
----- Transactions (COMMIT & ROLLBACK) -----

-- 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).

-- Pensez à bien définir un AUTOCOMMIT = 0 avant d'effectuer des TRANSACTION

SET AUTOCOMMIT = 0;

-- Tant que l'on est en 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).

-- On récupère l'id du pays Malte (base de donnée music-shop)

SET @id = (

SELECT id

FROM country

WHERE country.name = 'Malta'

);

-- On déclare une transaction : c'est à dire que les requêtes exécutées dans ce code attendent un ROLLBACK (annule les requêtes) ou un COMMIT (confirme l'exécution des requêtes)

START TRANSACTION;

DELETE FROM country

WHERE country.id = @id

-- Ici le delete ne passera pas car il y a un ROLLBACK

ROLLBACK;

```
START TRANSACTION;

UPDATE country

SET country.nationality = 'Maltese'

WHERE country.id = @id;

-- L'update sera bien pris en compte car il y a un COMMIT

COMMIT;
```

-- Exemple de gestion des transactions avec SAVEPOINT --

-- Jalon de transaction :

-- Il s'agit d'un point de repère qui permet d'annuler toutes les requêtes exécutées depuis ce jalon seulement et non toutes les requêtes de la transaction.

```
START TRANSACTION;

// traitement 1

SAVEPOINT jalon1 ; (jalon1 = alias)

// Traitement 2

ROLLBACK TO SAVEPOINT jalon1 ;

// Traitement 3

COMMIT ;
```

Dans cette exemple, traitement 2 ne sera pas effectué

Cependant certaines requêtes ne peuvent pas être « annulées » via la commande ROLLBACK, il s'agit des commandes qui influent sur une structure de données, comme par exemple :

CREATE DATABASE, DROP DATABASE, CREATE TABLE, ALTER TABLE, RENAME TABLE, DROP TABLE, CREATE INDEX ou encore DROP INDEX.

```
SET AUTOCOMMIT = 0;
```

```
-- On récupère l'id du pays Slovenia
```

```
SET @id = (
```

```
    SELECT id
```

```
    FROM country
```

```
    WHERE country.name = 'Slovenia'
```

```
);
```

```
-- On déclare une transaction : c'est à dire que les requêtes exécutées dans ce code attendent un ROLLBACK (annule les requêtes) ou un COMMIT (confirme l'exécution des requêtes)
```

```
START TRANSACTION;
```

```
UPDATE country
```

```
SET country.nationality = 'Slovene'
```

```
WHERE country.id = @id;
```

```
-- On effectue un point de sauvegarde après l'UPDATE nommé update_slovenia
```

```
SAVEPOINT update_slovenia;
```

```
DELETE FROM country
```

```
WHERE country.id = @id;
```

```
-- On fait un ROLLBACK à notre point de sauvegarde update_slovenia
```

```
ROLLBACK TO SAVEPOINT update_slovenia;
```

```
-- Il faut bien COMMIT nos changements
```

```
COMMIT;
```

```
-----  
-- /!\ pour les ROLLBACK --  
-----
```

```
-- Un rollback ne peut pas permettre d'annuler un changement de la structure de données, c'est à dire un
```

-- changement de l'état de la table ou de la base de données (CREATE / DROP DATABASE, CREATE / ALTER / DROP TABLE etc)

-- Les TRIGGER --

Les triggers ne s'appliquent que sur INSERT - UPDATE et DELETE ('evenement_trigger', voir schéma)

'moment_trigger' correspond à BEFORE ou AFTER (voir schéma)

création du trigger avec son nom

quand et comment le trigger est déclenché

```
CREATE TRIGGER nom_trigger moment_trigger evenement_trigger
```

```
ON nom_table FOR EACH ROW
```

pour chaque ligne, selon comment le trigger est déclenché, via insert/update/delete

```
corps_trigger;
```

sur quelle table le trigger est attaché

contenu du trigger, procédures stockées ou instruction ou bloc d'instructions

-- /!\ On ne peut avoir qu'une seule combinaison de 'moment_trigger' et 'evenement_trigger' par table (total de 6 par table)

-- Mots clés "offert" par les TRIGGER : OLD & NEW

-- OLD : ancienne valeur, avant que le le trigger ne s'applique (selon le 'moment_trigger', voir schéma)

-- NEW : nouvelle valeur, après que le le trigger ne s'applique (selon le 'moment_trigger', voir schéma)

-- Supprime un trigger

```
DROP TRIGGER nom_trigger;
```

-- **BONUS** --

-- ALTER table permet de modifier une table

ALTER TABLE listing

-- Ici on ajoute une column de nom updated_at de type DATETIME nullable

ADD COLUMN updated_at DATETIME DEFAULT NULL

-- Facultatif, par défaut la colonne sera ajoutée à la fin de votre table

-- ici on lui précise juste qu'on la veut après publish_at

AFTER publish_at;

