

Aide-mémoire SQL



Antoine Dinimant / Epita 2025

Généralités

Subdivisions du SQL et principales instructions

DML

data manipulation language

SELECT DELETE
INSERT UPDATE

DDL

data definition language

CREATE [OR REPLACE]
DROP
ALTER

DCL

data control language

GRANT
REVOKE

TCL

transaction control language

START TRANSACTION
COMMIT
ROLLBACK

Programing Language

<i>CREATE TRIGGER</i>	<i>CASE / IF</i>
<i>CREATE FUNCTION</i>	<i>LOOP / WHILE</i>
<i>CREATE PROCEDURE</i>	<i>FETCH</i>
<i>BEGIN... END</i>	<i>RETURN</i>
<i>DECLARE</i>	<i>...</i>

**Forte spécificité
de chaque SGBD**

Structure d'une requête SELECT

WITH sous-requêtes nommées

SELECT colonnes à afficher

FROM table(s)

WHERE condition(s) individuelles

GROUP BY critère(s) de regroupement

HAVING condition(s) statistiques

ORDER BY critères de regroupement

Sauf mention contraire, les syntaxes présentées dans ce document relèvent du SQL standard et fonctionnent sous MySQL 8 en mode ANSI.

Les syntaxes spécifiques à MySQL ou à un autre SGBD sont indiquées en rouge.

Opérateurs de comparaison (prédicats)

=	<>	!=	
<	>	<=	>=
IN (<i>élément1</i> , <i>élément2</i> , ...)	NOT IN (...)		
BETWEEN <i>début</i> AND <i>fin</i>	NOT BETWEEN ... AND ...		
LIKE ' <i>début</i> %'	NOT LIKE ' <i>C_cile</i> '		
IS NULL	IS NOT NULL		
AND	EXISTS (<i>sous-requête</i>)		
OR	> ANY (<i>sous-requête</i>)		
NOT	> ALL (<i>sous-requête</i>)		

Syntaxes pour les formules

```
-- échappement des apostrophes
```

```
'aujourd' 'hui'
```

```
-- écriture des dates et heures
```

```
DATE '2023-07-06'
```

```
TIME '11:59:59'
```

```
TIMESTAMP '2023-07-06 09:30'
```

```
-- concaténation
```

```
PRENOM || ' ' || NOM
```

```
CONCAT(PRENOM, ' ', NOM)
```

- **Sous MySQL, on doit forcer le *sql_mode* pour pouvoir concaténer avec ||**
- **Sous SQL Server et Sybase, on concatène avec +**

```
-- CASE, searched syntax
```

```
CASE
```

```
  WHEN Age < 18 THEN 'Enfant'
```

```
  WHEN Genre = 'F' THEN 'Femme'
```

```
  WHEN Genre = 'M' THEN 'Homme'
```

```
  ...
```

```
  ELSE 'Non-binaire'
```

```
END
```

```
-- CASE, simple syntax
```

```
CASE Mois
```

```
  WHEN 1 THEN 'Janvier'
```

```
  WHEN 2 THEN 'Février'
```

```
  ...
```

```
  ELSE 'Mois inconnu'
```

```
END
```

Quelques fonctions scalaires

LOWER('TeXTe') UPPER('TeXTe')

SUBSTR(*texte*, *début*, *longueur*)

De manière générale, les fonctions scalaires et leur syntaxe sont assez variable d'un SGBD à l'autre... RTFM !

COALESCE(*colonne*, *valeur de remplacement*)

NULLIF(*colonne*, *valeur à remplacer par NULL*)

CAST('123' AS FLOAT) - 23 ;

CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP

EXTRACT(YEAR FROM CURRENT_DATE)

DATE(CURRENT_TIMESTAMP)

STR_TO_DATE('01/05/2023 14h55 12"', '%d/%m/%Y %Hh%i %S"')

DATE_FORMAT(CURRENT_TIMESTAMP, '%d/%m/%Y %Hh%i %S"')

Agrégation et regroupement

Fonctions d'agrégation

COUNT (*) COUNT (Colonne) COUNT (DISTINCT Colonne)

MIN () MAX () SUM () AVG ()

-- agrégation par concaténation avec MySQL

GROUP_CONCAT(DISTINCT *expr* ORDER BY *expr* SEPARATOR ', ')

-- ... avec Oracle

LISTAGG(DISTINCT *expr*, ', ') WITHIN GROUP (ORDER BY *expr*)

-- ... avec SQL Server

STRING_AGG(~~DISTINCT~~ *expr*, ', ') WITHIN GROUP (ORDER BY *expr*)

-- ... avec PostgreSQL

STRING_AGG(DISTINCT *expr*, ', ' ORDER BY *expr*)

