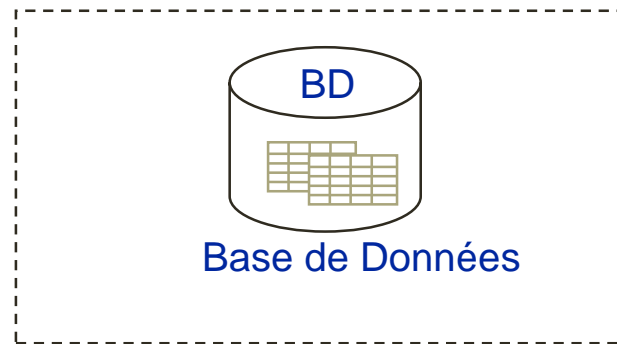


# SGBD RELATIONNELS

## COURS Transaction



# TRANSACTION

## TRANSACTION

Programming Language

**PL**

PROCEDURE/FUNCTION / TRIGGER

Langage de Manipulation des Données

**SQL LMD**

SELECT/INSERT/UPDATE/DELETE

Langage de Définition des Données

**SQL LDD**

CREATE/DROP/ALTER

INDEX

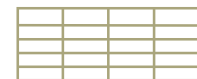
VUE

TRIGGER

PROCEDURE STOCKEE

SEQUENCE

TABLE



*Données*



**Base de Données**

**Norme SQL**

SQL1  
1986

SQL2  
1992

SQL3  
2003

SQL4

# TRANSACTION

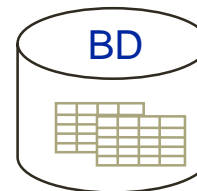
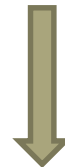
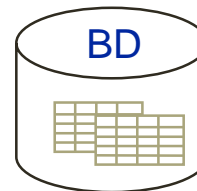
TRANSACTION = suite d'instructions SQL qui fait passer une BD d'un état initial cohérent à un état final cohérent.

## TRANSFERT BANCAIRE

```
UPDATE COMPTE SET Solde = Solde -100  
WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100  
WHERE idCompte=2;
```

A noter : la consistance d'un ordre SQL est toujours assuré

*Etat initial*



*Etat final*

*Si un problème  
intervient, les  
instructions sont  
annulées pour revenir  
à l'état initial*

Le mécanisme transactionnel des SGBD relationnels empêche tout scénario problématique avec la technique de *journalisation* et de *verrou*

# Début de TRANSACTION

## Quelles instructions mettront dans une transaction ?

- **Norme SQL** : Toute Instruction SQL (LMD, LDD) possible
- **Oracle, Mysql** : Uniquement SQL LMD
- **SQLITE/Postgresql** : SQL LDD + SQL LMD

## Syntaxe SQL des transactions :

La norme SQL prévoit que toute connexion à une base de données crée une nouvelle transaction.

- **VRAI pour ORACLE**
  - *Pas de syntaxe de début de transaction*
- **FAUX pour MYSQL/POSTGRESQL**
  - *Mode autocommit par défaut*
  - *Syntaxe début de transaction : BEGIN / START TRANSACTION*

# Fin de TRANSACTION

## Une transaction peut se terminer :

### 1. Explicitement :

- ORDRES SQL : COMMIT / ROLLBACK

### 2. Implicitement :

- Déconnexion de la session
- ORDRE SQL DDL

### COMMIT

- Validation de la transaction.
- Les MAJ sont disponibles pour les autres transactions.
- Les MAJ sont durables.

### ROLLBACK

- Les MAJ sont annulées.
- Retour à l'état initial de la transaction

Début de la transaction

Fin de la transaction

```
UPDATE COMPTE SET Solde = Solde -100 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE idCompte=2;  
COMMIT;
```

**ORACLE**

Début de la transaction

Fin de la transaction

### **START TRANSACTION**

```
UPDATE COMPTE SET Solde = Solde -100 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE idCompte=2;  
COMMIT;
```

**MYSQL**

# Fin de TRANSACTION

## SESSION

Début transaction 1

Fin transaction 1

Début transaction 2

Fin transaction 2

Début transaction 3

Fin transaction 3

Ordre Auto validé

Début transaction 5

Fin transaction 5

```
UPDATE COMPTE SET Solde = Solde -100 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE idCompte=2;  
COMMIT;
```

```
UPDATE COMPTE SET Solde = Solde -200 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +200 WHERE idCompte=2;  
COMMIT;
```

```
UPDATE COMPTE SET Solde = Solde -300 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +400 WHERE idCompte=2;  
L'ordre SQL DDL ajoute un COMMIT implicite
```

```
CREATE TABLE HISTO ( Tuser VARCHAR(30));
```

```
UPDATE COMPTE SET Solde = Solde -500 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +500 WHERE idCompte=2;  
ROLLBACK;
```

*Le ROLLBACK annule les instruction de la transaction en cours.*

# VALIDATION/ANNULATION D'UNE TRANSACTION (ORACLE)

Avant l'exécution d'une instruction LDD, une validation implicite de la transaction en cours a lieu

- *Cette validation est maintenue quelle que soit l'issue de l'instruction*
- *Si l'instruction LDD réussit, une validation implicite à sa fin est également exécutée*

Quel que soit l'environnement d'exécution

- *Une déconnexion normale provoque une validation implicite de la transaction en cours*
- *Une déconnexion anormale implique une invalidation de la transaction en cours*



Sous SQLPLUS, la fermeture de la fenêtre est considérée comme une déconnexion anormale → ROLLBACK

# TRANSACTION ET NORME SQL

- La commande COMMIT

*COMMIT [ WORK ]*

- La commande ROLLBACK

*ROLLBACK [ WORK ] [ TO [ SAVEPOINT ] savepoint ]*

- Le point de reprise est créé par la commande SAVEPOINT

*SAVEPOINT savepoint*

Les points de reprise permettent de défaire (ROLLBACK TO) une partie d'une transaction



