Conception des bases de données III : Droits, optimisations et

bdd3.pdf

utc Formation

Stéphane Crozat

Publiée le 16 février 2020

Table des matières

Vues et gestion des droits

A. Cours

1. Notion de schéma externe et de vue

a) Exercice : Problème de vue

*Soit la vue V suivante, sélectionner **toutes** les assertions correctes.*

```
1 CREATE VIEW V (n, p) AS SELECT nom, prenom FROM T ;
```

- ☐ Le contenu de la table T est calculé dynamiquement à partir de la vue V.
- ☐ Le contenu de la table T est stocké directement dans la base de données.
- ☐ Le contenu de la vue V est calculé dynamiquement à partir de la table T.
- ☐ Le contenu de la vue V est stocké directement dans la base de données.
- ☐ La requête `SELECT n FROM V` est valide.
- ☐ La requête `SELECT nom FROM V` est valide.
- ☐ L'instruction `CREATE VIEW V2 (n) AS SELECT n FROM V` est valide.

b) Schéma externe

L'architecture ANSI/SPARC est à l'origine de la conception des SGBDR, elle postule deux principes fondamentaux : la séparation logique/physique et la notion de schéma externe.

Rappel : La séparation du niveau logique et du niveau physique

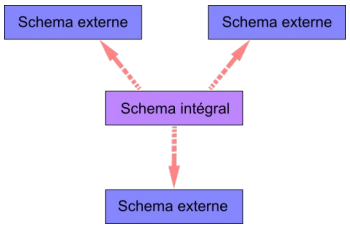
L'utilisateur du SGBDR ne se préoccupe pas de la façon dont celui-ci est implémenté, il raisonne directement en relationnel grâce au langage SQL.

Notion de schéma externe

Le schéma relationnel de la base de données intègre **toutes** les données que la base gère pour **toutes** les applications d'une organisation. Or chaque application n'utilise en général qu'une partie de ces données.

On souhaite que chaque application ne **voit** que la partie des données qui la concerne :

- pour des raisons de simplicité (la complexité globale lui est masquée)
- pour des raisons de sécurité (on ne souhaite pas qu'une application accède à des données qui ne la concerne pas).



Schémas interne et externes

Définition

On appelle schéma externe un sous-ensemble du schéma intégral, destiné à une utilisation spécifique de la base de données.

Dans les SGBDR les schémas externes sont implémentés par des vues.

c) Création de vues en SQL (CREATE VIEW)

Définition : Vue

Une vue est une définition logique d'une relation, sans stockage de données, obtenue par interrogation d'une ou plusieurs tables de la BD. Une vue peut donc être perçue comme une fenêtre dynamique sur les données, ou encore une requête stockée (mais dont seule la définition est stockée, pas le résultat, qui reste calculé dynamiquement).

Une vue permet d'implémenter le concept de schéma externe d'un modèle conceptuel.

Synonymes : Relation dérivée, Table virtuelle calculée

Syntaxe

```
1  CREATE VIEW <nom de vue> <nom des colonnes>
2  AS <spécification de question>
```

La spécification d'une question se fait en utilisant le LMD.

Le nombre de colonnes nommées doit être égal au nombre de colonnes renvoyées par la question spécifiée. Le nom des colonnes est optionnel, s'il n'est pas spécifié, c'est le nom des colonnes telle qu'elles sont renvoyées par la question, qui sera utilisé.

Exemple

```
1  CREATE VIEW Employe (Id, Nom)
2  AS
3    SELECT N°SS, Nom
```

```
4    FROM Personne
```

La vue Employe est ici une projection de la relation Personne sur les attributs N°SS et Nom, renommés respectivement Id et Nom.

Remarque : Vue en lecture et vue en écriture

Une vue est toujours disponible en lecture, à condition que l'utilisateur ait les droits spécifiés grâce au LCD. Une vue peut également être disponible en écriture dans certains cas, que l'on peut restreindre aux cas où la question ne porte que sur une seule table (même si dans certains cas, il est possible de modifier une vue issue de plusieurs tables).

Dans le cas où une vue est destinée à être utilisée pour modifier des données, il est possible d'ajouter la clause "WITH CHECK OPTION" après la spécification de question, pour préciser que les données modifiées ou ajoutées doivent effectivement appartenir à la vue.

Remarque : Vue sur une vue

Une vue peut avoir comme source une autre vue.

Rappel : Vues et héritage

Les vues sont particulièrement utiles pour restituer les relations d'héritage perdues lors de la transformation MCD vers MLD.

2. Passage UML-Relationnel : Expression des vues pour l'héritage et les méthodes

Objectifs

Savoir ajouter des vues pour améliorer l'expression de l'héritage en relationnel.

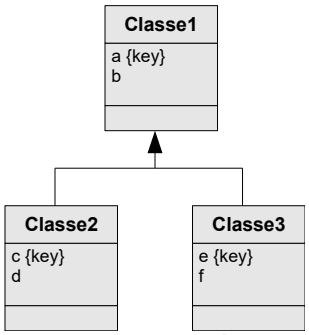
a) Héritage par une référence et vues

Rappel

Transformation de la relation d'héritage par référence

Méthode

Une vue est créée pour chaque classe fille en réalisant une jointure avec la classe mère.



Graphique 1 Héritage

```
Classe1(#a,b)
Classe2(#a=>Classe1,c,d) avec c KEY
Classe3(#a=>Classe1,e,f) avec e KEY
vClasse2=jointure(Classe1,Classe2,a=a)
vClasse3=jointure(Classe1,Classe3,a=a)
```

Exemple

Soit la classe A avec la clé K et les attributs A1 et A2. Soit la classe B, classe fille de A, comprenant la clé K' et les attributs B1 et B2.

Le modèle relationnel correspondant selon cette transformation est :

```
1  A (#K, A1, A2)
2  B (#K=>A, K', B1, B2)
3  vB = Jointure (A, B, A.K=B.K)
```

Complément : Algèbre relationnel

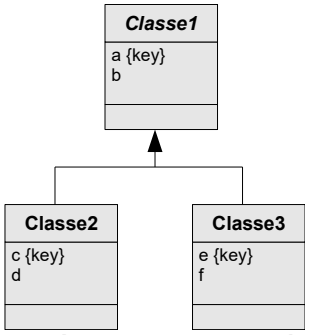
Jointure

b) Héritage par les classes filles et vues

Rappel

Transformation de la relation d'héritage par les classes filles

Méthode : Héritage absorbé par les classes filles (classe mère abstraite)

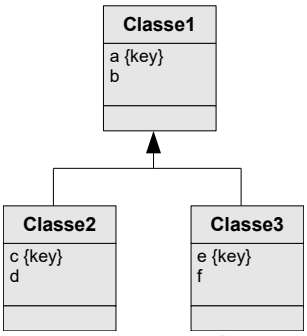


Graphique 2 Héritage (classe mère abstraite)

```
Classe2(#a,b,c,d) avec c KEY
Classe3(#a,b,e,f) avec e KEY
```

```
vClasse1=Union(Projection(Classe2,a,b),Projection(Classe3,a,b))
```

Méthode : Héritage absorbé par les classes filles (classe mère non abstraite)



Graphique 3 Héritage

```
Classe1(#a,b)
Classe2(#a,b,c,d) avec c KEY
Classe3(#a,b,e,f) avec e KEY
vClasse1=Union(Union(Classe1,Projection(Classe2,a,b)),Projection(Class
e3,a,b))
```

Exemple : Héritage absorbé par les classes filles

Soit la classe abstraite A avec la clé K et les attributs A1 et A2. Soit la classe B, classe fille de A avec les attributs B1 et B2. Soit la classe C, classe fille de A avec les attributs C1 et C2.

Le modèle relationnel correspondant selon cette transformation est :

```
1 B (#K, A1, A2, B1, B2)
2 C (#K, A1, A2, C1, C2)
3 vA = Union (Projection (B, K, A1, A2), Projection (C, K, A1, A2))
```

Complément : Algèbre relationnel

Opérateurs ensemblistes

Projection

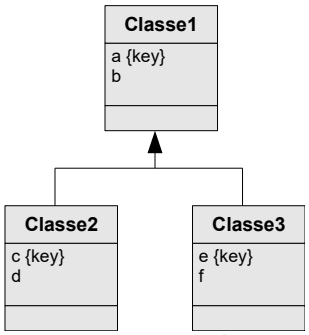
c) Héritage par la classe mère et vues

Rappel

Transformation de la relation d'héritage par la classe mère

Méthode

Chaque classe est représentée par une vue qui restreint aux tuples de la relation correspondants et les projette sur les attributs correspondants.



