Gestion des transactions : fiabilité et concurrence

dans les bases de données relationnelles

Bernard ESPINASSE
Professeur à Aix-Marseille Université (AMU)
Ecole Polytechnique Universitaire de Marseille

Décembre 2017

- **Transactions**
- **Concurrence et transactions**
- **Fiabilité et transactions**
- **Gestion des transactions réparties**
- **Modèles étendus de gestion des transactions**

Plan

1. Transactions

- **Requêtes primitives et super primitive**
- **Définition et déclaration d'une transaction**
- **Validation et annulation d'une transaction**

Requête primitive

Une instruction ou « requête primitive » (unique) jouit des propriétés ACID (garantie contractuelle du SGBD)

Exemple : **DELETE FROM Professeur WHERE numProf = 1 ;**

Cette requête conduit généralement à plusieurs modifications dans la base de données :

- suppression d'une ligne de la table « Professeur »
- suppression de plusieurs lignes de la table « Charge »

Requête primitive et propriétés ACID :

- **Atomicité** : l'ensemble des actions élémentaires est réalisé complètement ou pas du tout
- **Cohérence** : si les données sont cohérentes au départ, elles le sont à l'arrivée (intégrité référentielle par exemple)
- **Indépendance** : l'exécution de primitives d'autres processus n'interfère pas avec l'exécution de cette primitive (les lignes de PROFESSEURS et CHARGE touchées sont protégées)
- **Durabilité** : si la primitive se termine normalement, les lignes supprimées le restent, quels que soient les incidents qui pourront se produire ultérieurement

Suite de requêtes primitives

Problème : la granularité des primitives peut être trop fine pour certaines opérations sur les données

Exemple : Transfert d'une somme d'un compte vers un autre = 2 primitives

… si incident: OK
update du compte source: - 1000 €
… si incident: incohérence
update du compte destination: + 1000 €
… si incident: OK

Solution :

… si incident: OK
SUPER PRIMITIVE (**update** du compte source: - 1000 €
update du compte destination: + 1000 €)
… si incident: OK

=> Transaction = « super primitive » construite par le programmeur

Notion de transaction

Transaction = Unité Logique de Traitement (ULT - LUW) c'est à dire une séquence d'instructions, dont l'exécution assure le passage de la BD d'un état cohérent à un autre état cohérent

- **Exemples de transactions** : Enregistrer une commande, enregistrer un professeur et ses charges, faire un transfert de compte à compte bancaire, …
- Les transactions sont une réponse générale aux problèmes de :
  - **Fiabilité** : Lorsque le système tombe en panne lors d'un traitement en cours, il y a risque qu'une partie seulement des instructions prévues soit exécutée, ce qui peut conduire à des incohérences.
  - **Concurrence** : Lorsque 2 accès concurrents se font sur les données, il y a un risque que l'un des accès rende l'autre incohérent. Ex : si 2 utilisateurs modifient une même donnée au même moment, seule une des 2 mises à jour sera effectuée.

La **gestion de transactions par les SGBD** permet d'assurer le maintien de la cohérence des BD, elle assure que toutes les contraintes de la BD seront toujours respectées, même en cas de panne et au cours d'accès concurrents.

Principe et déclaration d'une transaction

Principe :

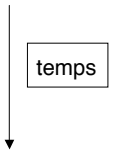
- **Le SGBD garantit les propriétés ACID lorsqu'il exécute une primitive**
- si le **programmeur** désire que cette propriété s'applique à une séquence d'instructions, il doit en faire une **transaction**

Déclaration d'une transaction :

Une transaction est déclarée à l'aide de **3 primitives offertes par le SGBD** :

ouverture d'une transaction	**begin-transaction**
clôture avec confirmation	**COMMIT**
clôture avec annulation	**ROLLBACK**

préparation
begin-transaction
acquisition des données
modification des données
si erreur **ROLLBACK** et sortie
COMMIT
suite du traitement



Ouverture et clôture d'une transaction

Ouverture d'une transaction :

- **begin transaction** : déclare **l'ouverture d'une nouvelle transaction**

clôture avec confirmation d'une transaction :

- **COMMIT** : les insertions, suppressions et modifications d'enregistrements spécifiées dans la transaction sont **impactées** sur la base de données

clôture avec annulation d'une transaction :

- **ROLLBACK** : les insertions, suppressions et modifications d'enregistrements spécifiées dans la transaction sont **annulées**

Mode « Autocommit »

- La plupart des SGBDR proposent un mode « **autocommit** » permettant d'encapsuler chaque instruction SQL dans une transaction
- Ce mode revient à avoir un COMMIT implicite après chaque instruction
- Il doit être désactivé pour permettre des transactions portant sur plusieurs instructions

Oracle :

- Sous Oracle SQL*Plus : SET AUTOCOMMIT ON et SET AUTOCOMMIT OFF

***PostgreSQL** avec psql :*

- \set AUTOCOMMIT on et \set AUTOCOMMIT off

Si l'autocommit est activé, il est néanmoins possible de démarrer une transaction sur plusieurs lignes en exécutant :

BEGIN TRANSACTION explicite : BEGIN ; ... ; COMMIT ;

Ainsi 2 modes sont possibles :

- Autocommit activé : BEGIN explicites, COMMIT implicites
- Autocommit désactivé : BEGIN implicites, COMMIT explicites

Annulation d'une transaction : Rollback

Evénements survenant lors de l'exécution d'une transaction :

- **Arrêt interne (géré par le programmeur)**

  Lors de l'exécution d'une transaction, une condition peut être détectée, rendant la poursuite de la transaction impossible

  Exemple : violation d'une contrainte d'intégrité

  Action du SGBD : effacer toute trace de la transaction annulée

- **Arrêt externe (panne - géré par le SGBD)**

  Au cours de l'exécution des primitives, un événement extérieur peut arrêter définitivement l'exécution de la transaction

  Exemple : panne

  Action du SGBD : effacer toute trace de la transaction annulée après la reprise (reprise après panne)

Propriétés des transactions

Une transaction vérifie, relativement à la base de données, les 4 propriétés « A.C.I.D. » :

- **Atomicity (atomicité)** : l'ensemble des instructions de la section sont exécutées complètement (en cas de succès), ou pas du tout (en cas d'incident), mais jamais partiellement;
- **Consistency (cohérence)** : si la BD est cohérente avant la transaction, elle le sera à la sortie de celle-ci (respect des contraintes d'intégrité);
- **Isolation (indépendance)** : les interactions entre la transaction et la base de données s'effectuent comme si la transaction était seule à travailler sur la BD (pas d'interférences nuisibles avec les autres transactions);
- **Durability (permanence)** : si la transaction s'est terminée correctement, les modifications faites dans la BD sont garanties permanentes (sauf modifications via d'autres transactions), quels que soient les incidents ultérieurs.

