GLO-2005 Modèles et langages des bases de données pour ingénieurs

Programmation SQL

Survol

- Variables locales
- Fonctions et procédures
- Contrôle et exceptions
- Curseurs
- Livres
  - Ramakrishnan & Gehrke : Ø
  - Connolly & Begg : sections 8.1 et 8.2

Introduction

- Le langage SQL est un langage de requêtes
- Mais il existe des extensions pour en faire un langage de programmation procédural
  - Permet les mêmes fonctionnalités qu'un autre langage de programmation
  - Permet d'implémenter des fonctions de traitement de données dans le SGBDR directement
- Chaque SGBDR implémente sa propre version
  - Mêmes fonctionnalités mais syntaxe différente
  - Oracle/MySQL : PL/SQL (*procedural language SQL*)
  - Microsoft SQL Server : T-SQL (*transact-SQL*)
  - PostgreSQL : PL/pgSQL

Variables locales

- On peut définir des variables locales
  - Identifiées avec le symbole-clé `@`
  - Assignées avec:
    - `SET @variable := valeur;`
    - `SELECT attribut INTO @variable ...`
  - Peuvent prendre une valeur spécifique ou le résultat d'une requête
  - Peuvent être utilisées comme des valeurs dans nos requêtes
  - Sont supprimées automatiquement à la fin de la session

Variables locales

- Exemples d'assignation
  - `SET @temp1 := 123456789;`
  - ```
    SELECT @temp2 := E.nom
    FROM  Etudiants E
    WHERE E.idul='BOUVW';
    ```
  - ```
    SELECT E.nom INTO @temp2
    FROM  Etudiants E
    WHERE E.idul='BOUVW';
    ```

Variables locales

- Exemples d'utilisation
  - `SELECT @temp1;`
  - `SELECT @temp1, @temp2;`
  - ```
    UPDATE Etudiants E
    SET E.nom := @temp2
    WHERE E.nas = @temp1;
    ```
    ```

Variables locales

- Une variable peut uniquement avoir une valeur discrète
- Mais peut prendre plusieurs valeurs durant une requête

Variables locales

- Exemples
  - ```
    SELECT @temp2 := E.nom
    FROM  Etudiants E;
    ```
    - Va prendre chaque valeur de E.nom durant l'exécution de la requête
    - Aura la valeur de E.nom du dernier tuple de la relation à la fin de l'exécution
  - ```
    SELECT @temp2 := E.nom, E.nas
    FROM  Etudiants E
    WHERE E.idul = 'BOUVW';
    ```
    - Va prendre la valeur de E.nom du tuple
    - La valeur de E.nas est aussi retournée par la requête mais pas assignée à la variable

Variables locales

- Remarquez l'opérateur d'assignation de valeur est :=
  - L'opérateur = est déjà utilisé pour la comparaison égalité
- Exemples
  - ```
    SELECT @temp2 := E.nom
    FROM   Etudiants E
    WHERE E.idul='BOUVW';
    ```
    - Retourne la valeur E.nom et l'assigne à temp2
  - ```
    SELECT @temp2 = E.nom
    FROM   Etudiants E
    WHERE E.idul='BOUVW';
    ```
    - Retourne le résultat de la comparaison entre E.nom et temp2 (1, 0, ou null)

Variables locales

- On peut échanger `:=` et `=` dans le contexte où il n'y a pas d'ambigüité sur la signification
  - `SET @temp1 := 123456789;`
  - `SET @temp1 = 123456789;`

Variables locales

- On peut également créer des relations temporaires
  - Permettent d'emmagasiner des tuples temporairement
  - Définies de la même manière qu'une relation normale, en spécifiant qu'elle est temporaire
    - `CREATE [TEMPORARY] TABLE [IF NOT EXISTS]` *nom*
      `{ ( nom_attribut domaine [contrainte_attribut] [,…],`
      `[contraintes_relation [,…]] )`
      `| LIKE ancien_nom | [AS requête] };`
  - Utilisées comme n'importe quelle relation
  - Supprimées automatiquement par le SGBDR à la fin de la session de travail
- Exemple:
  - `CREATE TEMPORARY TABLE tempEtudiants LIKE Etudiants;`

Variables locales

- On peut remplir cette relation temporaire de tuples sélectionnés de la relation originale
  - Comme on peut faire avec des relations permanentes
- Exemple:
  - ```
    CREATE TEMPORARY TABLE tempEtudiants
    AS
    SELECT *
    FROM Etudiants
    WHERE moyenne > 4;
    ```
    ```