Graphique 4 Héritage

```
Classe1(#a,b,c,d,e,f,t:{1,2,3}) avec c UNIQUE et e UNIQUE
vClasse1=projection(restriction(Classe1,t=1),a,b)
vClasse2=projection(restriction(Classe1,t=2),a,b,c,d)
vClasse3=projection(restriction(Classe1,t=3),a,b,e,f)
```

Exemple : Héritage absorbé par la classe mère

Soit la classe A abstraite avec la clé K et les attributs A1 et A2. Soit la classe B, classe fille de A avec les attributs B1 et B2. Soit la classe C, classe fille de A avec les attributs C1 et C.

Le modèle relationnel correspondant selon cette transformation est :

```
1  A (#K, A1, A2, B1, B2, C1, C2, T:{'B','C'})
2  vB = Projection (Restriction (A, T='B'), K, A1, A2, B1, B2)
3  vC = Projection (Restriction (A, T='C'), K, A1, A2, C1, C2)
```

Complément : Algèbre relationnel

Projection

Restriction

d) Transformation des méthodes par des vues

Méthode

Lorsqu'une méthode est spécifiée sur un diagramme UML pour une classe C, si cette méthode est une fonction relationnelle (elle renvoie une unique valeur et elle peut être résolue par une requête SQL), alors on crée une vue qui reprend les attributs de la classe C et ajoute des colonnes calculées pour les méthodes.

Remarque : Attributs dérivés

Les attributs dérivés étant apparentés à des méthodes, ils peuvent également être gérés par des vues.

e) Évaluation des enseignants

A la fin du semestre, chaque enseignant est évalué par les étudiants pour chacune de ses UV. Chaque *note* correspond à une UV assurée par cet enseignant, et est égale à la moyenne des *évaluations* attribuées par les étudiants de l'UV. Dans le relevé de note apparaît une *appréciation* générale sur l'enseignant. Cette appréciation est la moyenne de toutes les notes de l'enseignant pour toutes ses UV. La figure suivante illustre le diagramme de classe et le modèle relationnel associé, qui modélisent l' « évaluation des enseignants ».

Les notes sont sur 20.

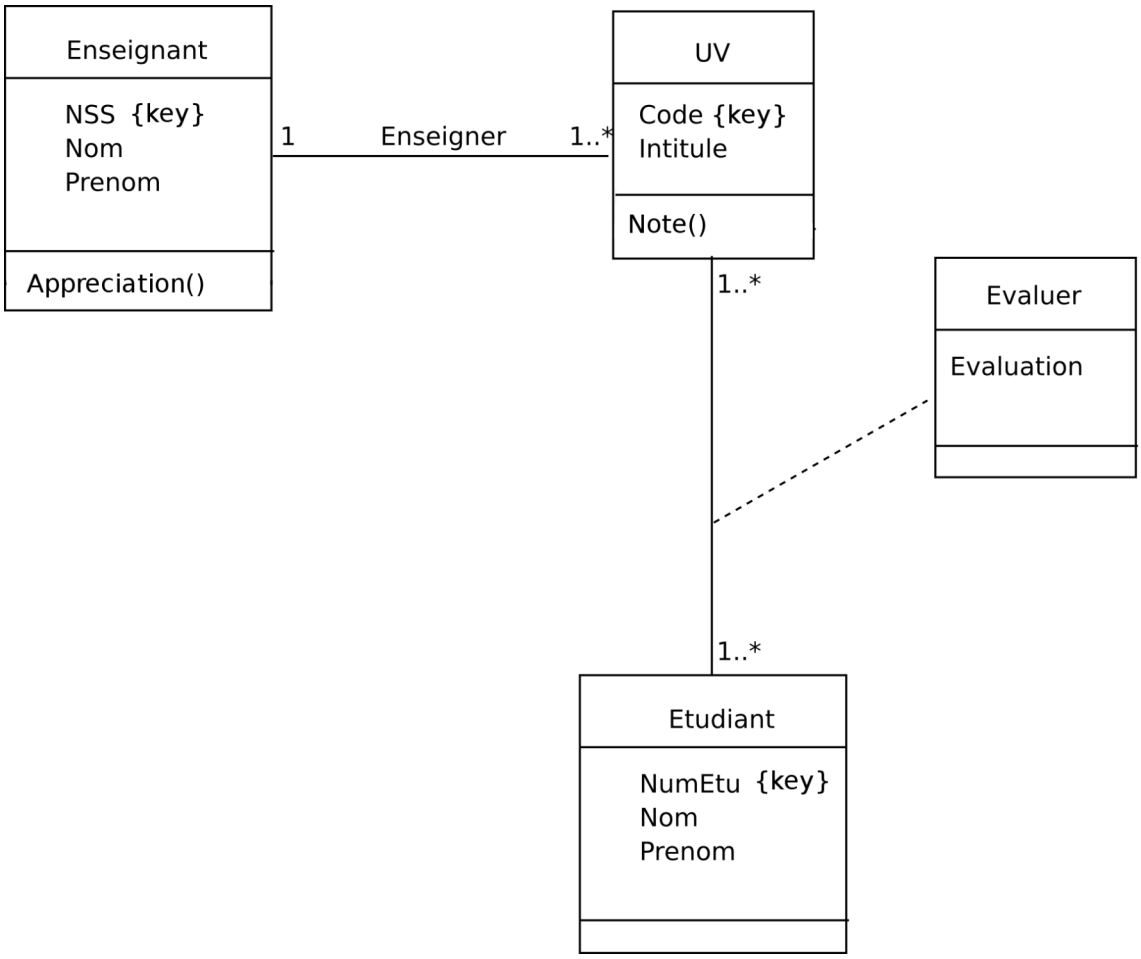


Image 1 Diagramme de classe

```
1  Enseignant(#NSS:char(13), Nom:varchar, Prenom:varchar)
2  UV(#Code:char(5), Intitulé:varchar, Prof=>Enseignant) avec Initulé
   KEY
3  Etudiant(#NumEtu:char(20), Nom:varchar, Prenom:varchar)
4  Evaluer(#NumEtu=>Etudiant, #uv=>UV, Evaluation:[0..20])
```

Question 1

Écrire la vue SQL qui permet de calculer les notes de chaque UV et donnant le résultat ci-dessous : méthode UV.Note()

NOM	PRENOM	INTITULE	NOTE
Dupont	Jean	Science de la Terre	15
Dupont	Jean	Informatique	7.5
Durand	Paul	Français	5

Tableau 1 Notes pour chaque UV

Question 2

Écrire la vue SQL qui permet d'afficher les enseignants avec leur appréciation générale : méthode Enseignant.Appreciation()

NOM	PRENOM	APPRECIATION
Dupont	Jean	11,25
Durand	Paul	5

Appréciation pour chaque enseignant

3. Le Langage de Contrôle de Données de SQL

Objectifs

Maîtriser les bases du SQL pour attribuer et révoquer des droits sur des objets d'une base de données.

Le LCD permet de créer les utilisateurs et de définir leurs droits sur les objets de la BD de façon déclarative. Il permet notamment l'attribution et la révocation de droits à des utilisateurs, sur l'ensemble des bases du SGBD, sur une BD en particulier, sur des relations d'une BD, voire sur certains attributs seulement d'une relation.

a) Exercice : Les droits Lambda

Quelles sont les instructions SQL (sous-ensemble LMD) autorisées pour l'utilisateur "Lambda", étant données les instructions SQL (sous-ensemble LCD) suivantes, exécutées antérieurement ?

```
1  REVOKE ALL PRIVILEGES ON * FROM Lambda;
2  GRANT UPDATE, SELECT ON a TO Lambda;
3  GRANT SELECT ON b TO Lambda;
```

- ☐
```
SELECT a.x, b.y
FROM a,b
WHERE a.x=b.x
```
- ☐
```
UPDATE b
SET y='y'
WHERE y='x'
```
- ☐
```
UPDATE a
SET x='x'
WHERE x='y'
```
- ☐
```
CREATE TABLE c (
x CHAR(50),
y CHAR(50))
```
- ☐
```
SELECT a.x, c.y
FROM a,c
WHERE a.x=c.x
```
- ☐
```
INSERT INTO a (x)
VALUES ('y')
```

b) Attribution de droits

SQL propose une commande pour attribuer des droits à des utilisateurs sur des tables.