A – Atomicité de la transaction

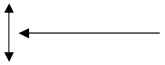
Atomicité :

Une transaction doit être **traitée comme une seule opération,** c-a-d que le SGBD doit assurer que **toutes les actions de la transaction sont exécutées ou bien qu'aucune**

Exemple :

```
begin-transaction;
INSERT ... ;
DELETE ... ;
```

Actions exécutées



Panne

```
INSERT ... ;
DELETE ..;
COMMIT;
```

Actions suspendues

Si l'exécution d'une transaction est interrompue à cause d'une panne quelconque, le SGBD doit être capable de décider des actions à entreprendre après la réparation de la panne :

- Soit **compléter la transaction en exécutant les actions restantes**
- Soit **défaire les actions qui avaient déjà été exécutées avant la panne** (journal des actions)

C - Cohérence d'une base de données

Base de données dans un état cohérent :

si les valeurs contenues dans la **base vérifient** toutes les **contraintes d'intégrité définies** sur la base

Problème : Des **contraintes** ne sont **satisfaites** que **suite à une séquence de plusieurs primitives** !

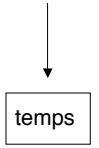
=> Etats intermédiaires incohérents

Ex 1 : Transfert d'un compte vers un autre (contrainte: la somme des comptes est invariante). Après la modification du premier compte, la BD est incohérente

Ex 2 : Après la suppression d'une commande (contrainte: toute ligne de commande doit être associée à une commande), la BD peut être incohérente

```
begin-transaction;
delete from COMMANDE where num_cde = 1 ;
delete from LIGNE where num_cde = 1 ;
COMMIT;
```

Etat cohérent
Etat incohérent !
Etat cohérent



Solutions SQL possibles :

- **Ordonnancement judicieux des primitives**
- **Débrayer la vérification des contraintes**

C - Cohérence d'une base de données

Débrayage de la vérification des contraintes :

```
CREATE TABLE LIGNE
(num_article integer not null,
num_commande integer not null,
primary key (num_article, num_commande)
CONSTRAINT ligne_article
FOREIGN KEY (num_cours) REFERENCES ARTICLES
CONSTRAINT ligne_commande
FOREIGN KEY (num_commande) REFERENCES COMMANDES
INITIALLY DEFFERABLE) ;
```

Sauf ordre contraire dans la transaction, la **vérification des contraintes déclarées sera d'office (initially) reportée (deferrable) au COMMIT :**

```
begin-transaction;

delete from COMMANDES where num_cde = 1 ;
delete from LIGNES where num_cde = 1 ;

COMMIT;
```

pas de vérification !
vérification !

I - Isolation de la transaction

Isolation :

les interactions entre la **transaction** et la **BD s'effectuent** comme **si la transaction était seule à travailler** sur la BD

=> propriété à assurer pour des **transactions concurrentes**

D - Durabilité & Isolation de la transaction

Durabilité :

propriété qui assure que **lorsqu'une transaction a été confirmée (COMMIT), ses effets deviennent permanents**

Importance: après un arrêt externe (panne) le SGBD doit s'assurer que :

- les effets d'une transaction validée apparaissent dans la base de données après la reprise
- les effets d'une transaction annulée n'apparaissent pas dans la base de données après la reprise

3. Concurrence et transactions

- Transactions concurrentes
- Problèmes liés à la concurrence
- Verrouillage
- Transactions dans les systèmes répartis

Transactions concurrentes

Transactions concurrentes :

2 transactions sont concurrentes si elles accèdent en même temps aux mêmes données

Problème possibles :

lorsque certaines transactions font des mises jour sur la BD :

- **Perte de mise à jour**
- **Lecture impropre**
- **Lecture non reproductible**

Pb de concurrence : Pertes de mises à jour

- **T1 et T2 transactions concurrentes**
- **T1 : lire (x) ; x <- x + 10 ; écrire (x)**
- **T2 : lire (x) ; x <- x + 20; écrire (x)**
- **Au départ x = 50**

T1	T2
Lire (x) (T1 lit x = 50 dans BD)	
x <- x + 10	
	Lire (x) (T1 lit x = 50 dans BD)
Ecrire x (dans BD : x = 60)	
	x <- x + 20
	Ecrire x (dans BD : x = 70)

temps ↓

Bilan :

- **la mise à jour effectuée par T1 a été écrasée par celle faite par T2**

Pb de concurrence : Lecture impropre (1)

- **T1 et T2 transactions concurrentes**

T1	T2
UPDATE comptes SET solde = 25000 WHERE num_compte = '9' ;	
	SELECT solde FROM comptes WHERE num_compte = '9' ;
ROLLBACK	

temps ↓

Bilan :

- Propriété **d'atomicité** : la mise à jour ne devrait pas être prise en compte
- Dès lors, la valeur lue par T2 est incorrecte.
- Cette valeur est dite **impropre** (T2 lit des données **non confirmées**)

Pb de concurrence : Lecture impropre (2)

- **T1 et T2 transactions concurrentes**
- **T1 transfère 10 euros du compte '9' vers le compte '5'**
- **T2 affiche le solde total des 2 comptes**

T1	T2
UPDATE Comptes SET solde = solde - 10 WHERE num_compte = '9' ;	
	SELECT SUM (solde) FROM Comptes WHERE num_compte in ('9', '5') ;
UPDATE Comptes SET solde = solde + 10 WHERE num_compte = '5' ;	
COMMIT	

temps ↓

Bilan :

