

Mon SQL

MySQL

Pré-requis

- Bases MySQL
- Connaître les opérations de MySQL (CRUD)

Installations

- WAMP
- PHPStorm
- Data Grip (IntelliJ)
- Git Bash (pour windows)

Intro

Qu'est qu'une base de données

- Une collection de fichiers qui contiennent des données logiquement liés entre elles
- Ces fichiers sont enregistrer dans une espace mémoire (pouvant être ddur, mémoire vive...)
- Ces fichiers sont organisés de manière à pouvoir créer, récupérer et mettre à jour les données rapidement
- Un fichier excel et une base de données ne sont pas la même chose (bien que je l'utiliserai plus tard pour schématiser un tableau d'une base de données)

Pourquoi utiliser une BDD








- Réduire la redondance et les erreurs
- Eviter que chaque utilisateur enregistre des informations dans des endroits différents sans les tenir à jour
- Permet la mise à jour d'informations centralisé

A quoi ressemble une BDD










Représentation logique:
Un silo

Design logique:
Entity-Relationship-diagram

user	
 id	int(11)
 full_name	varchar(255)
 first_name	varchar(255)
 name	varchar(255)
 gender	varchar(15)
 email	varchar(255)
 phone	varchar(255)

Un tableau de BDD

 id	 full_name	 first_name	 name	 gender	 email	 phone
1	Jonathan Hunter	Jonathan	Hunter	male	jHunter@example.com	7624148697
2	Marcus Taylor	Marcus	Taylor	male	mTaylor@example.com	7624346622
3	James Gill	James	Gill	male	jGill@example.com	7584693847
4	Cole Barnett	Cole	Barnett	male	cBarnett@example.com	7599748596
5	Lucy Palmer	Lucy	Palmer	female	lPalmer@example.com	7624917534
6	Samuel Alvarez	Samuel	Alvarez	male	sAlvarez@example.com	7527378790

Comment sont enregistrer les données

- Les données sont enregistrés dans des fichiers
- Ces fichiers peuvent être sur votre DDUR, un DDUR distant, sur plusieurs DDUR ou sur plusieurs DDUR dans différents endroits
- Comment sont enregistrer les données ne sont pas vraiment pertinent pour nous, le DBMS (Database Management System) gère la complexité pour nous
- C'est pas tout à fait vrai, car nous aurons besoins de définir quels DBMS est plus pertinent pour notre cas d'usage et ou on veut enregistrer nos données.

Que fait un Admin BDD

- Conception logique et physique de la BDD
- Décide du ou des DBMS, logiciel de conception et méthodes de récupération des données
- Conçois, implémente et sécurise les données et roles
- Maintenance
- Optimisation et performance
- Évaluation permanente du système pour déterminer des upgrades ou mises à jour
- Est garant des données

SQL est un langage déclarative

- Bien qu'il contiens des éléments procédural, le SQL est déclarative
- On ne lui dit pas comment faire mais juste de faire

Quelques BDD

- Oracle => Très robuste, réactive, tolérant aux erreurs, scalabilité, très cher
- Microsoft SQL Server => Intégration naturel avec windows et AD
- MySQL => Le RDBMS open-source le plus utilisé
- PostgreSQL => Open-source, fonctionnalités avancés, pas de bon support
- SQLite => Open-source, portable, n'est pas scalable
- Microsoft Access => A été très utilisé dans le passé pour les applications fait sous Excel/VB, quelques site l'utilise encore

Histoire

- Première version en 1995 par MySQL AB
- Est nommé après le nom de la fille, My, du créateur Michael Widenius
- MySQL AB est racheté par Sun Microsystems en 2008
- Oracle rachète Sun Microsystems en 2009 devenant le détenteur des 2 bases de données les plus utilisés dans le monde
- A l'annonce de ce rachat, Michael Widenius décide de faire un fork de MySQL afin d'assurer le développement en tant que logiciel libre et appelant le nouveau RDBMS MariaDB (du nom de sa deuxième fille Maria)

NOTE

- On utilisera dans ce cours le PowerShell de windows pour se connecter en CLI

Rentrons dans le vif du sujet

Bases

- Show databases
- Create / drop database
- Create / drop table
- Show tables
- Describe `tablename`
- Insert / Update / Delete / Select

Databases

```
SHOW DATABASES;  
CREATE DATABASE `toto`;  
SHOW DATABASES;
```

```
DROP DATABASE `toto`;  
SHOW DATABASES;
```

```
CREATE DATABASE `toto`;  
SHOW DATABASES;
```


Tables

- Avant toute opération dans une BDD (sauf pour le show/create/drop databases) il faut indiquer à MySQL quelle DB on utilise avec la commande « use »

```
USE DATABASE `toto`;
```

```
CREATE TABLE `user` (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  email VARCHAR(255),  
  password VARCHAR(255)  
) ENGINE = InnoDB
```

```
DROP TABLE `user`;
```

```
SHOW TABLES;
```

```
DESCRIBE `user`;
```

INSERT / UPDATE / DELETE / SELECT

```
INSERT INTO `user` (`email`, `password`) values ('khalid.sookia@iknsa.com', 'paris');  
SELECT * FROM `user`;
```

```
UPDATE `user` SET `password`='123' WHERE `id`=1;  
SELECT * FROM `user`;
```

```
DELETE FROM `user` WHERE id=1;  
SELECT * FROM `user`;
```

