V.1. Transactions

Pour commencer cette partie, nous allons voir ce que sont les transactions, à quoi elles servent exactement, et comment les utiliser avec MySQL.

Les transactions sont une fonctionnalité absolument indispensable, permettant de sécuriser une application utilisant une base de données. Sans transactions, certaines opérations risqueraient d'être à moitié réalisées, et la moindre erreur, la moindre interruption pourrait avoir des conséquences énormes. En effet, les transactions permettent de regrouper des requêtes dans des blocs, et de faire en sorte que tout le bloc soit exécuté en une seule fois, cela afin de préserver l'intégrité des données de la base.

Les transactions ont été implémentées assez tard dans MySQL, et qui plus est, elles ne sont pas utilisables pour tous les types de tables. C'est d'ailleurs un des principaux arguments des détracteurs de MySQL.

V.1.0.1. Etat actuel de la base de données

Note : les tables de test ne sont pas reprises.

Contenu masqué n°46

V.1.1. Principe

Une transaction, c'est un **ensemble de requêtes** qui sont **exécutées en un seul bloc**. Ainsi, si une des requêtes du bloc échoue, on peut décider d'annuler tout le bloc de requêtes (ou de quand même valider les requêtes qui ont réussi).

?

À quoi ça sert ?

Imaginez que Monsieur Durant fasse un virement de 300 euros à Monsieur Dupont via sa banque en ligne. Il remplit toutes les petites cases du virement, puis valide. L'application de la banque commence à traiter le virement quand soudain, une violente panne de courant provoque l'arrêt des serveurs de la banque.

Deux jours plus tard, Monsieur Durant reçoit un coup de fil de Monsieur Dupont, très énervé, qui lui demande pourquoi le paiement convenu n'a toujours pas été fait. Intrigué, Monsieur Durant va vérifier son compte, et constate qu'il a bien été débité de 300 euros.

? Mais que s'est-il donc passé ?

Normalement, le traitement d'un virement est plutôt simple, deux étapes suffisent :

— étape 1 : on retire le montant du virement du compte du donneur d'ordre ;
— étape 2 : on ajoute le montant du virement au compte du bénéficiaire.

Seulement voilà, pas de chance pour Monsieur Durant, la panne de courant qui a éteint les serveurs est survenue pile entre l'étape 1 et l'étape 2. Du coup, son compte a été débité, mais le compte de Monsieur Dupont n'a jamais été crédité.

La banque de Monsieur Durant n'utilisait pas les transactions. Si c'était le cas, la seconde requête du traitement n'ayant jamais été exécutée, la première requête n'aurait jamais été validée.

V.1.1.0.1. Comment se déroule une transaction ?

Voici un schéma qui devrait vous éclairer sur le principe des transactions.

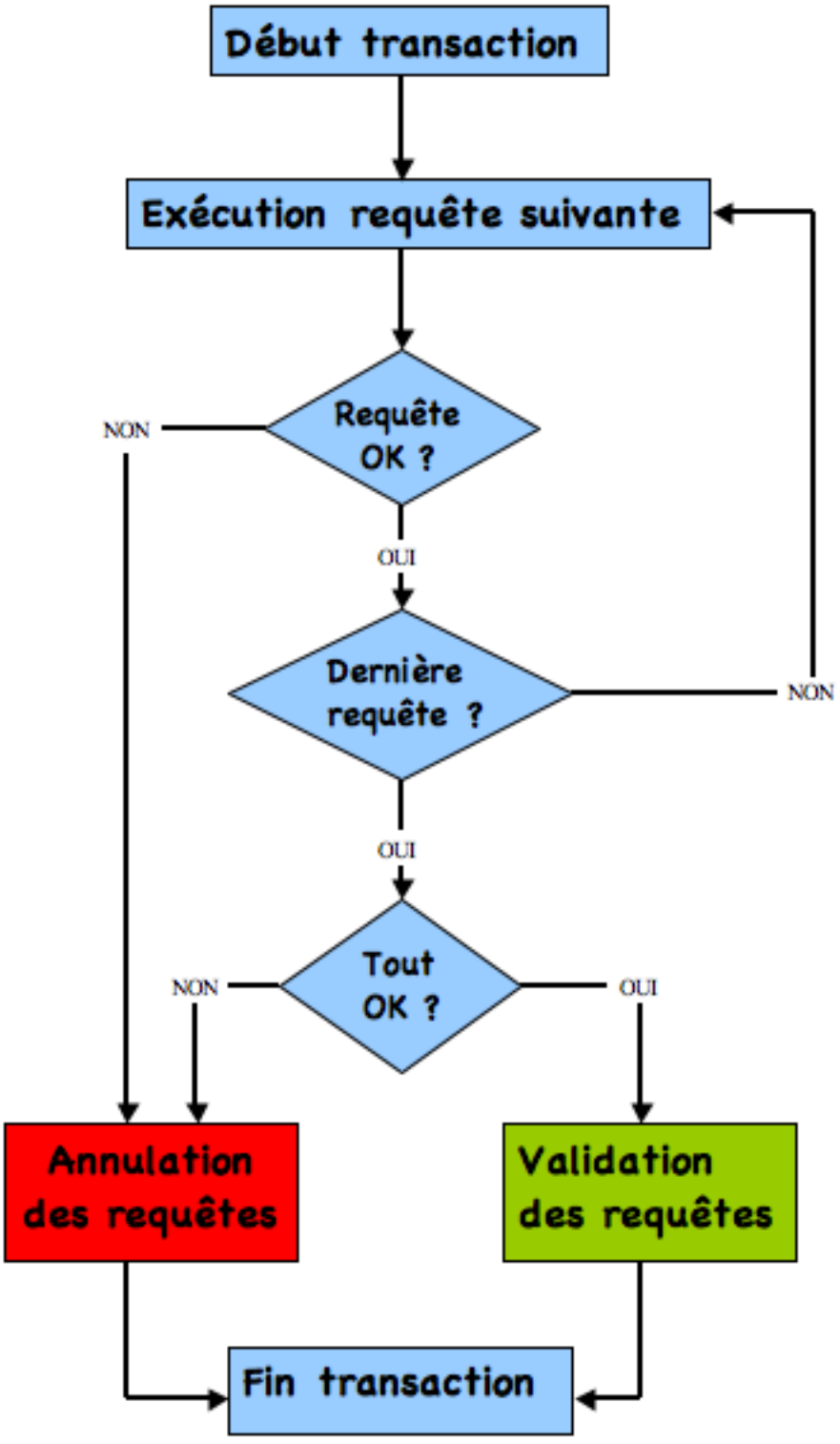


FIGURE V.1.1. – Schéma d'une transaction

— On démarre une transaction.
— On exécute les requêtes désirées une à une.
— Si une des requêtes échoue, on annule toutes les requêtes, et on termine la transaction.
— Par contre, si à la fin des requêtes, tout s'est bien passé, on valide tous les changements, et on termine la transaction.

— 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.

V.1.1.1. Support des transactions

Il n'est pas possible d'utiliser les transactions sur n'importe quelle table. Pour les supporter, une table doit être **transactionnelle**, ce qui, avec MySQL, est **défini par le moteur de stockage** utilisé pour la table.

Rappelez-vous, nous avons vu dans le chapitre sur la création des tables qu'il existait différents moteurs de stockage possibles avec MySQL, dont les plus connus sont **MyISAM** et **InnoDB**.

MyISAM ne supportant pas les contraintes de clés étrangères, nos tables ont été créées avec le moteur InnoDB, ce qui tombe plutôt bien pour la suite de ce chapitre. En effet :

— les tables MyISAM sont non-transactionnelles, donc ne supportent pas les transactions ;
— les tables InnoDB sont transactionnelles, donc supportent les transactions.

V.1.2. Syntaxe et utilisation

V.1.2.0.1. Vocabulaire

Lorsque l'on valide les requêtes d'une transaction, on dit aussi que l'on **commite** les changements. À l'inverse, l'annulation des requêtes s'appelle un **rollback**.

V.1.2.0.2. Comportement par défaut

Vous l'aurez compris, par défaut MySQL ne travaille pas avec les transactions. Chaque requête effectuée est directement **commitée** (validée). On ne peut pas revenir en arrière. On peut donc en fait considérer que chaque requête constitue une transaction, qui est automatiquement commitée. Par défaut, MySQL est donc en mode "**autocommit**".

Pour quitter ce mode, il suffit de lancer la requête suivante :

```
1 SET autocommit=0;
```

Une fois que vous n'êtes plus en autocommit, chaque modification de donnée devra être commitée pour prendre effet. Tant que vos modifications ne sont pas validées, vous pouvez à tout moment les annuler (faire un rollback).

V.1.2.1. Valider/annuler les changements

Les commandes pour commiter et faire un rollback sont relativement faciles à retenir :

```
COMMIT;   -- pour valider les requêtes
ROLLBACK; -- pour annuler les requêtes
```

V.1.2.1.1. Exemples de transactions en mode non-autocommit

Si ce n'est pas déjà fait, changez le mode par défaut de MySQL grâce à la commande que nous venons de voir.

Première expérience : annulation des requêtes.

Exécutez ces quelques requêtes :

```
INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
VALUES ('Baba', 5, '2012-02-13 15:45:00', 'F');
INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
VALUES ('Bibo', 5, '2012-02-13 15:48:00', 'M');
INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
VALUES ('Buba', 5, '2012-02-13 18:32:00', 'F'); -- Insertion de 3
    rats bruns

UPDATE Espece
SET prix = 20
WHERE id = 5;  -- Les rats bruns coûtent maintenant 20 euros au
    lieu de 10
```

Faites maintenant un `SELECT` sur les tables *Espece* et *Animal*.

```
SELECT *
FROM Animal
WHERE espece_id = 5;

SELECT *
FROM Espece
WHERE id = 5;
```

Les changements faits sont bien visibles. Les rats bruns valent maintenant 20 euros, et nos trois nouvelles bestioles ont bien été insérées. Cependant, un simple rollback va annuler ces changements.

```
ROLLBACK;
```

Nos rats coûtent à nouveau 10 euros et Baba, Bibo et Buba ont disparu.

Deuxième expérience : Interruption de la transaction.

Exécutez à nouveau les trois requêtes `INSERT` et la requête `UPDATE`. Ensuite, quittez votre client MySQL (fermez simplement la fenêtre, ou tapez `quit` ou `exit`).

Reconnectez-vous et vérifiez vos données : les rats valent 10 euros, et Baba, Bibo et Buba n'existent pas. Les changements n'ont pas été commités, c'est comme s'il ne s'était rien passé !

Le mode autocommit est de nouveau activé ! Le fait de faire `SET autocommit = 0;` n'est valable que pour la session courante. Or, en ouvrant une nouvelle connexion, vous avez créé une nouvelle session. Désactivez donc à nouveau ce mode.