- la somme des comptes affichée par T2 est incorrecte puisque le solde du compte '5' n'a pas encore été mis à jour
- **Etat intermédiaire incohérent de la base de données**

Pb de concurrence : Lecture non reproductible

- **T1 et T2 transactions concurrentes**
- **T1 effectue plusieurs fois la lecture du même objet**

temps ↓

T1	T2
SELECT points FROM Resultats WHERE num_cours = 5 AND num_eleve = 7 ;	
	UPDATE Resultats SET points = points * 1.1 WHERE num_cours = 5 AND nom_eleve = 7 ;
SELECT points FROM Resultats WHERE num_cours = 5 AND num_eleve = 7 ;	
COMMIT	

- **En principe, T1 doit obtenir à chaque lecture la même valeur**
- **Mais, si entre 2 lectures de l'objet, celui-ci est modifié par T2, T1 n'obtient plus le même valeur : lecture non reproductible !**

Verrouillage

Diverses techniques selon les SGBD, consistant toutes à réaliser un verrouillage temporaire de la BD :

- d'une **partie** plus ou moins grande de la BD (BD, sous-ensemble de la BD, table, page, tuple,....),
- pendant un **temps** plus ou moins long (le temps d'une transaction,....),
- tenir compte du fait que les opérations effectuées par les utilisateurs sur la base peuvent être soit en **consultation** soit en **mise à jour:**

2 types de verrous :

- **verrou exclusif (write lock):** ® *installé pour une opération de MAJ, il interdit à d'autres utilisateurs d'effectuer simultanément d'autres MAJ ou consultations sur une même partie de la BD*
- **verrou partagé (read lock):** ® *installé pour une opération de consultation, il permet à d'autres utilisateurs d'effectuer simultanément d'autres consultations sur une même partie de la BD, et interdit toute mise à jour*

Verrouillage à "2 phases" :

- 1 - phase de **croissance d'acquisition de verrous** (SEIZE BLOCK)
- 2 - phase de **décroissance de libération de verrous**

Exemple de verrouillage

Séquence de verrouillage/transaction

1- verrouiller des pages ou une table
2- lecture ou écriture des données
3- si d'autres verrouillages sont nécessaires, retourner en 1
4- quand la transaction est terminée, relâcher tous les verrous.

Exemple de verrouillage/transaction (en SQL) :

1- begin transaction
2- **update** employé **set** salaire = salaire * 1.05
3- **select** * **from** departement
4- end transaction

2- l'utilisateur installe un verrou exclusif sur la table employé
3- l'utilisateur installe un verrou partagé sur la table departement
4- l'utilisateur relâche tous les verrous.

Epaisseur du verrou (SGBD « INGRES »)

- **2 types de verrou: exclusif et partagé**
- **durée du verrouillage: durée d'une transaction**
- **épaisseur du verrou: table ou page**

Choix de l'épaisseur du verrou dans un SGBD : ***en général est effectué par l'optimiseur de requête du SGBD:***

- **verrou sur table lorsque:**
  - la requête touche une table entière
  - la requête n'est pas très restrictive
  - *la requête n'utilise pas des structures indexées*

  exemples:
  - **update** employé **set** salaire = salaire * 1.05
  - **update** employé **set** salaire = salaire * 1.05 **where** salaire > 10000;

- **verrou sur page lorsque:**
  - la requête est très restrictive
  - *la requête utilise des structures indexées*

  exemples (si les tables sont indexées):

  select * from employé **where** nom = "Durant";

  update departement **set** vente=200000 **where** nom_departement='commercial';

L'attente d'un verrou

Utilisateur 1	Utilisateur 2
1. begin transaction	2. begin transaction
3. **update** employé **set** salaire = 4000	4. **update** employé **set** salaire = 3000
5. **select * from** departement **where** nom_departement ='commercial'	… attente …
6. end transaction	7. end transaction

3 - l'utilisateur 1 installe un verrou exclusif sur la table emp

4 - l'utilisateur 2 doit attendre la libération du verrou installé sur emp

5 - l'utilisateur 1 installe un verrou partagé sur la table dept

6 - l'utilisateur 1 libère les 2 verrous et l'utilisateur 2 peut continuer sa transaction

Verrou mortel (dead lock)

Utilisateur 1	Utilisateur 2
1. begin transaction	2. begin transaction
3. **select * from** employé	4. **select * from** departement
5. **update** departement **set** nom_departement ='commercial'	6. **insert into** employé **value** (Duval, ...)
Blocage !	Blocage !

3 - l'utilisateur 1 installe un verrou partagé sur la table emp

4 - l'utilisateur 2 installe un verrou partagé sur la table dept

5 - l'utilisateur 1 doit attendre la libération du verrou installé par l'utilisateur 2 sur la table dept

6 - l'utilisateur 2 doit attendre la libération du verrou installé par l'utilisateur 1 sur la table emp

bloquage !

Verrou mortel (dead lock)

- apparaît lorsque les transactions de 2 utilisateurs se bloquent mutuellement
- peut aussi apparaître lors d'une simple requête à cause d'un verrouillage sur page
- lorqu'un verrou mortel survient, le SGBD doit:
  - **détecter** le bloquage
  - **avorter une transaction** et laisser continuer l'autre transaction
- la manipulation des dead-lock est un travail important pour les programmeurs

```
EXEC SQL .....
IF SQLCODE < 0 (valeur indiquant l'interblocage)
THEN DO
    EXEC SQL ROLLBACK
    ...
    END.
```

2. Fiabilité et transactions

- **Types de pannes liés au transactionnel**
- **Validation d'une transaction en différentiel**
- **Validation d'une transaction en place avec journalisation**
- **Mécanismes de reprise après panne**

Types de pannes liés au transactionnel

Panne sur une action (plusieurs fois par minute) :
- survient lorsqu'une commande au SGBD est mal exécutée
- est détectée par le système qui retourne un code erreur au programme applicatif qui peut traiter alors cette erreur et continuer la transaction

Panne de transaction (transaction failure) (plusieurs fois par minute) :
- survient en cours d'exécution d'une transaction ne pouvant continuer du fait d'une erreur de programmation, accès concurrents, dead-lock, violation d'une contrainte d'intégrité, …
- nécessité de **défaire** les mises à jour effectuées puis **relancer** la transaction