TP

- Depuis votre PowerShell, créer une base de données 'toto' avec 2 tableaux avec les données tels qu'ils sont présenter:

user

id	email	password
2	khalid.sookia@iknsa.com	paris
3	moussa.camara@iknsa.com	Paris
4	moustakime.kifia@iknsa.com	mkifia
5	virgil.aonicesei@iknsa.com	virgil
6	jean.domche@iknsa.com	jean

employee_manager

employee	manager
3	2
4	2
5	4
6	4

TP

```
CREATE DATABASE `toto`;
USE toto;
CREATE TABLE `user` (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255),
  password VARCHAR(255)
) ENGINE=InnoDB;
INSERT INTO `user` (id, email, password) VALUES
(2, 'khalid.sookia@iknsa.com', 'paris'),
(3, 'moussa.camara@iknsa.com', 'Paris'),
(4, 'moustakime.kifia@iknsa.com', 'mkifia'),
(5, 'virgil.aonicesei@iknsa.com', 'virgil'),
(6, 'jean.domche@iknsa.com', 'jean');

CREATE TABLE employee_manager (
  employee INT,
  manager INT,
  FOREIGN KEY (employee) REFERENCES user(id),
  FOREIGN KEY (manager) REFERENCES user(id)
) ENGINE = InnoDB;

INSERT INTO employee_manager (employee, manager) VALUES
(3, 2), (4, 2), (5, 4), (6, 4);
```

TP

- Foreign Key ?? References ?

TP

- Depuis votre PowerShell

user

id	email	password
2	khalid.sookia@iknsa.com	paris
3	moussa.camara@iknsa.com	Paris
4	moustakime.kifia@iknsa.com	mkifia
5	virgil.aonicesei@iknsa.com	virgil
6	jean.domche@iknsa.com	jean

employee_manager

employee	manager
3	2
4	2
5	4
6	4

- Rédiger une requête pour récupérer les employés dont le manager a l'email 'khalid.sookia@iknsa.com'

TP

```
SELECT * FROM toto.user JOIN toto.employee_manager ON user.id = employee_manager.employee WHERE manager=  
(SELECT id FROM toto.user WHERE email='khalid.sookia@iknsa.com');
```

TP

- On delete cascade

```
CREATE TABLE employee_manager (  
  employee INT,  
  manager INT,  
  FOREIGN KEY (employee) REFERENCES user(id) ON DELETE CASCADE,  
  FOREIGN KEY (manager) REFERENCES user(id) ON DELETE CASCADE  
) ENGINE = InnoDB;
```

```
INSERT INTO employee_manager (employee, manager) VALUES  
(3, 2),  
(4, 2),  
(5, 4),  
(6, 4);
```


Mon SQL Avancée

Importons une base de données

Pour la suite du cours, nous allons utiliser DataGrip.

Téléchargez le fichier de base de donnée ici:

~~Ensuite importez le avec la ligne de commande suivante:~~

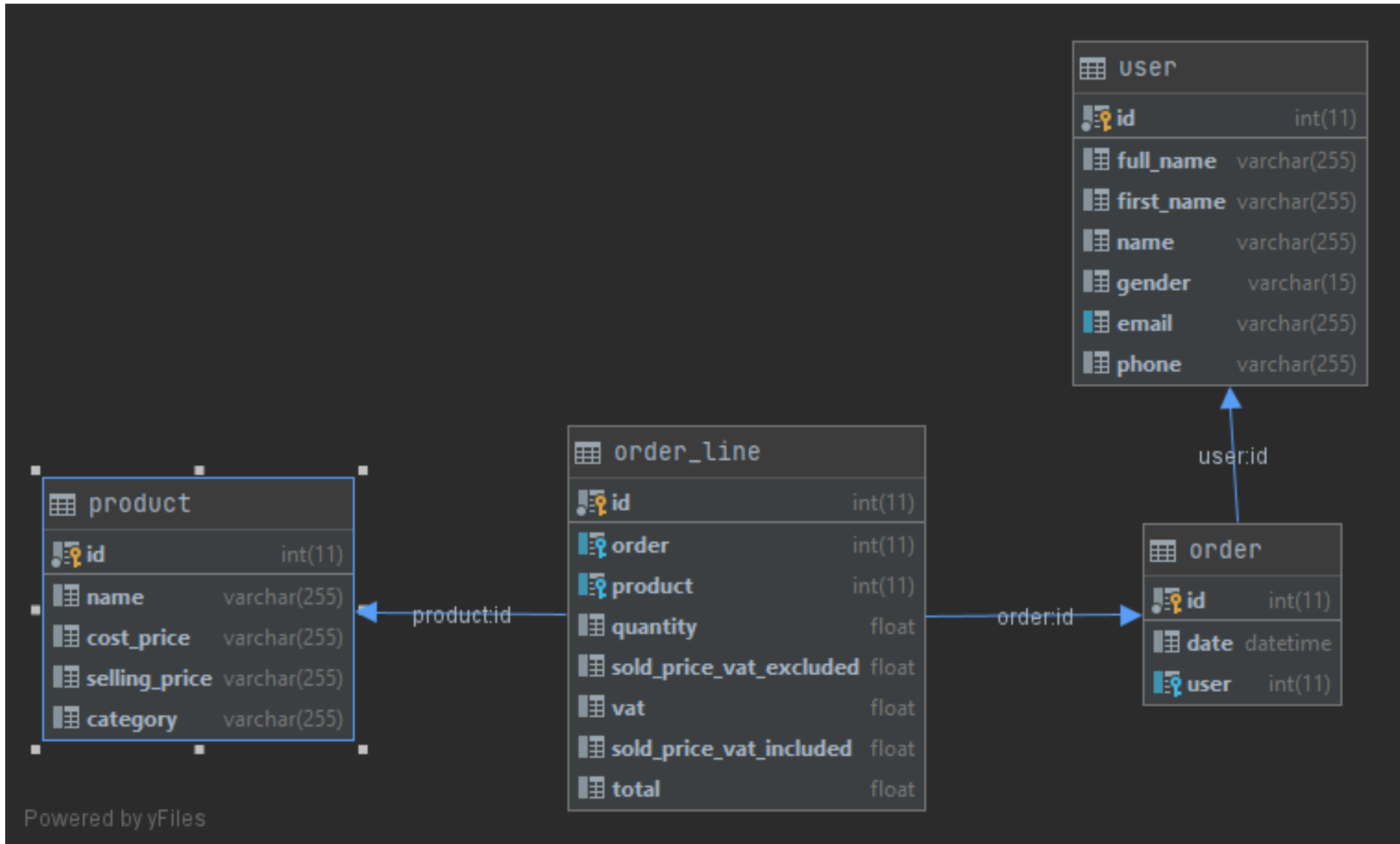
Notre mission

- Assurer la maintenance évolutive d'une partie de la base de données de l'entreprise my-ecommerce

Notre mission

- Comme vous le savez un site e-commerce est g rer   partir de plusieurs bases de donn es, plusieurs tableaux, etc...
- Nous allons intervenir sur une partie de l'infrastructure qui g re les tableaux suivants:
 - User
 - Product
 - Order
 - OrderLine

Schéma de notre parc



Etats des lieux

- Commençons par voir un peu notre BDD avec quelques select
- Voyons maintenant la quantité de données dans les différents tableaux avec l'agrégateur « count »

Demande 1

- Vérifier que pour chaque commande passé, nous avons bien un utilisateur en BDD

Demande 1

- Admettons maintenant que je veuille sélectionner seulement les colonnes date et fullname de ces 2 tableaux

```
SELECT date, full_name FROM `order`, `user`;
```

- Dans ce cas précis, il n'y a qu'une colonne date et full_name. Si par exemple nous avons une colonne date dans user, nous n'aurions pas eu le résultats attendu car MySQL ne saurait pas quelle colonne date on veut afficher.

Demande 1

- De manière générale, quand nous avons plusieurs tableaux, nous allons utiliser des alias.

```
SELECT o.`date`, u.`full_name` FROM `order` o, `user` u;
```

Demande 1

- Comme vous pouvez le voir on récupère les 500 premiers enregistrements.
- Cette limitation est celle de DataGrip, si vous lancez la commande dans votre terminal, vous allez récupérer tous les enregistrements.

```
SELECT o.`date`, u.`full_name` FROM `order` o, `user` u;
```

- Et ce n'est pas forcément pratique. On pourrait donc limiter le nombre de output

Demande 1

```
SELECT o.`date`, u.`full_name` FROM `order` o, `user` u LIMIT 10;
```

- Limit fonctionne avec un début et nombre d'enregistrement à afficher

Demande 1

- Pour bien le visualiser, rajoutons l'id de l'utilisateur

```
SELECT u.id, o.`date`, u.`full_name` FROM `order` o, `user` u LIMIT 5, 10;
```

Demande 1

- Et on pourrait même le combiner avec une clause where

```
SELECT u.id, o.`date`, u.`full_name` FROM `order` o, `user` u WHERE u.`id` > 100 LIMIT 5, 10;
```

TP

- Rédiger une requête pour sélectionner (id, full_name, email) de 100 femmes dans notre bdd à partir du 100^{ème}.

Fonctions agrégateur

Demande 1

- Une des fonctions agrégateur les plus utilisé est le count(). Elle retourne le nombre d'enregistrement

```
select count(*) from `order`;# affiche 103280  
select count(*) from `user`; # affiche 4149
```


Demande 1

- Revenons à notre demande initiale:
Vérifier que pour chaque commande passé, nous avons bien un utilisateur en BDD
- Il faut donc que pour chaque 'user' dans 'order' on ait bien le user correspondant

```
SELECT COUNT(*) FROM `order`, `user`; # affiche 428 508 720
```

- On aurait pu faire quelque chose comme ça

Demande 1

- Si on fait une clause where qui match l'id user avec order.user on devrait avoir le même nombre d'enregistrement que dans order.

```
SELECT COUNT(*) FROM `order`, `user` WHERE `order`.`user` = `user`.id; # affiche 103 280 qui est le nombre de 'order'
```

Bien, on peut dire qu'on a pour chaque order un user en bdd

Demande 1

```
SELECT COUNT(*) FROM `order`, `user` WHERE `order`.`user` = `user`.id; # affiche 103 280 qui est le nombre de 'order'
```

Bien que cette solution soit correcte avec le résultat attendu, ce n'est pas ce qu'on ferait réellement

Demande 1

- On ferait plutôt quelque chose comme ça

```
SELECT COUNT(*) FROM `order` JOIN `user` ON `order`.`user` = `user`.id; # affiche 103 280 qui est le nombre de 'order'
```

La raison est une question de performance. Dans le premier cas avec le WHERE, MySQL créer un tableau intermédiaire pour mettre tout les enregistrement et ensuite filtre le contenu en fonction de la clause WHERE et dans le 2^{ème} cas, il commence par faire la relation entre les deux.

