Cours 2

La Gestion des Transactions

BDA
Master RT - 2018- 2019

OUARED Abdelkader

Plan du cours

- Le concept de transaction
- Fonctionnement du système de base de données
- Contrôle de concurrence
- Reprise après une panne

Traitement de fichiers (Pb)

Redondance et inconsistance des données

– certaines info se trouvent sur plusieurs fichiers

Difficulté d'accès aux informations non prévues

– nécessité d'écrire de nouveaux prog. d'accès

Dépendance : rep. interne / Applications

– changement de structure -> re-programmation des App

Atomicité et pb de concurrence

– erreur, pannes, accès concurrents -> introduisent des inconsistances

1960: les SGFs

➲Problème de surcharge ☹
➲Comment libérer le programmeur ?

⇨ SGBD ☺

- **1970**: Codd paper; les fondements des BDR
- **1980**: Les SGBD-R sur le marché

SGBD

SGBD (Système de Gestion de Bases de Données)

- ensemble de programmes qui gèrent la BD

Objectifs

– efficacité (très important)
– sécurité
– facilité d'utilisation

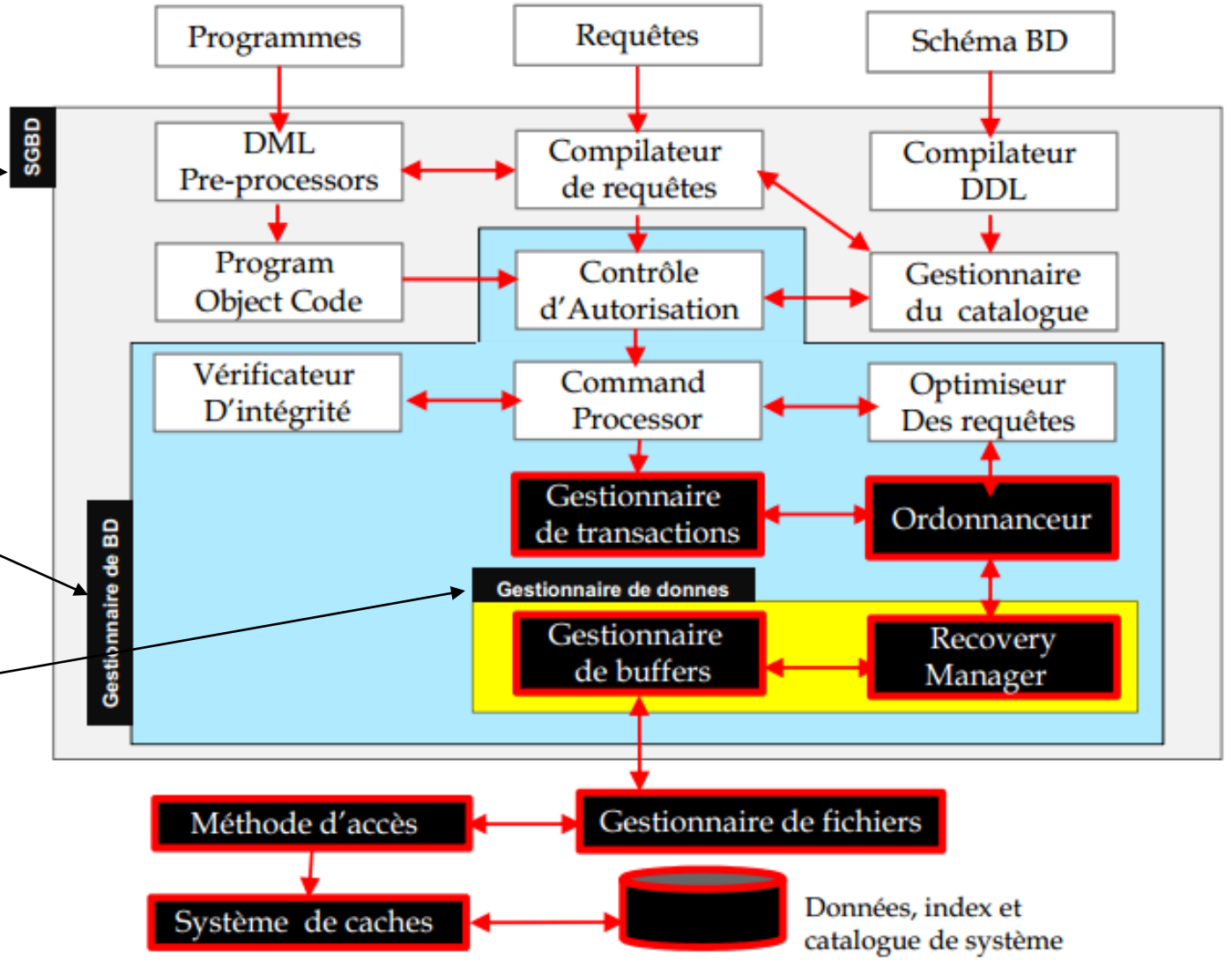
➲ **SGBD** : A quoi ca sert?
➲ simplifier la vie des utilisateurs et des programmeurs à gérer les bases de données BDs d'une manière efficace

Composantes d'un SGBD

SGBD

Gestionnaire de BD

Gestionnaire de données



Le concept de transaction

- **Définition:**
  - Une **transaction** est une séquence d'opérations (lecture/ écriture) qui forment une seule unité de travail.
  - Une transaction est souvent déclenchée par un programme d'application
    - commencer une transaction **START TRANSACTION**
    - accès à la base (lire/écrire)
    - calculs en MC
    - fin transaction (COMMIT ou ROLLBACK)

  Exemple: transfert d'argent d'un compte à un autre
- **Transaction**: <begin transaction> Opérations <end transaction>

Responsabilité du SGBD

- SGBD doit vérifier les propriétés règles **ACID** (*atomicity*, *consistency*, *isolation* et *durability*)
- Garantir l'exécution correcte des transactions
- Gérer l'exécution concurrente des transactions

Propriétés des Transaction ACID

Atomicité: Le fait qu'une transaction est indivisible, elle doit s'exécuter entièrement ou pas du tout

Cohérence: Une transaction doit préserver la cohérence de la base de données.

- contraintes d'intégrités
- règles métier définies
- règles complexes
- responsabilité du développeur/DBA

Propriétés des Transaction ACID

Isolation: Vu qu'une transaction est une unité indivisible, ces actions ne doivent pas être séparées par des opérations de base de données ne faisant pas partie de la transaction.

- pas **d'interférence** entre les transactions qui s'exécutent en même temps dans le système

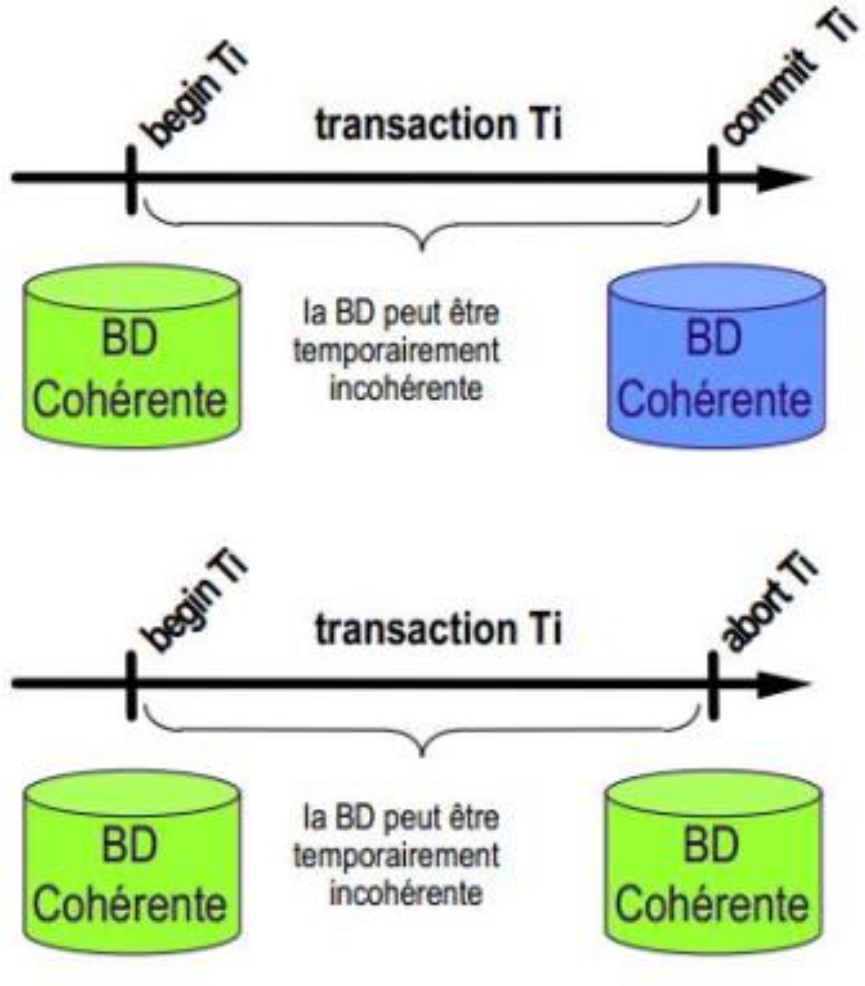
Durabilité: Même si le système exécute correctement les transactions, ceci ne servira pas à grand chose si une situation d'erreur fait que le système perd la trace des transactions terminées avec succès. Pour cela, **les actions des transactions doivent persister** même en cas des situations d'échec les plus graves.

- Gestionnaire de pannes : journalisation des opérations, algorithme REDO
- **Point fort des SGBD**, qui peuvent résister aux pannes, sans perdre de données et en restituant la base dans un état cohérent.

Propriétés des Transaction ACID

A retenir :

- **A**tomicité
  - tout ou rien
- **C**onsistance
  - cohérence sémantique
- **I**solation
  - pas de propagation de résultats non validés
- **D**urabilité
  - persistance des effets validés



Exercice 01

Transaction de transfert de 50$ à partir du compte A au compte B :

1. Lire (A)
2. A:= A - 50
3. Écriture (A)
4. Lire (B)
5. B : = B + 50
6. Écriture (B)

– **Atomicité** : ?
– **Cohérence** : ?
– **Durabilité** : ?
– **Isolation** : ?

Types de transaction

Type	Description
Langage de manipulation de données (LMD)	Un ensemble d'instructions Insert, Update, Delete
Langage de définition de données (LDD)	Une instruction Create, Alter, Drop. L'instruction Truncate fait aussi partie du LDD
Langage de contrôle de données (LCD)	Une instruction Grant ou Revoke

Deux principaux problèmes de transaction ..