Panne système (system failure) (plusieurs fois par mois) :
- perte de la mémoire centrale
- nécessite l'arrêt du système et son redémarrage
- toutes les transactions en cours doivent être **défaites**

Panne mémoire secondaire (media failure) (rare, une fois par an):
- survient suite à une défaillance matérielle ou logicielle impliquant de mauvaises écritures
- une partie de la mémoire secondaire est perdue

Panne réseau (communication failure) (plusieurs fois par mois) :
- survient suite à un problème dans le réseau

Objectifs de la résistance au panne

Objectif général : Minimiser le travail perdu tout en assurant un retour à des données cohérentes de la base après panne

Unité de reprise :

- En général la **transaction** est l'unité de traitement atomique ou « unité de reprise »
- Parfois unité de reprise plus petite définie par l'explicitation de **points de reprise** (savepoint, commitpoint)

Moyens utilisés : Assurer l'atomicité de la transaction

- Validation d'une transaction (transaction commitment) : intégration définitive de toutes les mises à jour de la transaction
- Annulation d'une transaction (transaction abort) : annulation de toutes les mises à jour effectuées par la transaction
- Reprise de transaction (transaction redo) : refait toutes les mises à jour d'une transaction précédemment annulée

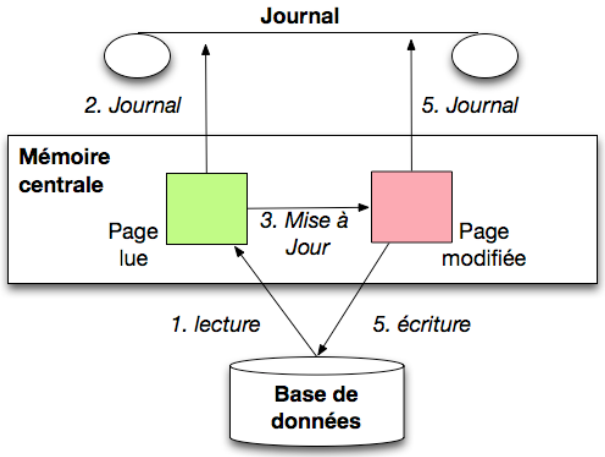
Validation d'une transaction

Lors de sa validation, toutes les mises à jour d'une transaction sont prises en compte ou aucune (atomicité de la transaction) :

2 techniques de base :

- **Mises à jour en place (stratégie DO-UNDO) :**
  - Règle du WAL (Write Ahead Logging)
  - Toute mise à jour (MAJ) est précédée d'une écriture dans un **journal d'images avant** permettant d'invalider cette MAJ
- **Mises à jour en différentiel (stratégie DO-REDO) :**
  - Les données sont **modifiées en pages différentielles** (non en place)
  - Les pages modifiées sont recopiées dans de **nouvelles pages** écrites dans de **nouveaux emplacements dans la base** et y seront intégrées lors de la validation (COMMIT)
  - Technique des **pages « ombres » (shadow pages)**

Principe de la journalisation



Ecriture d'abord dans le journal, puis dans la BD

Journaux et gestion des transactions

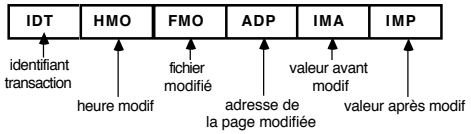
Méthode la plus classique pour la validation, l'annulation et la reprise de transactions

Journal des images AVANT:

- contient les valeurs d'enregistrement **avant mises à jour (images avant)** dans l'ordre des modifications avec les identifications des transactions modifiantes, précise les états des transactions (début, validation, annulation)
- permet de **défaire** (**UNDO**) les mises à jour effectuées par une transaction

Journal des images APRES:

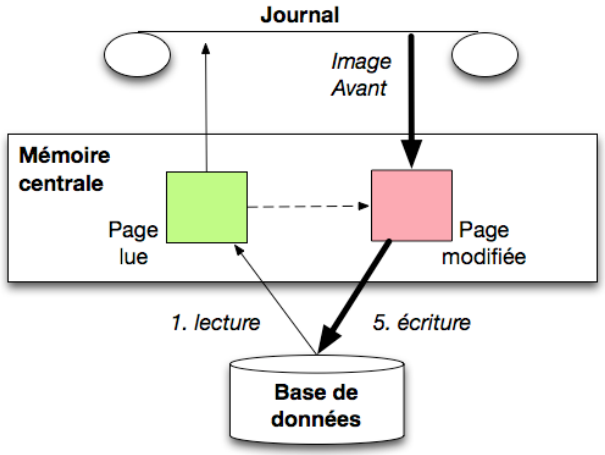
- contient les valeurs d'enregistrement **après mises à jour (images après)**, les fins de transactions (COMMIT ou ROLLBACK)
- permet de **refaire (REDO)** les mises à jour effectuées par une transaction



Ces 2 journaux peuvent être unifiés dans un même fichier

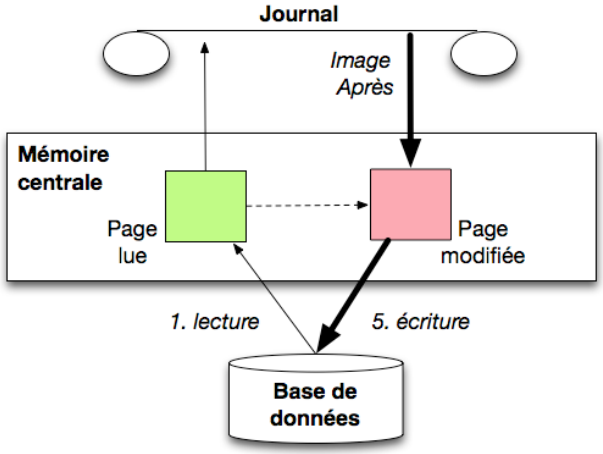
Usage du journal des images avant : UNDO