Demande 1

- Il y a plusieurs types de jointures. Nous venons d'utiliser un 'inner join' même si je ne l'ai pas explicitement mis.

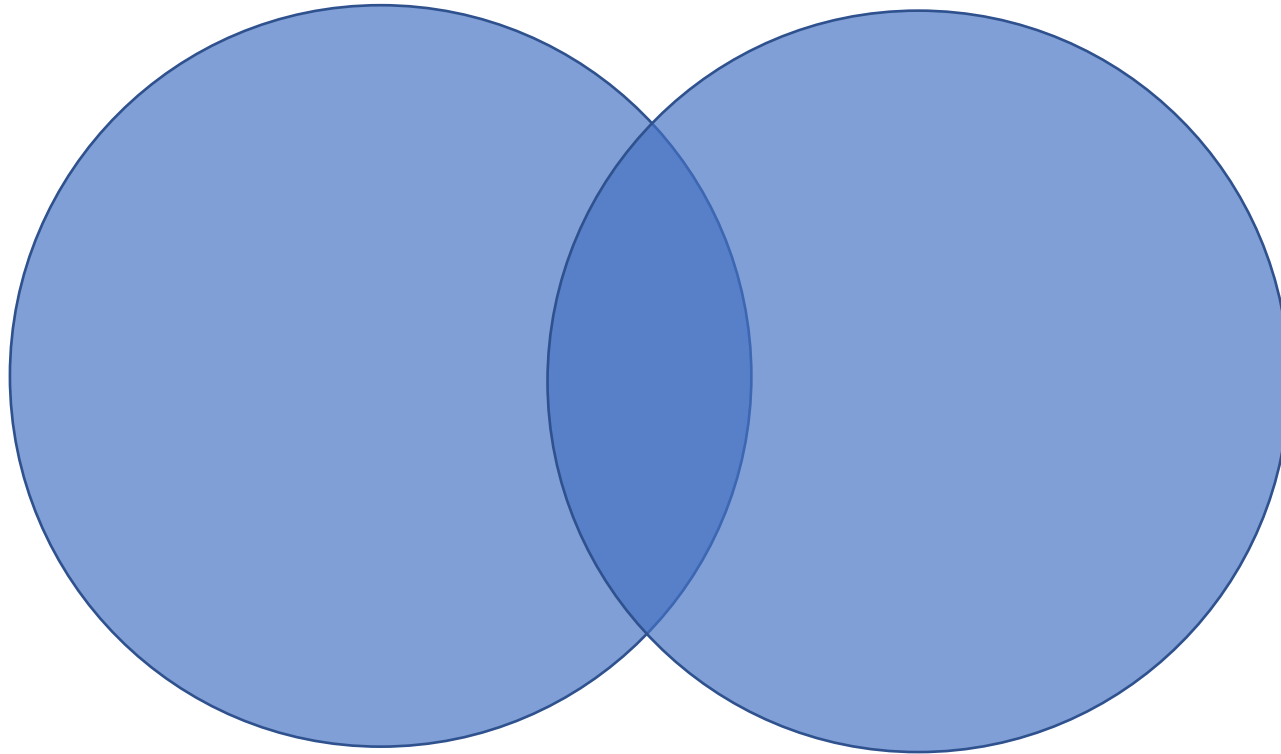
```
SELECT COUNT(*) FROM `order` JOIN `user` ON `order`.`user` = `user`.id; # affiche 103 280 qui est le nombre de 'order'
```

Demande 1

- Il y a plusieurs types de jointures. Nous venons d'utiliser un 'inner join' même si je ne l'ai pas explicitement mis.
- Les jointures les plus utilisés sont :
 - Inner
 - Left
 - Right

