Transactions

Les transactions

- En mode, classique, les requêtes s'enchaînent. La première peut fonctionner alors que la suivante peut rencontrer une erreur. La base de données contient alors des données dans certaines tables et pas dans d'autres.
  - Afin de palier à ce problème, il est possible d'utiliser **des transactions.**
- Au sens SGBD du terme, une transaction est un bloc d'instructions faisant passer la base de données d'un état initial (cohérent) à un état intermédiaire ou final cohérent.
  - Si un problème logiciel ou matériel survient au cours d'une transaction, aucune des instructions de la transaction ne doit être effectuée. En invalidant toutes les opérations depuis le début de la transaction, la base retourne à son état initial cohérent (**suivant le principe du tout ou rien**).

Les transactions

- Une transaction permet d'effectuer des opération en mode hypothétique sur la base. Ce n'est que lorsque l'ordre de confirmation (ou d'annulation) sera transmis que les modifications (ou non) prendront effet sur la base de données.
- Un exemple classique de transaction est l'opération qui transfère une somme S d'un compte bancaire A à un compte bancaire B :

```
début transaction
    solde(A) = solde(A) - S
    solde(B) = solde(B) + S
fin transaction
```

Il est clair que cette opération ne doit pas être interrompue entre le débit de A et le crédit de B.

Déroulement d'une transaction

1. On démarre une transaction: START TRANSACTION
2. On exécute les requêtes désirées une à une.
3. Si une des requêtes échoue, on annule toutes les requêtes (ROLLBACK), et on termine la transaction.
4. Par contre, si à la fin des requêtes, tout s'est bien passé, on valide (COMMIT) tous les changements, et on termine la transaction.
5. Si le traitement est interrompu (entre deux requêtes par exemple), les changements ne sont jamais validés, et donc les données de la base restent les mêmes qu'avant la transaction.

Instructions des transactions

Instruction	Objectif
`SET AUTOCOMMIT = {0 | 1}`	Change le mode de validation (1 par défaut, 0 à adopter pour contrôler une transaction)
`SET TRANSACTION ISOLATION LEVEL [ option ];`	Mise en place du niveau d'isolation
`START TRANSACTION [ options ];`	Début de la transaction (avec options possibles qui seront étudiées plus loin)
`SAVEPOINT [ nom_savepoint ];`	Déclare un point de validation au cours de la transaction
`COMMIT [ options ];`	Termine avec succès la transaction (validation)
`ROLLBACK [ options ];`	Termine avec échec la transaction (invalidation)

Le mode autocommit

- Par défaut MySQL ne travaille pas avec les transactions. Chaque requête effectuée est **directement commitée** (validée). On ne peut pas revenir en arrière.
- Par défaut, MySQL est donc en mode "**autocommit**". Pour quitter ce mode, il suffit de lancer la requête suivante : SET autocommit=0;
- En désactivant le mode autocommit, en réalité, on démarre une transaction. Et chaque fois que l'on fait un ROLLBACK ou un COMMIT (ce qui met fin à la transaction), une nouvelle transaction est créée automatiquement, et ce tant que la session est ouverte et tant que le mode autocommit n'a pas été remi a 1: SET autocommit=1;

Démarrer explicitement une transaction

- Il est également possible de démarrer explicitement une transaction, auquel cas on peut laisser le mode autocommit activé.
- Avec MySQL une transaction est démarrée avec START TRANSACTION;
- Une fois la transaction ouverte, les requêtes devront être validées pour prendre effet.
- Attention au fait qu'un COMMIT ou un ROLLBACK met fin automatiquement à la transaction, donc les commandes suivantes seront à nouveau commitées automatiquement si une nouvelle transaction n'est pas ouverte (car par défaut en mode autocommit est à 1 mais reste à 0 s'il y avait été positionné avant).