Images avant utilisées pour **défaire** les mises à jour : **UNDO**



Usage du journal des images après : REDO

Images après utilisées pour **refaire** les mises à jour : **REDO**

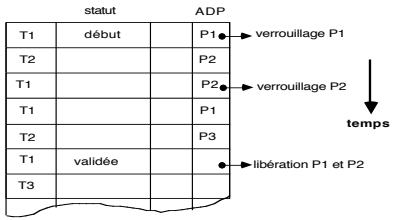


Validation de transaction par écriture en place (stratégie DO-UNDO)

- **Mises à jour en place** :
  - elles sont effectuées directement dans la base (en place)
  - immédiatement ou dès que possible
  - avant la validation de la transaction
- **Utilisation des images avant :**
  - copie avant mise à jour,
  - utilisée pour défaire en cas de panne

Validation de transaction par écriture en place (stratégie DO-UNDO)

- **Atomicité de la transaction assurée par le journal :**
  - les mises à jour de la transaction sont :
    - écrites dans le journal
    - restent invisibles aux autres transactions tant qu'elles ne sont pas validées par verrouillage des pages
    - deviennent visibles aux autres transactions quand la transaction est validée (écriture "transaction validée" dans le journal)

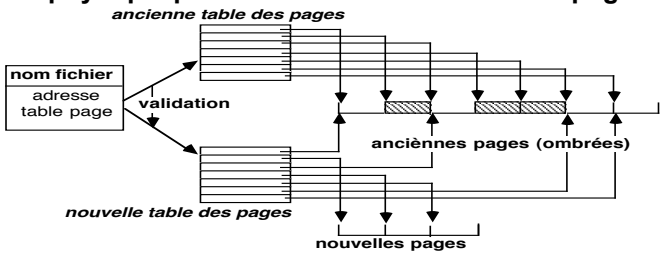


Validation de transaction par écriture en différentiel (stratégie DO-REDO)

- **Mises à jour en différentiel :**
  - Les données sont modifiées en **page différentielle** (non en place, cad pas sur les pages initiales)
- **Les pages modifiées :**
  - sont recopiées dans de **nouvelles pages**,
  - écrites dans de **nouveaux emplacements** dans la **base**
  - seront **intégrées dans la base** lors de la **validation** (COMMIT)

Validation de transaction par écriture en différentiel (stratégie DO-REDO)

- **Atomicité de la transaction assurée par la technique des pages « ombres » (shadow pages) :**
  - les nouvelles pages (après modifications) = **pages « différentielles »**
  - les anciennes pages (avant modifications) = **pages « ombres »**
  - avant toute lecture le SGBD consulte les pages différentielles validées et non encore physiquement intégrées à la base (lourd)
- **Intégration physique par basculement de la table de pages :**

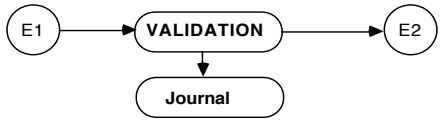


ceci pour chaque transaction et pour chaque fichier de la base modifié par la transaction …

Protocole de validation des transactions

Validation d'une transaction :

- nouvel état cohérent de la BD
- consignée dans le journal



Annulation d'une transaction ayant conduit à des mises à jour :

- difficile
- utilise des images avant du journal (utilisation du journal en remontant le temps)
- nécessité d'annuler toutes les transactions ayant lu des données écrites par cette transaction.

Protocole de validation généralement en 2 étapes :

Etape 1 : Préparation de la validation : *images avant et après enregistrées dans journal cf précédemment*

Etape 2 : Réalisation de la validation (ou annulation) : *analyse du journal, si ok --> validation.*

Gestion du journal

- Journaux avant et après sont unifiés (parfois même fichier)
- Écrits dans un tampon en mémoire et vidé sur disque en début de COMMIT
- Structure type d'un enregistrement du journal :
  - N° transaction (TrId) ; Type enregistrement {début, update, insert, commit, abort} ; TupleId ; [Attribut modifié, Ancienne valeur, Nouvelle valeur] ...
  - Problème de taille : on tourne sur N fichiers de taille fixe (ex : fichiers « vrap around », …)

Problèmes sur le journal (très grave)

Tout ou une partie du journal est **perdu** :

- **2° copie du journal** sur autre mémoire secondaire ?
- **écrire des transactions spéciales** chargées de tester la cohérence de la base et de compenser les effets de mises à jour malheureuses

La gestion des buffers

- Bufferisation des journaux : on écrit le journal lorsqu'un buffer est plein ou lorsqu'une transaction commet
- Bufferisation des bases : on modifie la page en mémoire, le vidage sur disque s'effectue en différé (processus E/S)
- Synchronisation journaux / base : le journal doit toujours être écrit avant modification de la base !

Commits bloqués

Pour éviter 3 E/S pour 1 :

- Le système reporte l'enregistrement des journaux au COMMIT
- Il force plusieurs transactions à commettre ensemble
- Il fait attendre les transactions au commit afin de bloquer un buffer d'écriture dans le journal

Permet d'améliorer les performances lors des pointes sans faire attendre trop sensiblement les transactions

Sauvegarde

- Sauvegarde périodique de la base :
  - toutes les heures, jours, ...
  - doit être effectuée en parallèle aux mises à jour
- Un Point de Reprise (checkpoint) est écrit dans le journal pour le synchroniser par rapport à la sauvegarde :
  - permet de situer les transactions effectuées après la sauvegarde
- Pose d'un point de reprise :
  - écrire les buffers de journalisation (Log)
  - écrire les buffers de pages (DB)
  - écrire un record spécial "checkpoint" dans le journal

Mécanismes de reprise après panne

Reprise normale :

Après arrêt normal de la machine,

- Un point de reprise système est écrit en fin du journal
- Restauration du contexte d'exécution sauvegardé du point de reprise

Reprise à chaud :

Après panne système :

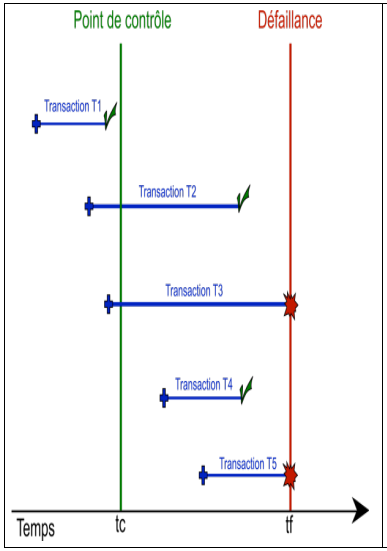
- Perte mémoire centrale
- Pas de perte mémoire secondaire

