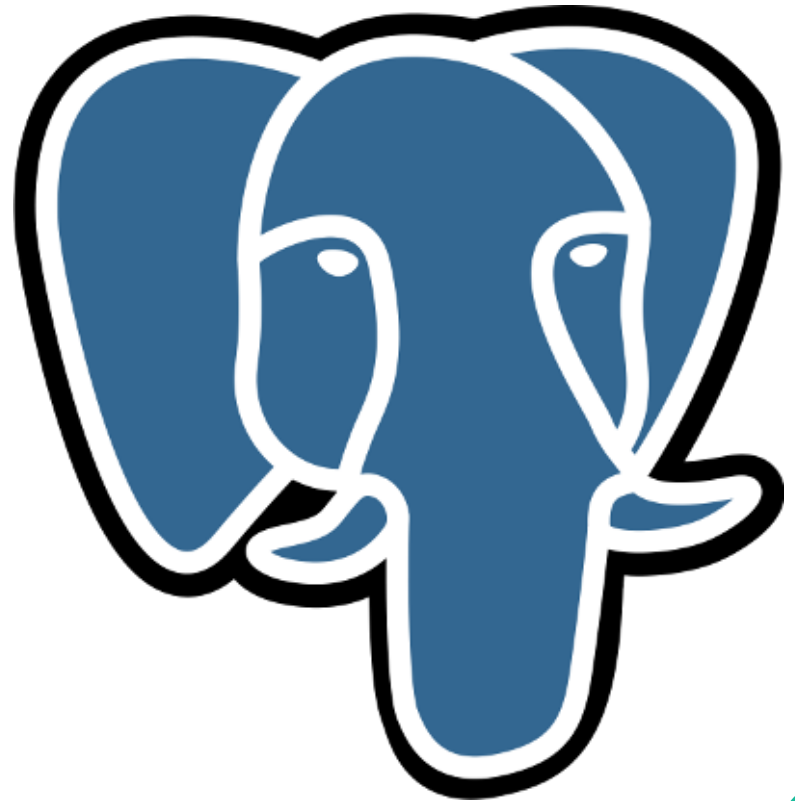


# SQL et BDD relationnelles : concepts, requêtes

## Au programme

- **C**reate **R**ead **D**elete  
**U**ppdate
- Bases d'administration
- Triggers
- Postgres >11.4



# Sources et documents annexes

A des fins de reproductibilité une image docker de la base de données finale est disponible à [cette adresse](#)

# Origine des données

<https://www.kaggle.com/ophi/mpi>

<https://datahub.io/core/corruption-perceptions-index>

<https://data.worldbank.org/indicator/sp.pop.totl>

# Distinctions préliminaires

- « **SQL** » est un **langage**, non un **système de gestion de base de données** (SGBD)
- Plusieurs SGBD sont basés sur SQL. Certains sont **open source** (Postgres, MariaDB), d'autres sont **propriétaires** (MSSQL, OracleDB)
- Il existe quelques différences dans le langage de requête de ces backends, mais minimales
- La vraie différence entre ces backends concerne leur **administration** (mais nous y viendrons plus tard)

# Distinctions préliminaires

## SQL

**ex** : Postgres, OracleDB

Données structurées sous la forme de tables

## No-SQL

**ex** : MongoDB, Cassandra

Données structurées sous la forme de graphs

**Ne soyez pas naïfs !** Le fait qu'un SGDB suive une structuration par graph n'implique pas *ipso facto* qu'il soit impossible de requêter à l'aide d'un langage « *à la SQL* »

# Installation de Postgres

**Windows** : Installer

<https://www.postgresql.org/download/windows/>

**Linux** (deb) : `apt-get install postgresql postgresql-client`

**MacOSX** : lol                      `~\_ ( ツ ) \_ /`

# Création d'un utilisateur et d'une BDD

- **CREATE DATABASE** countries;
- **CREATE USER** nom\_de\_l'utilisateur **WITH PASSWORD** 'le\_mdp\_de\_votre\_choix';
- **GRANT ALL PRIVILEGES ON DATABASE** countries **TO** nom\_de\_l'utilisateur;

Essayez maintenant de vous logger avec ce nouvel utilisateur dans cette base :

```
psql -U nom_de_l'utilisateur -d countries
```

# Création d'une table première table

```
CREATE TABLE IF NOT EXISTS mpi_national(  
  iso CHAR(3) PRIMARY KEY NOT NULL,  
  country_name VARCHAR(80) NOT NULL UNIQUE,  
  mpi_urban REAL NOT NULL,  
  headcount_ratio_urban REAL NOT NULL,  
  intensity_urban REAL NOT NULL,  
  mpi_rural REAL NOT NULL,  
  headcount_ratio REAL NOT NULL,  
  intensity_rural REAL NOT NULL  
);
```



# Types et contrainte (liste non exhaustive)

## Typage de données

**CHAR** caractères (longueur constante)

**VARCHAR** caractères (longueur variable)

**REAL** nombres (jusqu'à 6 décimales)

**INTEGER** entiers écrits sur 4 bytes

**BIGINT** entiers écrits sur 8 bytes

**SERIAL** entiers auto incrémentés

## Contraintes d'intégrité

**NULL** donnée absente  
permise

**UNIQUE** occurrence  
unique d'une même valeur

# Importation de données dans cette table

```
COPY mpi_national(iso, country_name, mpi_urban,  
headcount_ratio_urban,intensity_urban,mpi_rural,headco  
unt_ratio,intensity_rural)  
FROM '/data/MPI_national.csv' DELIMITER ',' NULL ''  
CSV HEADER ;
```

# Reconnaissance

Lister les tables d'une DB :

**\d**

Décrire une table :

**\d nom\_de\_ma\_table**

# Commençons simple

Sélectionnez (dans la table : **mpi\_national**)

- Le nom des pays dont le taux urbain de pauvreté (mpi\_urban) est supérieur à 0,2
- Le nom des pays dont les taux urbain (mpi\_urban) est supérieur à 0,1 et le taux rural (mpi\_rural) est supérieur 0,6
- Le nom des pays où la pauvreté en milieu urbain est supérieure à celle du milieu rural

# Commençons simple

Selection : pattern général

```
SELECT [colonnes] FROM [une table] WHERE  
[condition_que_les_lignes_doivent_remplir]
```

# Commençons simple

Sélectionnez

- **SELECT** country\_name **FROM** mpi\_national **WHERE** mpi\_urban > 0.2;
- **SELECT** country\_name **FROM** mpi\_national **WHERE** (mpi\_rural > 0.6 **AND** mpi\_urban > 0.1);
- **SELECT** country\_name **FROM** mpi\_national **WHERE** mpi\_urban < mpi\_rural;