# MODE AUTOCOMMIT

- Le mode autocommit permet d'annuler le mode transactionnel.
- S'il est activé, chaque ordre SQL LMD (UPDATE/INSERT/DELETE) est automatiquement validé

## SYNTAXE :

- **MYSQL** : SET AUTOCOMMIT { 0 | 1 } (1 par défaut)
  - *1 par défaut. Active le mode autocommit*
  - *0 à adapter pour mettre en place des transactions*
- **ORACLE** : SET AUTOCOMMIT { ON | OFF } (OFF par défaut)

```
SQL T1> SET AUTOCOMMIT ON
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100;

3 rows updated.
Commit complete.
SQL T1>
```

Le UPDATE a été validé automatiquement

# PROPRIETE ACID

*Une transaction doit respecter quatre propriétés fondamentales :*

## **Atomicité:**

Les transactions constituent l'unité logique de travail : toute la transaction est exécutée ou bien rien du tout, mais jamais une partie seulement de la transaction. Tout ou Rien.

## **Cohérence:**

Les transactions préservent la cohérence de la , c'est à dire qu'elle transforme la BD d'un état initial cohérent à un état final cohérent.

## **Durabilité :**

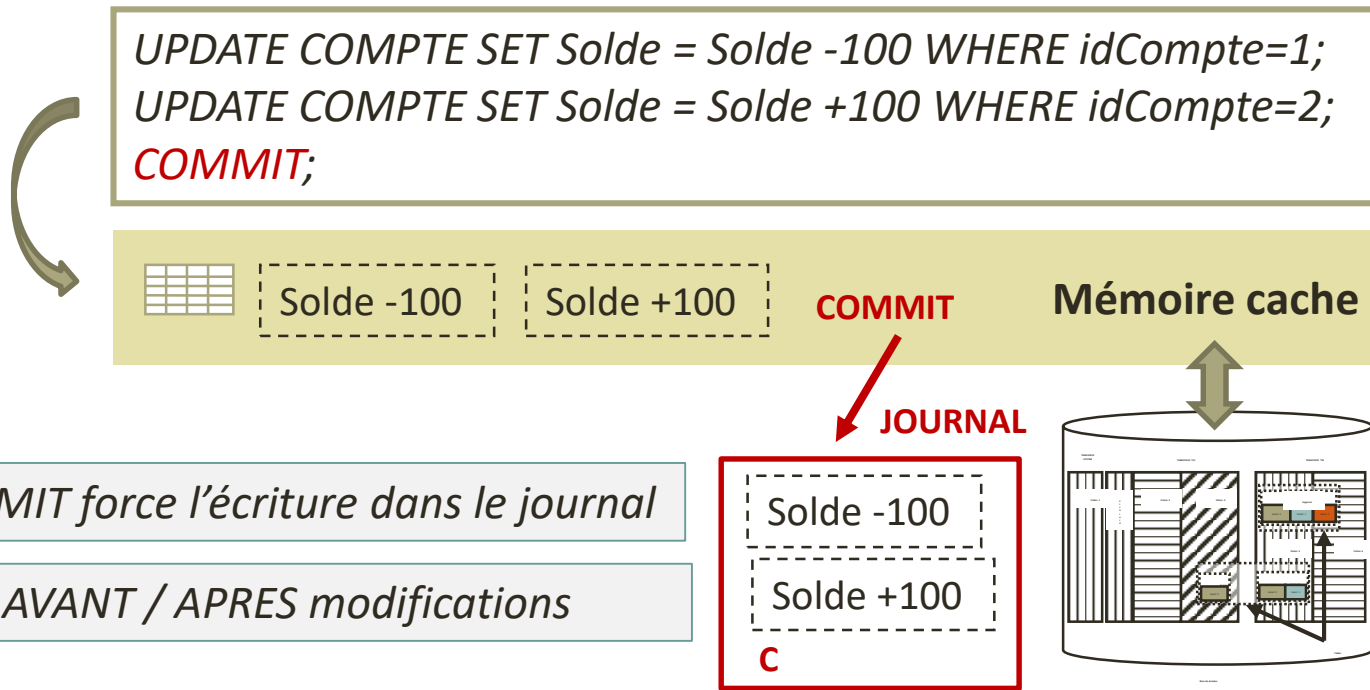
La transaction assure que les modifications qu'elle induit perdurent, même en cas de défaillance ou panne.

## **Isolation :**

Les transactions sont isolées les unes des autres : leur exécution est indépendante des autres transactions en cours. Elles accèdent donc à la BD comme si elles étaient seules à s'exécuter.

# JOURNALISATION

Le mécanisme transactionnel des SGBD relationnels empêche tout scénario problématique avec la technique de **journalisation** et de **verrou**



Le journal (LOG) permet :

1. Les ROLLBACK
  - par parcours arrière du LOG
2. La mise en place d'un mécanisme de reprise sur panne (DURABILITE)
  - Restauration Sauvegarde + parcours du LOG

# ANOMALIES DE CONCURRENCES (1/5)

- Un SGBD reçoit plusieurs transactions simultanées sur les mêmes données
- L'exécution séquentielle des transactions ne pose pas de problème
- **PROBLEME** : les transactions sont susceptibles de s'exécuter de manière entrelacées ( Multitâche / parallélisme)

## TRANSACTION 1

```
UPDATE COMPTE SET Solde = Solde -100 WHERE  
idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE  
idCompte=2;  
COMMIT;
```

## TRANSACTION 2

```
UPDATE COMPTE SET Solde = Solde -100 WHERE  
idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE  
idCompte=2;  
COMMIT;
```