Syntaxe

```
1  GRANT <liste de droits> ON <nom table> TO <utilisateur> [WITH GRANT
   OPTION]
```

Les droits disponibles renvoient directement aux instructions SQL que l'utilisateur peut exécuter :

- SELECT
- INSERT
- DELETE
- UPDATE
- ALTER

De plus il est possible de spécifier le droit ALL PRIVILEGES qui donne tous les droits à l'utilisateur (sauf celui de transmettre ses droits).

La clause WITH GRANT OPTION est optionnelle, elle permet de préciser que l'utilisateur a le droit de transférer ses propres droits sur la table à d'autres utilisateur. Une telle clause permet une gestion décentralisée de l'attribution des droits et non reposant uniquement dans les mains d'un administrateur unique.

La spécification PUBLIC à la place d'un nom d'utilisateur permet de donner les droits spécifiés à tous les utilisateurs de la BD.

Exemple

```
1  GRANT SELECT, UPDATE ON Personne TO Pierre;
2  GRANT ALL PRIVILEGES ON Adresse TO PUBLIC;
```

Remarque : Droits sur une vue

Il est possible de spécifier des droits sur des vues plutôt que sur des tables, avec une syntaxe identique (et un nom de vue à la place d'un nom de table).

Remarque : Catalogue de données

Les droits sont stockés dans le catalogue des données, il est généralement possible de modifier directement ce catalogue à la place d'utiliser la commande GRANT. Cela reste néanmoins déconseillé.

Remarque : Création des utilisateurs

Les modalités de création d'utilisateurs, voire de groupes d'utilisateurs dans le SGBD, reste dépendantes de celui-ci. Des commande SQL peuvent être disponibles, telles que CREATE USER, ou bien la commande GRANT lorsque qu'elle porte sur un utilisateur non existant peut être chargée de créer cet utilisateur. Des modules spécifiques d'administration sont généralement disponibles pour prendre en charge la gestion des utilisateurs.

c) Révocation de droits

SQL propose une commande pour révoquer les droits attribués à des utilisateurs.

Syntaxe

```
1 REVOKE <liste de droits> ON <nom table> FROM <utilisateur>
```

Exemple

```
1 REVOKE SELECT, UPDATE ON Personne FROM Pierre;
2 REVOKE ALL PRIVILEGES ON Adresse FROM PUBLIC;
```

Remarque : Révocation du droit de donner les droits

Pour retirer les droits de donner les droits à un utilisateur (qui l'a donc obtenu par la clause WITH GRANT OPTION), il faut utiliser la valeur GRANT OPTION dans la liste des droits révoqués.

Remarque : Révocation en cascade

Lorsque qu'un droit est supprimé pour un utilisateur, il l'est également pour tous les utilisateurs qui avait obtenu ce même droit par l'utilisateur en question.

d) Création d'utilisateurs

Le standard SQL laisse la gestion des utilisateurs à la discrétion du SGBD.

Néanmoins le commande `CREATE USER` est couramment proposée (Oracle, Postgres, ...).

e) The show must go on

[10 minutes]

Soit la base de données suivantes :

```
1 SPECTACLE (#nospectacle:int, nom:str, durée:minutes, type:{théâtre|
  danse|concert})
2 SALLE (#nosalle:int, nbplaces:int)
3 REPRESENTATION (#date:timestamp, #nospectacle=>SPECTACLE,
  #nosalle=>SALLE, prix:decimal)
```

On suppose des classes d'utilisateurs qui ont accès à tout ou partie de ce schéma relationnel :

- Le programmateur qui entre les spectacles dans la base de données,
- Le régisseur qui gère les salles et les représentations,
- Les clients qui peuvent accéder au programme.

Question

Donner les droits associés à chaque classe d'utilisateurs.

B. Exercice

1. Du producteur au consommateur++

[30 min]

Soit la base de données suivante :

```
1  CREATE TABLE  Producteur (
2  raison_sociale VARCHAR (25),
3  ville VARCHAR(255),
4  PRIMARY KEY (raison_sociale)
5  );
6
7  CREATE TABLE Consommateur (
8  login VARCHAR(10),
9  email VARCHAR(50),
10 nom VARCHAR(50) NOT NULL,
11 prenom VARCHAR(50) NOT NULL,
12 ville VARCHAR(255) NOT NULL,
13 PRIMARY KEY (login,email),
14 UNIQUE (nom,prenom,ville)
15 );
16
17 CREATE TABLE Produit (
18 id INTEGER,
19 description VARCHAR(100),
20 produit_par VARCHAR(25) NOT NULL,
21 consomme_par_login VARCHAR(10),
22 consomme_par_email VARCHAR(50),
23 PRIMARY KEY (id),
24 FOREIGN KEY (produit_par) REFERENCES Producteur(raison_sociale),
25 FOREIGN KEY (consomme_par_login,consomme_par_email) REFERENCES
   Consommateur(login,email)
26 );
```

Question 1

Établissez les instructions LCD permettant d'attribuer :

- les droits en lecture seule pour tous les utilisateurs pour la table `Produit`
- les droits en lecture et en écriture pour l'utilisateur `Admin` sur toutes les tables.

Question 2

Afin d'alimenter une application de suivi nommée *Big Brother* écrivez les trois vues SQL LMD permettant de connaître :

- Les produits produits et consommés dans la même ville
- Les produits qui ne sont pas consommés
- Le nombre de produits produits par chaque producteur

Établissez un schéma externe limité à ces trois vues pour l'utilisateur `BB` (sous lequel se connecte l'application *Big Brother*).

2. Gauloiseries

[30 minutes]

Un vieux druide perdant la mémoire souhaite réaliser une base de données pour se souvenir de la composition de ses potions. Un de ses apprentis ayant suivi un cours sur les bases de données lors de son initiation druidique, il réalise le schéma conceptuel suivant :

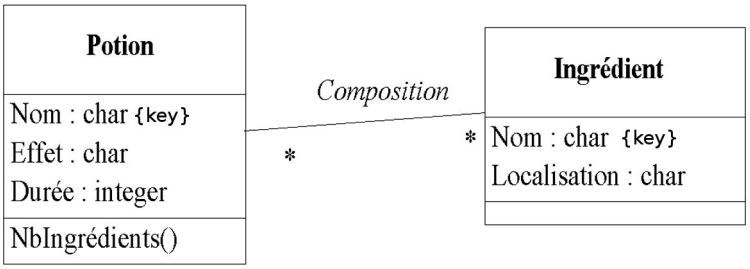


Image 2 Potion : modèle UML

Vous pouvez tester vos requêtes directement sur votre base de données en l'initialisant avec le fichier de données.

```
/**
DROP TABLE Composition ;
DROP TABLE Ingredient ;
DROP TABLE Potion ;
**/

CREATE TABLE Potion (
 Nom VARCHAR PRIMARY KEY,
 Effet VARCHAR,
 Duree INTEGER
);

CREATE TABLE Ingredient (
 Nom VARCHAR PRIMARY KEY,
 Localisation VARCHAR
);

CREATE TABLE Composition (
 NomP VARCHAR REFERENCES Potion(Nom),
 NomI VARCHAR REFERENCES Ingredient(Nom),
 PRIMARY KEY (NomP, NomI)
);

INSERT INTO Potion (Nom, Effet, Duree) VALUES ('Potion Magique','Force', 60);
INSERT INTO Ingredient (Nom, Localisation) VALUES ('Eau', 'Partout');
INSERT INTO Ingredient (Nom, Localisation) VALUES ('Gui', 'Forêt');
INSERT INTO Ingredient (Nom, Localisation) VALUES ('Pomme', 'Pommier');
INSERT INTO Composition (NomP, NomI) VALUES ('Potion Magique', 'Eau');
INSERT INTO Composition (NomP, NomI) VALUES ('Potion Magique', 'Gui');
INSERT INTO Composition (NomP, NomI) VALUES ('Potion Magique', 'Pomme');
INSERT INTO Ingredient (Nom, Localisation) VALUES ('Bière', 'Pic');
INSERT INTO Potion (Nom, Effet, Duree) VALUES ('Potion Inutile','Rien', 0);
INSERT INTO Composition (NomP, NomI) VALUES ('Potion Inutile', 'Eau');
INSERT INTO Potion (Nom, Effet, Duree) VALUES ('Eau Aromatisée','Goût', 10);
INSERT INTO Composition (NomP, NomI) VALUES ('Eau Aromatisée', 'Eau');
```

```
36  INSERT INTO Composition (NomP, NomI) VALUES ('Eau Aromatisée',
    'Gui');
```

Question 1

Réaliser le modèle logique correspondant en relationnel, en faisant apparaître les clés primaires et étrangères (sans clé artificielle).

