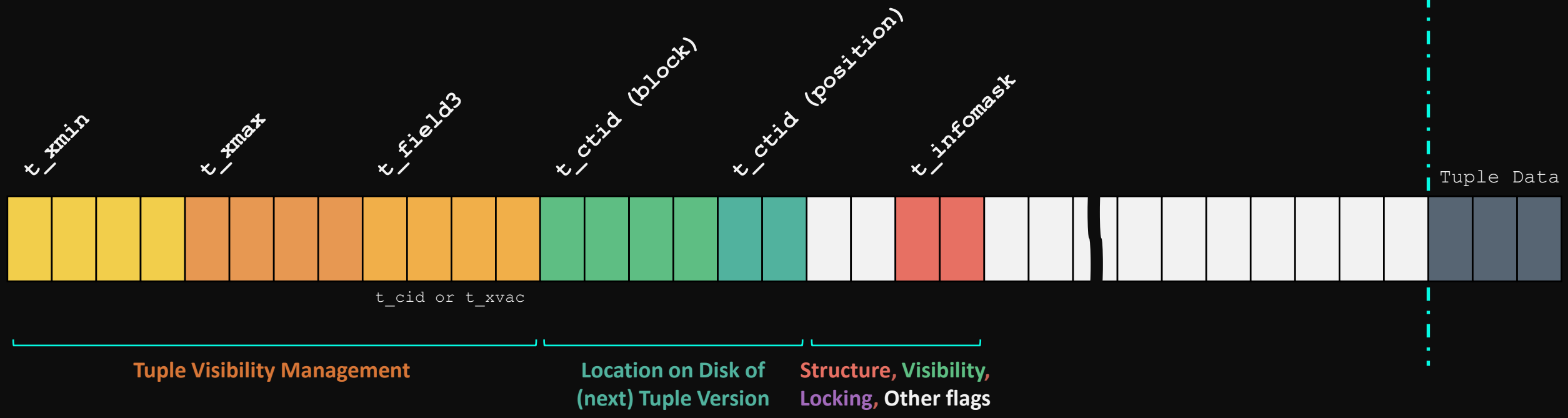


- t_xmin reçoit le xid qui a créé la version de tuple
- t_xmax reçoit le xid qui a masqué la version de tuple
- t_ctid pointe vers la prochaine version de tuple
- t_infomask indique la nature et l'état de t_xmin et t_xmax



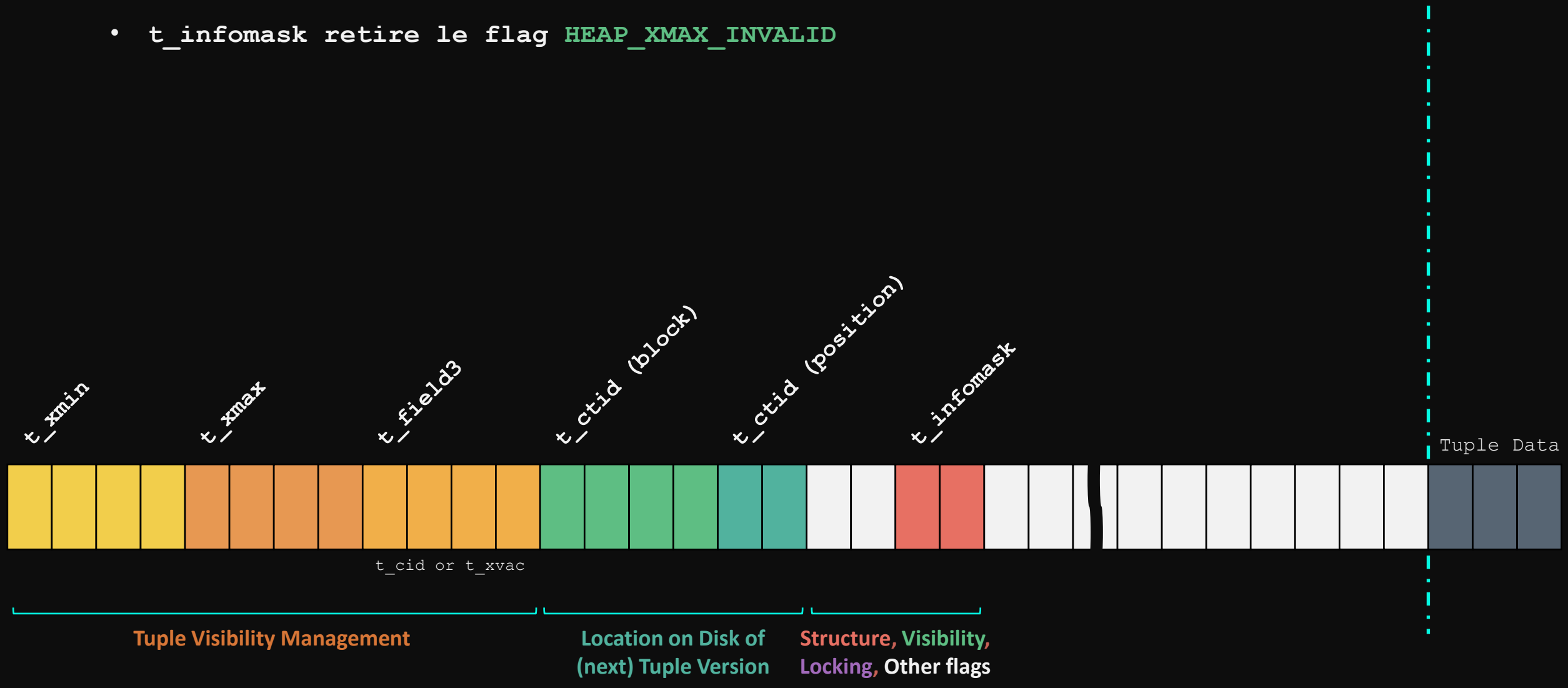
INSERT produit une nouvelle version de tuple

- t_xmin reçoit xid
- t_xmax = 0
- t_ctid pointe sur cette version de tuple
- t_infomask spécifie HEAP_XMAX_INVALID



DELETE masque la version de tuple

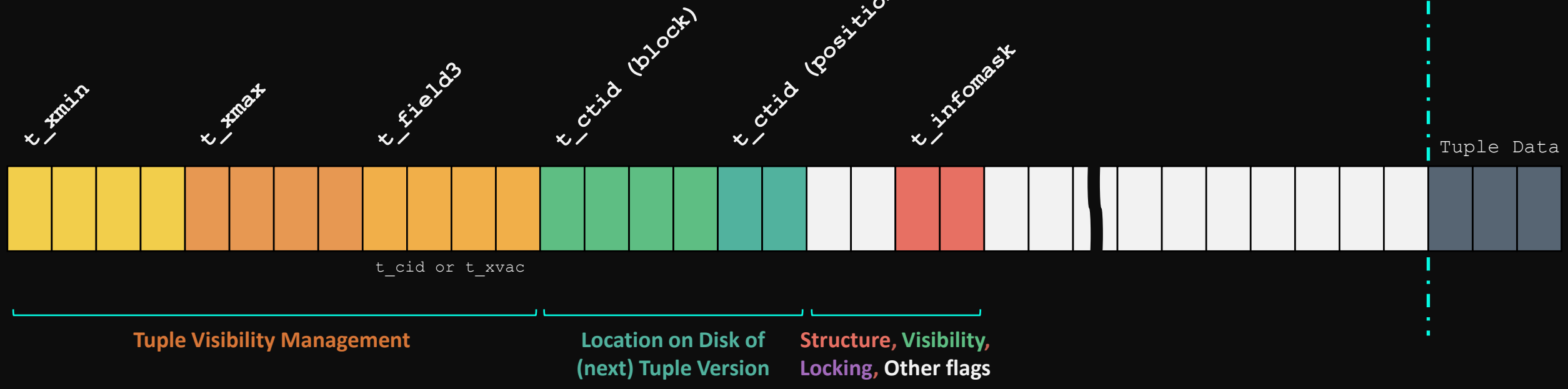
- t_xmax = xid
- t_infomask retire le flag HEAP_XMAX_INVALID



UPDATE

- masque la version de tuple modifiée
  - t_xmax = xid
  - t_infomask spécifie HEAP_XMAX_EXCLUSIVELOCK
  - t_ctid pointe vers la nouvelle version de tuple

- crée une nouvelle version de tuple modifiée
  - t_xmin reçoit xid
  - t_xmax = 0
  - t_ctid pointe sur elle-même
  - t_infomask spécifie HEAP_XMAX_INVALID | HEAP_UPDATED



Locking

Le Minimum Vital du Locking

- **Virtual Transaction ids** (vxid = pid/seq) pendant `StartTransaction()`.
  - Le backend **verrouille** vxid avant publication (wait-lock)
- **Transaction ids** (xid assigné sur ordre DDL et DML).
  - Le backend **verrouille** xid avant publication (wait-lock)
- vxid et xid sont associés à un seul processus.
- Verrous de niveau Table (DDL).
- Verrous de niveau Tuple (DML).
- Lorsqu'un conflit d'accès se produit, le backend courant se met en attente de verrouillage sur le processus bloquant, formant une chaîne d'attente sur verrou.
- Les verrous sont relâchés à la fin d'un **COMMIT** ou d'un **ROLLBACK**.

Snapshot

a.k.a. database version,

photographie de l’état des transactions à un instant donné

Snapshot

- Permet de séparer les transactions Actives des transactions Annulées ou Validées
  - à un instant donné,
  - utilisable aussi longtemps que nécessaire.