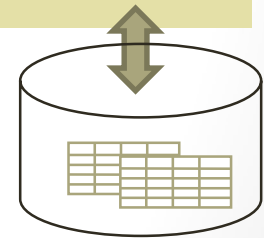
## TRANSACTION 3

```
UPDATE COMPTE SET Solde = Solde -100 WHERE  
idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE  
idCompte=2;  
COMMIT;
```

Les 3 transactions accèdent à la même table et mêmes données

4 types d'anomalies liées à la concurrence sont recensés :

1. **LECTURES SALES (DIRTY READS)**
2. **LECTURES NON REPRODUCTIBLES (NON REPEATABLE READ)**
3. **LECTURES FANTÔMES (PHANTOM READS)**
4. **PERTE DE MISE A JOUR**



# ANOMALIES DE CONCURRENCES (2/5)

1. **DIRTY READS** : lecture de données sales. Se produit lorsqu'une transaction accède aux données d'une autre transaction non encore validée.

Temps	Transaction A	Transaction B
t1		UPDATE T
t2	LIRE T	...
t3		ROLLBACK

- La transaction A accède au tuple T qui a été modifié par la transaction B.
- B annule sa modification et A a donc accédé à une valeur qui n'aurait jamais dû exister (virtuelle) de T.
- Pire A pourrait faire une mise à jour de T

# ANOMALIES DE CONCURRENCES (3/5)

## **NON REPEATABLE READ (LECTURE NON REPRODUCTIBLES)**

*Se produit quand 2 lectures successives d'une même donnée dans une transaction ne donnent pas le même résultat*

Temps	Transaction A	Transaction B
t1	LIRE T	
t2	...	UPDATE T
t3	...	COMMIT
t4	LIRE T	

- La transaction A accède deux fois à la valeur d'un tuple T
- T est, entre les deux, modifié par une autre transaction B
- Alors la lecture de A est inconsistente.

Ceci peut entraîner des incohérences par exemple si un calcul est en train d'être fait sur des valeurs par ailleurs en train d'être mises à jour par d'autres transactions.

# ANOMALIES DE CONCURRENCES (4/5)

## **PHANTOM READS :**

*Se produit lorsque de nouvelles données apparaissent ou disparaissent au cours de lectures successives.*

Temps	Transaction A	Transaction B
t1	LIRE T	
t2	...	INSERT / DELETE T
t3	...	COMMIT
t4	LIRE T	

- La transaction A accède deux fois à la table T
- T est, entre les deux, modifiée par une autre transaction B qui ajoute ou supprime des tuples
- Des données sont apparues ou ont disparues
- La lecture de A devient inconsistente.

# ANOMALIES DE CONCURRENCES (5/5)

## PERTE DE MISE A JOUR

*Peut se produire quand 2 transactions accèdent en écriture aux même données*

Temps	Transaction A	Transaction B
t1	LIRE T	
t2	...	LIRE T
t3	UPDATE T	...
t4	...	UPDATE T
t5	COMMIT	...
t4		COMMIT

- Les transaction A et B accèdent au même tuple T ayant la même valeur respectivement à t1 et t2.
- A et B modifient la valeur de T.
- Les modifications effectuées par A seront perdues puisqu'elle avait lu T avant sa modification par B.



# GESTION DES ANOMALIES DE CONCURRENCES

Pour gérer les problèmes de concurrences, les SGBD relationnels mettent en place des NIVEAUX D'ISOLATION des transactions

- *La norme SQL prévoit 4 niveaux d'isolation des transactions pour résoudre ces anomalies.*
- *Chaque niveau d'ISOLATION sert à résoudre un type d'anomalie*
- *Ces niveaux sont implémentés par 2 mécanismes :*
  - *Pose de VERROUS*
  - *Exploitation des journaux des opérations réalisées par les transactions*

# NIVEAUX D'ISOLATION (NORME SQL)

Niveau d'isolation	Lectures Sales	Lectures Non Reproductibles	Lectures Fantômes
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

**Syntaxe SQL :** SET TRANSACTION ISOLATION LEVEL

{READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}

**Exemple ORACLE**

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
UPDATE COMPTE SET Solde = Solde -100 WHERE idCompte=1;  
UPDATE COMPTE SET Solde = Solde +100 WHERE idCompte=2;  
COMMIT;
```

**ORACLE:** ne fournit que les niveaux READ COMMITTED (par défaut) et SERIALIZABLE

**MYSQL :** fournit les 4 niveaux (sur InnoDB). REPEATABLE READ par défaut.

# CONFIGURATION du niveau d'isolation sous ORACLE

- Pour toutes les transactions d'une session
  - ALTER SESSION SET ISOLATION LEVEL SERIALIZABLE
  - ALTER SESSION SET ISOLATION LEVEL READ COMMITTED
- Pour une seule transaction (A mettre en début de transaction)
  - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
  - SET TRANSACTION ISOLATION LEVEL READ COMMITTED

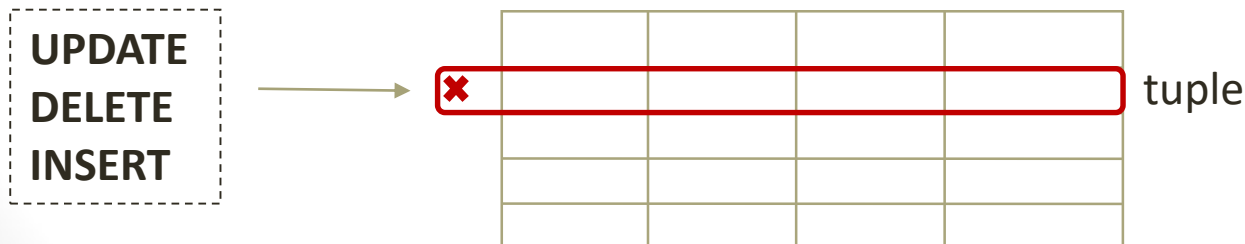
# VERROUILLAGE

- Une solution générale à la gestion de la concurrence est une technique très simple appelée verrouillage.
- Poser un verrou sur un objet (TUPLE/ TABLE) par une transaction signifie rendre cet objet inaccessible aux autres transactions.
- Dans le norme SQL, il existe plusieurs types de verrous (X, S, ...)
- Un verrou **exclusif**, noté X, est posé par une transaction lors d'un accès en écriture sur cet objet.

On peut poser des verrous de manière :

**IMPLICITE** : ordres SQL UPDATE/DELETE/INSERT

**EXPLICITE** : ordres SQL « LOCK » ou (SELECT ... FOR UPDATE)



Les ordres SQL d'écriture posent implicitement des verrous X sur les tuples concernés

# ACCES CONCURRENTS DE TRANSACTIONS

## 3 scénarios associés au comportement standard du SGBD

### SGBD : ORACLE

*Mode d'isolation par défaut d'oracle : READ COMMITTED*

# SCENARIO 1 : MAJ / LECTURE

## Transaction 1

2° ) UPDATE Personne SET Salaire = Salaire + 100;

3° ) SELECT Salaire FROM Personne;

5° ) COMMIT;

