

SQL AVANCÉ

VUES

Définition

- requête compilée
- est utilisée comme une table

exemple

```
CREATE OR REPLACE VIEW vue_personnes AS
SELECT
    UCASE(nom) as nom,
    prenom,
    CONCAT_WS(' ',prenom, UCASE(nom)) as nom_complet,
    YEAR(CURDATE()) - YEAR(date_naissance) as age
FROM personnes;
```

```
SELECT * FROM vue_personnes;
```

Utilité

- sécurité / confidentialité
masque les tables et les colonnes
- simplification
 - colonnes calculées / renommées
 - dénormalisation
 - jointures complexes

Restrictions

- pas d'index sur les vues
- pas de sous requêtes dans la clause FROM
(mais il est possible d'y mettre une vue)
- pas de variables

Algorithme

- MERGE**
la vue utilise la requête SQL
- TEMPTABLE**
la vue utilise une table temporaire, elle libère donc plus rapidement les verrous sur ses tables
- UNDEFINED (défaut)**
le moteur choisit l'algorithme (MERGE le plus souvent)

```
CREATE OR REPLACE  
ALGORITHM=TEMPTABLE  
VIEW ma_vue AS ...
```

Vue modifiable

- une seule table (pas de jointures)**
- présence de toutes les colonnes NOT NULL**
- pas de fonction ou de colonne calculée**
- pas de regroupement ni de tri
GROUP BY, ORDER BY**
- ALGORITHM = MERGE**

Contrainte sur une vue

- une vue modifiable contient des critères dans la clause WHERE
- WITH CHECK OPTION = contrainte sur ces critères

ici impossible de créer un mineur à partir de cette vue

```
CREATE OR REPLACE VIEW
vue_pers_majeures
AS SELECT * FROM personnes
WHERE date_naissance >=
DATE_ADD(CURDATE(), INTERVAL -18 YEAR)
WITH CHECK OPTION;
```

CONDITIONS

La condition remplace

- une colonne dans la clause SELECT
- une valeur dans les clauses WHERE, ORDER BY ou HAVING

Condition binaire

```
IF(  
    condition,  
    valeur si vrai,  
    valeur si faux  
)
```

exemple

```
SELECT
    nom,
    IF(age < 18, 'mineur', 'majeur') as statut
FROM personnes;
```

```
SELECT
    SUM(
        IF(MONTH(date_vente) = 1, montant, 0)
    ) as ventes_janvier
FROM ventes;
```

Conditions multiples

```
CASE
    WHEN condition1 THEN
        valeur
    WHEN condition2 THEN
        valeur
    ELSE
        valeur par défaut
END
```

exemple

```
SELECT
    nom,
CASE
    WHEN age BETWEEN 0 AND 5 THEN 'bébé'
    WHEN age BETWEEN 5 AND 13 THEN 'enfant'
    WHEN age BETWEEN 13 AND 18 THEN 'ado'
    ELSE
        'adulte'
END as tranche_age
FROM vue_personnes
```

exemple avec agrégats et tri

```
SELECT
    CASE
        WHEN age BETWEEN 0 AND 5 THEN 'bébé'
        WHEN age BETWEEN 5 AND 13 THEN 'enfant'
        WHEN age BETWEEN 13 AND 18 THEN 'ado'
        ELSE 'adulte'
    END as tranche_age,
    COUNT(*) as nb
FROM vue_personnes
GROUP BY tranche_age
ORDER BY
    CASE
        WHEN age BETWEEN 0 AND 5 THEN 1
        WHEN age BETWEEN 5 AND 13 THEN 2
        WHEN age BETWEEN 13 AND 18 THEN 3
        ELSE 4
    END
```

COALESCE

Permet de spécifier une valeur lorsqu'une colonne est nulle. Très utile pour les regroupement avec WITH ROOLUP

```
SELECT  
COALESCE(ville, 'total des villes'),  
COUNT(*) as nb  
FROM contacts  
GROUP BY ville WITH ROLLUP
```

Variables de session

Déclaration

```
SET      @nom_variable1 := valeur,  
        @nom_variable2 := valeur;
```

- commence par @
- opérateur = ou :=
- valide pour la session

Exemples

```
SET @rang := 0;  
  
SELECT *, (@rang:= @rang +1) as rang FROM personnes;
```

```
SET @total := 0;  
  
SELECT *, (total:= @total + montant) as cumul  
FROM ventes;
```

Import de données

Principes

**Importation d'un fichier texte
délimité dans une table**

Syntaxe

```
LOAD DATA [LOCAL] INFILE chemin-fichier  
INTO TABLE table  
(liste-des-colonnes);
```

**LOCAL si le fichier se trouve
sur le poste exécutant la requête,
sinon, le fichier doit se trouver
sur le serveur de base de données**

**Liste des colonnes dans
l'ordre d'apparition
dans le fichier texte**

Options FIELDS

option	description	défaut
TERMINATED BY	délimiteur de colonne	\t (tabulation)
[OPTIONALLY] ENCLOSED BY	délimiteur de valeur	
ESCAPED BY	caractère d'échappement	\

Options LINES

option	description	défaut
TERMINATED BY	délimiteur de fin de ligne	\n (retour ligne)
STARTING BY	délimiteur de début de ligne	
IGNORE	nombre de lignes ignorées (en-tête)	