Règle du GROUP BY

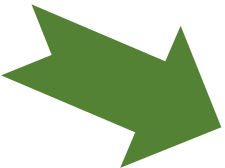
La clause GROUP BY doit obligatoirement comporter toutes les colonnes non-agrégées du SELECT.

```
SELECT cli.IDclient, cli.Nom, MIN(com.DateComm)
FROM clients cli
  INNER JOIN commandes com
    ON cli.IDclient = com.Idclient
GROUP BY cli.IDclient ;
```



Colonnes agrégée par MIN

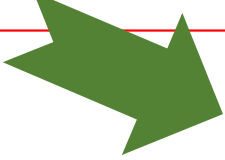
Colonnes non-agrégées



```
SELECT cli.IDclient, cli.Nom, MIN(com.DateComm)
FROM clients cli
  INNER JOIN commandes com
    ON cli.IDclient = com.Idclient
GROUP BY cli.IDclient, cli.Nom ;
```

S'il n'y a pas de GROUP BY, le SELECT ne doit comporter que des colonnes agrégées, et le résultat comportera une seule ligne.

```
SELECT IDclient, MIN(Nom) AS Premier, COUNT(*) AS Nombre
FROM clients ;
```



```
SELECT MIN(Nom) AS Premier, COUNT(*) AS Nombre
FROM clients ;
```

Totaux et sous-totaux avec ROLLUP

- La fonction ROLLUP a la particularité de s'utiliser dans la clause GROUP BY. Elle génère des lignes de total, la ou les dimensions concernées étant représentée(s) comme NULL sur ces lignes de total.

Utilisation de COALESCE pour remplacer les NULL par un libellé

```
-- CA par gamme avec total toutes gammes
SELECT COALESCE(g.NomGamme, 'TOUTES GAMMES') AS Gamme,
       COALESCE(p.NomProduit, 'SOUS-TOTAL ' || g.NomGamme) AS Produit,
       SUM(quantite * prixu) AS CA
FROM lignes_comm lc
     INNER JOIN produits p ON lc.IDproduit = p.IDproduit
     INNER JOIN gammes g ON p.IDgamme = g.IDgamme
GROUP BY g.NomGamme, p.NomProduit WITH ROLLUP
ORDER BY COALESCE(g.NomGamme, 'zzz'),
         COALESCE(p.NomProduit, 'zzz')
```

Utilisation de COALESCE pour mettre les NULL en dernier (contrairement à la plupart

ROLLUP est adaptée pour des dimensions hiérarchisées, comme produits et gammes.

Pour obtenir des sous-totaux croisés, plusieurs autres SGBD proposent WITH CUBE.

des autres SGBD, MySQL n'accepte pas les locutions NULLS FIRST et NULLS LAST.)

Produit	Gamme	CA
Sodas	Coca Zéro	1225.5
Sodas	Fanta	1585.5
Sodas	SOUS-TOTAL Sodas	5080.5
Viandes	Alice Mutton	23230.160614013672
Viandes	Mishi Kobe Niku	6305.350074768066
Viandes	Pâté chinois	13933.91974067688
Viandes	Perth Pasties	15363.579818725586
Viandes	Thüringer Rostbrat...	62667.25880432129
Viandes	Tourtière	3655.850086212158
Viandes	SOUS-TOTAL Viandes	125156.11913871765
TOUTES GAMMES	NULL	967936.063644886

Requêter
plusieurs tables

Jointures normalisées

```
FROM Clients cli
```

```
    INNER JOIN Commandes com
```

```
        ON cli.IDclient = com.IDclient
```

```
INNER JOIN -- clients avec commande seulement
```

```
LEFT JOIN -- clients avec ou sans commande
```

```
RIGHT JOIN -- commandes avec ou sans client
```

```
FULL JOIN -- tous les cas
```

***Contrairement à la plupart des SGBD,
MySQL n'accepte pas le FULL JOIN***

Hiérarchie en autojointure

EMP_ID	NAME	JOB	MGR_ID	HIREDATE
7566	JONES	MANAGER	7839	2011-04-02
7654	MARTIN	SALESMAN	7698	2011-09-28
7698	BLAKE	MANAGER	7839	2011-05-01
7782	CLARK	MANAGER	7839	2011-06-09
7788	SCOTT	ANALYST	7566	2012-12-09
7839	KING	PRESIDENT	NULL	2011-11-17
7844	TURNER	SALESMAN	7698	2011-09-08

Le manager de Martin est Blake.

*King est le président, il n'a pas de manager.
Ses subordonnés directs sont Blake et Clark.*

```
-- liste des employés avec nom de leur manager
SELECT emp.emp_id, emp.name, mgr.name AS Manager
FROM employees emp
LEFT JOIN employees mgr ON emp.MGR_ID = mgr.EMP_ID
;
```

On utilise deux fois la même table, avec deux alias différents, ce qui permet de distinguer les colonnes de chaque instance de la table.