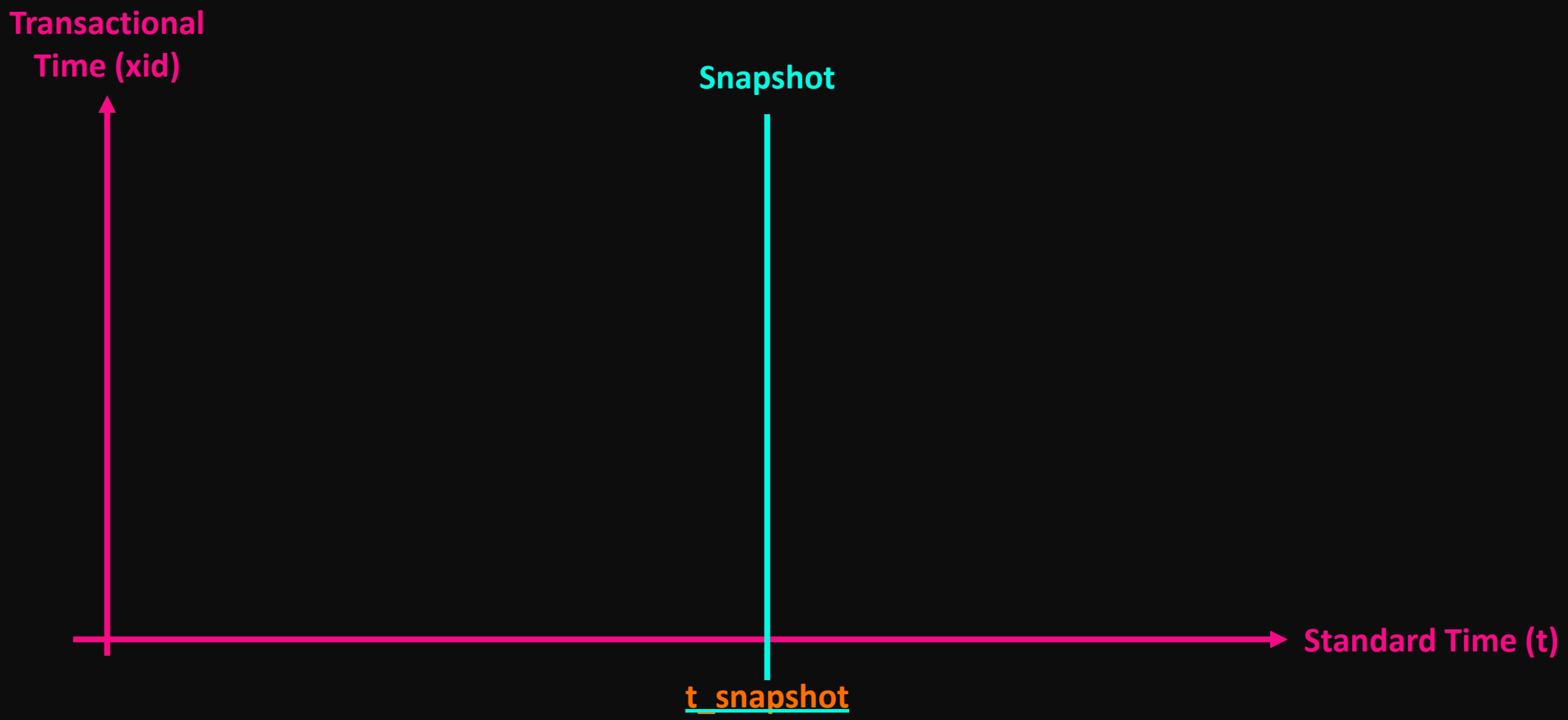
- Soumis à des contraintes de performances fortes

  - La prise de snapshot doit être **raisonnablement** courte,

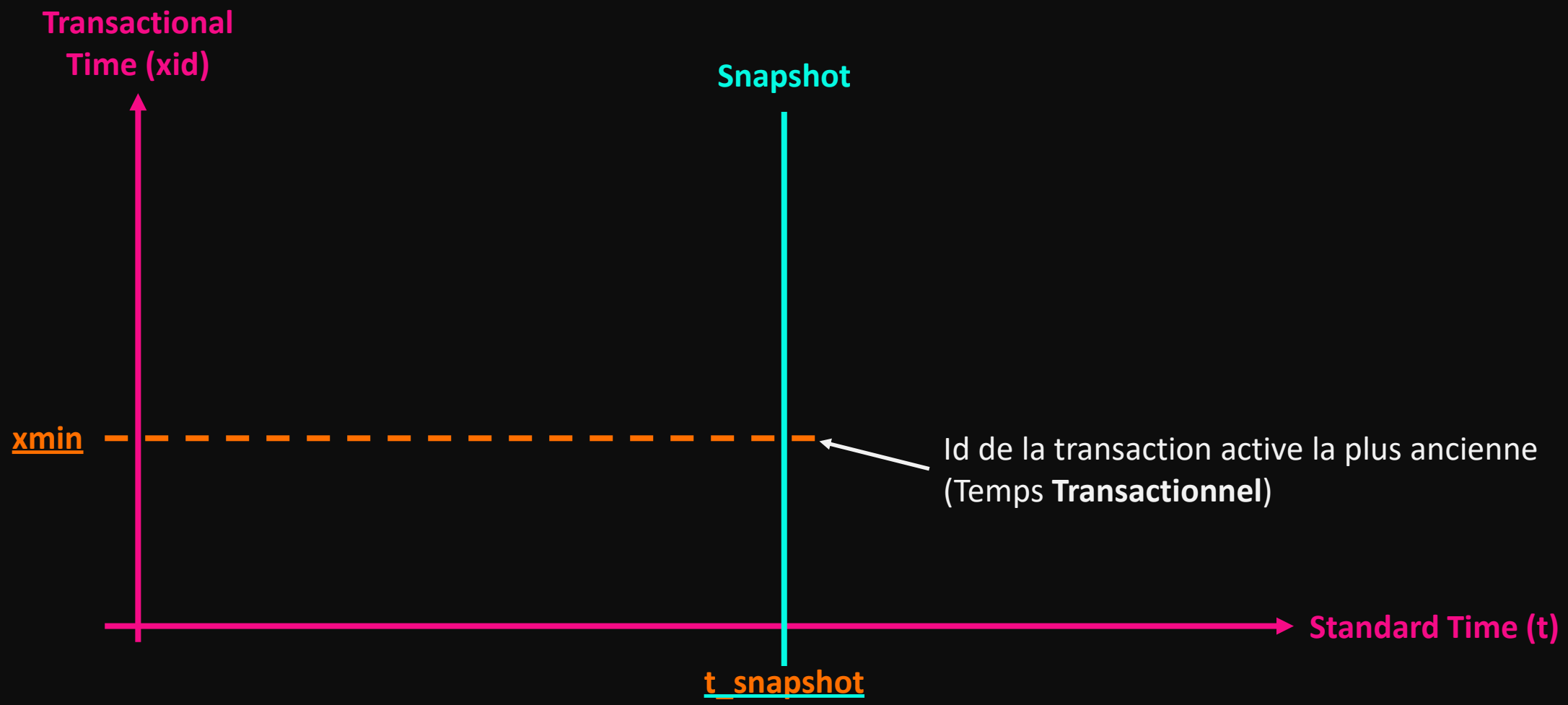
  - L'empreinte mémoire doit être **raisonnablement** faible.

  - Par exemple, il n'est pas **raisonnable** de lister **l'intégralité** des transactions depuis **l'initialisation** de l'instance PostgreSQL, ni même de parcourir intégralement le "`transaction status cache`".