Limites

- Le marqueur de limite d'une commande est `;`
  - Tout le contenu écrit depuis le dernier `;` jusqu'au `;` actuel est envoyé par le client SQL au serveur SQL comme une commande
- Parfois on a besoin de définir une série de commandes
  - Combinaison de plusieurs commandes et plusieurs marqueurs
  - Mais chaque marqueur va envoyer la portion de commande en cours immédiatement!

Limites

- Définir un nouveau marqueur de limite
  - `DELIMITER` *marqueur*
    - Commande du client MySQL
  - Marqueur de notre choix
  - Un seul marqueur est actif à la fois
- Permet d'écrire un bloc de commandes avec le marqueur `;` après chacune
- On termine le bloc avec le nouveau marqueur
  - Tout le bloc de commandes contenu est envoyé d'un coup au serveur
  - Ne pas oublier de restaurer la limite au `;` après!

Limites

- ```
  DELIMITER //
      Commande 1;
      Commande 2;
      …
      Commande n; //
  DELIMITER ;
  ```
  ```

Limites

- SQL prend chaque commande comme indépendante
- On définit un bloc de commandes `BEGIN` … `END;`
  - Peut être identifié avec des étiquettes
- La première chose à faire est de déclarer les variables locales au bloc (s'il y en a) avec `DECLARE` *variable domaine* [`DEFAULT` *valeur*] ;

Limites

- [*étiquette*:] BEGIN
  [DECLARE *variable domaine* [DEFAULT *valeur*];]
    commandes;
  END [*étiquette*];

Fonctions et procédures

- On peut créer des routines en SQL
  - Permet d'automatiser la validation et le traitement routiniers de données
  - Permet de contenir du code fréquemment utilisé
    - Élimine la duplication d'efforts à le réécrire chaque fois, et les risques d'erreurs
  - Comparé à une application externe : exécution dans le SGBDR
    - Pas de transfert de données (surcharge, risque de vol des données)

Fonctions et procédures

- Trois types de routines
  - Fonction (*function*)
  - Procédure (*procedure*)
  - Gâchette (*trigger*)

	Fonction	Procédure	Gâchette
Appel	Avec une requête `SELECT`	Avec une requête `CALL`	Automatiquement lorsqu'une condition est satisfaite
Appel par une autre requête?	Oui	Non	Automatiquement
Paramètres	D'entrée	D'entrée et de sortie	Aucun
Nombre de valeurs retournées	1	0 à 1024	0
Type de retour	Commande `RETURN`	Paramètres `OUT` ou `INOUT`	

Fonctions et procédures

- `CREATE PROCEDURE` *nom* `(` `[`*paramètres*`]` `)`
  - On doit spécifier le nom des paramètres, le domaine de valeurs, et s'ils servent à passer ou retourner des données (ou les deux)
    - Paramètres :

      `{ IN | OUT | INOUT }` *nom domaine* `[, ... ]`
- `CALL` *nom* `(` `[`*paramètres*`]` `)` `;`
- `DROP PROCEDURE` *nom*`;`

Fonctions et procédures

- `CREATE FUNCTION` *nom* `(`[*paramètres*]`)` `RETURNS` *domaine* `DETERMINISTIC`
  - On doit spécifier le domaine de la valeur retournée
  - On doit spécifier le nom des paramètres et le domaine de valeurs
    - Paramètres : *nom domaine* `[, ... ]`
  - On doit spécifier si la fonction est déterministe ou non
    - Retourne toujours le même résultat pour les mêmes entrées
    - C'est le cas dans notre cours
- `SELECT` *nom* `(`[*paramètres*]`);`
- `DROP FUNCTION` *nom;*

Fonctions et procédures

- Le corps de la routine doit être encadré par des mots clés `BEGIN` … `END;`
- Comme il y a plusieurs commandes, il faut redéfinir le marqueur de limite
  - `DELIMITER //`

    `CREATE {FUNCTION|PROCEDURE}` *nom*

    `BEGIN`

    code

    `END//`

  `DELIMITER ;`

Fonctions et procédures

- Exemple: routine pour obtenir le nombre d'étudiants de moins d'un certain âge

- Procédure:

```
DELIMITER //
CREATE PROCEDURE
  EtudiantAge
  ( IN limite integer,
    OUT compte integer )
BEGIN

  SELECT COUNT(E.idul)
    INTO compte
    FROM Etudiants E
    WHERE E.age < limite;

END//
DELIMITER ;
```

- Utilisation:
  - `CALL EtudiantAge(25,@total);`
  - `SELECT @total;`

- Fonction:

```
DELIMITER //
CREATE FUNCTION
   EtudiantAge
   ( limite integer )
   RETURNS integer
   DETERMINISTIC
BEGIN
DECLARE compte integer;
  SELECT COUNT(E.idul)
     INTO compte
     FROM Etudiants E
     WHERE E.age < limite;
  RETURN compte;
END//
DELIMITER ;
```

- Utilisation:
  - `SELECT EtudiantAge(25);`

Fonctions et procédures

- Alternative: procédure à un seul paramètre

- Deux paramètres:

```
DELIMITER //
CREATE PROCEDURE
  EtudiantAge
  ( IN limite integer,
    OUT compte integer )
BEGIN
  SELECT COUNT(E.idul)
    INTO compte
    FROM Etudiants E
    WHERE E.age < limite;
END//
DELIMITER ;
```

- Utilisation:
  - `CALL EtudiantAge(25,@total);`
  - `SELECT @total;`

- Un paramètre:

```
DELIMITER //
CREATE PROCEDURE
  EtudiantAge
  ( INOUT valeur integer )

BEGIN
  SELECT COUNT(E.idul)
    INTO valeur
    FROM Etudiants E
    WHERE E.age < valeur;
END//
DELIMITER ;
```

- Utilisation:
  - `SET @param := 25;`
  - `CALL EtudiantAge(@param);`
  - `SELECT @param;`

Fonctions et procédures

- Exemple: routine pour masquer la moyenne des étudiants de moins d'un certain âge

- Procédure:

```
DELIMITER //
CREATE PROCEDURE
  MasqueAge
  ( IN limite integer )

BEGIN
  UPDATE Etudiants E
    SET E.moyenne = null
    WHERE
          E.age < limite;

END//
DELIMITER ;
```

- Utilisation:
  - `CALL MasqueAge(25);`

- Fonction:

```
DELIMITER //
CREATE FUNCTION
  MasqueAge
  ( limite integer )
  RETURNS integer
  DETERMINISTIC
BEGIN
  UPDATE Etudiants E
    SET E.moyenne = null
    WHERE
          E.age < limite;
  RETURN 1;
END//
DELIMITER ;
```

- Utilisation:
  - `SELECT MasqueAge(25);`

Fonctions et procédures

- Exemple: sélectionner les étudiants ayant une moyenne de plus que 4.0 et les mettre dans une nouvelle relation

- Procédure:

```
DELIMITER //
CREATE PROCEDURE AList()

BEGIN
  CREATE TEMPORARY TABLE
     IF NOT EXISTS
     Etudiants_A
     LIKE Etudiants;
  INSERT INTO Etudiants_A
     SELECT *
     FROM Etudiants E
     WHERE E.moyenne > 4.0;

END//
DELIMITER ;
```

Utilisation:

- `CALL AList;`

- Fonction:

```
DELIMITER //
CREATE FUNCTION AList()
   RETURNS integer
   DETERMINISTIC
BEGIN
   CREATE TEMPORARY TABLE
      Etudiants_A
      LIKE Etudiants;
   INSERT INTO Etudiants_A
      SELECT *
      FROM Etudiants E
      WHERE E.moyenne > 4.0;
   RETURN 1;
END//
DELIMITER ;
```

- Utilisation:
  - `SELECT AList();`

Commentaires

- C'est une bonne habitude de commenter notre code
  - Permet de noter les décisions et suppositions qu'on fait durant le design
  - Permet de se rappeler de ce qu'on a fait il y a plusieurs mois/années
  - Permet à un développeur futur de comprendre notre travail
- SQL permet deux formats de commentaires
  - `--` *commentaire*
    - Commentaire à partir du symbole jusqu'à la fin de la ligne en cours
  - `/*` *commentaire* `*/`
    - Commentaire multi-lignes ou intra-ligne

Commentaires

- Exemple:

```
/*Sélectionner tous les étudiants
pour une ville V spécifiée
avec une moyenne M spécifiée */
SELECT * FROM Etudiants --Students (ancien code!)
WHERE ville = V /*AND pays = 'Canada'*/ AND note = M;
```

Contrôle

- L'extension du langage SQL définit les fonctions de contrôle standards
  - `IF`
  - `CASE`
  - `WHILE`
  - `REPEAT`
  - `LOOP`
- Pas de boucle `FOR`
  - `WHILE` offre les mêmes fonctionnalités
  - Certains rares SGBDR (PostgreSQL) l'implémentent quand même

Contrôle : IF

- Choisir un chemin d'exécution selon certaines conditions
  - MySQL contraint utilisation du `IF` à l'intérieur de routines seulement
- `IF` *condition* `THEN` *commandes*
  [`ELSEIF` *condition* `THEN` *commandes*]
  […]
  [`ELSE` *commandes*]
  `END IF;`

Contrôle : IF

- Exemple: fonction pour calculer les cotes

```
DELIMITER //
CREATE FUNCTION
   CalculCotes(moyenne double) RETURNS char(1)
   DETERMINISTIC
      BEGIN
      DECLARE cote char(1);
         IF moyenne >= 4 THEN
            SET cote := 'A';
         ELSEIF moyenne >= 3 THEN
            SET cote := 'B';
         ELSE
            SET cote := 'C';
         END IF;
         RETURN cote;
      END//
DELIMITER ;
```

idul	nom	age	moyenne	cote
ALXYZ	Alice	24	4.1	null
BOUVW	Bob	27	3.6	null
CERST	Cédric	22	2.2	null
DEOPQ	Denis	15	4.2	null
ERLMN	Érica	23	2.2	null

Contrôle : IF

```
SELECT E.nom, E.moyenne,
       CalculCotes(E.moyenne)
FROM Etudiants E;
```

nom	moyenne	CalculCotes(E.moyenne)
Alice	4.1	A
Bob	3.6	B
Cédric	2.2	C
Denis	4.2	A
Érica	2.2	C

```
SELECT E.nom, E.moyenne
FROM Etudiants E
WHERE CalculCotes(E.moyenne)
      = 'A';
```

nom	moyenne
Alice	4.1
Denis	4.2

```
UPDATE etudiants
SET cote :=
    CalculCotes(moyenne);
```

idul	nom	age	moyenne	cote
ALXYZ	Alice	24	4.1	A
BOUVW	Bob	27	3.6	B
CERST	Cédric	22	2.2	C
DEOPQ	Denis	15	4.2	A
ERLMN	Érica	23	2.2	C

Contrôle : CASE

- Sélectionner une action dans une liste de conditions
  - Peut être utilisé dans une requête (pas nécessairement une routine)
- CASE
  WHEN *condition* THEN *commande*
  […]
  [ELSE *commande*]
  END

Contrôle : CASE