Troisième expérience : validation et annulation.

Exécutez la séquence de requêtes suivante :

```
1  INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
2  VALUES ('Baba', 5, '2012-02-13 15:45:00', 'F');
3  INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
4  VALUES ('Bibo', 5, '2012-02-13 15:48:00', 'M');
5  INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
6  VALUES ('Buba', 5, '2012-02-13 18:32:00', 'F'); -- Insertion de 3
       rats bruns
7
8  COMMIT;
9
10 UPDATE Espece
11 SET prix = 20
12 WHERE id = 5;  -- Les rats valent 20 euros
13
14 ROLLBACK;
```

Si vous n'avez pas oublié de réactiver le mode non-autocommit, vous avez maintenant trois nouveaux rats bruns (les requêtes d'insertion ayant été validées), et ils ne valent toujours que 10 euros chacun (la modification de l'espèce ayant été annulée).

Quatrième expérience : visibilité des changements non-commités.

Exécutez la requête suivante :

```
1  UPDATE Animal
2  SET commentaires = 'Queue coupée'
3  WHERE nom = 'Bibo' AND espece_id = 5;
```

Ensuite, tout en laissant ce client MySQL ouvert, ouvrez-en un deuxième. Connectez-vous comme d'habitude à la base de données *elevage*. Vous avez maintenant deux sessions ouvertes, connectées à votre base de données. Sélectionnez les rats bruns.

```sql
SELECT id, sexe, nom, commentaires, espece_id, race_id
FROM Animal
WHERE espece_id = 5;
```

id	sexe	nom	commen-taires	espece_id	race_i
69	F	Baba	NULL	5	NULL
70	M	Bibo	NULL	5	NULL
71	F	Buba	NULL	5	NULL

Les commentaires de Bibo sont toujours vides. Les changements non-commités ne sont donc pas visibles à l'extérieur de la transaction qui les a faits. En particulier, une autre session n'a pas accès à ces changements.

Annulez la modification de Bibo dans la première session avec un `ROLLBACK`. Vous pouvez fermer la seconde session.

V.1.2.2. Démarrer explicitement une transaction

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.

Il est également possible de démarrer explicitement une transaction, auquel cas on peut laisser le mode autocommit activé, et décider au cas par cas des requêtes qui doivent être faites dans une transaction.

Repassons donc en mode autocommit :

```sql
SET autocommit=1;
```

Pour démarrer une transaction, il suffit de lancer la commande suivante :

```sql
START TRANSACTION;
```

Avec MySQL, il est également possible de démarrer une transaction avec `BEGIN` ou `BEGIN WORK`. Cependant, il est conseillé d'utiliser plutôt `START TRANSACTION`, car il s'agit de la commande SQL standard.

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.

V.1.2.2.1. Exemples de transactions en mode autocommit

```sql
-- Insertion d'un nouveau rat brun, plus vieux
INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
VALUES ('Momy', 5, '2008-02-01 02:25:00', 'F');

-- Ouverture d'une transaction
START TRANSACTION;

-- La nouvelle rate est la mère de Buba et Baba
UPDATE Animal
SET mere_id = LAST_INSERT_ID()
WHERE espece_id = 5
AND nom IN ('Baba', 'Buba');

-- On annule les requêtes de la transaction, ce qui termine celle-ci
ROLLBACK;

-- La nouvelle rate est la mère de Bibo
UPDATE Animal
SET mere_id = LAST_INSERT_ID()
WHERE espece_id = 5
AND nom = 'Bibo';

-- Nouvelle transaction
START TRANSACTION;

-- Suppression de Buba
DELETE FROM Animal
WHERE espece_id = 5
AND nom = 'Buba';

-- On valide les requêtes de la transaction, ce qui termine celle-ci
COMMIT;
```

Si vous avez bien suivi, vous devriez savoir les changements qui ont été faits.

Contenu masqué n°47

id	nom	espece_id	mere_id
69	Baba	5	NULL
70	Bibo	5	72
72	Momy	5	NULL

V.1.2.3. Jalon de transaction

Lorsque l'on travaille dans une transaction, et que l'on constate que certaines requêtes posent problème, on n'a pas toujours envie de faire un rollback depuis le début de la transaction, annulant toutes les requêtes alors qu'une partie aurait pu être validée. **Il n'est pas possible de démarrer une transaction à l'intérieur d'une transaction**. Par contre, on peut poser des **jalons de transaction**. Il s'agit de **points de repère**, qui permettent d'annuler toutes les requêtes exécutées depuis ce jalon, et non toutes les requêtes de la transaction.

V.1.2.3.1. Syntaxe

Trois nouvelles commandes suffisent pour pouvoir utiliser pleinement les jalons :

```
SAVEPOINT nom_jalon; -- Crée un jalon avec comme nom "nom_jalon"

ROLLBACK [WORK] TO [SAVEPOINT] nom_jalon; -- Annule les requêtes
    exécutées depuis le jalon "nom_jalon", WORK et SAVEPOINT ne
    sont pas obligatoires

RELEASE SAVEPOINT nom_jalon; -- Retire le jalon "nom_jalon" (sans
    annuler, ni valider les requêtes faites depuis)
```

Exemple : exécutez les requêtes suivantes.

```
START TRANSACTION;

INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
VALUES ('Popi', 5, '2007-03-11 12:45:00', 'M');

SAVEPOINT jalon1;

INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
VALUES ('Momo', 5, '2007-03-12 05:23:00', 'M');

ROLLBACK TO SAVEPOINT jalon1;

```

```
13 INSERT INTO Animal (nom, espece_id, date_naissance, sexe)
14 VALUES ('Mimi', 5, '2007-03-12 22:03:00', 'F');
15
16 COMMIT;
```

On n'utilise qu'une seule transaction, on valide à la fin, et pourtant la seconde insertion n'a pas été faite au final, puisqu'elle a été annulée grâce au jalon. Seuls Popi et Mimi existent.

```
1 SELECT id, sexe, date_naissance, nom, espece_id, mere_id, pere_id
2 FROM Animal
3 WHERE espece_id = 5;
```

id	sexe	date_naissance	nom	espece_id	mere_id	pere_
69	F	2012-02-13 15:45:00	Baba	5	NULL	NULL
70	M	2012-02-13 15:48:00	Bibo	5	72	NULL
72	F	2008-02-01 02:25:00	Momy	5	NULL	NULL
73	M	2007-03-11 12:45:00	Popi	5	NULL	NULL
75	F	2007-03-12 22:03:00	Mimi	5	NULL	NULL

V.1.3. Validation implicite et commandes non-annulables

Vous savez déjà que pour terminer une transaction, il faut utiliser les commandes `COMMIT` ou `ROLLBACK`, selon que l'on veut valider les requêtes ou les annuler.

Ça, c'est la manière classique et recommandée. Mais il faut savoir qu'un certain nombre d'autres commandes auront aussi pour effet de clôturer une transaction. Et pas seulement la clôturer, mais également **valider toutes les requêtes** qui ont été faites dans cette transaction. Exactement comme si vous utilisiez `COMMIT`.

Par ailleurs, ces commandes ne peuvent pas être annulées par un `ROLLBACK`.

V.1.3.0.1. Commandes DDL

Toutes les commandes qui créent, modifient, suppriment des objets dans la base de données valident implicitement les transactions.

Ces commandes forment ce qu'on appelle les requêtes DDL, pour *Data Definition Langage*.

Cela comprend donc :

— la création et suppression de bases de données : `CREATE DATABASE`, `DROP DATABASE` ;
— la création, modification, suppression de tables : `CREATE TABLE`, `ALTER TABLE`, `RENAME TABLE`, `DROP TABLE` ;
— la création, modification, suppression d'index : `CREATE INDEX`, `DROP INDEX` ;
— la création d'objets comme les procédures stockées, les vues, etc., dont nous parlerons plus tard.

De manière générale, tout ce qui influe sur la **structure de la base de données**, et non sur les données elles-mêmes.

V.1.3.0.2. Utilisateurs

La création, la modification et la suppression d'utilisateurs (voir partie 7) provoquent aussi une validation implicite.

V.1.3.0.3. Transactions et verrous

Je vous ai signalé qu'il n'était pas possible d'imbriquer des transactions, donc d'avoir une transaction à l'intérieur d'une transaction. En fait, la commande `START TRANSACTION` provoque également une validation implicite si elle est exécutée à l'intérieur d'une transaction. Le fait d'activer le mode *autocommit* (s'il n'était pas déjà activé) a le même effet.

La création et suppression de **verrous de table** clôturent aussi une transaction en la validant implicitement (voir chapitre suivant).

V.1.3.0.4. Chargements de données

Enfin, le chargement de données avec `LOAD DATA` provoque également une validation implicite.

V.1.4. ACID

Derrière ce titre mystérieux se cache un concept très important !

?

Quels sont les **critères** qu'un système utilisant les transactions doit respecter pour être **fiable** ?

Il a été défini que ces critères sont au nombre de quatre : **A**tomicité, **C**ohérence, **I**solation et **D**urabilité. Soit, si on prend la première lettre de chaque critère : ACID. Voyons donc en détail ces quatre critères.

V.1.4.1. A pour Atomicité

Atome signifie étymologiquement "**qui ne peut être divisé**". Une transaction doit être atomique, c'est-à-dire qu'elle doit former une **entité complète et indivisible**. Chaque élément de la transaction, chaque requête effectuée, ne peut exister que dans la transaction.

Si l'on reprend l'exemple du virement bancaire, en utilisant les transactions, les deux étapes (débit du compte donneur d'ordre, crédit du compte bénéficiaire) ne peuvent exister indépendamment l'une de l'autre. Si l'une est exécutée, l'autre doit l'être également. Il s'agit d'un tout.

? Peut-on dire que nos transactions sont atomiques ?

Oui. Si une transaction en cours est interrompue, aucune des requêtes exécutées ne sera validée. De même, en cas d'erreur, il suffit de faire un `ROLLBACK` pour annuler toute la transaction. Et si tout se passe bien, un `COMMIT` validera l'intégralité de la transaction en une fois.

V.1.4.2. C pour cohérence

Les données doivent rester **cohérentes dans tous les cas** : que la transaction se termine sans encombre, qu'une erreur survienne, ou que la transaction soit interrompue. Un virement dont seule l'étape de débit du donneur d'ordre est exécutée produit des données incohérentes (la disparition de 300 euros jamais arrivés chez le bénéficiaire). Avec une transaction, cette incohérence n'apparaît jamais. Tant que la totalité des étapes n'a pas été réalisée avec succès, les données restent dans leur état initial.

? Nos transactions permettent-elles d'assurer la cohérence des données ?

Oui, les changements de données ne sont validés qu'une fois que toutes les étapes ont été exécutées. De l'extérieur de la transaction, le moment entre les deux étapes d'un virement n'est jamais visible.