Il y a deux principaux problèmes de transaction

– exécution simultanée de plusieurs opérations (concurrence)

– récupération après des pannes matérielles et des plantages du système (panne)

• Pour préserver **l'intégrité des données,** le SGBD doit se assurer que les propriétés **ACID** pour toute transaction

Problèmes rencontrés par l'exécution concurrente

perte des modifications

• 2 transactions accèdent au même élément d'une BD et l'exécution des opérations est décalée

• X=4, Y=8, N=2, M=3

T1: (joe)	T2: (fred)	X	Y
read_item(X);			
X:= X - N;			
	read_item(X);		
	X:= X + M;		
write_item(X);			
read_item(Y);			
	write_item(X);		
Y:= Y + N;			
write_item(Y);			

Résultat incorrect: : X=7 et Y=10
Résultat correct : X= 5 et Y=10

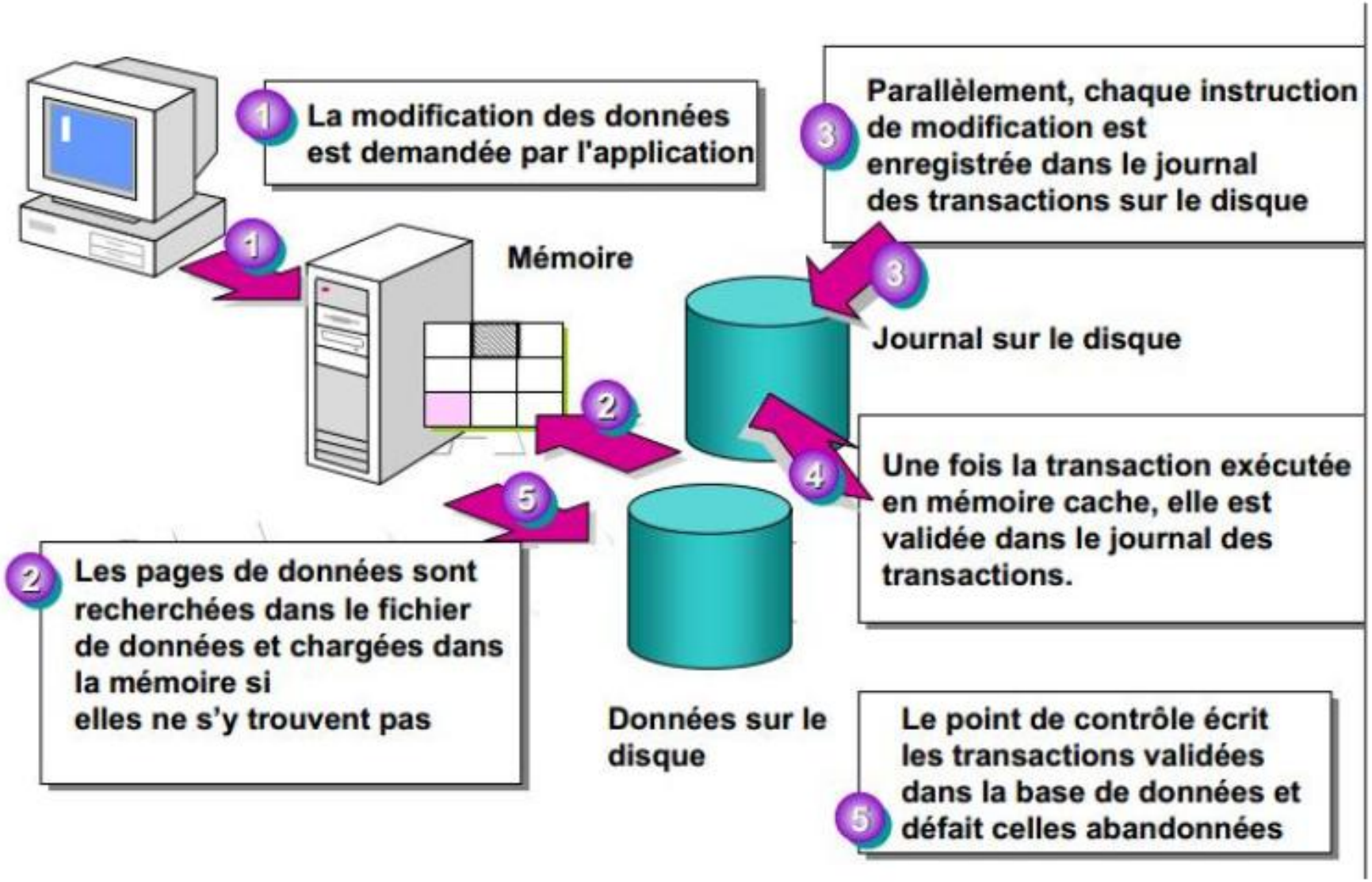
➲ Solution ??

Question ?

Comment maintenir ces propriétés ?

- **Reprise sur panne** (Atomicité, Durabilité)

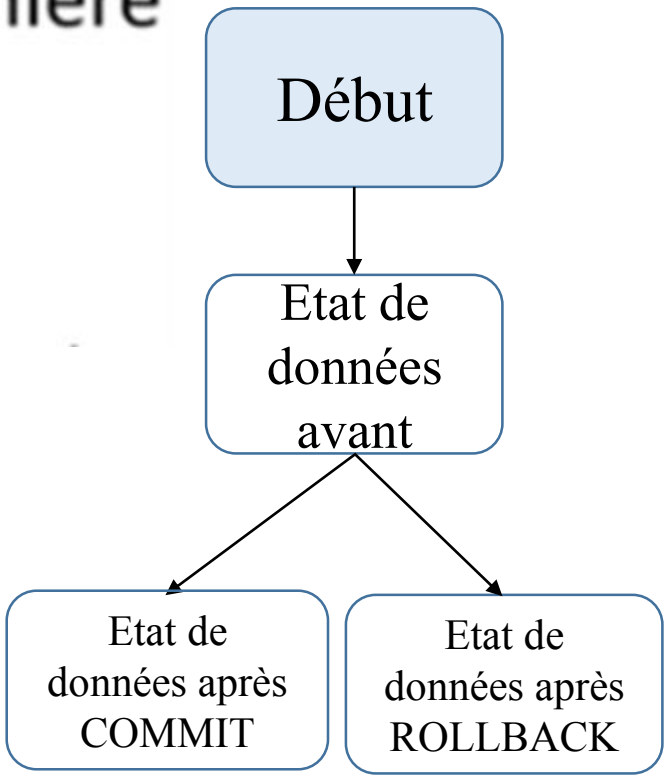
Gestion de la transaction



Transactions de base de données Début et fin

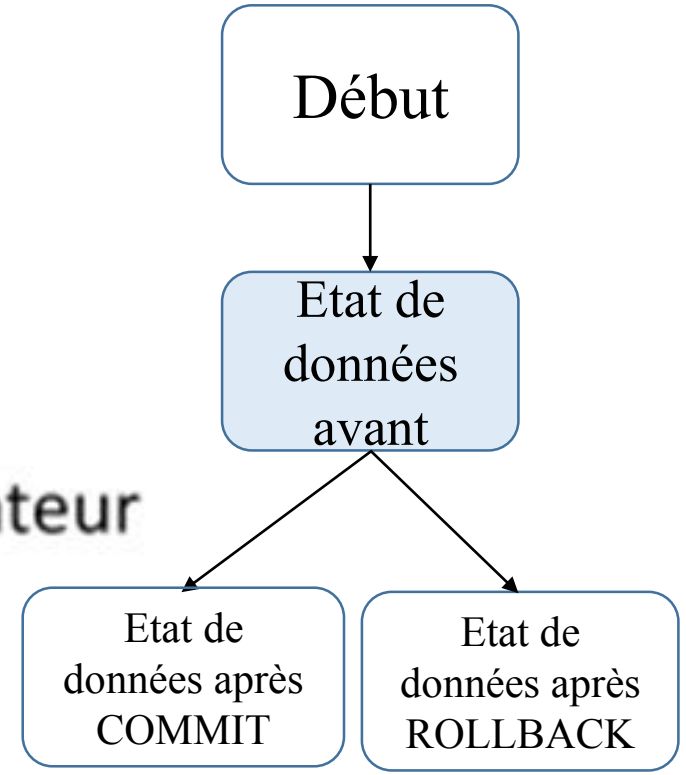
- Elles commencent avec l'exécution de la première instruction SQL LMD.
- Elles finissent lorsque l'un des événements suivants se produit :

• Une instruction COMMIT ou ROLLBACK est exécutée.



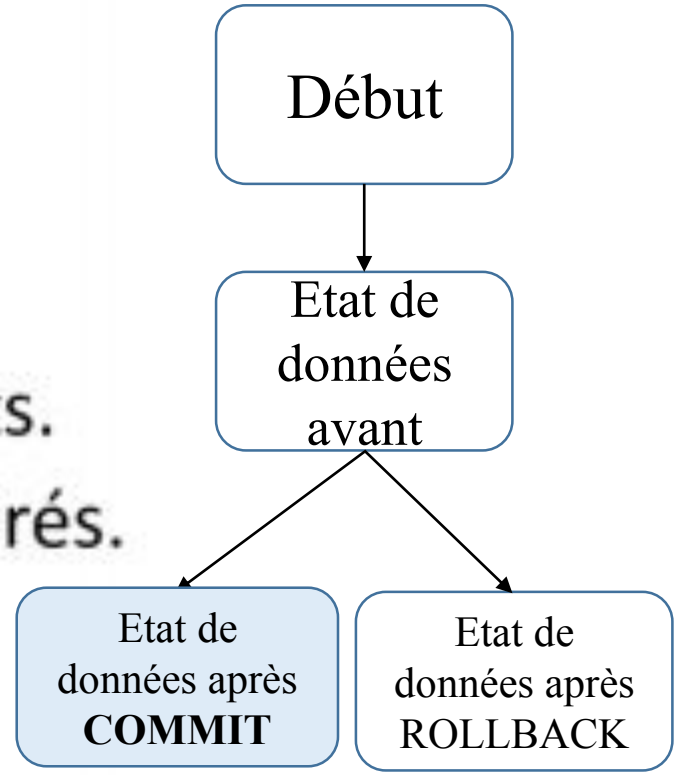
Etat des données avant exécution de l'instruction COMMIT ou ROLLBACK

- L'état antérieur des données peut être récupéré.
- L'utilisateur actuel peut visualiser les résultats des opérations LMD à l'aide de l'instruction `SELECT`.
- Les autres utilisateurs *ne peuvent pas* afficher les résultats des instructions LMD exécutées par l'utilisateur actuel.
- Les lignes affectées sont *verrouillées*. Les autres utilisateurs ne peuvent donc pas modifier ces lignes.



