Bases de données

6. Programmation MySQL

Vincent Zucca

vincent.zucca@univ-perp.fr

Université de Perpignan Via Domitia

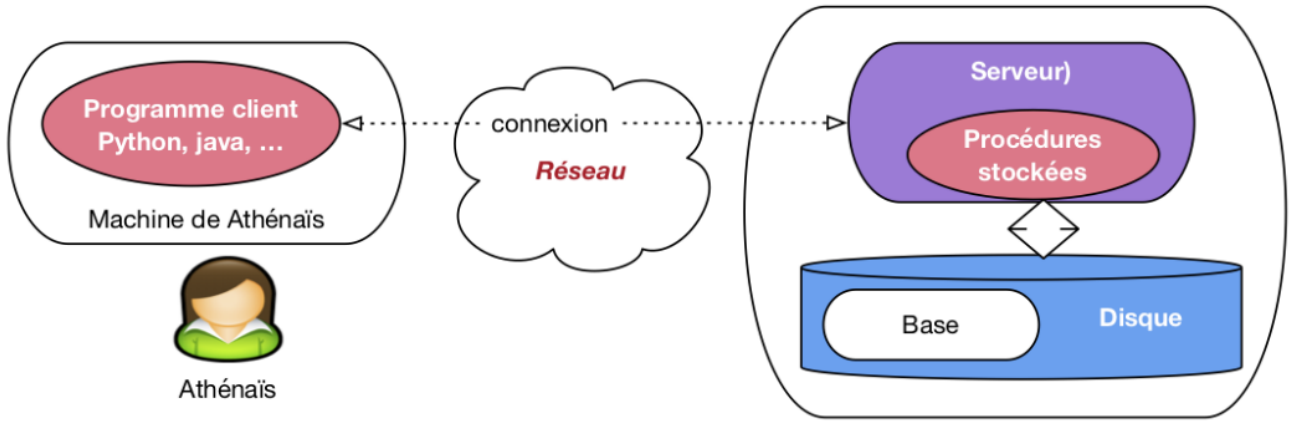
S3 Licence 2022-2023

Motivations

- Le langage SQL n'est pas un langage de programmation au sens usuel
  - Pas de variables, fonctions, boucles ou structures de contrôle
- C'est un choix délibéré de la norme afin d'orienter SQL vers les opérations de recherches dans des bases volumineuses.
  - priorité donnée à la simplicité et à l'efficacité
  - la terminaison d'une requête SQL est toujours garantie (pas de boucle infinie)
- Pour palier à ce manque, les SGBD relationnels proposent des interfaces afin d'associer SQL à d'autres langages (C, Java, ...).
  - SQL est utilisé comme un outil pour accéder à la base de données
- Dans certains cas, le recours à un langage "externe" n'est pas forcément adapté pour l'application que l'on souhaite développer.
  - les SGBD proposent des primitives de programmation procédurale "interne"

Motivations

- Un programme externe se connecte au serveur du SGBD avant de faire des requêtes via le réseau
  - les échanges réseaux affectent les performances du programme
- Possibilité de stocker de manière durable des séries d'instructions `SQL` directement dans le SGBD.
  - un seul échange réseau afin d'exécuter une série d'instructions
  - quand la suite d'instructions est nommée on parle de procédure stockée



- Les procédures stockées peuvent s'avérer utiles même en l'absence de problème de performance :
  - exemple : le programme met à jour des données dont la cohérence doit être absoluement maintenue

Motivations

- Avantages des procédures stockées :
  - elles permettent de sécuriser une base de données
    - ★ possibilité de restreindre les droits des utilisateurs de façon à ce qu'ils puissent uniquement exécuter des procédures.
    - ★ pas de `DELETE` ou d'`INSERT` "sauvages"
  - elles permettent de s'assurer qu'un traitement est toujours exécuté de la même manière
    - ★ traitement indépendant du client/application qui le lance
- Inconvénients des procédures stockées :
  - chaque SGBD propose son propre langage procédural
    - ★ problèmes de compatibilité
  - elles ajoutent de la charge sur le serveur de données
    - ★ moins d'espace disponible pour les données

Variables utilisateurs

- Une variable utilisateur est une variable, définie par l'utilisateur...
  - Les variables utilisateur MySQL doivent être précédées du signe @

- La manière la plus classique de créer ou modifier une variable utilisateur est d'utiliser la commande SET

```
SET @age = 24;  -- Ne pas oublier le @
-- On peut creer plusieurs variables en meme temps
SET @salut = 'Hello World !', @poids = 7.8;
```

  - si la variable utilisateur existe déjà, sa valeur sera modifiée, sinon, elle sera créée.

