Transactions

Anne-Cécile Caron, Anne Étien, Mikaël Monet,
Sylvain Salvati

Pourquoi ?

Le concept de transaction va permettre de définir des processus garantissant que l'état de la base est toujours cohérent

- même en cas d'accès concurrents à la base (cf TP).
- même en cas de panne logicielle ou matérielle.

Plan :

- Présentation des problèmes rencontrés
- Concept de transaction
- Gestion de la concurrence : les solutions
- norme SQL et Postgres
- Reprise après panne

Outline

Exemple

On considère le célèbre exemple de virement bancaire :

```
procedure virement(A,B,X) {
  A := A-X ;
  B := B+X ;
}
```

- A et B sont des "entités" de la base de donnée, X est la valeur du virement.
- A := A-X est une façon simplifiée d'écrire :
  `update COMPTE set solde = solde-X`
  `where refCompte = refA ;`
- Par la suite, un appel à cette procédure se fera à l'intérieur d'une *transaction*.

Lecture/Ecriture

Pour mettre en évidence les problèmes, on s'intéressera aux lectures et écritures faites par une transaction. La procédure de virement devient alors :

```
debut transaction
lire(A)
ecrire(A)
lire(B)
ecrire(B)
fin transaction
```

Premier problème

Un utilisateur exécute un virement bancaire :

```
debut transaction
lire(A)
ecrire(A)
PANNE SYSTEME
```

- Le rôle du système transactionnel est de garantir que la transaction se fait complètement ou pas du tout,
- il doit donc annuler la modification de A (`rollback`).
- Une transaction est *Atomique*.

Gestion de la concurrence

Deux utilisateurs exécutent des virements des mêmes comptes, à l'aide de deux transactions T1 et T2 :

```
T1 : virement(A,B,100)
T2 : virement(A,B,200)
```

Pour des raisons de performance, les actions des différentes transactions sont entrelacées.
Il faut différencier les actions de T1 de celles de T2, et considérer l'ordre dans lequel ces actions vont s'exécuter.

Ordonnancement

Un ordonnancement est une séquence d'actions de la forme (nomTransaction, opération, donnée).

Exemple d'ordonnancement O_1 de T1 et T2 :

```
(T1, lire, A)
(T1, ecrire, A)
(T2, lire, A)
(T2, ecrire, A)
(T1, lire, B)
(T1, ecrire, B)
(T2, lire, B)
(T2, ecrire, B)
```

Deuxième problème

Considérons l'ordonnancement O_2

```
(T1, lire, A)
(T2, lire, A)
(T1, ecrire, A)
(T1, lire, B)
(T1, ecrire, B)
(T2, ecrire, A)
(T2, lire, B)
(T2, ecrire, B)
```

Exercice : quelles sont les modifications faites par les transactions `T1` et `T2`.

Deuxième problème

Considérons l'ordonnancement O_2

```
(T1, lire, A)
(T2, lire, A)
(T1, ecrire, A)
(T1, lire, B)
(T1, ecrire, B)
(T2, ecrire, A)
(T2, lire, B)
(T2, ecrire, B)
```

Exercice : quelles sont les modifications faites par les transactions `T1` et `T2`.
Ici, on a perdu une instruction de A et la base est dans un état *inconsistant*. Les effets des transactions sont modifiés à cause de la *concurrence*.

Propriétés des ordonnancements

- Deux actions d'un ordonnancement sont *conflictuelles* si elles concernent la même entité et qu'au moins l'une des deux est une écriture.
- Deux ordonnancements O_1 et O_2 des mêmes transactions sont *équivalents* si pour toutes actions conflictuelles a et a', a est avant a' dans O_1 ssi a est avant a' dans O_2.
- Un ordonnancement est *sériable* s'il est équivalent à une exécution en série des transactions.
- Seuls les ordonnancements sériables sont corrects.

Example

- L'ordonnancement O_1 est sériable car équivalent à `T1;T2`.
- L'ordonnancement O_2 n'est pas sériable.

Outline

Synthèse : Concept de transaction

- Une transaction est un programme qui modifie la base de données et forme une unité de traitement.
- Elle doit respecter les propriétés ACID
  - Atomicité : une transaction s'effectue entièrement ou pas du tout
  - Consistance : Une transaction qui prend la base dans un état cohérent doit la rendre dans un état cohérent.
  - Isolement : pas d'interférence avec les utilisateurs concurrents.
  - Durabilité : Les actions effectuées par une transaction terminée sont prise en compte dans la base de données.