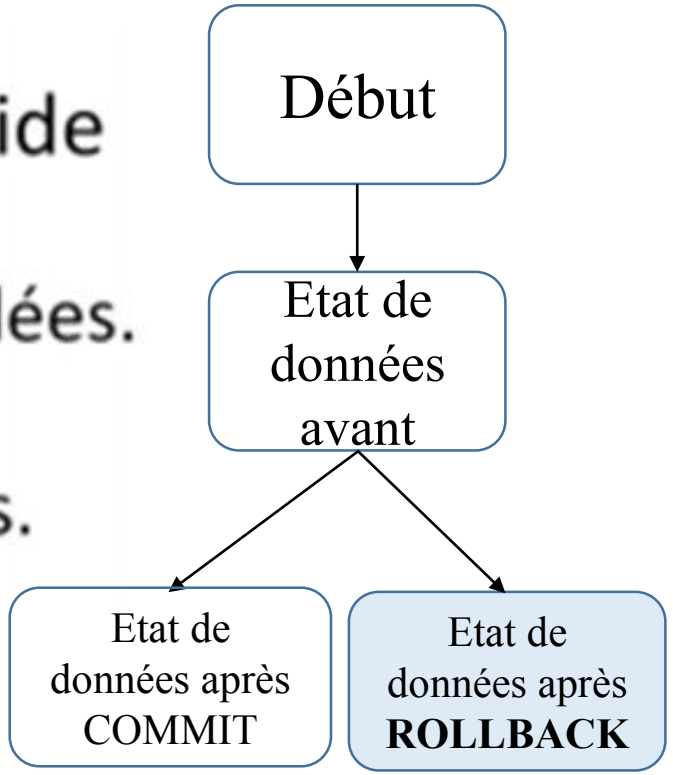
Etat des données après exécution de l'instruction COMMIT

- Les modifications apportées aux données sont enregistrées dans la base.
- L'état antérieur des données est écrasé.
- Tous les utilisateurs peuvent visualiser les résultats.
- Les verrous externes des lignes affectées sont libérés. Ces lignes peuvent alors être manipulées par les autres utilisateurs.

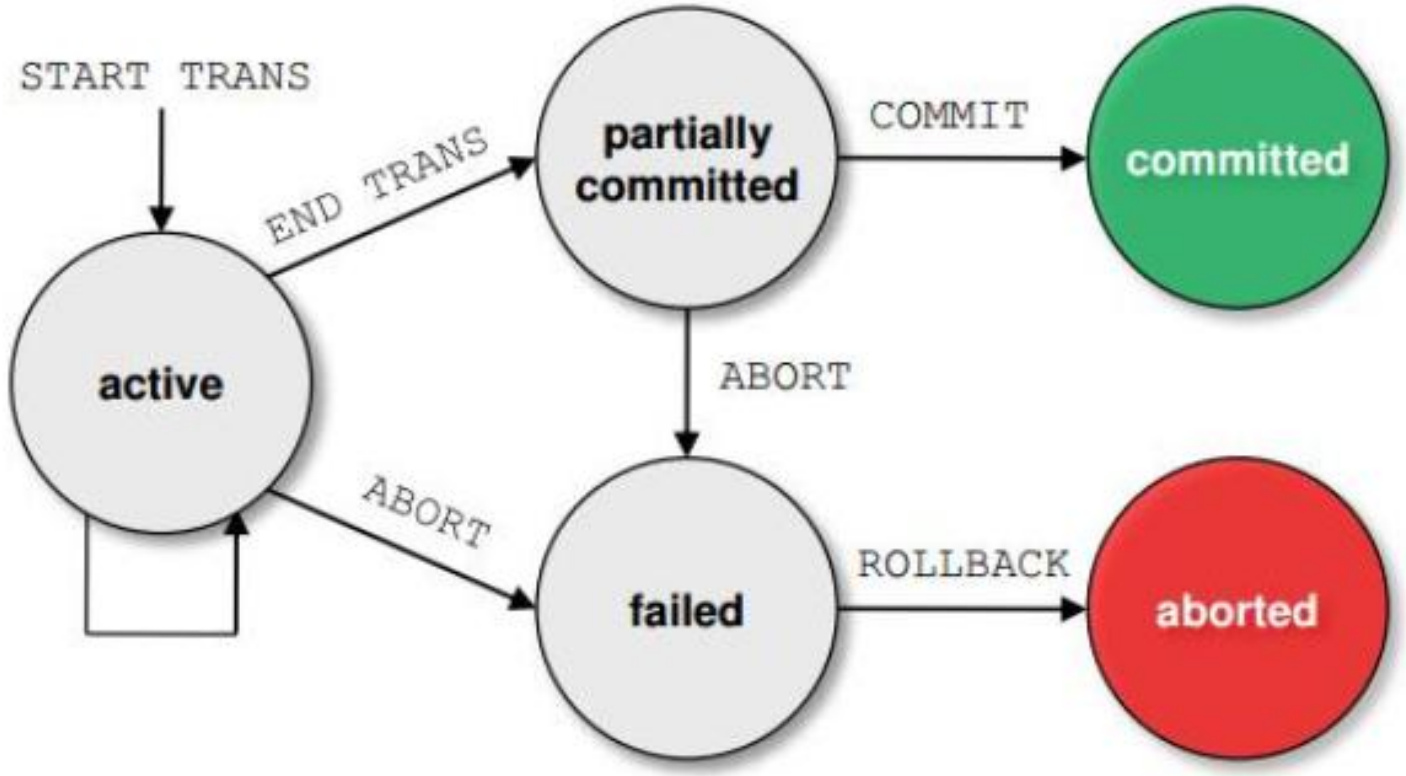


Etat des données après exécution de l'instruction ROLLBACK

- Annulez toutes les modifications en attente à l'aide de l'instruction `ROLLBACK` :
  - Les modifications apportées aux données sont annulées.
  - L'état antérieur des données est restauré.
  - Les verrous externes des lignes affectées sont libérés.



Etat d'une transaction



Etat d’une transaction

• **Actif** : état initial; transaction est dans cet état pendant l'exécution. Notons qu’une transition est interrompue uniquement si elle a été soit validée ou annulée .
• **Partiellement validée**: juste après l’exécution de la dernière opération . A ce point la défaillance est toujours possible puisque les modifications ont été faites uniquement dans la mémoire principale, une panne de matériel pourrait encore se produire.
• **Validée**: après l’exécution avec succès de la dernière opération

• **Echec**: après découverte qu'une exécution normale n’est plus possible
– Erreur de logique (par exemple mauvaise entrée), une erreur système (par exemple de blocage) ou plantage du système
• **Avortée (Abondonnée)**: après l'annulation d'une transactio
Après que toutes les modifications faites par la transaction soient annulées (Roll back). Autrement dit, la base de données restaure son état avant le début de la transaction. Deux options
– Ré-exécuter la transaction. Une transaction peut uniquement être redémarré par suite de certains matériels ou une erreur du logiciel. Un un redémarrage transaction est considérée comme une nouvelle transaction.
– Tuer la transaction

Exemple: SELECT...FOR UPDATE, NOWAIT

instruction SELECT

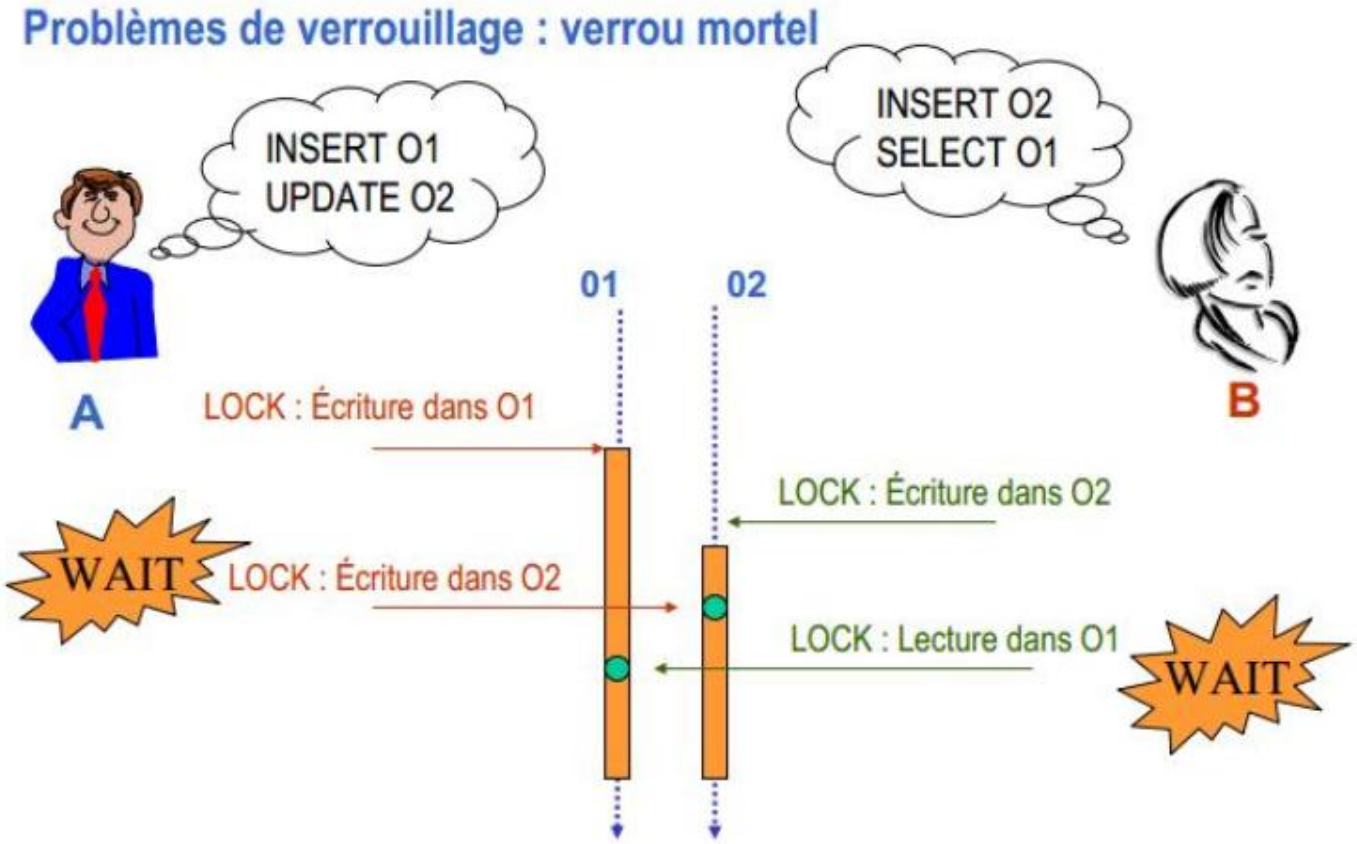
- Dans la table EMPLOYEES, verrouillez les lignes pour lesquelles job_id = SA_REP.

```
SELECT employee_id, salary, commission_pct, job_id
FROM employees
WHERE job_id = 'SA_REP'
FOR UPDATE
ORDER BY employee_id;
```

- Le verrou est libéré uniquement lorsque vous exécutez une instruction ROLLBACK ou COMMIT.
- Si l'instruction SELECT tente de verrouiller une ligne qui est déjà verrouillée par un autre utilisateur, la base de données attend que la ligne soit disponible pour renvoyer les résultats de l'instruction SELECT.

Problèmes du Verrouillage

Illustration



➲ Comment maintenir ces propriétés ?
Contrôle de concurrence (Isolation, Cohérence)

Les pannes

Erreur de Transaction

Erreurs logiques (opération incorrecte, condition non satisfaite, ...)

