UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Chapitre 1: Manipuler les données

Contrôler les transactions

Université de la Nouvelle-Calédonie
frederic.flouvat@univ-nc.nc

Quelques références bibliographiques

- Documentation PostgreSQL, Chapitre 13. "Contrôle d'accès simultané"
- Livre "Database Management Systems", Raghu Ramakrishnan et Johannes Gehrke.
- "Database Systems: The Complete Book", Pr. Jeffrey Ullman, Stanford University
- "PostgreSQL Concurrency with MVCC", Heroku dev center
- "Transaction Processing in PostgreSQL" de Tom Lane, Great Bridge
- "Bases de données: Aspects système", Philippe Rigaux, Université Paris-Dauphine
- "Bases de données avancées", Jacques Le Maitre, Université du Sud Toulon-Var
- "Conception des bases de données", Stéphane Crozat, UTC

Rappel

- Une bases de données est un ensemble de données
  - structurées
  - persistantes
  - **cohérentes**

- Cohérence garantie par des contraintes d'intégrités (contraintes de domaine, intégrité référentielles, dépendances fonctionnelles …)
  - cf "Contraindre les données", chapitre 2

- Mais cohérence peut être remise en cause par:
  - Défaillances du système
  - **Accès concurrents**

Pourquoi des accès concurrents ?

- Les systèmes de gestion de bases de données sont généralement utilisés par plusieurs utilisateurs et/ou processus à la fois.
  - Interrogations et modifications.

- Toutes ces opérations arrivent dans un certain ordre au SGBD qui les met en file d'attente pour traitement (***"Scheduler"***)

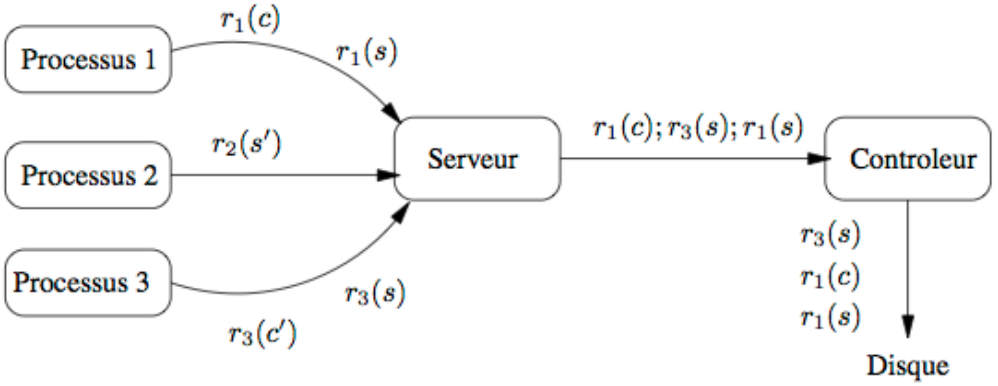


schéma issu du cours de Philippe Rigaux, Université Paris-Dauphine

- Exemple:
  - Deux personnes titulaires d'un compte commun retirent au même moment $100 dans deux distributeurs de billets différents.
  - ➢ Le SGBD doit alors s'assurer que les deux retraits sont appliqués correctement au compte sous-jacent.