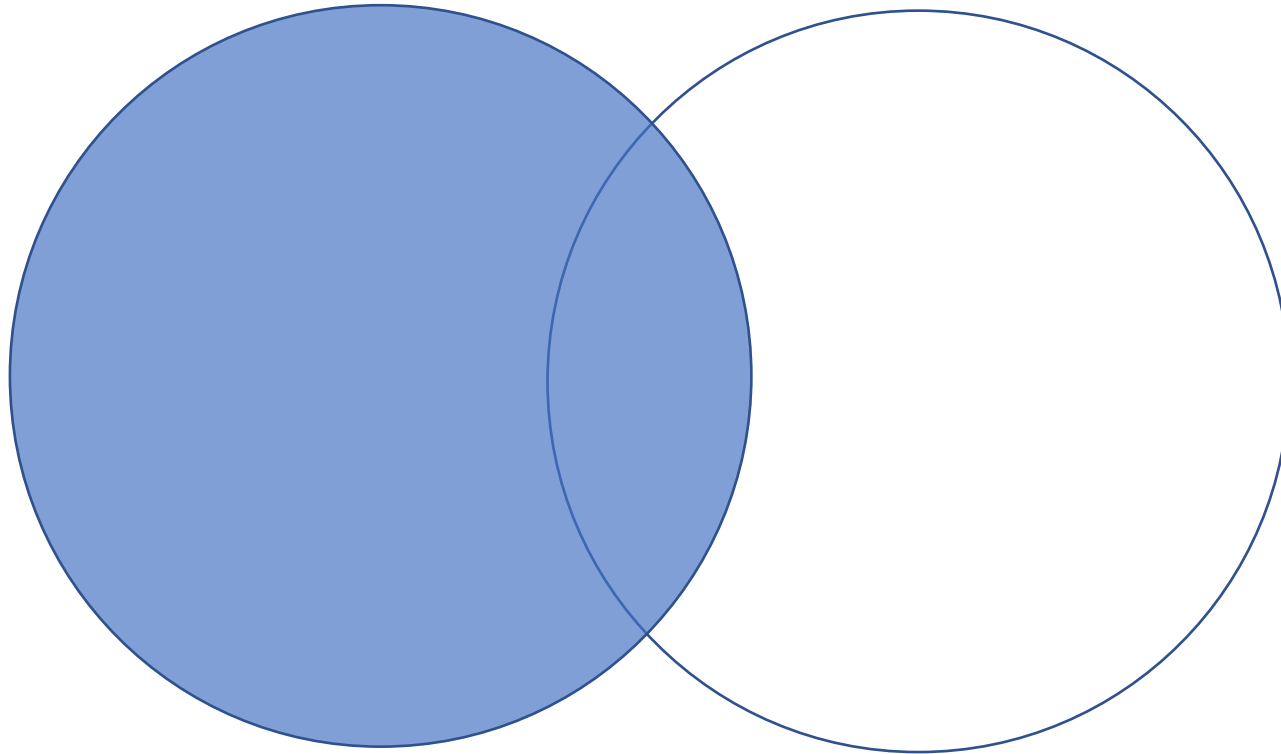
Inner Join

Intersection des tableaux



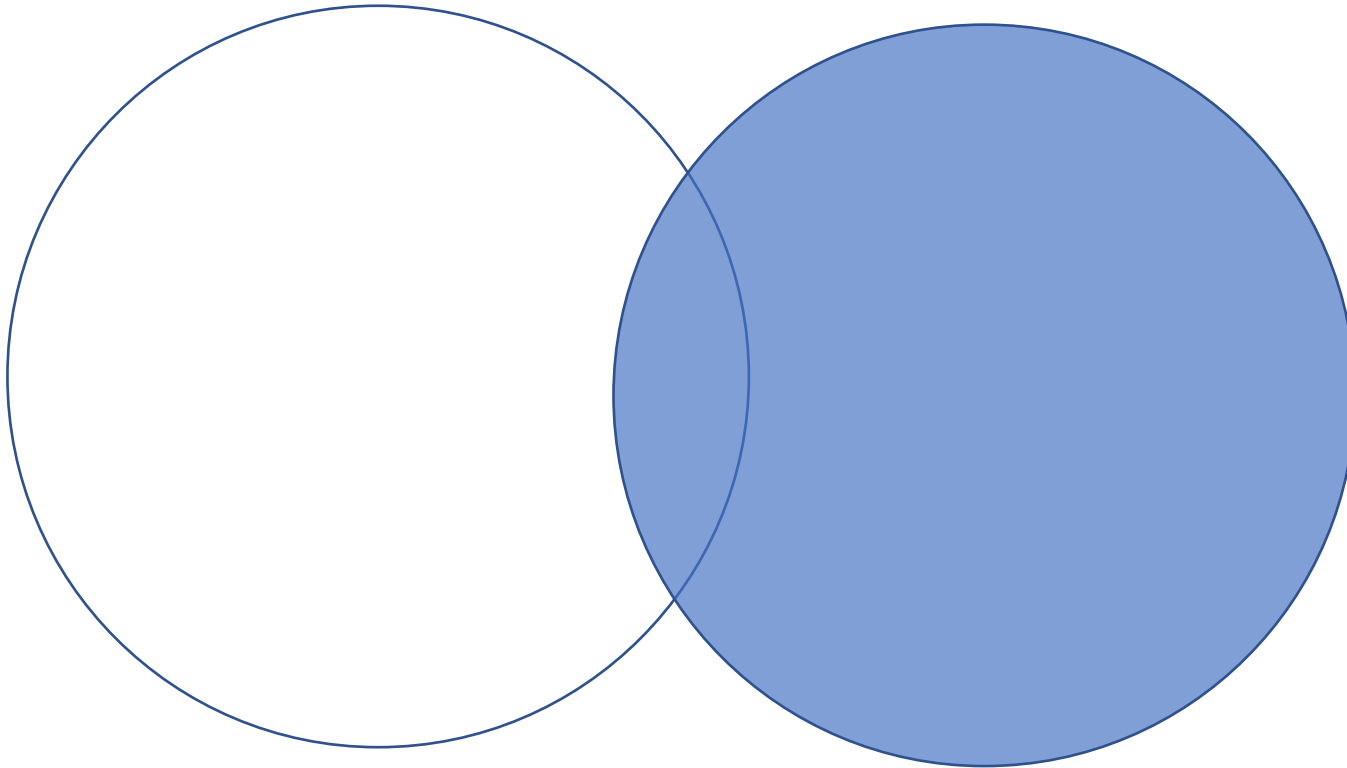
LEFT Join

Les données de LEFT et les intersections + les Données de RIGHT qui seront à null