V.1.4.3. I pour Isolation

Chaque transaction doit être isolée, donc **ne pas interagir avec une autre transaction**.

? Nos transactions sont-elles isolées ?

V.1.4.3.1. Test

Dans votre client MySQL, exécutez les requêtes suivantes (ne commitez pas) pour modifier le *pere_id* du rat Bibo :

Copiez-collez tout le bloc dans votre client MySQL

```sql
START TRANSACTION; -- On ouvre une transaction

UPDATE Animal     -- On modifie Bibo
SET pere_id = 73
WHERE espece_id = 5 AND nom = 'Bibo';

SELECT id, nom, commentaires, pere_id, mere_id
FROM Animal
WHERE espece_id = 5;
```

À nouveau, ouvrez une deuxième session, tout en laissant la première ouverte (démarrez un deuxième client SQL et connectez-vous à votre base de données). Exécutez les requêtes suivantes, pour modifier les *commentaires* de Bibo.

À nouveau, prenez bien tout le bloc d'un coup, vous suivrez plus facilement les explications qui suivent.

```sql
START TRANSACTION;   -- On ouvre une transaction

SELECT id, nom, commentaires, pere_id, mere_id
FROM Animal
WHERE espece_id = 5;

UPDATE Animal       -- On modifie la perruche Bibo
SET commentaires = 'Agressif'
WHERE espece_id = 5 AND nom = 'Bibo';

SELECT id, nom, commentaires, pere_id, mere_id
FROM Animal
WHERE espece_id = 5;
```

Le résultat n'est pas du tout le même dans les deux sessions. En effet, dans la première, on a la confirmation que la requête `UPDATE` a été effectuée :

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Et le `SELECT` renvoie bien les données modifiées (*pere_id* n'est plus `NULL` pour Bibo) :

id	nom	commentaires	pere_id	mere_id
69	Baba	NULL	NULL	NULL
70	Bibo	NULL	73	72
72	Momy	NULL	NULL	NULL
73	Popi	NULL	NULL	NULL
75	Mimi	NULL	NULL	NULL

Par contre, dans la deuxième session, on a d'abord fait un `SELECT`, et Bibo n'a toujours pas de père (puisque ça n'a pas été commité dans la première session). Donc on s'attendrait à ce que la requête `UPDATE` laisse *pere_id* à `NULL` et modifie *commentaires*.

id	nom	commentaires	pere_id	mere_id
69	Baba	NULL	NULL	NULL
70	Bibo	NULL	NULL	72
72	Momy	NULL	NULL	NULL
73	Popi	NULL	NULL	NULL
75	Mimi	NULL	NULL	NULL

Seulement voilà, la requête `UPDATE` ne fait rien ! La session semble bloquée : pas de message de confirmation après la requête `UPDATE`, et le second `SELECT` n'a pas été effectué.

```
mysql>
mysql> UPDATE Animal       -- On modifie Bibo
    -> SET commentaires = 'Agressif'
    -> WHERE espece_id = 5 AND nom = 'Bibo';
_
```

Commitez maintenant les changements dans la première session (celle qui n'est pas bloquée). Retournez voir dans la seconde session : elle s'est débloquée et indique maintenant un message de confirmation aussi :

```
Query OK, 1 row affected (5.17 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Qui plus est, le `SELECT` a été exécuté (vous devrez peut-être appuyer sur Entrée pour que ce soit envoyé au serveur) et les modifications ayant été faites par la session 1 ont été prises en compte : *commentaires* vaut `'Agressif'` et *pere_id* vaut `73` !

id	nom	commentaires	pere_id	mere_id
69	Baba	NULL	NULL	NULL
70	Bibo	Agressif	73	72
72	Momy	NULL	NULL	NULL
73	Popi	NULL	NULL	NULL
75	Mimi	NULL	NULL	NULL

Il est possible que votre seconde session indique ceci : `ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction`. Cela signifie que la session est restée bloquée trop longtemps et que par conséquent la transaction a été automatiquement fermée (avec un rollback des requêtes effectuées). Dans ce cas, recommencez l'opération.

Il n'y a plus qu'à commiter les changements faits par la deuxième session, et c'est terminé ! Si vous ne commitez pas, *commentaires* restera `NULL`. Par contre, *pere_id* vaudra toujours `73` puisque ce changement-là a été commité par la première session.

V.1.4.3.2. Conclusion

La deuxième session n'a pas interagi avec les changements faits par la première session, chaque transaction est bien **isolée**.

Et la première session qui bloque la seconde, ce n'est pas une interaction ça ?

Pas dans le cadre des critères ACID. Oui, la première session provoque un retard dans l'exécution des requêtes de la deuxième session, mais les critères de fiabilité que nous examinons ici concernent **les données impactées par les transactions**, et non le déroulement de celles-ci (qui importe peu finalement). Ce blocage a pour effet d'empêcher la deuxième session d'écraser un changement fait par la première. Donc, ce blocage a bien pour effet l'isolation des transactions.

V.1.4.3.3. Verrous

Le blocage de la deuxième session vient en fait de ce que la première session, en faisant sa requête `UPDATE`, a **automatiquement posé un verrou** sur la ligne contenant Bobi le rat, empêchant toute modification tant que la transaction était en cours. Les verrous faisant l'objet du prochain chapitre, je n'en dis pas plus pour l'instant.

V.1.4.3.4. Utilité

Je vous l'accorde, vous n'allez pas vous amuser tous les jours à ouvrir deux sessions MySQL. Par contre, pour une application pouvant être utilisée par plusieurs personnes en même temps (qui toutes travaillent sur la même base de données), il est impératif que ce critère soit respecté. Prenons l'exemple simple d'un jeu par navigateur : de nombreux joueurs peuvent être connectés en même temps, et effectuer des opérations différentes. Si les transactions ne sont pas isolées, une partie des actions des joueurs risquerait de se voir annulées. On isole donc les transactions grâce aux verrous (qui sont ici automatiquement posés mais ce n'est pas toujours le cas).

V.1.4.4. D pour Durabilité

Une fois la transaction terminée, les données résultant de cette transaction doivent être **stockées de manière durable**, et pouvoir être récupérées, en cas de crash du serveur par exemple.

? Nos transactions modifient-elles les données de manière durable ?

Oui, une fois les changements commités, ils sont stockés définitivement (jusqu'à modification par une nouvelle transaction).

V.1.4.5. En résumé

- Les transactions permettent de **grouper plusieurs requêtes**, lesquelles seront validées (`COMMIT`) ou annulées (`ROLLBACK`) toutes en même temps.
- Tous les changements de données (insertion, suppression, modification) faits par les requêtes à l'intérieur d'une transaction sont **invisibles pour les autres sessions tant que la transaction n'est pas validée**.
- Les transactions permettent d'**exécuter un traitement nécessitant plusieurs requêtes en une seule fois**, ou de l'annuler complètement si une des requêtes pose problème ou si la transaction est interrompue.
- Certaines commandes SQL provoquent une **validation implicite des transactions**, notamment toutes les commandes DDL, c'est-à-dire les commandes qui créent, modifient ou suppriment des objets dans la base de données (tables, index,…).
- Les critères ACID sont les critères qu'un système appliquant les transactions doit respecter pour être fiable : **Atomicité, Cohérence, Isolation, Durabilité**.

Contenu masqué

Contenu masqué n°46

```
SET NAMES utf8;


DROP TABLE IF EXISTS Animal;
DROP TABLE IF EXISTS Race;
DROP TABLE IF EXISTS Espece;


CREATE TABLE Espece (
  id smallint(6) unsigned NOT NULL AUTO_INCREMENT,
  nom_courant varchar(40) NOT NULL,
  nom_latin varchar(40) NOT NULL,
  description text,
  prix decimal(7,2) unsigned DEFAULT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY nom_latin (nom_latin)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=latin1;

LOCK TABLES Espece WRITE;
INSERT INTO Espece VALUES
    (1,'Chien','Canis canis','Bestiole à quatre pattes qui aime les caresses et
(4,'Perroquet amazone','Alipiopsitta xanthops','Joli oiseau parleur vert et jau
UNLOCK TABLES;


CREATE TABLE Race (
  id smallint(6) unsigned NOT NULL AUTO_INCREMENT,
  nom varchar(40) NOT NULL,
  espece_id smallint(6) unsigned NOT NULL,
  description text,
  prix decimal(7,2) unsigned DEFAULT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=latin1;

LOCK TABLES Race WRITE;
INSERT INTO Race VALUES
    (1,'Berger allemand',1,'Chien sportif et élégant au pelage dense, noir-marr
(4,'Bleu russe',2,'Chat aux yeux verts et à la robe épaisse et argentée.',835.0
(8,'Nebelung',2,'Chat bleu russe, mais avec des poils longs...',985.00),(9,'Ro
UNLOCK TABLES;


CREATE TABLE Animal (
  id smallint(6) unsigned NOT NULL AUTO_INCREMENT,
```

```
  sexe char(1) DEFAULT NULL,
  date_naissance datetime NOT NULL,
  nom varchar(30) DEFAULT NULL,
  commentaires text,
  espece_id smallint(6) unsigned NOT NULL,
  race_id smallint(6) unsigned DEFAULT NULL,
  mere_id smallint(6) unsigned DEFAULT NULL,
  pere_id smallint(6) unsigned DEFAULT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY ind_uni_nom_espece_id (nom,espece_id)
) ENGINE=InnoDB AUTO_INCREMENT=63 DEFAULT CHARSET=utf8;

LOCK TABLES Animal WRITE;
INSERT INTO Animal VALUES
    (1,'M','2010-04-05 13:43:00','Rox','Mordille beaucoup',1,1,18,22),(2,NULL,
(4,'F','2009-08-03 05:12:00',NULL,'Bestiole avec une carapace très dure',3,NULL
(7,'F','2008-12-06 05:18:00','Caroline',NULL,1,2,NULL,NULL),(8,'M','2008-09-11
(10,'M','2010-07-21 15:41:00','Bobo',NULL,1,NULL,7,21),(11,'F','2008-02-20 15:4
(13,'F','2007-04-24 12:54:00','Rouquine',NULL,1,1,NULL,NULL),(14,'F','2009-05-2
(16,'F','2009-05-26 08:50:00','Louya',NULL,1,NULL,NULL,NULL),(17,'F','2008-03-1
(19,'F','2009-05-26 09:02:00','Java',NULL,1,2,NULL,NULL),(20,'M','2007-04-24 12
(22,'M','2007-04-24 12:42:00','Bouli',NULL,1,1,NULL,NULL),(24,'M','2007-04-12 0
(26,'M','2006-05-14 15:48:00','Samba',NULL,1,1,NULL,NULL),(27,'M','2008-03-10 1
(29,'M','2009-05-14 06:30:00','Fiero',NULL,2,3,NULL,NULL),(30,'M','2007-03-12 1
(32,'M','2009-07-26 11:52:00','Spoutnik',NULL,3,NULL,52,NULL),(33,'M','2006-05-
(35,'M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance',2,4,N
(38,'F','2009-05-14 06:45:00','Boule',NULL,2,3,NULL,NULL),(39,'F','2008-04-20 0
(41,'F','2006-05-19 15:59:00','Feta',NULL,2,4,NULL,NULL),(42,'F','2008-04-20 03
(44,'F','2006-05-19 16:16:00','Cawette',NULL,2,8,NULL,NULL),(45,'F','2007-04-01
(47,'F','2009-03-26 01:24:00','Scroupy','Bestiole avec une carapace très dure'
(50,'F','2009-05-25 19:57:00','Cheli','Bestiole avec une carapace très dure',3
(54,'M','2008-03-16 08:20:00','Bubulle','Bestiole avec une carapace très dure'
(57,'M','2007-03-04 19:36:00','Safran','Coco veut un gâteau !',4,NULL,NULL,NULL
(60,'F','2009-03-26 07:55:00','Parlotte','Coco veut un gâteau !',4,NULL,NULL,NU
UNLOCK TABLES;