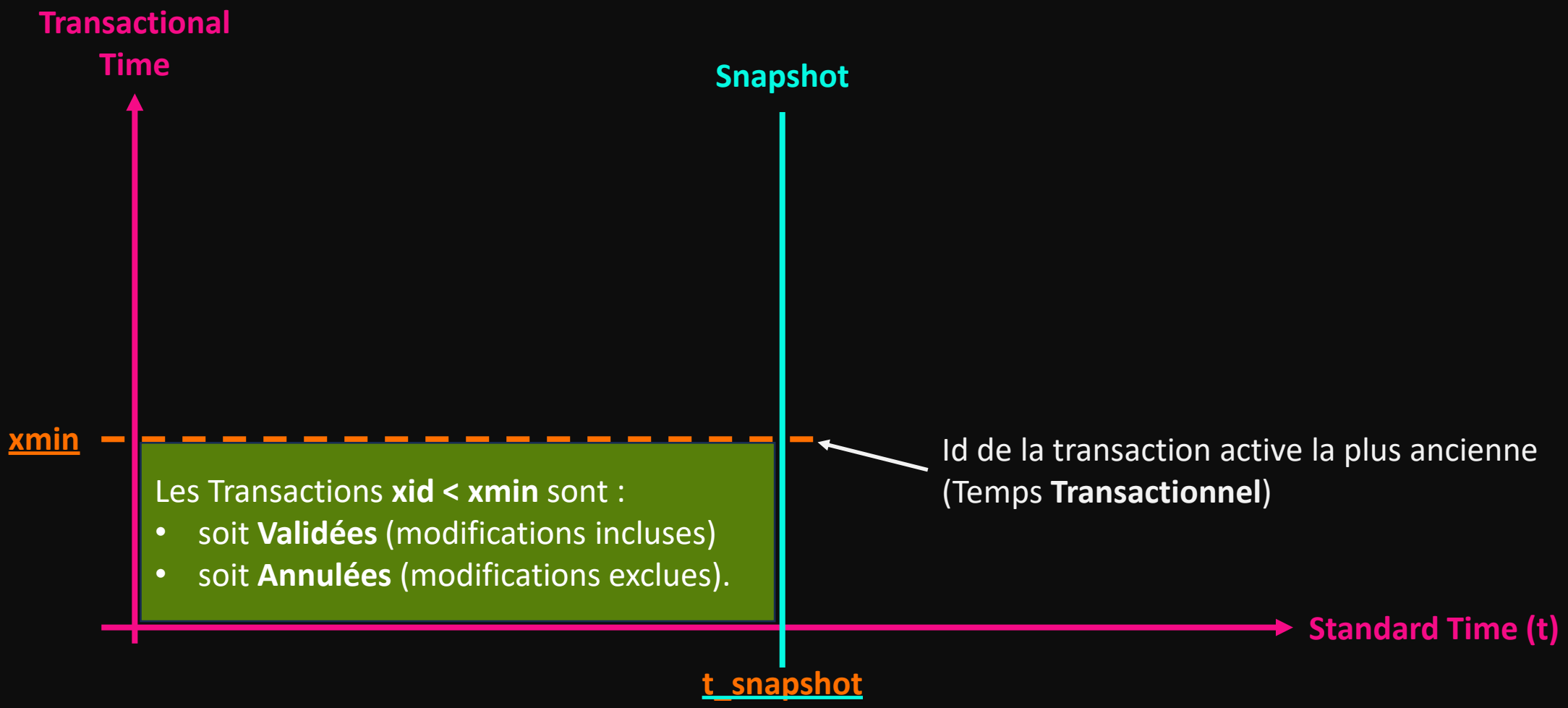
Snapshot – Anatomie



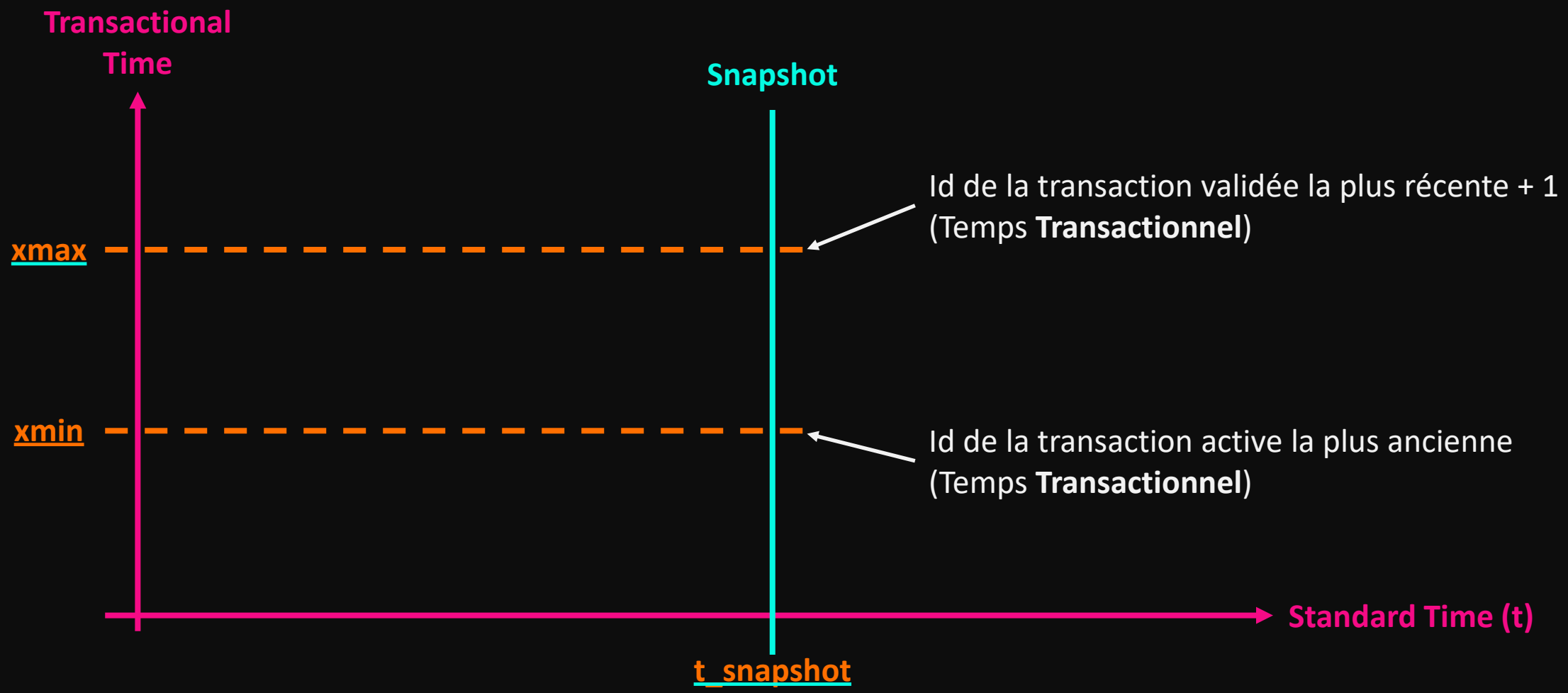
Snapshot – Anatomie



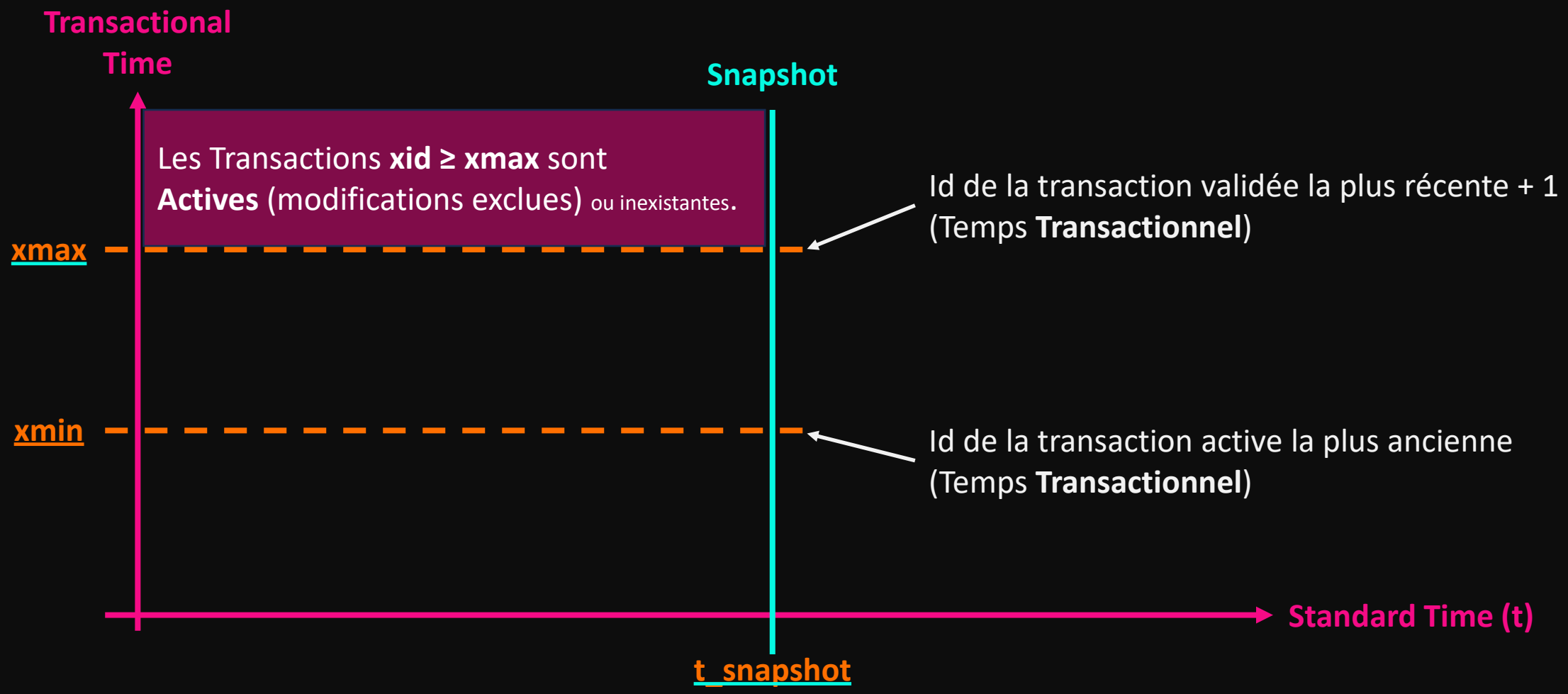
Snapshot – Anatomie



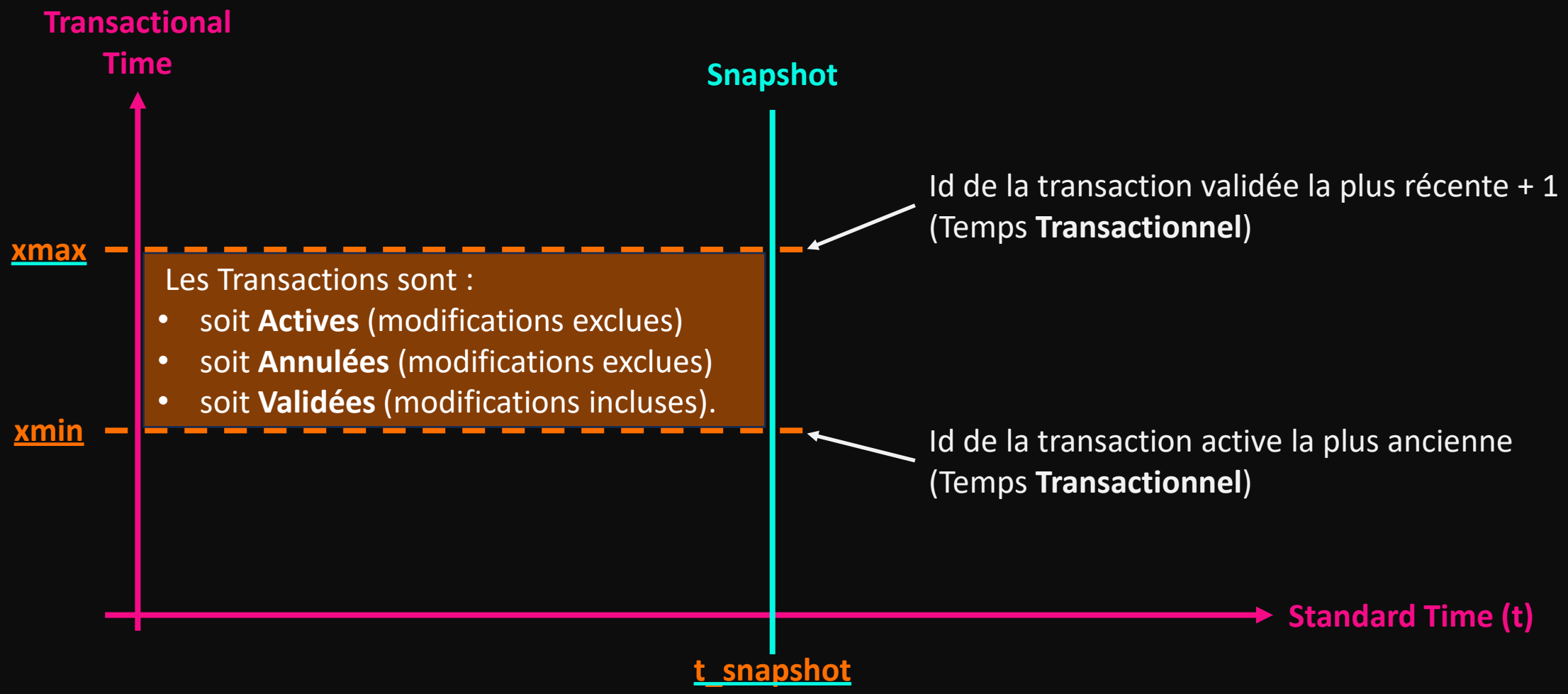
Snapshot – Anatomie



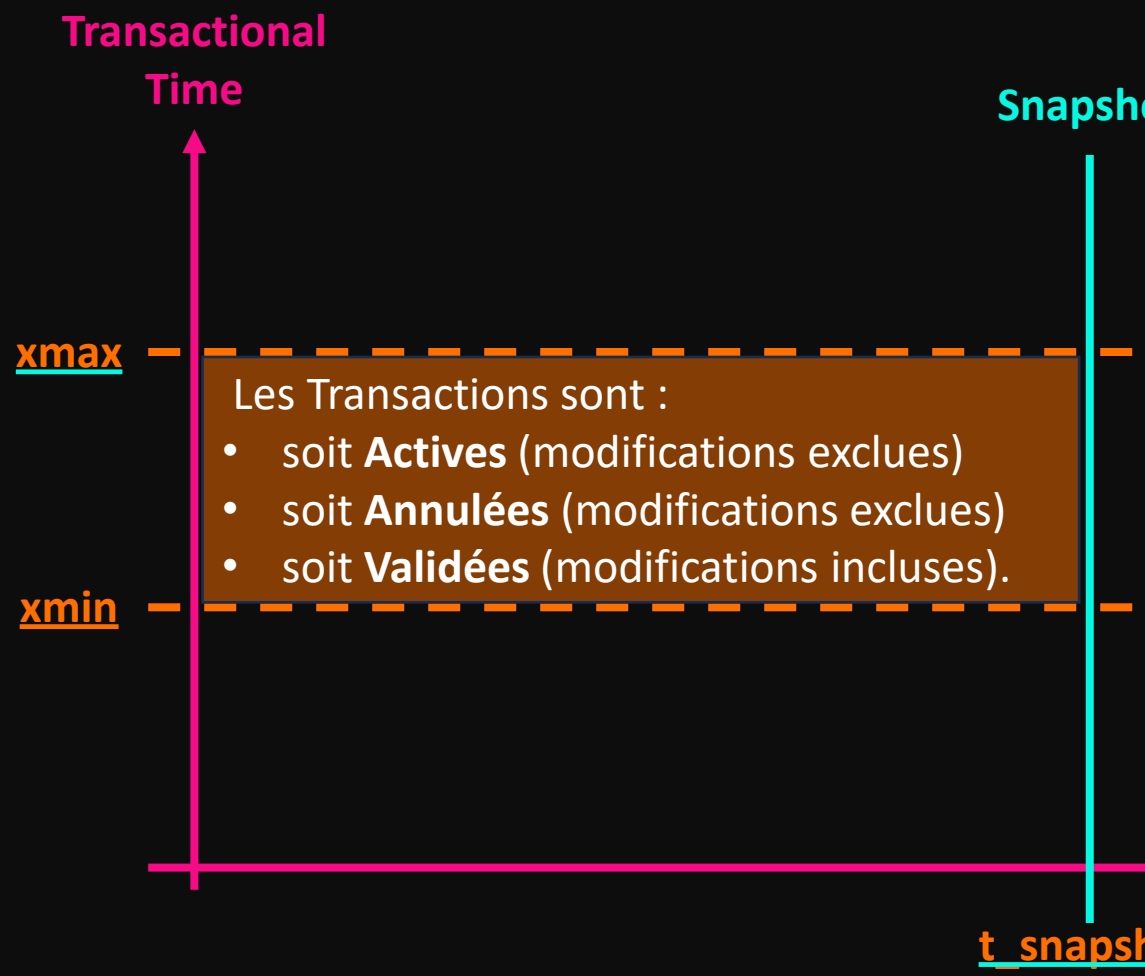
Snapshot – Anatomie



Snapshot – Anatomie



Snapshot – Anatomie



Une transaction **Active** peut devenir **Validée** pendant l'exécution d'une requête de la transaction courante.