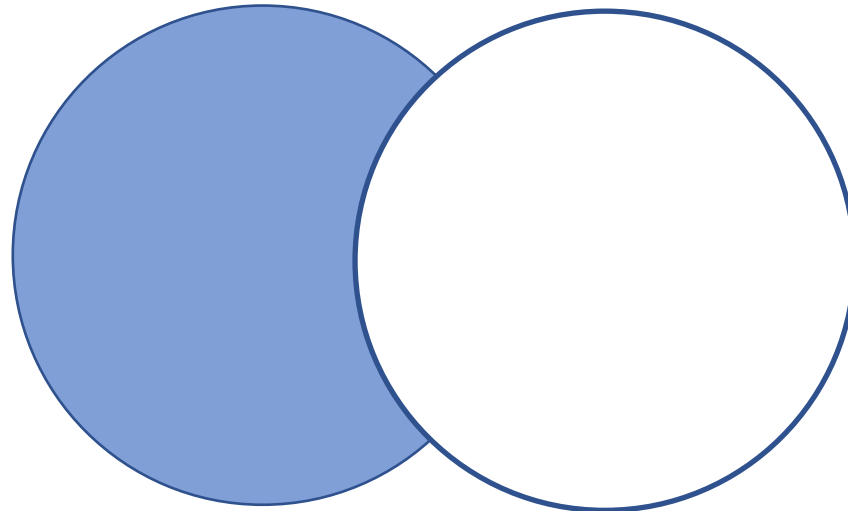
RIGHT Join

Les données de RIGHT + les intersections et les Données de LEFT qui seront à null



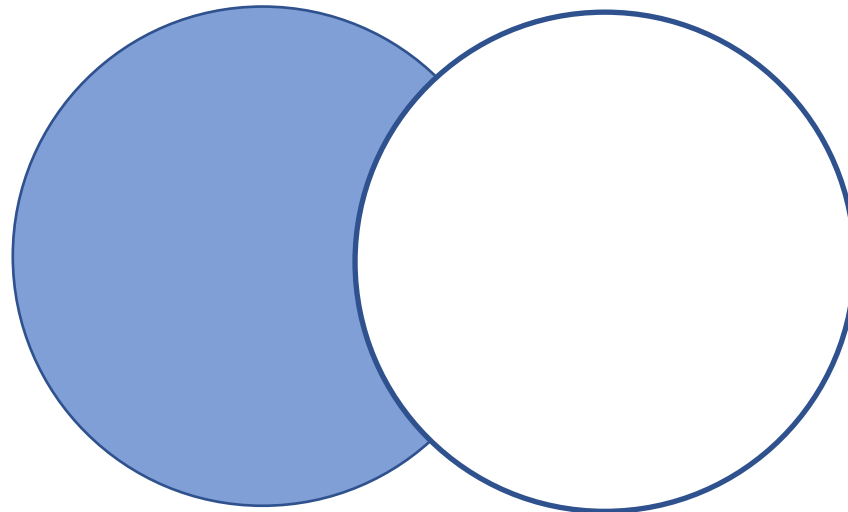
NO Intersect

- Maintenant qu'on sait comment récupérer des données qui sont liés, voyons comment on peut récupérer des données qui n'existe pas.
- Par exemple, on a des 'user' et des 'order'. Sauf qu'on ne sait pas si tous nos utilisateurs ont passés des commandes.



NO Intersect

- Quand on y pense, cela ressemble beaucoup à un LEFT JOIN sauf qu'on ne veut pas de la partie d'intersection.



No Intersect

```
SELECT COUNT(*) FROM user LEFT JOIN `order` ON user.id = `order`.user WHERE `order`.user IS NULL;
```

SUM

- La fonction SUM, comme le COUNT, est une fonction agrégateur.

```
SELECT SUM(cost_price) FROM product;
```

ORDER BY

- La commande ORDER BY est utilisé pour trier les résultats en ASC ou DESC

```
SELECT * FROM product ORDER BY selling_price ASC;
```

GROUP BY

- La commande GROUP BY, est un 'agrégateur' qui s'utilise avec une fonction agrégateur.
- Par exemple, on pourrait l'utiliser pour avoir les totaux de chaque 'order' depuis 'order_line'

```
SELECT SUM(total) FROM order_line GROUP BY `order`;
```

TP

- Réaliser une requête pour connaître les 20 produits réalisant le plus de chiffre d'affaires.

TP

- Réaliser une requête pour connaître les 20 produits ayant réalisé le plus de chiffre d'affaires.

```
SELECT SUM(total) AS 'Total produit', product FROM order_line GROUP BY `product` ORDER BY `Total produit` DESC LIMIT 20;
```

VIEWS

VIEW

- Une 'view' en MySQL est une collection de commande retournant un tableau.

Admettons que dans notre application, nous avons souvent besoin de retourner les 20 premiers `user`. On aurait une requête select comme ceci:

```
SELECT * FROM user LIMIT 20;
```

VIEW

- Au lieu de re-écrire cette requête à chaque fois, on pourrait l'enregistrer dans une `view` pour pouvoir réutiliser

```
CREATE VIEW getFirst20Users AS SELECT * FROM user LIMIT 20;
```

```
SELECT * FROM getFirst20Users;
```

TP

- Créer une vue pour récupérer les 20 commandes ayant généré le plus de chiffres d'affaires

TP

- Créer une vue pour récupérer les 20 commandes ayant généré le plus de chiffres d'affaires

```
CREATE OR REPLACE VIEW get20MostImportantOrders AS
  SELECT `order` AS 'Numero commande', ROUND(SUM(total), 2) AS 'Total commande'
  FROM order_line GROUP BY `order` ORDER BY `Total commande` DESC LIMIT 20;

SELECT * FROM get20MostImportantOrders;
```

Stored Procedure

Delimiter

- Le 'délimiter' est le symbole utilisé pour finir une instruction en SQL
- Par défaut il est ';'
- Nous pouvons le changer comme ceci:

```
DELIMITER $$
```


Delimiter

- Nous ne devons pas oublier de le rechanger quand on a fini de l'utiliser

```
DELIMITER $$  
SELECT COUNT(*) FROM user $$  
DELIMITER ;
```

Session variable

```
SET @myvar = 'contenu de ma variable de session';
```

```
SELECT @myvar;
```

Stored Procedure (Procédure Stockée)

- Une procédure stockée est un concept utilisé pour exécuter une suite d'instruction. Un peu comme une fonction avec un nom...
- Étant donnée que au sein de la suite d'instruction nous allons utiliser un délimiteur, afin de contourner le conflit de délimiteur au sein de la suite et la fin de la procédure elle-même, nous devons changer le délimiteur avant et le réinitialiser après la procédure.