Gestion des doublons

```
LOAD DATA [LOCAL] INFILE  
[REPLACE | IGNORE]  
chemin-fichier  
INTO table  
(liste-des-colonnes);
```

- REPLACE** : remplace la ligne en doublon
- IGNORE** : ignore la ligne en doublon

Il faut que le fichier chargé contienne la clef primaire

Exemple

```
LOAD DATA LOCAL INFILE  
'c:/import.txt'  
INTO TABLE personnes  
FIELDS  
    TERMINATED BY ';'  
    OPTIONALLY ENCLOSED BY '\"'  
LINES  
    TERMINATED BY '\r\n'  
IGNORE 1 LINES  
  
(nom, prenom, date_naissance);
```

```
"nom";"prenom";"date"  
"martin";"pierre";"2000-10-5"  
"petri";"aline";NULL
```

**NULL n'est pas entre guillemets
il sera interprété comme null
et non comme une chaîne 'null'**

Colonnes calculées

On désire traiter les valeurs de la source de données avant de les insérer

```
LOAD DATA LOCAL INFILE  
'c:/import.txt'  
INTO TABLE personnes  
  
...  
(@nom, prenom, @date_naissance)  
SET  
date_naissance =  
    str_to_date(@date_naissance, '%d/%m/%Y') ,  
nom = UCASE(@nom) ;
```

```
"nom";"prenom";"date"  
"martin";"pierre";"15/12/1983"  
"petri";"aline";NULL
```

Sous-requêtes

Principes

- une requête dans une autre requête entre parenthèses
- le résultat de la sous requête remplace
 - une valeur
 - une liste de valeur
 - une table

Les retours

retour

clauses

une valeur

SELECT, WHERE, HAVING

une colonne

IN()

une ligne

(col1, col2) IN()

une table

FROM, ALL(), ANY()

sous requête non corrélée

le résultat n'est calculé qu'une fois

```
SELECT
    YEAR(date_vente) as année,
    COUNT(*) as nb,
    COUNT(*)/(SELECT COUNT(*) FROM ventes)
    as ratio
FROM ventes
GROUP BY YEAR(date_vente)
```

sous requête corrélée

le résultat est calculé pour chaque ligne

```
SELECT
    c.id_client,
    c.nom,
    c.prenom,
    (   SELECT MAX(cdes.date_cde) FROM cdes
        WHERE cdes.id_client=c.id_client
    ) as date_derniere_commande
FROM clients as c
```

Sous-requête dans la clause WHERE

```
-- meilleur vendeur
SELECT
    vendeurs.id_vendeur,
    vendeurs.nom,
    vente.date_vente
FROM vendeurs
INNER JOIN ventes as v1
ON vendeurs.id_vendeur = v1.id_vendeur
WHERE
    v1.vente = (SELECT MAX(v2.vente) FROM ventes as v2)
```

Intersection de deux ensembles

```
-- Les clients ayant passé une commande
SELECT
    clients.id_client,
    clients.nom,
    clients.prenom
FROM clients
WHERE clients.id_client IN
(SELECT cdes.id_client FROM cdes)
```

même résultat avec une jointure

```
-- Les clients ayant passé une commande
SELECT DISTINCT
    clients.id_client,
    clients.nom,
    clients.prenom
FROM clients
INNER JOIN cdes
ON clients.id_client = cdes.id_client
```

Soustraction de deux ensembles

```
-- Les clients n'ayant pas passé de commande
SELECT
    clients.id_client,
    clients.nom,
    clients.prenom
FROM clients
WHERE clients.id_client NOT IN
(SELECT cdes.id_client FROM cdes)
```

même chose avec une jointure gauche

```
-- Les clients n'ayant pas passé de commande
SELECT
    clients.id_client,
    clients.nom,
    clients.prenom
FROM clients
LEFT JOIN cdes ON clients.id_client = cdes.id_client
WHERE cdes.id_client IS NULL
```

EXISTENCE

```
-- Les clients ayant passé une commande
SELECT
    clients.id_client,
    clients.nom,
    clients.prenom
FROM clients
WHERE
    EXISTS(SELECT cdes.id_cde FROM cdes
    WHERE cdes.id_client = clients.id_client);
```

NON EXISTENCE

```
-- Les clients n'ayant pas passé de commande
SELECT
    clients.id_client,
    clients.nom,
    clients.prenom
FROM clients
WHERE
    NOT EXISTS(SELECT cdes.id_cde FROM cdes
    WHERE cdes.id_client = clients.id_client);
```

UNION

**L'union fusionne plusieurs requêtes en une seule,
pour cela il faut respecter les règles suivantes :**

- toutes les requêtes doivent retourner
le même nombre de colonnes**
- les types des colonnes doivent être compatibles**
- les noms des colonnes sont déterminés
par la première requête**

```
-- liste des clients et des vendeurs
SELECT
    id_client as id,
    nom,
    cp,
    'client' as categorie
FROM clients

UNION

SELECT
    id_vendeur,
    nom,
    cp,
    'vendeur'
FROM vendeurs

ORDER BY cp;
```