Concept de transaction (2)

- Les transactions sont gérées par un moniteur transactionnel.
- Une transaction peut être dans différents états :
  - Active : pendant le déroulement du programme, tant qu'aucun problème n'apparaît.
  - Partiellement validée : Lorsque la dernière instruction a été atteinte
  - Validée : Après une exécution totalement terminée (ordre `commit`)
  - Echouée : après un problème qui a interrompu la transaction
- L'instruction `commit` permet de signaler que la transaction s'est bien terminée, les modifications qu'elle a effectuées sont rendues visibles aux autres transactions.
- Si une transaction échoue, le `rollback` permet d'annuler toutes les actions effectuées par cette transaction.

Outline

Outline

Verrouillage 2-phases

- Verrous : une transaction ne peut accéder à une entité dans un certain mode (lecture/écriture) que si elle dispose du verrou correspondant
- Deux verrous conflictuels ne peuvent pas être accordés en même temps
- Deux phases : Une phase pour poser des verrous, une phase pour retirer les verrous. Quand une transaction a enlevé un verrou, elle ne peut plus en poser (phase 2).

Si un ordonnancement est à 2 phases (i.e. toutes ses transactions sont à 2 phases) alors il est sériable.

transaction ”virement”

```
vl(A)
lire(A)
ve(A)
ecrire(A)
vl(B)
lire(B)
ve(B)
ecrire(B)
dl(A)
de(A)
dl(B)
de(B)
```

Inconvénients

- Dead-lock $\rightarrow$ interruption de l'une des transactions.
  Détection :
  - timeout : si une transaction attend un verrou depuis un certain temps, on suppose qu'elle est bloquée par un deadlock
  - détection de cycle dans le graphe des attentes : les noeuds sont les transactions actives et il y a un arc de T_i vers T_j ssi T_i attend une ressource verrouillée par T_j.
- Comment choisir la transaction à interrompre ?
  - laisser les transactions proches de la fin
  - laisser les transactions qui ont fait beaucoup de mise à jour (coût du rollback)
  - ne pas toujours tuer la même transaction
- En pratique, les verrous sont tous posés en début de transaction, pour éviter les interruptions *en cascade*.

Prévention des deadlocks

On peut éviter les deadlocks en donnant à chaque transaction une priorité :

- Chaque transaction est estampillée par la date de son début.
- La plus vieille (estampille plus basse) a la priorité la plus forte
- Si T_i a besoin d'un verrou et que T_j possède déjà un verrou conflictuel, il y a 2 politiques :
  1. **wait-die :** Si T_i a une priorité plus forte alors elle attend pour poser son verrou, sinon elle meurt.
  2. **wound-wait :** Si T_i a une priorité plus forte on tue T_j sinon T_i attend

  Dans les 2 cas, on n'aura jamais de deadlock. On redémarre la transaction avortée avec la même estampille, pour qu'elle puisse à un moment être exécutée (elle va "vieillir" et donc devenir prioritaire).

Performances

- Le verrouillage utilise 2 techniques : le blocage et l'interruption des transactions. Ces 2 mécanismes pénalisent les performances.
  - Les transactions bloquées possèdent des verrous et entraînent le blocage d'autres transactions.
  - L'interruption suivie du redémarrage d'une transaction est évidemment du temps perdu.
- En pratique, moins de 1% des transactions sont impliquées dans un deadlock, et il y a relativement peu d'interruptions.
  $\rightarrow$ Les problèmes de performances viennent plutôt des attentes.

Performances

- Le verrouillage utilise 2 techniques : le blocage et l'interruption des transactions. Ces 2 mécanismes pénalisent les performances.
  - Les transactions bloquées possèdent des verrous et entraînent le blocage d'autres transactions.
  - L'interruption suivie du redémarrage d'une transaction est évidemment du temps perdu.
- En pratique, moins de 1% des transactions sont impliquées dans un deadlock, et il y a relativement peu d'interruptions.
  $\rightarrow$ Les problèmes de performances viennent plutôt des attentes.
- Pour améliorer les performances :
  - Poser des verrous les plus petits possibles, pour diminuer le risque que 2 transactions aient besoin du même verrou
  - Réduire la durée des transactions, pour que les verrous ne soient pas conservés trop longtemps.
  - Eviter les ressources critiques (**hotspot** = objet fréquemment lu ou modifié).