```
SQL T1> select * from PERSONNE;
```

ID	SALAIRE
1	300
2	250
3	500

3 rows selected.

SQL T1>

## Transaction 2

1°) SELECT Salaire FROM Personne;

4°) SELECT Salaire FROM Personne;

6°) SELECT Salaire FROM Personne;

```
SQL T1> select * from PERSONNE;
```

ID	SALAIRE
1	300
2	250
3	500

3 rows selected.

SQL T1>

**On ouvre 2 sessions  
SQL PLUS sur la même base**

- T1 et T2 sont 2 transactions concurrentes sur la même base
- On simule un entrelacement possible d'exécution des transactions (1, 2, 3 ..)
- T1 écrit sur la base
- T2 fait 3 fois la même lecture sur la base

# SCENARIO 1 : MAJ / LECTURE

## Transaction 1

ID	SALAIRE
1	200
2	150
3	400

2° ) UPDATE PERSONNE SET Salaire = Salaire + 100;

*Le SELECT « voit » les MAJ du UPDATE*

3° ) SELECT Salaire FROM PERSONNE;

ID	SALAIRE
1	300
2	250
3	500

5° ) COMMIT;

*Le COMMIT valide la transaction*

OK

## Transaction 2

1°) SELECT Salaire FROM PERSONNE;

ID	SALAIRE
1	200
2	150
3	400

*Le SELECT ne « voit » pas les MAJ du UPDATE*

4°) SELECT Salaire FROM PERSONNE;

*Le SELECT « voit » les MAJ du UPDATE*

6°) SELECT Salaire FROM PERSONNE;

ID	SALAIRE
1	300
2	250
3	500

## Isolation READ COMMITTED

- Une transaction voit ses propres MAJ
- Pas de lectures sales : on ne voit que les données validées.

Anomalie constatée : Lectures non reproductibles dans T2

2 solutions possibles

## Solution 1 : READ ONLY

```
SQL T1> SELECT * from PERSONNE;
```

ID	SALAIRE
1	700
2	650
3	600

3 rows selected.

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100;
```

3 rows updated.

```
SQL T1> SELECT * from PERSONNE;
```

ID	SALAIRE
1	800
2	750
3	700

3 rows selected.

```
SQL T1> COMMIT;
```

Commit complete.

On ne change pas le niveau d'isolation  
mais on change le mode d'accès aux  
données de la transaction

### SET TRANSACTION READ ONLY

- Ordre de début de transaction
- Modifie le mode d'accès au données
- Plus de lecture non reproductibles

```
SQL T2> SET TRANSACTION READ ONLY;
```

Transaction set.

```
SQL T2> SELECT * FROM PERSONNE;
```

ID	SALAIRE
1	700
2	650
3	600

```
SQL T2> SELECT * FROM PERSONNE;
```

ID	SALAIRE
1	700
2	650
3	600

```
SQL T2> SELECT * FROM PERSONNE;
```

ID	SALAIRE
1	700
2	650
3	600

```
SQL T2> COMMIT;
```

Commit complete.

MAIS uniquement des lectures  
dans la transaction



## Solution 2 : SERIALIZABLE

```
SQL T1> select * from PERSONNE;
```

ID	SALAIRE
1	800
2	750
3	700

3 rows selected.

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100;
```

3 rows updated.

```
SQL T1> select * from PERSONNE;
```

ID	SALAIRE
1	900
2	850
3	800

3 rows selected.

```
SQL T1> COMMIT;
```

Commit complete.

```
SQL T2> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Transaction set.

```
SQL T2> SELECT * FROM PERSONNE;
```

ID	SALAIRE
1	800
2	750
3	700

```
SQL T2> SELECT * FROM PERSONNE;
```

ID	SALAIRE
1	800
2	750
3	700

```
SQL T2> SELECT * FROM PERSONNE;
```

ID	SALAIRE
1	800
2	750
3	700

```
SQL T2> COMMIT;
```

Commit complete.

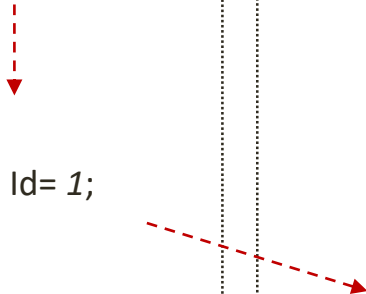
### On change le niveau d'isolation NIVEAU SERIALIZABLE

- Ordre de début de transaction
- Assure des lectures consistantes
- La transaction ne peut avoir accès qu'à des données validées avant son début.