ANY

```
-- Les clients avec le même code postal
-- qu'au moins un des vendeurs
SELECT
    c.id_client,
    c.nom,
    c.prenom
FROM clients as c
WHERE c.cp = ANY(SELECT v.cp FROM vendeurs as v);
```

ALL

```
-- client avec un code postal différent  
-- de celui de tous les vendeurs  
SELECT  
    c.id_client,  
    c.nom,  
    c.prenom  
FROM clients as c  
WHERE c.cp != ALL(SELECT v.cp FROM vendeurs as v);
```

Sous requête dans la clause FROM

```
SELECT
    t.age,
    count(*) as nb
FROM (
    SELECT
        p.personne_id,
        (YEAR(CURDATE()) - YEAR(p.date_naissance)) as age
    FROM personnes p
) as t
GROUP BY t.age;
```

```
SELECT t.id, t.nom, t.cp, villes.nom_ville, t.categorie
FROM
  (SELECT
    id_client as id,
    nom,
    cp,
    'client' as categorie
  FROM clients

  UNION

  SELECT
    id_vendeur,
    nom,
    cp,
    'vendeur'
  FROM vendeurs) as t
INNER JOIN villes ON villes.cp = t.cp
ORDER BY cp
```

**Si une sous requête est complexe ou
qu'elle est utilisée plus d'une fois,
il est possible d'en faire une vue**

Procédures stockées

Points clefs

- du code SQL
- du code procédural
- des paramètres
- gestion des erreurs

Pourquoi

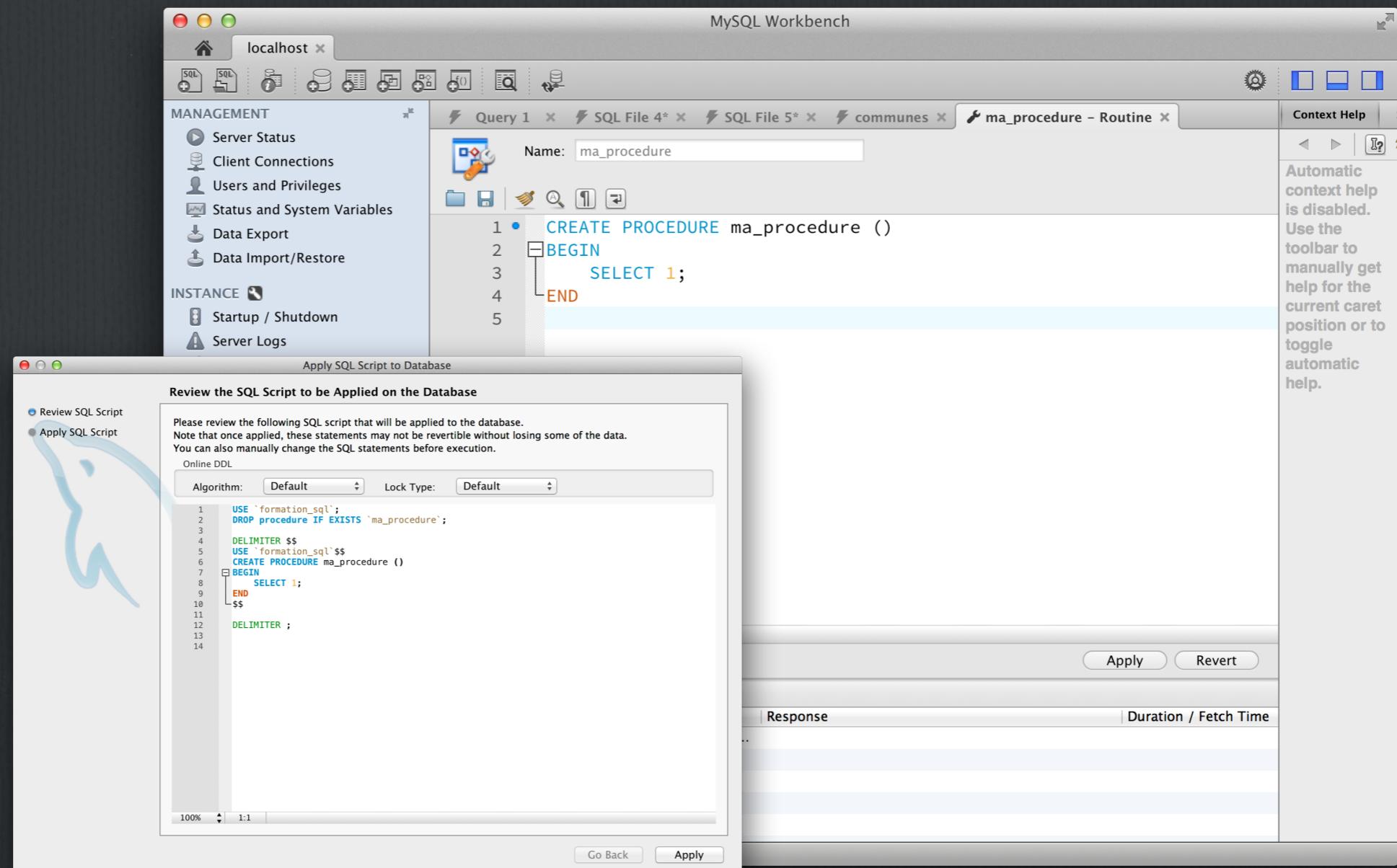
- coder les règles métier et contraintes
- exécuter plusieurs requêtes SQL
- sécuriser les accès à la base de données
- limiter le traffic sur le réseau

