SGBD : Bases de données avancées [M3106C]

Hocine ABIR

10 septembre 2014

IUT Villetaneuse
E-mail: abir@iutv.univ-paris13.fr

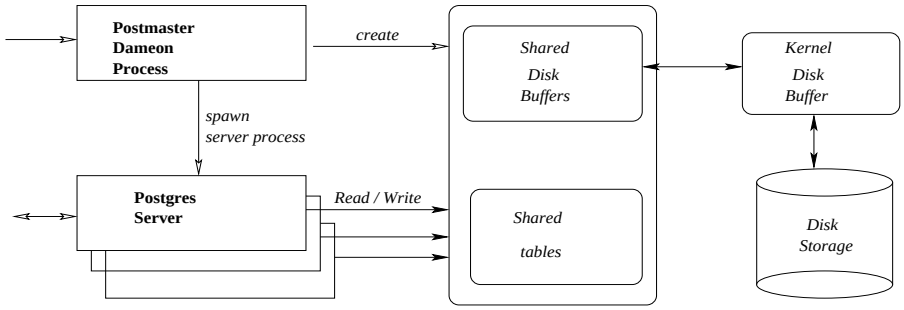
TABLE DES MATIÈRES

CHAPITRE

QUATRE

Gestion des Accès Concurrents sous PostgreSQL

4.1 Introduction




```
$ ps auxww
postgres  1862  0.0  1.2  20172 3084 ?  S  07 :20 0 :00  /usr/bin/postmaster -p 5432 -D /var/lib/pgsql/data
postgres  1864  0.0  0.2  9960   560  ?  S  07 :20 0 :00  postgres : logger process
postgres  1866  0.0  0.4  20308 1056 ?  S  07 :20 0 :00  postgres : writer process
postgres  1867  0.0  0.2  10960 548  ?  S  07 :20 0 :00  postgres : stats buffer process
postgres  1868  0.0  0.2  10192 748  ?  S  07 :20 0 :00  postgres : stats collector process
postgres  2429  0.0  1.5  21084 4004 ?  S  07 :21 0 :00  postgres : abir abir [local] idle in transaction
postgres  6110  0.0  0.9  20848 2464 ?  S  08 :43 0 :00  postgres : abir abir [local] DELETE waiting
```

4.2 Transaction

4.2.1 Structure d'une transaction

Une transaction peut être :

Simple :

c'est à dire constituée d'une seule requête `SQL`, comme dans l'exemple suivant :

```
-------------------------------------------------------------
# DELETE FROM fournisseur    -- Transaction 1 : Validée
#      WHERE fo_numero='F5';
DELETE 1
-------------------------------------------------------------
# DELETE FROM fournisseur    -- Transaction 2 : Avortée
#      WHERE fo_numero='F2';
ERROR:  update or delete on "fournisseur" violates \
   foreign key constraint "commande_co_fournisseur_fk" \
   on "commande"
DETAIL:  Key (fo_numero)=(F2) is still referenced \
   from table "commande".
-------------------------------------------------------------
# SELECT * FROM fournisseur;  -- Transaction 3 : Validée
 fo_numero | fo_nom | fo_categorie | fo_ville
-----------+--------+--------------+----------
 F2        | Dupont |           20 | Paris
 F3        | Dubois |           20 | Paris
 F4        | Durant |           10 | Londres
 F1        | Martin |           10 | Londres
(4 rows)
-------------------------------------------------------------
```

Composée :

constituée d'une ou plusieurs requêtes SQL contenues dans un bloc `BEGIN TRANSACTION` et `END TRANSACTION` :

```
BEGIN TRANSACTION;
  <suite_de_requêtes>
END TRANSACTION;
```

Comme dans l'exemple suivant :

```
-------------------------------------------------------------
# BEGIN TRANSACTION;       -- Transaction 1 : Validée
BEGIN
# DELETE FROM fournisseur
#      WHERE fo_numero='F1';
DELETE 1

# select * from fournisseur;
 fo_numero | fo_nom | fo_categorie | fo_ville
-----------+--------+--------------+----------
 F2        | Dupont |           20 | Paris
 F3        | Dubois |           20 | Paris
 F4        | Durant |           10 | Londres
(3 rows)

# end transaction;
COMMIT
```