- On peut afficher une variable utilisateur avec la commande SELECT

```
SELECT @age, @poids, @salut;
```

- On peut assigner une valeur à une variable utilisateur directement dans une requête, en utilisant l'opérateur d'assignation :=

```
SELECT @age := 32, @poids := 48.15, @perroquet := 4;
```

  - ne pas oublier les : sinon il s'agit de l'opérateur de comparaison de valeurs

Variables utilisateurs

- Une fois votre variable utilisateur créée, vous pouvez l'afficher avec SELECT.

- Vous pouvez également l'utiliser dans des requêtes ou faire des calculs avec.

```
SELECT id, sexe, nom, commentaires, espece_id
FROM Animal
WHERE espece_id = @perroquet; -- On selectionne les perroquets

-- On cree une variable contenant le taux de conversion EUR/USD
SET @conversionDollar = 1.31564;

-- On selectionne le prix des races, en euros et en dollars
SELECT prix AS prix_en_euros,
-- En arrondissant a deux decimales
ROUND(prix * @conversionDollar, 2) AS prix_en_dollars,
nom FROM Race;
```

  - si vous utilisez une variable utilisateur qui n'a pas été définie, vous n'obtiendrez aucune erreur.
  - la variable aura comme valeur NULL.

Variables utilisateurs

- Il n'est pas possible d'utiliser les variables utilisateur pour stocker un nom de table ou de colonne
- Les variables utilisateur stockent des données !

```
SET @colonnes = 'id, nom, description';

SELECT @colonnes FROM Race WHERE espece_id = 1;
```

@colonnes
id, nom, description
id, nom, description
id, nom, description

- Une variable utilisateur n'existe que pour la session dans laquelle elle a été définie
  - toutes les variables utilisateurs sont réinitialisées lors d'une déconnexion

Création et utilisation d'une procédure

- La syntaxe pour créer une procédure est la suivante :

```
CREATE PROCEDURE nom_procedure ([parametre1, parametre2, ...])
corps de la procedure;
```

  - après le nom de la procédure viennent des parenthèses **obligatoires**
  - à l'intérieur de ces parenthèses, on définit les éventuels paramètres de la procédure.

- Exemple (sans bloc d'instructions)

```
-- procedure sans parametre
CREATE PROCEDURE afficher_races_requete()
SELECT id, nom, espece_id, prix FROM Race;
```

- Exemple (avec bloc d'instructions)

```
-- procedure sans parametre
CREATE PROCEDURE afficher_races_bloc()
BEGIN
    SELECT id, nom, espece_id, prix FROM Race;
END;
```

  - pour délimiter un bloc d'instructions (qui peut contenir plus d'une instruction), on utilise les mots `BEGIN` et `END`.

Création et utilisation d'une procédure

```
-- procedure sans parametre
CREATE PROCEDURE afficher_races_bloc()
BEGIN
    SELECT id, nom, espece_id, prix FROM Race;
END;
```

- **Attention :** La requête ci-dessus va déclencher une erreur !
  - le ; est utilisé pour terminer une instruction SQL
  - SQL va vouloir s'arrêter lorsqu'il rencontrera le ; dans le bloc `SELECT * FROM Race`
  - Erreur car le bloc d'instructions n'est pas complet
- Il faut donc changer le délimiteur (par défaut ;)
  - on pourra utiliser ; dans une procédure stockée
- Pour changer le délimiteur on utilise la commande `DELIMITER`

  DELIMITER |
  - en pratique on change le délimiteur pour un caractère peu utilisé → |.
  - à la fin de la procédure | est toujours le délimiteur.