Interblocage, …

Crash système

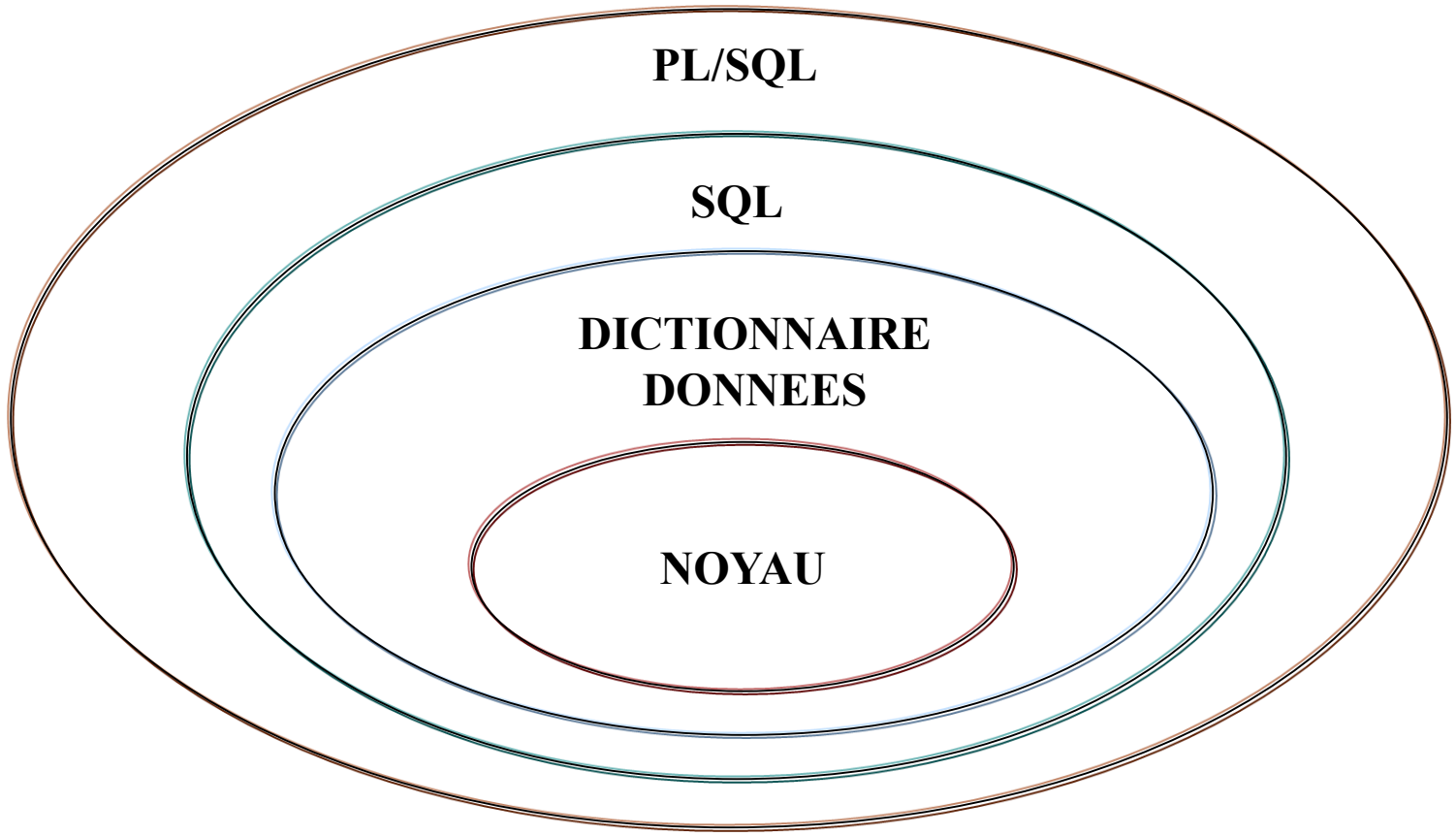
Pb d'alimentation, Reboot, …

Erreur de disque

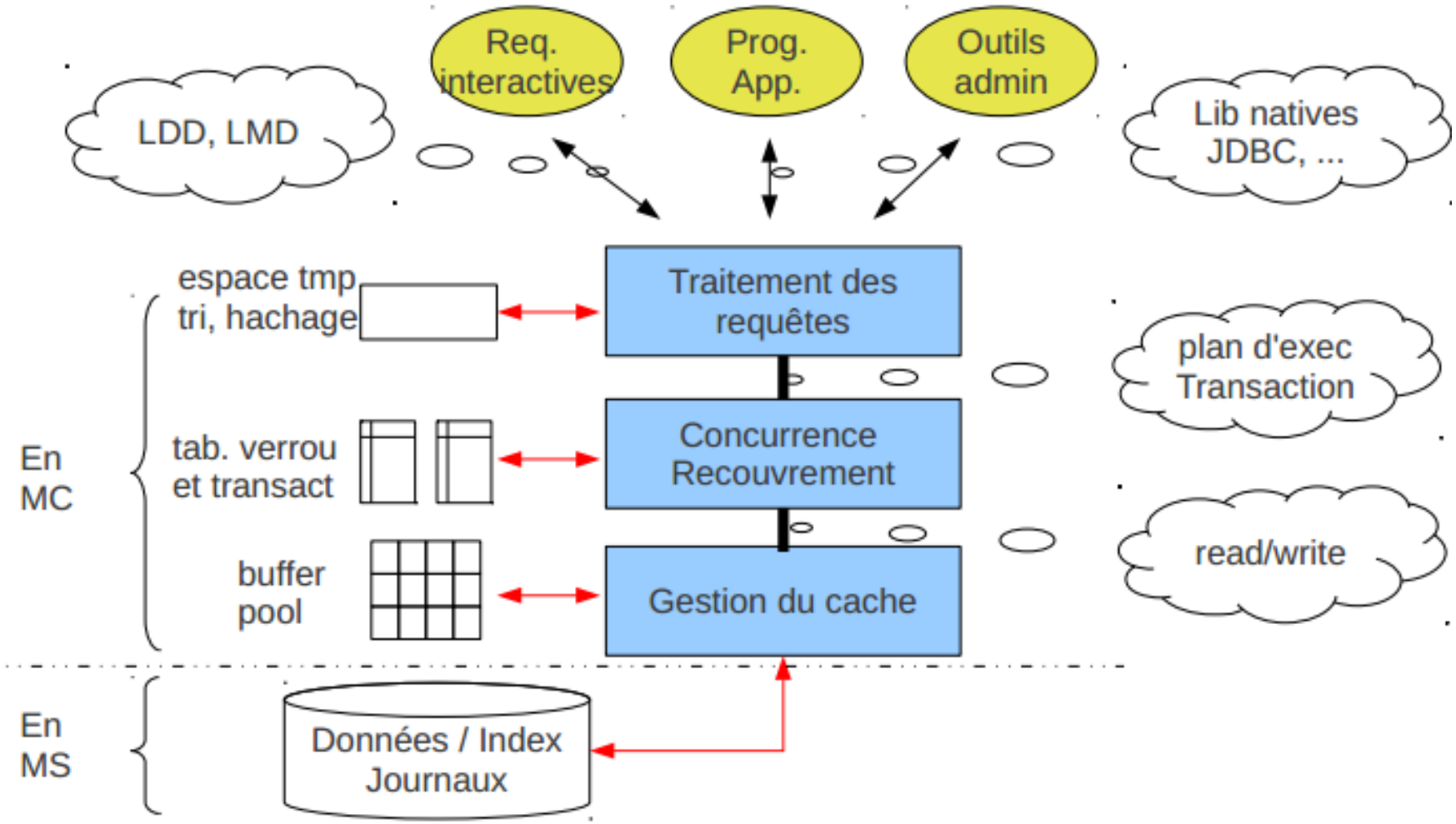
En partie ou en totalité

Contrôle de concurrence …

Organisation des Couches



Architecture système de base de données



Architecture système

Traitement de requêtes (Gestionnaire de transaction):

- Optimisation de requêtes (plan d'exécution)
- Petit espace de RAM pour réaliser des opérations < **join, tri** >

Contrôleur de concurrence:

- Envoie les opérations suivant un ordre sérialisable et strict

Gestionnaire de recouvrement :

- Gère le stockage et l'indexation
- Le Gestionnaire de recouvrement (GR) fait en sorte qu'il y ait toujours assez d'informations sur mémoire stable pour qu'une reprise correcte soit possible après l'occurrence d'une panne
  - **Les opérations du gestionnaire de recouvrement**
    read, *write*, *commit*, *rollback* et *restart* (en cas de reprise)
- **Retour en arrière:** Restaurer la BD dans le cas d'annulation de la transaction par user, ou le système(ABORT, COMMIT)

Architecture système

Gestionnaire de cache :

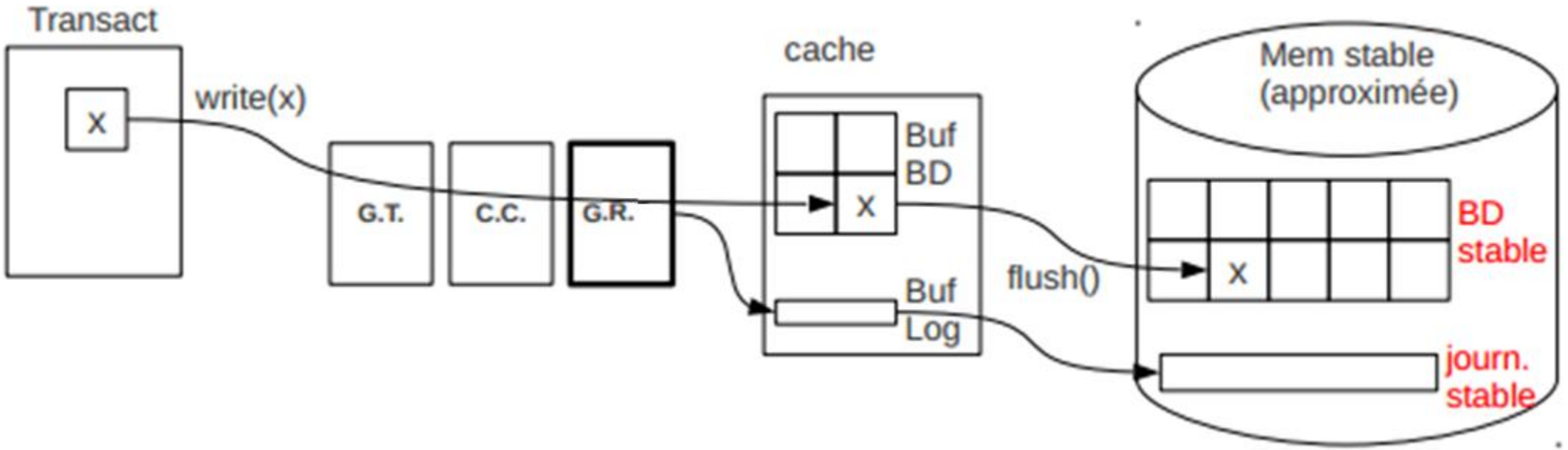
- La BD stable et le journal stable résident sur disque
- Les accès transitent par le cache en MC

- Synchronise la page de cache avec la page de la BD stable
- Choix de la page victime → besoin de l'espace !!