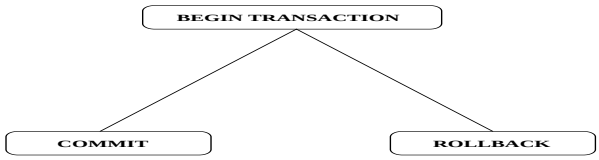
```
-----------------------------------------------------------
# BEGIN TRANSACTION;       -- Transaction 2 : Avortée
BEGIN
# DELETE FROM fournisseur
abir@[local]<-abir-*#      WHERE fo_numero='F2';
ERROR:  update or delete on "fournisseur" violates \
   foreign key constraint "commande_co_fournisseur_fk" \
   on "commande"
DETAIL:  Key (fo_numero)=(F2) is still referenced from \
        table "commande".

# SELECT * FROM fournisseur;
ERROR:  current transaction is aborted, commands \
    ignored until end of transaction block

# end transaction;
ROLLBACK
-----------------------------------------------------------
```

4.2.2 Définition

Une transaction est une unité :
- logique : la plus petit unité que l'on peut exécuter,
- atomique : les requêtes qui la constituent sont indécomposables.
  Elle se termine de deux façons :
  1. `COMMIT` : les mises à jour des requêtes qui la composent sont validées dans la base de données. On dit que la transaction est validée.
  2. `ROLLBACK` : les mises à jour des requêtes qui la composent, sont annulées dans la base de données. On dit que la transaction est défaite ou avortée.



Une transaction est défaite si durant son exécution, une des requêtes qui la composent, engendre :

Une erreur :

```
-------------------------------------------------
# BEGIN TRANSACTION;
BEGIN
#    INSERT INTO Commande
#       VALUES ('F2','P4',25);
INSERT 203132 1

#    INSERT INTO Commande
       VALUES ('F2',P8,25);
ERROR:  column "p8" does not exist

# END TRANSACTION;
ROLLBACK



# SELECT * FROM Commande;
 co_fournisseur | co_piece | co_quantite
----------------+----------+-------------
 F2             | P3       |          10
 F3             | P3       |          50
 F3             | P2       |          10
(3 rows)
-------------------------------------------------
```

Une violation de contrainte d'intégrité :

```
-------------------------------------------------
# BEGIN TRANSACTION;
BEGIN
# INSERT INTO Commande
       VALUES ('F4','P3',22);
INSERT 0 1

# INSERT INTO Commande
       VALUES ('F2','P8',25);
ERROR:  insert or update on table "commande" \
     violates foreign key constraint \
     "commande_co_piece_fk"
DETAIL:  Key (co_piece)=(P8) is not present \
         in table "piece".

# END TRANSACTION;
ROLLBACK
```

```
# SELECT * FROM commande;
 co_fournisseur | co_piece | co_quantite
----------------+----------+-------------
 F2             | P3       |          10
 F3             | P3       |          50
 F3             | P2       |          10
(3 rows)
-------------------------------------------------
```

Un conflit d'accès :

transaction non sérialisable, blocage mutuel.

```
-------------------------------------------------
# BEGIN TRANSACTION;
BEGIN
# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
# SELECT * FROM fournisseur;
 fo_numero | fo_nom | fo_categorie | fo_ville
-----------+--------+--------------+----------
 F2        | Dupont |           20 | Paris
 F3        | Dubois |           20 | Paris
 F4        | Durant |           10 | Londres
 F1        | Martin |           10 | Londres
(4 rows)

# DELETE FROM fournisseur WHERE fo_numero='F1';
ERROR:  could not serialize access due to concurrent update
# END TRANSACTION;
ROLLBACK
-------------------------------------------------
```

```
-------------------------------------------------
# BEGIN TRANSACTION;
BEGIN
# DELETE FROM commande where co_fournisseur='F3';
DELETE 2

# DELETE FROM commande where co_fournisseur='F2';
ERROR:  deadlock detected
DETAIL:  Process 1410 waits for ShareLock on transaction 71818;\
             blocked by process 1858.
          Process 1858 waits for ShareLock on transaction 71819; \
             blocked by process 1410.

# END TRANSACTION;
ROLLBACK
-------------------------------------------------
```

4.2.3 Test ACID

Une transaction est :

A- `Atomique` : terminée normalement dans son intégralité c'est à dire tout ou rien,

C- `Consistante` : préserve les contraintes d'intégrité,

I- `Isolée` : non affectée par le comportement des autres transactions concurrentes,

D- `Durable` : les mises à jour effectuées sont persistentes.

4.2.4 Gestion d'une transaction

Gestion Automatique : `AUTOCOMMIT`

Le standard ISO SQL définit un modèle particulier de transaction : qui dit que toute connection à une base de données se fait dans une transaction. Chaque transaction se termine alors avec un `COMMIT` ou `ROLLBACK` et une nouvelle transation commence à la première requête suivant le `COMMIT` ou `ROLLBACK` (`Oracle`).