Question 2

Écrivez une requête SQL permettant de trouver la recette de la potion magique.

Question 3

Qu'a-t-on gagné à ne pas avoir utilisé de clé artificielle dans notre modèle relationnel ?

Question 4

Écrivez une requête SQL permettant de trouver les ingrédients utilisés par aucune potion.

Question 5

Écrivez une vue SQL permettant de préparer l'implémentation de la méthode *NbIngrédients*. Cette vue devra renvoyer le nombre d'ingrédients pour chaque potion.

Question 6

Critiquez le schéma conceptuel de l'apprenti : Est-ce qu'un même ingrédient peut apparaître deux fois dans la même potion ? Peut-on gérer la quantité d'ingrédients pour chaque potion ? Proposer une solution et redessiner le schéma conceptuel en Entité-Association, en y intégrant votre amélioration.

Introduction à l'optimisation des bases de données

II

A. Cours

La conception des SGBDR exige qu'une attention particulière soit portée à la modélisation conceptuelle, afin de parvenir à définir des modèles logiques relationnels cohérents et manipulables. De tels modèles relationnels, grâce au langage standard SQL, présentent la particularité d'être implémentables sur toute plate-forme technologique indépendamment de considérations physiques.

Néanmoins l'on sait que dans la réalité, il est toujours nécessaire de prendre en considération les caractéristiques propres de chaque SGBDR, en particulier afin d'**optimiser** l'implémentation. Les optimisations concernent en particulier la question des **performances**, question centrale dans les applications de bases de données, qui, puisqu'elles manipulent des volumes de données importants, risquent de conduire à des temps de traitement de ces données trop longs par rapport aux besoins d'usage.

Chaque SGBDR propose généralement des mécaniques propres pour optimiser les implémentations, et il est alors nécessaire d'acquérir les compétences particulières propres à ces systèmes pour en maîtriser les arcanes. Il existe néanmoins des principes généraux, que l'on retrouvera dans tous les systèmes, comme par exemple les index, les groupements ou les vues matérialisées. Nous nous proposerons d'aborder rapidement ces solutions pour en examiner les principes dans le cadre de ce cours.

Nous aborderons également quelques techniques de conception, qui consistent à revenir sur la structure proposée par l'étape de modélisation logique, pour établir des modèles de données plus aptes à répondre correctement à des questions de performance. La dénormalisation ou le partitionnement en sont des exemples.

1. Introduction à l'optimisation du schéma interne

Objectifs

Assimiler la problématique de la performance en bases de données

Connaître les grandes classes de solutions technologiques existantes aux problèmes de performance

Connaître et savoir mobiliser les techniques de conception permettant d'optimiser les performances d'une BD

a) Schéma interne et performances des applications

La passage au schéma interne (ou physique), i.e. l'implémentation du schéma logique dans un SGBD particulier, dépend de considérations pratiques liées aux performances des applications.

Les possibilités d'optimisation des schémas internes des BD dépendent essentiellement des fonctions offertes par chaque SGBD.

On peut néanmoins extraire certains principes d'optimisation des schémas internes suffisamment généraux pour être applicables dans la plupart des cas.

Proposition de solutions

Parmi les solutions d'optimisation existantes, on pourra citer :

- L'indexation
- La dénormalisation
- Le partitionnement vertical et horizontal
- Les vues concrètes
- Le regroupement (*clustering*) de tables
- ...

Méthode : Démarche d'optimisation

1. Des problèmes de performance sont identifiés,
2. des solutions d'optimisation sont proposées,
3. les solutions sont évaluées pour vérifier leur impact et leur réponse au problème posé.

b) Évaluation des besoins de performance

Éléments à surveiller

Pour optimiser un schéma interne, il est d'abord nécessaire de repérer les aspects du schéma logique qui sont susceptibles de générer des problèmes de performance.

On pourra citer à titre d'exemple les paramètres à surveiller suivants :

- Taille des tuples
- Nombre de tuples
- Fréquence d'exécution de requêtes
- Complexité des requêtes exécutées (nombre de jointures, etc.)
- Fréquence des mises à jour (variabilité des données)
- Accès concurrents
- Distribution dans la journée des accès

- Qualité de service particulière recherchée
- Paramétrabilité des exécutions
- etc.

Évaluation des coûts

Une fois les éléments de la BD à évaluer repérés, il faut mesurer si oui ou non ils risquent de poser un problème de performance.

L'évaluation des performances peut se faire :

- **Théoriquement**

  En calculant le coût d'une opération (en temps, ressources mémoires, etc.) en fonction de paramètres (volume de données, disparité des données, etc.). En général en BD le nombre de paramètres est très grand, et les calculs de coûts trop complexes pour répondre de façon précise aux questions posées.
- **Empiriquement**

  En réalisant des implémentations de parties de schéma et en les soumettant à des tests de charge, en faisant varier des paramètres. Ce modèle d'évaluation est plus réaliste que le calcul théorique. Il faut néanmoins faire attention à ce que les simplifications d'implémentation faites pour les tests soient sans influence sur ceux-ci.

c) Indexation

Définition : Index

Un index est une structure de données qui permet d'accélérer les recherches dans une table en associant à une clé d'index (la liste des attributs indexés) l'emplacement physique de l'enregistrement sur le disque.

Les accès effectuées sur un index peuvent donc se faire sur des structures optimisées pour la recherche (liste triée, B-tree...) au lieu de se faire par parcours séquentiel et intégral des enregistrements.

Méthode : Cas d'usage des index de type B-Tree

Les index doivent être utilisés sur les tables qui sont fréquemment soumises à des recherches. Ils sont d'autant plus pertinents que les requêtes sélectionnent un petit nombre d'enregistrements (moins de 25% par exemple).

Les index doivent être utilisés sur les attributs :

- souvent mobilisés dans une restriction
- très discriminés (c'est à dire pour lesquels peu d'enregistrements ont les mêmes valeurs)
- rarement modifiés

Attention : Inconvénients des index

- Les index diminuent les performances en mise à jour (puisqu'il faut mettre à jour les index en même temps que les données).
- Les index ajoutent du volume à la base de données et leur volume peut devenir non négligeable.

Syntaxe : Créer un index en SQL

```
1  CREATE INDEX nom_index ON nom_table (NomColonneClé1,
```

```
    NomColonneClé2, ...);
```

Remarque : Index implicites

Les SGBD créent en général un index pour chaque clé (primaire ou candidate). En effet la vérification de la contrainte d'unicité à chaque mise à jour des données justifie à elle seule la présence de l'index.

Attention : Attributs indexés utilisés dans des fonctions

Lorsque les attributs sont utilisés dans des restrictions ou des tris par l'intermédiaire de fonctions, l'index n'est généralement pas pris en compte. L'opération ne sera alors pas optimisée. Ainsi par exemple dans le code suivant, le restriction sur X ne sera pas optimisée même s'il existe un index sur X.

```
1  SELECT *
2  FROM T
3  WHERE ABS(X) > 100
```

Cette non prise en compte de l'index est logique dans la mesure où, on le voit sur l'exemple précédent, une fois l'attribut soumis à une fonction, le classement dans l'index n'a plus forcément de sens (l'ordre des X, n'est pas l'ordre des valeurs absolues de X).

Lorsqu'un soucis d'optimisation se pose, on cherchera alors à sortir les attributs indexés des fonctions.