Les opérations du gestionnaire de cache

Fetch (pour obliger une lecture physique)
Flush (pour obliger une écriture physique)
Pin (pour bloquer une page dans le cache)
Unpin (pour débloquer une page du cache)

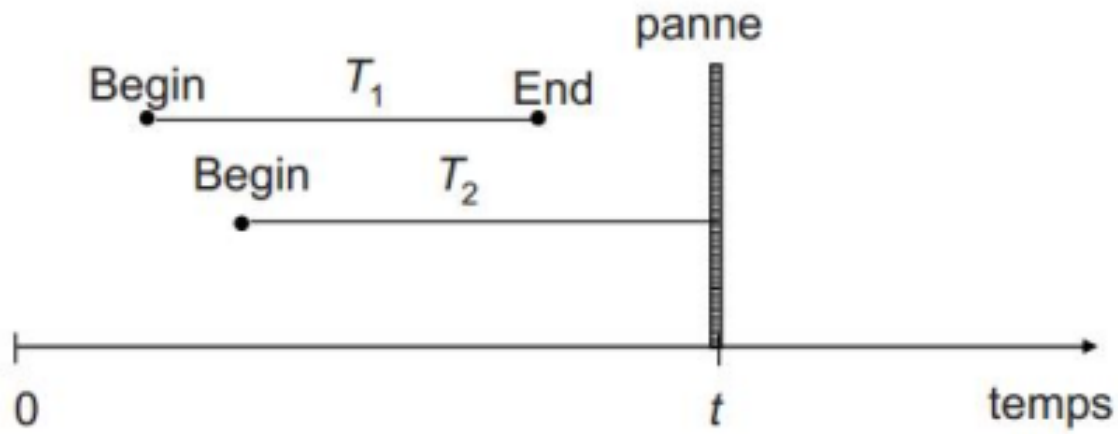
Accès aux données



Reprise après une panne

Pourquoi journaliser?

- Lors de la reprise



– toutes les mises-à-jour de T1 doivent être faites dans la BD

– aucune mise-à-jour de T2 ne doit être faite dans la BD

La journalisation

- Un journal est une séquence d'enregistrements décrivant les mises à jours effectuées par les transactions
- C'est l'historique d'une exécution sur fichier séquentiel
- Un journal est dit « **physique** » s'il garde la trace des modifications au niveau octet à l'intérieur des pages

  Ex: **<Ti, numPg, Depl, Long, img_avant, img_après>**

- Un journal est dit « **logique** » s'il garde la trace de la description de haut niveau des opérations de mise à jour

  Ex: « insérer le tuple x dans la table T et mettre à jour les index »

Utilisations du journal

Lors de l'annulation

remettre les images avant (Undo) en parcourant le journal en arrière

Lors de la validation

sauvegarder toutes les images après en mémoire stable (journal)

- optimisation : « group commit + precommit »

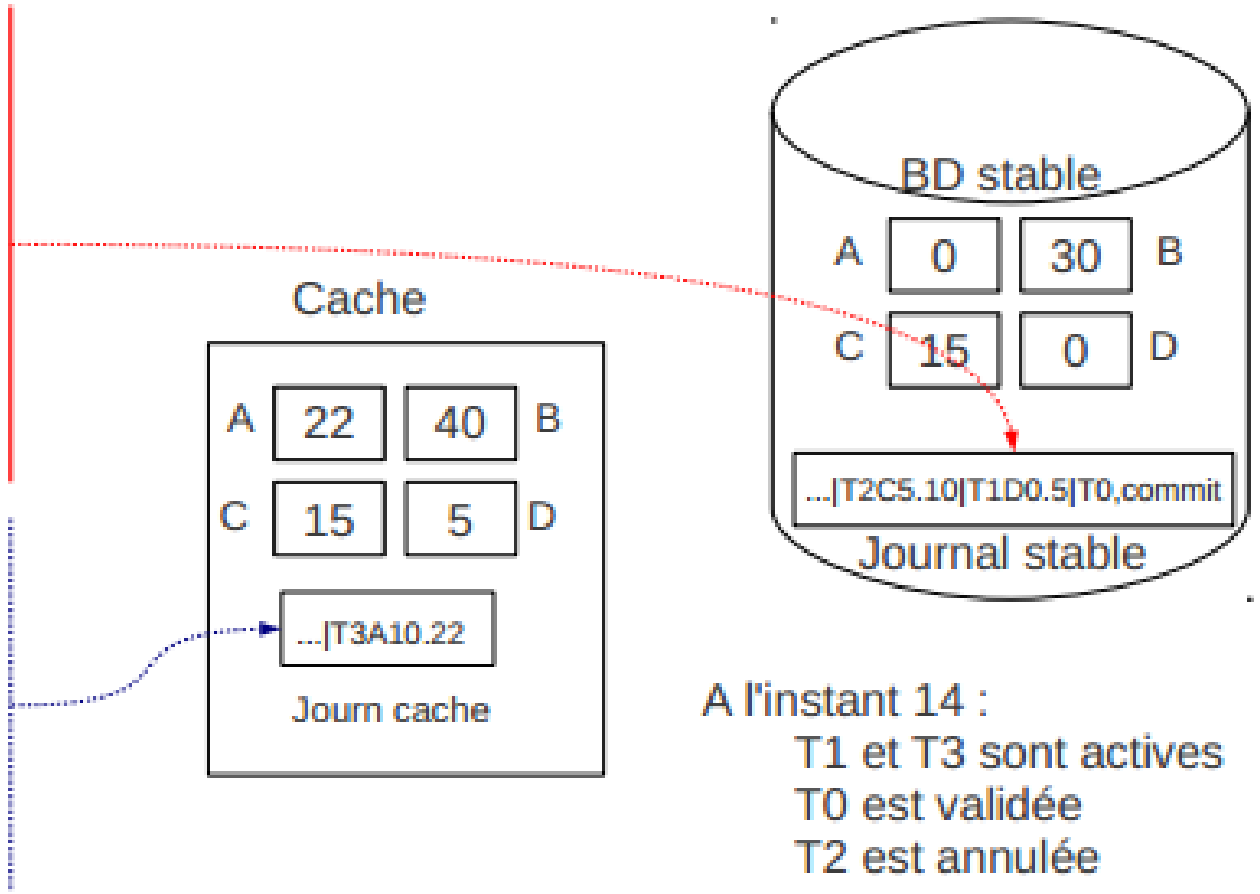
Lors d'une reprise après panne

Refaire l'historique en exécutant les opérations du journal du début jusqu'au moment de la panne (dernier enregistrement)

- très coûteux et beaucoup de travail inutile

Exemple

1 <T0, start>
2 <T0, A, 0, 10>
3 <T1, start>
4 <T0, B, 20,30>
5 <T2, start>
6 <T2, C,5 ,10>
7 <T1, D, 0, 5>
8 <T0, commit>
9 <T2, A, 10, 15>
10 <T1, B, 30, 40>
11 <T2, abort>
12 <T1, C, 5, 15>
13 <T3, start>
14 <T3, A, 10, 22>
15 ...



A l'instant 14 :
T1 et T3 sont actives
T0 est validée
T2 est annulée

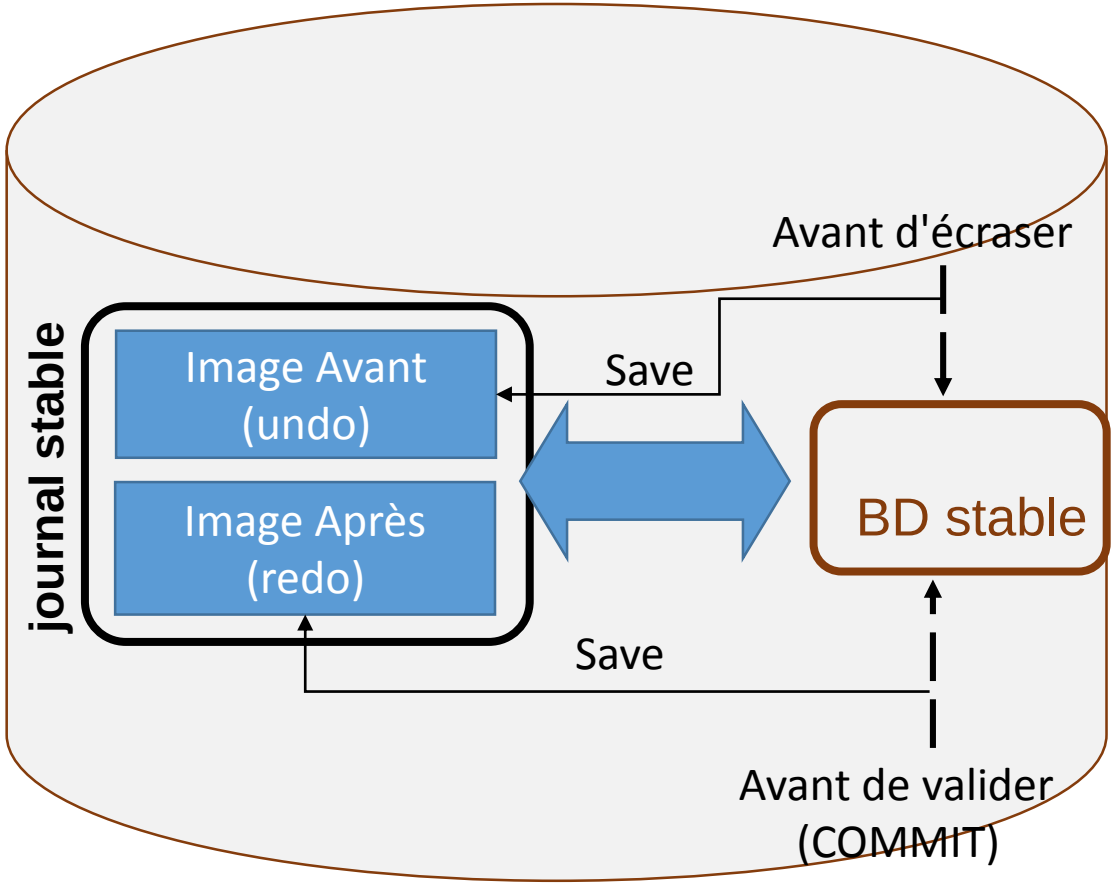
Règles du undo et du redo

Règle du Avant (Undo)

Avant d'écraser une ancienne valeur par une nouvelle, dans la BD stable, sauvegarder l'ancienne (l'image avant) dans un autre emplacement stable (dans le journal stable)