Outline

Alternative au verrouillage

- Le verrouillage est une façon "pessimiste" de traiter la concurrence : on part du principe qu'il va y avoir des conflits et on gère des verrous. Si les conflits sont rares, la gestion des verrous fait perdre en performance.
- On peut aussi traiter le problème de façon optimiste, en partant du principe qu'il y aura très peu de conflits :
  1. Lecture : la transaction s'exécute, lit des données dans la base et écrit dans un espace privé
  2. Validation : Quand la transaction est terminée, le SGBD vérifie qu'elle peut être validée, i.e. qu'il n'y a pas eu de conflit avec une autre transaction pendant son exécution. En cas de conflit, la transaction est annulée, son espace privé est vidé, et la transaction est relancée
  3. Ecriture : Si la transaction termine sans conflit, les données écrites dans l'espace privé sont copiées dans la base.

Estampillage pour éviter le verrouillage

La gestion "optimiste" de la concurrence utilise l'estampillage des transactions.

- Chaque transaction T_i reçoit une estampille TS_i au début de la phase de validation.
- Validation : pour chaque couple (T_i, T_j) tel que $TS(T_i) < TS(T_j)$, T_j est validée si l'une des 3 conditions est vrai :
  1. T_i a terminé ses 3 phases avant que T_j commence (exécution en série).
  2. T_i a terminé avant que T_j commence la phase d'écriture, et T_i n'écrit aucun objet lu par T_j
  3. T_i a terminé sa phase de lecture avant que T_j termine sa phase de lecture, et T_i n'écrit pas d'objet qui est lu ou écrit par T_j

Estampillage pour éviter le verrouillage

- Pour vérifier ces critères de validation, il faut gérer une liste des objets lus et écrits par chaque transaction
- Il faut aussi s'assurer qu'au plus une transaction est en cours de validation/écriture, donc gérer ce process comme une section critique.
- *donc s'il y a beaucoup de conflits, ça devient un goulot d'étranglement qui pénalise les performances.*

Conclusion :

- Une gestion pessimiste est intéressante s'il y a beaucoup de conflits, pénalisante sinon car on gère des verrous pour rien
- Une gestion optimiste est intéressante s'il y a peu de conflits, donc peu d'attente liée à la section critique des phases de validation/écriture.

Outline

Multi-versions

- On conserve les versions successives d'une même donnée, ce qui revient à donner une estampille aux objets, pas seulement aux transactions.
- Lorsqu'une transaction T a besoin de lire une donnée, elle va chercher la version qui lui faut, i.e. la version la plus récente qui précède $TS(T)$.
- Lorsqu'une transaction T_i a besoin d'écrire un objet, il faut vérifier qu'il n'a pas déjà été lu par T_j telle que $TS(T_i) < TS(T_j)$.
- on associe une estampille en lecture $RTS(O)$ à chaque objet O, et une estampille $TS(T)$ à chaque transaction T à son démarrage.
  1. Si T veut lire O alors $RTS(O)$ prend la valeur $sup(RTS(O), TS(T))$.
  2. Si T veut écrire O et $TS(T) < RTS(O)$, T est avortée et redémarrée avec une nouvelle estampille.
  3. Sinon, T crée une nouvelle version de O avec l'estampille $RTS(O) = TS(T)$
- Coût du stockage ; algo permettant de jeter les versions qui ne serviront plus.
- Avantage : les lectures ne sont jamais bloquées.

Outline

Norme SQL2 - 1992

La norme définit deux caractéristiques pour une transaction :

- Le mode, i.e. les opérations possibles,
  - `READ ONLY` *transaction-level read consistency*
  - `READ WRITE` (par défaut) *statement-level read consistency*
- Le niveau d'isolement : Le TP illustre quelques problèmes que l'on peut rencontrer avec SQL utilisé de manière concurrente. La norme définit des niveaux d'isolement pour empêcher ces problèmes.

Niveau d'isolement de la norme SQL

- `READ UNCOMMITTED` : aucun isolement des transactions. On peut avoir des lectures inconsistantes (dirty read) des tables, i.e. lire une valeur modifiée par une autre transaction mais non validée.
- `READ COMMITTED` : évite les lectures inconsistantes. On ne lit que des données dont les modifications ont été validées.
- `REPEATABLE READ` : empêche le problème des lectures répétées tq entre 2 lectures, certaines lignes ont disparu ou ont été modifiées. N'empêche pas le problème des lignes "fantômes", ajoutées entre une lecture et la suivante.
- `SERIALIZABLE` : Garantit la sériabilité des ordonnancements. Évidemment ça pose des problèmes de performance.