Création et utilisation d'une procédure

- Pour créer notre procédure on utilisera donc :

```
DELIMITER |
CREATE PROCEDURE afficher_races()
BEGIN
  SELECT id, nom, espece_id, prix
  FROM Race;
END|
```

- Pour appeler une procédure stockée, il faut utiliser le mot-clé `CALL`, suivi du nom de la procédure

```
CALL afficher_races()| -- le delimiteur est toujours |
```

- les instructions à l'intérieur du corps de la procédure sont toujours délimitées par ;
- Si on veut récupérer le ; après avoir défini la procédure on réutilise la commande `DELIMITER`

Les paramètres d'une procédure stockée

- Un paramètre peut être de trois sens différents : entrant (`IN`), sortant (`OUT`), ou les deux (`INOUT`)
  - `IN` : un paramètre dont la valeur est fournie à la procédure stockée
  - `OUT` : un paramètre dont la valeur va être établie au cours de la procédure et qui pourra ensuite être utilisé en dehors de cette procédure
  - `INOUT` : sera utilisé pendant la procédure, verra éventuellement sa valeur modifiée par celle-ci, et sera ensuite utilisable en dehors
- Lorsque l'on crée une procédure avec un ou plusieurs paramètres, chaque paramètre est défini par trois éléments
  - Son sens : entrant, sortant, ou les deux. Si aucun sens n'est donné, il s'agira d'un paramètre `IN` par défaut.
  - Son nom : indispensable pour le désigner à l'intérieur de la procédure.
  - Son type : `INT`, `VARCHAR(10)`, ...

Les paramètres d'une procédure stockée

- Procédure avec un seul paramètre entrant

```
DELIMITER |
CREATE PROCEDURE aff_race_selon_espece (IN p_espece_id INT)
BEGIN
  SELECT id, nom, espece_id, prix
  FROM Race
  WHERE espece_id = p_espece_id;
END |
DELIMITER ;
```

- Pour l'utiliser, il faut donc passer une valeur en paramètre de la procédure.
  - directement ou par l'intermédiaire d'une variable utilisateur.

```
CALL afficher_race_selon_espece(1);
SET @espece_id := 2;
CALL afficher_race_selon_espece(@espece_id);
```

Les paramètres d'une procédure stockée

- Procédure avec un paramètre entrant et un paramètre sortant

```
DELIMITER |
CREATE PROCEDURE compter_races_selon_espece
                 (p_espece_id INT, OUT p_nb_races INT)
BEGIN
    SELECT COUNT(*) INTO p_nb_races
    FROM Race
    WHERE espece_id = p_espece_id;
END |
DELIMITER ;
```

  - le résultat de la requête est stockée dans le paramètre `p_nb_races` à l'aide du mot-clé `INTO`
  - pour pouvoir utiliser `INTO` il faut que :
    - `SELECT` ne renvoie qu'une seule ligne
    - le nombre de valeurs sélectionnées et le nombre de variables à assigner soient égaux.

```
SELECT id, nom INTO @var1, @var2
FROM Animal
WHERE id = 7;
SELECT @var1, @var2;
```

  - `CALL compter_races_selon_espece (2, @nb_races_chats);`

Suppression d'une procédure

- Pour supprimer une procédure, on utilise `DROP` en précisant qu'il s'agit d'une procédure.

```
DROP PROCEDURE afficher_races;
```

- Les procédures stockées ne sont pas détruites à la fermeture de la session mais bien enregistrées comme un élément de la base de données, au même titre qu'une table par exemple.
  - les procédures sont associées à des bases de données
  - il faut donc spécifier la base de données considérée pour créer/supprimer une procédure
- Il n'est pas possible de modifier une procédure directement.
  - Pour modifier une procédure on la supprime puis on la recrée avec les modifications