D'autres systèmes (`PostgreSQL`) définissent un mécanisme d'auto-commit qui agit comme si après chaque requête, un `COMMIT` ou `ROLLBACK` est exécuté à chaque fois.

```
----------------------------------------
# show autocommit;
 autocommit
------------
 on
(1 row)
# set autocommit to off;
ERROR:  SET AUTOCOMMIT TO OFF is no longer supported
----------------------------------------
```

Annulation d'une Transaction : `ROLLBACK`

L'annulation d'une transaction consiste à défaire (ou annuler) les modifications effectuées par les requêtes qui composent la transaction. C'est l'action effectuée par défaut par `END TRANSACTION` quand la transaction est avortée.

```
--------------------------------------------------
# BEGIN TRANSACTION;
BEGIN

# INSERT INTO commande
          VALUES ('F4','P5',50);
INSERT 0 1

# ROLLBACK;
ROLLBACK
```

```
# SELECT * FROM commande;
 co_fournisseur | co_piece | co_quantite
----------------+----------+-------------
 F2             | P3       |          10
 F3             | P3       |          50
 F3             | P2       |          10
(3 rows)
-----------------------------------------------------
```

Validation d'une Transaction : `COMMIT`

La validation d'une transaction consiste à rendre les modifications effectuées par la transaction permanentes (ou persistentes), en les intégrant dans la base de données. C'est l'action effectuée par défaut par `END TRANSACTION` quand la transaction n'est pas avortée.

```
-----------------------------------------------------
# BEGIN TRANSACTION;
BEGIN

# INSERT INTO commande
          VALUES ('F4','P5',50);
INSERT 0 1

# COMMIT;
COMMIT

# SELECT * FROM commande;
 co_fournisseur | co_piece | co_quantite
----------------+----------+-------------
 F2             | P3       |          10
 F3             | P3       |          50
 F3             | P2       |          10
 F4             | P5       |          50
(4 rows)
-----------------------------------------------------
```

4.3 "Transaction Isolation Mode"

- Le mode d'isolation d'une transaction décrit le comportement de la transaction dans un ordonnancement de transactions auquel elle participe.
- Le mode serialisable garantit que les transactions sont exécutées de façons equivalentes à un ordonnacement série : isolation complète ou parfaite.
- Le mode sérialisable entraine la chute des performances : plusieurs niveaux d'isolation adaptés aux problèmes des accès concurrents.

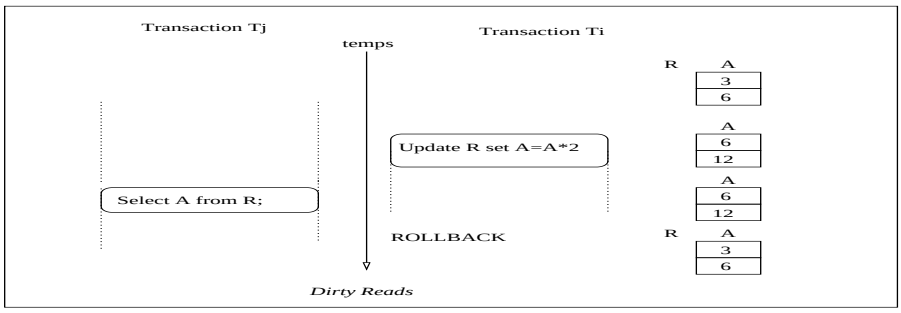
4.3.1 Classification des Problèmes d'accès

Pour améliorer les performances et avoir des niveaux d'isolation graduels et adaptés à différents contextes , la norme ANSI.ISO Sql standard (SQL92) a défini quatre (4) niveaux par rapport à trois (3) problèmes d'accès. Ces problèmes appelés phénomènes sont définis par rapport au concept classique de sérialisation et d'ordonnancements non-acceptables.

1. "reading uncommitted data" ou "dirty read"
   $$< WRITE^i(X), READ^j(X), (ROLLBACK^i | COMMIT^j) >$$
2. non-repeatable (fuzzy) read
   $$< READ^i(X), WRITE^j(X), COMMIT^j, READ^i(X), COMMIT^i >$$
3. phantom read
   $$< READ^i(C), WRITE^j(XinC), COMMIT^j, READ^i(C), COMMIT^i >$$