Stored Procedure (Procédure Stockée)

- Création d'une procédure stockée

```
DELIMITER $$  
CREATE PROCEDURE numberOfUsers()  
BEGIN  
    SELECT COUNT(*) FROM user;  
END $$  
DELIMITER ;
```

Stored Procedure (Procédure Stockée)

- Nous pouvons appeler notre procédure stockée avec 'CALL'

```
DELIMITER $$  
CREATE PROCEDURE numberOfUsers()  
BEGIN  
    SELECT COUNT(*) FROM user;  
END $$  
DELIMITER ;  
  
CALL numberOfUsers();
```

Stored Procedure (Procédure Stockée)

- Nous pouvons supprimer notre procédure stockée avec 'drop'

```
DROP PROCEDURE numberOfUsers();
```

Stored Procedure (Procédure Stockée)

- On avait vu qu'une procédure stockée était fait pour exécuter plusieurs instructions. Essayons le

```
DELIMITER $$  
CREATE PROCEDURE numberOfUsers()  
BEGIN  
    SELECT COUNT(*) FROM user;  
    SELECT COUNT(*) FROM product;  
END $$  
DELIMITER ;  
CALL numberOfUsers();
```

[S1000] Attempt to close streaming result set com.mysql.cj.protocol.a.result.ResultsetRowsStreaming@315d103b that was not registered. Only one streaming result set may be open and in use per-connection. Ensure that you have called .close() on any active result sets before attempting more queries.

Stored Procedure (Procédure Stockée)

- Ce que nous pouvons faire c'est d'utiliser des variables de sessions pour enregistrer les valeurs et les réutiliser en dehors de notre proc-stock

```
SET @numberOfUsers = 0;
SET @numberOfProducts = 0;

DROP PROCEDURE numberOfUsers;
DELIMITER $$
CREATE PROCEDURE numberOfUsers()
BEGIN
    SELECT COUNT(*) FROM user INTO @numberOfUsers;
    SELECT COUNT(*) FROM product INTO @numberOfProducts;
END $$
DELIMITER ;

CALL numberOfUsers();

select @numberOfUsers;
select @numberOfProducts;
```


Stored Procedure (Procédure Stockée)

- Ce qui serait encore plus intéressant, c'est de passer des paramètres dans notre procédure.
- Prenons une requête simple, la récupération du nombre de femmes dans notre BDD.

```
SELECT COUNT(*) FROM user WHERE gender='female'
```

Stored Procedure (Procédure Stockée)

- Mettons la requête dans une procédure stockée 'numberOfWomen'

```
DELIMITER $$  
CREATE PROCEDURE numberOfWomen()  
BEGIN  
    SELECT COUNT(*) FROM user WHERE gender='female';  
END $$  
DELIMITER ;  
CALL numberOfWomen();
```

Stored Procedure (Procédure Stockée)

- Nous devons maintenant récupérer le nombre d'hommes de notre BDD. En soit, il faudrait que puisse passer (male/female) dans notre procédure et faire que la valeur soit dans notre requête.

```
DELIMITER $$
CREATE PROCEDURE numberOfWomen()
BEGIN
    SELECT COUNT(*) FROM user WHERE gender='female';
END $$
DELIMITER ;
CALL numberOfWomen();
```

Stored Procedure (Procédure Stockée)

- Il y a 3 types de paramètres pour les procédures stockées:
 - IN (la valeur initiale n'est pas modifiée)
 - OUT (La variable de sortie, équivalent aux variables de sessions avec valeur par défaut NULL)
 - INOUT (La valeur initiale peut être modifié par la procédure)

Stored Procedure (Procédure Stockée)

IN (la valeur initiale n'est pas modifiée)

```
DROP PROCEDURE numberOfWomen;  
DELIMITER $$  
CREATE PROCEDURE numberOfWomen(IN genderParam VARCHAR(255))  
BEGIN  
    SELECT COUNT(*) FROM user WHERE gender=genderParam;  
END $$  
DELIMITER ;  
CALL numberOfWomen('male');  
CALL numberOfWomen('female');
```

Stored Procedure (Procédure Stockée)

OUT (La variable de sortie, équivalent aux variables de sessions avec valeur par défaut NULL)

```
DROP PROCEDURE numberOfWomen;  
  
DELIMITER $$  
CREATE PROCEDURE numberOfWomen(IN genderParam VARCHAR(255), OUT numberOfGivenGender INT)  
BEGIN  
    SELECT COUNT(*) INTO numberOfGivenGender FROM user WHERE gender=genderParam;  
END $$  
DELIMITER ;  
CALL numberOfWomen('male', @numberOfGivenGender);  
  
SELECT @numberOfGivenGender;
```

Stored Procedure (Procédure Stockée)