Blocs d'instructions et variables locales

- Un bloc d'instructions est délimité par les mots-clés `BEGIN` et `END`, entre lesquels on met les instructions qui composent le bloc
  - de zéro à autant d'instructions que l'on veut, chacune séparée par un ;

```
BEGIN
 SELECT 'Bloc d''instructions principal';

 BEGIN
      SELECT 'Bloc 2, imbrique dans le bloc principal';

      BEGIN
           SELECT 'Bloc 3, imbrique dans le bloc 2';
      END;
 END;

 BEGIN
      SELECT 'Bloc 4, imbrique dans le bloc principal';
 END;

END;
```

Blocs d'instructions et variables locales

- Les variables locales sont des variables définies dans un bloc d'instructions.
  - dès que le `END` est atteint, toutes les variables locales du bloc sont détruites.

- La déclaration d'une variable locale se fait avec l'instruction `DECLARE`

```
DECLARE nom_variable type [DEFAULT valeur_defaut];
```

- Cette instruction doit se trouver au tout début du bloc d'instructions dans lequel la variable locale sera utilisée → directement après `BEGIN`.

```
BEGIN
-- Declarations de variables locales

-- Instructions (dont eventuels blocs d'instructions imbriques)
END;
```

- Si aucune valeur par défaut n'est précisée, la variable vaudra `NULL` tant que sa valeur n'est pas changée.
  - pour changer la valeur d'une variable locale, on peut utiliser `SET` ou `SELECT ... INTO`

Structure conditionelle

- Les structures conditionnelles permettent de déclencher une action ou une série d'instructions lorsqu'une condition préalable est remplie.
- MySQL propose deux structures conditionnelles : IF et CASE.
  - on ne présentera que IF dans ce cours
- Voici la syntaxe de la structure IF :

```
IF condition THEN instructions
[ELSEIF autre_condition THEN instructions
[ELSEIF ...]]
[ELSE instructions]
END IF;
```

  - les ELSEIF et le ELSE sont optionnels
  - ELSE ne doit PAS être suivi de THEN

Structure conditionelle

- La procédure suivante affiche 'J'ai ete adopte !' si c'est le cas à partir de l'id d'un animal

```
DELIMITER |
CREATE PROCEDURE est_adopte(IN p_animal_id INT)
BEGIN
-- On cree une variable locale
DECLARE v_nb INT DEFAULT 0;

-- On met le nombre de lignes correspondant a l'animal
-- dans Adoption dans notre variable locale
SELECT COUNT(*) INTO v_nb
FROM Adoption
WHERE animal_id = p_animal_id;

-- On teste si v_nb est superieur a 0
-- (donc si l'animal a ete adopte)
IF v_nb > 0 THEN
SELECT 'J''ai deja ete adopte !';
-- On n'oublie surtout pas le END IF et le ; final
END IF;
END |
DELIMITER ;
```

Boucles

- Une boucle est une structure qui permet de répéter plusieurs fois une série d'instructions.

- Il existe trois types de boucles en `MySQL` : `WHILE`, `LOOP` et `REPEAT`
  - La boucle `WHILE` permet de répéter une série d'instructions tant que la condition donnée reste vraie.

```
WHILE condition DO
instructions
END WHILE;
```

  - La boucle `REPEAT` foncitonne de manière opposée à `WHILE`, puisqu'elle exécute des instructions de la boucle jusqu'à ce que la condition donnée devienne vraie.

```
REPEAT
instructions
UNTIL condition
END REPEAT;
```

Boucles

- Procédure avec `WHILE` :

```
DELIMITER |
CREATE PROCEDURE compter_jusque_while(IN p_nombre INT)
BEGIN
  DECLARE v_i INT DEFAULT 1;
  WHILE v_i <= p_nombre DO
    SELECT v_i AS nombre;
    -- On incremente v_i, sinon la condition restera vraie
    SET v_i = v_i + 1;
  END WHILE;
END |
```

- Même procédure avec `REPEAT`

```
DELIMITER |
CREATE PROCEDURE compter_jusque_repeat(IN p_nombre INT)
BEGIN
  DECLARE v_i INT DEFAULT 1;
  REPEAT
    SELECT v_i AS nombre;
    -- On incremente v_i, sinon la condition restera vraie
    SET v_i = v_i + 1;
  UNTIL v_i > p_nombre END REPEAT;
END |
```