Exemple 1

- **DELETE FROM Passagers;**
  START TRANSACTION;
  INSERT INTO Passagers VALUES
  ('AF6140','20121228',‘A14',220.00);
  COMMIT;
  INSERT INTO Passagers VALUES
  ('AF6140','20121228',‘C45',189.00);

- **SELECT * FROM Passagers;**

Vol_num	Vol_date	Cli_code	Prix
AF6140	2012-12-28	A14	220,00
AF6140	2012-12-28	C45	189,00

Exemple 2

- **DELETE FROM Passagers;**
  SET autocommit=0;
  START TRANSACTION;
  INSERT INTO Passagers VALUES
  ('AF6140','20121228',‘A14',220.00);
  COMMIT;
  INSERT INTO Passagers VALUES
  ('AF6140','20121228',‘C45',189.00);

- **SELECT * FROM Passagers;**

Vol_num	Vol_date	Cli_code	Prix
AF6140	2012-12-28	Brouard	220,00

Exemple 3

- **DELETE FROM Passagers;**
  INSERT INTO Passagers VALUES ('AF6140','20121228','A14',220.00);
  START TRANSACTION;
  INSERT INTO Passagers VALUES ('AF6140','20121228','B18',256.00);
  INSERT INTO Passagers VALUES ('AF6140','20121228',‘C45',189.00);
  DELETE FROM Passgers WHERE Cli_code = 'A14‘;
  ROLLBACK;

- **SELECT * FROM Passagers;**

Vol_num	Vol_date	Cli_code	Prix
AF6140	2012-12-28	A14	220,00

Exemple 4: Exceptions nommées

```
CREATE PROCEDURE bdsoutou.procExceptionNommee
      (IN p_comp VARCHAR(4))
BEGIN
 DECLARE flagerr BOOLEAN DEFAULT 0;
 BEGIN
   DECLARE erreur_ilResteUnPilote CONDITION
           FOR SQLSTATE '23000';
   DECLARE EXIT HANDLER
           FOR erreur_ilResteUnPilote SET flagerr :=1;
   SET AUTOCOMMIT=0;
    DELETE FROM Compagnie WHERE comp = p_comp;
    SELECT CONCAT('Compagnie ',p_comp, ' détruite')
           AS 'Resultat procExceptionNommee';
  END;
  IF flagerr THEN
    ROLLBACK;
     SELECT CONCAT('Désolé, il reste encore un pilote à
                    la compagnie ',p_comp)
          AS 'Resultat procExceptionNommee';
   ELSE
    COMMIT;
 END IF;
END;
```

Il s'agit de contrôler si la compagnie à détruire est encore rattachée à un enregistrement référencé dans la table Pilote. Donc soit on supprime tout ou rien !

Les transactions en interblocage

- Attention : lorsqu'une modification sur une table est en cours, les données sont verrouillées en lecture et en écriture.
- Il est possible de rencontrer des situation de verrouillages mutuels entre deux transactions : les **interblocages** (dead-lock).
- Dans ce cas, le système de base de données le repère et émet automatiquement un rollback sur l'une des transactions en cours d'exécution.

Les transactions en interblocage

Transaction 1	Transaction 2
START TRANSACTION; UPDATE T_vols SET vol_places_libres = vol_places_libres - 7 WHERE vol_num = 'AF6140' AND vol_datevol ='2012-12-28'; SELECT SLEEP(5); UPDATE T_vols SET vol_places_libres = vol_places_libres - 4 WHERE vol_num = 'AF6144' AND vol_datevol ='2012-12-28'; COMMIT;	START TRANSACTION; UPDATE T_vols SET vol_places_libres = vol_places_libres - 5 WHERE vol_num = 'AF6144' AND vol_datevol ='2012-12-28'; SELECT SLEEP(5); UPDATE T_vols SET vol_places_libres = vol_places_libres - 3 WHERE vol_num = 'AF6140' AND vol_datevol ='2012-12-28'; COMMIT;

Les transactions en interblocage

- Quand MySQL identifie un tel phénomène:
  1. Il met fin à une des deux transactions.
  2. Le message ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction est retourné au client
  3. L'autre transaction termine son exécution

Triggers

Les triggers

- Un trigger (déclencheur) peut être considéré comme un sous-programme associé à un événement particulier sur la base (action sur une table ou une vue, par exemple).
- C'est une règle, dite active, de la forme:
  - **« événement - condition - action »**
  - L'action est déclenchée à la suite de l'événement, si la condition est vérifiée.
  - Une action peut être une vérification ou une mise à jour.
- À la différence des sous-programmes, l'exécution d'un déclencheur n'est pas explicite (par CALL par exemple). Un trigger est **activé automatiquement par une requête de mise à jour**.