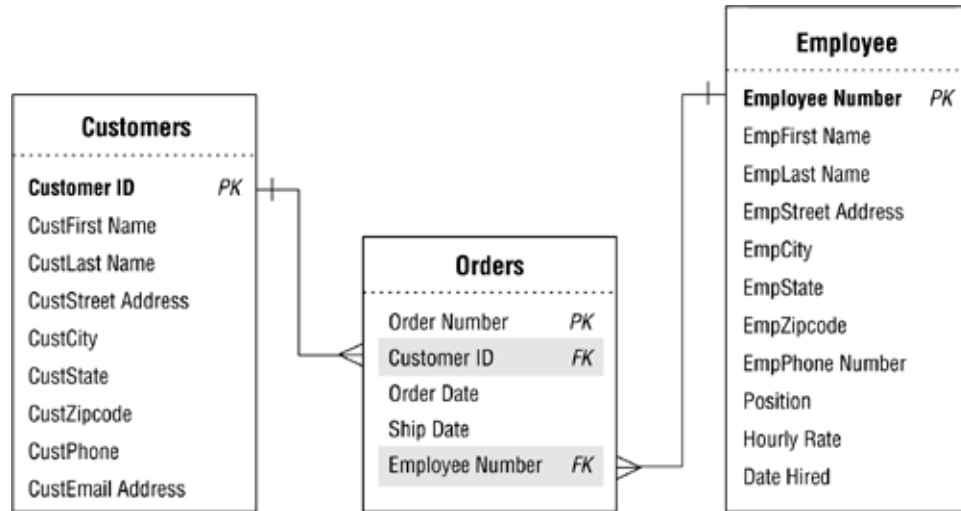
# Création d'une nouvelle table : à vous

Vous allez créer une seconde table (**mpi\_subnational**) à partir du CSV du même nom.

Seule variante par rapport à table précédente : le champ iso reposera sur une **clef étrangère**

iso **CHAR**(3) **REFERENCES** mpi\_national(iso),

# Clefs et clefs étrangères



- Raison pour laquelle on parle de « **base relationnelles** »

Mécanisme garant de :

- **l'intégrité** d'une base
- Sa **consistance**

(importante structuration métier)



# Jointures

De par la relation entre nos deux tables, nous pourrions avoir besoin d'extraire des lignes de l'une **compte tenu des données dans l'autre**

# Quelques JOIN

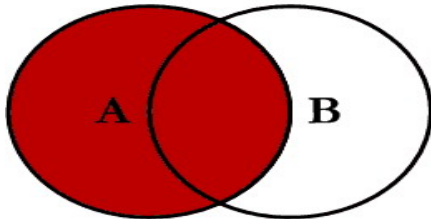
## reminder

Sélectionnez (**mpi\_national** et **mpi\_subnational**)

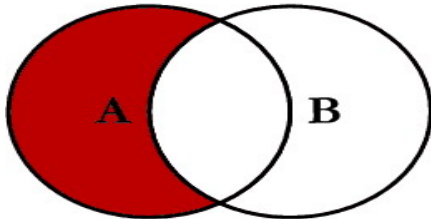
- les régions (**subregion\_name**) des pays dont le ratio urbain (**headcount\_ratio\_urban**) est supérieur à 70
- La moyenne du taux de pauvreté urbain des pays d'asie (**world\_region\_name**)

# Rappel sur les joins, ou de l'art de « penser avec des patates »

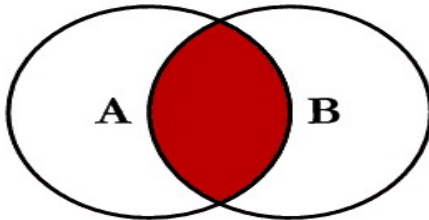
## SQL JOINS



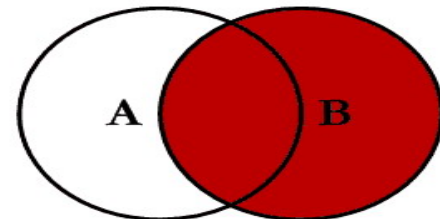
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



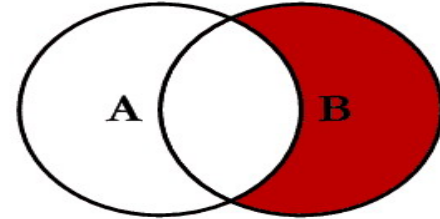
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



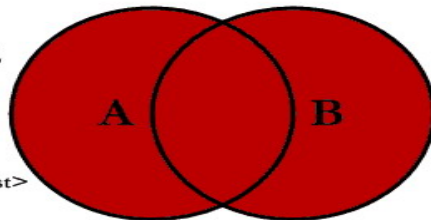
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



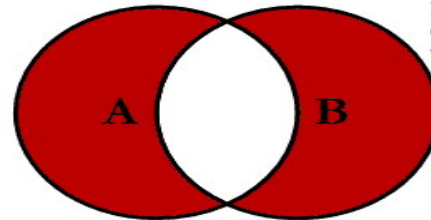
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```

# Quelques JOIN

Sélectionnez (mpi\_national et mpi\_subnational)

- **SELECT** mpi\_subnational.region **FROM**  
mpi\_subnational  
**INNER JOIN** mpi\_national **ON** mpi\_subnational.iso  
= mpi\_national.iso  
**WHERE** mpi\_national.headcount\_ratio\_regional  
>70;

# Quelques JOIN

Sélectionnez (mpi\_national et mpi\_subnational)

- **SELECT** AVG(mpi\_national.mpi\_urban) **FROM**  
mpi\_national

**INNER JOIN** mpi\_subnational **ON**  
mpi\_subnational.iso = mpi\_national.iso

**WHERE**  
mpi\_subnational.world\_region\_name="East Asia  
and the Pacific";

# Quelques JOIN

**Sélectionnez (MPI\_national et MPI\_subnational)**

- **Les régions d'Asie i) dont le MPI\_regional est supérieur au MPI\_national et ii) dont le pays a un Headcount\_Ratio\_Rural supérieur à 70**
- **.... les classer par iso** en indiquant sur chaque ligne le nom du pays correspondant

# Quelques JOIN

Sélectionnez (mpi\_national et mpi\_subnational)

```
SELECT mpi_subnational.region, mpi_subnational.country  
FROM mpi_subnational INNER JOIN mpi_national ON  
MPI_national.country_code = mpi_subnational.country_code  
WHERE (mpi_subnational.mpi_regional >  
mpi_subnational.mpi_national AND  
mpi_national.headcount_ratio > 70)  
ORDER BY mpi_subnational.country_code;
```

# Deux nouvelles tables à créer et on est bon

Nous allons ajouter des données temporelles à notre DB

Importez les données contenues dans :

- corruption.zip
- population.zip

L'importation va probablement nécessiter quelques tweaks :/



# Pivot d'une table en Postgres

Ces données temporelles mériteraient probablement d'être « pivotées ».

postgresql > 9.4 dispose d'une fonction **crosstab()** pour cela

Un exemple proche

Pivot

```
df.pivot(index='foo',  
          columns='bar',  
          values='baz')
```

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t



bar	A	B	C
foo			
one	1	2	3
two	4	5	6

# Regardons du côté de la corruption et de la population

## Sur population

- Quel pays possède le plus d'habitants en 2017?
- Quel pays a connu la plus faible progression de sa population depuis 1990 ?
- "" les trois pays
- Quel pays a connu la plus forte progression de sa population entre son premier sondage et 2010 (attention, varie d'un pays à l'autre)?