Notons que certains SGBD, tels qu'Oracle à partir de la version 8i, offrent des instructions d'indexation sur les fonctions qui permettent une gestion de l'optimisation dans ce cas particulier SQL2 SQL3, applications à Oracle [Delmal01].

d) Exercice

Soit les deux tables créées par les instructions suivantes :

```
1  CREATE TABLE T2 (
2    E char(10) Primary Key);
3
4  CREATE TABLE T1 (
5    A number(5) Primary Key,
6    B number(2) Not Null,
7    C char(10) References T2(E),
8    D char(5));
```

Soit une requête dont on cherche à optimiser les performances :

```
1  SELECT A
2  FROM T1, T2
3  WHERE C=E AND Abs(B)>50
4  ORDER BY D;
```

Sachant que tous les attributs sont très discriminés (c'est à dire que les enregistrements contiennent souvent des valeurs différentes les uns par rapport aux autres) et que les deux tables contiennent un grand nombre d'enregistrements, quelles sont les instructions de création d'index qui vont permettre d'optimiser l'exécution de cette requête ?

☐	CREATE INDEX idxA ON T1(A);
☐	CREATE INDEX idxB ON T1(B);
☐	CREATE INDEX idxC ON T1(C);
☐	CREATE INDEX idxD ON T1(D);
☐	CREATE INDEX idxE ON T2(E);

e) Dénormalisation

Rappel

La normalisation est le processus qui permet d'optimiser un modèle logique afin de le rendre non redondant. Ce processus conduit à la fragmentation des données dans plusieurs tables.

Définition : Dénormalisation

Processus consistant à regrouper plusieurs tables liées par des références, en une seule table, en réalisant statiquement les opérations de jointure adéquates.

L'objectif de la dénormalisation est d'améliorer les performances de la BD en recherche sur les tables considérées, en implémentant les jointures plutôt qu'en les calculant.

Remarque : Dénormalisation et redondance

La dénormalisation est par définition facteur de redondance. A ce titre elle doit être utilisée à bon escient et des moyens doivent être mis en œuvre pour contrôler la redondance créée.

Méthode : Quand utiliser la dénormalisation ?

Un schéma doit être dénormalisé lorsque les performances de certaines recherches sont insuffisantes et que cette insuffisance à pour cause des jointures.

Attention : Inconvénients de la dénormalisation

La dénormalisation peut également avoir un effet néfaste sur les performances :

- **En mise à jour**
  Les données redondantes devant être dupliquées plusieurs fois.
- **En contrôle supplémentaire**
  Les moyens de contrôle ajoutés (*triggers*, niveaux applicatifs, etc.) peuvent être très coûteux.
- **En recherche ciblée**
  Certaines recherches portant avant normalisation sur une "petite" table et portant après sur une "grande" table peuvent être moins performantes après qu'avant.

Fondamental : Redondance et bases de données

La redondance volontaire est autorisée dans une base de données à trois conditions :

1. avoir une bonne raison d'introduire de la redondance (améliorer des performances dans le cas de la dénormalisation)
2. documenter la redondance en explicitant les DF responsables de la non 3NF

3. contrôler la redondance par des mécanismes logiciels (triggers par exemple)

f) Les trois principes à respecter pour introduire de la redondance dans une base de données

Fondamental

La redondance volontaire est autorisée dans une base de données à condition de respecter trois principes :

1. il faut avoir **une bonne raison** d'introduire de la redondance (améliorer des performances dans le cas de la dénormalisation)
2. il faut documenter la redondance en **explicitant les DF** responsables de l'absence de 3NF
3. il faut contrôler la redondance par des mécanismes logiciels qui vont **assurer que la redondance n'introduit pas d'incohérence** (*triggers* par exemple)

g) Exercice

Soit les deux tables créées par les instructions suivantes :

```
1  CREATE TABLE T2 (
2    C char(10) Primary Key,
3    E char(10));
4
5  CREATE TABLE T1 (
6    A char(10) Primary Key,
7    B char(10),
8    C char(10) References T2(C),
9    D char(10));
```

Parmi les instructions suivantes qui viendraient remplacer les deux créations précédentes, lesquelles implémenteraient une dénormalisation ?

- ☐
```
CREATE TABLE T3 (
B char(10) Primary Key,
D char(10));
CREATE TABLE T2 (
C char(10) Primary Key,
E char(10));
CREATE TABLE T1 (
A char(10) Primary Key,
B char(10) References T3(B),
C char(10) References T2(C));
```
- ☐
```
CREATE TABLE T1 (
A char(10) Primary Key,
B char(10),
C char(10) Unique Not Null,
D char(10),
E char(10));
```
- ☐
```
CREATE TABLE T1 (
A char(10) Primary Key,
B char(10),
C char(10),
D char(10),
E char(10));
```
- ☐
```
CREATE TABLE T5 (
E char(10) Primary Key);

CREATE TABLE T4 (
D char(10) Primary Key);

CREATE TABLE T3 (
B char(10) Primary Key);

CREATE TABLE T2 (
C char(10) Primary Key,
E char(10) References T5(E));

CREATE TABLE T1 (
A char(10) Primary Key,
B char(10) References T3(B),
C char(10) References T2(C),
D char(10) References T4(D));
```

h) Partitionnement de table

Définition : Partitionnement de table

Le partitionnement d'une table consiste à découper cette table afin qu'elle soit

moins volumineuse, permettant ainsi d'optimiser certains traitements sur cette table.

On distingue :

- Le partitionnement vertical, qui permet de découper une table en plusieurs tables, chacune ne possédant qu'une partie des attributs de la table initiale.
- Le partitionnement horizontal, qui permet de découper une table en plusieurs tables, chacune ne possédant qu'une partie des enregistrements de la table initiale.

Définition : Partitionnement vertical

Le partitionnement vertical est une technique consistant à **implémenter des projections** d'une table T sur des tables T1, T2, etc. en répétant la clé de T dans chaque Ti pour pouvoir recomposer la table initiale par **jointure** sur la clé (Bases de données : objet et relationnel [Gardarin99]).

Remarque

Ce découpage équivaut à considérer l'entité à diviser comme un ensemble d'entités reliées par des associations 1:1.

Méthode : Cas d'usage du partitionnement vertical

Un tel découpage permet d'isoler des attributs peu utilisés d'autres très utilisés, et ainsi améliore les performances lorsque l'on travaille avec les attributs très utilisés (la table étant plus petite).

Cette technique diminue les performances des opérations portant sur des attributs ayant été répartis sur des tables différentes (une opération de jointure étant à présent requise).

Le partitionnement vertical est bien entendu sans intérêt sur les tables comportant peu d'attributs.

Définition : Partitionnement horizontal

Technique consistant à diviser une table T **selon des critères de restriction** en plusieurs tables T1, T2... et de telle façon que tous les tuples de T soit conservés. La table T est alors recomposable par **union** sur les Ti (Bases de données : objet et relationnel [Gardarin99]).

Méthode : Cas d'usage

Un tel découpage permet d'isoler des enregistrements peu utilisés d'autres très utilisés, et ainsi améliore les performances lorsque l'on travaille avec les enregistrements très utilisés (la table étant plus petite). C'est le cas typique de l'archivage. Un autre critère d'usage est le fait que les enregistrements soient toujours utilisés selon un partitionnement donné (par exemple le mois de l'année).

Cette technique diminue les performances des opérations portant sur des enregistrements ayant été répartis sur des tables différentes (une opération d'union étant à présent requise).

Le partitionnement horizontal est bien entendu sans intérêt sur les tables comportant peu d'enregistrements.

i) Exercice

Soit la table suivante contenant des listes de références produits :

```
1  CREATE TABLE Tref (
2    ref1 char(100) Primary Key,
3    ref2 char(100) Unique Not Null,
4    ref3 char(100) Unique Not Null,
5    ref4 char(100) Unique Not Null,
6    ref5 char(100) Unique Not Null,
7    ref6 char(100) Unique Not Null,
8    ref7 char(100) Unique Not Null);
```

On cherche à optimiser la requête suivante ("_" et "%" sont des jokers qui signifient respectivement "1 caractère quelconque" et "0 à N caractères quelconques") :

```
1  SELECT ref1
2  FROM Tref
3  WHERE (ref2 like 'REF42%' or ref2 like 'REF__7%' or ref2 like
   'REF_78%');
```

Quelles sont les opérations de partitionnement qui permettront d'optimiser cette requête de façon maximale (indépendamment des conséquences pour d'éventuelles autres requêtes) ?