Niveau trop exigeant  
pour de simples lectures

## SCENARIO 2 : MAJ / MAJ

### Transaction 1

- 2° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 2;
- 3° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;
- 

### Transaction 2

- 1°) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;
- 4°) COMMIT;

- T1 et T2 font toutes les 2 des écritures sur la même table (UPDATE)
- T1 tente d'écrire sur la même ligne que T2

## EXEMPLE 2 : SCENARIO MAJ / MAJ

### Transaction 1

ID	SALAIRE
1	400
2	450
3	500

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=2;
```

1 row updated.

```
SQL T1> select * from PERSONNE;
```

ID	SALAIRE
1	400
2	550
3	500

*T1 obtient le verrouillage du tuple 2*

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=1;
```

*T1 est mis en attente. T2 a déjà un verrou sur le 1*

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=1;
```

1 row updated.

*T1 obtient le verrou sur le tuple 1 et peut reprendre*

### Transaction 2

```
SQL T2> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=1;
```

1 row updated.

```
SQL T2> select * FROM PERSONNE;
```

ID	SALAIRE
1	500
2	450
3	500

*T2 obtient le verrouillage du tuple 1*

```
SQL T2> COMMIT;
```

Commit complete.

*Le COMMIT relâche tous les verrous de T2*

ID	SALAIRE
1	500
2	450
3	500

Mode d'isolation par défaut d'oracle : READ COMMITTED

## EXEMPLE 2 : SCENARIO MAJ / MAJ

### Transaction 1

- 2° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 2;
- 3° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;

### Transaction 2

- 1°) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;
- 4°) COMMIT;
- 

- Granularité niveau tuple
- Toute écriture requière l'acquisition d'un verrou Exclusif pour s'exécuter
- UPDATE/DELETE/INSERT pose implicitement des verrous
- Libération des verrous uniquement en fin de transaction (COMMIT/ROLLBACK)
- Pas de perte de MAJ

## SCENARIO 3 : Interblocage

### Transaction 1

- 2° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 2;
- 3° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;

### Transaction 2

- 1°) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;
- 4°) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 2

- T1 et T2 font toutes les 2 des écritures sur la même table (UPDATE)
- T1 tente d'écrire sur la même ligne que T2
- T2 tente d'écrire sur la même ligne que T1

ATTENTE MUTUELLE ?

# EXEMPLE 3 : Interblocage

## Transaction 1

ID	SALAIRE
1	400
2	450
3	500

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=2;
```

1 row updated.

```
SQL T1> select * from PERSONNE;
```

ID	SALAIRE
1	400
2	550
3	500

*T1 obtient le verrouillage du tuple 2*

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=1;
```

*T1 est mis en attente. T2 a déjà un verrou sur le 1*

```
SQL T1> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=1;  
UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
*  
ERROR at line 1:  
ORA-00060: deadlock detected while waiting for resource
```

*Le SGBD détecte une attente mutuelle et ANNULE une transaction (T1)*

## Transaction 2

```
SQL T2> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=1;
```

1 row updated.