Syntaxe

```
DROP PROCEDURE IF EXISTS nom_procedure;  
DELIMITER $$  
CREATE PROCEDURE nom_procedure ()  
BEGIN  
    SELECT 1;  
END $$  
DELIMITER ;
```

Il faut indiquer un délimiteur personnalisé car les instructions de la procédure sont délimitées par ";"

MySQL Workbench



Appel d'une procédure

```
CALL nom_procedure();
```

Une procédure peut en appeler une autre

```
CREATE PROCEDURE proc_parent ()  
BEGIN  
    CALL proc_enfant();  
END
```

Commentaires

```
-- un commentaire mono-ligne  
  
# un autre commentaire mono-ligne  
  
/* Un commentaire  
   multi-ligne  
*/
```

Paramètres

- comme pour une fonction PHP ou Java
- type obligatoire
- préfixe conseillé

```
CREATE PROCEDURE bonjour (
    p_age TINYINT,
    p_nom VARCHAR(30)
)
BEGIN
    SELECT CONCAT_WS(' ', 
        'Bonjour', p_nom,
        'vous avez', p_age, 'ans'
    );
END
```

Déclaration des variables

- déclaration des variables au début
- déclaration multiples
- valeur par défaut
- préfixe conseillé

```
CREATE PROCEDURE test_var ()  
BEGIN  
    DECLARE v_nb_clients INT;  
    DECLARE v_cpt INT DEFAULT 0;  
    DECLARE v_nom, v_prenom  
    VARCHAR(30);  
    # Instructions  
  
    SELECT v_cpt as compteur;  
END
```

Affectation des variables

- instruction SET
- opérateur "=" ou ":="

```
DECLARE v_nb INT;
DECLARE v_cpt INT;
DECLARE v_message VARCHAR(30);
SET v_nb_clients := (SELECT count(*) FROM clients);
SET v_cpt := v_cpt + 1;
SET v_message := CONCAT('il y a ', v_nb, ' clients');

SELECT v_message;
```

Affectations multiples

```
DECLARE v_nom, v_prenom VARCHAR(30);
DECLARE v_cp VARCHAR(5);

SELECT nom, prenom, cp FROM clients WHERE id_client=1
INTO v_nom, v_prenom, v_cp;
```

Direction des paramètres

- IN (entrée) par défaut
- OUT (sortie)
- INOUT (entrée sortie)

Paramètre de sortie

```
CREATE PROCEDURE proc_statistiques(
    IN p_annee SMALLINT,
    OUT p_total_ventes DOUBLE,
    OUT p_nb_vendeurs INT,
    OUT p_nb_departements INT
)
BEGIN

SELECT
    SUM(ventes.montant),
    COUNT(DISTINCT ventes.vendeur_id),
    COUNT(DISTINCT departement_id)
FROM ventes
WHERE YEAR(date_vente) = p_annee
INTO p_total_ventes, p_nb_vendeurs, p_nb_departements;

END
```

Paramètre de sortie utilisation

```
SET @total_ventes = 0;
SET @nb_vendeurs = 0;
SET @nb_departements = 0;

CALL formation_sql.proc_statistiques(
    2015,
    @total_ventes,
    @nb_vendeurs,
    @nb_departements
);

SELECT @total_ventes, @nb_vendeurs, @nb_departements;
```

Paramètres d'entrée-sortie

```
CREATE PROCEDURE proc_vente_total_inout(
    IN p_annee INT,
    INOUT p_total DECIMAL(10,2)
)
BEGIN
DECLARE v_total DECIMAL(10,2);

SELECT SUM(montant) FROM ventes
WHERE YEAR(date_vente) = p_annee
INTO v_total;

SET p_total := p_total + v_total;
END
```

Paramètres d'entrée-sortie utilisation

```
SET @total := 0;
-- cumul des ventes dans la variable @total
CALL proc_vente_total_inout(2014, @total);
CALL proc_vente_total_inout(2015, @total);

SELECT FORMAT(@total,2);
```

Structures algorithmiques

Condition

```
DECLARE v_remise DECIMAL(3,2);

IF p_total > 500 THEN
    SET v_remise := 0.1;
ELSE IF p_total > 100 THEN
    SET v_remise := 0.05;
ELSE
    SET v_remise := 0;
END IF;
```

CASE

```
DECLARE v_remise DECIMAL(3,2);

CASE
    WHEN p_total > 500 THEN
        SET v_remise := 0.1;
    WHEN p_total > 100 THEN
        SET v_remise := 0.05;
    ELSE
        SET v_remise := 0;
END CASE;
```

Boucle

"tant que"

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

Un exemple

```
-- somme de tous les nombres de 1 à 100
DECLARE v_somme INT DEFAULT 0;
DECLARE v_entier SMALLINT DEFAULT 1;

WHILE (v_entier <= 100) DO
    SET v_somme := v_somme+v_entier;
    SET v_entier := v_entier+1;
END WHILE;

SELECT v_somme;
```

Boucle

"jusqu'à ce que"