Norme SQL3 - 1999

- *points de contrôle* dans les transactions.

```
begin
  insert into joueur
    values ('165789','Bisk','Otto');
  savepoint p1;
  insert into joueur
    values ('376487','Biss','Scott');
  rollback to p1;
  commit ;
end ;
```

  La première instruction `insert` est validée, pas la seconde.

- Le concept de transaction atomique est étendu à celui de **transactions imbriquées**
- Ces deux ajouts permettent de manipuler des transactions plus longues, mais étant composées de sous-transactions (courtes) qui peuvent être annulées indépendamment.
- Les points de contrôle sont juste un support pour la forme la plus simple d'imbrication (1 niveau)

Les transactions sous Postgres

- Une transaction commence par l'instruction `BEGIN TRANSACTION;` ou juste `BEGIN;`
- Une transaction se termine par l'instruction `COMMIT` ou `ROLLBACK`.
- L'instruction `commit` valide toutes les modifications effectuées depuis le début de la transaction.
- L'instruction `rollback` annule toutes les modifications effectuées depuis le début de la transaction.
- Une erreur (par exemple contrainte non satisfaite) entraîne un `rollback` implicite.
- Une instruction SQL qui n'est pas dans une transaction est automatiquement validée par un commit implicite ; sauf si elle provoque une erreur, elle est alors annulée par un rollback implicite. Elle constitue donc à elle seule une transaction.
- A l'intérieur d'une transaction, on peut définir des points de contrôle (`savepoint`).
- Postgres suit la norme concernant les modes et les niveaux d'isolement possibles.

Niveaux d'isolement des transactions sous Postgres

- La commande `set transaction` permet de fixer le niveau au sein d'une transaction (après instruction `begin`)
  Syntaxe Postgres : `set transaction isolation level` *`niveau`*
  niveau peut correspondre à : `SERIALIZABLE`, `REPEATABLE READ`, `READ COMMITTED` ou `READ UNCOMMITTED`
- Comme dans beaucoup de SGBD, le niveau `read committed` est le niveau par défaut.
- Le standard SQL accepte qu'un niveau soit plus strict que la norme, c'est le cas pour Postgres pour les niveaux `Uncommitted read` et `Repeatable read`. Cela implique que les lectures inconsistantes sont impossibles.

Isolation Level	Dirty Read	Nonrepeatable 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

Verrouillage

- Postgres utilise à la fois du multi-versions et du verrouillage pour que les lectures n'entrent pas en conflit avec les écritures, i.e. une lecture ne bloque pas une écriture et réciproquement une écriture ne bloque pas une lecture.
- A chaque instruction SQL utilisant une table, un verrou est posé sur cette table (le type de verrou dépend de l'instruction exécutée) et éventuellement sur certaines lignes (e.g. celles modifiées). Selon le verrou, d'autres instructions vont pouvoir accéder ou pas à cette table.
- Postgres gère 8 niveaux de verrouillage des tables, et 4 niveaux de verrouillage des lignes. Pour simplifier :
  - Une transaction pose un verrou en lecture quand elle exécute un `SELECT` ;
  - Une transaction pose un verrou en écriture quand elle exécute un `INSERT`, `DELETE`, `UPDATE` ou `SELECT ... FOR UPDATE` ;
  - Les instructions du DDL entraînent aussi la pose de verrous plus restrictifs vis à vis des accès concurrents.
- Si le comportement par défaut ne convient pas, l'utilisateur peut explicitement poser des verrous avec la commande `LOCK TABLE`.
- Le verrouillage dure le temps de la transaction : il prend fin au premier commit ou rollback (commande explicite ou implicite).

Outline

Reprise après panne

Il existe un gestionnaire de reprise après panne, qui est responsable de 2 propriétés :

1. La durabilité : les effets des actions effectuées par les transactions validées au moment de la panne doivent être enregistrées dans la base.
2. L'atomicité : une transaction se fait entièrement ou pas du tout. Il faut donc défaire le travail des transactions non validées qui étaient actives au moment de la panne.

Cause de l'échec d'une transaction

- Problème logique (erreur logicielle, contrainte non satisfaite ...) : l'opération `rollback` permet de remettre la base dans l'état qu'elle avait avant le début de la transaction.
- Panne du serveur ou de l'OS : remettre la base dans l'état qu'elle avait avant la panne,
- Problème physique (panne serveur, crash disque ...) : utiliser des sauvegardes des fichiers.

En fait, le gestionnaire de reprise après panne est aussi responsable de l'exécution des rollbacks.
$\rightarrow$ Les procédés mis en oeuvre pour annuler une transaction sont aussi utilisés pendant une reprise après une panne système.