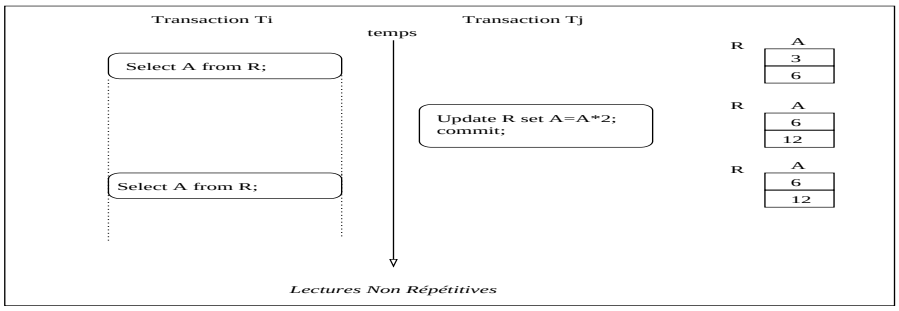
Dirty read

Une transaction T^i modifie une donnée A. Une autre transaction T^j lit la donnée A avant que T^i se termine (commit, rollback). Si T^i effectue un ROLLBACK alors T^j a lu une donnée A inconsistente.



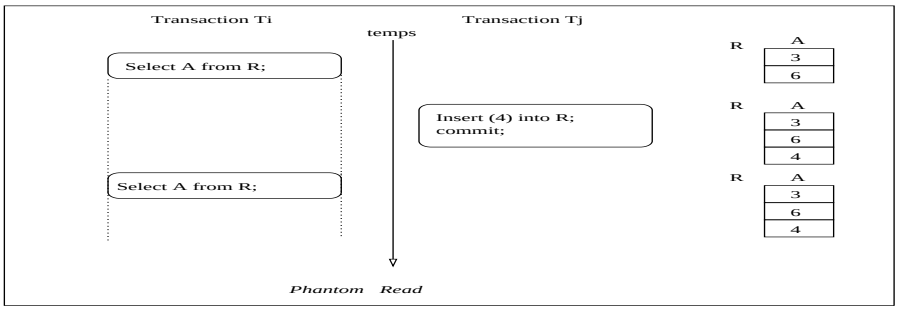
Fuzzy ou Non-reproductible read

Dans une transaction T^i, deux requêtes identiques (ou qui lisent un même ensemble de données A) lisent des ensembles de données différents : l'ensemble des données lu par ces transactions ont été modifiés par une autre transaction T^j terminée (committed).



Phantom reads

Dans une transaction, deux requêtes identiques (ou qui lisent un même ensemble de données) lisent des ensembles de données différents : l'ensemble des données lu par ces transactions ont été modifiés par d'autres transactions committed.



Les 4 niveaux de la norme ANSI.ISO Sql standard (SQL92) sont définis par rapport à ces classes d'ordonnancement comme décrit dans la table suivante :

Level	*Isolation Level*	*Dirty R*	*Non-repeatable R*	*Phantom R*
1	READ UNCOMMITTED	possible	possible	possible
2	READ COMMITTED	IMPOSSIBLE	possible	possible
3	REPEATABLE READ	IMPOSSIBLE	IMPOSSIBLE	possible
4	SERIALIZABLE	IMPOSSIBLE	IMPOSSIBLE	IMPOSSIBLE

4.3.2 Niveaux d'isolation sous PostgreSQL

1. Read Committed [Niveau 2] (niveau par défaut)
   Chaque requête R exécutée par une transaction T peut accéder aux données mises à jour par une transaction terminée (committed) avant le début de la **r**equête R (pas de la transaction T) :
   la requête n'accède jamais aux données mises à jour par une transaction non terminée (uncommitted) [pas de dirty reads].
2. Serializable [Niveau 4] Chaque requête exécutée par une transaction ne peut accéder qu'aux données mises à jour par une transaction terminée (committed) avant le début de la **t**ransaction.

4.3.3 Choix du Niveau d'Isolation

```
-----------------------------------------------------
  SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-----------------------------------------------------
```

4.3.4 Niveau READ COMMITTED vs SERIALIZABLE

Problème

On considère la transaction T^A suivante :

```
-----------------------------------------------------
  UPDATE piece SET pi_poids=pi_poids + 1
        WHERE pi_numero='P1';
-----------------------------------------------------
```

qui augmente le poids de pièce 'P1' de 1. Supposons qu'une transaction T^B effectue la même opération (i.e. le même tuple) avant le COMMIT de T^A.

- T^B doit attendre que T^A se termine ? Si T^B utilise l'ancienne valeur de pi_poids alors ce n'est pas acceptable car l'opération T^A est perdue !
- Si T^A se termine anormalement (ROLLBACK) alors T^B peut poursuivre avec la valeur pré-existante de pi_poids.