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

un exemple

```
-- somme de tous les nombres de 1 à 100
DECLARE v_somme INT DEFAULT 0;
DECLARE v_entier SMALLINT DEFAULT 1;

REPEAT
    SET v_somme := v_somme + v_entier;
    SET v_entier := v_entier + 1;
UNTIL v_entier > 100 END REPEAT;

SELECT v_somme;
```

Boucle infinie

```
[etiquette:] LOOP  
    instructions;  
END LOOP [etiquette];
```

Un exemple

```
-- somme de tous les nombres de 1 à 100
DECLARE v_somme INT DEFAULT 0;
DECLARE v_entier SMALLINT DEFAULT 1;

boucleSomme: LOOP
    SET v_somme := v_somme + v_entier;
    SET v_entier := v_entier + 1;

    -- condition pour sortir de la boucle
    IF v_entier > 100 THEN
        LEAVE boucleSomme;
    END IF;

END LOOP boucleSomme;
```

Quelques exemples

Contrôle de validité

```
CREATE PROCEDURE personne_insert (
    p_nom VARCHAR(30),
    p_prenom VARCHAR(30),
    p_date_naissance DATE
)
BEGIN
    -- n'insère une personne que si son nom n'est pas vide
    IF p_nom IS NOT NULL AND trim(p_nom) != '' THEN
        INSERT INTO personnes (nom, prenom, date_naissance)
        VALUES (UPPER(p_nom), p_prenom, p_date_naissance);
    END IF;
END
```

avec un paramètre de sortie

```
CREATE PROCEDURE personne_insert(
    p_nom VARCHAR(30),
    p_prenom VARCHAR(30),
    p_date_naissance DATE,
    OUT p_personne_id INT
)
BEGIN
    -- n'insère une personne que si son nom n'est pas vide
    IF p_nom IS NOT NULL AND trim(p_nom) != '' THEN
        INSERT INTO personnes (nom, prenom, date_naissance)
        VALUES (UPPER(p_nom), p_prenom, p_date_naissance);

        -- Récupération de l'identifiant de la nouvelle personne
        SET p_personne_id := LAST_INSERT_ID();
    END IF;
END
```

insertion d'une adresse

```
CREATE PROCEDURE adresse_insert(
    p_adresse VARCHAR(30),
    p_code_postal VARCHAR(5),
    p_ville VARCHAR(30),
    p_personne_id INT
)
BEGIN
    DECLARE v_commune_id INT;

    -- Récupération de l'identifiant de la commune
    SELECT commune_id FROM communes WHERE
    code_postal = p_code_postal AND
    commune = p_ville INTO v_commune_id;

    -- Insertion de l'adresse
    INSERT INTO adresses (personne_id, commune_id, adresse)
    VALUES (p_personne_id, v_commune_id, p_adresse);
END
```

```
-- insertion contact et adresse
CREATE PROCEDURE personne_adresse_insert_1(
    p_nom VARCHAR(30), p_prenom VARCHAR(30),
    p_date_naissance DATE, p_adresse VARCHAR(50),
    p_code_postal CHAR(5), p_ville VARCHAR(30)
)
BEGIN
    DECLARE v_commune_id, v_adresse_id INT;

    SET @personne_id := NULL;

    -- Insertion de la personne
    CALL personne_insert(
        p_nom, p_prenom, p_date_naissance, @personne_id);

    -- Insertion de l'adresse
    CALL adresse_insert(
        p_adresse, p_code_postal, p_ville, @personne_id
    );
END
```

Tests

- utiliser la procédure pour ajouter une personne et une adresse
- ajouter deux fois la même personne
- ajouter une personne avec un mauvais code postal

Gestion des erreurs

Gestionnaires d'erreurs

```
-- gestion d'erreur avec action
DECLARE action HANDLER FOR condition
BEGIN
    -- gestion de l'erreur
    SELECT 'erreur';
END;

-- gestion d'erreur avec affectation d'une variable
DECLARE action HANDLER FOR condition
SET variable := 1;
```

Actions

EXIT

sort du bloc BEGIN..END en cours

CONTINUE

ignore l'erreur

Conditions

SQLEXCEPTION

toute erreur
hors SQLWARNING et NOT FOUND

NOT FOUND

SQLSTATE 02...

SQLWARNING

SQLSTATE 01...