Reprise à froid :

Lorsque :

- panne de mémoire secondaire, perte d'une partie des données
- ou quand la base est devenue incohérente

Reprise à chaud : 5 cas possibles



- **Transactions de type T1** : sont terminées avant tc. Elles n'interviennent pas dans le processus de reprise.
- **Transactions de type T2** : ont débuté avant tc et se sont terminées entre tc et tf. Elles devront être rejouées (il n'est pas sûr que les données qu'elles manipulaient aient été correctement inscrites en mémoire centrale, puisque après tc, or le COMMIT impose la durabilité).
- **Transactions de type T3** : ont débuté avant tc, mais n'était pas terminées à tf. Elles devront être annulées (pas de COMMIT).
- **Transactions de type T4** : ont débuté après tc et se sont terminées avant tf. Elles devront être rejouées.
- **Transactions de type T5** : Elles ont débuté après tc et ne se sont pas terminées. Elles devront être annulées

Reprise à chaud : procédure générale

1. **Recherche dans journal du dernier point de reprise**
2. **Traitement (en avant) du journal du point de reprise vers la panne :**

   Permet de connaître les transactions validées et les transactions pas encore validées

3. **Traitement du journal en arrière :**

   Défaire les transactions non validées (UNDO)

4. **Traitement du journal en avant :**

   Refaire les transactions validées (REDO)

Reprise à froid : procédure

1. **Utilisation d'une sauvegarde cohérente de la base + le journal des activités qui ont suivies**
2. **Le système retrouve le checkpoint associé**
3. **Il ré-applique toutes les transactions validées** depuis ce point (for each committed Ti : REDO (Ti)) – application des **images après** à la **sauvegarde**
4. **En général enchaîne sur une reprise à chaud**

4. Transactions réparties

- **Transactions réparties**
- **Commit à 2 phases**
- **Commit à 3 phases**

Transactions réparties (1)

Transaction répartie ou "globale" = unité atomique d'interaction et de maintien de la cohérence, composée d'entités coopérantes s'exécutant sur des systèmes différents

Objectif : Garantir que toutes les mises à jour d'une transaction sont exécutées sur tous les sites ou qu'aucune ne l'est

Exemple 1: cas de panne …

Soit 2 sites, site 1 et site 2, une transaction T qui effectue un transfert de la somme X du compte A (site 1) vers le compte B (site 2) :

DEBUT

- site 1: A = A - X
- site 2: B = B + X PANNE --> INCOHERENCE DONNEES

FIN

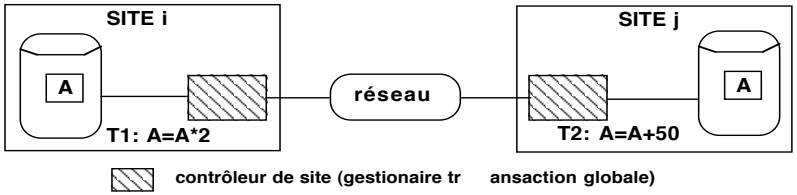
Problème :

Le contrôle est réparti : chaque site peut décider de valider ou d'annuler ...

Transactions réparties (2)

Exemple 2: ordre de transactions …

Soit T1 et T2 initialisées sur 2 sites différents gérant une entité dupliquée A:



Supposons que :

1. le site i fasse l'opération locale A = A*2 +100 (T1)
2. le site j fasse l'opération A = (A+50)*2 (T2)
3. le site i fasse l'opération locale A = (A+50)*2 (T1)
4. le site j fasse l'opération A = A*2 +100 (T2)

le contenu des 2 occurrences de l'entité A est devenu incohérent car **l'ordre** de traitement des transactions est différent

=> besoin de **coordonner les 2 transactions T1 et T2** pour assurer un verrouillage global et un maintient de la cohérence : **Protocoles spécifiques de COMMIT en 2 ou en 3 phases**

Commit en 2 Phases

Tous les sites terminent (COMMIT) ou annulent (ABORT) en respectant :

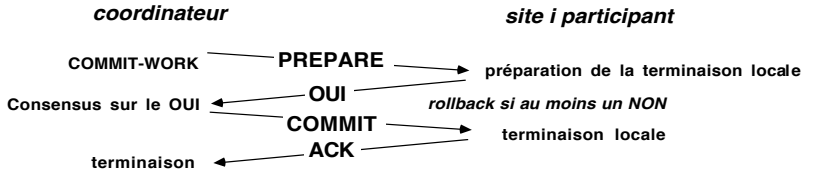
1- chaque site doit arriver à la même décision de terminaison ou d'annulation
2- chaque site a un droit de véto et l'unanimité requise pour décider une terminaison
3- chaque site ne peut plus annuler sa décision une fois qu'une décision globale est prise

Protocole à 2 phases : commande COMMIT en 2 phases :

Phase 1 : Préparer à écrire les résultats des mises à jour dans la BD (contrôle centralisé)
Phase 2 : Écrire ces résultats dans la BD

Protocole avec un site coordinateur :

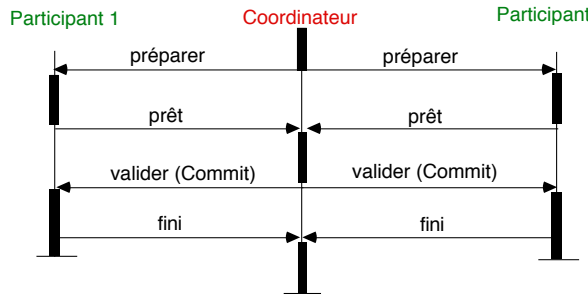
Coordinateur : composant système d'un site qui applique le protocole
Participant : composant système d'un autre site participant à l'exécution de la transaction



Cas Favorable

Validation en 2 étapes réussite :