L'utilité des triggers

- Un trigger permet de :
  - Programmer toutes les règles de gestion qui n'ont pas pu être mises en place par des contraintes au niveau des tables. Par exemple, la condition: *une compagnie ne fait voler un pilote que s'il a totalisé plus de 60 heures de vol dans les 2 derniers mois, sur le type d'appareil du vol en question,* ne pourra pas être programmée par une contrainte et nécessitera l'utilisation d'un déclencheur.
  - Déporter des contraintes au niveau du serveur et alléger ainsi la programmation client.
  - Renforcer des aspects de sécurité et d'audit.
  - Programmer l'intégrité référentielle et la réplication dans des architectures distribuées, avec l'utilisation de liens de bases de données.

Les événements déclencheurs

- Les événements déclencheurs peuvent être :
  - une instruction INSERT, UPDATE, ou DELETE sur une table (ou vue). On parle de **déclencheurs LMD**;
  - une instruction CREATE, ALTER, ou DROP sur un objet (table, vue, index, etc.). On parle de **déclencheurs LDD**;
  - le démarrage ou l'arrêt de la base (*startup* ou *shutdown*), une erreur spécifique (*not found*, *duplicate key*, etc.), une connexion ou une déconnexion d'un utilisateur. On parle de **déclencheurs d'instances**.