Transaction T^A	Transaction T^B
`# BEGIN TRANSACTION;` `BEGIN`	

`# SELECT * FROM piece`
`#  WHERE pi_numero='P1';`
` pi_numero | pi_poids`
`----------+----------`
` P1  |  12.60`
`(1 row)`	

`# UPDATE piece`
`#  SET pi_poids=pi_poids + 1`
`#  WHERE pi_numero='P1';`
`UPDATE 1`	

	`# BEGIN TRANSACTION;` `BEGIN`

	`# SELECT * FROM piece`
	`#  WHERE pi_numero='P1';`
	` pi_numero | pi_poids`
	`----------+----------`
	` P1  |  12.60`
	`(1 row)`

	`# UPDATE piece`
	`#  SET pi_poids=pi_poids + 1`
	`#  WHERE pi_numero='P1';`

`# ROLLBACK;<-` `ROLLBACK`	 `UPDATE 1`

	`# SELECT * FROM piece`
	`#  WHERE pi_numero='P1';`
	` pi_numero | pi_poids`
	`----------+----------`
	` P1  |  13.60`
	`(1 row)`

- Si T^A se termine normalement , Que se passe-t-il ?
  - Si T^B utilise l'ancienne valeur de pi_poids alors ce n'est pas acceptable car l'opération T^A est perdue !
  - Si T^B utilise la nouvelle valeur de pi_poids alors T^B aura lu des données de transactions terminées (COMMIT) pendant son exécution : T^B viole la règle d'isolation des transactions

Solution

Deux modes d'isolations :

1. `READ COMMITTED` : La transaction T^B va poursuivre en lisant la nouvelle valeur de pi_poids.

Transaction T^A	Transaction T^B
# BEGIN TRANSACTION; BEGIN	
# SELECT pi_numero,pi_poids # FROM piece # WHERE pi_numero='P1'; pi_numero | pi_poids -----------+---------- P1 | 12.60 <- (1 row)	
# UPDATE piece # SET pi_poids=pi_poids + 1 # WHERE pi_numero='P1'; UPDATE 1	
	# BEGIN TRANSACTION; BEGIN
	# show transaction # isolation level; transaction_isolation ----------------------- read committed <- (1 row)
	# SELECT pi_numero,pi_poids # FROM piece # WHERE pi_numero='P1'; pi_numero | pi_poids -----------+---------- P1 | 12.60 <- (1 row)
	# UPDATE piece # SET pi_poids=pi_poids + 1 # WHERE pi_numero='P1';
# COMMIT;<- COMMIT	 UPDATE 1
	# SELECT pi_numero,pi_poids # FROM piece # WHERE pi_numero='P1'; pi_numero | pi_poids -----------+---------- P1 | 14.60 <- (1 row)

2. `SERIALIZABLE` : La transaction T^B va être avortée (ROLLBACK).

Transaction T^A	Transaction T^B
# BEGIN TRANSACTION; BEGIN	
# SELECT pi_numero,pi_poids # FROM piece # WHERE pi_numero='P1'; pi_numero | pi_poids ----------+---------- P1 | 12.60<- (1 row)	
# UPDATE piece # SET pi_poids=pi_poids + 1 # WHERE pi_numero='P1'; UPDATE 1	
	# BEGIN TRANSACTION; BEGIN
	# set transaction isolation # level serializable; SET
	# show transaction # isolation level; transaction_isolation ---------- serializable <- (1 row)
	# SELECT pi_numero,pi_poids # FROM piece # WHERE pi_numero='P1'; pi_numero | pi_poids ----------+---------- P1 | 12.60 <- (1 row)
	# UPDATE piece # SET pi_poids=pi_poids + 1 # WHERE pi_numero='P1';
# COMMIT; COMMIT <-	ERROR: could not serialize access due to concurrent update <-

4.3.5 Ordonnancement SERIALISABLE vs Ordonnancement SERIE

Transaction T^A	Transaction T^B
# BEGIN TRANSACTION; BEGIN	
# set transaction isolation # level serializable; SET	
# INSERT INTO fournisseur # VALUES ('F1');<- INSERT 0 1	
# SELECT fo_numero # FROM fournisseur fo_numero ----------- F2 F3 F4 F1<- (4 rows)<-	
	# BEGIN TRANSACTION; BEGIN
	# set transaction isolation # level serializable; SET
	# INSERT INTO fournisseur # VALUES ('F5'); <- INSERT 0 1
	# SELECT fo_numero # FROM fournisseur fo_numero ----------- F2 F3 F4 F5 <- (4 rows) <-

Un ordonnancement SERIALISABLE au sens PostreSQL n'est pas équivalent à un ordonnancement série !