SQLSTATE

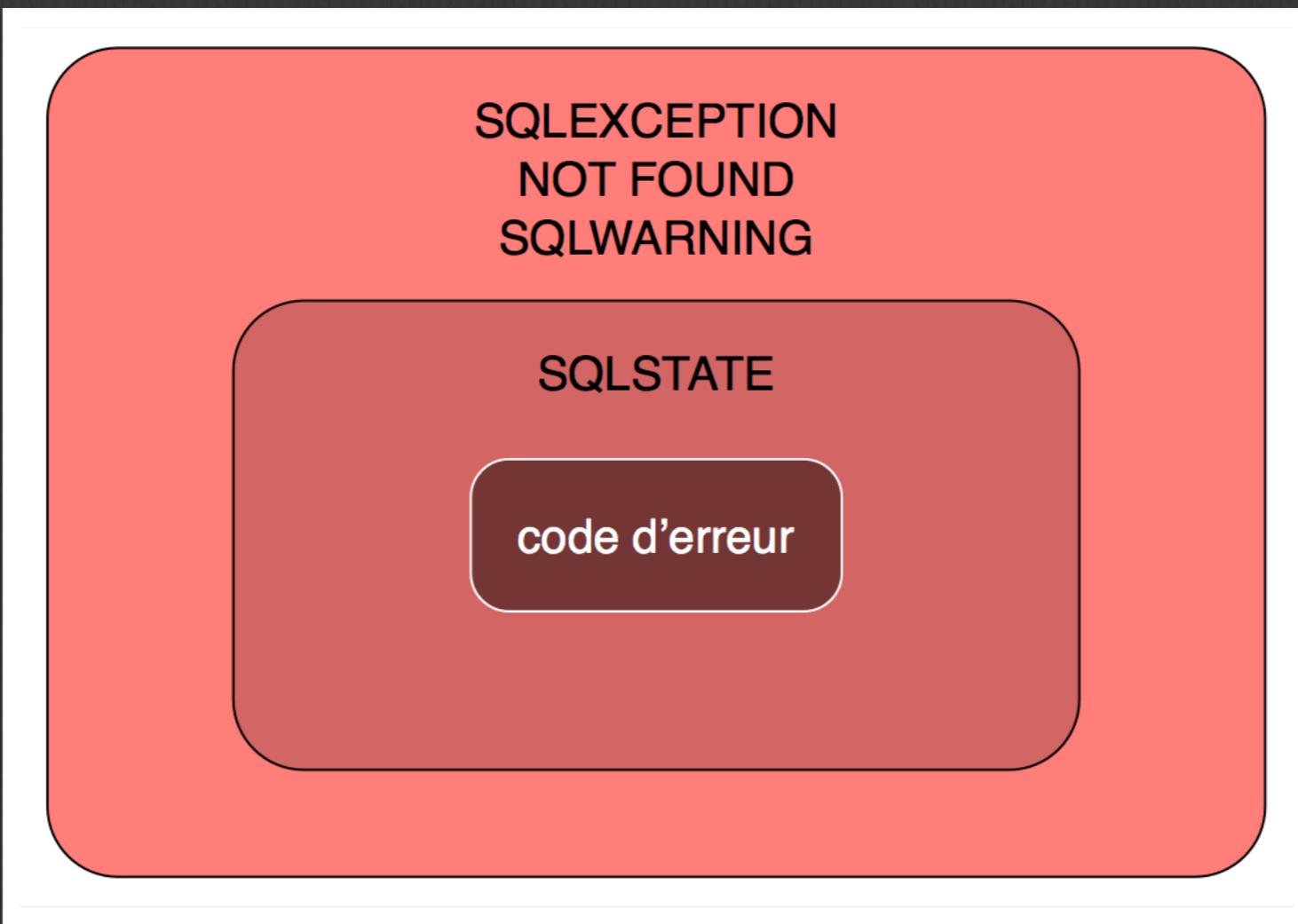
un code d'état de 5 caractères

code

un code d'erreur à 4 chiffres

Imbrication des conditions

liste des codes d'erreurs



Exemple

```
-- Affichage en cas d'erreur
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    SELECT false as succes;
END;

-- Affectation d'une variable en cas d'erreur
DECLARE CONTINUE HANDLER FOR NOT FOUND SET found := FALSE;
```

Lever une exception

```
SIGNAL condition
```

```
    SET MESSAGE_TEXT = 'message d'erreur';
```

```
IF p_nom IS NULL OR TRIM(p_nom) = '' THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'le nom doit être renseigné';
END IF;
```

Exception nommée

```
DECLARE erreur_metier CONDITION FOR SQLSTATE '45000';

SIGNAL erreur_metier
    SET MESSAGE_TEXT = 'message d'erreur';
```

Obtenir des détails sur les erreurs

MySQL 5.6+ uniquement

```
GET DIAGNOSTICS CONDITION 1
    @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;
SELECT @p1 as sqlstate, @p2 as message;
```

**CONDITION 1 retourne la dernière erreur,
CONDITION 2 retourne l'avant dernière erreur**

Une gestion d'erreur

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    GET DIAGNOSTICS CONDITION 1
    @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;

    SELECT false as succes, @p1 as etat, @p2 as message;
END;
```

```
CREATE PROCEDURE personne_adresse_insert_2(
    p_nom VARCHAR(30),
    p_prenom VARCHAR(30),
    p_date_naissance DATE,
    p_adresse VARCHAR(50),
    p_code_postal CHAR(5),
    p_ville VARCHAR(30)
)
BEGIN
    DECLARE v_commune_id, v_adresse_id INT;

    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        GET DIAGNOSTICS CONDITION 1
        @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;

        SELECT false as succes, @p1 as etat, @p2 as message;
    END;

    -- Création de la personne et de l'adresse
    -- insérer le code ici
END
```

Les fonctions

En bref

- Procédures stockées avec une valeur de retour
- Doit retourner une et une seule valeur scalaire
- Peut être utilisée dans une expression ou remplacer une colonne dans un SELECT

Limites

- paramètres IN uniquement
- pas de SQL dynamique
- pas de transactions
- ne peut pas modifier la table parente
(qui fait appel à la fonction)

Syntaxe