Exemple: Transfert entre 2 comptes

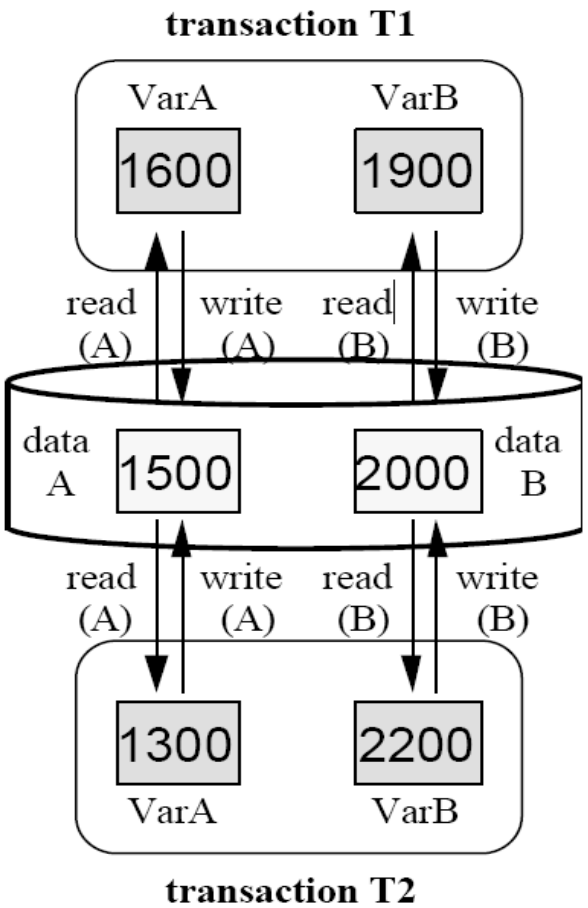
```
Transfert(A,B,M)
Begin
        varA := Read(A)
        varA := varA + M
        Write(A,varA)
        varB := Read(B)
        varB := varB - M
        Write(B,varB)
Commit
```

- A= 1500 et B=2000
- Le guichet 1 exécute T1 :
  - Transfert(A,B,100)
- Le guichet 2 exécute T2 :
  - Transfert(A,B,-200)

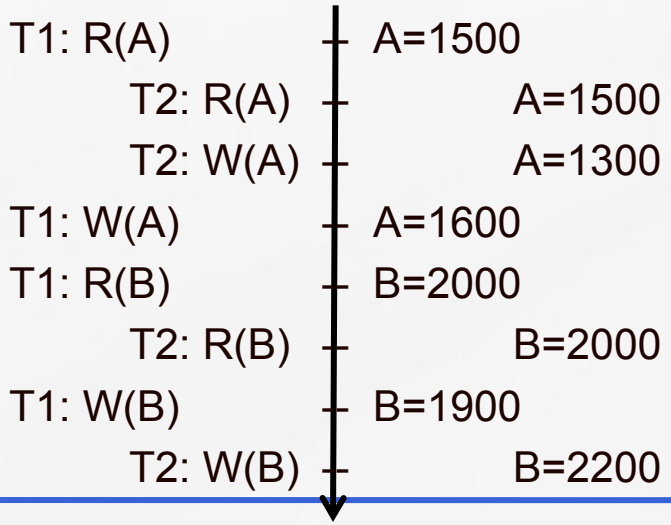
- Si exécution à la suite (i.e. en série), pas de problème

Exemple: Transfert entre 2 comptes

```
Transfert(A,B,M)
Begin
        varA := Read(A)
        varA := varA + M
        Write(A,varA)
        varB := Read(B)
        varB := varB - M
        Write(B,varB)
Commit
```



- Si exécution "entrelacée", problèmes possibles



➢ A= 1600 et B= 2200 !!!

Les problèmes liés aux accès concurrents

- Perte de mise à jour

T1	T2	BD A=10
Read(A)		
	Read(A)	
A=A+10		
Write(A)		A = 20
	A=A+50	
	Write(A)	A = 60

- Lectures incohérentes

T1	T2	BD A=120 B=80
Read(A)		
A=A-50		
Write(A)		A = 70
	Read(A)	
	Read(B)	
	display A+B (affiche 150)	
Read(B)		
B=B+50		
Write(B)		B = 130

Les problèmes liés aux accès concurrents

- Lectures non reproductibles

T1	T2	BD A=10
	Read(A) (lit 10)	
A=20		
Write(A)		A = 20
	Read(A) (lit 20)	

- Objets fantômes
  - la 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.