# Regardons du côté de la corruption et de la population

## Sur population

- **SELECT** country\_name **FROM** world\_population **WHERE** "2017" = (**SELECT** MAX("2017") **FROM** population);
- **SELECT** country\_name **FROM** world\_population **WHERE** ("2010" / "1990") = (**SELECT** **MIN**("2010" / "1990") **FROM** world\_population);

# Chaud chaud, cacao (pour ceux qui veulent suer un coup)

- En moyenne, les pays à la pauvreté urbaine la plus importante sont-ils également ceux dont la population a connu la plus grande croissance entre les années 50 et aujourd'hui ?

# Chaud chaud, cacao (pour ceux qui veulent suer un coup)

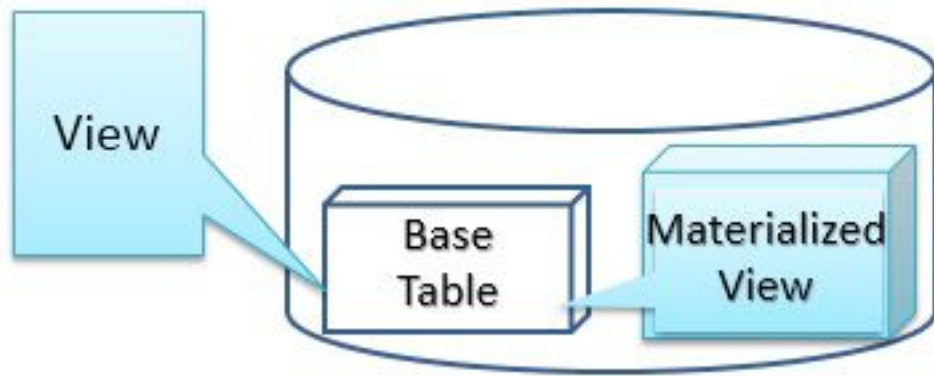
- En moyenne, les pays à la pauvreté urbaine la plus importante sont-ils également ceux dont la population a connu la plus grande croissance entre les années 50 et aujourd'hui ?

# Création d'une vue

Une vue est une version épurée d'une table (ou plus vraisemblablement d'une grosse jointure) destinée être visualisée facilement par le métier sans avoir à écrire cette même requête

**CREATE VIEW** nom\_de\_votre\_vue **AS** SELECT..... **CE QUE VOUS VOULEZ**

# Vue Matérialisée ? Kesako ?



Au lieu d'être le raccourci d'une requête,

- les données sont physiquement stockées sur le disque
- Réponse plus rapide (la requête n'est pas exécutée chaque fois)
- Nécessite d'être actualisé par un trigger

# Créons une vue complexe

Créez une vue du nom de `european_view` qui contiendra :

- **l'iso**
- Le **nom** du pays
- Le **mpi national**
- Le **taux de croissance** de population entre 1971 et 2000
- Le taux de corruption **maximal** entre 1998 et 2015  
de l'ensemble des pays d'europe



# Parlons un peu d'indexation

- **Indexer les lignes permet de dispenser le moteur de parcourir l'ensemble des lignes**  
→ **gain de vitesse et de ressource énorme !!!!**

# Sur quoi indexer ?

**Rule of thumb** : il vaut mieux indexer sur les champs typiquement impliqués dans des clauses de type :

- **WHERE**
- **ORDER BY**
- **GROUP BY**
- **MIN MAX** (dans le cas de valeurs numériques)

Il n'y a donc de bonnes et de mauvaise indexation que **relativement** i) aux requêtes adressées à la BDD ii) dans le **contexte d'un usage** de celle-ci