ALTER TABLE Race ADD CONSTRAINT fk_race_espece_id FOREIGN KEY
    (espece_id) REFERENCES Espece (id) ON DELETE CASCADE;

ALTER TABLE Animal ADD CONSTRAINT fk_race_id FOREIGN KEY (race_id)
    REFERENCES Race (id) ON DELETE SET NULL;
ALTER TABLE Animal ADD CONSTRAINT fk_espece_id FOREIGN KEY
    (espece_id) REFERENCES Espece (id);
ALTER TABLE Animal ADD CONSTRAINT fk_mere_id FOREIGN KEY (mere_id)
    REFERENCES Animal (id) ON DELETE SET NULL;
ALTER TABLE Animal ADD CONSTRAINT fk_pere_id FOREIGN KEY (pere_id)
    REFERENCES Animal (id) ON DELETE SET NULL;
```

Retourner au texte.

Contenu masqué n°47

— On a inséré Momy (insertion hors transaction)
— Momy n'est pas la mère de Baba (modification dans une transaction dont les requêtes ont été annulées)
— Momy est la mère de Bibo (modification hors transaction)
— Buba a été supprimée (suppression dans une transaction dont les requêtes ont été commitées)

```sql
SELECT id, nom, espece_id, mere_id
FROM Animal
WHERE espece_id = 5;
```

Retourner au texte.

V.2. Verrous

Complément indispensable des transactions, les verrous permettent de **sécuriser les requêtes** en bloquant ponctuellement et partiellement l'accès aux données.

Il s'agit d'un gros chapitre, avec beaucoup d'informations. Il y a par conséquent un maximum d'exemples pour vous aider à comprendre le comportement des verrous selon les situations.

Au sommaire de ce chapitre :

— Qu'est-ce qu'un verrou ?
— Quel est le comportement par défaut de MySQL par rapport aux verrous ?
— Quand et comment poser un verrou de table ?
— Quand et comment poser un verrou de ligne ?
— Comment modifier le comportement par défaut de MySQL ?

V.2.1. Principe

Lorsqu'une session MySQL pose un verrou sur un élément de la base de données, cela veut dire qu'il **restreint, voire interdit, l'accès à cet élément aux autres sessions MySQL** qui voudraient y accéder.

V.2.1.1. Verrous de table et verrous de ligne

Il est possible de poser un **verrou sur une table entière**, ou **seulement sur une ou plusieurs lignes d'une table**. Étant donné qu'un verrou empêche l'accès d'autres sessions, il est en général plus intéressant de poser un verrou sur la plus petite partie de la base possible.

Par exemple, si l'on travaille avec les chiens de la table *Animal*.

— On peut poser un verrou sur toute la table *Animal*. Dans ce cas, les autres sessions n'auront pas accès à cette table, tant que le verrou sera posé. Qu'elles veuillent en utiliser les chiens, les chats, ou autre, tout leur sera refusé.
— On peut aussi poser un verrou uniquement sur les lignes de la table qui contiennent des chiens. De cette manière, les autres sessions pourront accéder aux chats, aux perroquets, etc. Elles pourront toujours travailler, tant qu'elles n'utilisent pas les chiens.

Cette notion d'accès simultané aux données par plusieurs sessions différentes s'appelle la **concurrence**. Plus la concurrence est possible, donc plus le nombre de sessions pouvant accéder aux données simultanément est grand, mieux c'est. En effet, prenons l'exemple d'un site web. En général, on préfère permettre à plusieurs utilisateurs de surfer en même temps, sans devoir attendre entre chaque action de pouvoir accéder aux informations chacun à son tour. Or, chaque utilisateur crée une session chaque fois qu'il se connecte à la base de données (pour lire les

informations ou les modifier). Préférez donc (autant que possible) les verrous de ligne aux verrous de table !

V.2.1.2. Avertissements

Les informations données dans ce chapitre **concernent exclusivement MySQL**, et en particulier les tables utilisant **les moteurs MyISAM ou InnoDB** (selon le type de verrou utilisé). En effet, les verrous sont implémentés différemment selon les SGDB, et même selon le moteur de table en ce qui concerne MySQL. Si le principe général reste toujours le même, certains comportements et certaines options peuvent différer d'une implémentation à l'autre. N'hésitez pas à vous renseigner plus avant.

Par ailleurs, je vous présente ici les principes généraux et les principales options, mais il faut savoir qu'il y a énormément à dire sur les verrous, et que j'ai donc dû faire un sérieux tri des informations avant de rédiger ce chapitre. À nouveau, en cas de doute, ou si vous avez besoin d'informations précises, je vous conseille vraiment de consulter la documentation officielle (si possible en anglais, car elle est infiniment plus complète qu'en français).

Enfin, dernier avertissement : de nombreux changements dans l'implémentation des verrous sont advenus lors du développement des dernières versions de MySQL. Aussi, la différence entre les verrous dans la version 5.0 et la version 5.5 est assez importante. Tout ce que je présente dans ce chapitre concerne la version 5.5. Vérifiez bien votre version, et si vous consultez la documentation officielle, prenez bien celle qui concerne votre propre version.

V.2.1.3. Modification de notre base de données

Nous allons ajouter deux tables à notre base de données, afin d'illustrer au mieux l'intérêt et l'utilisation des verrous : une table *Client*, qui contiendra les coordonnées des clients de notre élevage, et une table *Adoption*, qui contiendra les renseignements concernant les adoptions faites par nos clients. Dorénavant, certains animaux présents dans notre table *Animal* ne seront plus disponibles, car ils auront été adoptés. Nous les garderons cependant dans notre base de données. Avant toute adoption, il nous faudra donc vérifier la disponibilité de l'animal.

Voici les requêtes à effectuer pour faire ces changements.

Contenu masqué n°48

La table *Adoption* ne contient pas de colonne *id* auto-incrémentée. Par contre, on a bien défini une clé primaire, mais une clé primaire **composite** (sur plusieurs colonnes). En effet, une adoption est définie par un client adoptant un animal. Il n'est pas nécessaire d'ajouter une colonne supplémentaire pour définir individuellement chaque ligne ; le couple (*client_id, animal_id*) fait très bien l'affaire (il est composé de deux `SMALLINT`, donc les recherches sur cette clé seront rapides). Notez que nous définissons également un index `UNIQUE` sur la colonne *animal_id*. Par conséquent, on aurait même pu définir directement *animal_id* comme étant la clé primaire. Je trouvais cependant plus logique d'inclure le client dans la définition d'une adoption. C'est un choix plutôt arbitraire, qui a surtout comme avantage de vous montrer un exemple de clé composite.

V.2.2. Syntaxe et utilisation : verrous de table

Les verrous de table sont **les seuls supportés par MyISAM**. Ils sont d'ailleurs principalement utilisés pour pallier en partie l'absence de transactions dans MyISAM. Les tables InnoDB peuvent également utiliser ce type de verrou.

Pour verrouiller une table, il faut utiliser la commande `LOCK TABLES` :

```
1 LOCK TABLES nom_table [AS alias_table] [READ | WRITE] [, ...];
```

— En utilisant `READ`, un verrou de lecture sera posé ; c'est-à-dire que les autres sessions pourront toujours lire les données des tables verrouillées, mais ne pourront plus les modifier.
— En utilisant `WRITE`, un verrou d'écriture sera posé. Les autres sessions ne pourront plus ni lire ni modifier les données des tables verrouillées.

Pour déverrouiller les tables, on utilise `UNLOCK TABLES`. Cela déverrouille toutes les tables verrouillées. Il n'est pas possible de préciser les tables à déverrouiller. Tous les verrous de table d'une session sont relâchés en même temps.

V.2.2.0.1. Session ayant obtenu le verrou

Lorsqu'une session acquiert un ou plusieurs verrous de table, cela a plusieurs conséquences pour cette session :

— elle ne peut plus accéder **qu'aux tables sur lesquelles elle a posé un verrou** ;
— elle ne peut accéder à ces tables qu'**en utilisant les noms qu'elle a donnés** lors du verrouillage (soit le nom de la table, soit le/les alias donné(s)) ;
— s'il s'agit d'un **verrou de lecture** (`READ`), elle peut **uniquement lire les données**, pas les modifier.

Exemples : on pose deux verrous, l'un `READ`, l'autre `WRITE`, l'un en donnant un alias au nom de la table, l'autre sans.

```
1 LOCK TABLES Espece READ,              -- On pose un verrou de
      lecture sur Espece
2             Adoption AS adopt WRITE;  -- et un verrou d'écriture
                  sur Adoption avec l'alias adopt
```

Voyons maintenant le résultat de ces différentes requêtes.

1. Sélection dans *Especе*, sans alias.

```
1 SELECT id, nom_courant FROM Espece;
```

id	nom_courant
1	Chien
2	Chat
3	Tortue d'Hermann
4	Perroquet amazone
5	Rat brun

Pas de problème, on a bien un verrou sur *Espece*, sans alias.

2. Sélection dans *Espece*, avec alias.

```
SELECT id, nom_courant
FROM Espece AS table_espece;
```

```
ERROR 1100 (HY000): Table 'table_espece' was not locked with LOCK TABLES
```

Par contre, si l'on essaye d'utiliser un alias, cela ne fonctionne pas. Le verrou est posé sur *Espece*, pas sur *Espece* `AS` *table_espece*.

3. Modification dans *Espece*, sans alias.

```
UPDATE Espece
SET description = 'Petit piaf bruyant'
WHERE id = 4;
```

```
ERROR 1099 (HY000): Table 'Espece' was locked with a READ lock and can't be upd
```

Avec ou sans alias, impossible de modifier la table *Espece*, puisque le verrou que l'on possède dessus est un verrou de lecture.

4. Sélection dans *Adoption*, sans alias.