Algorithme ARIES

- ARIES = base des algo de reprise après panne des SGBD actuels.
- Cet algorithme utilise la **journalisation**
- Toutes les actions effectuées par le SGBD sont tracées dans un journal (fichier de log).
- En cas de panne, le journal est utilisé pour remettre la base dans l'état où elle se trouvait au moment de la panne.
- Cet algorithme fonctionne parce que le journal est sauvegardé sur disque à chaque commit, et surtout AVANT que les pages modifiées (données) ne soient écrites sur disque.

Algorithme ARIES (2)

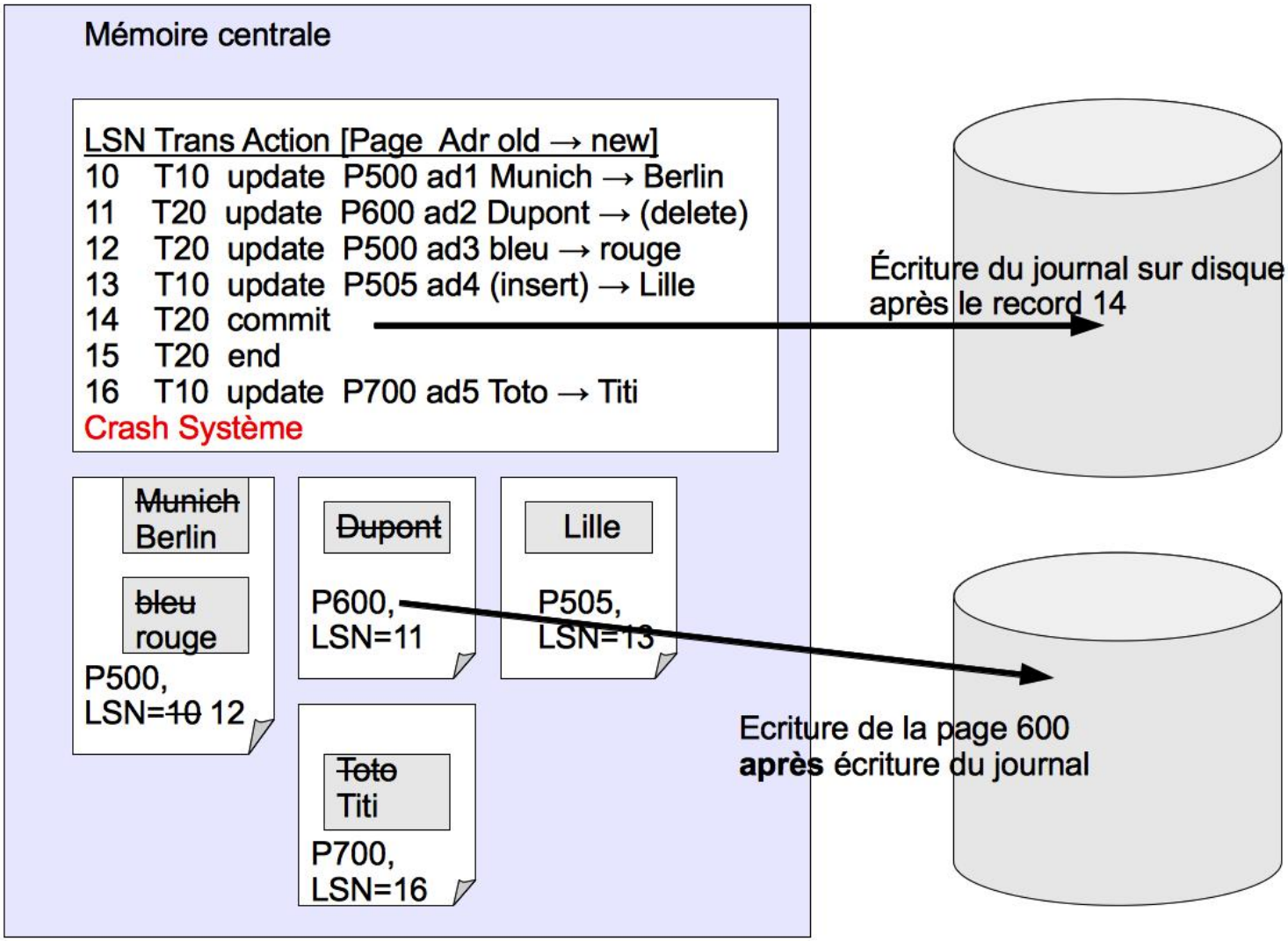
Reprise après une panne :

1. Analyse :
   - identifier les pages "sales" dans le buffer, i.e. les pages dont les changements n'ont pas été écrits sur disque.
   - identifier les transactions actives au moment du crash, i.e. celles en court d'exécution qui n'ont pas atteint de `commit`.
2. Redo : répéter toutes les actions à partir d'un point p donné dans le journal, afin de restaurer l'état de la base au moment du crash. Ce point p est déterminé au moment de la phase d'analyse.
3. Undo : défaire les actions des transactions qui n'étaient pas validées au moment du crash, afin que la base ne contienne que les effets des transactions validées.