# Comment indexer ?

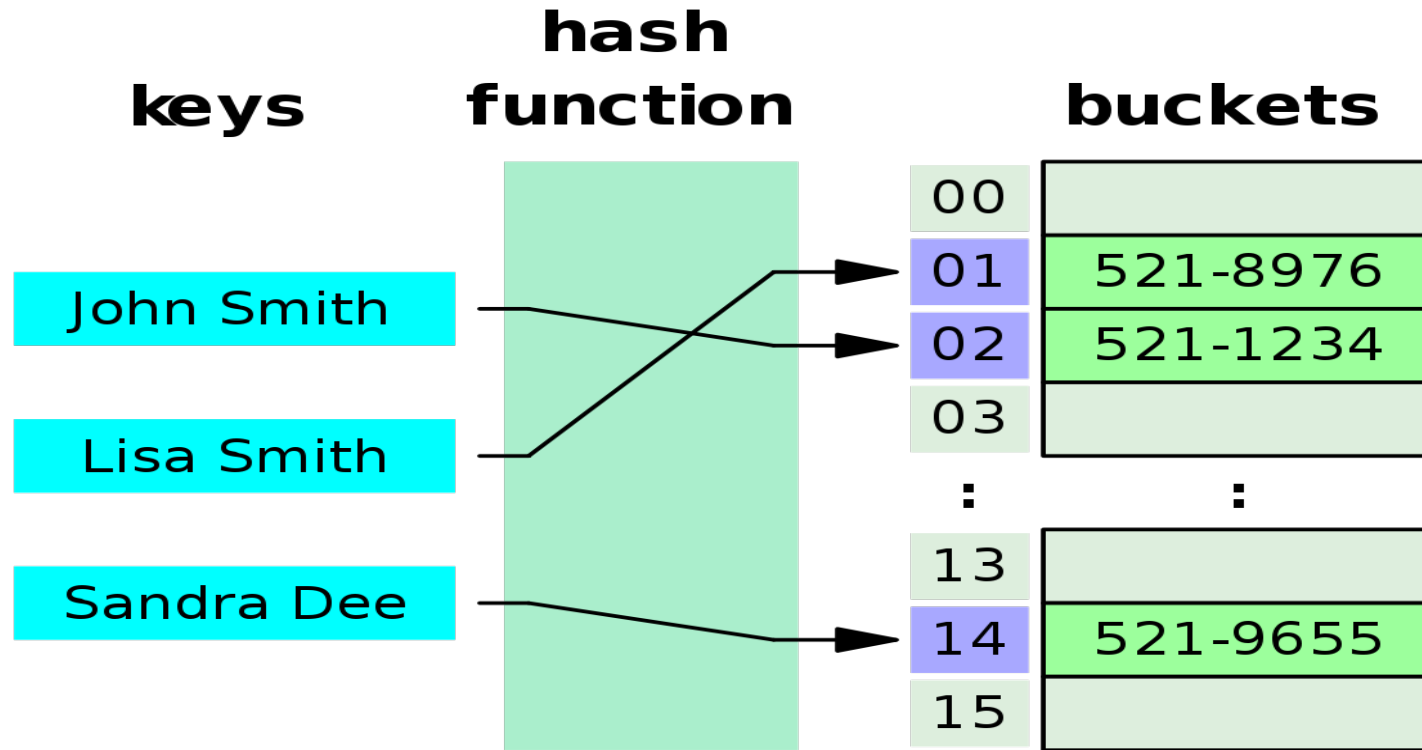
## En arbre équilibré

- Ou B-tree (balanced)
- InnoDB
- MyISAM

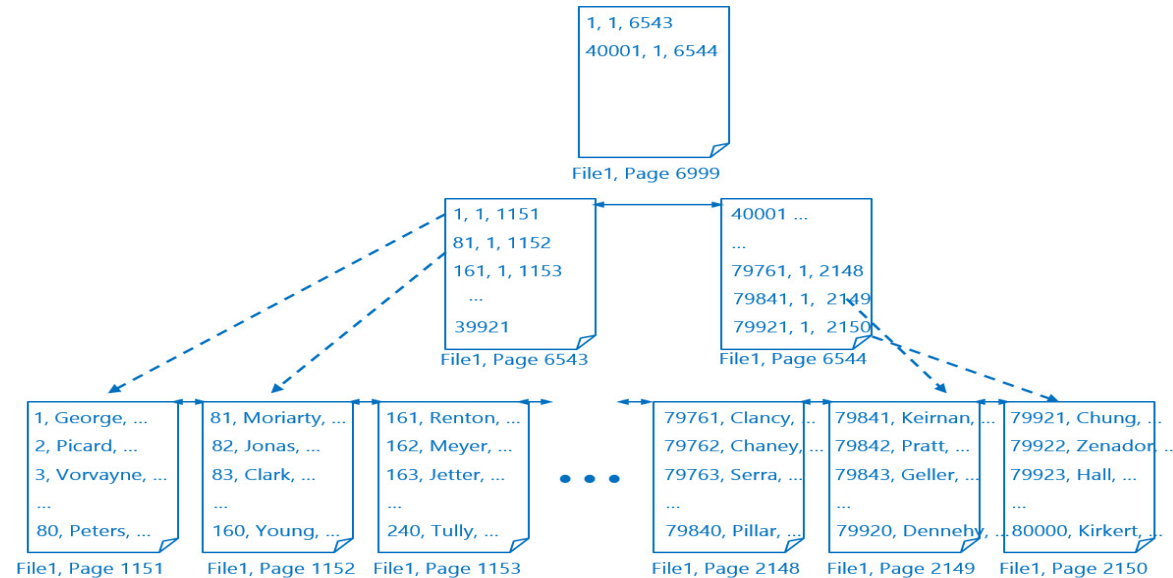
## Par table de Hashage

**NB** : ces deux méthodes ne sont pas nécessairement exclusives (pensez à **HEAP** ou **NDB cluster**)

# Par table de Hashage



# Deux types d'indexation en arbre à distinguer



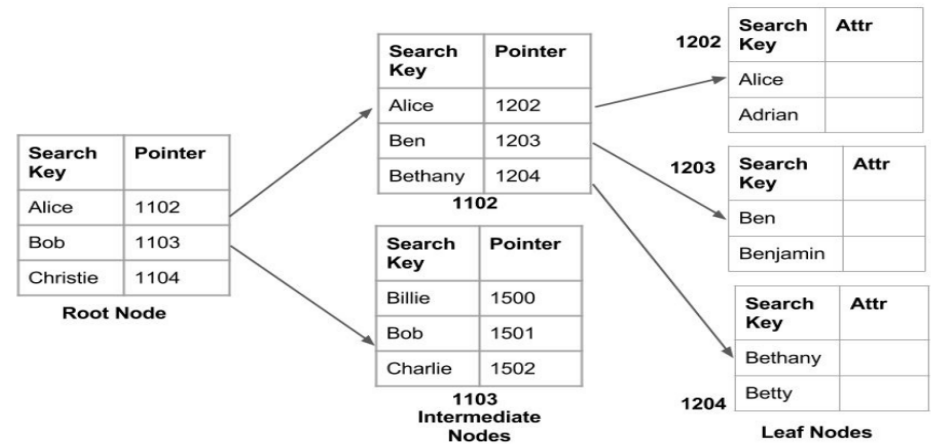
## Clustered indexes

- Semblable à une recherche dans l'annuaire
- Solidaire de la BDD
- 1 seul possible arbre possible (car dépendant de l'ordre des données)

# Deux types d'indexation en arbre à distinguer

## Non Clustered indexes

- Système de pointeurs (identifiant les lignes)
- Distinct de la DBB
- Plusieurs indexes possibles



Non clustered index

# Lister des indexes existants

SELECT

tablename,  
indexname,  
indexdef

FROM

pg\_indexes

WHERE

schemaname = 'public'

ORDER BY

tablename,  
indexname;

Vous remarquerez que des indexes ont été créés automatiquement par l'action :

- primary keys
- foreign keys

# La fonction EXPLAIN

Pour nous assurer du bon fonctionnement des indexations, nous allons avoir besoin d'un moyen de comparer de les requêtes avant et après indexation

```
EXPLAIN ANALYZE SELECT * FROM  
mpi_subnational WHERE world_region_name =  
'Sub-Saharan Africa';
```



# Let's explain EXPLAIN

Seq Scan on  
mpi\_subnational  
(**cost**=0.00..24.30  
**rows**=431 **width**=60)

**Cost**  
**rows**  
**with**

# Creation d'un index

```
CREATE INDEX nom_index ON nom_table  
(colonne_a_indexer);
```

## Convention :

le nom des index contient  
« idx »

## Nota Bene :

- Par défaut, Postgres crée des index de type B-Tree
- En cas de valeur unique, a des fins d'optimisation, vous pouvez ajouter **UNIQUE**

# Indexons nos tables rationnellement

## Notez au passage :

- Nous venons de réaliser une indexation de type « FULL TEXT »
- Seuls InnoDB et MyISAM permettent ce type d'indexation
- Restreint aux colonnes de données de type CHAR, VARCHAR, TEXT

# Indexons nos tables rationnellement

Commençons par MPI\_subnational. Créez un index pour world\_region\_name

```
CREATE INDEX idx_world ON  
mpi_subnational(world_region_name);
```

Comparez **EXPLAIN**. Qu'en deduisez-vous ?

# Indexons nos tables rationnellement

N'est-ce pas un peu c\*\*, ce que nous venons de faire ?!  
Comparons deux cas de figure

## Valeurs enchassées



## Colonnes non alignées



# Indexons nos tables rationnellement

Dans le cas où les valeurs d'une des colonnes matchent toujours celle d'une autre colonnes, mieux vaut une **indexation multi-colonnes** que deux **index distincts**

- Point d'entrée unique
- Moins de charge en mémoire

# Indexons nos tables rationnellement

Et passons à notre index multi-colonnes

```
CREATE INDEX idx_world_country ON MPI_subnational  
(world_region,country_code);
```

**NB :**

- Règle du left most prefix
- Ne s'applique qu'à des indexes de type B-tree

# Quelques down sides à garder à l'esprit

## **N'abusez pas des index car :**

- Ils prennent de la place de mémoire
- Ils doivent être mis à jour à chaque requêtes d'insertion, modification, suppression

## **Solution rationnelle :** distinguez, selon les besoins

- Des tables « stables » fortement indexées (requêtes de recherche)
- Des tables « volatiles » pas ou peu indexées