```
SELECT client_id, animal_id
FROM Adoption;
```

```
ERROR 1100 (HY000): Table 'Adoption' was not locked with LOCK TABLES
```

Cette fois, c'est le contraire, sans alias, ça ne passe pas.

5. Sélection dans *Adoption*, avec alias.

```
SELECT client_id, animal_id
FROM Adoption AS adopt
WHERE client_id = 4;
```

client_id	animal_id
4	26
4	41

6. Modification dans *Adoption*, sans alias.

```
UPDATE Adoption
SET paye = 0
WHERE client_id = 10 AND animal_id = 49;
```

```
ERROR 1100 (HY000): Table 'Adoption' was not locked with LOCK TABLES
```

Idem pour la modification, l'alias est indispensable.

7. Modification dans *Adoption*, avec alias.

```
UPDATE Adoption AS adopt
SET paye = 0
WHERE client_id = 10 AND animal_id = 49;
```

```
Query OK, 1 row affected (0.03 sec)
```

Il faut donc penser à acquérir tous les verrous nécessaires aux requêtes à exécuter. De plus, il faut les obtenir **en une seule requête** `LOCK TABLES`. En effet, `LOCK TABLES` commence par enlever tous les verrous de table de la session avant d'en acquérir de nouveaux. Il est bien entendu possible de poser plusieurs verrous sur la même table en une seule requête afin de verrouiller son nom ainsi qu'un ou plusieurs alias.

Exemples : on pose un verrou de lecture sur *Adoption*, puis avec une seconde requête, on pose deux verrous de lecture sur la table *Espece*, l'un avec alias, l'autre sans.

```
UNLOCK TABLES; -- On relâche d'abord les deux verrous précédents

LOCK TABLES Adoption READ;
LOCK TABLES Espece READ, Espece AS table_espece READ;
```

Une fois ces deux requêtes effectuées, nous aurons donc bien deux verrous de lecture sur la table *Espece* : un avec son nom, l'autre avec un alias. Par contre, le verrou sur *Adoption* n'existera plus puisqu'il aura été relâché par l'exécution de la seconde requête `LOCK TABLES`.

1. Sélection dans *Espece*, sans alias.

```
SELECT id, nom_courant FROM Espece;
```

id	nom_courant
1	Chien
2	Chat
3	Tortue d'Hermann
4	Perroquet amazone
5	Rat brun

2. Sélection dans *Espece*, avec alias.

```
SELECT id, nom_courant FROM Espece AS table_espece;
```

id	nom_courant
1	Chien
2	Chat
3	Tortue d'Hermann
4	Perroquet amazone
5	Rat brun

Avec ou sans alias, on peut sélectionner les données de la table *Espece*, puisque l'on a un verrou sur *Espece* et sur *Espece* `AS` *table_espece*.

3. Sélection dans *Espece*, avec mauvais alias.

```
SELECT id, nom_courant FROM Espece AS table_esp;
```

```
ERROR 1100 (HY000): Table 'table_esp' was not locked with LOCK TABLES
```

Bien entendu, cela ne fonctionne que pour l'alias que l'on a donné lors du verrouillage.

4. Sélection dans *Adoption*, sans alias.

```
SELECT * FROM Adoption;
```

```
ERROR 1100 (HY000): Table 'Adoption' was not locked with LOCK TABLES
```

Le verrou sur *Adoption* a été relâché lorsque l'on a posé les verrous sur *Espece*. On ne peut donc pas lire les données d'*Adoption* (avec ou sans alias).

V.2.2.0.2. Conséquences pour les autres sessions

Si une session a obtenu un **verrou de lecture** sur une table, les autres sessions :

- peuvent lire les données de la table ;
- peuvent également acquérir un verrou de lecture sur cette table ;
- ne peuvent pas modifier les données, ni acquérir un verrou d'écriture sur cette table.

Si par contre une session a obtenu un **verrou d'écriture**, les autres sessions ne peuvent absolument pas accéder à cette table tant que ce verrou existe.

Exemples : ouvrez un deuxième client MySQL et connectez-vous à votre base de données, afin d'avoir deux sessions ouvertes.

1. Sélection sur des tables verrouillées à partir d'une autre session.

Session 1 :

```
LOCK TABLES Client READ,        -- Verrou de lecture sur Client
            Adoption WRITE;     -- Verrou d'écriture sur Adoption
```

Session 2 :

```sql
SELECT id, nom, prenom, ville, email
FROM Client
WHERE ville = 'Houtsiplou';
```

id	nom	prenom	ville	email
1	Dupont	Jean	Houtsiplou	jean.dupont@email.com
12	Broussaille	Virginie	Houtsiplou	vibroussaille@email.com

La sélection sur *Client* se fait sans problème.

Session 2 :

```sql
SELECT *
FROM Adoption
WHERE client_id = 4;
```

Par contre, la sélection sur *Adoption* ne passe pas. La session se bloque, jusqu'à ce que la session 1 déverrouille les tables avec `UNLOCK TABLES`.

2. Modification sur des tables verrouillées à partir d'une autre session.

Reverrouillez les tables avec la session 1 :

```sql
LOCK TABLES Client READ,        -- Verrou de lecture sur Client
            Adoption WRITE;     -- Verrou d'écriture sur Adoption
```

Session 2 :

```sql
UPDATE Client
SET pays = 'Suisse'
WHERE id = 5;
```

La modification sur *Client*, contrairement à la sélection, est bloquée jusqu'au déverrouillage. Déverrouillez puis verrouillez à nouveau avec la session 1.

Session 2 :

```sql
UPDATE Adoption
SET paye = 1
WHERE client_id = 3;
```

Bien entendu, la modification sur la table *Adoption* attend également que les verrous soient relâchés par la session 1.

En ce qui concerne la pose de verrous de table par les autres sessions, faites vos propres tests, mais simplement : si une session peut lire les données d'une table, elle peut également poser un verrou de lecture. Si une session peut modifier les données d'une table, elle peut également poser un verrou d'écriture.

V.2.2.0.3. Interaction avec les transactions

Si l'on utilise des tables MyISAM, il n'y a évidemment aucune précaution particulière à prendre par rapport aux transactions lorsqu'on utilise des verrous de table (les tables MyISAM étant non-transactionnelles). Par contre, si on utilise des tables InnoDB, il convient d'être prudent. En effet :

— `START TRANSACTION` ôte les verrous de table ;
— les commandes `LOCK TABLES` et `UNLOCK TABLES` provoquent une validation implicite si elles sont exécutées à l'intérieur d'une transaction.

Pour utiliser à la fois les transactions et les verrous de table, il faut renoncer à démarrer explicitement les transactions, et donc utiliser le mode non-autocommit. Lorsque l'on est dans ce mode, il est facile de contourner la validation implicite provoquée par `LOCK TABLES` et `UNLOCK TABLES` : il suffit d'appeler `LOCK TABLES` avant toute modification de données, et de commiter/annuler les modifications avant d'exécuter `UNLOCK TABLES`.

Exemple :

```
1  SET autocommit = 0;
2  LOCK TABLES Adoption WRITE; -- La validation implicite ne commite
       rien puisque aucun changement n'a été fait
3
4  UPDATE Adoption SET date_adoption = NOW() WHERE client_id = 9 AND
       animal_id = 54;
5  SELECT client_id, animal_id, date_adoption FROM Adoption WHERE
       client_id = 9;
6
7  ROLLBACK;
8  UNLOCK TABLES; -- On a annulé les changements juste avant donc la
       validation implicite n'a aucune conséquence
9  SELECT client_id, animal_id, date_adoption FROM Adoption WHERE
       client_id = 9;
10 SET autocommit = 1;
```

V.2.3. Syntaxe et utilisation : verrous de ligne

Ces verrous ne peuvent pas être posés sur une table utilisant le moteur MyISAM ! Tout ce qui est dit ici concerne les tables **InnoDB** uniquement.

Comme les verrous de table, les verrous de ligne peuvent être de deux types :

— **Les verrous partagés** : permettent aux autres sessions de lire les données, mais pas de les modifier (équivalents aux verrous de table de lecture) ;
— **Les verrous exclusifs** : ne permettent ni la lecture ni la modification des données (équivalents aux verrous d'écriture).

V.2.3.1. Requêtes de modification, insertion et suppression

— Les requêtes de **modification et suppression** des données **posent automatiquement un verrou exclusif sur les lignes concernées**, à savoir les lignes sélectionnées par la clause `WHERE`, ou toutes les lignes s'il n'y a pas de clause `WHERE` (ou s'il n'y a pas d'index, sur les colonnes utilisées comme nous verrons plus loin).
— Les requêtes d'insertion quant à elles posent un **verrou exclusif sur la ligne insérée**.

V.2.3.2. Requêtes de sélection

Les requêtes de sélection, par défaut, ne posent pas de verrous. Il faut donc en poser explicitement au besoin.

V.2.3.2.1. Verrou partagé

Pour poser un verrou partagé, on utilise `LOCK IN SHARE MODE` à la fin de la requête `SELECT`.

```
1 SELECT * FROM Animal WHERE espece_id = 5 LOCK IN SHARE MODE;
```

Cette requête pose donc un verrou partagé sur les lignes de la table *Animal* pour lesquelles *espece_id* vaut 5.

Ce verrou signifie en fait, pour les autres sessions : "Je suis en train de lire ces données. Vous pouvez venir les lire aussi, mais pas les modifier tant que je n'ai pas terminé.".

V.2.3.2.2. Verrou exclusif

Pour poser un verrou exclusif, on utilise `FOR UPDATE` à la fin de la requête `SELECT`.