Pour **maintenir la cohérence** du snapshot, les transactions **Actives** sont conservées dans la liste **xip_list**. Leurs modifications sont alors exclues.

Snapshot - Résumé

- Anatomie
  - Un **Snapshot** est matérialisé par { **xmin**=x, **xmax**=y, **xip_list**=a,b,c }
  - **xip_list** contient les transactions actives vérifiant **xmin** ⩽ **xid** < **xmax**.

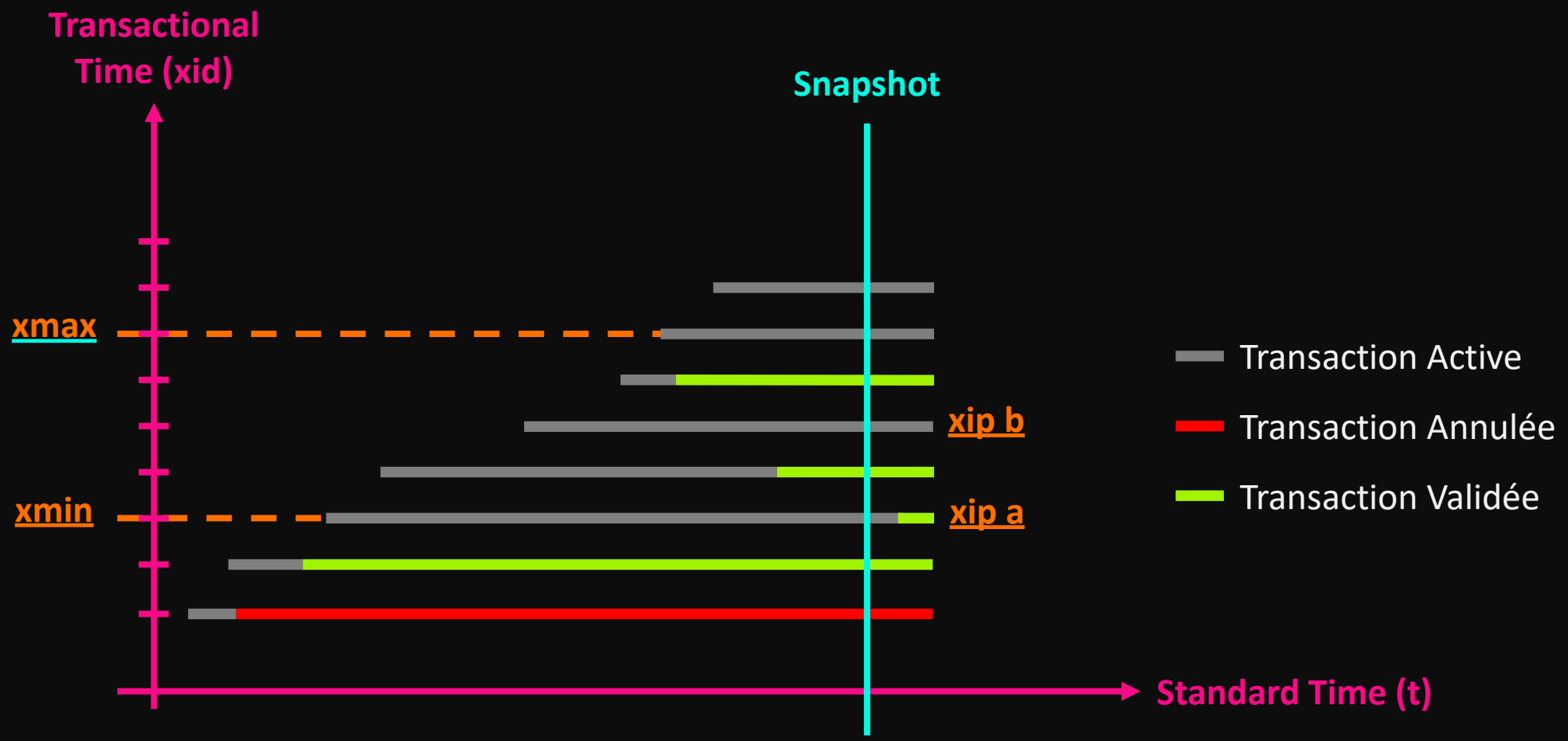
- Règles de prise en compte de l'impact de la transaction xid