Pour descendre ou remonter sur plusieurs niveaux, il faut utiliser un alias de la table pour chaque niveau. Il est donc nécessaire de connaître la profondeur maximale de la hiérarchie.

Une modélisation plus complexe mais plus puissante est la **représentation intervallaire**, cf <https://sqlpro.developpez.com/cours/arborescence>

Opérateurs ensemblistes

```
-- Liste des clients, qu'ils soient entreprises ou individus  
SELECT NomEntreprise AS NomClient, 'Entrep' AS TypeClient  
FROM entreprise_clientes
```

UNION ALL

```
SELECT Nom || ', ' || Prenom, 'Indiv' AS TypeClient  
FROM individus_clientes  
ORDER BY 1
```

- *Même nombre de colonnes et types compatibles*
- *ALL pour éviter un dédoublement (DISTINCT implicite)*
- *Noms ou alias de colonnes sur le premier SELECT*
- *Un seul ORDER BY, situé après le dernier SELECT*

```
-- Entreprises avec les deux rôles de client et de fournisseur  
SELECT NomEntreprise FROM entreprise_clientes
```

INTERSECT ALL

```
SELECT NomEntreprise FROM fournisseurs
```

```
-- Entreprises clientes, sauf si elles sont aussi fournisseuses  
SELECT NomEntreprise FROM entreprise_clientes
```

EXCEPT ALL

```
SELECT NomEntreprise FROM fournisseurs
```

- *Sous Oracle, EXCEPT doit être remplacé par MINUS*

Les instructions DDL

Instructions DDL liées aux tables

Création : CREATE TABLE

Modification : ALTER TABLE

Suppression : DROP TABLE

Suppression et re-création : TRUNCATE TABLE

```
CREATE TABLE Etudiants (  
    IDetudiant INT AUTO_INCREMENT PRIMARY KEY,  
    Nom VARCHAR(255) NOT NULL,  
    Prenom VARCHAR(255) NOT NULL,  
    DateNaissance DATE ) ;
```

```
ALTER TABLE Etudiants  
    ADD COLUMN Email VARCHAR(255) AFTER DateNaissance ;
```

```
TRUNCATE TABLE Etudiants ;
```

```
DROP TABLE Etudiants ;
```

Les contraintes

- Contrainte NOT NULL : la colonne doit obligatoirement être renseignée.
- Contrainte d'unicité (UNIQUE) : les doublons sont interdits dans la colonne (ou la combinaison de colonnes).
- Contrainte de clef primaire (PRIMARY KEY) : NOT NULL + UNIQUE.
- Contrainte d'intégrité référentielle (FOREIGN KEY) : la colonne est une clef étrangère et n'accepte que les valeurs existantes dans la clef primaire référencée.
- Contrainte CHECK : la ou les colonne(s) doi(ven)t respecter une condition précisée.

```
ALTER TABLE Etudiants ADD CHECK (Email LIKE '%@%') ;
```

```
CREATE TABLE Programmes (  
    IDprog INT AUTO_INCREMENT PRIMARY KEY,  
    Prog VARCHAR(255)) ;
```

```
ALTER TABLE Etudiants  
    ADD COLUMN IDprog INT AFTER Email,  
    ADD FOREIGN KEY (IDprog) REFERENCES Programmes (IDprog) ;
```

Les requêtes de
modification des
données

Instructions DML de modification

- Ajout d'une nouvelle ligne : INSERT
- Modification de lignes existantes : UPDATE
- Suppression de lignes existantes : DELETE

```
INSERT INTO Etudiants (Nom, Prenom, DateNaissance)  
VALUES ('Bond', 'James', DATE '1999-06-25');
```

```
UPDATE Etudiants  
SET Email = 'jb007.junior@efrei.net'  
WHERE Nom = 'Bond' AND Prenom = 'James' ;
```

```
UPDATE Etudiants SET Nom = UPPER(Nom) ;
```

```
DELETE  
FROM Etudiants  
WHERE DateNaissance < DATE '1980-01-01' ;
```

```
DELETE  
FROM Etudiants ;
```