T1	T2	BD Clients ={}
SELECT COUNT(name) FROM "Clients"; (résultat 0)		
	INSERT INTO "Clients"(name) VALUES ('Bob');	Clients = {<'Bob', …>}
SELECT COUNT(name) FROM "Clients"; (résultat 1)		

Les transactions en Base de Données

- ***Transaction*** = ensemble d'instructions dont le résultat sera visible au reste du système d'un seul bloc si la transaction est validée (*commit*), ou pas du tout visible si elle est annulée (*abort*)

- Exemples: réservation d'une place d'avion, passer une commande sur un catalogue en ligne, …

- Hypothèses:
  - Les transactions interagissent entre elles uniquement via des **lectures** et des **écritures** dans la base de données.
    - pas de messages échangés
  - Une base de données est une collection **fixe** d'objets **indépendants.**

Transactions et SQL

- Définir une transaction

  Begin;

  <instructions SQL>

  COMMIT;

- L'instruction SQL **COMMIT** valide une transaction.
  - Ses modifications sont maintenant permanentes dans la base de données.
  - Attention mode *Autocommit*: valide automatiquement toutes les instructions
    - dans PostgreSQL, les requêtes "isolées" sont par défaut validées automatiquement

- L'instruction SQL **ROLLBACK** implique aussi la fin d'une transaction, mais en *l'annulant*.
  - Aucun effet sur la base de données.
  - Une erreur comme une division par zéro ou une violation de contraintes peut également provoquer un rollback, même si le programmeur ne l'a pas déclenché explicitement.

Propriétés ACID

- Cohérence globale des données garantie par un certain nombre de propriétés sur les transactions

- *Les transactions ACID* sont:
  - *Atomiques* : La transaction est appliquée intégralement ou pas du tout.
  - *Cohérentes* : Les contraintes de la base de données sont préservées et les données sont cohérentes pour l'application.
  - *Isolées* : Une transaction ne doit jamais voir les résultats intermédiaires des autres transactions (comme si elle était là seule à être exécutée).
  - *Durables* : Dès qu'une transaction est validée, le système doit assurer que ses effets ne seront jamais perdus, même en cas de panne.

Garantir Atomicité & Durabilité des transactions

- La **Durabilité** des transactions est garantie par l'instruction SQL **COMMIT**
  - car une fois la transaction validée par un commit, le SGBD garantie que ses modifications seront permanentes dans la base de données (même après une panne)

- L'**Atomicité** des transactions est garantie par l'instruction SQL **ROLLBACK**
  - car une fois la transaction annulée par un rollback, toutes ses opérations le sont aussi (tout ou rien)

➢ Rendu possible grâce à l'utilisation de journaux (fichiers)

Garantir la Cohérence des transactions

- Garantie par le système via les **contraintes d'intégrités** associées au schéma
  - Clés primaires, clés étrangères, *triggers*, …

- Garantie par le développeur **en utilisant correctement les propriétés des transactions** de façon à prendre en compte les spécificités de l'application
  - Bonne utilisation des commit/rollback, définition du bon niveau d'isolation, …
  - Exemple des réservations dans une compagnie aérienne:
    - données cohérentes si le nombre de places occupées est le même que le nombre de places réservées
    - ➢ difficile via des contraintes
    - ➢ si niveau d'isolation trop faible ou mauvaise utilisation des commit/ rollback, possibilité d'entrelacement des transactions (et donc d'incohérences)
      - comme pour le virement entre deux comptes bancaires

Garantir l'Isolation des transactions

- Possibilité en SQL de définir le niveau d'isolation d'une transaction
  - p.ex. dans PostgreSQL,
    SET TRANSACTION ISOLATION LEVEL *<level>;*
    où *<level>* =
    1. SERIALIZABLE
    2. REPEATABLE READ
    3. READ COMMITTED
    4. READ UNCOMMITTED

- Garantir l'Isolation en utilisant le mode ***SERIALIZABLE***
  - Isolation totale des transactions, i.e. les transactions s'exécutent indépendamment les unes des autres *comme si* elles étaient exécutées en série

- **Problème**: très "coûteux"

➢ Possibilité d'améliorer l'efficacité en abaissant le niveau d'isolation, mais attention à bien étudier les besoins de l'application …

Les différents niveaux d'isolation SQL

- READ UNCOMMITED = lecture de données non validées
  - La transaction peut voir toutes les données de la base de données, même si elles ont été écrites par une transaction non validées.