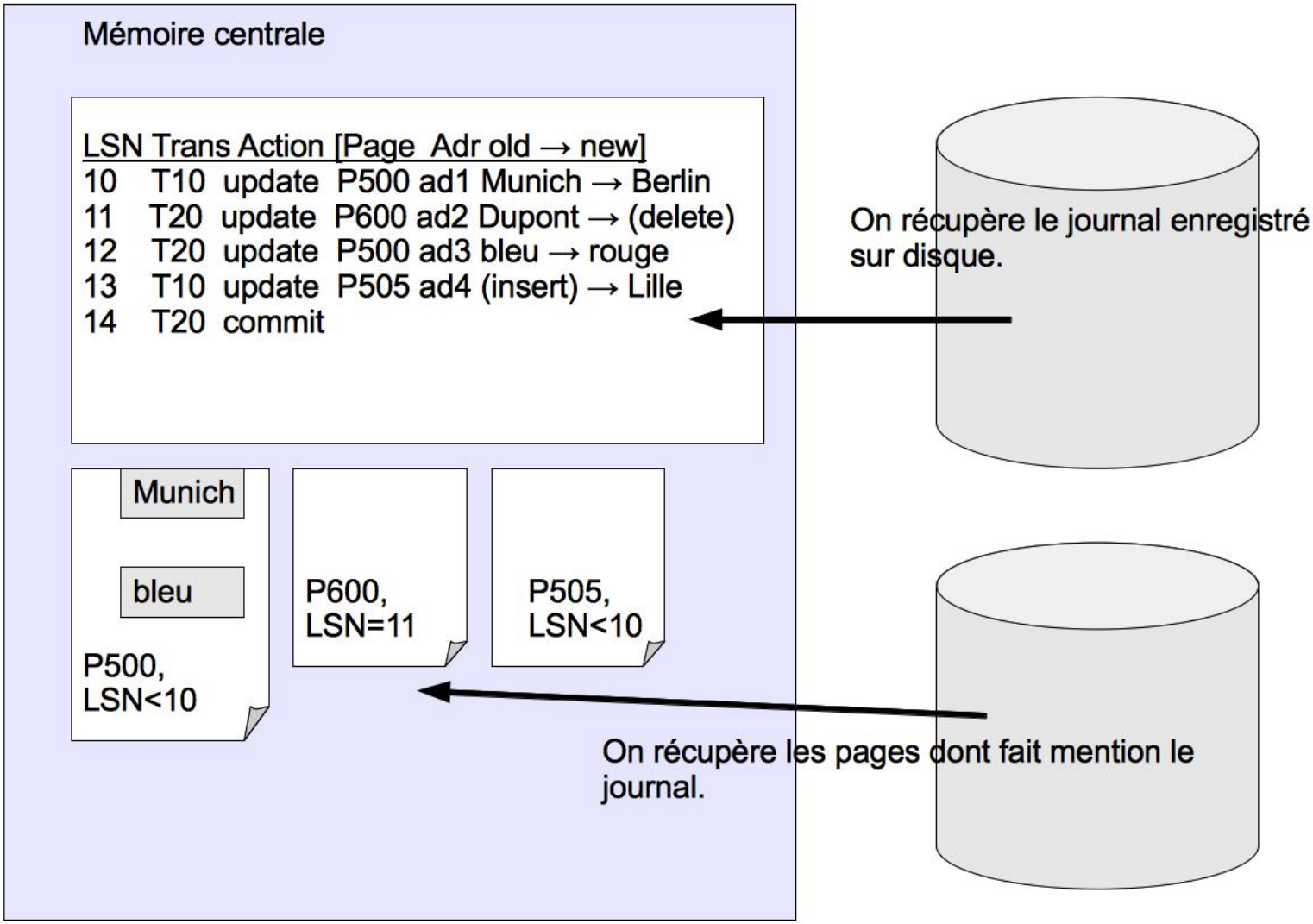
Structures utilisées

- Le journal : il contient une séquence d'enregistrements appelés log records. Chaque record comporte un identifiant (LSN) attribué de façon strictement croissante, selon l'ordre chronologique, ainsi que des informations sur l'action exécutée.
- Table des transactions actives : permet de mémoriser les transactions qui n'ont pas encore exécuté de commit ou de rollback. Dès qu'un commit ou rollback est complètement exécuté, un record de type `end` est inscrit au journal et la transaction est retirée de la table des transactions actives.
- Table des pages "sales". Lorsqu'une page est écrite sur disque, elle a comme information le LSN de la dernière action du journal qui la concerne (dernière au moment de l'écriture). On retire alors la page de la table des pages sales, jusqu'à ce qu'une nouvelle action la modifie.