```sql
SELECT * FROM Animal WHERE espece_id = 5 FOR UPDATE;
```

Ce verrou signifie aux autres sessions : "Je suis en train de lire ces données dans le but probable de faire une modification. Ne les lisez pas avant que j'aie fini (et bien sûr, ne les modifiez pas).".

V.2.3.3. Transactions et fin d'un verrou de ligne

Les verrous de ligne ne sont donc pas posés par des commandes spécifiques, mais par des requêtes de sélection, insertion ou modification. Ces verrous existent donc uniquement tant que la requête qui les a posés interagit avec les données.

Par conséquent, ce type de verrou s'utilise en conjonction avec les transactions. En effet, hors transaction, dès qu'une requête est lancée, elle est effectuée et les éventuelles modifications des données sont immédiatement validées. Par contre, dans le cas d'une requête faite dans une transaction, les changements ne sont pas validés tant que la transaction n'a pas été commitée. Donc, à partir du moment où une requête a été exécutée dans une transaction, et jusqu'à la fin de la transaction (`COMMIT` ou `ROLLBACK`), la requête a **potentiellement** un effet sur les données. C'est à ce moment-là (quand une requête a été exécutée mais pas validée ou annulée) qu'il est intéressant de verrouiller les données qui vont **potentiellement être modifiées** (ou supprimées) par la transaction.

Un verrou de ligne est donc lié à la transaction dans laquelle il est posé. Dès que l'on fait un `COMMIT` ou un `ROLLBACK` de la transaction, le verrou est levé.

V.2.3.4. Exemples

V.2.3.4.1. Verrou posé par une requête de modification

Session 1 :

```sql
START TRANSACTION;

UPDATE Client SET pays = 'Suisse'
WHERE id = 8;           -- un verrou exclusif sera posé sur la
    ligne avec id = 8
```

Session 2 :

```sql
START TRANSACTION;

SELECT * FROM Client
WHERE id = 8;       -- pas de verrou

```

```
6 SELECT * FROM Client
7 WHERE id = 8
8 LOCK IN SHARE MODE; -- on essaye de poser un verrou partagé
```

La première session a donc posé un verrou exclusif automatiquement en faisant un `UPDATE`.

La seconde session fait d'abord une simple sélection, sans poser de verrou. Pas de problème, la requête passe.

? Ah ? La requête passe ? Et c'est normal ? Et le verrou exclusif alors ?

Oui, c'est normal et c'est important de comprendre pourquoi. En fait, lorsqu'une session **démarre une transaction**, **elle prend en quelque sorte une photo des tables dans leur état actuel** (les modifications non commitées n'étant pas visibles). La transaction va alors travailler sur la base de cette photo, tant qu'on ne lui demande pas d'aller vérifier que les données n'ont pas changé. Donc le `SELECT` ne voit pas les changements, et ne se heurte pas au verrou, puisque celui-ci est posé sur les lignes de la table, et non pas sur la photo de cette table que détient la session.

? Et comment fait-on pour demander à la session d'actualiser sa photo ?

On lui demande de poser un verrou ! Lorsqu'une session **pose un verrou** sur une table, elle est obligée de travailler vraiment avec la table, et pas sur sa photo. Elle va donc aller chercher les dernières infos disponibles, et actualiser sa photo par la même occasion. On le voit bien avec la seconde requête, qui tente de poser un verrou partagé (qui vise donc uniquement la lecture). Elle va d'abord chercher les lignes les plus à jour et tombe sur le verrou posé par la première session ; elle se retrouve alors bloquée jusqu'à ce que la première session ôte le verrou exclusif.

Session 1 :

```
1 COMMIT;
```

En committant les changements de la session 1, le verrou exclusif posé par la requête de modification est relâché. La session 2 est donc libre de poser à son tour un verrou partagé.

On peut également essayer la même manœuvre, avec cette fois-ci un `UPDATE` plutôt qu'un `SELECT ... LOCK IN SHARE MODE` (donc une requête qui va tenter de poser un verrou exclusif plutôt qu'un verrou partagé).

Session 1 :

```
1 START TRANSACTION;
2
3 UPDATE Adoption SET paye = 0
```

```
4 WHERE client_id = 11;
```

Session 2 :

```
1 START TRANSACTION;
2
3 UPDATE Adoption SET paye = 1
4 WHERE animal_id = 32; -- l'animal 32 a été adopté par le client 11
```

Comme prévu, la seconde session est bloquée, jusqu'à ce que la première session termine sa transaction. Validez la transaction de la première session, puis de la seconde. Le comportement sera le même si la deuxième session fait un `DELETE` sur les lignes verrouillées, ou un `SELECT ... FOR UPDATE`.

V.2.3.4.2. Verrou posé par une requête d'insertion

Session 1 :

```
1 START TRANSACTION;
2
3 INSERT INTO Adoption (client_id, animal_id, date_reservation, prix)
4 VALUES (12, 75, NOW(), 10.00);
```

Session 2 :

```
1 SELECT * FROM Adoption
2 WHERE client_id > 13
3 LOCK IN SHARE MODE;
4
5 SELECT * FROM Adoption
6 WHERE client_id < 13
7 LOCK IN SHARE MODE;
```

La première session insère une adoption pour le client 12 et pose un verrou exclusif sur cette ligne. La seconde session fait deux requêtes `SELECT` en posant un verrou partagé : l'une qui sélectionne les adoptions des clients avec un id supérieur à 13 ; l'autre qui sélectionne les adoptions des clients avec un id inférieur à 13. Seule la seconde requête `SELECT` se heurte au verrou posé par la première session, puisqu'elle tente de récupérer notamment les adoptions du client 12, dont une est verrouillée.

Dès que la session 1 commite l'insertion, la sélection se fait dans la session 2.

Session 1 :

```
COMMIT;
```

V.2.3.4.3. Verrou posé par une requête de sélection

Voyons d'abord le comportement d'un verrou partagé, posé par `SELECT ... LOCK IN SHARE MODE`.

Session 1 :

```
START TRANSACTION;

SELECT * FROM Client
WHERE id < 5
LOCK IN SHARE MODE;
```

Session 2 :

```
START TRANSACTION;

SELECT * FROM Client
WHERE id BETWEEN 3 AND 8;

SELECT * FROM Client
WHERE id BETWEEN 3 AND 8
LOCK IN SHARE MODE;

SELECT * FROM Client
WHERE id BETWEEN 3 AND 8
FOR UPDATE;
```

La première session pose un verrou partagé sur les clients 1, 2, 3 et 4. La seconde session fait trois requêtes de sélection. Toutes les trois concernent les clients 3 à 8 (dont les deux premiers sont verrouillés).

— Requête 1 : ne pose aucun verrou (travaille sur une "photo" de la table et pas sur les vraies données) donc s'effectue sans souci.
— Requête 2 : pose un verrou partagé, ce qui est faisable sur une ligne verrouillée par un verrou partagé. Elle s'effectue également.
— Requête 3 : tente de poser un verrou exclusif, ce qui lui est refusé.

Bien entendu, des requêtes `UPDATE` ou `DELETE` (posant des verrous exclusifs) faites par la deuxième session se verraient, elles aussi, bloquées.

Terminez les transactions des deux sessions (par un rollback ou un commit).

Quant aux requêtes `SELECT ... FOR UPDATE` posant un verrou exclusif, elles provoqueront exactement les mêmes effets qu'une requête `UPDATE` ou `DELETE` (après tout, un verrou exclusif, c'est un verrou exclusif).

Session 1 :

```
START TRANSACTION;

SELECT * FROM Client
WHERE id < 5
FOR UPDATE;
```

Session 2 :

```
START TRANSACTION;

SELECT * FROM Client
WHERE id BETWEEN 3 AND 8;

SELECT * FROM Client
WHERE id BETWEEN 3 AND 8
LOCK IN SHARE MODE;
```

Cette fois-ci, même la requête `SELECT ... LOCK IN SHARE MODE` de la seconde session est bloquée (comme le serait une requête `SELECT ... FOR UPDATE`, ou une requête `UPDATE`, ou une requête `DELETE`).

V.2.3.5. En résumé

— On pose un verrou partagé lorsqu'on fait une requête dans le but de lire des données.
— On pose un verrou exclusif lorsqu'on fait une requête dans le but (immédiat ou non) de modifier des données.
— Un verrou partagé sur les lignes x va permettre aux autres sessions d'obtenir également un verrou partagé sur les lignes x, mais pas d'obtenir un verrou exclusif.
— Un verrou exclusif sur les lignes x va empêcher les autres sessions d'obtenir un verrou sur les lignes x, qu'il soit partagé ou exclusif.

En fait, ils portent plutôt bien leurs noms ces verrous !

V.2.3.6. Rôle des index

Tentons une nouvelle expérience.

Session 1 :

```sql
START TRANSACTION;
UPDATE Animal
SET commentaires = CONCAT_WS(' ', 'Animal fondateur.',
    commentaires) -- On ajoute une phrase de commentaire
WHERE date_naissance < '2007-01-01';
   -- à tous les animaux nés avant 2007
```

Session 2 :

```sql
START TRANSACTION;
UPDATE Animal
SET commentaires = 'Aveugle'                    -- On modifie les
    commentaires
WHERE date_naissance = '2008-03-10 13:40:00'; -- De l'animal né le
    10 mars 2008 à 13h40
```

Dans la session 1, on fait un `UPDATE` sur les animaux nés avant 2007. On s'attend donc à pouvoir utiliser les animaux nés après dans une autre session, puisque InnoDB pose des verrous sur les lignes et pas sur toute la table. Pourtant, la session 2 semble bloquée lorsque l'on fait un `UPDATE` sur un animal né en 2008. Faites un rollback sur la session 1 ; ceci débloque la session 2. Annulez également la requête de cette session.

? Ce comportement est donc en contradiction avec ce qu'on obtenait précédemment. Quelle est la différence ?

Le sous-titre vous a évidemment soufflé la réponse : la différence se trouve au niveau des index. Voyons donc ça ! Voici une commande qui va vous afficher les index présents sur la table *Animal* :

```sql
SHOW INDEX FROM Animal;
```

Table	Non_unique	Key_name	Column_name	Null
Animal	0	PRIMARY	id	
Animal	0	ind_uni_nom_espece_id	nom	YES
Animal	0	ind_uni_nom_espece_id	espece_id	
Animal	1	fk_race_id	race_id	YES
Animal	1	fk_espece_id	espece_id	
Animal	1	fk_mere_id	mere_id	YES

Animal	1	fk_pere_id	pere_id	YES

Une partie des colonnes du résultat montré ici a été retirée pour des raisons de clarté.

Nous avons donc des index sur les colonnes suivantes : *id*, *nom*, *mere_id*, *pere_id*, *espece_id* et *race_id*. Mais aucun index sur la colonne *date_naissance*.

Il semblerait donc que lorsque l'on pose un verrou, avec dans la clause `WHERE` de la requête une colonne indexée (*espece_id*), le verrou est bien posé uniquement sur les lignes pour lesquelles *espece_id* vaut la valeur recherchée. Par contre, si dans la clause `WHERE` on utilise une colonne non-indexée (*date_naissance*), MySQL n'est pas capable de déterminer quelles lignes doivent être bloquées, donc on se retrouve avec toutes les lignes bloquées.

Pourquoi faut-il un index pour pouvoir poser un verrou efficacement ?

C'est très simple ! Vous savez que lorsqu'une colonne est indexée (que ce soit un index simple, unique, ou une clé primaire ou étrangère), MySQL stocke les valeurs de cette colonne **en les triant**. Du coup, lors d'une recherche sur l'index, pas besoin de parcourir toutes les lignes, il peut utiliser des algorithmes de recherche performants et trouver facilement les lignes concernées. S'il n'y a pas d'index par contre, toutes les lignes doivent être parcourues chaque fois que la recherche est faite, et il n'y a donc pas moyen de verrouiller simplement une partie de l'index (donc une partie des lignes). Dans ce cas, MySQL verrouille toutes les lignes.

Cela fait une bonne raison de plus de mettre des index sur les colonnes qui servent fréquemment dans vos clauses `WHERE` !

Encore une petite expérience pour illustrer le rôle des index dans le verrouillage de lignes :

Session 1 :