Le DCL et la sécurité

Concepts utilisés par le DCL

Database

Schéma

=

Utilisateur

Privilèges

Objets

Groupe

Système

Rôle

Modèle de sécurité Oracle

Concepts utilisés par le DCL

Database

Schéma

Privilèges

Objets

Système

Utilisateur

=

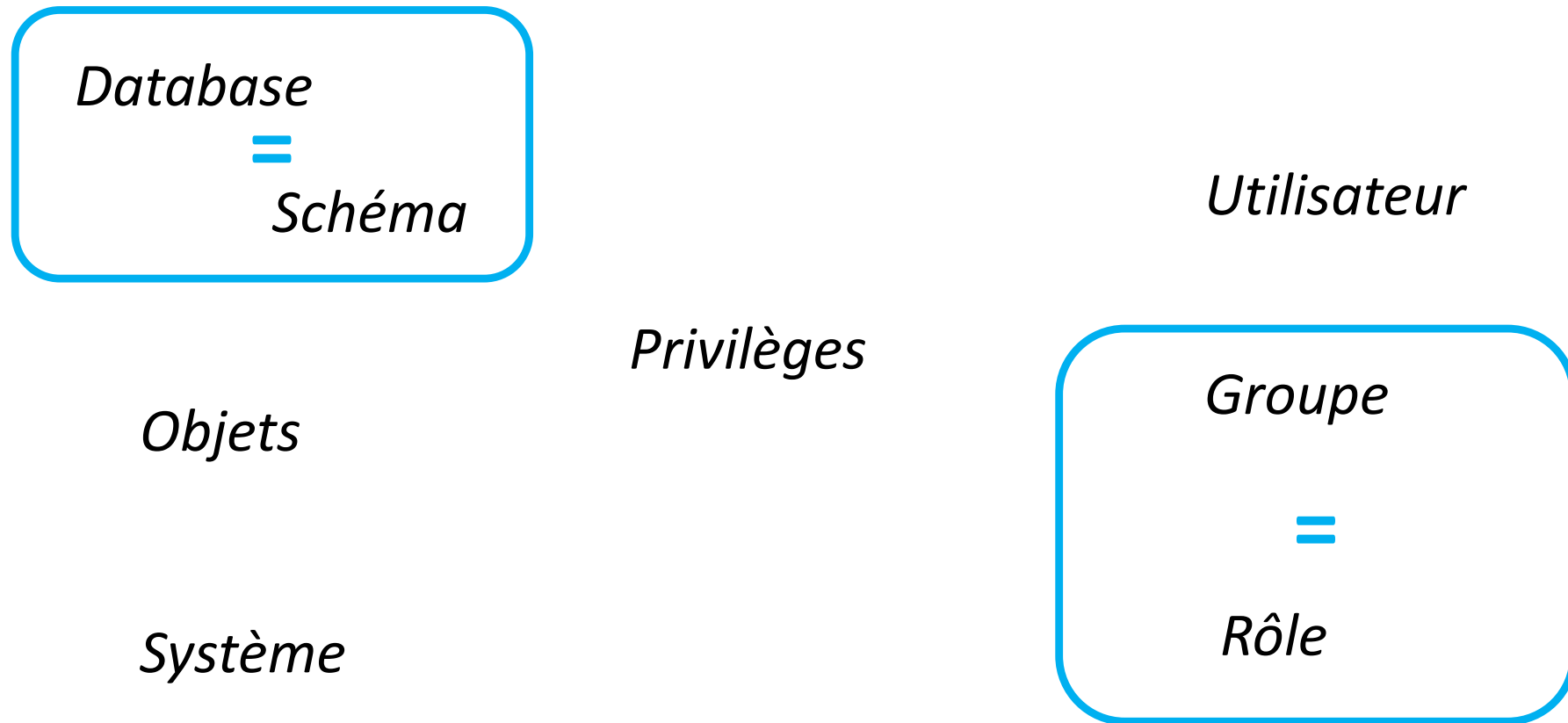
Groupe

=

Rôle

Modèle de sécurité PostgreSQL

Concepts utilisés par le DCL



Modèle de sécurité MySQL

Les instructions DCL

```
GRANT privilege ON objet TO rôle/utilisateur ;
```

```
GRANT rôle TO rôle / utilisateur ;
```

```
REVOKE privilege ON objet FROM rôle/utilisateur ;
```

```
REVOKE rôle FROM rôle/utilisateur ;
```

```
SHOW GRANTS ;
```

```
SHOW GRANTS FOR rôle/utilisateur ;
```

```
CREATE USER utilisateur IDENTIFIED BY 'motdepasse' ;
```

```
CREATE ROLE rôle ;
```

```
ALTER USER... ; DROP USER ... ;
```

Sous-requêtes

Sous-requêtes 1 : scalaires, tables

- **Sous-requêtes scalaires** (une seule ligne, une seule colonne) : utilisable n'importe où

-- dernier client par ordre d'ID, avec numéro d'ordre

```
SELECT Nom, Prenom,
```

```
    (SELECT COUNT(*) FROM clients) AS Rang_d_arrivee
```

```
FROM clients
```

```
WHERE IDclient = (SELECT MAX(IDclient) FROM clients) ;
```