```
SQL T2> select * FROM PERSONNE;
```

ID	SALAIRE
1	500
2	450
3	500

*T2 obtient le verrouillage du tuple 1*

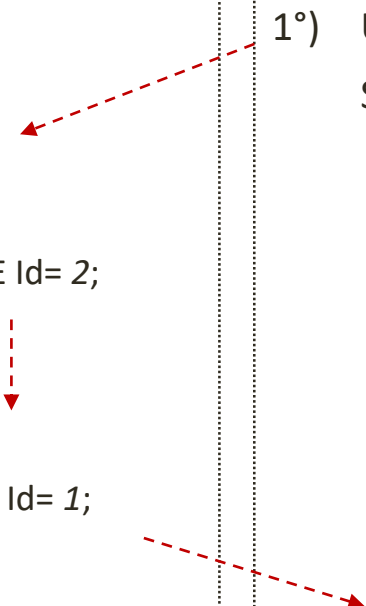
```
SQL T2> UPDATE PERSONNE SET SALAIRE=SALAIRE+100  
2 WHERE Id=2;
```

*T2 demande un Verrou sur le tuple 2.  
Se met en attente*

ID	SALAIRE
1	500
2	450
3	500

## SCENARIO 3 : Interblocage

### Transaction 1

- 2° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 2;
- 3° ) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;
- 

### Transaction 2

- 1°) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 1;
- 4°) UPDATE PERSONNE  
SET Salaire = Salaire + 100 WHERE Id= 2

- Le SGBD détecte les interblocages (DEAD LOCK)
- Il annule une transaction et libère ses verrous
- L'autre transaction peut alors continuer
- La transaction annulée devra être relancée

# ***SYNTHESE***

AUTO COMMIT ON : PAS DE TRANSACTIONS

AUTO COMMIT OFF

- Niveau d'isolation par défaut
- Acquisition de verrous graduelle au niveau des tuples
- Un verrou, une fois acquis, n'est libéré qu'à la fin de la transaction
- Détection automatique d'interblocages
- Verrouillage au niveau de tuples



**La configuration standard d'un SGBD assure  
un blocage minimal des transactions concurrentes**



# TRANSACTIONS et PROGRAMMES STOCKES

## TRANSACTION

**PL**

PROCEDURE/FUNCTION / TRIGGER

Programming Language

**SQL LMD**

SELECT/INSERT/UPDATE/DELETE

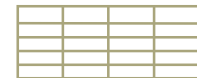
Langage de Manipulation des Données

**SQL LDD**

CREATE/DROP/ALTER

Langage de Définition des Données

INDEX  
VUE  
TRIGGER  
PROCEDURE STOCKEE  
SEQUENCE  
TABLE



*Données*



**Base de Données**

**Norme SQL**

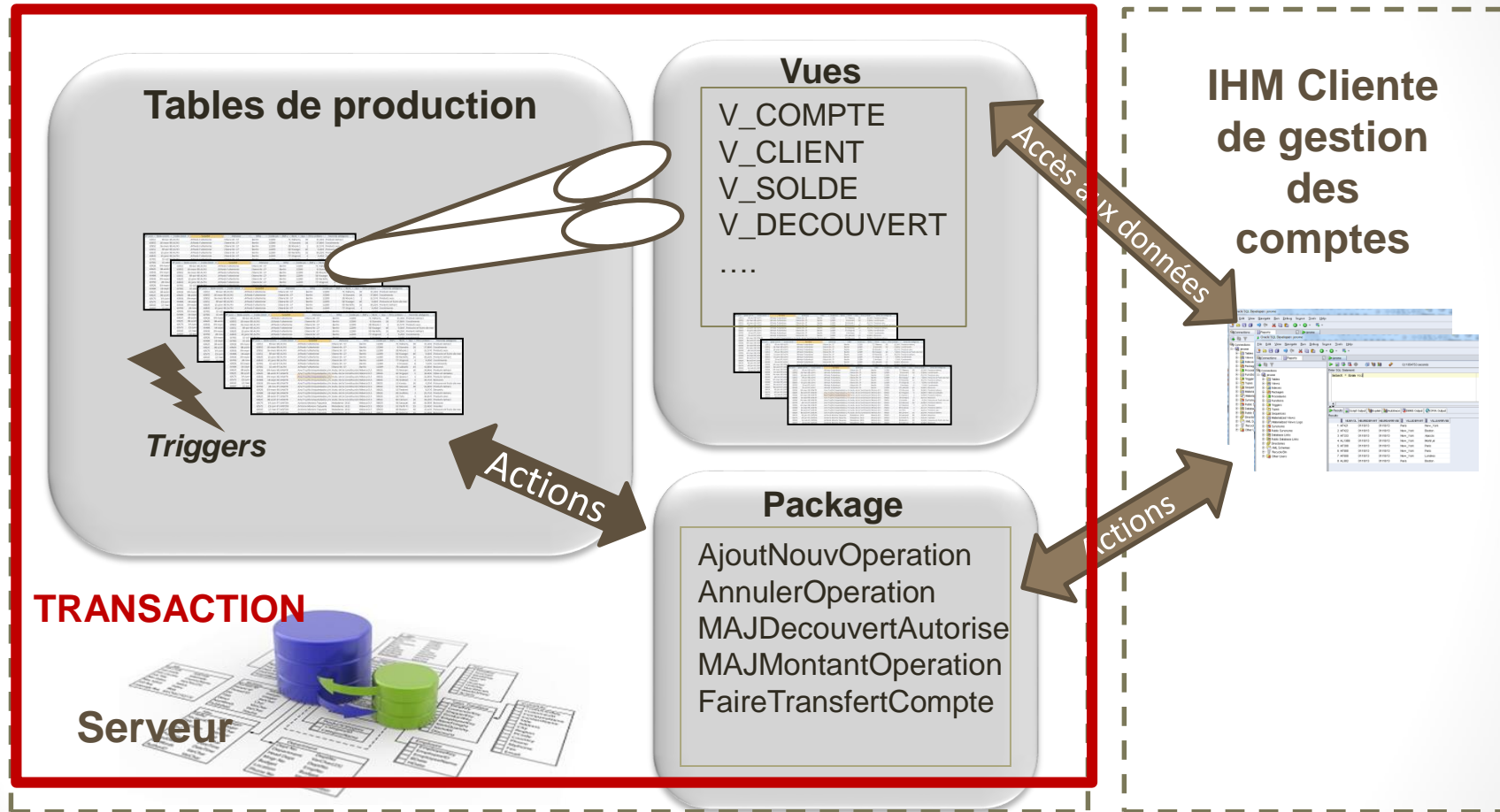
SQL1  
1986

SQL2  
1992

SQL3  
2003

SQL4

# APPEL DES TRANSACTIONS(1/4)



A quel niveau se gère la transaction : programme client , objets serveurs ?

# APPEL DES TRANSACTIONS(2/4)

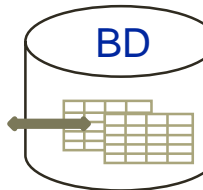
*La gestion d'une transaction doit « normalement » s'effectuer au niveau le plus haut*

## Application cliente

### TRANSACTION