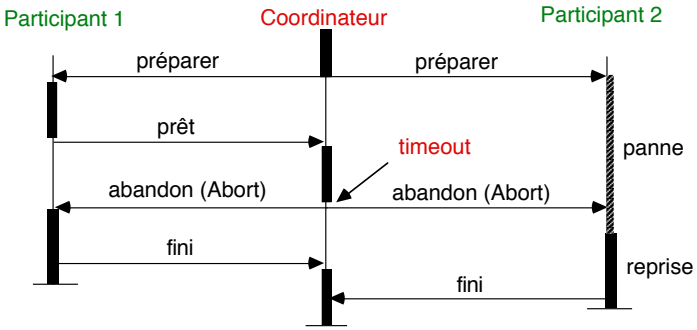
1 – le coordinateur demande aux sites s'ils sont prêts à « commettre » leurs MàJ
2 – si les sites sont prêts, le coordinateur leur donne ordre de faire leurs Commit
3 – si un participant n'est pas prêt, ou ne répond pas, le coordinateur demande à tous les sites de défaire leurs transactions (Abort)
4 – qd la transaction est terminée, le site envoie un acquittement (fini) au coordinateur



Cas Défavorable (1)

Validation en 2 étapes avec panne totale du site participant 2 :

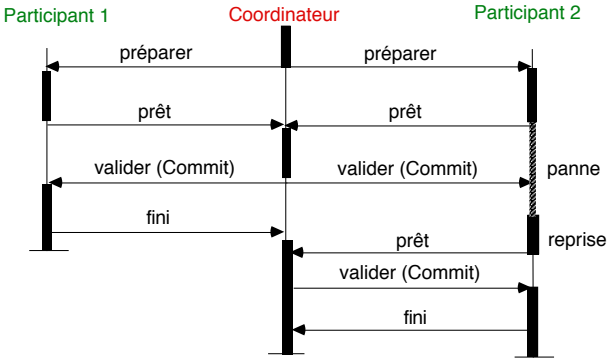
1 – le site 2 est en panne et ne peut répondre au coordinateur
2 – le coordinateur annule la transaction (ordre Abort) - une absence de réponse = refus
3 – le site annule la transaction et la reprendra (reprise)



Cas Défavorable (2)

Validation en 2 étapes avec panne partielle du site participant 2 :

1 – le site 2 est en panne après avoir répondu favorablement au coordinateur
2 – le coordinateur envoie le message Commit qui n'est pas reçu par le site
3 – après reprise le site 2 est prêt et en informe le coordinateur
4 – les 2 transactions sont finalement validées



Transactions bloquées

Que faire en cas de doute ?

- **Demander l'état aux autres transactions : STATUS**
  - conservation des états nécessaires
  - message supplémentaire
- **Forcer la transaction locale : ABORT**
  - toute transaction annulée peut être ignorée
  - cohérence garantie avec un réseau sans perte de message
- **Forcer la transaction locale : COMMIT**
  - toute transaction commise peut être ignorée
  - non garantie de cohérence avec le coordinateur

Commit en 3 Phases (1)

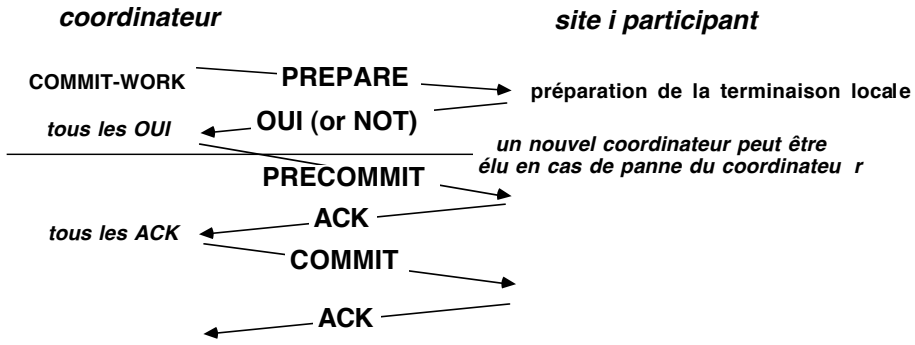
Commit à 3 phases :

- En cas de time-out en état “Prêt”, le participant n’est pas bloqué
- Il permet d’éviter les blocages
- Il intègre une phase de terminaison en cas de panne du coordinateur

Messages du commit à 3 phases :

- *Prepare*
- *Prepare to Commit (Precommit)*
- *Global-Commit*
- *Global-Abort*

Commit en 3 phases (2)



ACK = fini (Acknowledge)

5. Modèles étendus de gestion des transactions

- **Points de sauvegarde**
- **Transactions Imbriquées**
- **Sagas**
- **Langage de Contrôle d'Activités**
- **Moniteurs transactionnels**

Modèles étendus

- Applications longues composées de plusieurs transactions coopérantes
- Seules les mises-à-jour sont journalisées
- Si nécessité de défaire une suite de transactions:
  - contexte ad-hoc dans une table temporaire
  - nécessité d'exécuter des compensations

- **Quelques modèles étendus:**
  - **Points de sauvegarde**
  - **Transactions Imbriquées**
  - **Sagas**
  - **Langage de Contrôle d'Activités**
  - **Moniteurs transactionnels**

Points de Sauvegardes

Introduction de points de sauvegarde intermédiaires

(**savepoint**, **commitpoint**)

Begin_Transaction

" update
" update
" **savepoint** ← Non perte du contexte
" update
" update

unité d'oeuvre
unité d'oeuvre

Commit

Transactions Imbriquées [Moss 85]

Principe :

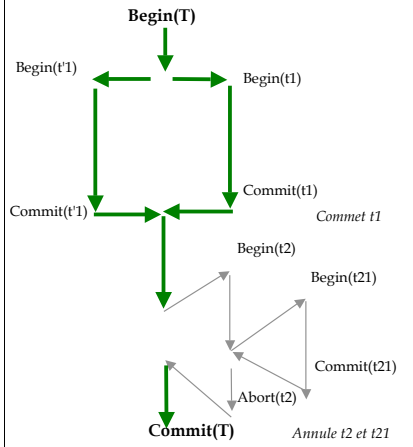
- Chaque transaction est découpée récursivement en sous-transaction : **arbre de transaction**
- Une transaction fille est lancée par sa mère par exécution d'un Begin
- Chaque transaction fille se termine par un Commit ou un Abort et peut être refaire (Redo) ou défaire (Undo) – journaux hiérarchisés