Syntaxe

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name
     trigger_time trigger_event
     ON tbl_name
     FOR EACH ROW
     BEGIN
        trigger_body
     END
```

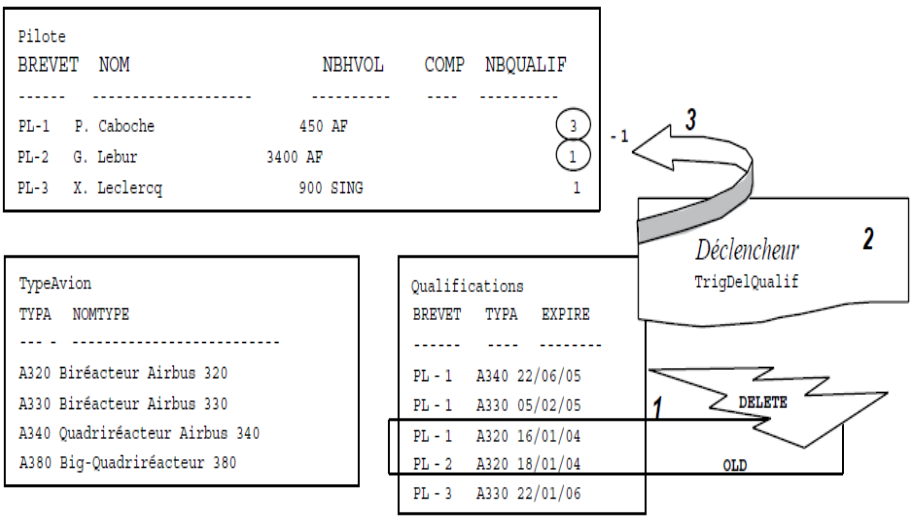
trigger_time: { BEFORE | AFTER }
trigger_event: { INSERT | UPDATE | DELETE}
trigger_order**:** { FOLLOWS | PRECEDES }

Etapes de définition des triggers

- Pour définir un trigger, il faut :
  - spécifier l'événement qui déclenche l'action en indiquant le type de la mise à jour (INSERT, UPDATE, DELETE), le nom de la relation et éventuellement le nom des attributs mis à jour ;
  - indiquer si l'action est réalisée avant, après ou à la place de la mise à jour ;
  - donner un nom à l'ancien et au nouveau tuple (uniquement le nouveau en cas d'insertion et uniquement l'ancien en cas de suppression) ;
  - décrire la condition sous laquelle se déclenche l'événement sous la forme d'une expression SQL booléenne, c-à-d. une expression pouvant être placée dans une clause WHERE ;
  - décrire l'action à réaliser sous la forme d'une procédure SQL ;
  - indiquer si l'action est réalisée pour chaque tuple mis à jour ou une seule fois pour la requête.

Exemple: directive OLD

- La suppression d'une qualification d'un pilote doit lancer le trigger **TrigDelQualif** qui décrémente le compteur *nbQualif* automatiquement.



Exemple: directive OLD

Code MySQL	Commentaires

`CREATE TRIGGER TriDelQualif`
`AFTER DELETE ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

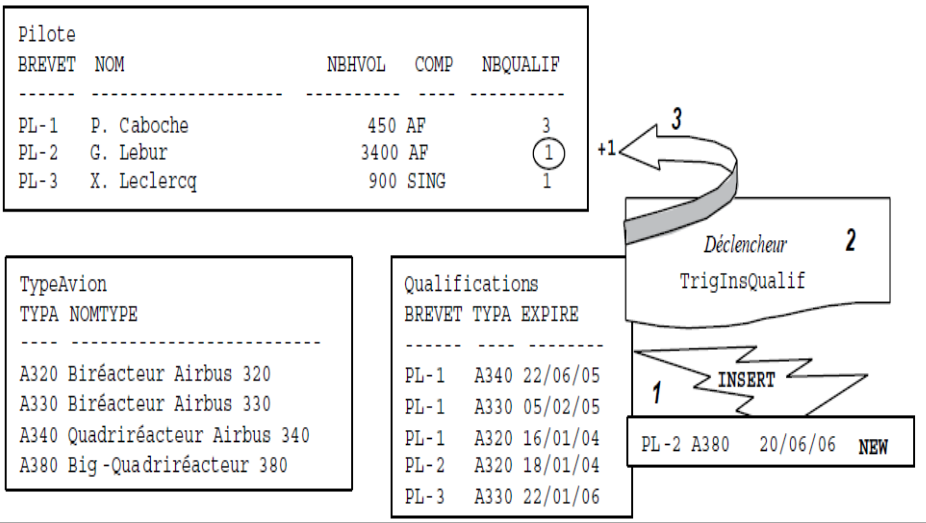
`BEGIN`
`UPDATE Pilote SET nbQualif = nbQualif - 1`
`WHERE brevet = OLD.brevet ;`
`END;`
	Corps du déclencheur.
	Mise à jour du pilote concerné par la suppression.

Événement déclencheur	Résultat

`DELETE FROM Qualifications`
`WHERE typa = 'A320';`
	`SELECT * FROM Pilote;`
	`+--------+-------------+---------+-------+----------+`
	`| brevet | nom | nbHVol | compa | nbQualif |`
	`+--------+-------------+---------+-------+----------+`
	`| PL-1 | P. Caboche | 450.00 | AF | 2 |`
	`| PL-2 | G. Lebur | 3400.00 | AF | 0 |`
	`| PL-3 | X. Leclercq | 900.00 | SING | 1 |`
	`+--------+-------------+---------+-------+----------+`

Exemple: directive NEW

- L'ajout d'une qualification à un pilote doit lancer le trigger **TrigInsQualif** qui incrémente le compteur *nbQualif* automatiquement.



Exemple: directive NEW

Déclencheur	Commentaires

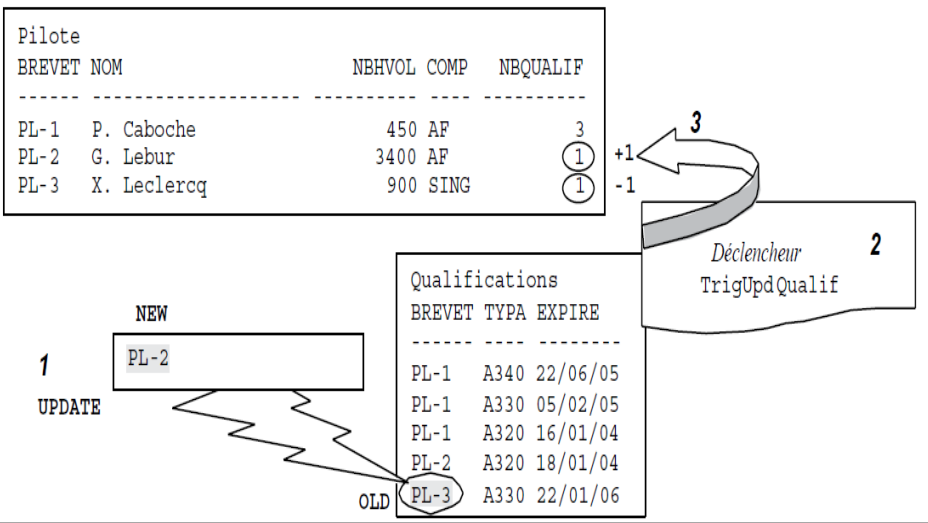
`CREATE TRIGGER TrigInsQualif`
`AFTER INSERT ON Qualifications`
`FOR EACH ROW`
`BEGIN`
`UPDATE Pilote SET nbQualif = nbQualif + 1`
`WHERE brevet = NEW.brevet;`
`END;`
	Déclaration de l'événement déclencheur.
	
	Corps du déclencheur.
	Mise à jour du pilote concerné par la qualification.

Événement déclencheur	Résultat
`INSERT INTO Qualifications VALUES ('PL-2', 'A380', SYSDATE());`	`SELECT * FROM Pilote$`