```
SELECT E.nom,
CASE
  WHEN E.moyenne > 4.0 THEN
    " est un excellent étudiant!"
  WHEN E.moyenne > 3.0 THEN
    " est un bon étudiant."
  ELSE " pourrait faire mieux."
END
FROM Etudiants E;
```

Alice	est un excellent étudiant!
Bob	est un bon étudiant.
Cédric	pourrait faire mieux.
Denis	est un excellent étudiant!
Érica	pourrait faire mieux.

Contrôle : WHILE

- Répéter un traitement tant qu'une condition est satisfaite
  - Si la condition est initialement fausse, ne s'exécute jamais
  - MySQL contraint utilisation du `WHILE` à l'intérieur de routines seulement
- [*étiquette*:] `WHILE` *condition* `DO`
  commandes
  `END WHILE` [*étiquette*];

Contrôle : REPEAT

- Répéter un traitement tant qu'une condition est satisfaite
  - Si la condition est initialement fausse, s'exécute une fois
  - MySQL contraint utilisation du `REPEAT` à l'intérieur de routines seulement
- [*étiquette* :] `REPEAT`
  commandes
  `UNTIL` *condition*
  `END REPEAT` [*étiquette*];

Contrôle : WHILE et REPEAT

- Exemple: fonction factorielle

- Avec WHILE

```
DELIMITER //
CREATE PROCEDURE
   FactorielleW
   (INOUT valeur integer)
BEGIN
DECLARE total integer;
   SET total = 1;
   WHILE valeur > 0 DO
      SET total =
         total * valeur;
      SET valeur =
         valeur - 1;

   END WHILE;
   SET valeur = total;
END //
DELIMITER ;
```

- Avec REPEAT

```
DELIMITER //
CREATE PROCEDURE
   FactorielleR
   (INOUT valeur integer)
BEGIN
DECLARE total integer;
   SET total = 1;
   REPEAT
      SET total =
         total * valeur;
      SET valeur =
         valeur - 1;
   UNTIL valeur < 1
   END REPEAT;
   SET valeur = total;
END //
DELIMITER ;
```

Contrôle : WHILE et REPEAT

- Exemple: fonction factorielle

- Avec WHILE

```
SET @val = 3;
CALL
FactorielleW(@val);
SELECT @val;
```

  - Retourne 6

```
SET @val = -3;
CALL
FactorielleW(@val);
SELECT @val;
```

  - Retourne 1

- Avec REPEAT

```
SET @val = 3;
CALL
FactorielleR(@val);
SELECT @val;
```

  - Retourne 6

```
SET @val = -3;
CALL
FactorielleR(@val);
SELECT @val;
```

  - Retourne -3

Contrôle : WHILE et REPEAT

- `ITERATE` *étiquette ;*
  - Termine l'exécution d'une boucle (ignore les commandes suivantes) et passe à l'itération suivante
  - L'utilisation de l'étiquette pour identifier la boucle est obligatoire!
  - Équivalent de `CONTINUE`
- `LEAVE` *étiquette ;*
  - Termine l'exécution d'une boucle (ignore les commandes suivantes) et sort de la boucle
  - L'utilisation de l'étiquette pour identifier la boucle est obligatoire!
  - Équivalent de `BREAK`

Contrôle : WHILE et REPEAT

```
DELIMITER //
CREATE PROCEDURE MoyenneBPlus()
BEGIN
   boucleNote: WHILE (SELECT AVG(E.moyenne)
                      FROM Etudiants E) < 3.33 DO
      UPDATE Etudiants E
             SET E.moyenne = E.moyenne * 1.1;
      IF (SELECT MAX(E.moyenne) FROM Etudiants E) > 4.33
      THEN
         UPDATE Etudiants E
                SET E.moyenne = 4.33
                WHERE E.moyenne > 4.33;
         SELECT 'Max atteint';
         LEAVE boucleNote;
      END IF;
      ITERATE boucleNote;
      SELECT 'Message caché';
   END WHILE boucleNote;
END //
DELIMITER ;
```

Contrôle : WHILE et REPEAT

nom	moyenne
Alice	3.3
Bob	3.0
Cédric	2.6
Denis	2.0
Érica	2.0

```
SELECT AVG(moyenne) FROM etudiants;
```

AVG(moyenne)
2.58

```
CALL MoyenneBPlus();
```

Max atteint
Max atteint

AVG(moyenne)
3.42

nom	moyenne
Alice	4.33
Bob	3.99
Cédric	3.46
Denis	3.66
Érica	3.66

Contrôle : LOOP

- Répéter un traitement inconditionnellement
  - On peut uniquement sortir de la boucle avec une commande `LEAVE` ou `RETURN`
  - MySQL contraint utilisation du `LOOP` à l'intérieur de routines seulement
- [*étiquette*:] `LOOP`
  commandes
  `END LOOP` [*étiquette*];

Contrôle : LOOP

- Exemple: fonction factorielle
  - Avec REPEAT