- ☐ Un partitionnement horizontal
- ☐ Un partitionnement vertical

j) Vues concrètes

Un moyen de traiter le problème des requêtes dont les temps de calcul sont très longs et les fréquences de mise à jour faible est l'utilisation de vues concrètes.

Définition : Vue concrète

Une vue concrète est un stockage statique (dans une table) d'un résultat de requête.

Un accès à une vue concrète permet donc d'éviter de recalculer la requête et est donc aussi rapide qu'un accès à une table isolée. Il suppose par contre que les données n'ont pas été modifiées (ou que leur modification est sans conséquence) entre le moment où la vue a été calculée et le moment où elle est consultée.

Une vue concrète est généralement recalculée régulièrement soit en fonction d'un événement particulier (une mise à jour par exemple), soit en fonction d'un moment de la journée ou elle n'est pas consultée et où les ressources de calcul sont disponibles (typiquement la nuit).

Synonymes : Vue matérialisée

Complément : CREATE TABLE AS (Postgres)

L'instruction `CREATE TABLE AS SELECT` permet de créer une table à partir d'un résultat de requête sous Postgres.

https://docs.postgresql.fr/current/sql-createtableas.html[1]

Complément : Voir aussi

- *Vue matérialisée sous Oracle*

1 - https://docs.postgresql.fr/current/sql-createtableas.html

- Vue matérialisée sous Postgres
  https://docs.postgresql.fr/current/sql-creatematerializedview.html[2]

k) Complément : Groupement de tables

Définition : Groupement de table

Un groupement ou *cluster* est une structure **physique** utilisée pour stocker des tables sur lesquelles doivent être effectuées de nombreuses requêtes comprenant des opérations de jointure.

Dans un groupement les enregistrements de plusieurs tables ayant une même valeur de champs servant à une jointure (clé du groupement) sont stockées dans un même bloc physique de la mémoire permanente.

Cette technique optimise donc les opérations de jointure, en permettant de remonter les tuples joints par un seul accès disque.

Définition : Clé du groupement

Ensemble d'attributs précisant la jointure que le groupement doit optimiser.

Syntaxe : Syntaxe sous Oracle

Il faut d'abord définir un groupement, ainsi que les colonnes (avec leur domaine), qui constituent sa clé. On peut ainsi ensuite, lors de la création d'une table préciser que certaines de ses colonnes appartiennent au groupement.

```
1  CREATE CLUSTER nom_cluster (NomColonneClé1 type, NomColonneClé2 type,
   ...);
2
3  CREATE TABLE nom_table (...)
4    CLUSTER nom_cluster (Colonne1, Colonne2, ...);
```

Remarque

Le *clustering* diminue les performances des requêtes portant sur chaque table prise de façon isolée, puisque les enregistrements de chaque table sont stockés de façon éclatée.

Remarque : Groupement et dénormalisation

Le groupement est une alternative à la dénormalisation, qui permet d'optimiser les jointures sans créer de redondance.

2. Mécanismes d'optimisation des moteurs de requêtes

Objectifs

Comprendre les principes de fonctionnement du moteur d'optimisation interne aux SGBD

a) Calcul du coût d'une requête

Soit deux relations `t1(a:char(8),b:integer)` et `t2(b:integer)` de respectivement 100 et 10 lignes.

2 - https://docs.postgresql.fr/current/sql-creatematerializedview.html

Soit la requête `SELECT * FROM t1 INNER JOIN t2 ON t1.b=t2.b WHERE t2.b=1`.

- Soit n1 le nombre de tuples tels que b=1 dans t1 ; n1 ∈[0..100]
- Soit n2 le nombre de tuples tels que b=1 dans t2 ; n2 ∈[0..10]
- Soit n3 le nombre de tuples tels que t1.b=t2.b ; : n3 ∈[0..1000]

Question 1

Il existe plusieurs façons de traduire cette requête en algèbre relationnel, proposer quelques exemples.

Question 2

Calculer le nombre d'instructions de chaque exemple d'expression relationnelle :

- en considérant que les restrictions sont effectuées par un parcours intégral dans des listes non triées (comme si c'était effectué avec des boucles de type FOR) ;
- en considérant que chaque opération est effectuée séparément.

Effectuer les calculs pour trois valeurs de n1,n2 et n3 (une valeur minimum, une valeur maximum et une valeur intermédiaire). Donner un intervalle [min..max] indépendant de n1, n2 et n3.

Question 3

Conclure en indiquant quelle opération nécessite le moins d'instructions.

b) Principe de l'optimisation de requêtes

L'exécution d'une requête SQL suit les étapes suivantes :

1. Analyse syntaxique : Vérification de la correction syntaxique et traduction en opérations algébriques
2. Contrôle : Vérification des droits d'accès
3. **Optimisation** : Génération de plans d'exécution et choix du meilleur
4. Exécution : Compilation et exécution du plan choisi

Définition : Plan d'exécution

L'analyse de la requête permet de produire un **arbre d'opérations** à exécuter.

Or il est possible de transformer cet arbre pour en obtenir d'autres équivalents, qui proposent des **moyens différents pour arriver au même résultat**, on parle de différents plans d'exécution.

Définition : Optimisation de la requête

L'optimisation de la requête est une opération informatique visant à réécrire l'arbre d'exécution de la requête en vue de choisir le plan d'exécution le plus performant.

c) Techniques pour l'optimisation de requêtes

Restructuration algébrique

Il existe des propriétés permettant de modifier l'ordre des opérateurs algébriques, afin d'obtenir des gains de performance par exemple :

- Le groupage des restrictions : Il permet d'effectuer plusieurs restrictions en un seul parcours de la table, au lieu d'un par restriction

  `Restriction (Restriction (t,x=a), x=b) <=> Restriction (t,x=a ET x=b)`
- La commutativité des restrictions et Jointures : Elle permet d'appliquer les restrictions avant les jointures

- L'associativité des jointures : Elle permet de changer l'ordre des jointures, afin d'utiliser des algorithmes plus performants dans certains cas
- ...

Heuristiques d'optimisation

Le moteur d'optimisation de la base de données va poser des règles de réécriture pour produire des arbres d'exécutions et choisir les moins coûteux.

Il va typiquement **appliquer d'abord les opérations réductrices** (restrictions et projections).

informations statistiques (modèle de coût)

Une des actions du moteur est d'ordonner les jointures (et les opérations ensemblistes) afin de :

- traiter le moins de tuples possibles ;
- mobiliser les index existants au maximum ;
- utiliser les algorithmes les plus performants.

Pour cela le moteur d'optimisation a besoin de connaître le contenu de la BD : nombre de tuples dans les tables, présence ou non d'index, ...

Le moteur de la BD met à sa disposition des informations statistiques répondant à ces besoins.

d) Collecte de statistiques pour le moteur d'optimisation

Le moteur d'optimisation d'une BD a besoin de collecter des informations sur les tables qu'il a à manipuler afin de choisir les meilleurs plans d'exécution, sur la taille des tables typiquement.

Il est nécessaire de commander au moteur de collecter ces statistiques lorsque des changements de contenu importants ont été effectués dans la BD.

Syntaxe : PostgreSQL : Analyse d'une table

```
1 ANALYSE <table> ;
```

Syntaxe : PostgreSQL : Analyse de toute la base

```
1 ANALYSE;
```

Syntaxe : PostgreSQL : Récupération des espaces inutilisés et analyse de toute la base

```
1 VACUUM ANALYZE ;
```

Complément : Exemple de données collectées

- nombre de lignes
- volume des données
- valeurs moyennes, minimales, maximales
- nombre de valeurs distinctes (indice de dispersion)
- nombre d'occurrences des valeurs les plus fréquentes (notamment pour les colonnes pourvues d'index...)

http://sqlpro.developpez.com/cours/sqlaz/fondements/#L2.1[3]

e) Jointures et optimisation de requêtes

Algorithmes

Il existe plusieurs algorithmes pour réaliser une jointure :

- Jointures par boucles imbriquées (*nested loop*) : chaque tuple de la première table est comparé à chaque tuple de la seconde table.
- Jointure par tri-fusion (*sort-merge*) : les deux tables sont d'abord triées sur l'attribut de jointure, puis elles sont fusionnées.
- Jointure par hachage (*hash join*) : les deux tables sont hachées, puis jointes par fragment.