4.4 Vue consistente des données

4.4.1 Définition

Une vue consistente des données est constituées des données mises à jour par des transactions terminées normalement (committed) par rapport à un **point temporel** : *le début de la transaction qui veut voir ces données.*

La consistence des lectures est assurée à deux niveaux :

1. Au niveau Requête :

La vue des données d'une **requête** est consistente par rapport au début d'exécution de la requête c'est à dire que la requête n'accèdent pas aux mises à jour effectuées par des transactions terminées durant l'exécution de la requête.

2. Au niveau Transaction :
   La vue des données d'une **transaction** est consistente par rapport au début d'exécution de la transaction c'est à dire que la reproduction des lectures est garantie.

4.4.2 Mise en oeuvre

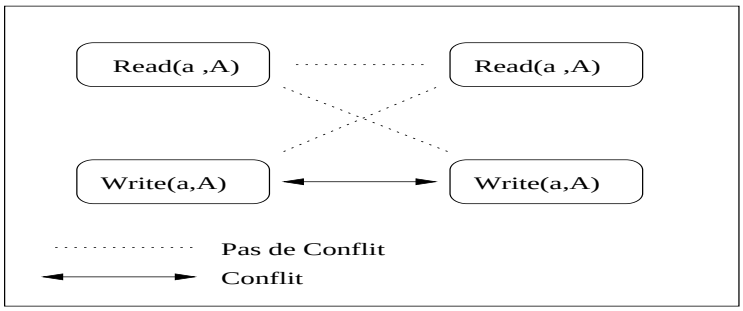
Sous `PostgreSQL`, la consistence des données vues par une transaction est assurée par :

1. un mécanisme de verrouillage à deux phases ,
2. un mécanisme de Muti-Version.

4.5 Mécanisme de verrouillage

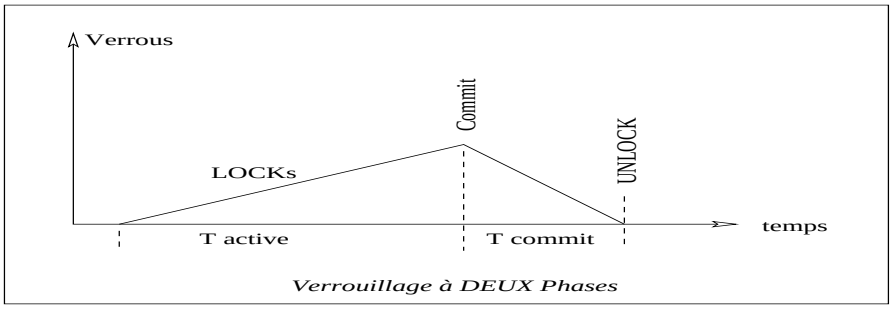
4.5.1 Degrès de Restriction de plus bas niveau possible

Ce qui permet d'avoir un degrès de concurrence élevé : seuls les `WRITE` sont en **conflits** (ou **bloquants**).
- Les lecteurs d'un tuple n'attendent pas les écrivains du même tuple,
- Les écrivains d'un tuple n'attendent pas les lecteurs du même tuple,
- Les écrivains d'un tuple attendent d'autres écrivains du même tuple,



4.5.2 Verrouillage à deux phases

Le mécanisme de verrouillage utilisé est un mécanisme à deux phases : c'est à dire qu'une transaction T^i ne peut voir que les données mises à jour par une transaction T^j terminée (committed) avant le début de T^i



Verrouillage à DEUX Phases

Verrous Niveau-Table

Une table peut être verrouillée automatiquement en fonction du type de requête suivant plusieurs modes. Ces verrous sont nécessaires pour prévenir certaines opérations destructives.

	Verrou	commandes	En conflit avec
1	ACCESS SHARE	SELECT, ANALYZE	8
2	ROW SHARE	SELECT FOR UPDATE, SELECT FOR SHARE	7,8
3	ROW EXCLUSIVE	UPDATE, DELETE, INSERT 1 sur les autres tables referenciees	5,6,7,8
4	SHARE UPDATE EXCLUSIVE	VACUUM	4,5,6,7,8
5	SHARE	CREATE INDEX	3,4,6,7,8
6	SHARE ROW EXCLUSIVE		3,4,5,6,7,8
7	EXCLUSIVE		2,3,4,5,6,7,8
8	ACCESS EXCLUSIVE	ALTER TABLE, DROP TABLE, REINDEX, CLUSTER, VACUUM FULL	1,2,3,4,5,6,7,8