```
+--------+-------------+---------+-------+----------+
| brevet | nom         | nbHVol  | compa | nbQualif |
+--------+-------------+---------+-------+----------+
| PL-1   | P. Caboche  |  450.00 | AF    |        3 |
| PL-2   | G. Lebur    | 3400.00 | AF    |        2 |
| PL-3   | X. Leclercq |  900.00 | SING  |        1 |
+--------+-------------+---------+-------+----------+
```

Exemple: directive OLD et NEW à la fois

- La modifiction d'une qualification d'un pilote doit lancer le trigger **TrigUpdQualif** qui met a jour le compteur *nbQualif* automatiquement.



Exemple: directive OLD et NEW à la fois

Déclencheur	Commentaires

`CREATE TRIGGER TrigUpdQualif`
`AFTER UPDATE ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`BEGIN`	Corps du déclencheur.

`UPDATE Pilote`
`SET nbQualif = nbQualif + 1`
`WHERE brevet = NEW.brevet;`
	Mise à jour des pilotes concernés par la modification de la qualification.

`UPDATE Pilote`
`SET nbQualif = nbQualif - 1`
`WHERE brevet = OLD.brevet;`	

`END;`	

Exemple: directive OLD et NEW à la fois

Événement déclencheur	Résultat

```
UPDATE Qualifications
  SET brevet = 'PL-2'
  WHERE brevet = 'PL-3'
  AND typa  = 'A330'$
```

```
SELECT * FROM Pilote$
+--------+-------------+---------+-------+----------+
| brevet | nom         | nbHVol  | compa | nbQualif |
+--------+-------------+---------+-------+----------+
| PL-1   | P. Caboche  |  450.00 | AF    |        3 |
| PL-2   | G. Lebur    | 3400.00 | AF    |        2 |
| PL-3   | X. Leclercq |  900.00 | SING  |        0 |
+--------+-------------+---------+-------+----------+

SELECT * FROM Qualifications$
+--------+------+------------+
| brevet | typa | expire     |
+--------+------+------------+
| PL-1   | A340 | 2005-06-22 |
| PL-1   | A330 | 2005-02-05 |
| PL-1   | A320 | 2004-01-16 |
| PL-2   | A320 | 2004-01-18 |
| PL-2   | A330 | 2006-01-22 |
+--------+------+------------+
```


Appel de sous-programmes

- **progTrigg** ajoute simplement une ligne dans la table Trace.

```
CREATE PROCEDURE bdsoutou.procTrigg(IN param DATETIME)
BEGIN
  INSERT INTO Trace VALUES
        (CONCAT('Insertion pilote, appel de bdsoutou.procTrigg le ',param));
END;
```

- L'utilisation de la procédure (**procTrigg**) dans le déclencheur (**espionAjoutPilote**) qui s'exécutera avant chaque ajout d'un nouveau pilote.