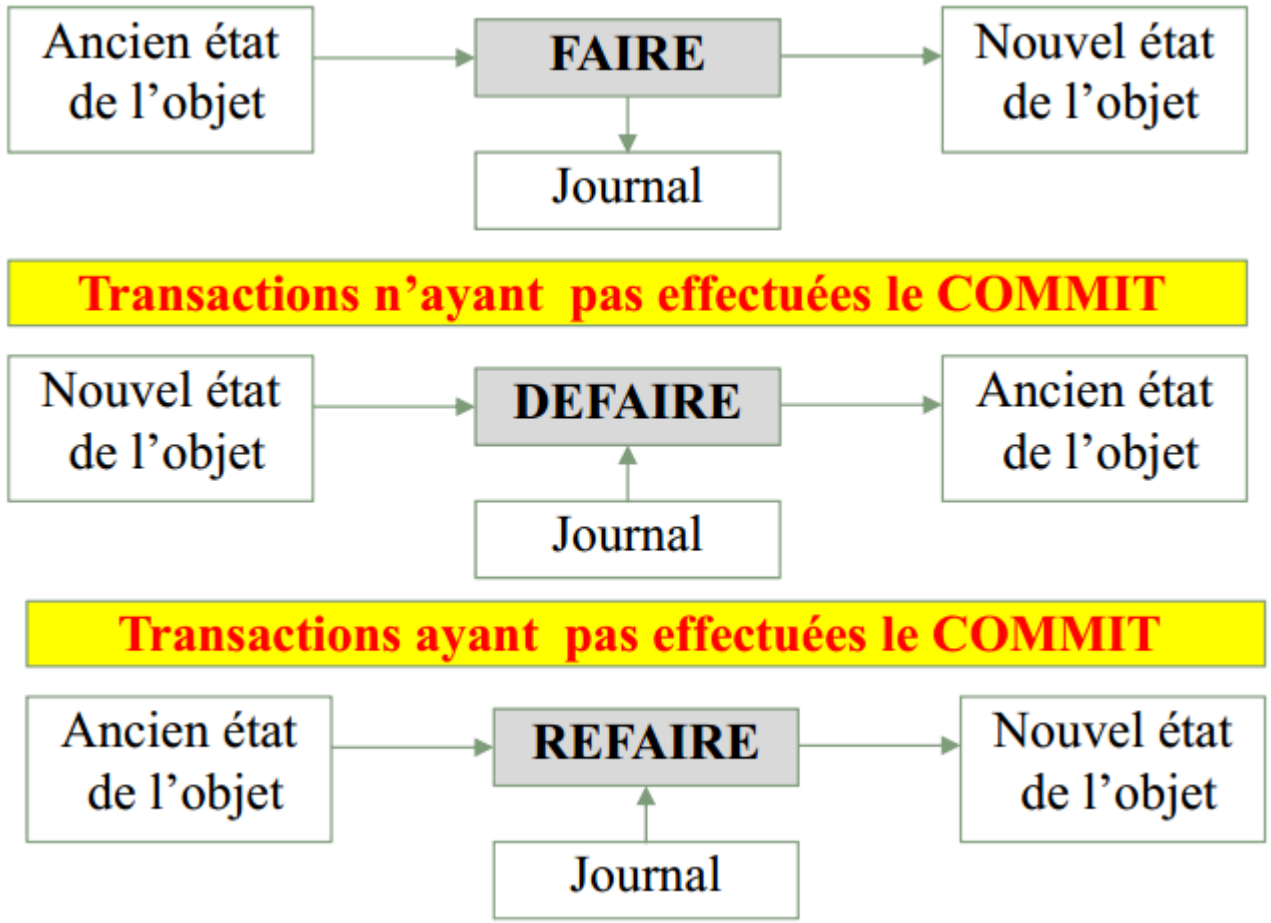
Règle du Après (redo)

Avant d'autoriser une transaction à validée, toutes les valeurs générées (les images- après de son journal) doivent d'abord être sauvegardées en mémoire stable (journal stable)



La journalisation

Paradigme faire-refaire-défaire



Utilisations du journal

Lors de la validation

sauvegarder toutes les images après en mémoire stable (journal)

- optimisation : « group commit + precommit »

Lors de l'annulation

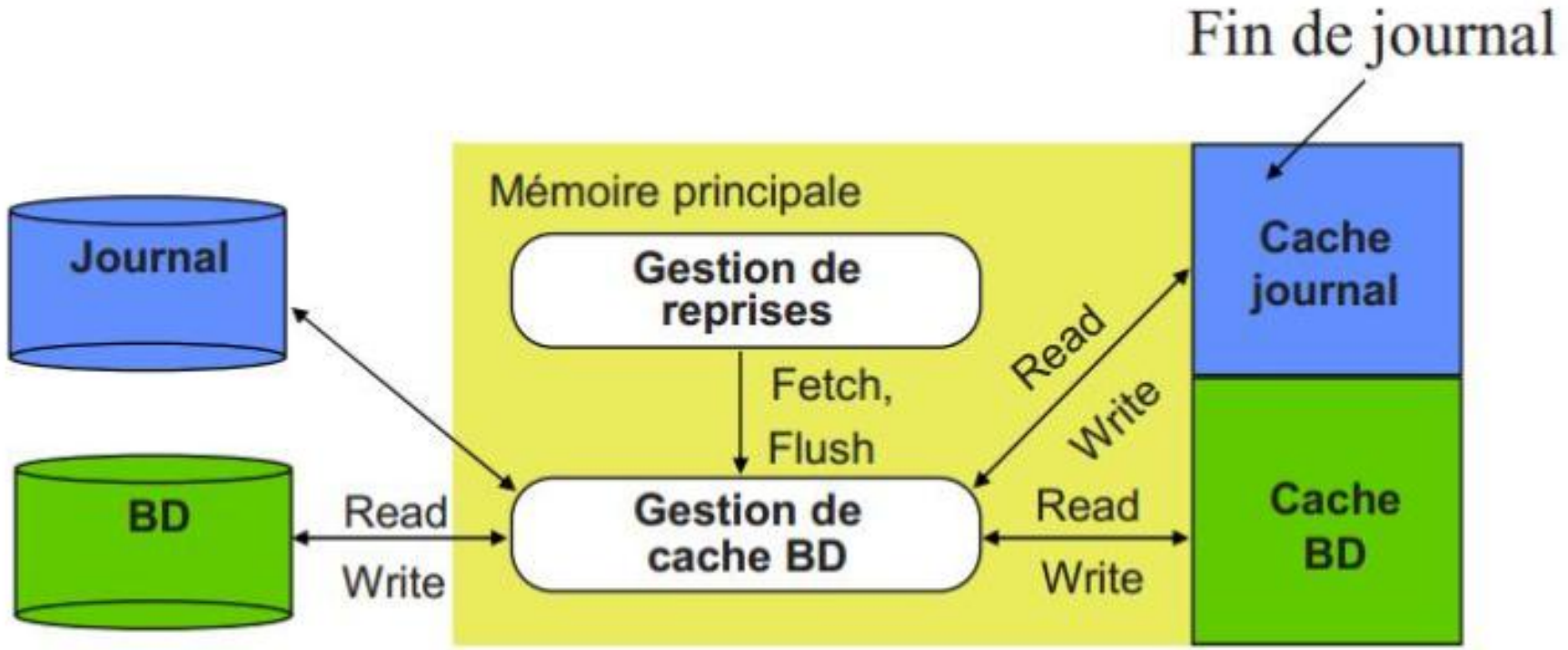
remettre les images avant (Undo) en parcourant le journal en arrière

Lors d'une reprise après panne

Refaire l'historique en exécutant les opérations du journal du début jusqu'au moment de la panne (dernier enregistrement)

- très coûteux et beaucoup de travail inutile

Interface du journal



Points de reprise

- Réduit la quantité de travail à refaire ou défaire lors d'une panne
- Un point de reprise enregistre une liste de transactions actives
- Pose d'un point de reprise:

– écrire un enreg. **begin_checkpoint** dans le journal

– écrire les buffers du journal et de la BD sur disque

– écrire un enreg. **end_checkpoint** dans le journal

Checkpoints (Points de Sauvegardes ou Points de reprise

C'est la sauvegarde (périodique) de certaines informations en **mémoire stable** durant le déroulement normal des transactions**, afin de réduire la quantité de travail** que doit faire la procédure **'Restart'** après une panne.

Checkpoint consistant

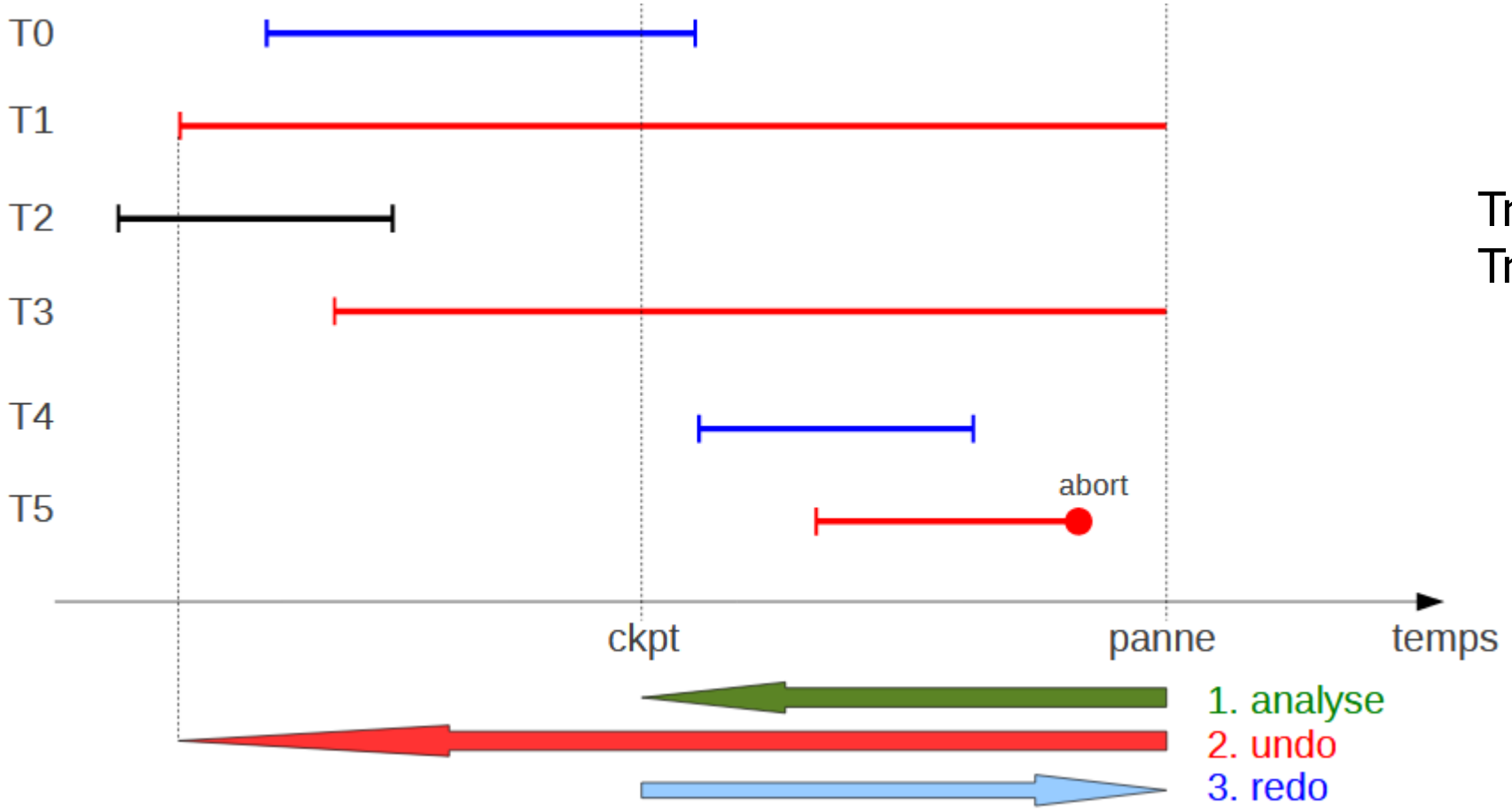
- Stopper l'acceptation de nouvelles transactions
- Attendre que les transactions actives se terminent
- Sauvegarder toutes les pages modifiées du cache
- Marquer le journal stable par un enreg <**ckpt**>

Lors de la reprise (Restart)

- Parcourir le journal en arrière jusqu'au <**ckpt**> le plus récent et défaire les opérations des transactions annulées
- Parcourir le journal en avant depuis la marque du <**ckpt**> et refaire les opérations des transactions validées

➲ Quel est inconvénient de cette solution ?
➲ Comment je peux optimiser ?