INOUT (La valeur initiale peut être modifiée par la procédure)

```
DROP PROCEDURE numberOfWomen;  
SET @counter = 0;  
  
DELIMITER $$  
CREATE PROCEDURE numberOfWomen(  
    IN genderParam VARCHAR(255),  
    OUT numberOfGivenGender INT,  
    INOUT counter INT  
)  
BEGIN  
    SELECT COUNT(*) INTO numberOfGivenGender FROM user WHERE gender=genderParam;  
    SET counter = counter + 1;  
END $$  
DELIMITER ;  
CALL numberOfWomen('male', @numberOfGivenGender, @counter);  
  
SELECT @counter;
```

TP

- Créer une procédure stockée pour récupérer le montant total pour une commande. L'id de la commande doit pouvoir être passer en paramètre et on doit pouvoir récupérer le montant total en dehors de la procédure en 2 décimal.

TP

- Créer une procédure stockée pour récupérer le montant total pour une commande. L'id de la commande doit pouvoir être passer en paramètre et on doit pouvoir récupérer le montant total en dehors de la procédure en 2 décimal.

```
DROP PROCEDURE totalCommande;  
DELIMITER $$  
CREATE PROCEDURE totalCommande(IN idOrder INT, OUT montantTotalCommande FLOAT)  
BEGIN  
    SELECT ROUND(SUM(total)) INTO montantTotalCommande FROM order_line WHERE `order`=idOrder;  
END $$  
DELIMITER ;  
  
CALL totalCommande(1, @montantTotalCommande);  
SELECT @montantTotalCommande;
```

IF EXISTS

- Comme pour la création des bases de données et tableaux, on peut utiliser IF EXIST pour les 'view' et procédures stockées.
- Désormais nous allons les utiliser pour éviter des erreurs lors de la création ou suppression

LOOPS

SI

```
DROP PROCEDURE IF EXISTS updateCounter;
```

```
DELIMITER $$
```

```
CREATE PROCEDURE updateCounter(
```

```
    i INT,
```

```
    INOUT counter INT
```

```
)
```

```
BEGIN
```

```
    IF counter <= i THEN
```

```
        SET counter = counter + 1;
```

```
    END IF;
```

```
END $$
```

```
DELIMITER ;
```

```
SET @counter = 0;
```

```
CALL updateCounter(10, @counter);
```

```
SELECT @counter;
```

La boucle WHILE

```
DROP PROCEDURE IF EXISTS updateCounter;
```

```
DELIMITER $$
```

```
CREATE PROCEDURE updateCounter(
```

```
    i INT,
```

```
    INOUT counter INT
```

```
)
```

```
BEGIN
```

```
    WHILE counter <= i DO
```

```
        SET counter = counter + 1;
```

```
    END WHILE;
```

```
END $$
```

```
DELIMITER ;
```

```
SET @counter = 0;
```

```
CALL updateCounter(10, @counter);
```

```
SELECT @counter;
```

La boucle FOR (du moins son équivalent)

```
DROP PROCEDURE IF EXISTS updateCounter;
```

```
DELIMITER $$
```

```
CREATE PROCEDURE updateCounter(
```

```
    i INT,
```

```
    INOUT counter INT
```

```
)
```

```
BEGIN
```

```
    nomDeMaBoucle: LOOP
```

```
        IF counter >= i THEN
```

```
            LEAVE nomDeMaBoucle;
```

```
        END IF;
```

```
        SET counter = counter + 1;
```

```
        ITERATE nomDeMaBoucle;
```

```
    END LOOP ;
```

```
END $$
```

```
DELIMITER ;
```

```
SET @counter = 0;
```

```
CALL updateCounter(5, @counter);
```

```
SELECT @counter;
```

Transactional

ACID

- **ATOMIC**

La propriété d'atomicité assure qu'une transaction se fait au complet ou pas du tout

- **COHERENCE**

La propriété de cohérence assure que chaque transaction amènera le système d'un état valide à un autre état valide

- **ISOLATION**

Toute transaction doit s'exécuter comme si elle était la seule sur le système

- **DURABILITY**

La propriété de durabilité assure que lorsqu'une transaction a été confirmée, elle demeure enregistrée même à la suite d'une panne d'électricité, d'une panne de l'ordinateur ou d'un autre problème

AUTOCOMMIT

- De manière générale, lorsqu'on exécute une requête dans notre bdd, la mise à jour se fait instantanément.
- Quand on a plusieurs requête interdépendantes a exécuter, dans un ordre, il pourrait être intéressant d'exécuter les requêtes une première fois virtuellement pour voir si tout se passe bien et les exécuter réellement seulement si tout c'est bien passé.
- C'est cette notion qu'on appelle une transaction

AUTOCOMMIT

- Pour commencer il faudrait dire à MySQL d'exécuter les commandes de façon virtuel seulement.

```
SET AUTOCOMMIT = 0;
```

Transaction

- Ensuite comme on voudra exécuter plusieurs requêtes, il faut les mettre dans un 'bloc'. Ce bloc sera notre transaction.
- Ce bloc commence avec 'START TRANSACTION' et se termine par 'ROLLBACK/COMMIT'

```
USE toto;  
SET AUTOCOMMIT = 0;  
START TRANSACTION;  
DELETE FROM employee_manager WHERE TRUE;  
SELECT COUNT(*) FROM employee_manager;  
ROLLBACK ;
```

Plus sérieusement...

Demande 2

- Il y a des problèmes de performances lors de l'affichage des listes de factures et des factures. Améliorer la performance