Exemple - crash système



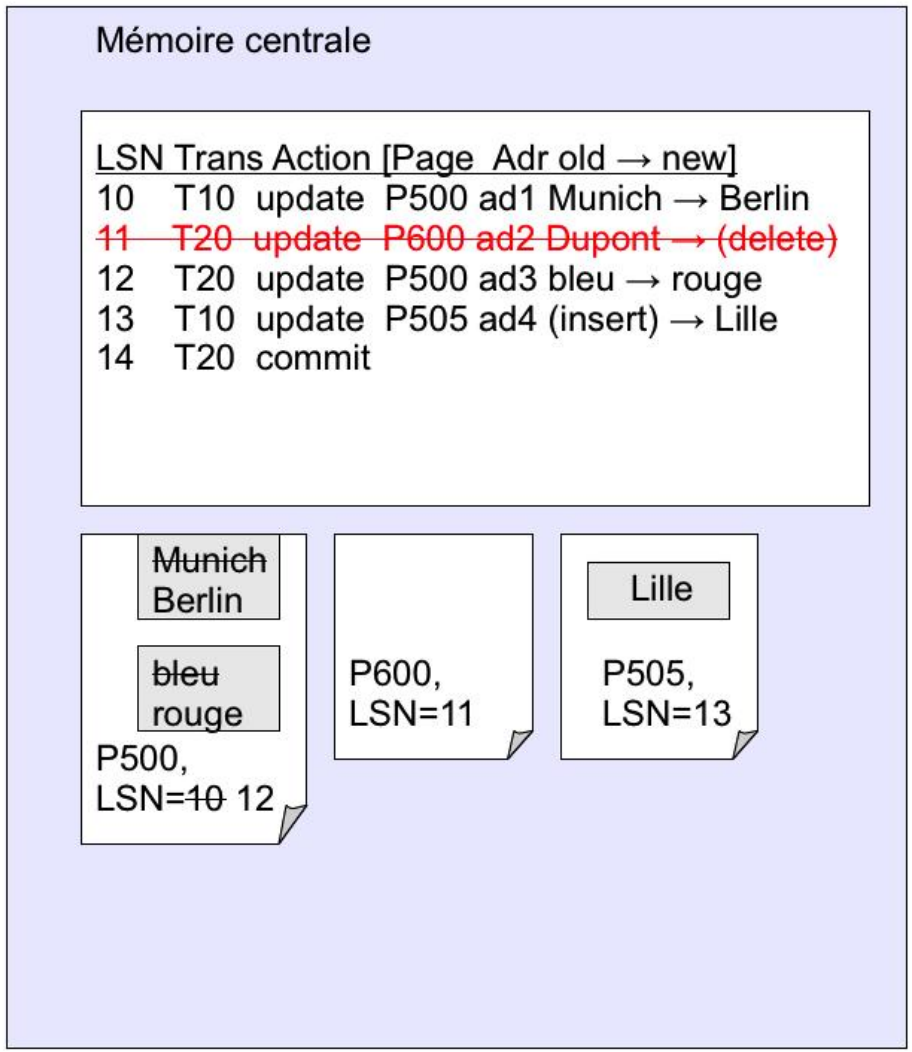
Exemple - redémarrage



Exemple - phase d'analyse

- On lit le journal sur le disque. On l'analyse à partir du LSN 10 (on verra plus tard comment choisir le point de départ)
- Ce journal permet de reconstruire les tables
  - la table des transactions actives ne contient que T10 (car on a lu un record `commit` pour T20).
  - la table des pages sales contient P500, P505, P600 (si on améliore un peu l'algo, on peut remarquer que P600 n'était plus une page sale au moment du crash ...)
- L'instruction de modification de la page P700 a complètement disparu car elle n'est pas inscrite dans le journal. Ce n'est pas grave puisque T10 va être annulée.