Verrous Niveau-Tuple

	Verrou	commandes	En conflit avec
1	SHARED	SELECT FOR SHARE	2
2	EXCLUSIVE	INSERT, DELETE, UPDATE, SELECT FOR UPDATE,	2

Exemple

```
-------------------------------------------------
# BEGIN TRANSACTION;
BEGIN
# SELECT * FROM piece WHERE pi_numero='P1';
 pi_numero | pi_nom | pi_poids | pi_couleur
-----------+--------+----------+------------
 P1        | Vis6   |    12.60 | rouge
(1 row)

# SELECT relname,transaction,mode,granted
  FROM pg_class c,pg_locks
  WHERE relation=c.oid AND relname='piece';
 relname | transaction |      mode       | granted
---------+-------------+-----------------+---------
 piece   |       25237 | AccessShareLock | t
(1 row)
# END TRANSACTION;
COMMIT
-------------------------------------------------
```

4.5.3 Verroouillage manuel

Verrou	Commande
Access Share	LOCK TABLE <table> IN ACCESS SHARE MODE
Row Share	LOCK TABLE <table> IN ROW SHARE MODE
Row Exclusive	LOCK TABLE <table> IN ROW EXCLUSIVE MODE
Share	LOCK TABLE <table> IN SHARE MODE
Share Row Exclusive	LOCK TABLE <table> IN SHARE ROW EXCLUSIVE MODE
Exclusive	LOCK TABLE <table> IN EXCLUSIVE MODE
Access Exclusive	LOCK TABLE <table> IN ACCESS EXCLUSIVE MODE

Exemple

Transaction T^A	Transaction T^B
`# BEGIN TRANSACTION;` `BEGIN`	

`# show transaction`
`#  isolation level;`
`transaction_isolation`
`----------`
`read committed <-`
`(1 row)`	

`# LOCK TABLE fournisseur`
`#  IN SHARE  ROW EXCLUSIVE MODE;`
`LOCK TABLE<-`	

	`SELECT ....` `SELECT ...  FOR UPDATE`

`# SELECT fo_numero`
`#  FROM fournisseur`
`fo_numero`
`----------`
`F2`
`F3`
`F4`
`F1`
`(4 rows)`	

4.5.4 Compatibilté des verrous

La table ci-dessous donne la compatibilité des verrous Niveau-Table :

Verrous	Modes Compatibles						
	AS	RS	RX	S	SRX	X	AX
AS	OUI	OUI	OUI	OUI	OUI	OUI	NON
RS	OUI	OUI	OUI	OUI	OUI	NON	NON
RX	OUI	OUI	OUI	NON	NON	NON	NON
S	OUI	OUI	NON	OUI	NON	NON	NON
SRX	OUI	OUI	NON	NON	NON	NON	NON
X	OUI	NON	NON	NON	NON	NON	NON
AX	NON	NON	NON	NON	NON	NON	NON

4.6 Modèle de Consistence Multi-version

4.6.1 Définition

le terme Muti-Version désigne le fait de fournir pour chaque requête (ou transaction) sa propre version de données consistantes par rapport au début d'exécution de la requête (ou transaction).

4.6.2 Principe

- Ne pas modifier les données de la base (de données) mais créer de nouvelles (versions de) données.
  Chaque tuple contient 4 champs additionnels :

Champs	Description
`OID`	Identificateur unique de l'enregistrement
`Xmin`	Identificateur de la Transaction d'Insertion
`Tmin`	Heure du commit de Xmin ou A partir de quand l'enregistrement est valide
`Cmin`	Identifiant de la commande d'Insertion
`Xmax`	Identificateur de la Transaction de Suppression
`Tmax`	Heure du commit de Xmax ou A partir de quand l'enregistrement est invalide
`Cmax`	Identifiant de la commande de Suppression
`PTR`	liste des enregistrements delta

```
# select xmin,cmin,xmax,cmax
# ,pi_numero from piece;
----------------------------------------

 xmin  | cmin | xmax | cmax | pi_numero
-------+------+------+------+-----------
 25179 |    0 |    0 |    0 | P1
 25180 |    0 |    0 |    0 | P2
 25181 |    0 |    0 |    0 | P3
 25182 |    0 |    0 |    0 | P4
(4 rows)
----------------------------------------
```

- Insertion d'un tuple
  Quand un tuple est inserré par la commande SQL `INSERT INTO`, un nouvel enregistrement est ajouté au fichier de la relation avec :
  – Un nouveau identifiant `OID` ,
  – Et les champs `Xmin` et `Cmin` recoivent l'identifiant de la transaction courante d'insertion,
  – Les autres champs additionnels restent `NULL`.