Exemple



Transaction refaites ??
Transaction defaites ??

reprise: T0 et T4 seront refaites. T1, T3 et T5 seront défaites

Question ?

Checkpoint consistant

➲ Quel est inconvénient de cette solution ?
➲ Comment je peux l'optimiser ?

Procédures de reprise

• **Reprise à chaud**

– perte de données en mémoire, mais sur disque

– à partir du dernier point de reprise , déterminer les transactions

validées : REDO

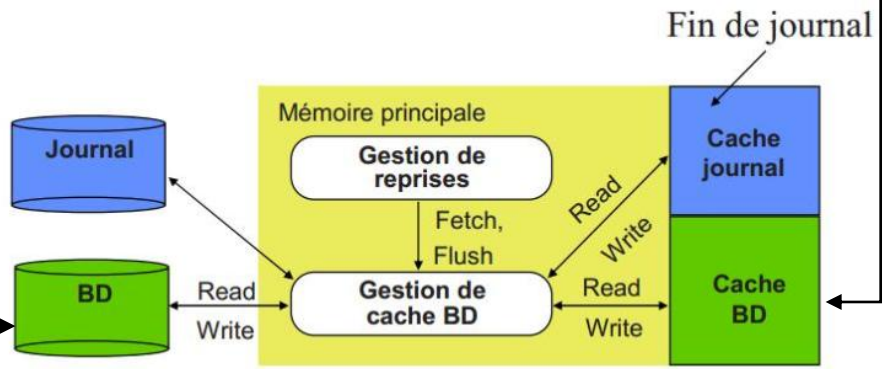
si Commit T, alors REDO := REDO + T ;

non validée : UNDO

– si Abort T, alors UNDO := UNDO + T ;

– si Begin T et T' ∉ REDO, alors UNDO := UNDO + T

•



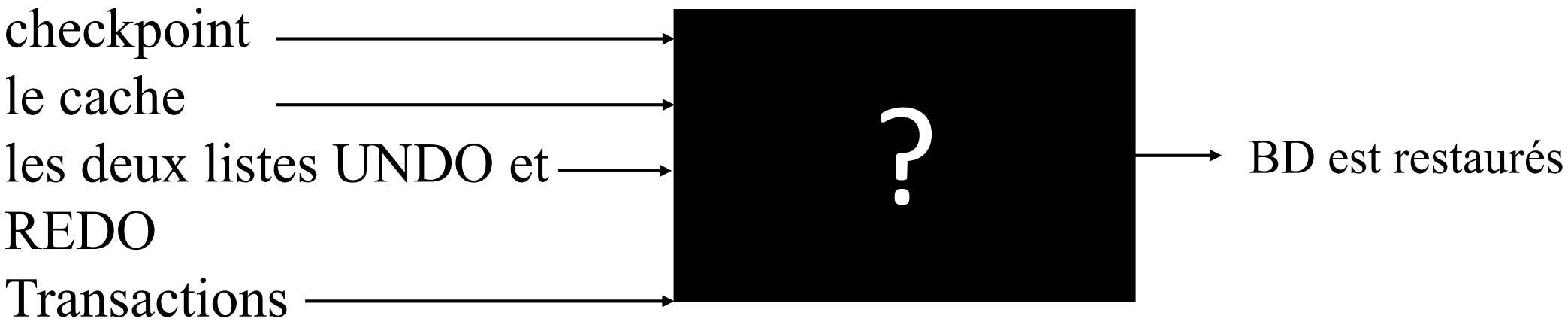
Reprise à froid

– perte de données sur disque

– à partir de la dernière sauvegarde et du dernier point de reprise, faire REDO des transactions validées

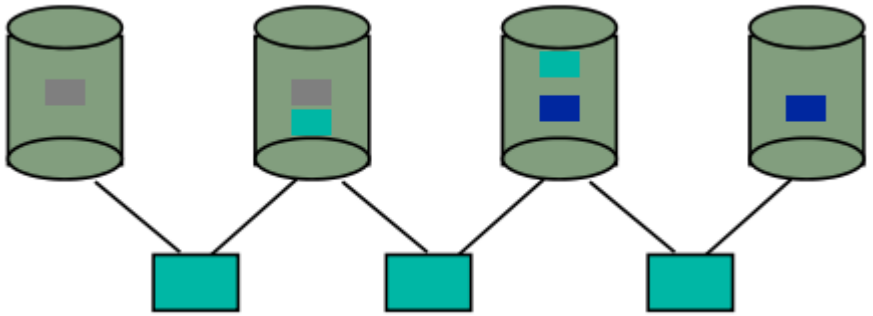
– UNDO inutile

Algo général de reprise



Tolérance aux pannes

- Possibilité de fonctionnement malgré les pannes
  - en général avec une reprise à chaud
- Approche traditionnelle
  - **duplication** en miroir du matériel et des données
- Avantage suppl. : le partage de charge



Multi-ordinateurs

- On peut stocker les données redondant d'une BD sur plusieurs sites
  - même distants
- données d'une SDDS
  - Structure de Données Distribuée et Scalable
- Une meilleure protection contre une panne catastrophique
  - explosion, panne de courant, grève...
- Une meilleure sécurité
  - il peut être nécessaire de pénétrer plusieurs sites pour avoir une donnée

Conclusion

Reprise après pannes

- Les SGBDs peuvent tomber en panne
- La sauvegarde et reprise fiable d'une BD est un problème capital pour un SGBD
- Toute donnée commise doit être préservée
- On peut restaurer les données à partir du journal
- On peut aussi prévenir la perte de données par le stockage redondant

– RAID tout particulièrement

– et SDDS (– Structure de Données Distribuée et Scalable) haute-disponibilité

TP 02: MySQL
Gestion des transactions

```
SET AUTOCOMMIT = 0
```

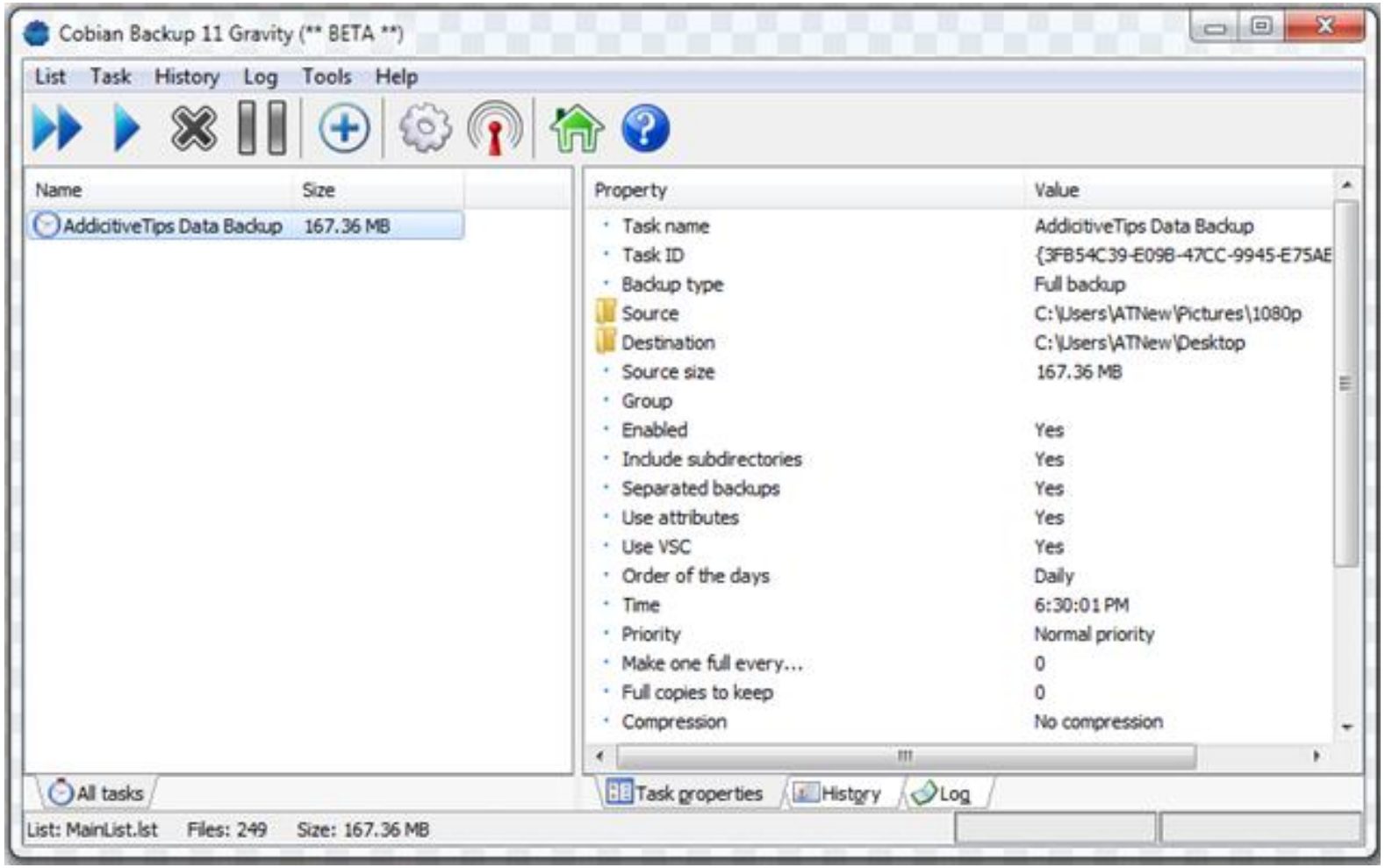
```
select @@autocommit ;
ou
select @@session.autocommit ;
```

```
SELECT ...FOR UPDATE ...
LOCK TABLES table ...{READ | WRITE } ...
```

```
UNLOCK TABLES
```

Exercice Pratique

- **client** FTP (FileZilla par exemple) **client** FTP
- un **serveur** FTP
- Lancer les différents type de sauvegarde:
  - complète,
  - incrémentielle,
  - différentielle.