```
TRANSFERT_BANCAIRE(1,2,150);  
COMMIT;
```

Appel de la procédure  
stockée

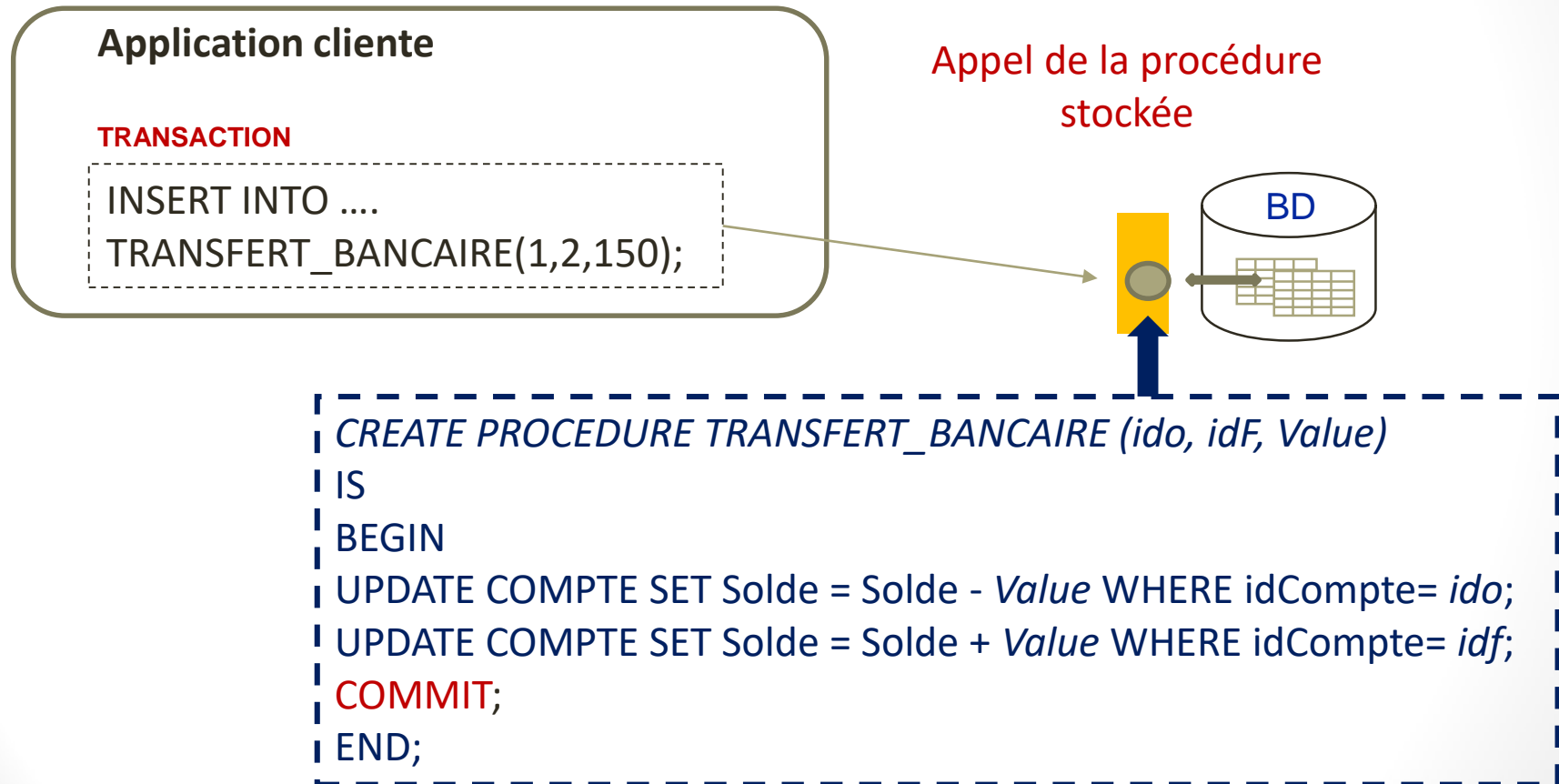


```
CREATE PROCEDURE TRANSFERT_BANCAIRE (ido, idF, Value)  
IS  
BEGIN  
  UPDATE COMPTE SET Solde = Solde - Value WHERE idCompte= ido;  
  UPDATE COMPTE SET Solde = Solde + Value WHERE idCompte= idf;  
END;  
/
```

- Le COMMIT valide les ordres UPDATE de la procédure
- Un ROLLBACK annulerait les autres UPDATE de la procédure

# APPEL DES TRANSACTIONS (3/4)

*Il est déconseillé (mais sujet à discussion) de mettre la validation (COMMIT) dans les procédures stockées (INTERDIT dans les TRIGGERS)*



**ATTENTION :** Le commit dans la procédure valide la transaction en cours donc le INSERT.

# APPEL DES TRANSACTIONS (4/4)

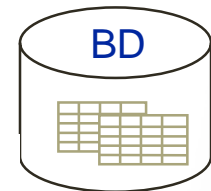
*Dans une programmation client, la gestion de la transaction doit se faire également au niveau le plus haut.*

## Exemple : Application cliente JAVA

### TRANSACTION : TRANSFERT BANCAIRE

```
Try { ...  
  cx.setAutoCommit (false);  
  
  Statement etatReq=cx.createStatement();  
  etatReq.executeUpdate("UPDATE COMPTE SET .....")  
  etatReq.executeUpdate("UPDATE COMPTE SET .....")  
  