```
CREATE FUNCTION bonjour()
RETURNS CHAR(5)
BEGIN
    RETURN 'Hello';
END
```

```
CREATE FUNCTION bonjour2(p_nom VARCHAR(50))
RETURNS VARCHAR(50)
BEGIN
    RETURN CONCAT_WS(' ', 'Hello', p_nom);
END
```

Calcul de l'âge

```
-- calcul de l'âge en fonction de la date de naissance
CREATE FUNCTION calc_age(p_date_naissance DATE)
RETURNS decimal(5,2)
BEGIN
RETURN ROUND(
    YEAR(CURDATE()) - YEAR(p_date_naissance) +
    (MONTH(CURDATE()) - MONTH(p_date_naissance)) / 12
    ,2
);
END

-- Utilisation de la fonction
SELECT nom, calc_age(date_naissance) FROM personnes;
```

```
-- nombre entier aléatoire
-- avec simulation de distribution normale
CREATE FUNCTION alea_entier(
    p_min INT,
    p_max INT,
    p_nb_tirage TINYINT)
RETURNS INT
BEGIN
    DECLARE i TINYINT DEFAULT 0;
    DECLARE v_total INT DEFAULT 0;

    REPEAT
        SET i := i+1;
        SET v_total := v_total + p_min + (RAND() * p_max);
    UNTIL i = p_nb_tirage END REPEAT;

    RETURN ROUND(v_total/p_nb_tirage);
END
```

Les tables temporaires

En bref

- détruite à la fin de la session
- même syntaxe que celle des tables permanentes
- mot clef TEMPORARY
- permet de simuler un tableau
- ENGINE=MEMORY (plus performant)

```
CREATE FUNCTION TRI_MOT(p_mot VARCHAR(50))
RETURNS VARCHAR(50)
BEGIN
    DECLARE v_mot VARCHAR(50) DEFAULT '';
    DECLARE v_taille_mot, v_pos TINYINT DEFAULT 0;
    DECLARE v_car CHAR(1) DEFAULT '';

    SET v_taille_mot := CHAR_LENGTH(p_mot);

    DROP TEMPORARY TABLE IF EXISTS lettres;
    CREATE TEMPORARY TABLE lettres (lettre CHAR(1) NOT NULL)
    ENGINE=memory DEFAULT CHARSET utf8 COLLATE utf8_unicode_ci;
    -- Truncate est interdit dans les fonctions
    DELETE FROM lettres;

    WHILE (v_pos < v_taille_mot) DO
        SET v_pos := v_pos +1;
        SET v_car := SUBSTRING(p_mot, v_pos, 1);

        INSERT INTO lettres (lettre) VALUES (v_car);
    END WHILE;

    SET v_mot := (SELECT GROUP_CONCAT(lettre
        ORDER BY lettre SEPARATOR '') FROM lettres);

    RETURN v_mot;
END
```

Les transactions

Principe

**Exécuter une suite d'instructions comme un seul bloc.
L'ensemble des instructions doit réussir pour que le bloc
soit validé.**

**Mécanisme de sécurité crucial pour les transferts bancaires
par exemple.**

Instructions

START TRANSACTION

début de transaction

COMMIT

validation de la transaction

ROLLBACK

annulation de la transaction

Exemple

```
START TRANSACTION;

INSERT INTO ventes
(date_vente, montant, vendeur_id, departement_id)
('2008-01-01', 500, 1, 1);

-- Cette requête échoue
-- car il n'y a pas de vendeur avec id=90
INSERT INTO ventes
(date_vente, montant, vendeur_id, departement_id)
('2008-01-01', 500, 90, 1);

COMMIT;
```

```
CREATE PROCEDURE `personne_transaction`(
    p_nom VARCHAR(30),
    p_prenom VARCHAR(30),
    p_date_naissance DATE,
    p_adresse VARCHAR(50),
    p_code_postal CHAR(5),
    p_ville VARCHAR(30)
)
BEGIN
    -- Erreur personnalisée
    DECLARE BUSINESS_RULE_VIOLATED CONDITION FOR SQLSTATE '45000';

    -- Gestionnaire d'erreur
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        GET DIAGNOSTICS CONDITION 1
        @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;

        SELECT false as succes, @p1 as etat, @p2 as message;
    END;

    -- Début de la transaction
    START TRANSACTION;

    SET @personne_id := NULL;

    -- Insertion de la personne
    CALL personne_insert(
        p_nom, p_prenom, p_date_naissance,
        @personne_id);

    -- Insertion de l'adresse
    CALL adresse_insert(
        p_adresse, p_code_postal,
        p_ville, @p_personne_id
    );

    SIGNAL BUSINESS_RULE_VIOLATED
    SET MESSAGE_TEXT = 'Invalid parameter values';

    -- Validation de la transaction
    COMMIT;

    -- Affichage du résultat
    SELECT 'OK' FROM DUAL;
END
```

Les triggers

Principe

Une procédure associée à un événement

- BEFORE, AFTER

- DELETE, INSERT, UPDATE

Syntaxe

```
CREATE TRIGGER nom_declencheur
  { BEFORE | AFTER } { DELETE | INSERT | UPDATE }
ON nom_table