# Trouvez des champs pertinents à indexer

## Approches

- Pensez aux requêtes typiques
- Un peu de bon sens
- Approche empirique

# A vous : essayons sur une grosse BDD

- Une BDD d'un million de lignes
- <https://www.sample-videos.com/download-sample-sql.php>
- - importez cette BDD dans une nouvelle BDD (videos)
- - trouvez des requêtes réalistes et utiles nécessitant plusieurs secondes d'exécution par le biais de joins entre ces tables
- - indexez de manière à mitiger ces latences

# Trigger et fonctions

- Triggers et fonctions sont un moyen commode d'assurer l'intégrité d'une base dont les valeurs de certaines tables **dépendent de celles d'autres**

## **Situations typiques :**

- Une table agrégée que l'on doit mettre à jour en cas de changement sur une autre
- Une table « *historique des changements* »

# Anatomie d'un trigger réalisé avec plpgsql

## La fonction

```
CREATE FUNCTION une_fonction() RETURNS  
trigger AS $$  
BEGIN  
    IF TG_OP = 'INSERT' THEN  
        ce_que_vous_voulez ;  
    ELSIF TG_OP = 'DELETE' THEN  
        ce_que_vous_voulez ;  
    END IF;  
    RETURN NEW;  
END $$ LANGUAGE plpgsql;
```

## Le trigger

```
CREATE TRIGGER  
trigger1 AFTER  
INSERT OR DELETE  
on doc_tag  
FOR EACH ROW  
EXECUTE  
PROCEDURE  
une_fonction();
```

# A nous

Création d'une table agrégée **par continent** :

- Le nombre de pays
- Taux de pauvreté urbain moyen
- Taux de pauvreté rural moyen

**Création d'un trigger à même d'actualiser cette table en cas de**

- **UPDATE**
- **INSERT**
- **DELETE**

# Sauvegarde d'une BDD

## En ligne de commande

**pg\_dump** -U nom  
d'utilisateur nom\_bdd >  
nom\_bdd.pgsql

## Graphiquement

**PgAdmin**

# Importation d'une BDD à partir d'un pgsql

## En ligne de commande

- créez une BDD
- **pg\_dump** -U nom d'utilisateur nom\_bdd < nom\_bdd.pgsql

# Importation d'une BDD à partir d'un pgsql

**Essayons d'importer cette base**

- <http://www.postgresqltutorial.com/wp-content/uploads/2019/05/dvdrental.zip>



# Assurons nous d'abord de quelques points

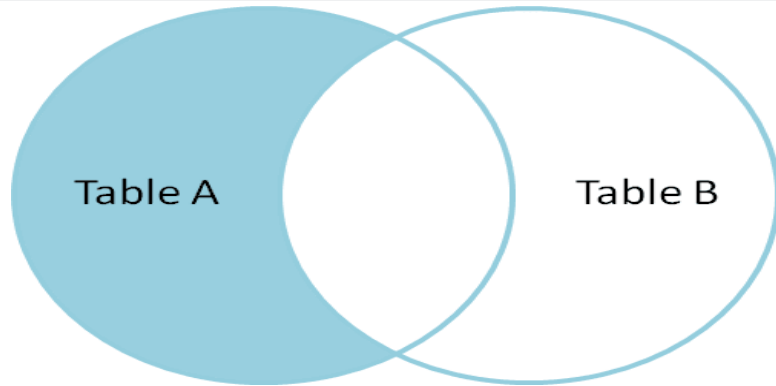
La consistance des  
`country_code` entre nos  
tables

Que les tables dénotent  
bien les mêmes  
ensembles de pays

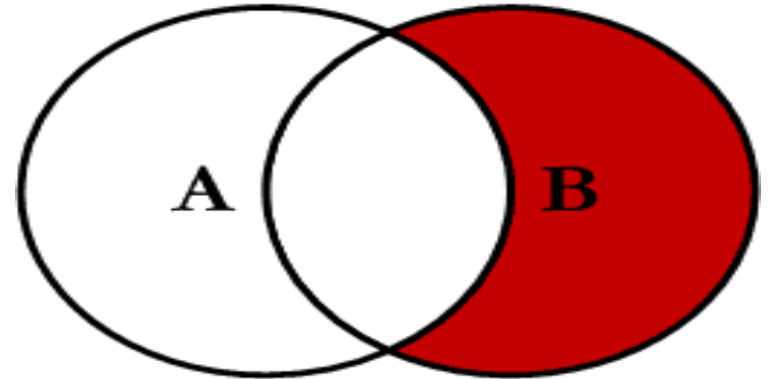
En termes ensemblistes, cela revient à vérifier si  $A - B = 0$  et  $B - A = 0$

NB : MySQL ne dispose pas de l'opérateur **MINUS**. Il va falloir penser avec des **JOIN** (une aide)

# Que les tables dénotent bien les mêmes ensembles de pays



```
SELECT MPI_national.country  
FROM MPI_national LEFT JOIN  
MPI_subnational ON  
MPI_national.country=MPI_subnational.country WHERE  
MPI_subnational.country IS  
NULL;
```

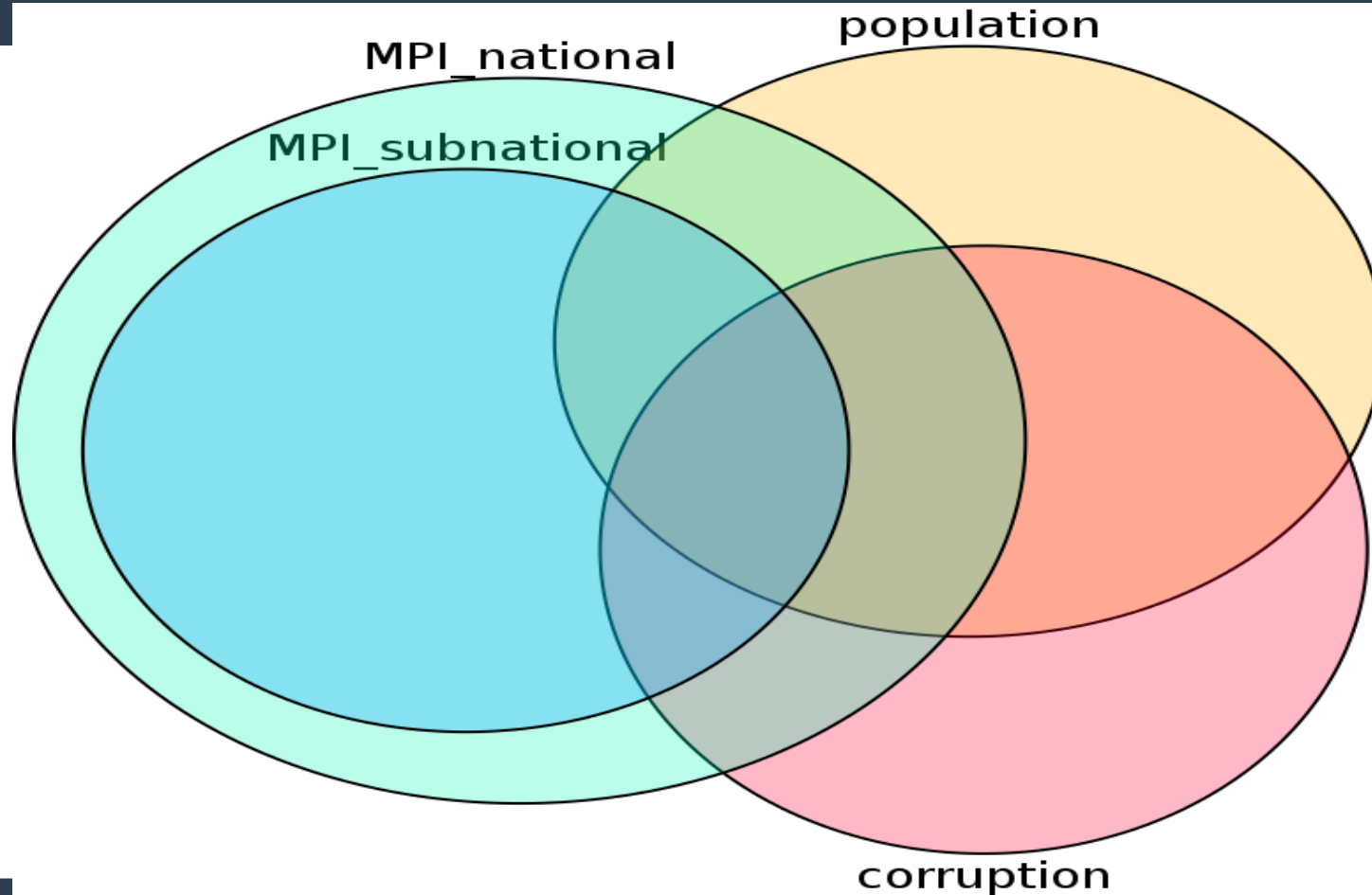


```
SELECT MPI_national.country  
FROM MPI_national RIGHT JOIN  
MPI_subnational ON  
MPI_national.country=MPI_subnational.country WHERE  
MPI_national.country IS NULL;
```