- **Tables virtuelles** : à utiliser dans le FROM, comme si c'était une table

-- idem, en utilisant une table virtuelle

```
SELECT nb AS Rang_d_arrivee, Nom, Prenom
```

```
FROM clients c
```

```
    INNER JOIN (
```

```
        SELECT COUNT(*) as nb, MAX(IDclient) as maxid
```

```
        FROM clients
```

```
) subq
```

```
    ON c.IDclient = subq.maxid
```

```
;
```

Alias de table obligatoire

Sous-requêtes 2 : listes

- **Sous-requêtes liste** (une seule colonne) : introduites par IN...

```
-- clients ayant commandé le matin (sans doublons)
SELECT Nom, Prenom
FROM clients
WHERE IDclient IN (SELECT Idclient
    FROM commandes
    WHERE IDtrancheHoraire = 1) ;
```

... ou par un opérateur de comparaison suivi de ANY ou ALL

```
-- dernier client par ordre d'ID
SELECT Nom, Prenom
FROM clients
WHERE IDclient >= ALL (SELECT IDclient FROM clients) ;
```

```
-- idem avec ANY
SELECT Nom, Prenom
FROM clients
WHERE NOT IDclient < ANY (SELECT IDclient FROM clients) ;
```

Sous-requêtes 3 : corrélation

- Les sous-requêtes de tous les types peuvent être **corrélées**, c'est-à-dire comporter une condition relative aux données de la requête principale

```
-- dernière commande de chaque client,  
-- avec une sous-requête scalaire corrélée  
SELECT IDclient, IDcommande, DateComm  
FROM commandes c  
WHERE DateComm = (  
    SELECT MAX(DateComm)  
    FROM commandes subc  
    WHERE subc.IDclient = c.IDclient  
)  
ORDER BY IDclient, IDcommande ;
```

*Alias de la table de la
requête principale*

*Alias de la table de la
sous-requête*

Corrélation


La corrélation présente un risque de ralentissement de la requête ; en cas de mauvaise performance, il est souvent souhaitable de réécrire la requête en évitant la corrélation (ici par exemple, on pourrait une utiliser une table virtuelle et remplacer la corrélation par une jointure).

La clause WITH et les CTE

La clause WITH permet de simplifier l'écriture des tables virtuelles : on commence par associer un alias à une sous-requête (ou plusieurs alias à plusieurs sous-requêtes), et on peut ensuite utiliser ces alias comme des tables.

Ces sous-requêtes sont appelées *Common Table Expressions* (CTE).

```
-- Durée d'ancienneté des clients, avec une CTE
WITH Com1 AS (
    SELECT IDclient, MIN(DateComm) AS DateAncnte
    FROM commandes
    GROUP BY Idclient
)
SELECT Nom, Prenom,
    DATEDIFF(CURRENT_DATE, Com1.DateAncnte) as DureeAncnte
FROM clients c
    INNER JOIN Com1 ON c.IDclient = Com1.Idclient
;
```



Ainsi utilisé, WITH n'est que du *syntax sugar*, c'est-à-dire qu'il simplifie la syntaxe mais n'apporte pas de fonctionnalité nouvelle.

Les CTE récurives

- La nouvelle fonctionnalité est la possibilité de faire une CTE réursive, c'est-à-dire qu'elle s'unione et se jointure avec elle-même.
- La principale utilisation est l'exploitation d'une hiérarchie en autojointure sans connaître la profondeur maximale.
- **MySQL impose l'utilisation du mot-clef RECURSIVE et la liste des colonnes, ce qui n'est pas standard.**

```
WITH RECURSIVE hie (Emp_Id, Nom, Niveau, Supérieurs) AS (  
    SELECT Emp_Id, Name as Nom, 1 AS Niveau,  
           CAST(NULL AS CHAR(200)) as Supérieurs  
    FROM employees  
    WHERE Mgr_ID IS NULL  
    UNION ALL  
    SELECT E2.Emp_Id, E2.Name, hie.Niveau + 1,  
           hie.Nom || CASE  
               WHEN hie.Supérieurs IS NULL THEN ''  
               ELSE ' > ' || hie.Supérieurs END  
    FROM employees E2  
    INNER JOIN hie ON E2.Mgr_ID = hie.Emp_Id  
)  
SELECT * FROM hie  
ORDER BY Niveau, Nom ;
```

*Début de la récursion
(pas de référence à la CTE)*

Appel récursif

Requête principale

Nom	Niveau	Supérieurs
KING	1	NULL
BLAKE	2	KING
CLARK	2	KING
JONES	2	KING
ALLEN	3	BLAKE > KING
FORD	3	JONES > KING
JAMES	3	BLAKE > KING
MARTIN	3	BLAKE > KING
MILLER	3	CLARK > KING
SCOTT	3	JONES > KING
TURNER	3	BLAKE > KING
WARD	3	BLAKE > KING
ADAMS	4	SCOTT > JONES > KING
SMITH	4	FORD > JONES > KING