  ++ pas de perte de mise à jour

  −− possibilité de lectures incohérentes, de lectures non reproductibles, et objets fantômes

- READ COMMITED = lecture de données validées
  - La transaction ne peut voir que les données validées, mais pas nécessairement les mêmes données à chaque fois.

  ++ pas de perte de mise à jour, pas de lectures incohérentes

  −− possibilité de lectures non reproductibles, et objets fantômes

Les différents niveaux d'isolation SQL

- REPEATABLE READ = lecture répétée
  - La transaction ne peut voir que les données validées, mais si les données sont lues à plusieurs reprises dans la même transaction, alors toutes les informations vues la première fois sont également vues la seconde fois.
    - données vues = données première lecture + données actuellement validées
    - ➢ Attention: la seconde lecture et celles suivantes peuvent voir *plus* de tuples que ce qu'il y a réellement dans la base de données.

  ++ pas de perte de mise à jour, pas de lectures incohérentes, pas de lectures non reproductibles

  −− possibilité d'avoir des objets fantômes

- SERIALIZABLE = lecture en série
  - La transaction ne peut voir que les données qui sont validées lorsqu'elle débute

  ++ pas de perte de mise à jour, pas de lectures incohérentes, pas de lectures non reproductibles, et pas d'objets fantômes

Ces différents niveaux dans PostgreSQL

- PostgreSQL READ UNCOMMITED = PostgreSQL READ COMMITED
- PostgreSQL READ COMMITED (niveau par défaut) identique à la norme SQL
- PostgreSQL REPEATABLE READ plus stricte que dans la norme SQL
  - La transaction ne peut voir que les données validées **au début de la transaction**
    - identique à SERIALIZABLE
  - La transaction peut être mise en attente et annulée en cas d'échec de sérialisation
- PostgreSQL SERIALIZABLE identique à la norme SQL
  - Identique à REPEATABLE READ
  - Le SGBD analyse en plus si la requête est compatible avec une exécution en série et déclenche une erreur s'il détecte un problème

Remarque : Isolation sérialisable et vrai sérialisation

- Définition "mathématique" d'exécution en série*: toute paire de transactions concurrentes validée avec succès apparaîtra comme ayant été exécutée en série, l'une après l'autre -- bien que celle survenant en premier n'est pas prévisible*
  - ➢ L'ordre d'exécution réel des transactions ne change pas le résultat

- Dans PostgreSQL, isolation « serializable » (avant la version 9.1 de PostgreSQL) et « repeatable read » ne respectent pas totalement cette définition
  - Evite les comportements non désirables précédents
  - Mais le résultat peut changer en fonction de l'ordre d'exécution

Remarque : Isolation sérialisable et vrai sérialisation

- Exemple:

classe	valeur
1	10
1	20
2	100
2	200

Transaction A:
```
Begin;
SELECT SUM(valeur) into sommeA FROM ma_table WHERE classe = 1;
INSERT INTO ma_table value(2,sommeA);
COMMIT;
```

Transaction B:
```
Begin;
SELECT SUM(valeur) into sommeB FROM ma_table WHERE classe = 2;
INSERT INTO ma_table value(1,sommeB);
COMMIT;
```