# Que les tables dénotent bien les mêmes ensembles de pays

**Avons nous des cas d'inclusion ?**

# Résumé synthétique de la situation



# Que les tables dénotent bien les mêmes ensembles de pays

**N'aurions pas des doublons ?**

**(« Nom\_de\_Pays » « Nom\_de\_Pays, Republic of »)**

# Parlons un peu du DBA

Aspects administratifs de la gestion d'une BDD

# Réplication Master-Slave

Soit deux serveurs MySQL

- **192.168.1.20** (master)
- **192.168.1.21** (slave)

## Côté Master

```
GRANT REPLICATION SLAVE ON *.* TO  
'replication_user'@'192.168.1.21' IDENTIFIED BY  
'motdepasse';
```

```
FLUSH PRIVILEGES;
```

```
FLUSH TABLES WITH READ LOCK;
```

# Réplication Master-Slave

Soit deux serveurs MySQL

- **192.168.1.20** (master)
- **192.168.1.21** (slave)

## Côté Master

```
GRANT REPLICATION SLAVE ON *.* TO  
'replication_user'@'192.168.1.21' IDENTIFIED BY  
'motdepasse';
```

```
FLUSH PRIVILEGES;
```

```
FLUSH TABLES WITH READ LOCK;
```



# Réplication Master-Slave

## Côté Master

**SHOW MASTER STATUS;**

•	+-----+-----+-----+-----+
•	File   Position   Binlog_Do_DB   Binlog_Ignore_DB
•	+-----+-----+-----+-----+
•	my_log.000001   <u>310</u>
•	+-----+-----+-----+-----+

**Si « empty set » → activez le bin-log de MySQL dans /etc/mysql/my.cnf**

**log\_bin = /var/log/mysql/my\_log.log**

# Réplication Master-Slave

## Côté slave

**STOP** slave;

- **CHANGE MASTER TO** MASTER\_HOST='192.168.1.20'  
MASTER\_USER='replication\_user',  
MASTER\_PASSWORD='motdepasse',  
MASTER\_LOG\_POS=310;
- **START** slave;

# Réplication Master-Slave

Côté master

**UNLOCK TABLES;**

Redémarrez les deux serveurs puis

**SHOW MASTER STATUS;**

**SHOW SLAVE STATUS;**

# Le cas de l'active-active replication (a.k.a master-master)

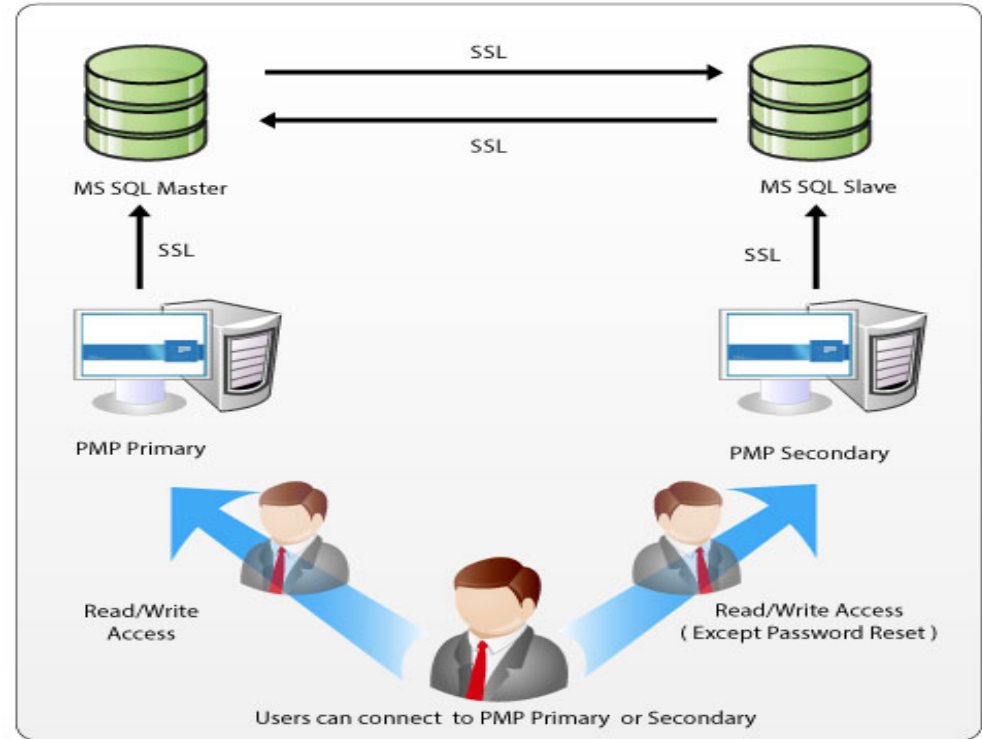
**On ne le fera pas mais sachez que cela existe (peu de variation avec la procédure précédente)**

- <https://www.linode.com/docs/databases/mysql/configure-master-master-mysql-database-replication/>