SQL Procédural

Types de routines

- **Fonction stockée** (*stored function*) : permet de créer une nouvelle fonction, utilisable comme une fonction standard.
- **Procédure stockée** (*stored procedure*) : exécute une série d'instructions, qui peuvent être du SQL procédural, du SQL déclaratif et/ou du SQL dynamique. Si elle comporte un SELECT, renvoie un jeu de données.
- **Déclencheur** (*trigger*) : procédure qui se déclenchera automatiquement à chaque événement INSERT, DELETE ou UPDATE d'une table.
 - Certains SGBD permettent aussi de poser des triggers sur des événements DDL comme ALTER ou DROP
 - Le trigger peut s'exécuter avant ou après l'événement (BEFORE, AFTER) ; certains SGBD acceptent aussi le INSTEAD OF (à la place de).
 - On distingue les triggers de ligne (FOR EACH ROW) et les triggers ensemblistes ou de requête (FOR STATEMENT).
- Le **SQL dynamique** est une syntaxe pour exécuter le contenu d'une variable VARCHAR comme une requête SQL, permettant ainsi de variabiliser des noms d'objets de base de données.

Même si ces quatre fonctionnalités existent dans la plupart des SGBD, les syntaxes et les possibilités exactes sont très spécifiques à chacun.

Création d'une fonction

```
-- syntaxe mono-instruction
DROP FUNCTION IF EXISTS AireCercle ;
CREATE FUNCTION AireCercle(Rayon FLOAT)
    RETURNS FLOAT
    DETERMINISTIC
    RETURN PI() * Rayon ^ 2 ;
```

Suppression d'une éventuelle version antérieure

Changement de délimiteur pour pouvoir envoyer une instruction multiple.

```
-- syntaxe multi-instructions avec BEGIN... END
DROP FUNCTION IF EXISTS Aire ;
DELIMITER //
CREATE FUNCTION Aire(Forme VARCHAR(20), Dim1 FLOAT, Dim2 FLOAT)
    RETURNS FLOAT
    DETERMINISTIC
BEGIN
    DECLARE Resultat FLOAT ;
    CASE UPPER(Forme)
        WHEN 'CERCLE' THEN SET Resultat := PI() * Dim1 * Dim1 ;
        WHEN 'CARRE' THEN SET Resultat := Dim1 * Dim1 ;
        WHEN 'RECTANGLE' THEN SET Resultat := Dim1 * Dim2 ;
        WHEN 'TRIANGLE' THEN SET Resultat := Dim1 * Dim2 / 2 ;
    ELSE SET Resultat := NULL ;
    END CASE ;
RETURN Resultat ;
END //
DELIMITER ;
```

Déclaration de variable

Attention, syntaxe différente du CASE... END déclaratif !

Rétablissement du délimiteur.

Création d'une procédure stockée + SQL dynamique

```
-- procédure mono-instruction                                -- exécution de la procédure
CREATE PROCEDURE Salut()                                       CALL Salut ;
SELECT 'Bonjour ' || CURRENT_USER || ' !' ;

-- procédure pour créer un compte et sa base de données personnelle
DROP PROCEDURE IF EXISTS CreerCompteAvecBdd ;
DELIMITER //
CREATE PROCEDURE CreerCompteAvecBdd(Compte VARCHAR(20), Passe VARCHAR(20))
BEGIN
    SET @sql := 'CREATE USER ' || Compte || ' IDENTIFIED BY ''' || Passe || ''' ; ' ;
    PREPARE dynamic_statement FROM @sql;
    EXECUTE dynamic_statement;
    SET @sql := 'CREATE DATABASE db' || Compte || ' ; ' ;
    PREPARE dynamic_statement FROM @sql;
    EXECUTE dynamic_statement;
    SET @sql := 'GRANT ALL ON db' || Compte || '.* TO ' || Compte || ' ; ' ;
    PREPARE dynamic_statement FROM @sql;
    EXECUTE dynamic_statement;
    DEALLOCATE PREPARE dynamic_statement;
END //
DELIMITER ;

-- exécution de la procédure
CALL CreerCompteAvecBdd('alibaba', 'sésame') ;
```

*Syntaxe du SQL dynamique,
afin de variabiliser les noms
d'objets de base de données.*