- Suppression d'un tuple
  Quand un tuple est supprimé par la commande SQL `DELETE FROM`, l'enregistrement qualifié par la clause `WHERE` est recherché dans le fichier de la relation et les champs additionnels sont modifiés comme suit :
  – les champs `Xmax` et `Cmax` recoivent l'identifiant de la transaction courante de suppression : l'enregistrement devient invalide,
  – Les autres champs additionnels restent inchangés.

- Mise à jour d'un tuple
  Quand un tuple est mis à jour par la commande SQL `UPDATE`, deux opérations sont alors effectuées :

  1. Suppression :
     L'enregistrement qualifié par la clause `WHERE` est recherché dans le fichier de la relation et les champs `Xmax` et `Cmax` recoivent l'identifiant de la transaction de mise à jour. : l'enregistrement devient invalide.

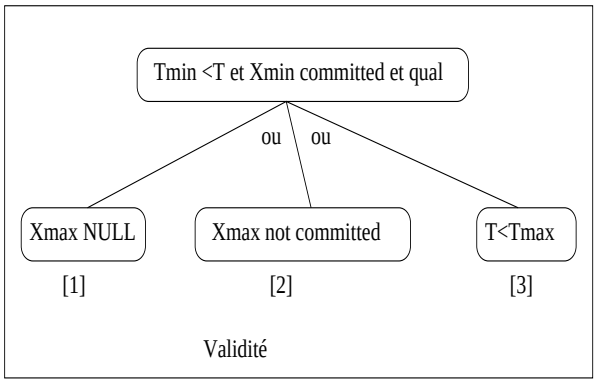
2. Insertion :
   Un nouvel enregistrement avec les mises à jour est inséré dans le fichier avec :
   – Un même identifiant `OID` que celui de l'enregsitrement supprimé,
   – Et les champs `Xmin` et `Cmin` recoivent l'identifiant de la transaction courante de suppression,
   – Les autres champs additionnels restent `NULL`.

4.6.3 Visibilité : "Snapshots"

A chaque accès à un enregistrement, le système doit vérifier la validité de cet enregistrement à l'instant correspondant à l'accès.

Les champs $Tmin$ et $Tmax$ sont estampillés au moment du commit d'une transaction c'est à dire de façon asynchrone.

La recherche d'un tuple ayant une qualification *qual* à un instant T est déterminé par le test suivant :



Ce test exprime le fait que seul les enregistrements créés par des transactions terminées normalement à l'instant T sont accessibles et ils doivent de plus avoir soit :

[1] $Xmax = NULL$ (ou $Tmax = NULL$) : aucune mise à jour n'est faite ou est en train de se faire sur cet enregistrement, ou

[2] il y a une transaction en cours qui met à jour cet enregistrement donc la version actuel est valide, ou

[3] il y a une transaction qui a modifié cet enregistrement ($Xmax$ committed), mais elle s'est terminé après la transaction en cours de consultation.

4.7 Multiversion avec 2PL

4.7.1 Lecture

Une transaction (ou commande) de lecture reçoit un estampille déterminée à partir de la valeur courante de *ts_counter* au moment de son initialisation.

4.7.2 Mise à jour

Une transaction (ou commande) de mise à jour acquière deux verrous : read, write qu'elle conserve jusqu'au commit :

- chaque écriture terminée avec succès entraine la création d'une nouvelle version de la donnée écrite,
- chaque version de donnée a un estampille $Xmin$ dont la valeur est obtenu d'un compteur $ts\_counter$ au moment du commit.

4.7.3 Production d'ordonnancement sérializable

Lecture

1. Une commande de Lecture d'un tuple :
   - obtient un verrou `AS` sur ce tuple,
   - incrément le compteur $ts\_counter$: $ts\_counter++$
   - estampille de la transaction est $ts\_counter$.
   - le protocole de lecture suit le protocole d'estampillage : rechercher une version tr du tuple t telque :
     $$Tmin^{tr} = max(Tmin^t / Tmin^t < estampille\_transaction)$$
2. Au moment du commit de la transaction :
   - lève le verrou `AS`

Mise à jour

1. Une transaction de Maj d'un tuple :
   - obtient un verrou `RS` sur ce tuple,
   - lit la dernière version du tuple,
   - obtient un verrou `RX` sur ce tuple,
   - crée une nouvelle version du tuple avec $Tmin = \infty$
2. Au moment du commit de la transaction :
   - incrément le compteur $ts\_counter$: $ts\_counter++$
   - valide la maj : $Tmin = ts\_counter$
   - lève le verrou `RS` et `RX`