# Le cas de l'active-active replication (a.k.a master-master)

## Contexte d'utilisation

- Load balancing
- Mirroring de server web



# Une leçon plus générale

## Théorème de CAP (Brewer)

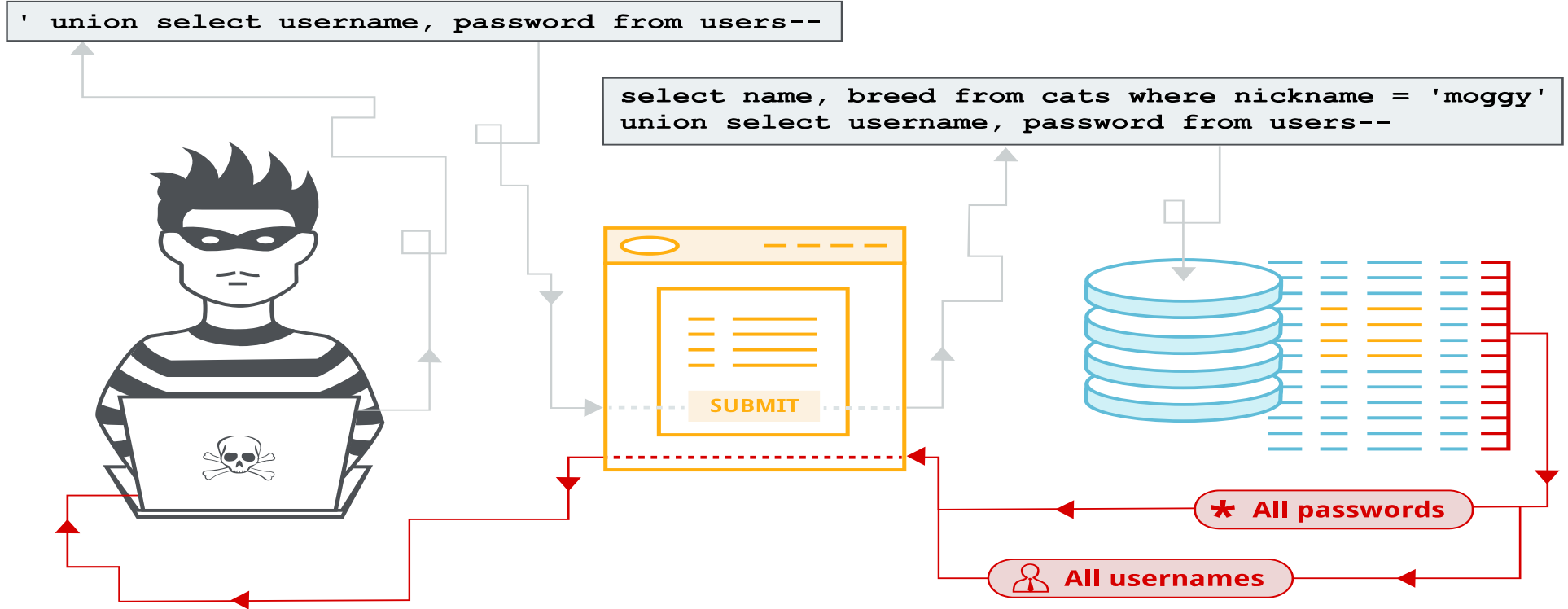
Dans un SGBD distribué sur le réseau, il est impossible de satisfaire simultanément ces trois exigences

- Consistence : chaque lecture reçoit la donnée la plus récente ou une erreur
- Disponibilité : chaque requête reçoit une réponse différente d'une erreur
- Tolérance au partitionnement : le système continue à fonctionner en dépit du delay ou perte induits par le réseau entre les nœuds

# Parlons un peu de sécurité des BDD



# Principe généraux d'une injection SQL





# Principe généraux d'une injection SQL

Mettons que vous ayez le code (PHP) suivant derrière un formulaire d'authentification

```
txtUserId = getRequestString("UserId");
```

```
txtSQL = "SELECT * FROM Users WHERE UserId = " +  
txtUserId;
```

Entrer ceci dans le  
formulaire....

**105 OR 1=1** →

.... donne la requête  
sql suivante:

```
SELECT * FROM Users  
WHERE UserId = 105 OR  
1=1;
```

# Principe généraux d'une injection SQL

**Or, cette expression est toujours vraie.**

**Dans un langage déclaratif comme sql, cela implique que cette requête sera toujours exécutée ( un parallèle avec un paradoxe logique )**

**Cette faille est ce que l'on appelle une disjonction (OR) avec une tautologie.**

→ notez qu'il existe plein d'autres manières d'injecter

# Finis la contemplation, passons à la création

- Jusqu'à présent, nous nous sommes contenté de « requêter » nos bases sans les altérer. Nous allons maintenant apprendre à
  - - créer de nouveaux champs au sein de tables existantes
  - - créer de nouvelles tables
- Afin d'enregistrer nos résultats en dur

# Création de nouvelles colonnes dans une table existante

- Pour créer une nouvelle colonne, nous allons avoir besoin d'ALTER TABLE
- 
- **ALTER TABLE** nom\_table
- **ADD** nom\_colonne type\_données ;
- Nous allons créer dans MPI\_national une colonne « ratio\_urban\_rural » (float)

# Création de nouvelles colonnes dans une table existante

- Maintenant que nous avons notre colonne, nous allons y ajouter le résultat de notre calcul à l'aide de la commande **UPDATE**
- **UPDATE** nom\_table **SET** colonne = valeur
- Nous allons définir le ratio\_urban\_rural comme le produit de la division  $\text{MPI\_URBAN} / \text{MPI\_Rural}$

# Création de nouvelles colonnes dans une table existante

- En résumé
- **ALTER TABLE** MPI\_national **ADD** ratio\_urban\_rural float;
- **UPDATE** MPI\_national **SET** ratio\_urban\_rural =  
MPI\_URBAN / MPI\_Rural;
- 
-

# Création de nouvelles colonnes dans une table existante

- Maintenant que vous disposez de la mécanique générale, créez :
- - [*population*] un champ « world\_region » contenant la région du pays en question (JOIN sur country\_code avec MPI\_subnational)
- - [*corruption*] le champ « country\_code » qui manquait jusqu'à présent (JOIN avec une table plus complète)
- - [*corruption*] un champ « highest » contenant le plus haut taux de corruption d'un pays (quelque soit la date)
-

# Création de nouvelles colonnes dans une table existante

- **drop procedure if exists yolo;**
- **create procedure yolo(out doom varchar(200))**
- **begin**
- **DECLARE v1 INT;**
- **set v1 = 1998;**
- **set doom = concat('year\_', v1);**
- 
- **WHILE v1 < 2015 DO**
- **SET v1 = v1 + 1;**
- **set doom = concat(doom, ', ', concat('year\_', v1));**
- **END WHILE;**
- **end;**
- **call yolo(@ho);**
- **set @que = concat('update corruption set highest = greatest(', @ho, ');');**
- **select @que;**
- **prepare blyyet from @que;**
- **execute blyyet;**
- 
-



# Création d'une nouvelle table

- Il serait pas mal de se doter d'une table agrégeant les valeurs par continent. Nous allons pour cela devoir la créer
- 
- **CREATE TABLE** nom\_table(
  - colonne1 type\_donnée NULL,
  - colonne2 type\_donnée NOT NULL,
  - colonne3 type\_donnée)

# Création d'une nouvelle table

- Créez une table « continent » dont le **DESCRIBE** donnerait le résultat suivant