Création d'un trigger 1

*Exemple de création de triggers
pour tracer les événements de
la table **Clients**.*

```
-- Table de traces
CREATE TABLE Histo_Clients (
    Createur VARCHAR(80),
    Horodatage DATETIME,
    Evenement VARCHAR(10),
    IdClient INT,
    Info VARCHAR(255)
) ;
```

```
DROP TRIGGER IF EXISTS Clients_AI ;
-- trigger pour tracer l'insertion de nouveaux clients
CREATE TRIGGER Clients_AI AFTER INSERT ON Clients
FOR EACH ROW
    INSERT INTO Histo_Clients (Createur, Horodatage, Evenement, IdClient)
    VALUES (CURRENT_USER, CURRENT_TIMESTAMP, 'INSERT', NEW.IdClient) ;
;
```

```
DROP TRIGGER IF EXISTS Clients_AD ;
-- trigger pour tracer la suppression de clients
CREATE TRIGGER Clients_AD AFTER DELETE ON Clients
FOR EACH ROW
    INSERT INTO Histo_Clients (Createur, Horodatage, Evenement, IdClient)
    VALUES (CURRENT_USER, CURRENT_TIMESTAMP, 'DELETE', OLD.IdClient) ;
;
```

Création d'un trigger 2

```
DROP TRIGGER IF EXISTS Clients_AU ;
DELIMITER //
CREATE TRIGGER Clients_AU AFTER UPDATE ON Clients
FOR EACH ROW
BEGIN
    DECLARE Txt VARCHAR(255) DEFAULT '' ;
    IF NEW.Nom != OLD.Nom
        THEN SET Txt := TXT || OLD.Nom || ' => ' || NEW.Nom || ', ' ;
    END IF ;
    IF NEW.Prenom != OLD.Prenom
        THEN SET Txt := TXT || OLD.Prenom || ' => ' || NEW.Prenom || ', ' ;
    END IF ;
    IF NEW.Profession != OLD.Profession
        THEN SET Txt := TXT || OLD.Profession || ' => ' || NEW.Profession || ', ' ;
    END IF ;
    IF Txt = ''
        THEN SET Txt := 'Modification non tracée' ;
    ELSE
        SET Txt = SUBSTRING(Txt FROM 1 FOR LENGTH(Txt) - 2) ;
    END IF ;
    INSERT INTO Clients_Histo (Createur, Horodatage, Evenement, IdClient, Info)
    VALUES (CURRENT_USER, CURRENT_TIMESTAMP, 'UPDATE', NEW.IdClient, Txt) ;
END //
DELIMITER ;
```

Fonctions analytiques
(window functions)

Modifier le périmètre d'une agrégation : la clause OVER

- La clause OVER peut s'appliquer à une fonction d'agrégation pour lui indiquer de travailler sur les lignes du résultat de la requête
- Elle reste ainsi limitée par le périmètre de la clause WHERE

```
-- répartition du CA à l'intérieur d'une commande
select V.idcomm, P.nomproduit, G.nomgamme, ca,
sum(ca) over () as total_commande,
       ca / sum(ca) over () * 100 pourcent_ligne
from dwh_facts V
inner join produits P on V.IDproduit = P.IDproduit
inner join gammes G on P.IDgamme = G.IDgamme
where V.idcomm = 11000
;
```

	nomproduit character varying (255)	nomgamme character varying (255)	ca numeric (8,2)	total_commande numeric	pourcent_ligne numeric
1	Chef Anton's Cajun Seasoning	Condiments	392.75	767.75	51.155975252
2	Guaraná Fantástica	Sirops	96.30	767.75	12.543145555
3	Original Frankfurter grüne Soße	Condiments	278.70	767.75	36.300879192

- OVER accepte deux options, PARTITION BY et ORDER BY...

Agrégat selon des caractéristiques communes : OVER (PARTITION BY)

- L'option PARTITION BY indique que l'agrégat doit s'intéresser uniquement aux lignes ayant des valeurs en commun avec la ligne courante.

```
select P.nomproduit, G.nomgamme, ca,  
       sum(ca) over (partition by G.IDgamme) as  
ca_gamme,  
       sum(ca) over () as total_commande  
from dwh_facts V  
inner join produits P on V.IDproduit = P.IDproduit  
inner join gammes G on P.IDgamme = G.IDgamme  
where V.idcomm = 11000  
;
```

	nomproduit character varying (255)	nomgamme character varying (255)	ca numeric (8,2)	ca_gamme numeric	total_commande numeric
1	Guaraná Fantástica	Sirops	96.30	96.30	767.75
2	Chef Anton's Cajun Seasoning	Condiments	392.75	671.45	767.75
3	Original Frankfurter grüne Soße	Condiments	278.70	671.45	767.75

OVER et GROUP BY : le problème

- Il est difficile de combiner des agrégats simples et des agrégats avec OVER. En effet, OVER n'accepte le GROUP BY que s'il contient toutes les colonnes sans OVER.