```
DELIMITER //
CREATE PROCEDURE
   FactorielleR
   (INOUT valeur integer)
BEGIN
DECLARE total integer;
   SET total = 1;
   REPEAT
      SET total =
        total * valeur;
      SET valeur =
        valeur - 1;
   UNTIL valeur < 1
   END REPEAT;
   SET valeur = total;
END //
DELIMITER ;
```

  - Avec LOOP

```
DELIMITER //
CREATE PROCEDURE
   FactorielleL
   (INOUT valeur integer)
BEGIN
DECLARE total integer;
SET total = 1;
   maBoucle: LOOP
      SET total =
        total * valeur;
      SET valeur = valeur - 1;
      IF valeur < 1 THEN
        SET valeur = total;
        LEAVE maBoucle;
      END IF;
   END LOOP maBoucle;
END //
DELIMITER ;
```

Contrôle : LOOP

- Exemple: fonction factorielle

- Avec REPEAT

```
SET @val = 3;
CALL
FactorielleR(@val);
SELECT @val;
```

  - Retourne 6

```
SET @val = -3;
CALL
FactorielleR(@val);
SELECT @val;
```

  - Retourne -3

- Avec LOOP

```
SET @val = 3;
CALL
FactorielleL(@val);
SELECT @val;
```

  - Retourne 6

```
SET @val = -3;
CALL
FactorielleL(@val);
SELECT @val;
```

  - Retourne -3

Exceptions

- C'est une bonne pratique de prévoir les erreurs possibles dans nos fonctions et des chemins d'exécution si elles se produisent
- Exemple: fonction de division sans gestion des erreurs

```
DELIMITER //
CREATE PROCEDURE DivisionA(IN num double,
   IN denom double, OUT resultat double)
BEGIN
   SET resultat = num / denom;
END //
DELIMITER ;
```

  - `CALL DivisionA(5,2,@R);`
    - Retourne 2.5
  - `CALL DivisionA(5,0,@R);`
    - Retourne `null`

Exceptions

- Fonction pour signaler une erreur
  - `SIGNAL SQLSTATE '`*valeur*`'`
    `[SET MESSAGE_TEXT = '`*message*`'];`
  - Valeur de 45000 = exception définie par l'utilisateur
  - Permet de terminer l'exécution et afficher un message d'erreur personnalisé

Exceptions

- Exemple: fonction de division avec gestion des erreurs

```
DELIMITER //
CREATE PROCEDURE DivisionB(IN num double,
  IN denom double, OUT resultat double)
BEGIN
  SET resultat = 0;
  IF denom = 0 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Division par zéro';
  END IF;
  SET resultat = num / denom;
END //
DELIMITER ;
```

  - `CALL DivisionB(5,2,@R);`
    - Retourne 2.5
  - `CALL DivisionB(5,0,@R);`
    - Affiche le message d'erreur
    - Ne retourne rien (@R ne change pas de valeur)

Exceptions

- On peut également définir un gestionnaire (*handler*) pour chaque cas d'erreur
- `DECLARE {CONTINUE | EXIT} HANDLER FOR` *condition* `SET` *action;*
  - Déclaré au début du bloc `BEGIN`…`END`, après les variables
  - Deux modes d'opération
    - `CONTINUE` : l'action est exécutée et la routine continue
    - `EXIT` : l'action est exécutée et la routine quitte immédiatement
  - Les conditions
    - Peut être un numéro `SQLSTATE`
    - Peut être une condition ou un état prédéfini en SQL

Exceptions

- Avec CONTINUE

```
DELIMITER //
CREATE PROCEDURE
   DivisionC(IN num double,
   IN denom double,
   OUT resultat double)
BEGIN
DECLARE CONTINUE HANDLER
   FOR SQLSTATE '45000'
   SET denom = 1;