•	+-----+-----+-----+-----+-----+-----+
•	Field   Type   Null   Key   Default   Extra
•	+-----+-----+-----+-----+-----+-----+
•	world_region   varchar(50)   NO     NULL
•	total_corruption   float   YES     NULL
•	total_population   int(11)   YES     NULL
•	mean_MPI   float   YES     NULL
•	+-----+-----+-----+-----+-----+-----+
•	

# Création d'une nouvelle table

- +-----+-----+-----+-----+-----+-----+
- | Field | Type | Null | Key | Default | Extra |
- +-----+-----+-----+-----+-----+-----+
- | world\_region | varchar(50) | NO | | NULL | |
- | total\_corruption | float | YES | | NULL | |
- | total\_population | int(11) | YES | | NULL | |
- | mean\_MPI | float | YES | | NULL | |
- +-----+-----+-----+-----+-----+-----+
- 
- Ajoutez les données en question (vous allez avoir besoin de MPI\_subnational)

# Création d'une nouvelle table

- **INSERT INTO continent(world\_region) (SELECT DISTINCT(world\_region) FROM MPI\_subnational);**

# Création d'une nouvelle table

**UPDATE continent**

- **SET continent.total\_corruption = (**
- **SELECT SUM(c.year\_2015) FROM corruption AS c**
- **INNER JOIN MPI\_subnational AS s**
- **ON c.country\_code=s.country\_code**
- **WHERE s.world\_region=continent.world\_region**
- **);**

# Triggers

- **Il serait bon que les données agrégées dans « continent » soient recalculées à chaque ajout/suppression de pays dans les autres tables. Nous allons pour cela avoir besoin de Triggers**

# Les Triggers

- Syntaxe générale
- **DELIMITER //**
- **CREATE TRIGGER** nom moment événement
- **ON** nom\_table
- **FOR EACH ROW**
- **BEGIN**
- ...
- **END; //**
- **DELIMITER ;**
- 

## Exemple particulier

- **DELIMITER //**
- **CREATE TRIGGER** trigger\_test **BEFORE UPDATE ON** MPI\_national
- **FOR EACH ROW**
- **BEGIN**
- **set new.updated = NOW() ;**
- **END; //**
- **DELIMITER ;**

# Les Triggers

À chaque UPDATE sur  
MPI\_national :

- .....  
• .....  
• .....  
• La colonne updated sera actualisée pour la ligne correspondante  
•

## Exemple particulier

- DELIMITER //
- **CREATE TRIGGER** trigger\_test  
BEFORE UPDATE ON  
MPI\_national
- FOR EACH ROW
- **BEGIN**
- set new.updated = NOW() ;
- **END**;//
- DELIMITER ;



# Les Triggers

- **A vous. Créez ces 3 triggers :**
- **- sur population : un trigger qui ajoute country en cas d'ajout sur MPI\_national**
- **- sur continent : un trigger qui actualise le nombre de pays par continent (colonne count crée au préalable) lorsqu'un nouveau est introduit sur MPI\_subnational**
- **- sur continent : un trigger qui actualise la population totale en cas d'ajout d'une nouvelle colonne year\_xxxx dans population**

•

# Les Triggers e1

- **DELIMITER //**
- **CREATE TRIGGER trigger\_population AFTER INSERT ON MPI\_national**
- **FOR EACH ROW**
- **BEGIN**
- **INSERT INTO population(country, country\_code)**  
**VALUES(NEW.country, NEW.iso);**
- **END; //**
- **DELIMITER ;**

# Les Triggers e2

- DELIMITER //
- **CREATE TRIGGER** trigger\_continent AFTER  
INSERT ON MPI\_subnational
- FOR EACH ROW
- **BEGIN**
- UPDATE continent INNER JOIN MPI\_subnational ON  
MPI\_subnational.world\_region = continent.world\_region
- SET continent.count = (SELECT count(country) FROM MPI\_subnational  
WHERE MPI\_subnational.world\_region = continent.world\_region);
- **END;** //
- DELIMITER ;

# Les Triggers e3

- DELIMITER //
- **CREATE TRIGGER** trigger\_continent AFTER  
INSERT ON MPI\_subnational
- FOR EACH ROW
- **BEGIN**
- UPDATE continent INNER JOIN MPI\_subnational ON  
MPI\_subnational.world\_region = continent.world\_region
- SET continent.count = (SELECT count(country) FROM MPI\_subnational  
WHERE MPI\_subnational.world\_region = continent.world\_region);
- **END;** //
- DELIMITER ;

# Un peu plus loin : les Procédures

- **Les procédures sont des sortes de scripts appelés / exécutés par un CALL**
- **Pourquoi recourir à ce type de scripts internes au SGBD ?**
- **- moins de trafic réseau (de multiples requêtes peuvent être exécutées simultanément)**
- **- non dépendante d'un langage tiers et de l'évolution de ses librairies, méthodes, etc etc**
- **- permanence, immuabilité**
-

# Un peu plus loin : les Procédures

- Syntaxe générale :
- Très proche de celle des triggers
- Que puis-je mettre entre **BEGIN** et **END** ?
- - loops (**WHILE**, **LOOP**)
- - variables (@)
- - requêtes
- **DELIMITER \$\$**
- **CREATE PROCEDURE**  
nom\_procedure(IN input  
type OUT output type)
- **BEGIN**
- .....
- **END\$\$**
- **DELIMITER ;**

# Un peu plus loin : les Procédures

- **DELIMITER \$\$**
- **CREATE PROCEDURE**  
**GetCountryInfo(IN c**  
**VARCHAR(25))**
- **BEGIN**
- **.....**
- **END\$\$**
- **DELIMITER ;**
- Créons une procédure assez simple renvoyant
- l'ensemble des données d'un pays passé en argument
- Pour l'appeler
- **CALL**  
**GetCountryInfo('Armenia'**  
**) ;**

# Un exemple fictif

- Vous assistez une équipe de recherche qui travaillait « salement » avec un tableur sur le NAS du labo dans sa transition vers une BDD avec interface web
- Vous commencerez par créer trois tables :
  - - users : login, mdp, emails, etc pour l'accès à la base
  - - patients : table restreinte aux users (vous aurez besoin de LOAD DATA INFILE) (<https://www.kaggle.com/jboysen/mri-and-alzheimers>)
  - - basic\_stats



# Un exemple fictif

- **Fonctionnalité de l'app pour l'utilisateur loggé**
- **- ajouter de nouveaux patients dans patients à l'aide de formulaires**
- **- afficher une liste des patients (ID)**
- **- afficher une proportion actualisée de la distribution H/F (calculée par triggers et stockée dans la table basic\_stats)**
- **++... en utilisant des procédures stockées**

# Optimisations diverses

# La recherche conditionnelle à travers deux tables

On voit parfois des choses de ce genre :

```
SELECT id1 FROM t1 where id1 =  
( SELECT id2 from t2 where critere=x )
```

quel est le soucis ?

# La recherche conditionnelle à travers deux tables

Utilisez des joins :

```
SELECT t1.id1 from t1,t2  
WHERE t1.id1 = t2.id2  
AND critere=x
```

# Plusieurs tables valent mieux qu'une seule grosse table

# Ressources additionnelles

## Challenges

- <https://www.hackerrank.com/domains/sql>