- si A exécuté avant : sommeB = 330
- si B exécuté avant : sommeB = 300

➢ Pas réellement sérialisable

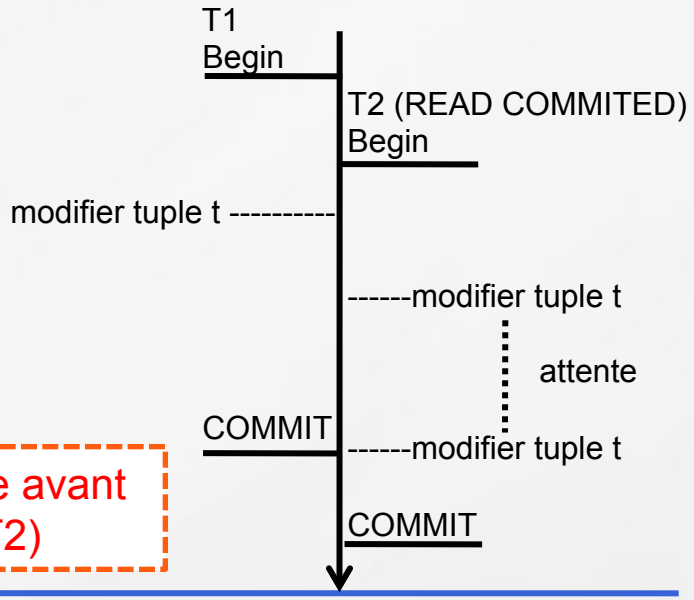
- Une autre solution pour garantir une vrai sérialisation: **système de verrous explicites des ressources**
  - mais complexe à mettre en place et très coûteux
  - PostgreSQL le permet (cf chapitre 13.3 Verrouillage explicite)

Lectures et modifications concurrentes dans PostgreSQL

- Cas des lectures : la transaction lit des données en cours d'utilisation dans d'autres transactions
  - Les lecteurs ne bloquent pas ceux qui écrivent et inversement
  - Seules les données visibles par la transaction changent en fonction de son niveau d'isolation

- Cas des modifications : la transaction essaye de modifier des données modifiées dans d'autres transactions
  - **Read commited** :

    La transaction va attendre si une autre transaction est en train de modifier un tuple qu'elle doit aussi modifier

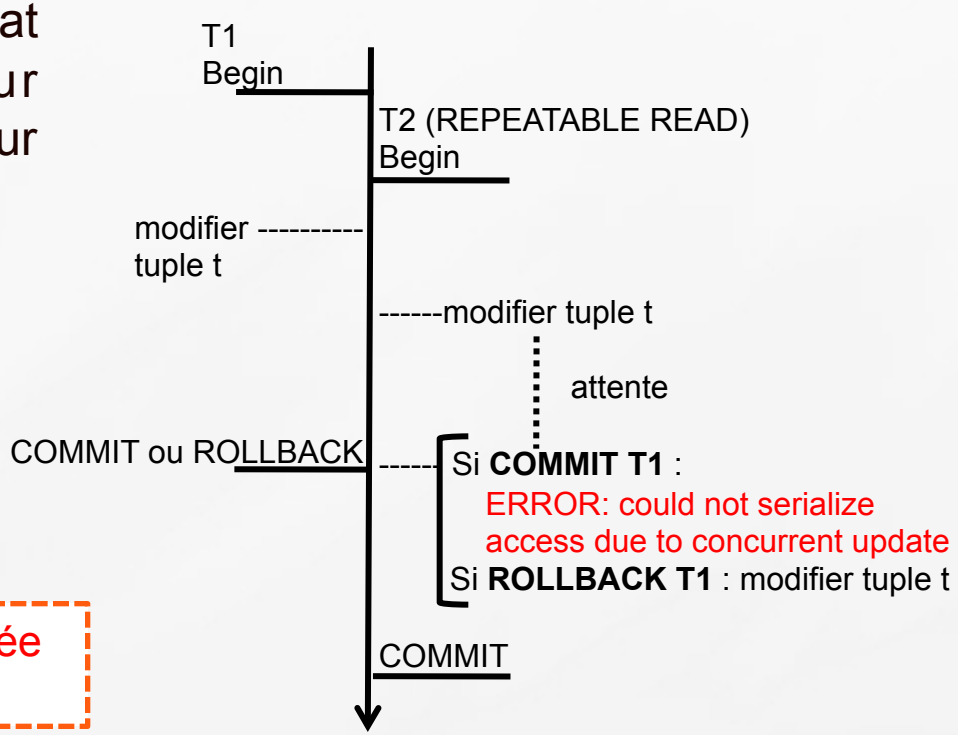


Attention: la création du tuple cible t a été validée avant sa modification dans T2 (sinon t pas visible par T2)