3. Analyse et optimisation manuelle des requêtes

Objectifs

Savoir afficher et interpréter un plan d'exécution de requête
Savoir écrire des requêtes performantes

a) Analyse de coûts de requêtes (EXPLAIN)

Définition : Plan d'exécution

Le moteur SQL d'une BD peut afficher le plan qu'il prévoit d'exécuter pour une requête donnée, c'est à dire la liste des opérations qui vont être exécutées, ainsi qu'une estimation du coup de ces opérations.

Syntaxe : Syntaxe PostgreSQL

```
1 EXPLAIN <requête SQL>
```

Exemple

Soit la table "taero" des aérodromes de France.

- `SELECT count(*) FROM taero;`
  retourne 421
- `SELECT * FROM taero LIMIT 7;`
  retourne la table ci-après.

	code character(4)	nom character varying(50)
1	LFAB	DIEPPE ST AUBIN
2	LFAC	CALAIS DUNKERQUE
3	LFAD	COMPIEGNE MARGNY
4	LFAE	EU MERS LE TREPORT
5	LFAF	LAON CHAMBRY
6	LFAG	PERONNE SAINT QUENTIN
7	LFAI	NANGIS LES LOGES

Image 3 Sept premières lignes de la table "taero"

3 - http://sqlpro.developpez.com/cours/sqlaz/fondements/#L2.1

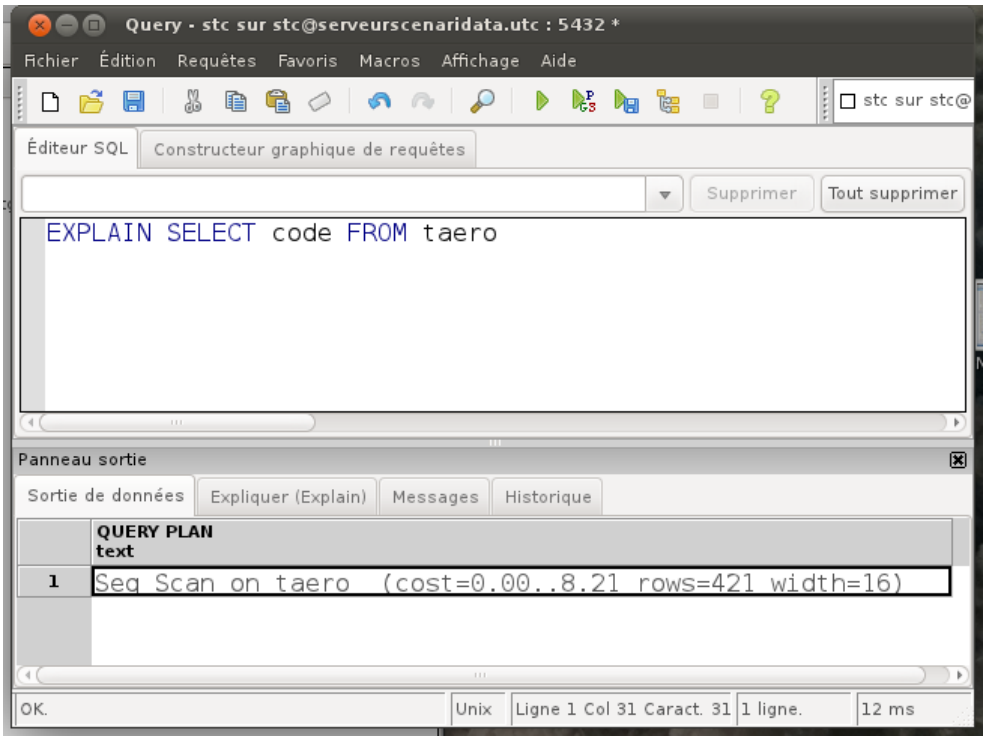


Image 4 EXPLAIN SELECT code FROM taero

`EXPLAIN SELECT code FROM taero` retourne `"Seq Scan on taero (cost=0.00..8.21 rows=421 width=16)"`

- L'opération exécute un `sequential scan` (parcours séquentiel de toute la table)
- Le **coût estimé de démarrage** (coût pour récupérer le premier enregistrement) est de 0.00 et le **coût estimé total** (coût pour récupérer tous les tuples) est de 8.21.

  L'unité est relative aux pages disque accédées, mais elle peut être considérée arbitrairement pour faire des comparaisons.
- Le **nombre estimé de lignes** rapportées est de 421, pour une **taille** de 16 octets chacune.

Définition : Analyse d'exécution

Le moteur SQL d'une BD peut exécuter une requête et afficher le coût réellement observé d'exécution des opérations.

Syntaxe : Syntaxe PostgreSQL

```
1 EXPLAIN ANALYSE <requête SQL>
```

Exemple

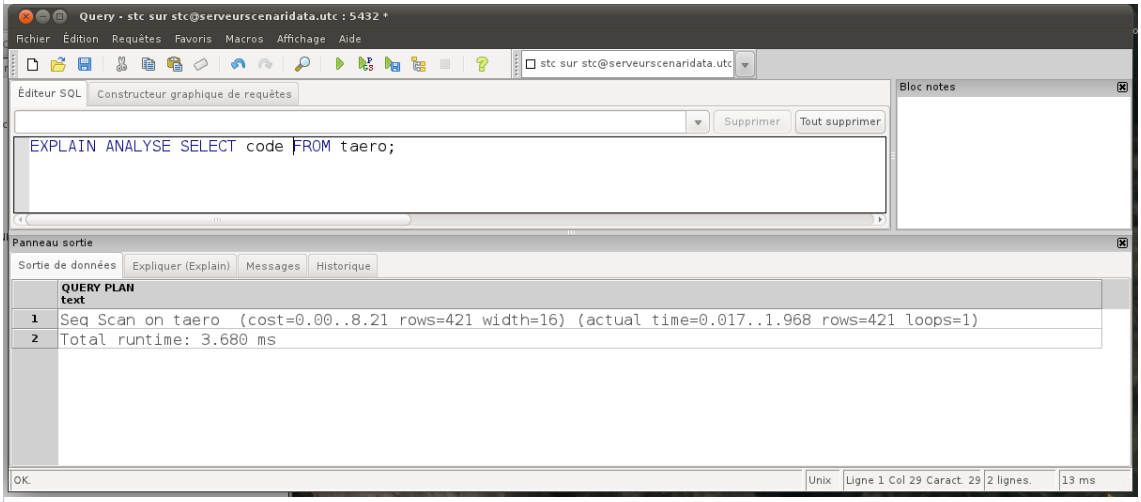


Image 5 EXPLAIN ANALYSE SELECT code FROM taero

`EXPLAIN ANALYSE SELECT code FROM taero` retourne `"Seq Scan on taero (cost=0.00..8.21 rows=421 width=16) (actual time=0.017..1.960 rows=421 loops=1) Total runtime: 3.668 ms"`

- Le **coût réel de démarrage** de l'opération est de 0.017ms et le **coût réel total** est de 1.960ms
- Le **nombre de lignes** réellement rapporté est de 421
- `loops=1` indique l'opération de `Seq Scan` a été exécuté une seule fois
- Le coût total de 3.668ms inclut le temps lié à l'environnement du moteur d'exécution (*start and stop*).

Attention

1. EXPLAIN ne donne qu'une valeur estimée algorithmiquement du coût :
   - cette estimation est **indépendante** de contingences matériel, c'est un coût théorique
   - cette estimation est **reproductible** (chaque exécution donne des informations identiques)
   - **l'instruction SQL n'est pas exécutée**
2. ANALYSE donne un temps de calcul mesuré réellement :
   - ce temps mesuré **dépend** de l'environnement matériel (état du réseau, charge du serveur, ...)
   - ce temps mesuré est **variable** à chaque exécution (en fonction des éléments précédents)
   - **l'instruction SQL est exécutée**
     (on peut insérer la requête à analyser dans une transaction et effectuer un ROLLBACK si l'on souhaite analyser l'exécution d'une requête sans l'exécuter)

Remarque

`EXPLAIN` n'existe pas dans le standard SQL.