```
START TRANSACTION;
UPDATE Animal  -- Modification de tous les rats
SET commentaires = CONCAT_WS(' ', 'Très intelligent.',
    commentaires)
WHERE espece_id = 5;
```

Session 2 :

```
START TRANSACTION;
UPDATE Animal
SET commentaires = 'Aveugle'
WHERE id = 34; -- Modification de l'animal 34 (un chat)
UPDATE Animal
```

```
6 SET commentaires = 'Aveugle'
7 WHERE id = 72; -- Modification de l'animal 72 (un rat)
```

La session 1 se sert de l'index sur *espece_id* pour verrouiller les lignes contenant des rats bruns. Pendant ce temps, la session 2 veut modifier deux animaux : un chat et un rat, en se basant sur leur *id*. La modification du chat se fait sans problème, par contre, la modification du rat est bloquée, tant que la transaction de la session 1 est ouverte. Faites un rollback des deux transactions.

On peut conclure de cette expérience que, bien que MySQL utilise les index pour verrouiller les lignes, il n'est pas nécessaire d'utiliser le même index pour avoir des accès concurrents.

V.2.3.7. Lignes fantômes et index de clé suivante

? Qu'est-ce qu'une ligne fantôme ?

Dans une session, démarrons une transaction et sélectionnons toutes les adoptions faites par les clients dont l'*id* dépasse 13, avec un verrou exclusif.

Session 1 :

```
1 START TRANSACTION;
2
3 SELECT * FROM Adoption WHERE client_id > 13 FOR UPDATE; -- ne pas
    oublier le FOR UPDATE pour poser le verrou
```

La requête va poser un verrou exclusif sur toutes les lignes dont *client_id* vaut 14 ou plus.

client_id	animal_id	date_reservation	date_adoption	prix	pay
14	58	2012-02-25	2012-02-25	700.00	1
15	30	2008-08-17	2008-08-17	735.00	1

Imaginons maintenant qu'une seconde session démarre une transaction à ce moment-là, insère et commite une ligne dans *Adoption* pour le client 15. Si, par la suite, la première session refait la même requête de sélection avec verrou exclusif, elle va faire apparaître une troisième ligne de résultat : l'adoption nouvellement insérée (étant donné que pour poser le verrou, la session va aller chercher les données les plus à jour, prenant en compte le commit de la seconde session).

Cette ligne nouvellement apparue malgré les verrous est une "ligne fantôme".

Pour pallier ce problème, qui est contraire au principe d'isolation, **les verrous posés par des requêtes de lecture, de modification et de suppression sont des verrous dits "de clé**

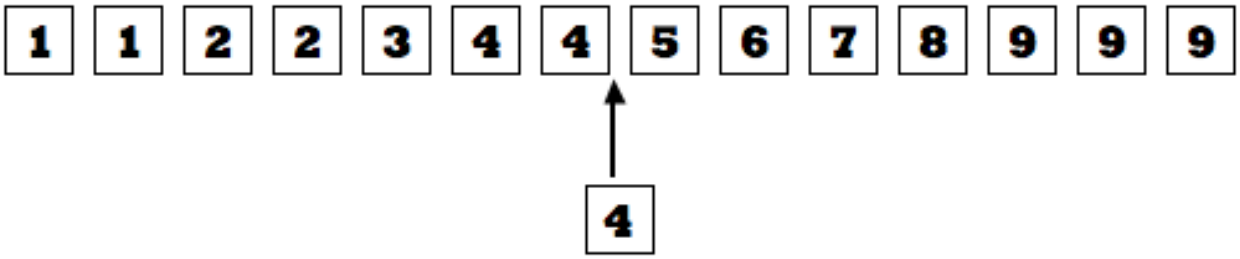
suivante" ; ils empêchent l'insertion d'une ligne dans les espaces entre les lignes verrouillées, ainsi que dans l'espace juste après les lignes verrouillées.

? L'espace entre ? L'espace juste après ?

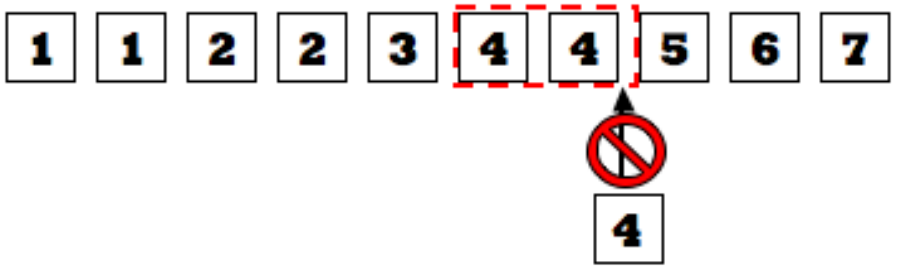
Nous avons vu que les verrous se basent sur les index pour verrouiller uniquement les lignes nécessaires. Voici un petit schéma qui vous expliquera ce qu'est cet "index de clé suivante".

On peut représenter l'index sur *client_id* de la table *Adoption* de la manière suivante (je ne mets que les *client_id* < 10) :

Si l'on insère une adoption avec 4 pour *client_id*, l'index va être réorganisé de la manière suivante :



Mais, si l'on pose un verrou de clé suivante sur l'index, sur les lignes dont *client_id* vaut 4, on va alors verrouiller **les lignes, les espaces entre les lignes et les espaces juste après**. Ceci va bloquer l'insertion de la nouvelle ligne :



Démonstration

On a toujours un verrou exclusif (grâce à notre `SELECT ... FOR UPDATE`) sur les *client_id* supérieurs à 14 dans la session 1 (sinon, reposez-le).

Session 2 :

```
START TRANSACTION;

INSERT INTO Adoption (client_id, animal_id, date_reservation, prix)
VALUES (15, 61, NOW(), 735.00);
```

L'insertion est bloquée ! Pas de risque de voir apparaître une ligne fantôme. Annulez les deux transactions.

V.2.3.7.1. Exception

Si la clause `WHERE` concerne un index `UNIQUE` (cela inclut bien sûr les clés primaires) et recherche une seule valeur (exemple : `WHERE id = 4`), alors seule la ligne concernée (si elle existe) est verrouillée, et pas l'espace juste après dans l'index. Forcément, s'il s'agit d'un index `UNIQUE`, l'insertion d'une nouvelle valeur ne changera rien : `WHERE id = 4` ne renverra jamais qu'une seule ligne.

V.2.3.8. Pourquoi poser un verrou exclusif avec une requête SELECT ?

Après tout, une requête `SELECT` ne fait jamais que lire des données. Que personne ne puisse les modifier pendant qu'on est en train de les lire, c'est tout à fait compréhensible. Mais pourquoi carrément interdire aux autres de les lire aussi ?

Tout simplement parce que certaines données sont lues dans le but prévisible et avoué de les modifier immédiatement après.

L'exemple typique est la vérification de stock dans un magasin (ou dans un élevage d'animaux). Un client arrive et veut adopter un chat, on vérifie donc les chats disponibles pour l'adoption, en posant un verrou partagé :

Session 1 :

```
START TRANSACTION;

SELECT Animal.id, Animal.nom, Animal.date_naissance, Race.nom as
    race, COALESCE(Race.prix, Espece.prix) as prix
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
LEFT JOIN Race ON Animal.race_id = Race.id              -- Jointure
    externe, on ne veut pas que les chats de race
WHERE Especе.nom_courant = 'Chat'                      --
    Uniquement les chats...
AND Animal.id NOT IN (SELECT animal_id FROM Adoption)  -- ... qui
    n'ont pas encore été adoptés
LOCK IN SHARE MODE;
```

id	nom	date_naissance	race	prix
2	Roucky	2010-03-24 02:23:00	NULL	150.00
8	Bagherra	2008-09-11 15:38:00	Maine coon	735.00
29	Fiero	2009-05-14 06:30:00	Singapura	985.00
31	Filou	2008-02-20 15:45:00	Bleu russe	835.00
34	Capou	2008-04-20 03:22:00	Maine coon	735.00

35	Raccou	2006-05-19 16:56:00	Bleu russe	835.00
36	Boucan	2009-05-14 06:42:00	Singapura	985.00
37	Callune	2006-05-19 16:06:00	Nebelung	985.00
38	Boule	2009-05-14 06:45:00	Singapura	985.00
43	Cracotte	2007-03-12 11:54:00	Maine coon	735.00
44	Cawette	2006-05-19 16:16:00	Nebelung	985.00
61	Yoda	2010-11-09 00:00:00	Maine coon	735.00

Je rappelle que la fonction `COALESCE()` prend un nombre illimité de paramètres, et renvoie le premier paramètre non `NULL` qu'elle rencontre. Donc ici, s'il s'agit d'un chat de race, *Race.prix* ne sera pas `NULL` et sera donc renvoyé. Par contre, s'il n'y a pas de race, *Race.prix* sera `NULL`, mais pas *Especе.prix*, qui sera alors sélectionné.

Si, pendant que le premier client fait son choix, un second client arrive, qui veut adopter un chat Maine Coon, il va également chercher la liste des chats disponibles. Et vu qu'on travaille pour l'instant en verrous partagés, il va pouvoir l'obtenir.

Session 2 :

```
START TRANSACTION;

SELECT Animal.id, Animal.nom, Animal.date_naissance, Race.nom as
    race, COALESCE(Race.prix, Espece.prix) as prix
FROM Animal
INNER JOIN Especе ON Animal.especе_id = Especе.id
INNER JOIN Race ON Animal.race_id = Race.id            -- Jointure
    interne cette fois
WHERE Race.nom = 'Maine Coon'                          --
    Uniquement les Maine Coon...
AND Animal.id NOT IN (SELECT animal_id FROM Adoption)  -- ... qui
    n'ont pas encore été adoptés
LOCK IN SHARE MODE;
```

id	nom	date_naissance	race	prix
8	Bagherra	2008-09-11 15:38:00	Maine coon	735.00
34	Capou	2008-04-20 03:22:00	Maine coon	735.00
43	Cracotte	2007-03-12 11:54:00	Maine coon	735.00

61	Yoda	2010-11-09 00:00:00	Maine coon	735.00

C'est alors que le premier client, M. Dupont, décide de craquer pour Bagherra.

```
INSERT INTO Adoption (client_id, animal_id, date_reservation, prix,
    paye)
SELECT id, 8, NOW(), 735.00, 1
FROM Client
WHERE email = 'jean.dupont@email.com';

COMMIT;
```

Et M. Durant jette également son dévolu sur Bagherra (qui est décidément très très mignon) !