Lectures et modifications concurrentes dans PostgreSQL

- Cas des modifications (suite)
  - **Repeatable read** :

    La transaction attend le résultat de l'autre transaction pour déclencher une erreur ou pour effectuer la modification



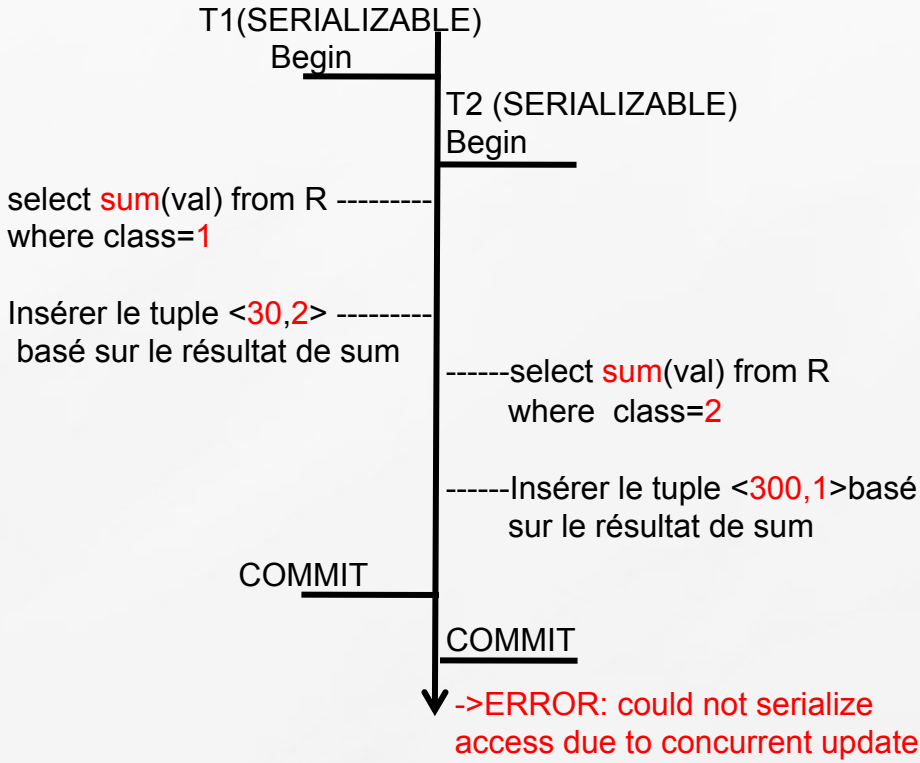
Attention: la création du tuple cible t a été validée avant le début T2 (sinon t pas visible par T2)

Lectures et modifications concurrentes dans PostgreSQL

- Cas des modifications (suite)
  - **Serializable** :
    Identique à « repeatable read » mais avec une surveillance supplémentaire des transactions « serializable » (même des lectures) afin de vérifier les conditions pouvant amener à un résultat incompatible avec une exécution sérielle

➢ surcoût pour surveiller
  - déclarer une transaction en « read only » pour limiter ce surcoût (quand c'est possible)

R	class	val
	1	10
	1	20
	2	100
	2	200



(car si T1 avant T2, on aurait 330 et non 300)

Implémentation de ces mécanismes par le SGBD

- Différentes stratégies mises en place en interne par les SGBD pour garantir la cohérence et les propriétés ACID des transactions

  - Verrouillage: une transaction verrouille certaines données qu'elle lit ou écrit pour interdire aux autres d'y accéder
    - verrouiller un tuple, une table, …
    - ➢ inconvénient: dégrade les performances

  - **Versionnement**: conserver plusieurs versions des données
    - p.ex. image avant mise à jour et image après
    - ➢ avantage: ne bloque pas les lectures/écritures
    - ➢ inconvénient: occupe plus d'espace

- ➢ PostgreSQL utilise un contrôle de concurrence multi-versions (MVCC ou ***Multi-version Concurrency Control***)