```
CREATE TRIGGER bdsoutou.espionAjoutPilote
 BEFORE INSERT ON Pilote
 FOR EACH ROW
BEGIN
 CALL bdsoutou.procTrigg(SYSDATE());
END;
```

Curseurs

Généralités

- Un curseur est une zone mémoire qui est générée côté serveur (mise en cache) et qui permet de traiter une instruction **SELECT renvoyant un nombre quelconque d'enregistrements**.
- Un programme peut travailler avec plusieurs curseurs en même temps.
- Un curseur, durant son existence (de l'ouverture à la fermeture), contient en permanence l'adresse de la ligne courante.
- Protocole d'utilisation:
  1. Déclaration
  2. Ouverture
  3. Utilisation
  4. Fermeture

Principes d'un curseur

```
CREATE  PROCEDURE  nomSousProgramme   [(...)]
 BEGIN
  DECLARE v1...;
  DECLARE v2...;
  DECLARE v3...;
  DECLARE  CURSOR  curs1  FOR  SELECT brevet,nbhVol,    comp
                               FROM  Pilote  WHERE  comp = 'AF' ;
  ...

   ...
   OPEN  curs1 ;
   FETCH  curs1  INTO  v1,v2,v3 ;
   ...
 END;
$
```

curs1 *Curseur*

PL-1	2450	AF
PL-2	900	AF
PL-5	200	AF

v1	v2	v3
PL-1	2450	AF

- Le curseur est décrit après les variables;
- Il est ouvert dans le code du programme ;
- Le programme peut parcourir tout le curseur en récupérant les lignes une par une dans une variable locale. Le curseur est ensuite fermé.

Propriétés des curseurs MySQL

- Tout curseur MySQL dispose des propriétés suivantes :
  - **lecture seule** : aucune modification dans la base n'est possible à travers ce dernier ;
  - **non navigable** : une fois ouvert, le curseur est parcouru du début à la fin sans pouvoir revenir à l'enregistrement précédent ;
  - **insensible** (*asensitive*) : toute mise à jour opérée dans la base n'est pas répercutée dans le curseur une fois ouvert.

Instructions pour les curseurs

Instructions	Commentaires et exemples

`CURSOR FOR requête;`	Déclaration du curseur.
	`DECLARE curs1 CURSOR FOR`
	`SELECT brevet,nbHVol,comp`
	`FROM bdsoutou.Pilote WHERE comp = 'AF';`

`OPEN nomCurseur;`	Ouverture du curseur (chargement des lignes). Aucune exception n'est levée si la requête ne ramène aucune ligne. `OPEN curs1;`
`FETCH nomCurseur INTO listeVariables;`	Positionnement sur la ligne suivante et chargement de l'enregistrement courant dans une ou plusieurs variables. `FETCH curs1 INTO var1,var2,var3;`
`CLOSE nomCurseur;`	Ferme le curseur. L'exception « `ERROR 1326 (24000): Cursor is not open` » se déclenche si des accès au curseur sont opérés après sa fermeture. `CLOSE curs1;`

Parcours d'un curseur

- Le parcours d'un curseur nécessite une des structures répétitives (tant que, répéter ou boucle loop).
- Il faut obligatoirement utiliser une exception (**handler** dans le vocable de MySQL) qui force le programme à continuer si on arrive en fin du curseur (au-delà du dernier enregistrement), tout en positionnant la variable curs1 à vrai (1).
- MySQL ne propose pas, pour l'heure, de fonctions sur les curseurs (FOUND, NOT_FOUND, IS_CLOSED, etc.).

Parcours d'un curseur: Exemple

Répéter **Tant que**

```
DECLARE fincurs1 BOOLEAN DEFAULT 0;
DECLARE v_nbHv   DECIMAL(7,2);
DECLARE v_tot    DECIMAL(8,2) DEFAULT 0;

DECLARE curs1 CURSOR FOR
        SELECT nbHVol FROM bdsoutou.Pilote WHERE comp = 'AF';
DECLARE CONTINUE HANDLER FOR NOT FOUND SET fincurs1 := 1;
OPEN curs1;
```