Boucles

- L'instruction `LEAVE` peut s'utiliser dans une boucle ou un bloc d'instructions
  - elle déclenche la sortie immédiate de la structure dont le label est donné

- Que fait la procédure suivante ?

```
DELIMITER |
CREATE PROCEDURE test_leave(IN p_nombre INT)
BEGIN
  DECLARE v_i INT DEFAULT 4;
  SELECT 'Avant la boucle WHILE';
  while1: WHILE v_i > 0 DO

    SET p_nombre = p_nombre + 1;
    IF p_nombre%10 = 0 THEN
      SELECT 'Stop !' AS 'Multiple de 10';
      LEAVE while1;
    END IF;

    SELECT p_nombre;
    SET v_i = v_i - 1;
  END WHILE while1;
  SELECT 'Apres la boucle WHILE';
END|
```

Boucles

- L'instruction `ITERATE` ne peut être utilisée que dans une boucle. Lorsqu'elle est exécutée, une nouvelle itération de la boucle commence.
  - Toutes les instructions suivant ITERATE dans la boucle sont ignorées.

```
DELIMITER |
CREATE PROCEDURE test_iterate()
BEGIN
  DECLARE v_i INT DEFAULT 0;

  boucle_while: WHILE v_i < 3 DO
    SET v_i = v_i + 1;
    SELECT v_i, 'Avant IF' AS message;

    IF v_i = 2 THEN
      ITERATE boucle_while;
    END IF;
    -- Ne sera pas execute pour v_i = 2
    SELECT v_i, 'Apres IF' AS message;
  END WHILE;
END |
DELIMITER ;
```

Boucles

- La boucle `LOOP` a la syntaxe suivante :

```
[label:] LOOP
  instructions
END LOOP [label]
```

- Il n'y a pas de condition !
  - `LOOP` doit intégrer dans ses instructions un élément qui va la faire s'arrêter : typiquement une instruction `LEAVE`

```
DELIMITER |
CREATE PROCEDURE compter_jusque_loop(IN p_nombre INT)
BEGIN
  DECLARE v_i INT DEFAULT 1;

  boucle_loop: LOOP
    SELECT v_i AS nombre;

    SET v_i = v_i + 1;

    IF v_i > p_nombre THEN
      LEAVE boucle_loop;
    END IF;
    END LOOP boucle_loop;
END |
```

Rôle d'un déclencheur

- Tout comme les procédures stockées, les déclencheurs servent à exécuter une ou plusieurs instructions.
- En revanche, il n'est pas possible d'appeler un déclencheur : un déclencheur doit être déclenché par un événement.
- Un déclencheur est attaché à une table, et peut être déclenché par :
  - une insertion dans la table (requête `INSERT`) ;
  - la suppression d'une partie des données de la table (requête `DELETE`) ;
  - la modification d'une partie des données de la table (requête `UPDATE`).
- Les instructions peuvent être exécutées soit juste avant l'exécution de l'événement déclencheur, soit juste après (`BEFORE` ou `AFTER`).

Rôle d'un déclencheur

- Un déclencheur exécute un traitement pour chaque ligne insérée, modifiée ou supprimée par l'événement déclencheur.
- Un déclencheur peut modifier et/ou insérer des données dans n'importe quelle table sauf les tables utilisées dans la requête qui l'a déclenché.
- Dans la table à laquelle il est attaché, un déclencheur peut lire et modifier uniquement la ligne insérée, modifiée ou supprimée qu'il est en train de traiter.