  - Lorsque **xid** < **xmin**, la modification validée est toujours prise en compte

  - Lorsque **xmin** ⩽ **xid** < **xmax**, la modification validée est prise en compte sauf si **xid** se trouve dans la liste des **xip**.

backend/storage/ipc/procarray.c, GetSnapshotData function

Snapshot – Exemple



Read Uncommitted

Read UnCommitted? Not in PostgreSQL dictionary!

- Mais, une extension permet de modifier le comportement de PostgreSQL.

- pg_dirtyread
  - 1.0 publiée par Phil Sorber en 2012,
  - 1.1 publiée par Christoph Berg en 2017
    - supporte les colonnes systèmes
  - Christoph est maintenant en charge du projet.

pg_dirtyread – Usage

```
ibiza=# SELECT * FROM pg_dirtyread('t')
        AS t(tableoid oid, ctid tid, xmin xid, xmax xid, cmin cid, cmax cid, dead boolean,
           id int, payload text);
```

tableoid	ctid	xmin	xmax	cmin	cmax	dead	id	payload
17285	(0,1)	1159	0	0	0	f	1	one
17285	(0,2)	1159	0	0	0	f	2	two
17285	(0,3)	1159	0	0	0	f	3	three

Read Committed

Niveau d’isolation transactionnelle par défaut en PostgreSQL

Session	Commands
A-1	`BEGIN;`
A-2	`SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`

- Aucun Snapshot n’est pris par les commandes de gestion de transaction.
- En READ COMMITTED, un Snapshot est pris à chaque statement.
- L’id de transaction (xid) est assigné avec la première commande DDL ou DML.

Session	Commands
A-1	`BEGIN;`
A-2	`SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`

- Aucun Snapshot n'est pris par les commandes de gestion de transaction.
- En READ COMMITTED, un Snapshot est pris à chaque statement.
- L'id de transaction (xid) est assigné avec la première commande DDL ou DML.

est équivalent à

A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+--------------------
     3 |   6 | 2.000000000000000
(1 row)
```

[Snapshot$_A$ = {xmin=1174:xmax=1174:xip=}]

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1173 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1173 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1173 c |    0 a |    0 |    0 | f    | 3 | three
```

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+--------------------
     3 |   6 | 2.000000000000000
(1 row)
```

B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`

- Aucun Snapshot n'est pris par les commandes de gestion de transaction.
- En READ COMMITTED, un Snapshot est pris à chaque statement.
- L'id de transaction (xid) est assigné avec la première commande DDL ou DML.

$[Snapshot_A = \{xmin=1174:xmax=1174:xip=\}]$

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1173 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1173 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1173 c |    0 a |    0 |    0 | f    | 3 | three
```

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+--------------------
     3 |   6 | 2.000000000000000
(1 row)
```

B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`INSERT INTO t VALUES (4, 'four');`

```
INSERT 0 1
```

[xid_B=1174]

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1173 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1173 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1173 c |    0 a |    0 |    0 | f    | 3 | three
 (0,4) | 1174 i |    0 a |    0 |    0 | f    | 4 | four
```

Session	Commands
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`INSERT INTO t VALUES (4, 'four');` `INSERT 0 1`

A-3	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`
	` count | sum | avg`
	`-------+-----+--------------------`
	`     3 |   6 | 2.000000000000000  <= même résultat que précédemment`
	`(1 row)`

	[Snapshot$_A$ = {xmin=1174:xmax=1174:xip=}, xid$_B$=1174]

pg_dirty	` ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |  s`
	`-------+--------+--------+------+------+------+---+-------`
	` (0,1) | 1173 c |    0 a |    0 |    0 | f    | 1 | one`
	` (0,2) | 1173 c |    0 a |    0 |    0 | f    | 2 | two`
	` (0,3) | 1173 c |    0 a |    0 |    0 | f    | 3 | three`
	` (0,4) | 1174 i |    0 a |    0 |    0 | f    | 4 | four`

Session	Commands
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`INSERT INTO t VALUES (4, 'four');` `INSERT 0 1`

A-3	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`
	` count | sum |         avg`
	`-------+-----+--------------------`
	`     3 |   6 | 2.000000000000000  <= même résultat que précédemment`
	`(1 row)`

B-3	`COMMIT;`

pg_dirty	$[xid_B=1174]$
	` ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s`
	`-------+--------+--------+------+------+------+---+-------`
	` (0,1) | 1173 c |    0 a |    0 |    0 | f    | 1 | one`
	` (0,2) | 1173 c |    0 a |    0 |    0 | f    | 2 | two`
	` (0,3) | 1173 c |    0 a |    0 |    0 | f    | 3 | three`
	` (0,4) | 1174 c |    0 a |    0 |    0 | f    | 4 | four`

Session	Commands
A-3	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+--------------------
     3 |   6 | 2.000000000000000  <= même résultat que précédemment
(1 row)
```

B-3 `COMMIT;`

A-4 `SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+--------------------
     4 |  10 | 2.500000000000000  <= le résultat a changé
(1 row)                             => Anomalie « Non repeatable read »
```

$[Snapshot_A = \{xmin=1175:xmax=1175:xip=\}]$

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1173 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1173 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1173 c |    0 a |    0 |    0 | f    | 3 | three
 (0,4) | 1174 c |    0 a |    0 |    0 | f    | 4 | four
```

Read Committed – MultiStatement Query

```
EXPLAIN (COSTS OFF, VERBOSE ON)
  SELECT *
  FROM t
  WHERE i = (SELECT max(i) FROM t);