  cx.commit();  
} catch (SQLException ex) { cx.rollback;}
```

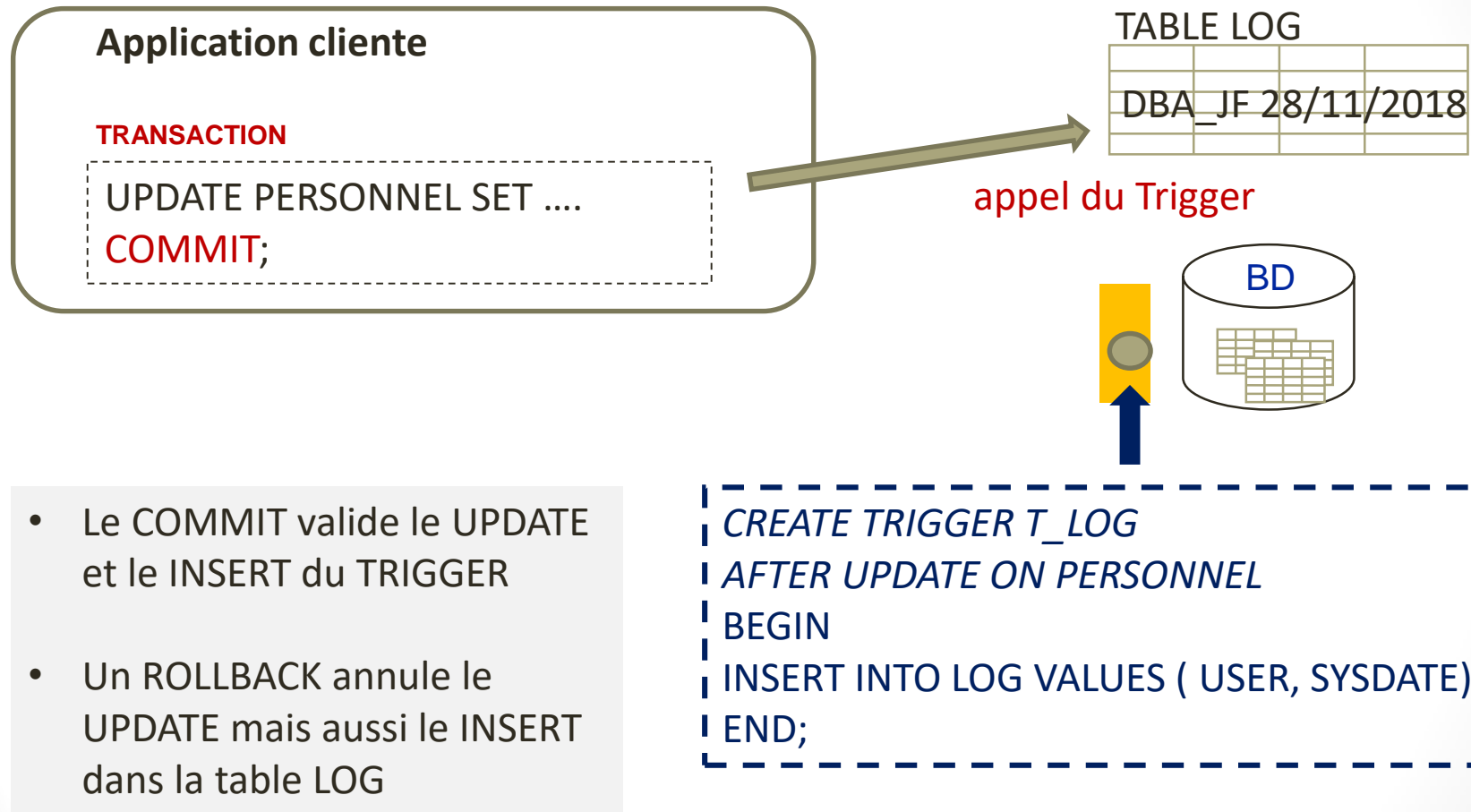
Connexion JDBC



ICI, la gestion de la transaction s'effectue au niveau du programme JAVA

# TRANSACTIONS AUTONOMES (1/2)

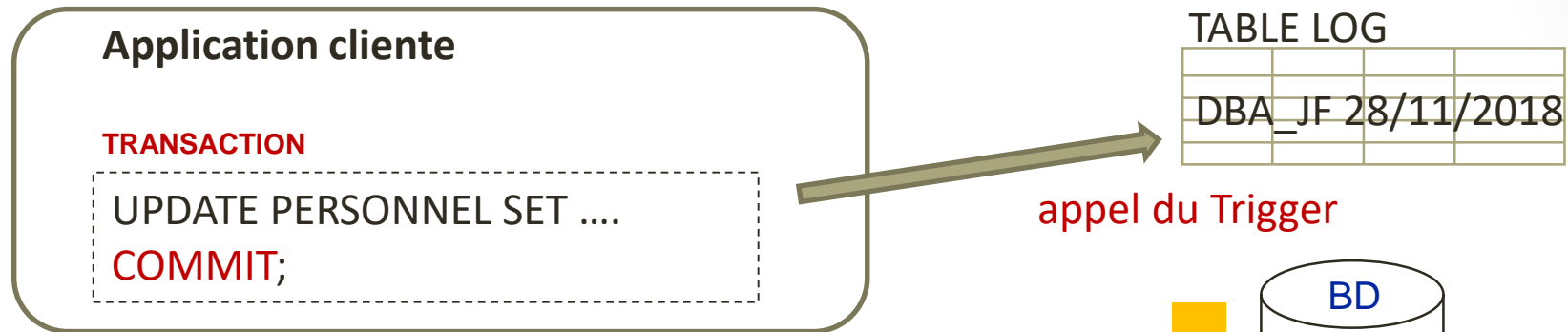
Le trigger *T\_LOG* permet d'historiser une tentative de MAJ sur la table *PERSONNEL*



**PROBLEME :** l'exécution du trigger devrait être autonome. L'annulation de la transaction doit garder la trace dans le LOG

# TRANSACTIONS AUTONOMES (2/2)

Le trigger `T_LOG` permet d'historiser une tentative de MAJ sur la table `PERSONNEL`



- Le trigger est déclaré comme une transaction autonome
- Un ROLLBACK de la transaction préserve le INSERT dans la table LOG
- Tous les SGBDs n'implémentent pas les transactions autonomes
- ORACLE : OUI

```
CREATE TRIGGER T_LOG  
AFTER UPDATE ON PERSONNEL  
DECLARE  
PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
INSERT INTO LOG VALUES ( USER, SYSDATE);  
COMMIT;  
END;
```

# CONCLUSION

1. Les SGBD relationnels gèrent correctement les transactions.
2. Le verrouillage doit donc être laissé en priorité à la charge du SGBD.
3. Le développeur gère la transaction au niveau le plus haut en général.
4. Le développeur désactive le mode AUTOCOMMIT .
5. Le développeur pense à utiliser pleinement l'aspect ensembliste du SQL dans son programme pour manipuler les données de manière performante.
6. Le développeur pense à faire des requêtes performantes.
7. Le développeur ou DBA ont mis en place les bons index.
8. A la création des tables, le plus de CI ont été mises en place pour que l'optimiseur puisse s'en servir.

```
Try { ...  
cx.setAutoCommit (false);  
Statement etatReq=cx.createStatement();  
etatReq.executeUpdate("UPDATE COMPTE SET .....")  
etatReq.executeUpdate("UPDATE COMPTE SET .....")  
cx.commit();  
} catch (SQLException ex) { cx.rollback;}
```

Connexion JDBC