Rôle d'un déclencheur

- Voici quelques exemples d'applications des déclencheurs :
  - Contraintes et vérification de données
    - ★ se déclenchant avant `INSERT` et avant `UPDATE`
    - ★ on vérifie les valeurs d'une colonne lors de l'insertion ou de la modification
    - ★ on les corriger si elles ne font pas partie des valeurs acceptables
  - Intégrité des données
    - ★ parfois utilisés pour remplacer les options des clés étrangères `ON UPDATE RESTRICT|CASCADE|SET NULL` et `ON DELETE RESTRICT|CASCADE|SET NULL`
    - ★ possibilité d'archiver les données dans une table plutôt que de les supprimer simplement avec un `DELETE`

Création d'un déclencheur

- Pour créer un déclencheur, on utilise la commande suivante :

```
CREATE TRIGGER nom_trigger moment_trigger evenement_trigger
ON nom_table FOR EACH ROW
corps_trigger
```

  - `CREATE TRIGGER nom_trigger` : les déclencheurs ont un nom
  - `moment_trigger evenement_trigger` : servent à définir quand et comment le trigger est déclenché.
  - `ON nom_table` : c'est là qu'on définit à quelle table le trigger est attaché.

- Trois événements différents peuvent mener à l'exécution d'un déclencheur
  - L'insertion de lignes (INSERT) dans la table attachée au trigger.
  - La modification de lignes (UPDATE) de cette table.
  - La suppression de lignes (DELETE) de la table.

Création d'un déclencheur

- Lorsqu'un déclencheur est déclenché, ses instructions peuvent être exécutées à deux moments différents.
  - avant que l'événement déclencheur n'ait lieu (`BEFORE`)
  - juste après que l'événement déclencheur ait eu lieu (`AFTER`)
- Exemple pour créer un déclencheur sur la table `Film` après l'insertion

```
CREATE TRIGGER after_insert_film AFTER INSERT
ON Film FOR EACH ROW
corps_trigger;
```

- Il ne peut exister qu'un seul déclencheur par combinaison `moment_trigger/evenement_trigger` par table.
  - on a donc un maximum de six déclencheurs par table

Création d'un déclencheur

- Dans le corps du déclencheur, `MySQL` met à disposition deux mots-clés : `OLD` et `NEW`
  - `OLD` : représente les valeurs des colonnes de la ligne traitée avant qu'elle ne soit modifiée par l'événement déclencheur.
    - ★ ces valeurs peuvent être lues, mais pas modifiées.
  - `NEW` : représente les valeurs des colonnes de la ligne traitée après qu'elle a été modifiée par l'événement déclencheur.
    - ★ Ces valeurs peuvent être lues et modifiées.
- Il n'y a que dans le cas d'un déclencheur `UPDATE` que `OLD` et `NEW` coexistent.
  - lors d'une insertion, `OLD` n'existe pas
  - lors d'une suppression, c'est `NEW` qui n'existe pas
- Dans le corps du déclencheur on accède aux valeur de la colonne `colonne` avec `OLD.colonne` ou `NEW.colonne`

Suppression d'un déclencheur

- La commande `DROP` permet de supprimer un déclencheur.

```
DROP TRIGGER nom_trigger;
```

- Tout comme pour les procédures stockées, il n'est pas possible de modifier un déclencheur
  - il faut le supprimer puis le recréer différemment.

- Lorsque l'on supprime une table, on supprime tous les déclencheurs qui lui sont attachés.

- Comme pour les tables ou les procédures il est possible d'indiquer dans vos scripts `.sql` de supprimer un déclencheur s'il a déjà été créé.

```
DROP TRIGGERS IF EXISTS nom_trigger;
```

  - permer d'éviter les erreurs lorsque l'on recharge un script avec `source` qui a déjà été chargé.

Restrictions

- Une suppression ou modification de données déclenchée par une clé étrangère ne provoquera pas l'exécution du trigger correspondant.
  - une clé étrangère définit avec l'option `ON DELETE SET NULL` verra sa valeur modifiée à `NULL` lorsque la valeur à laquelle elle fait référence n'existera plus.
  - une donnée a été modifiée mais elle ne déclenchera pas un déclencheur éventuel `BEFORE UPDATE` ou `AFTER UPDATE`
  - si on définit un déclencheur sur une table contenant des clés étrangères avec ces options il vaut mieux supprimer ces dernières
- On ne peut modifier les données d'une table utilisée par la requête ayant déclenché le déclencheur à l'intérieur de celui-ci.