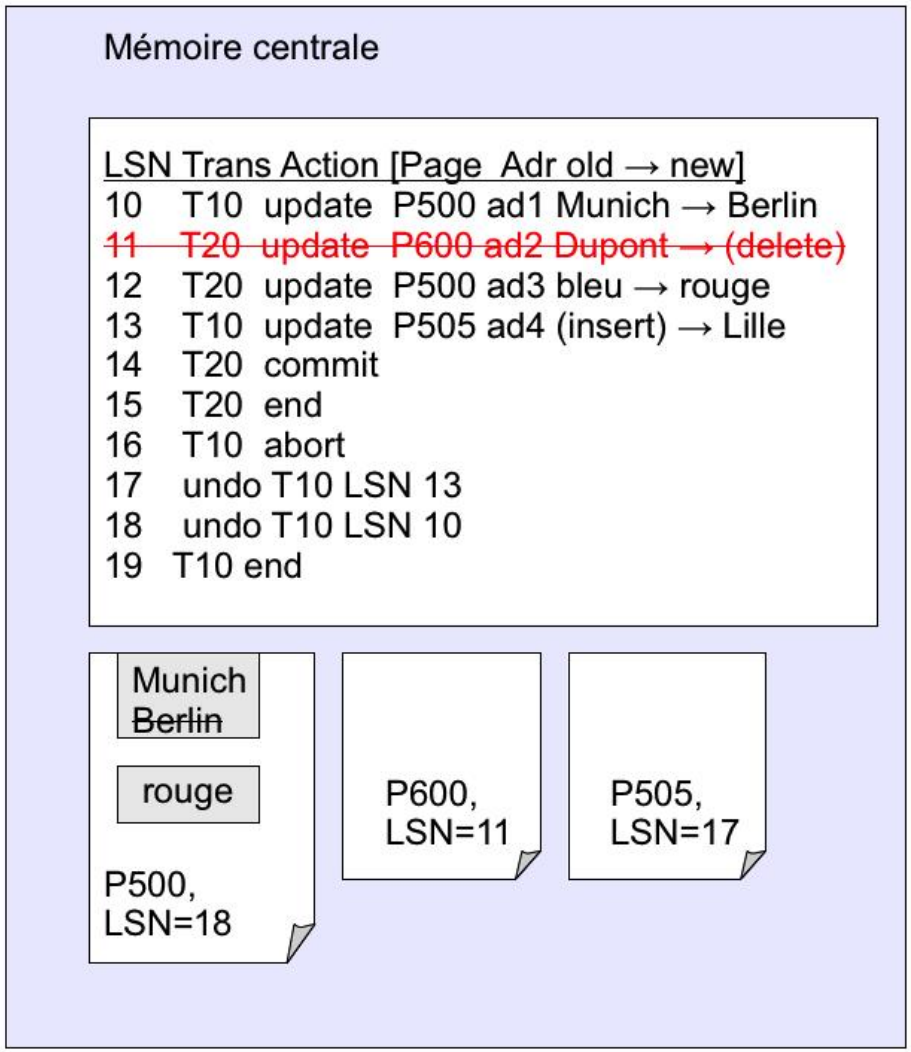
Exemple - phase REDO



REDO : on exécute les actions du journal.

On n'exécute pas le record de LSN=11 car la page 600 a aussi un LSN=11 (donc la modif a déjà été enregistrée en base).

Exemple - phase UNDO



UNDO : on « défait » les transactions actives (T10) en lisant le journal par ordre décroissant.
Ces actions sont aussi inscrites dans le journal, en cas de panne pendant le redémarrage.

Le rollback

Le rollback utilise des traitements que l'on vient de voir pour la reprise après panne :

- On utilise les pages en mémoire et le journal pour défaire la transaction : comme dans la phase undo (sur l'exemple les records de 16 à 19 correspondent à un rollback de T10).
- On enlève les verrous posés par la transaction.
- On enlève la transaction de la table des transactions actives.

Checkpoints

En cas de reprise après panne, combien de lignes du journal doit-on traiter ?

- Il faut savoir à quelle moment la base s'est trouvée dans un état cohérent.
- A un moment donné, tous les buffers de données (pages "sales") modifiés présents en mémoire principale sont écrits sur disque. Ce moment est appelé **checkpoint**.
- Dans le journal, on inscrit le début et la fin du checkpoint (ce qui n'est pas fait dans l'exemple).
- Comme les checkpoints sont inscrits dans le journal, le REDO démarre à partir du checkpoint achevé le plus récent.