Exemple de trigger

Delimiter permet de définir un "nouveau" delimiter à SQL permettant ainsi de lui dire que l'on exécute du code jusqu'à ce qu'il rencontre ce nouveau delimiter : permet de palier à une erreur lors de la création de TRIGGER

DELIMITER |

-- création d'un trigger de nom listing_before_update

-- qui s'exécutera BEFORE un UPDATE sur la table listing

CREATE TRIGGER listing_before_update BEFORE UPDATE

ON listing FOR EACH ROW

BEGIN

-- le trigger mettra à un jour le champ updated_at automatiquement à la date du jour

SET NEW.updated_at = NOW();

END |

DELIMITER ;

"Gestion des erreurs"

-- Cela passe par une création de table

CREATE TABLE errors (

id INT(6) PRIMARY KEY AUTO_INCREMENT,

message VARCHAR(255) NOT NULL

)

-- Exemple de cas de figure avec gestion d'erreur

DELIMITER |

-- Le drop trigger permet de supprimer un trigger, vu que l'on ne peut en avoir qu'un par combinaison de BEFORE INSERT autant le faire...

DROP TRIGGER IF EXISTS listing_before_insert |

CREATE TRIGGER listing_before_insert BEFORE INSERT

ON listing FOR EACH ROW

BEGIN

IF year(NEW.publish_at) < NEW.creation_year

THEN

INSERT INTO errors VALUES (1, 'La date de sortie de la voiture ne peut pas être plus élevée que la date de publication de la vente !');

END IF;

END |

DELIMITER ;

Si l'on essaie d'insérer une nouvelle vente (table listing), avec une creation_year de la voiture supérieure à l'année de publication de la vente, alors on va ajouter une ligne dans la table erreur. Cependant, au préalable on aura "configuré" notre table 'errors', en ayant déjà ajouté la même ligne.

L'erreur MySQL qui sortira sera un duplicata de la PRIMARY KEY '1', autrement dit l'insertion de notre message d'erreur de notre IF

Définir un squelette de requête "dynamique"

```
PREPARE select_model_from_brand
```

```
FROM 'SELECT * FROM model JOIN brand ON model.brand_id = brand.id WHERE brand.name = ?';
```

```
-- Il faut les exécuter dans la même invite de requêtes de PhpMyAdmin
```

```
EXECUTE select_model_from_brand USING 'Ferrari';
```

```
EXECUTE select_model_from_brand USING 'Opel';
```

```
EXECUTE select_model_from_brand USING 'Toyota';
```

```
EXECUTE select_model_from_brand USING 'Volkswagen';
```

```
-----
```

```
-- PROCEDURE --
```

```
-----
```

```
-- Exemple de procédure
```

```
DELIMITER //
```

```
DROP PROCEDURE IF EXISTS getCa //
```

```
CREATE PROCEDURE getCa(idBrand INT)
```

```
BEGIN
```

```
    SELECT SUM(listing.price), brand.name
```

```
    FROM listing
```

```
    JOIN model ON model.id = listing.model_id
```

```
    JOIN brand ON brand.id = model.brand_id
```

```
    WHERE brand.id = idBrand;
```

```
END //
```

```
DELIMITER ;
```

-- Exécution de la procédure avec le mot-clé CALL puis le nom de la procédure et son paramètre, si nécessaire

CALL getCa (1);

-- Procédure permettant de réduire de la valeur de percent le prix de toute les ventes de la marque idBrand

DELIMITER //

DROP PROCEDURE IF EXISTS setRistourne //

CREATE PROCEDURE setRistourne(idBrand INT, percent INT)

BEGIN

 UPDATE listing

 SET listing.price = listing.price * ((100 - percent)/100)

 WHERE listing.model_id IN (

 SELECT model.id

 FROM model

 JOIN brand ON model.brand_id = brand.id

 WHERE brand.id = idBrand

);

END //

DELIMITER ;

-- Les paramètres sont séparés par des virgules

CALL setRistourne (1, 10);

-- NB : les procédures n'ont pas de valeur de retour ! On ne peut effectuer que des INSERT, SELECT, UPDATE et DELETE

-- FONCTIONS --

-- Sur certains SGBD il faut mettre un @ devant le paramètre de la fonction

-- Une fonction a une valeur de retour

DELIMITER //

CREATE FUNCTION getTotalListingByBrand (idBrand INT)

RETURNS INT

BEGIN

 RETURN (

 SELECT COUNT(listing.id)

 FROM listing

 JOIN model ON model.id = listing.model_id

 JOIN brand ON model.brand_id = brand.id

 WHERE brand.id = idBrand

 GROUP BY brand.id

);

END //

DELIMITER ;