```
REPEAT
  FETCH curs1 INTO v_nbHv;
  IF NOT fincurs1 THEN
     SET v_tot := v_tot+v_nbHv;
  END IF;
  UNTIL fincurs1
END REPEAT;
```

```
FETCH curs1 INTO v_nbHv;
WHILE (NOT fincurs1) DO
      SET v_tot := v_tot+v_nbHv
      FETCH curs1 INTO v_nbHv;
END WHILE;
```

```
CLOSE curs1;
SELECT v_tot AS 'Total heures pour les pilotes de ''AF''';
```


Accès concurrents [FOR UPDATE]

- Il est souvent intéressant de pouvoir modifier facilement la ligne courante d'un curseur (UPDATE ou DELETE) sans qu'un autre utilisateur ne la modifie en même temps.
- Il faut utiliser la clause **FOR UPDATE :**
  - Elle s'emploie lors de la déclaration du curseur et verrouille les lignes concernées dès l'ouverture du curseur. Les verrous sont libérés à la fin de la transaction.

Accès concurrents: Exemple

- L'exemple suivant décrit un bloc qui se sert du curseur FOR UPDATE pour :
  - augmenter le nombre d'heures de 100 pour les pilotes de la compagnie de code 'AF' ;
  - diminuer ce nombre de 100 pour les pilotes de la compagnie de code 'SING' ;
  - supprimer les pilotes des autres compagnies.

Code MySQL	Commentaires
BEGIN	
DECLARE fincurs BOOLEAN DEFAULT 0;	
DECLARE v_brevet VARCHAR(6);	
DECLARE v_nbHv DECIMAL(7,2);	
DECLARE v_comp VARCHAR(4);	
DECLARE curs **CURSOR FOR**	
SELECT brevet,nbHVol,comp	
FROM bdsoutou.Pilote **FOR UPDATE**;	Déclaration du curseur avec verrou.
DECLARE CONTINUE HANDLER FOR NOT FOUND	
SET fincurs := 1;	
SET AUTOCOMMIT = 0;	
OPEN curs;	
FETCH curs INTO v_brevet,v_nbHv,v_comp;	
WHILE (NOT fincurs) DO	Chargement et parcours du curseur.
IF (v_comp='AF') THEN	
UPDATE bdsoutou.Pilote	
SET nbHVol = nbHVol + 100	Mise à jour de la table `Pilote` par l'intermédiaire du curseur.
WHERE brevet = v_brevet;	
ELSEIF (v_comp='SING') THEN	
UPDATE bdsoutou.Pilote	
SET nbHVol = nbHVol - 100	
WHERE brevet = v_brevet;	
ELSE	
DELETE FROM bdsoutou.Pilote	
WHERE brevet = v_brevet;	
END IF;	
FETCH curs INTO v_brevet,v_nbHv,v_comp;	
END WHILE;	
CLOSE curs;	
COMMIT;	Validation de la transaction.

Accès concurrents: Restrictions MySQL

- Une validation (COMMIT) avant la fermeture d'un curseur FOR UPDATE aura des effets de bord fâcheux.
- Il n'est pas possible de déclarer un curseur FOR UPDATE en utilisant dans la requête les directives DISTINCT ou GROUP BY, un opérateur ensembliste, ou une fonction d'agrégat.
- Il n'existe pas encore de directive WHERE CURRENT OF pour modifier l'enregistrement courant d'un curseur avec verrou.
- Un curseur, comme un tableau, ne peut pas être passé en paramètre d'un sous-programme ni en entrée (IN), ni en sortie (OUT).

Requêtes préparées

Requêtes préparées

- Les développeurs d'applications écrivent souvent du code qui interagit avec une BD à l'aide de paramètres fournis par les utilisateurs de l'application.
  - En règle générale, les applications BD traitent de gros volumes d'instructions presque identiques, avec uniquement des modifications des valeurs littérales ou variables dans les clauses, telles que WHERE pour les requêtes et les suppressions, SET pour les mises à jour et VALUES pour les insertions.
  - MySQL implémente des requêtes préparées à cet effet.
- Une **requête préparée** (prepared statement), ou requête paramétrée, est utilisée pour exécuter à plusieurs reprises la même requête SQL, avec des paramètres variables, d'une manière extrêmement efficace.
- Elles offrent les principaux avantages suivants:
  - Moins de surcharge pour l'analyse de l'instruction à chaque exécution.
  - Protection contre les attaques par injection SQL.

Utilisation

- Principalement **dans des applications**:
  - A partir d'un programme C (*MySQL C API client library*), Java (*MySQL Connector/J*), .Net (*MySQL Connector/NET*) ou PHP par une API écrite en C (*mysqli extension*).
  - Exemple java:

```
PreparedStatement ps = conn.prepareStatement(
                "UPDATE emp SET sal = ? WHERE name = ?");
```

- Peut être utiliser **dans des Script SQL** pour par exemple:
  - Tester le fonctionnement des requêtes préparées dans votre application avant de la coder.
  - Créer un scénario de test qui reproduit un problème avec les requêtes préparées, afin que vous puissiez déposer un rapport de bug.

Syntaxe

- La construction des (*prepared statements*) est basée sur les trois directives suivantes :

  PREPARE *nomEtat* FROM *étatPréparé*;

  - associe un nom (insensible à la casse) à une instruction dynamique. *étatPréparé* est soit une chaîne soit une variable de session contenant le texte de la requête SQL construite. Dans cette chaîne, le caractère **«?»** (appelé *placeholder*) permet de se substituer à un paramètre.

  EXECUTE *nomEtat* [USING *@var1* [, *@var2*] ...];

  - exécute la requête préparée avec éventuellement la clause USING qui reliera les paramètres aux variables de session.

  {DEALLOCATE | DROP} PREPARE *nomEtat*;

  - supprime le contenu de l'ordre (une fin de session désalloue tous les ordres ouverts).

Exemples

- **DELETE**

```
SET @vs_nbhVol  = 1000 $
CREATE PROCEDURE bdsoutou.sousProg()
BEGIN
 PREPARE etat FROM
        'DELETE FROM Avion WHERE nbHVol > ?';
 EXECUTE etat USING @vs_nbhVol;
 DEALLOCATE PREPARE etat;
END;
```

- **SELECT**

```
SET @vs_chaine =
    'SELECT * FROM Avion WHERE nbHVol=?'$
SET @vs_nbhVol  = 1000$
CREATE PROCEDURE bdsoutou.sousProg()
BEGIN
 PREPARE etat FROM  @vs_chaine;
 EXECUTE etat USING @vs_nbhVol;
 DEALLOCATE PREPARE etat;
END;
```

Exemples

- **UPDATE**

```
SET @vs_chaine =
    'UPDATE Avion SET nbHVol=nbHVol*?
          WHERE immat=?'$
SET @vs_immat     = 'F-GLFS'$
SET @vs_pourcent = 1.1$
```

```
CREATE PROCEDURE bdsoutou.sousProg()
BEGIN
 PREPARE etat FROM  @vs_chaine;
 EXECUTE etat USING @vs_pourcent,@vs_immat;
 DROP PREPARE etat;
END;
```

L'appel de cette procédure aura pour conséquence d'augmenter de 10 % le nombre d'heures de vol de l'avion immatriculé 'F-GLFS'.

Références

- **Références bibliographiques:**
  - **Introduction to Information Systems** [15 ed.], James A. O'Brien, George M. Marakas - McGraw-Hill/Irwin
  - **Bases de données Concepts, utilisation et développement,** Jean-Luc Hainaut – Dunod.
  - **Bases de données - de la modélisation au SQL,** Laurent Audibert – Ellipses
  - **UML 2 pour les bases de données ,** Christian Soutou – Eyrolles
  - **Programmer avec MySQL:** SQL - Transactions - PHP - Java - Optimisations - Avec 40 exercices corrigés**,** Christian Soutou – Eyrolles
- **Références webographiques:**
  - https://dev.mysql.com/doc/refman/8.4/en/
  - https://mysql.developpez.com/cours
  - http://www.mysqltutorial.org
  - https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql

FIN