```
INSERT INTO Client (nom, prenom, email)
VALUES ('Durant', 'Philippe', 'phidu@email.com');

INSERT INTO Adoption (client_id, animal_id, date_reservation, prix,
    paye)
VALUES (LAST_INSERT_ID(), 8, NOW(), 735.00, 0);
```

L'insertion dans *Client* fonctionne mais l'insertion dans *Adoption* pose problème :

```
ERROR 1062 (23000): Duplicate entry '8' for key 'ind_uni_animal_id'
```

Et pour cause : Bagherra vient d'être adopté, à l'instant. Furieux, M. Durant s'en va, et l'élevage a perdu un client. Il ne reste plus qu'à annuler sa transaction.

C'est pour éviter ce genre de situation qu'il vaut parfois mieux mettre un verrou exclusif sur une sélection. Si l'on sait que cette sélection sert à déterminer quels changements vont être faits, ce n'est pas la peine de laisser quelqu'un d'autre lire des informations qui cesseront d'être justes incessamment sous peu.

V.2.4. Niveaux d'isolation

Nous avons vu que par défaut :

— lorsque l'on démarre une transaction, la session prend une photo des tables, et travaille uniquement sur cette photo (donc sur des données potentiellement périmées) tant qu'elle ne pose pas un verrou ;
— les requêtes `SELECT` ne posent pas de verrous si l'on ne le demande pas explicitement ;

— les requêtes `SELECT ... LOCK IN SHARE MODE`, `SELECT ... FOR UPDATE`, `DELETE` et `UPDATE` posent un verrou de clé suivante (sauf dans le cas d'une recherche sur index unique, avec une valeur unique).

Ce comportement est **défini par le niveau d'isolation** des transactions.

V.2.4.1. Syntaxe

Pour définir le niveau d'isolation des transactions, on utilise la requête suivante :

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL { READ
    UNCOMMITTED | READ COMMITTED  | REPEATABLE READ  | SERIALIZABLE
    }
```

— Le mot-clé `GLOBAL` définit le niveau d'isolation pour toutes les sessions MySQL qui seront créées dans le futur. Les sessions existantes ne sont pas affectées.
— `SESSION` définit le niveau d'isolation pour la session courante.
— Si l'on ne précise ni `GLOBAL`, ni `SESSION`, le niveau d'isolation défini ne concernera que la prochaine transaction que l'on ouvrira dans la session courante.

V.2.4.2. Les différents niveaux

V.2.4.2.1. REPEATABLE READ

Il s'agit du **niveau par défaut**, celui avec lequel vous travaillez depuis le début. *Repeatable read* signifie "lecture répétable", c'est-à-dire que si l'on fait plusieurs requêtes de sélection (**non-verrouillantes**) de suite, elles donneront toujours le même résultat, quels que soient les changements effectués par d'autres sessions. Si l'on pense à bien utiliser les verrous là où c'est nécessaire, c'est un niveau d'isolation tout à fait suffisant.

V.2.4.2.2. READ COMMITTED

Avec ce niveau d'isolation, chaque requête `SELECT` (non-verrouillante) va reprendre une "photo" à jour de la base de données, même si plusieurs `SELECT` se font dans la même transaction. Ainsi, un `SELECT` verra toujours les derniers changements commités, même s'ils ont été faits dans une autre session, après le début de la transaction.

V.2.4.2.3. READ UNCOMMITTED

Le niveau `READ UNCOMMITTED` fonctionne comme `READ COMMITTED`, si ce n'est qu'il autorise la "lecture sale". C'est-à-dire qu'une session sera capable de lire des changements encore non commités par d'autres sessions.

Exemple

Session 1 :

```
START TRANSACTION;

UPDATE Race
SET prix = 0
WHERE id = 7;
```

Session 2 :

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;

SELECT id, nom, espece_id, prix FROM Race;
```

id	nom	espece_id	prix
1	Berger allemand	1	485.00
2	Berger blanc suisse	1	935.00
3	Singapura	2	985.00
4	Bleu russe	2	835.00
5	Maine coon	2	735.00
7	Sphynx	2	0.00
8	Nebelung	2	985.00
9	Rottweiller	1	600.00

La modification faite par la session 1 n'a pas été commitée. Elle ne sera donc potentiellement jamais validée, auquel cas, elle n'affectera jamais les données. Pourtant, la session 2 voit ce changement de données non-commitées. "Lecture sale" n'a pas une connotation négative par hasard, bien entendu ! Aussi, évitez de travailler avec ce niveau d'isolation. Annulez la modification de données réalisée par la session 1 et terminez la transaction de la seconde session.

V.2.4.2.4. SERIALIZABLE

Ce niveau d'isolation se comporte comme `REPEATABLE READ`, sauf que lorsque le mode autocommit est désactivé, tous les `SELECT` simples sont implicitement convertis en `SELECT ... LOCK IN SHARE MODE`.

V.2.4.3. En résumé

- Les verrous permettent de **restreindre, voire interdire l'accès, à une partie des données**.
- Les **verrous de table** peuvent s'utiliser sur des tables transactionnelles et non-transactionnelles, contrairement aux **verrous de ligne** qui ne sont disponibles que pour des tables transactionnelles.
- Les **verrous de lecture (tables) et partagés (lignes)** permettent aux autres sessions de lire les données verrouillées, mais pas de les modifier. Les **verrous d'écriture (tables) et exclusif (lignes)** par contre, ne permettent aux autres sessions ni de lire, ni de modifier les données verrouillées.
- Les verrous de ligne s'utilisent **avec les transactions**, et dépendent des **index**.
- Les requêtes de **suppression, modification et insertion posent automatiquement un verrou** de ligne exclusif de clé suivante sur les lignes concernées par la requête. Les requêtes de sélection par contre, ne posent pas de verrou par défaut, il faut en poser un explicitement.
- Le comportement par défaut des verrous de ligne est défini par le **niveau d'isolation des transactions**, qui est modifiable.

Contenu masqué

Contenu masqué n°48

```sql
-- Table Client
CREATE TABLE Client (
        id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
        nom VARCHAR(100) NOT NULL,
        prenom VARCHAR(60) NOT NULL,
        adresse VARCHAR(200),
        code_postal VARCHAR(6),
        ville VARCHAR(60),
        pays VARCHAR(60),
        email VARBINARY(100),
        PRIMARY KEY (id),
        UNIQUE INDEX ind_uni_email (email)
) ENGINE = InnoDB;

```

```sql
-- Table Adoption
CREATE TABLE Adoption (
        client_id SMALLINT UNSIGNED NOT NULL,
        animal_id SMALLINT UNSIGNED NOT NULL,
        date_reservation DATE NOT NULL,
        date_adoption DATE,
        prix DECIMAL(7,2) UNSIGNED NOT NULL,
        paye TINYINT(1) NOT NULL DEFAULT 0,
        PRIMARY KEY (client_id, animal_id),
        CONSTRAINT fk_client_id FOREIGN KEY (client_id) REFERENCES
            Client(id),
        CONSTRAINT fk_adoption_animal_id FOREIGN KEY (animal_id)
            REFERENCES Animal(id),
        UNIQUE INDEX ind_uni_animal_id (animal_id)
) ENGINE = InnoDB;

-- Insertion de quelques clients
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Jean', 'Dupont', 'Rue du Centre, 5', '45810',
    'Houtsiplou', 'France', 'jean.dupont@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Marie', 'Boudur', 'Place de la Gare, 2',
    '35840', 'Troudumonde', 'France', 'marie.boudur@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Fleur', 'Trachon', 'Rue haute, 54b', '3250',
    'Belville', 'Belgique', 'fleurtrachon@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Julien', 'Van Piperseel',  NULL, NULL, NULL,
    NULL, 'jeanvp@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Johan', 'Nouvel',  NULL, NULL, NULL, NULL,
    'johanetpirlouit@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Frank', 'Germain',  NULL, NULL, NULL, NULL,
    'francoisgermain@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Maximilien', 'Antoine', 'Rue Moineau, 123',
    '4580', 'Trocoul', 'Belgique', 'max.antoine@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Hector', 'Di Paolo', NULL, NULL, NULL, NULL,
    'hectordipao@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Anaelle', 'Corduro',  NULL, NULL, NULL, NULL,
    'ana.corduro@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Eline', 'Faluche', 'Avenue circulaire, 7',
    '45870', 'Garduche', 'France', 'elinefaluche@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays,
    email) VALUES ('Carine', 'Penni', 'Boulevard Haussman, 85',
    '1514', 'Plasse', 'Suisse', 'cpenni@email.com');
```

```sql
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays, email) VALUES ('Virginie', 'Broussaille', 'Rue du Fleuve, 18', '45810', 'Houtsiplou', 'France', 'vibrousaille@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays, email) VALUES ('Hannah', 'Durant', 'Rue des Pendus, 66', '1514', 'Plasse', 'Suisse', 'hhdurant@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays, email) VALUES ('Elodie', 'Delfour', 'Rue de Flore, 1', '3250', 'Belville', 'Belgique', 'e.delfour@email.com');
INSERT INTO Client (prenom, nom, adresse, code_postal, ville, pays, email) VALUES ('Joel', 'Kestau', NULL, NULL, NULL, NULL, 'joel.kestau@email.com');

-- Insertion de quelques adoptions
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (1, 39, '2008-08-17', '2008-08-17', 735.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (1, 40, '2008-08-17', '2008-08-17', 735.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (2, 18, '2008-06-04', '2008-06-04', 485.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (3, 27, '2009-11-17', '2009-11-17', 200.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (4, 26, '2007-02-21', '2007-02-21', 485.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (4, 41, '2007-02-21', '2007-02-21', 835.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (5, 21, '2009-03-08', '2009-03-08', 200.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (6, 16, '2010-01-27', '2010-01-27', 200.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (7, 5, '2011-04-05', '2011-04-05', 150.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (8, 42, '2008-08-16', '2008-08-16', 735.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (9, 55, '2011-02-13', '2011-02-13', 140.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation, date_adoption, prix, paye) VALUES (9, 54, '2011-02-13', '2011-02-13', 140.00, 1);
```

```sql
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (10, 49, '2010-08-17',
    '2010-08-17', 140.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (11, 62, '2011-03-01',
    '2011-03-01', 630.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (12, 69, '2007-09-20',
    '2007-09-20', 10.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (13, 57, '2012-01-10',
    '2012-01-10', 700.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (14, 58, '2012-02-25',
    '2012-02-25', 700.00,  1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (15, 30, '2008-08-17',
    '2008-08-17', 735.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (11, 32, '2008-08-17',
    '2010-03-09', 140.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (9, 33, '2007-02-11',
    '2007-02-11', 835.00, 1);
INSERT INTO Adoption (client_id, animal_id, date_reservation,
    date_adoption, prix, paye) VALUES (2, 3, '2011-03-12',
    '2011-03-12', 835.00, 1);
```

Retourner au texte.