Multi-Version Concurrency Control MVCC

- Les transactions sont numérotées par ordre chronologique
- Plusieurs versions de chaque tuple sont conservées
  - chaque version est associée au numéro de la transaction l'ayant créée ou modifiée (insert, update ou delete)
  - PostgreSQL stocke aussi l'état de la transaction (validée, en cours ou annulée)
- En fonction du niveau d'isolation de la transaction et des opérations réalisées, PostgreSQL accède aux bonnes informations et effectue les traitements adaptés

Multi-Version Concurrency Control MVCC

- Exemple simplifié :
  - Soit la transaction 10 en mode SERIALIZABLE qui lit les données *a, b, c, d, e, f* et *g*
  - Soit les transactions 7 et 8 dans l'état "validé"
  - La valeur lue pour *c* par la transaction 10 est celle associée à la transaction 8
  - La valeur lue pour *e* par la transaction 10 est celle associée la transaction 7

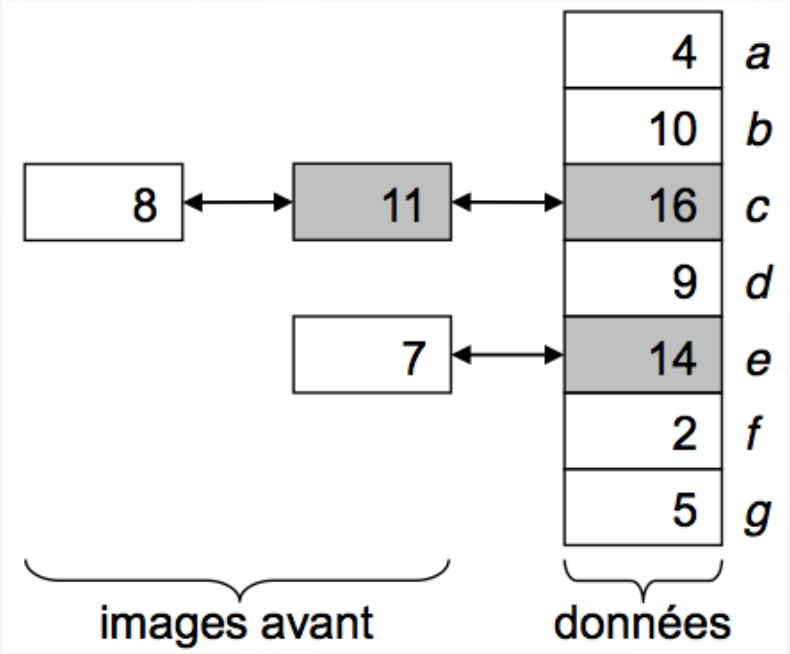


schéma issu du cours de Jacques Le Maitre, Université Sud Toulon-Var

Où sont stockées ces versions ?

- Dans le **journal des transactions**
  - Journal = fichier système qui stocke (p.ex. sur disque dur) tout ce qui passe dans le système, i.e. un historique persistant
  - Il est indispensable pour la validation, l'annulation, la gestion des accès concurrents mais aussi pour la reprise après panne

- 3 fichiers dans PostgreSQL :
  - fichier *pg_log* pour les activités de la base de données
    - p.ex. messages d'erreurs, requêtes, messages de démarrage/arrêt
  - fichier *pg_xlog* pour stocker une image des données manipulées dans les transactions récentes
  - fichier *pg_clog* pour l'état des transactions (validée, annulée, en cours)

Les limites du MVCC

- Nécessité de maintenir des tuples potentiellement obsolètes
  - En réalité, UPDATE créé un nouveau tuple et DELETE ne le supprime pas vraiment (il est simplement marqué comme étant supprimé)
  - Certains tuples vont être conservés alors qu'ils ne peuvent plus être utilisés dans des transactions futures

- Les identifiants des transactions ne peuvent dépasser une certaine valeur maximale
  - Identifiants sont des entiers 32 bits
  - ➢ "Pas plus" de 4 milliards de transactions

➢ Solution: faire des nettoyages réguliers de la base de données (instruction **VACUUM**)
  - Nettoyage automatique configurable (AUTOVACUUM)