Voici par exemple le CA par gamme en 2021.

```
select G.nomgamme, sum(ca) as ca_gamme
from dml_facts V
inner join produits P on V.IDproduit = P.IDproduit
inner join gammes G on P.IDgamme = G.IDgamme
where extract(year from V.DateComm) = 2021
group by G.nomgamme
order by G.nomgamme
;
```

	nomgamme character varying (255)	ca_gamme numeric
1	Alcools	54243.48
2	Condiments	33019.57
3	Desserts	50058.80
4	Pâtes et céréales	31121.25
5	Poissons et fruits de mer	48194.56
6	Produits laitiers	116029.12
7	Produits secs	36708.02
8	Sirops	14533.15
9	Sodas	132540.04
10	Vianades	59708.67

Il pourrait sembler facile d'utiliser OVER () pour avoir le CA total de l'année, puis de s'en servir pour calculer la part représentée par chaque gamme.

```
select G.nomgamme, sum(ca) as ca_gamme, sum(ca) over () as total_ca
from dml_facts V
inner join produits P on V.IDproduit = P.IDproduit
inner join gammes G on P.IDgamme = G.IDgamme
where extract(year from V.DateComm) = 2021
group by G.nomgamme
order by G.nomgamme
;
```

Data Output	Explain	Messages	Notifications
ERROR: column "v.ca" must appear in the GROUP BY clause or be used in an aggregate function LINE 1: select G.nomgamme, sum(ca) as ca_gamme, sum(ca) over () as t... ^			
SQL state: 42803 Character: 45			

OVER et GROUP BY : quelques solutions

- Isoler le GROUP BY dans une table virtuelle ou une CTE :

```
select nomgamme, ca_gamme, sum(ca_gamme) over () as total_ca
from (
  select G.nomgamme, sum(ca) as ca_gamme
  from dml_facts V
  inner join produits P on V.IDproduit = P.IDproduit
  inner join gammes G on P.IDgamme = G.IDgamme
  where extract(year from V.DateComm) = 2021
  group by G.nomgamme
) sous_totaux ;
```

```
with sous_totaux as (
  select G.nomgamme, sum(ca) as ca_gamme
  from dml_facts V
  inner join produits P on V.IDproduit = P.IDproduit
  inner join gammes G on P.IDgamme = G.IDgamme
  where extract(year from V.DateComm) = 2021
  group by G.nomgamme
)
select nomgamme, ca_gamme, sum(ca_gamme) over () as total_ca
from sous_totaux ;
```

	nomgamme character varying (255)	ca_gamme numeric	total_ca numeric
1	Alcools	54243.48	576156.66
2	Condiments	33019.57	576156.66
3	Desserts	50058.80	576156.66
4	Pâtes et céréales	31121.25	576156.66
5	Poissons et fruits de mer	48194.56	576156.66
6	Produits laitiers	116029.12	576156.66
7	Produits secs	36708.02	576156.66
8	Sirops	14533.15	576156.66
9	Sodas	132540.04	576156.66
10	Viandes	59708.67	576156.66

- utiliser OVER sur tous les agrégats, et remplacer GROUP BY par DISTINCT:

```
select distinct G.nomgamme, sum(ca) over (partition by G.IDgamme) as ca_gamme, sum(ca) over () as total_ca
from dml_facts V
inner join produits P on V.IDproduit = P.IDproduit
inner join gammes G on P.IDgamme = G.IDgamme
where extract(year from V.DateComm) = 2021 ;
```

Cumul progressif avec OVER (ORDER BY)

- L'option ORDER BY de la clause OVER permet de faire un cumul progressif (*running aggregate*) :

```
select distinct moisnum,  
               sum(ca) over (partition by moisnum),  
               sum(ca) over (order by moisnum) as cumul_ca  
from dml_facts V  
  inner join temps T on V.DateComm = T.date  
where annee = 2021  
;
```

	moisnum integer	sum numeric	cumul_ca numeric
1	1	22248.33	22248.33
2	2	13659.20	35907.53
3	3	34117.45	70024.98
4	4	46495.81	116520.79
5	5	54901.38	171422.17
6	6	34126.21	205548.38
7	7	62106.30	267654.68
8	8	118288.37	385943.05
9	9	18366.04	404309.09
10	10	41617.92	445927.01
11	11	113847.48	559774.49
12	12	16382.17	576156.66

ORDER BY est bien sûr compatible avec PARTITION BY :

```
-- CA mensuel cumulé, avec remise à zéro chaque année  
select distinct annee, moisnum,  
               sum(ca) over (partition by annee, moisnum),  
               sum(ca) over (partition by annee order by moisnum) as cumul_ca  
from dml_facts V  
  inner join temps T on V.DateComm = T.date  
order by annee, moisnum  
;
```

	annee integer	moisnum integer	ca_mensuel numeric	cumul_ca numeric
8	2020	9	8661.02	102727.52
9	2020	10	16460.48	119188.00
10	2020	11	20893.58	140081.58
11	2020	12	52419.49	192501.07
12	2021	1	22248.33	22248.33
13	2021	2	13659.20	35907.53
14	2021	3	34117.45	70024.98