                                  QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on public.t
   Output: t.i                                        Snapshot 2
   Filter: (t.i = (InitPlan 1).col1)
   InitPlan 1
     ->  Aggregate
           Output: max(t_1.i)                         Snapshot 1
           ->  Seq Scan on public.t t_1
                 Output: t_1.i
(8 rows)
```

Read Committed

Prévenir les incohérences

Read Committed – Prévenir les incohérences

- Les accès concurrents sont un problème ? Verrouillez !

Read Committed – Prévenir les incohérences

- Les accès concurrents sont un problème ? Verrouillez !
- LOCK TABLE est efficace mais… détruit la concurrence.

Read Committed – Prévenir les incohérences

- Les accès concurrents sont un problème ? Verrouillez !
- LOCK TABLE est efficace mais… détruit la concurrence.
- Verrou de niveau ligne
  - SELECT FOR SHARE/UPDATE
  - UPDATE => Optimistic Locking

Read Committed

Prévenir les incohérences – SELECT FOR SHARE

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT * FROM orders WHERE id=1 FOR SHARE;` <= marquées en ShareLock
A-x	… processing …
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`SELECT * FROM orders WHERE id=1 FOR SHARE;` <= no-wait-lock sur backend
B-y	… processing …

FOR SHARE permet le partage d'un référentiel commun à plusieurs transactions actives tout en empêchant sa modification.

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT * FROM orders WHERE id=1 FOR SHARE;` <= marquées en ShareLock
A-x	… processing …
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`SELECT * FROM orders WHERE id=1 FOR UPDATE;` <= RowExclusiveLock
	=> Cette session est bloquée suite à un conflit de niveau de lock.

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT * FROM orders WHERE id=1 FOR SHARE;`
A-x	… processing …
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE orders SET s='xxx' WHERE id=1;`
	=> Cette session est bloquée par A-2
C-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
C-2	`SELECT * FROM orders WHERE id=1 FOR SHARE;`
	=> Cette session est bloquée par B-2

Read Committed

Prévenir les incohérences – SELECT FOR UPDATE

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT * FROM t WHERE id=1 FOR UPDATE;` <= RowExclusiveLock
A-x	… processing …
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`SELECT * FROM t WHERE id=1 FOR SHARE;`
	=> Cette session est bloquée.

FOR UPDATE marque les versions de tuples sélectionnées en accès exclusif.

Entre en conflit avec FOR SHARE, FOR UPDATE, UPDATE et DELETE.

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT * FROM t WHERE id=1 FOR UPDATE;` <= RowExclusiveLock
A-x	… processing …
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`SELECT * FROM t WHERE id=1 FOR UPDATE;`
	=> Cette session est bloquée.

FOR UPDATE marque les versions de tuples sélectionnées en accès exclusif.

Entre en conflit avec FOR SHARE/UPDATE, UPDATE et DELETE.

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`SELECT * FROM t WHERE id=1 FOR UPDATE;` <= RowExclusiveLock
A-x	… processing …
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE t SET s='xxx' WHERE id=1;`
	=> Cette session est bloquée.

FOR UPDATE marque les versions de tuples sélectionnées en accès exclusif.

Entre en conflit avec FOR SHARE/UPDATE, UPDATE et DELETE.

Read Committed

Prévenir les incohérences – UPDATE bloque aussi…

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`

A-2	`SELECT i,s FROM t WHERE i=4;`
	` i |  s`
	`---+------`
	` 4 | four`
	`(1 row)`

	[Snapshot$_A$ = {xmin=1175:xmax=1175:xip=}]
pg_dirty	ctid | xmin | xmax | cmin | cmax | dead | i | s (0,1) | 1173 c | 0 a | 0 | 0 | f | 1 | one (0,2) | 1173 c | 0 a | 0 | 0 | f | 2 | two (0,3) | 1173 c | 0 a | 0 | 0 | f | 3 | three (0,4) | 1174 c | 0 a | 0 | 0 | f | 4 | four

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`

A-2	`SELECT i,s FROM t WHERE i=4;`
	` i |  s`
	`---+------`
	` 4 | four`
	`(1 row)`

A-3	`UPDATE t SET s = 'quatre' WHERE i=4;`
	$[Snapshot_A = \{xmin=1179:xmax=1179:xip=\}, xid_A = 1179]$

pg_dirty	` ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |  s`
	`-------+--------+--------+------+------+------+---+-------`
	` (0,1) | 1173 c |  0 a |  0 |  0 | f  | 1 | one`
	` (0,2) | 1173 c |  0 a |  0 |  0 | f  | 2 | two`
	` (0,3) | 1173 c |  0 a |  0 |  0 | f  | 3 | three`
	` (0,5) | 1174 c | 1179 i |  0 |  0 | f  | 4 | four`
	` (0,5) | 1179 i |  0 a |  0 |  0 | f  | 4 | quatre`

Session	Commands
A-3	`UPDATE t SET s = 'quatre' WHERE i=4;`
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE t SET s='four' WHERE i=4;`

- La version de tuple (0,4) est marquée en RowExclusive
- En attente du COMMIT/ROLLBACK de la Session A (lock du backend)

$[Snapshot_B = \{xmin=1179:xmax=1180:xip=1179\}, xid_A = 1179, xid_B = 1180]$

pg_dirty

ctid	xmin	xmax	cmin	cmax	dead	i	s
(0,1)	1173 c	0 a	0	0	f	1	one
(0,2)	1173 c	0 a	0	0	f	2	two
(0,3)	1173 c	0 a	0	0	f	3	three
(0,5)	1174 c	1179 i	0	0	f	4	four
(0,5)	1179 i	0 a	0	0	f	4	quatre

Session	Commands
A-3	`UPDATE t SET s = 'quatre' WHERE i=4;`
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE t SET s='four' WHERE i=4;`

- La version de tuple (0,4) est marquée en RowExclusive
- En attente du COMMIT/ROLLBACK de la Session A (lock du backend)

A-4	`COMMIT;`

[xid_A était 1179 , xid_B = 1180]

pg_dirty

```
 ctid   |   xmin   |   xmax   | cmin | cmax | dead | i |   s
--------+----------+----------+------+------+------+---+-------
 (0,1)  | 1173 c |     0 a |    0 |    0 | f    | 1 | one
 (0,2)  | 1173 c |     0 a |    0 |    0 | f    | 2 | two
 (0,3)  | 1173 c |     0 a |    0 |    0 | f    | 3 | three
 (0,5)  | 1174 c | 1179 c |    0 |    0 | t    | 4 | four
 (0,5)  | 1179 c |     0 a |    0 |    0 | f    | 4 | quatre
```

Session	Commands
A-3	`UPDATE t SET s = 'quatre' WHERE i=4;`
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE t SET s='four' WHERE i=4;`

- La version de tuple (0,4) est marquée en RowExclusive
- En attente du COMMIT/ROLLBACK de la Session A (lock du backend)

A-4 `COMMIT; => relâche des locks du backend A`

[xid_A était 1179 , xid_B = 1180]

pg_dirty

ctid	xmin	xmax	cmin	cmax	dead	i	s
(0,1)	1173 c	0 a	0	0	f	1	one
(0,2)	1173 c	0 a	0	0	f	2	two
(0,3)	1173 c	0 a	0	0	f	3	three
(0,5)	1174 c	1179 c	0	0	t	4	four
(0,5)	1179 c	0 a	0	0	f	4	quatre

Session	Commands
A-3	`UPDATE t SET s = 'quatre' WHERE i=4;`
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE t SET s='four' WHERE i=4;`

- La version de tuple (0,4) est marquée en RowExclusive
- En attente du COMMIT/ROLLBACK de la Session A (lock du backend)

A-4	`COMMIT;` => relâche des locks du backend A
B-3	UPDATE re-lit (0,4), tient compte des changements et continue sa tâche

[xid_A était 1179 , xid_B = 1180]

pg_dirty

ctid	xmin	xmax	cmin	cmax	dead	i	s
(0,1)	1173 c	0 a	0	0	f	1	one
(0,2)	1173 c	0 a	0	0	f	2	two
(0,3)	1173 c	0 a	0	0	f	3	three
(0,5)	1174 c	1179 c	0	0	t	4	four
(0,6)	1179 c	1180 i	0	0	f	4	quatre
(0,6)	1180 i	0 a	0	0	f	4	four

Session	Commands
A-3	`UPDATE t SET s = 'quatre' WHERE i=4;`
B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`UPDATE t SET s='four' WHERE i=4;`
	• La version de tuple (0,4) est marquée en RowExclusive • En attente du COMMIT/ROLLBACK de la Session A (lock du backend)
A-4	`COMMIT;` => relâche des locks du backend A
B-3	UPDATE re-lit (0,4), tient compte des changements et continue sa tâche
B-4	`COMMIT;`

[xid_A était 1179 , xid_B = 1180]

pg_dirty

ctid	xmin	xmax	cmin	cmax	dead	i	s
(0,1)	1173 c	0 a	0	0	f	1	one
(0,2)	1173 c	0 a	0	0	f	2	two
(0,3)	1173 c	0 a	0	0	f	3	three
(0,5)	1174 c	1179 c	0	0	t	4	four
(0,6)	1179 c	1180 c	0	0	t	4	quatre
(0,6)	1180 c	0 a	0	0	f	4	four

Read Committed – Cas d’usages typiques

- **Application Web générant des opérations CRUD**
- **Optimistic Locking**
- **Queueing**
- …

Read Committed

Prévenir les incohérences – Optimistic Locking

Optimistic Locking (quelques mots)

```
CREATE TABLE t ( id int, c text, version int );

SELECT id,c,version FROM t WHERE id = 53;
...
UPDATE t SET c = 'new', version = version + 1
WHERE id = 53 AND version = 1;
```

=> zero tuple modifié => conflit => on recommence depuis le SELECT version

L'optimistic locking peut s'étaler sur plusieurs transactions (e.g.; micro-services)

mais peut générer de nombreuses requêtes en cas de conflit.

Read Committed

Prévenir les incohérences – Gestion de queue

Session	Commands

I-1	`CREATE TABLE queue(`
	`  id bigint GENERATED ALWAYS AS IDENTITY`
	`  , inserted_at timestamp with time zone default now()`
	`  , modified_at timestamp with time zone default now()`
	`  , state int default 0  -- 0 = nouveau, 1 = en attente de, …`
	`  , payload jsonb  -- details de l'objet, de la tâche, autre…`
	`) WITH (fillfactor = 70);`

P-1	`INSERT INTO queue (payload)` `VALUES ('{}'::jsonb),('{}'::jsonb),('{}'::jsonb);`
C-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
C-2	`SELECT id,payload FROM queue WHERE state = 0 ORDER BY id ASC LIMIT 1` `FOR UPDATE SKIP LOCKED;`
	…

C-3	`UPDATE queue`
	`SET state = 1, modified_at = now(), payload = '{"a":1}'::jsonb`
	`WHERE id = 1;`

C-4	`COMMIT;`

La gestion de la concurrence des accès
avec FOR SHARE, FOR UPDATE et UPDATE,
oblige les développeurs à écrire du code
spécifique à de nombreux endroits.

La gestion de la concurrence des accès
avec FOR SHARE, FOR UPDATE et UPDATE,
oblige les développeurs à écrire du code
spécifique à de nombreux endroits.
(sans compter les tris pour éviter les deadlocks…)

La gestion de la concurrence des accès
avec FOR SHARE, FOR UPDATE et UPDATE,
oblige les développeurs à écrire du code
spécifique à de nombreux endroits.
(sans compter les tris pour éviter les deadlocks…)

Chaque oubli conduit à un bug de concurrence.

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


UPDATE t SET i=i+1;

COMMIT;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


COMMIT;
```

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;
```

- La table t ne contient qu'une seule version de tuple pour laquelle i=1.

Session B

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;
```

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

- Les commandes de gestion de transaction ne s'occupent pas de Snapshots, elles définissent le niveau d'isolation transactionnelle requis.
- Le backend acquiert un lock sur lui-même pour gérer les wait-locks à venir.

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

- L'ordre d'exécution des BEGIN et SET ne joue aucun rôle dans la prise de Snapshot ni de la génération des id des transactions.