   SET resultat = 0;
   IF denom = 0 THEN
      SIGNAL SQLSTATE '45000';
   END IF;
   SET resultat =
      num / denom;
END //
DELIMITER ;
```

- Avec EXIT

```
DELIMITER //
CREATE PROCEDURE
   DivisionD(IN num double,
   IN denom double,
   OUT resultat double)
BEGIN
DECLARE EXIT HANDLER
   FOR SQLSTATE '45000'
   SET denom = 1;

   SET resultat = 0;
   IF denom = 0 THEN
      SIGNAL SQLSTATE '45000';
   END IF;
   SET resultat =
      num / denom;
END //
DELIMITER ;
```

Exceptions

- Exemple: fonction factorielle

- Avec CONTINUE

```
CALL DivisionC
(5,2,@val);
SELECT @val;
```

  - Retourne 2.5

```
CALL DivisionC
(5,0,@val);
SELECT @val;
```

  - Retourne 5

- Avec EXIT

```
CALL DivisionD
(5,2,@val);
SELECT @val;
```

  - Retourne 2.5

```
CALL DivisionD
(5,0,@val);
SELECT @val;
```

  - Retourne 0

Curseurs

- Une commande `SELECT` retourne tous les tuples d'un coup
- Il peut être désirable de les traiter un par un
  - Traitement différent selon certaines conditions
- Un curseur permet de lire les résultats d'une requête tuple par tuple
  - Mode lecture seule
  - Dans l'ordre de la requête : impossible de changer l'ordre ou de sauter des tuples
  - Dans une routine seulement

Curseurs

- Un curseur est défini au début du bloc `BEGIN`…`END`
  - Après les variables mais avant les gestionnaires
  - `DECLARE` *nom* `CURSOR FOR` *requête;*
- Un curseur doit être ouvert avant d'être utilisé et fermé lorsqu'il n'est plus utile
  - `OPEN` *nom;*
  - `CLOSE` *nom;*

Curseurs

- On lit un tuple avec la commande `FETCH`
  - Les attributs sélectionnés sont obtenus et doivent être emmagasinés dans des variables
  - `FETCH` *nom* `INTO` *variables*;
- Si tous les tuples ont été lus, on obtient un état `NOT FOUND`
  - Doit être géré par un gestionnaire

Curseurs

```
DELIMITER //
CREATE PROCEDURE Lecture()
BEGIN
DECLARE n char(20);
DECLARE a integer;
DECLARE m double;
DECLARE lecture_complete integer DEFAULT FALSE;
DECLARE curseur CURSOR FOR SELECT E.nom, E.age, E.moyenne FROM Etudiants E;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET lecture_complete = TRUE;

   CREATE TABLE IF NOT EXISTS Liste (nom char(20), moyenne double);
   OPEN curseur;
   lecteur: LOOP
      FETCH curseur INTO n, a, m;
      IF lecture_complete THEN
         LEAVE lecteur;
      END IF;

      IF a < 18 THEN
         INSERT INTO Liste VALUES (n, null);
      ELSE
         INSERT INTO Liste VALUES (n, m);
      END IF;
   END LOOP lecteur;
   CLOSE curseur;
END //
DELIMITER ;
```

Résumé

- Programmation en SQL
  - Création de variables locales
  - Création de relations temporaires
  - Limites de blocs de code avec `DELIMITER` et `BEGIN`…`END`
  - Création de commentaires dans le code
  - Création de fonctions
  - Création de procédures
  - Contrôle d'exécution avec `IF` et `CASE`
  - Contrôle de boucles avec `WHILE`, `REPEAT`, et `LOOP`
  - Exceptions avec `SIGNAL` et `DECLARE HANDLER`
  - Lecture de résultats avec `CURSOR` et `FETCH`

Exercices

- Ramakrishnan & Gehrke
  - Ø
- Connolly & Begg
  - 8.2, 8.5, 8.10
- Comme il y a peu d'exercices pour ce module, j'ai rendu disponible une série d'exercices supplémentaires sur le site web du cours