FOR EACH ROW
BEGIN
  instructions
END;
```

Suppression

```
DROP TRIGGER nom_declencheur;
```

Lors d'un **DROP TABLE**, tous les déclencheurs associés à cette table sont également supprimés

Limites

- pas de sql dynamique
- ne peut modifier la table parente
- si la structure de la base change, il faut recompiler le trigger
- pas d'activation sur les actions générées par les contraintes

```
-- Empêcher l'inscription d'une personne
-- à une formation de niveau inférieur
DROP TRIGGER IF EXISTS inscription_BEFORE_INSERT;

DELIMITER $$

CREATE TRIGGER inscription_BEFORE_INSERT
BEFORE INSERT ON inscription_formation FOR EACH ROW
BEGIN

DECLARE v_niveau_formation TINYINT;

SELECT niveau FROM inscription_formation
WHERE formation_id = NEW.formation_id
AND personne_id = NEW.personne_id
INTO v_niveau_formation;

IF v_niveau_formation IS NOT NULL
    AND v_niveau_formation < NEW.niveau THEN
    SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cette personne est déjà inscrite
                            à une formation de niveau supérieur';
END IF;
END $$

DELIMITER ;
```

```
-- Mise en forme des données
DROP TRIGGER IF EXISTS personnes_BEFORE_INSERT;
DROP TRIGGER IF EXISTS personnes_BEFORE_UPDATE;

DELIMITER $$

CREATE TRIGGER personnes_BEFORE_INSERT
BEFORE INSERT ON personnes FOR EACH ROW
BEGIN
    SET NEW.nom := TRIM(UPPER(NEW.nom));
END $$

CREATE TRIGGER personnes_BEFORE_UPDATE
BEFORE UPDATE ON personnes FOR EACH ROW
BEGIN
    SET NEW.nom := TRIM(UPPER(NEW.nom));
END $$

DELIMITER ;
```

Les tâches planifiées

En bref

- planification des traitements
ce soir à 23h
- répétition à intervalle réguliers
tous les vendredi à 20h
- sorte de trigger temporel

Utilité

- opération de maintenance
- pré-calcul de statistiques lourdes
- audit et tests de cohérence
- archivage automatisé

Syntaxe sans répétition

```
DELIMITER $$  
CREATE EVENT [IF NOT EXISTS] nom_evenement  
ON SCHEDULE AT timestamp [+ INTERVAL intervalle]  
[ON COMPLETION [NOT] PRESERVE]  
[COMMENT 'commentaire']  
DO  
    BEGIN  
        -- instructions_SQL  
    END;  
$$
```

Syntaxe tâche répétée

```
DELIMITER $$  
CREATE EVENT [IF NOT EXISTS] nom_evenement  
ON SCHEDULE EVERY intervalle  
[STARTS timestamp [+ INTERVAL intervalle] ...]  
[ENDS timestamp [+ INTERVAL intervalle] ...]  
[ON COMPLETION [NOT] PRESERVE]  
[COMMENT 'commentaire']  
DO  
    BEGIN  
        -- instructions_SQL;  
    END;  
$$
```

Exemples de planification

AT '2016-01-01 00:00:00'

une seule fois le 1er janvier 2016 à minuit

AT CURRENT_TIMESTAMP + INTERVAL
30 MINUTE

une seule fois 30 minutes après la création

AT NOW + INTERVAL 1 YEAR +
INTERVAL 1 DAY

une seule fois un mois et un jour après la création

EVERY 1 DAY
STARTS '2015-12-01 02:00:00'

tous les jours à 2h du matin à partir du 1er décembre 2015

EVERY 5 MINUTE
STARTS NOW
ENDS NOW + INTERVAL 1 WEEK

toutes les 5 minutes à partir de maintenant et pendant une semaine

Suppression

```
DROP EVENT IF EXISTS nom_evenement;
```

Modifications

```
ALTER EVENT nom_evenement DISABLE;  
  
-- réactivation d'un événement  
ALTER EVENT nom_evenement ENABLE;  
  
-- changement de nom  
ALTER EVENT nom_evenement RENAME TO autre_nom;  
  
-- changement de planification  
ALTER EVENT nom_evenement ON SCHEDULE  
AT '2015-11-20 12:00:00';  
  
ALTER EVENT nom_evenement ON SCHEDULE EVERY 1 WEEK  
STARTS NOW + INTERVAL 5 HOUR;
```

Modifications

```
-- changement de code
DELIMITER $$

ALTER EVENT nom_evenement DO
BEGIN
    -- instructions
END;
$$
```

Exemple

```
DROP EVENT IF EXISTS archivage_ventes;

DELIMITER $$

CREATE EVENT archivage_ventes
ON SCHEDULE EVERY 1 MONTH
STARTS '2015-01-01 01:00:00'
DO
BEGIN
    CALL ARCHIVER_VENTES();
END;

$$
```

```
DROP procedure IF EXISTS ARCHIVER_VENTES;
DELIMITER $$

CREATE PROCEDURE ARCHIVER_VENTES()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        BEGIN
            ROLLBACK;
        END;

    START TRANSACTION;
    INSERT INTO ventes_archivage
        SELECT * FROM ventes WHERE date_vente <
        DATE_ADD(CURDATE(), INTERVAL - 1 MONTH);

    DELETE v FROM ventes as v INNER JOIN
        ventes_archive as va ON v.vente_id = va.vente_id;
    COMMIT;

END$$
DELIMITER ;
```