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;
```

- La commande UPDATE génère un xid et un Snapshot.
- Elle marque la version de tuple i=1 en RowExclusiveAccess avec son xid dans t_xmax
- Elle produit une nouvelle version de tuple i=2 avec son xid dans t_xmin.

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;
```

- La commande UPDATE génère un xid et un Snapshot.
- Elle lit la version de tuple i=1 marquée RowExclusiveLock par xid de A.
- Xid A permet d'identifier le backend de A pour attendre qu'il relâche le lock sur lui-même

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;

UPDATE t SET i=i+1;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1; [bloqué par Session A]
```

- Aucun conflit n'existe à ce stade, une nouvelle version de tuple est produite i=3

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;

UPDATE t SET i=i+1;

COMMIT;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1; [bloqué par Session A]
```

- Le backend A relâche le lock sur lui-même.

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


UPDATE t SET i=i+1;

COMMIT;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;


UPDATE t SET i=i+1;
```

- Le backend A a relâché le lock sur lui-même
- L'UPDATE n'est plus bloqué
- L'UPDATE relit la version de tuple i=1 et constate qu'il y a eu des modifications jusqu'à i=3
- L'UPDATE produit une nouvelle version de tuple i=4

Où en êtes-vous sur le sujet ?

Session A

```
CREATE TABLE t AS SELECT 1::int AS i;

BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


UPDATE t SET i=i+1;

COMMIT;
```

Session B

```
BEGIN ISOLATION LEVEL READ COMMITTED;

UPDATE t SET i=i+1;


COMMIT;  => maintenant, i=4 (+3 dead tuples)
```

Repeatable Read (snapshot isolation)

Un seul Snapshot pour toute la transaction !

Repeatable Read (snapshot isolation)

READ vs WRITE

Session	Commands
A-1	`CREATE TABLE t(i int, s text);`
A-2	`INSERT INTO t VALUES (1, 'one'),(2,'two'),(3,'three');`
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+---------------------
     3 |   6 | 2.000000000000000
(1 row)
```

- En REPEATABLE READ, un Snapshot de référence est pris au premier statement exécuté.
- L'id de transaction (xid) est assigné avec la première commande DDL ou DML.

$[Snapshot_A = \{xmin=1184:xmax=1184:xip=\}]$

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1183 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1183 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1183 c |    0 a |    0 |    0 | f    | 3 | three
```

Session	Commands
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+---------------------
     3 |   6 | 2.0000000000000000
```

B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`INSERT INTO t VALUES (4,'four');`

$[Snapshot_A = \{xmin=1184:xmax=1184:xip=\}, xid_B = 1184]$

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1183 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1183 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1183 c |    0 a |    0 |    0 | f    | 3 | three
 (0,4) | 1184 i |    0 a |    0 |    0 | f    | 3 | four
```

Session	Commands
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+---------------------
     3 |   6 | 2.0000000000000000
```

B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`INSERT INTO t VALUES (4,'four');`
B-3	`COMMIT;`

[$Snapshot_A$ = {xmin=1184:xmax=1184:xip=}, xid_B = 1184]

pg_dirty

```
 ctid  |  xmin  |  xmax  | cmin | cmax | dead | i |   s
-------+--------+--------+------+------+------+---+-------
 (0,1) | 1183 c |    0 a |    0 |    0 | f    | 1 | one
 (0,2) | 1183 c |    0 a |    0 |    0 | f    | 2 | two
 (0,3) | 1183 c |    0 a |    0 |    0 | f    | 3 | three
 (0,4) | 1184 c |    0 a |    0 |    0 | f    | 3 | four
```

Session	Commands
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+---------------------
     3 |   6 | 2.0000000000000000
```

B-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
B-2	`INSERT INTO t VALUES (4,'four');`
B-3	`COMMIT;`
A-5	`SELECT COUNT(*), SUM(i), AVG(i) FROM t;`

```
 count | sum |         avg
-------+-----+---------------------
     3 |   6 | 2.0000000000000000     <= invariance du résultat/snapshot
```

[$Snapshot_A$ = {xmin=1184:xmax=1184:xip=}, xid_B = 1184]

pg_dirty

```
 ctid   |  xmin   |  xmax   | cmin | cmax | dead | i |   s
--------+---------+---------+------+------+------+---+-------
 (0,1)  | 1183 c  |    0 a  |    0 |    0 | f    | 1 | one
 (0,2)  | 1183 c  |    0 a  |    0 |    0 | f    | 2 | two
 (0,3)  | 1183 c  |    0 a  |    0 |    0 | f    | 3 | three
 (0,4)  | 1184 c  |    0 a  |    0 |    0 | f    | 3 | four
```

Repeatable Read

WRITE vs WRITE

Session	Commands
A-1	`BEGIN ISOLATION LEVEL READ COMMITTED;`
A-2	`UPDATE t SET s='xxx' WHERE i=3;`
B-1	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
B-2	`UPDATE t SET s='yyy' WHERE i=3;`
	=> Bloquée
A-3	`COMMIT;`
B-2	`UPDATE t SET s='yyy' WHERE i=3;`
	=> actualise la version de tuple i=3
	=> une modification a été détectée
	=> ERROR: could not serialize access due to concurrent update
B-3	`ROLLBACK;`
	Rejeu de la transaction B tant que (40001,serialization_failure) se produit.

Repeatable Read

WRITE vs WRITE et indépendance

Session	Commands
A-1	`CREATE TABLE t ( i int, c int );`
A-2	`INSERT INTO t VALUES (1,1),(2,1),(3,2),(4,2);`
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`UPDATE t SET c = 2 WHERE c = 1;`
B-1	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
B-2	`UPDATE t SET c = 1 WHERE c = 2;`
A-5	`COMMIT;`
B-3	`COMMIT;`
A-6	`SELECT * FROM t;`

```
 i | c
---+---
 1 | 2
 2 | 2
 3 | 1
 4 | 1
(4 rows)
```

Session	Commands
A-1	`CREATE TABLE t ( i int, c int );`
A-2	`INSERT INTO t VALUES (1,1),(2,1),(3,2),(4,2);`
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`UPDATE t SET c = 2 WHERE c = 1;`
B-1	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
B-2	`UPDATE t SET c = 1 WHERE c = 2;`
A-5	`COMMIT;`
B-3	`COMMIT;`
A-6	`SELECT * FROM t;`

```
 i | c
---+---
 1 | 2
 2 | 2
 3 | 1
 4 | 1
(4 rows)
```

Le résultat me semble correct.
Et pourtant, qu'est-ce qui ne va pas ?

Session	Commands
A-1	`CREATE TABLE t ( i int, c int );`
A-2	`INSERT INTO t VALUES (1,1),(2,1),(3,2),(4,2);`
A-3	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
A-4	`UPDATE t SET c = 2 WHERE c = 1;`
B-1	`BEGIN ISOLATION LEVEL REPEATABLE READ;`
B-2	`UPDATE t SET c = 1 WHERE c = 2;`
A-5	`COMMIT;`
B-3	`COMMIT;`
A-6	`SELECT * FROM t;`