Intérêts :

- Obtenir un mécanisme de reprise multi-niveaux
- Permettre de reprendre des parties logiques de transactions
- Faciliter l'exécution parallèle de sous-transactions

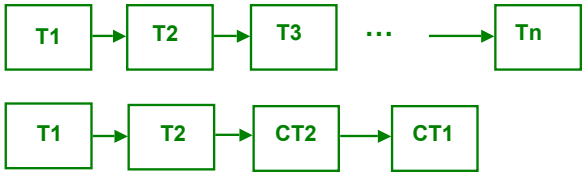
Schéma :

- Reprises et abandons partiels
- Possibilité d'ordonner ou non les sous-transactions



Sagas [Garcia-Molina 87]

- Permet de **supporter des transactions longues sans trop de blocage**
- **Une saga** = **séquence ordonnée de sous-transactions** ACID {T1, …Tn} associée à une séquence de transactions **compensatrices** {CT1, …CTn}
- Tout abandon d'une sous-transaction provoque l'abandon de la saga complète et en cas de panne du groupe, on exécute les compensations



(CT1 = transaction compensatrice de T1)

L'intérêt est de **relâcher le principe d'isolation** :

- chaque transaction composante relâche ses verrous dès qu'elle est terminée, une autre saga peut alors voir les résultats
- l'annulation de la saga doit provoquer l'annulation de l'autre saga, ceci par un enchaînement de transactions compensatrices

Langages de contrôle d'activités

Introduction d'un langage de contrôle de transactions permettant de définir des activités:

Activités = collection de transactions avec enchaînement conditionnel ou compensation

Exemple:

l'activité « réservation de vacances » composée des transactions :

- T1 : réservation avion
- T2 : réservation hôtel
- T3 : location voiture
- T1': location voiture (alternative de T1)
- CT1, CT2, … transactions compensatrices

Langage de contrôle d'activités :

- Possibilité de transactions vitales (ex: réservation hôtel)
- Langage du type : *If abort, If commit, Run alternative, Run compensation, …*

Intérêts :

- contexte persistant
- rollforward, rollback avec compensations
- flot de contrôle dépendant des succès et échecs (différencier les échecs systèmes des échecs de programmes)
- monitoring d'activités (état d'une activité, arrêt, ...).

Moniteurs transactionnels

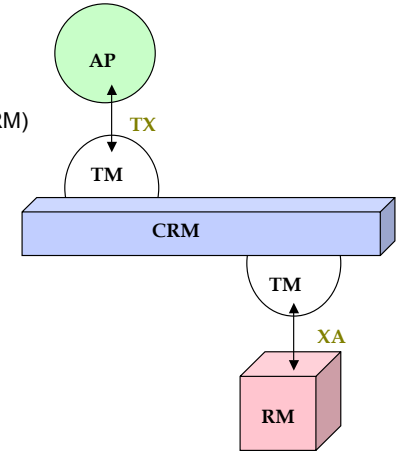
Objectifs :

- Supporter les **transactions ACID**
- Permettre un **accès continu aux données**
- Permettre une **reprise rapide du système en cas de panne**
- Assurer une **sécurité d'accès**
- Garantir des **performances optimisées** :
  - Dans le partage de connexions
  - Dans la réutilisation de transactions
- Permettre un **partage de charge**
  - Distribution de transactions
- Supporter des **bases hétérogènes**
- Permettre un **respect des normes et des standards**

Moniteur transactionnel : Modèle de l'X/OPEN

Modèle DTP de l'X/OPEN :

- Programme d'application (AP)
- Gestionnaire de transactions (TM)
- Gestionnaire de communication (CRM)
- Gestionnaire de ressources (RM)



Interfaces standards :

- TX = interface du TM
- XA = interface du RM
- intégration de TP

Types de RM :

- gestionnaire de fichiers
- SGBD
- périphérique

Interface applicative TX

- **tx_open** : ordonne au TM d'initialiser la communication avec tous les RM dont les librairies d'accès ont été liées à l'application.
- **tx_begin** : ordonne au TM de demander aux RM de débuter une transaction.
- **tx_commit** ou **tx_rollback** : ordonne au TM de coordonner soit la validation soit l'abandon de la transaction sur tous les RM impliqués.
- **tx_set_transaction_timeout** : positionne un " timeout " sur les transactions
- **tx_info** : permet d'obtenir des informations sur le statut de la transaction.

Interface ressource XA

- **xa_open** : ouvre un contexte pour l'application.
- **xa_start** : débute une transaction.
- **xa_end** : indique au RM qu'il n'y aura plus de requêtes pour le compte de la transaction courante.
- **xa_prepare** : lance l'étape de préparation du commit à deux phases.
- **xa_commit** : valide la transaction.
- **xa_rollback** : abandonne la transaction.

Principaux moniteurs

Encina de Transarc
- issu de CMU (1992), racheté par IBM
- construit sur DCE (OSF) pour la portabilité et la sécurité
- transactions imbriquées
- conformité DTP : Xa, CPI-C, TxRPC

Open CICS de IBM
- construit sur Encina (et DCE)
- reprise de l'existant CICS (API et outils)
- conformité DTP : Xa, CPI-C

Tuxedo de USL
- éprouvé (depuis 1984), à la base de DTP
- supporte l'asynchronisme, les priorités et le routage dépendant des données
- conformité DTP: Xa, Tx, XaTMI, CPI-C, TxRPC

Top End de NCR pour Unix
- produit stratégique d'AT&T
- respecte le modèle des composants DTP (AP, RM, TM, CRM)
- haute disponibilité
- conformité DTP: Xa, Xa+, Xap-Tp, Tx

…

6. Conclusion sur la gestion des transactions

Conclusion sur la gestion des transactions

- Pose des problèmes bien maîtrisés dans les SGBDR
- Met pour cela en œuvre des techniques complexes
- La concurrence complique la gestion de transactions
- Les transactions longues restent encore problématiques
- Constitue notamment un enjeu majeur pour le commerce électronique :
  - validation fiable
  - reprise et copies
  - partage de connections
  - partage de charge