Exercice

A table named 'GAMES' has the following contents:

GCode	GameName	Number_of_Players	PrizeMoney
101	Carom Board	2	5000
102	Badminton	2	12000
103	Table Tennis	4	8000

Write the output that will be displayed by statements (i) and (ii).

```
SELECT * FROM GAMES;
SET AUTOCOMMIT = 0;
INSERT INTO GAMES VALUES(105,'CHESS',2,9000);
ROLLBACK;
SAVEPOINT S1;
SELECT * FROM GAMES;  ------------ (i)
INSERT INTO GAMES VALUES(108,'LAWN TENNIS',4,25000);
SAVEPOINT S2;
INSERT INTO GAMES VALUES(109,'CRICKET',11,20000);
ROLLBACK TO  S2;
SELECT * FROM ITEM;  ------------ (ii)
```

(i)

GCode	GameName	Number_of_Players	PrizeMoney
101	Carom Board	2	5000
102	Badminton	2	12000
103	Table Tennis	4	8000

(ii)

GCode	GameName	Number_of_Players	PrizeMoney
101	Carom Board	2	5000
102	Badminton	2	12000
103	Table Tennis	4	8000
108	Lawn Tennis	4	25000

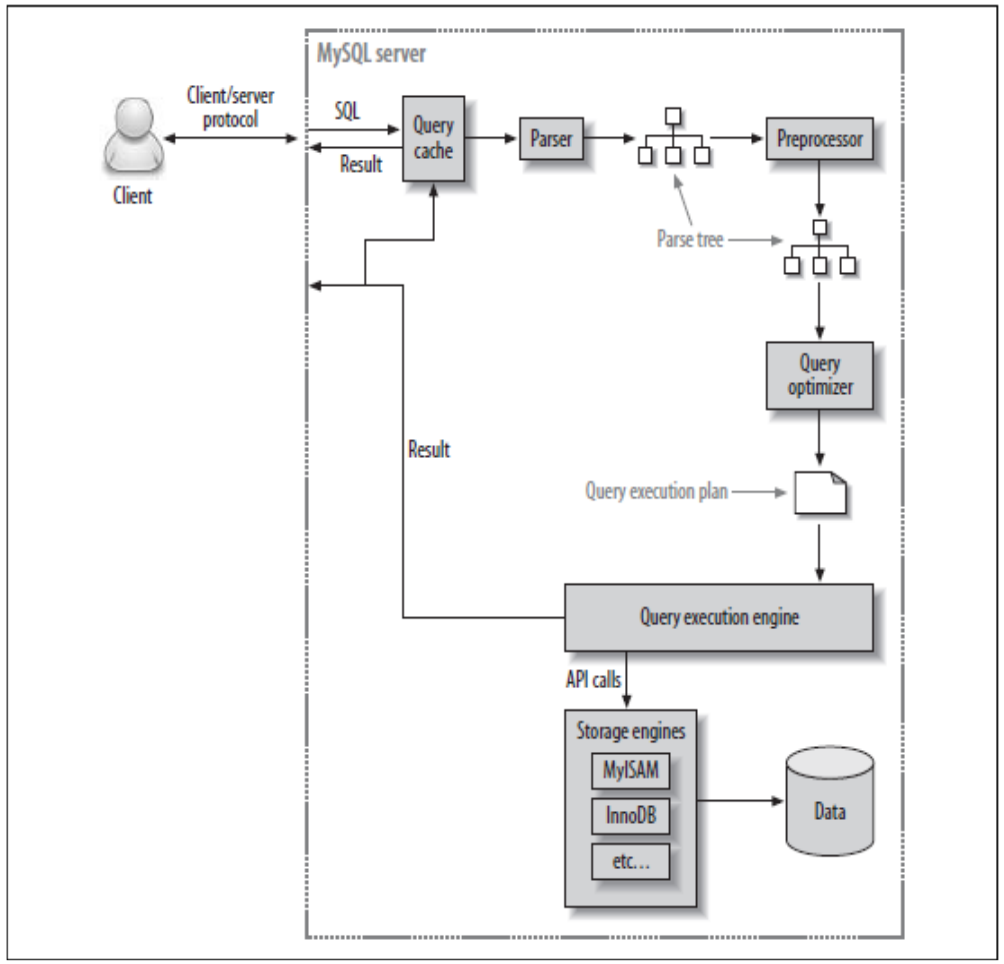
MySQL: Le choix de moteur

- Actuellement, MySQL dispose de nombreux moteurs, avec chacun une utilité particulière et des cas spécifiques d'utilisation.

  Voici les principaux :

  - MyISAM
  - InnoDB
  - MEMORY (anciennement HEAP)
  - MERGE
  - BLACKHOLE
  - BerkeleyDB ou BDB
  - ARCHIVE
  - CSV
  - FEDERATED

MySQL: Le choix de moteur



`MyISAM` **Beaucoup lectures / Recherche textuelle**

`InnoDB` **Read+Write / Transactions / Accès clé primaire**

`NDB` **Petites Transactions**, traitements parallèles

`MEMORY` **Uniquement en mémoire**

- CREATE TABLE nomTable (...) ENGINE nomMoteur

MySQL: Le choix de moteur

```sql
/* A la création de la table */
CREATE TABLE maTable(
...
)ENGINE=MonMoteurDeStockage;

/* En modifiant une table déjà créée */
ALTER TABLE maTable ENGINE=UnAutreMoteur;

```

```sql
SET storage_engine=NomDuMoteur;

```

- Change storage engine using **ALTER TABLE**
  >ALTER TABLE t ENGINE = MEMORY;

MySQL: Le choix de moteur

```
mysql> SET storage_engine=InnoDB;
-----
CREATE TABLE employee (
    IDpk  INTEGER NOT NULL AUTO_INCREMENT,
    ssn   CHAR(11) NOT NULL,
    name  CHAR(64),
    phone CHAR(32),
    PRIMARY KEY (IDpk)
) ENGINE = InnoDB;
-----
```

Le mot clé ENGINE

```
mysql> SHOW ENGINES;
+------------+---------+----------------------------------------------------
| Engine     | Support | Comment
+------------+---------+----------------------------------------------------
| InnoDB     | YES     | Supports transactions, row-level locking, and foreig
| MRG_MYISAM | YES     | Collection of identical MyISAM tables
| BLACKHOLE  | YES     | /dev/null storage engine (anything you write to it di
| CSV        | YES     | CSV storage engine
| MEMORY     | YES     | Hash based, stored in memory, useful for temporary ta
| FEDERATED  | NO      | Federated MySQL storage engine
| ARCHIVE    | YES     | Archive storage engine
| MyISAM     | DEFAULT | Default engine as of MySQL 3.23 with great performanc
+------------+---------+----------------------------------------------------
8 rows in set (0.00 sec)
```

TP

TP 02: MySQL
Gestion des transactions

- Créer une base de données avec deux tables (Client, Compte)
- Afficher la valeur AUTOCOMMIT
- Modifier la valeur auto commit
- Utiliser les COMMIT et ROLLBACK
- Ouvrir deux sessions , utiliser le **verro** des tables
- Utiliser les sauvegardes intermédiaires

VALIDATION

- Commit ;

- **ANNULATION**
  - Rollback ;
- **NB :** basculez en SET AUTOCOMMIT = 0; si par défaut c'est à 1.

```
SET AUTOCOMMIT = 0;
START TRANSACTION;
INSERT INTO villes (cp, nom_ville) VALUES
('24300','AILLAC');
SELECT * FROM VILLES;
ROLLBACK;
SELECT * FROM villes;
```

TP 02: MySQL
Gestion des transactions

- **LES SAVEPOINTS**

  Objectif

  Diviser une transaction en plusieurs sous-parties.

  Syntaxes

  Création d'un point de sauvegarde

- SAVEPOINT point_de_sauvegarde;

  Création d'un point de sauvegarde

  ROLLBACK TO SAVEPOINT point_de_sauvegarde;

  Suppression d'un point de sauvegarde

  RELEASE SAVEPOINT point_de_sauvegarde;

Exemple

Au final vous n'aurez que 75031.

```
SELECT * FROM villes;
SET AUTOCOMMIT=0;
START TRANSACTION;
SAVEPOINT sp1;
INSERT INTO villes(cp, nom_ville)
VALUES('75031','Paris 31');
SAVEPOINT sp2;
INSERT INTO villes(cp, nom_ville)
VALUES('75032','Paris 32');
ROLLBACK TO SAVEPOINT sp2;
COMMIT;
SELECT * FROM villes;
```

TP 02: MySQL
Gestion des transactions

- **1.5 LE VERROUILLAGE DE TABLE**

Les commandes LOCK et UNLOCK permettent de verrouiller et de déverrouiller une ou plusieurs tables en lecture ou en lecture/écriture.

Syntaxes

LOCK TABLES nom_de_table verrouillage [, nom_de_table verrouillage];

UNLOCK TABLES;

- **Exemple**

Ouvrez deux sessions clients (mysql et MySQL Query Browser par exemple).

Utilisateur	Autre utilisateur
SELECT * FROM villes; SET AUTOCOMMIT=0; START TRANSACTION; **LOCK TABLES** villes **READ**; UPDATE villes SET nom_ville = 'Marsiglia' WHERE cp = '13000'; **UNLOCK TABLES**; COMMIT;	SELECT * FROM villes; -- OK UPDATE villes SET nom_ville = 'Marsilia' WHERE cp = '13000'; -- KO

Utilisateur	Autre utilisateur
SELECT * FROM villes; SET AUTOCOMMIT=0; START TRANSACTION; **LOCK TABLES** villes **WRITE**; UPDATE villes SET nom_ville = 'Marsiglia' WHERE cp = '13000'; **UNLOCK TABLES**; COMMIT;	SELECT * FROM villes; -- KO UPDATE villes SET nom_ville = 'Marsilia' WHERE cp = '13000'; -- KO

TP 02: MySQL
Gestion des transactions

- **1.6 LE VERROUILLAGE DE LIGNE**

  La clause FOR UPDATE appliquée à un SELECT permet de verrouiller un ou plusieurs lignes en écriture.

  - **Syntaxe**

```
SELECT * FROM nomDeTable WHERE condition FOR UPDATE;
```

  - **Exemple**

  Ouvrez deux sessions clients (mysql et MySQL Query Browser par exemple).

Utilisateur	Autre utilisateur
SET AUTOCOMMIT=0; START TRANSACTION; SELECT * FROM pays WHERE id_pays = '033' **FOR UPDATE**; UPDATE pays SET nom_pays = 'FR' WHERE id_pays = '033'; COMMIT;	 SET AUTOCOMMIT=0; START TRANSACTION; UPDATE pays SET nom_pays = 'fr' WHERE id_pays = '033'; -- Attente bloquante (*) COMMIT;

  (*) UPDATE sur un autre pays c'est OK.

TP 02: MySQL
Gestion des transactions

- **1.6 LE VERROUILLAGE DE LIGNE**

  La clause FOR UPDATE appliquée à un SELECT permet de verrouiller un ou plusieurs lignes en écriture.

  - **Syntaxe**

```
SELECT * FROM nomDeTable WHERE condition FOR UPDATE;
```

  - **Exemple**

  Ouvrez deux sessions clients (mysql et MySQL Query Browser par exemple).

Utilisateur	Autre utilisateur
SET AUTOCOMMIT=0; START TRANSACTION; SELECT * FROM pays WHERE id_pays = '033' **FOR UPDATE**; UPDATE pays SET nom_pays = 'FR' WHERE id_pays = '033'; COMMIT;	SET AUTOCOMMIT=0; START TRANSACTION; UPDATE pays SET nom_pays = 'fr' WHERE id_pays = '033'; -- Attente bloquante (*) COMMIT;

  (*) UPDATE sur un autre pays c'est OK.