```
 i | c
---+---
 1 | 2
 2 | 2
 3 | 1
 4 | 1
(4 rows)
```

Le résultat me semble correct.
Et pourtant, qu'est-ce qui ne va pas ?

Le résultat dépend de l'ordre d'exécution des transactions.
L'exécution de A puis B ou de B puis A ou de A et B entrelacées produit 3 résultats différents…

Ceci est le phénomène Serialization Anomaly.

Repeatable Read

Les cas d’usages typiques

REPEATABLE READ – Les cas d'usages typiques

- Construction de rapports nécessitant la cohérence des données traitées.
- Lorsqu'un partage de Snapshots permet la parallélisation des traitements
  - pg_export_snapshot()
- Des erreurs sont attendues (40001, serialization_failure), on recommence…
- Avez-vous déjà utilisé ?
  - REPEATABLE READ
  - pg_export_snapshot()

REPEATABLE READ – Les cas d'usages typiques

- Construction de rapports nécessitant la cohérence des données traitées.
- Lorsqu'un partage de Snapshots permet la parallélisation des traitements
  - pg_export_snapshot()
- Des erreurs sont attendues (40001, serialization_failure), on recommence…
- Avez-vous déjà utilisé ?
  - REPEATABLE READ => pg_dump
  - pg_export_snapshot() => pg_dump --jobs

Serializable

Offre la garantie que l’ordre d’exécution des transactions actives n’influe pas sur le résultat de l’exécution de ces transactions.

SERIALIZABLE Isolation Level

- Basé sur la technique « Serializable Snapshot Isolation ».
- Thèse de Cahill et al. en 2008
- Implémenté dans PostgreSQL par Kevin Grittner and Dan Ports en 2012
- Détails d'implémentation: https://arxiv.org/pdf/1208.4179
- SERIALIZED est le niveau d'isolation par défaut défini par la norme SQL

SERIALIZABLE - Strict 2 Phase Locking

- Tous les tuples modifiés sont verrouillés
- Tous les tuples lus sont verrouillés

- Ce verrouillage strict permet la détection de conflit de concurrence
  - Dus aux dépendances Read/Read, Read/Write, Write/Write et Write/Read entre transactions

- Impacte les performances et provoque beaucoup d'écritures disques.

- En conflit avec «**readers don't block writers, and writers don't block readers**»

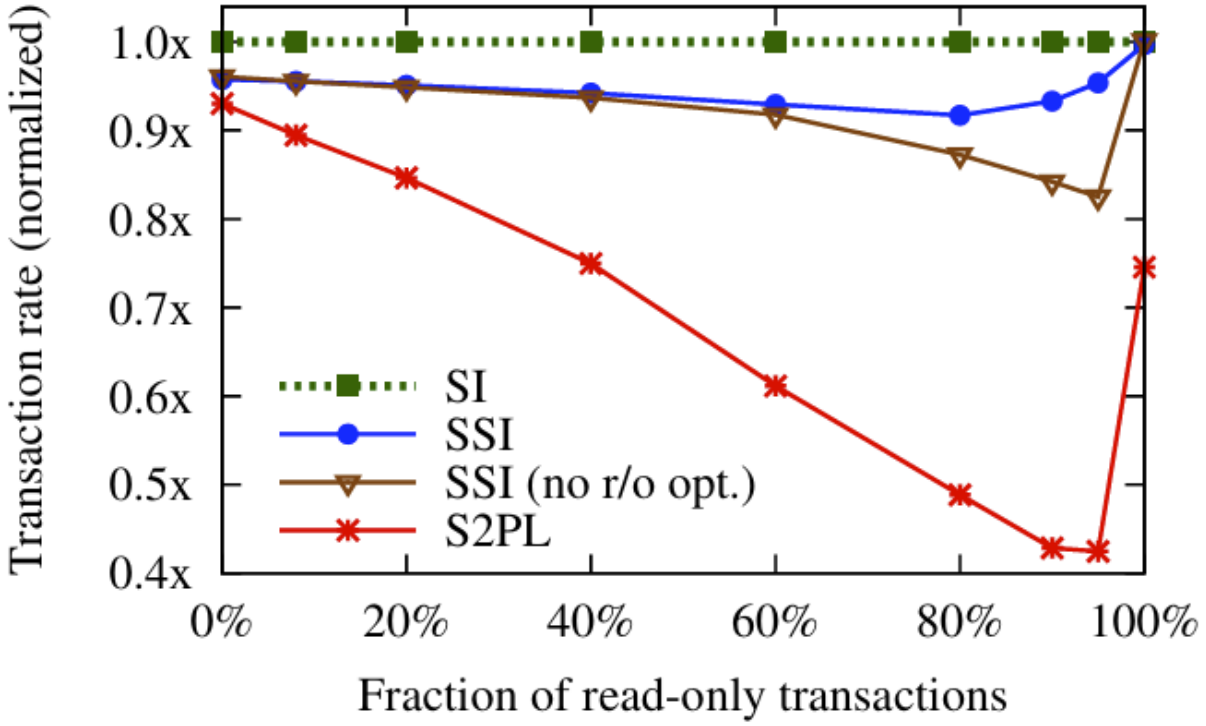
SERIALIZABLE – Serialized Snapshot Isolation

- La technique **Snapshot Isolation**
  - est déjà implémentée dans PostgreSQL (REPEATABLE READ)
  - provoque l'arrêt avec erreur d'une transaction en conflit de concurrence
  - impose le rejeu de la transaction sur erreur (40001, serialization_failure)

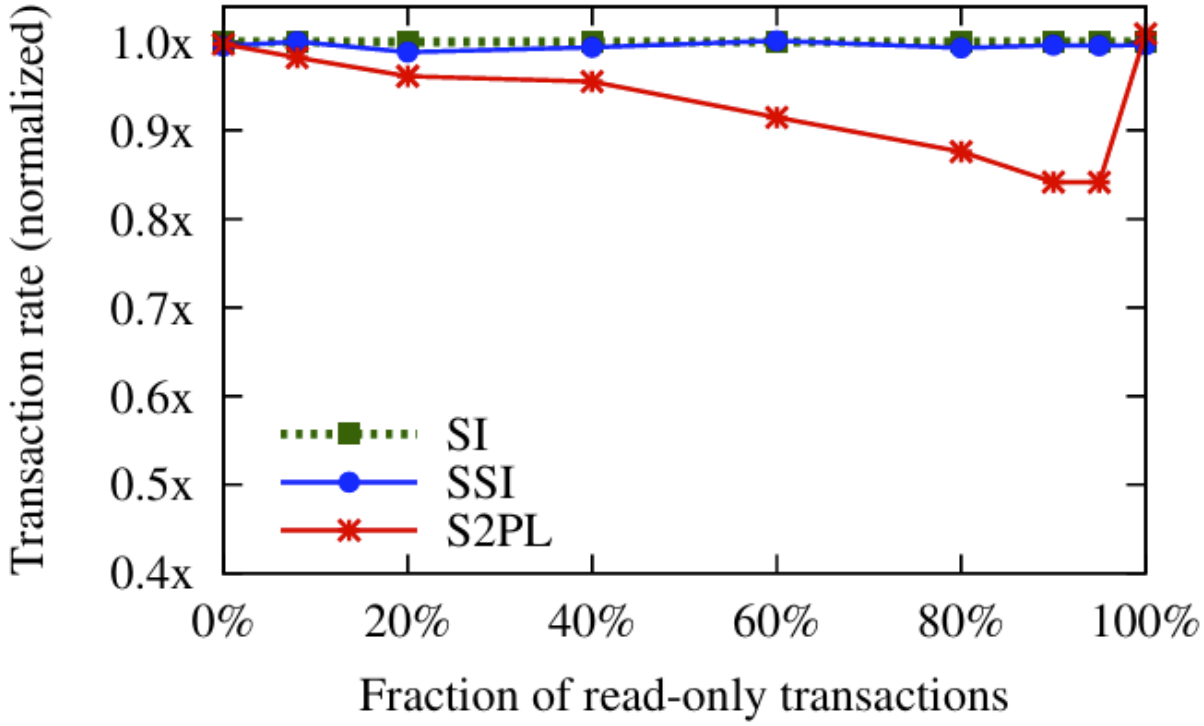
- La technique **Serialized Snapshot Isolation**
  - est basée sur un graphe d'historisation de dépendance d'accès aux « **objets** »
  - remplace la détection de cycles par
    - la détection du motif $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$
    - lorsque T_1 est Read Only et T_3 est validée avant que T_1 ne prenne son snapshot
    - ce motif est toujours présent lors d'un conflit de dépendance => Serialization Failure
  - mais produit des faux-positifs (1%) qui génèrent (40001, serialization_failure)
  - meilleures performances que S2PL et autorise une meilleure concurrence des accès

  - Hint: BEGIN TRANSACTION READ ONLY DEFERRABLE force un **safe snapshot** qui évite un **SIREAD lock**.

SERIALIZABLE - Performance



(a) in-memory configuration (25 warehouses)



(b) disk-bound configuration (150 warehouses)

Figure 5: DBT-2++ transaction throughput for SSI and S2PL as a percentage of SI throughput

SERIALIZABLE – taux d'échec de sérialisation

	Throughput (req/s)	Serialization failures
SI	435	0.004%
SSI	422	0.03%
S2PL	208	0.76%

Figure 6: RUBiS performance

Session	Commands
A-1	`BEGIN ISOLATION LEVEL SERIALIZABLE;`
A-2	`SELECT …`
A-3	`UPDATE …` Le locking est toujours à l'oeuvre, attention aux deadlocks.
A-4	`COMMIT;`
B-1	`BEGIN ISOLATION LEVEL SERIALIZABLE;`
B-2	`SELECT …`
B-3	`UPDATE …` Le locking est toujours à l'oeuvre, attention aux deadlocks.
B-4	`COMMIT;`

SERIALIZABLE – Cas d'usages typiques

- Quand tu parles de manipuler l'argent sur mon compte bancaire…
- Quand tu ne veux pas gérer spécifiquement la concurrence des accès
  - Les Devs n'ont plus à s'inquiéter d'un oubli de FOR SHARE/UPDATE/Locks,
  - SERIALIZABLE gère automatiquement la détection des conflits de concurrence,
  - Mais les Devs restent responsables des deadlocks.
- Des échecs sont attendus (40001, serialization_failure), on recommence…
- Avez-vous déjà utilisé ce niveau d'isolation ?

pgbench

Ne prenez pas ces mesures trop au sérieux…

même si elles montrent un point fondamental.

READ COMMITTED vs SERIALIZABLE

pgbench -p 5440 -d ibiza -T 30 -c 4
pgbench (17.5 (Ubuntu 17.5-1.pgdg20.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 4
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 42780
number of failed transactions: 0 (**0.000**%)
latency average = 2.803 ms
initial connection time = 23.635 ms
tps = **1426.920935** (without initial connection time)

pgbench -p 5440 -d ibiza -T 30 -c 4
pgbench (17.5 (Ubuntu 17.5-1.pgdg20.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 4
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 13692
number of failed transactions: 40702 (**74.828**%)
latency average = 2.205 ms (including failures)
initial connection time = 19.526 ms
tps = **456.654950** (without initial connection time)

READ COMMITTED vs SERIALIZABLE (la revanche)

pgbench -p 5440 -d ibiza -T 30 -c 4 **-N**
pgbench (17.5 (Ubuntu 17.5-1.pgdg20.04+1))
starting vacuum...end.
transaction type: <builtin: **simple update**>
scaling factor: 10
query mode: simple
number of clients: 4
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 57731
number of failed transactions: **0 (0.000%)**
latency average = 2.077 ms
initial connection time = 22.274 ms
tps = **1925.657178** (without initial connection time)

pgbench -p 5440 -d ibiza -T 30 -c 4 **-N**
pgbench (17.5 (Ubuntu 17.5-1.pgdg20.04+1))
starting vacuum...end.
transaction type: <builtin: **simple update**>
scaling factor: 10
query mode: simple
number of clients: 4
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 57769
number of failed transactions: **0 (0.000%)**
latency average = 2.076 ms
initial connection time = 20.025 ms
tps = **1926.565213** (without initial connection time)

Merci, des questions ?

frederic.delacourt@data-bene.io