Complément : Pour aller plus loin : PostgreSQL

- *http://www.postgresql.org/docs/current/static/sql-explain.html*[4]

4 - http://www.postgresql.org/docs/current/static/sql-explain.html

- *http://www.postgresql.org/docs/current/static/using-explain*[5]
- *http://www.bortzmeyer.org/explain-postgresql.html*[6]

Syntaxe : Syntaxe sous Oracle

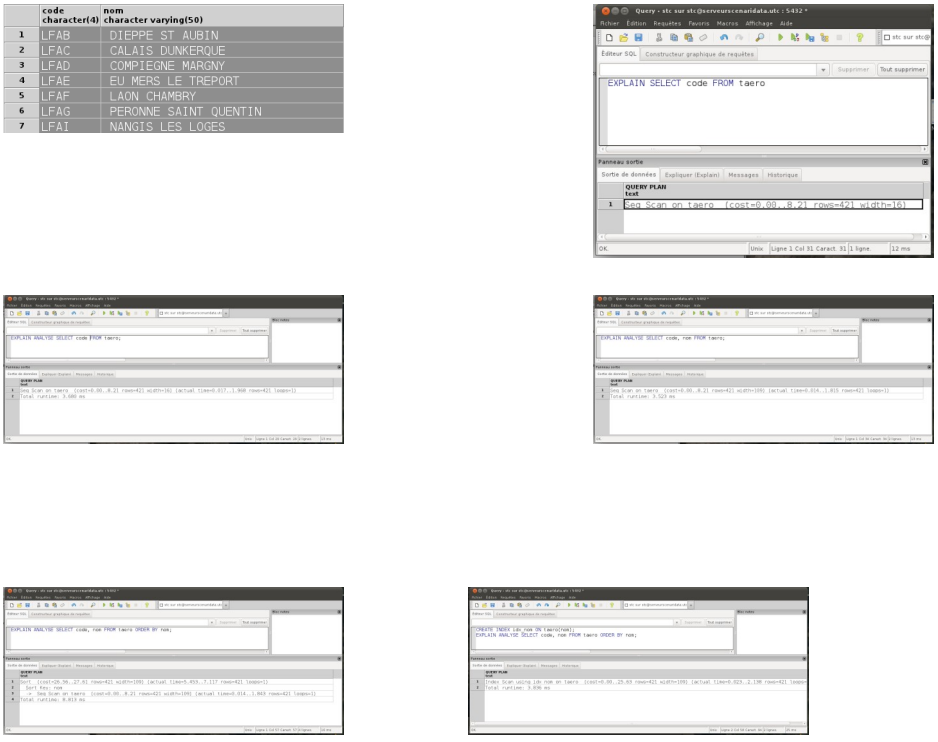
```
1  EXPLAIN PLAN FOR <requête SQL>
```

Il faut préalablement avoir créé ou avoir accès à une table particulière PLAN_TABLE qui sert à stocker les résultats du EXPLAIN.

Complément : Pour aller plus loin : Oracle

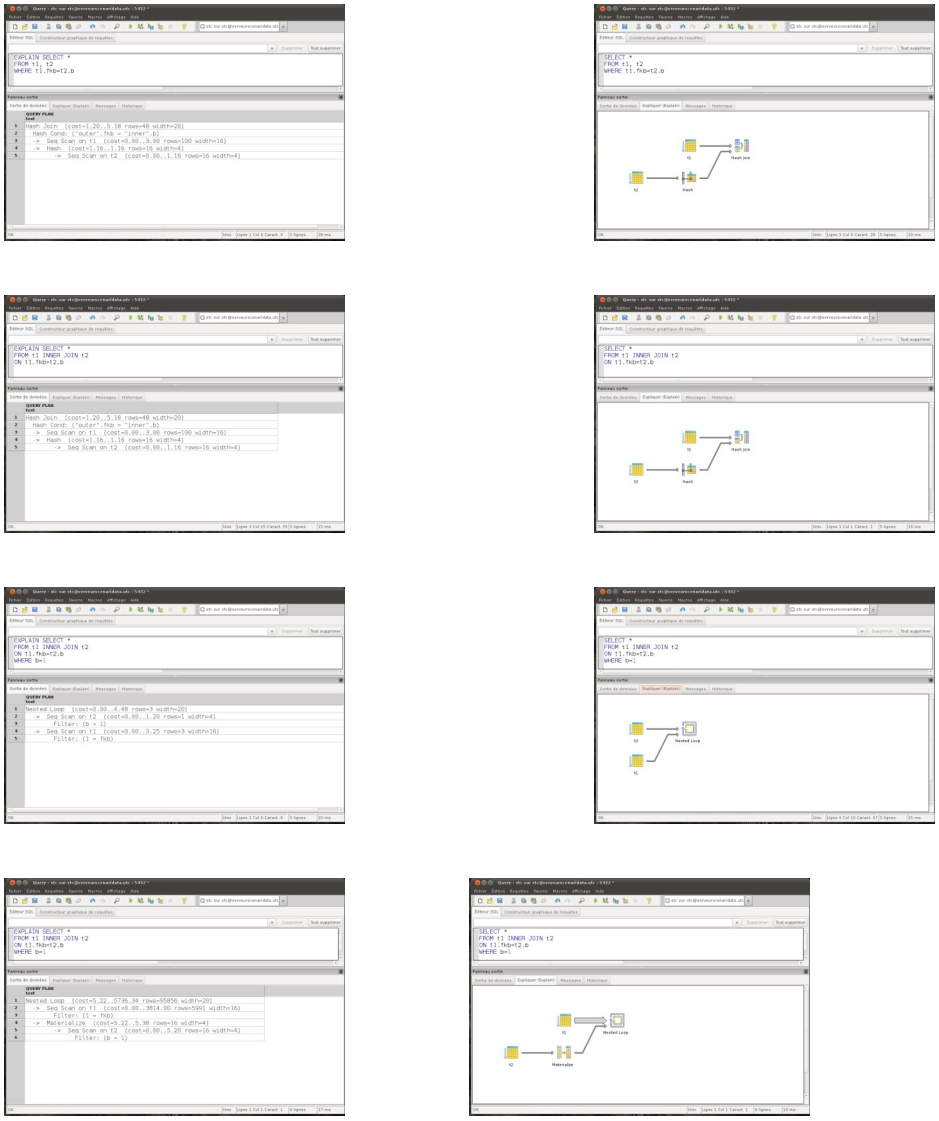
- *http://jpg.developpez.com/oracle/tuning/*[7]

b) Exemple de plans de requêtes



Animation 1 Exemple d'exécution EXPLAIN ANALYSE sur une table

5 - http://www.postgresql.org/docs/current/static/using-explain
6 - http://www.bortzmeyer.org/explain-postgresql.html
7 - http://jpg.developpez.com/oracle/tuning/



Animation 2 Exemple d'exécution EXPLAIN sur une jointure

c) Optimisation manuelle des requêtes

La façon d'écrire des requêtes influe sur l'exécution. L'idée générale est d'écrire des requêtes qui :

- minimisent les informations retournées (par exemple en faisant toutes les projections possibles) ;
- minimisent les traitements (par exemple en faisant toutes les restrictions possibles) ;
- évitent des opérations inutiles (par exemple : `SELECT DISTINCT`, `ORDER BY`...) ;

Complément : Trucs et astuces...

http://sqlpro.developpez.com/cours/optimiser/#L9[8]

8 - http://sqlpro.developpez.com/cours/optimiser/#L9

4. Synthèse : L'optimisation

Optimisation

Modification du schéma interne d'une BD pour en améliorer les performances.

- Techniques au niveau physique
  - Indexation
  - Regroupement physique
  - Vue concrète
- Techniques de modélisation
  - Dénormalisation
  - Partitionnement
    - Horizontal
    - Vertical

5. Bibliographie commentée sur l'optimisation

Complément : Synthèses

SQL2 SQL3, applications à Oracle [Delmal01]

Une bonne description des index (page 79) et clusters (page 81) par l'image. Une description intéressante d'une **fonction** (sous Oracle 8i) qui permet d'indexer des fonctions, ce qui répond à un problème important d'optimisation (page 84). Une description des vues matérialisées, leur implémentation sous Oracle, des exemples d'usage dans le domaine du *datawarehouse*.

Bases de Données Relationnelles et Systèmes de Gestion de Bases de Données Relationnels, le Langage SQL [w_supelec.fr/~yb]

Des informations générales sur les *index*[9] et sur les *clusters*[10].

B. Exercices

1. Film concret

[5 minutes]

Soit le schéma relationnel suivant :

```
1  Film (#isan:char(33), titre:varchar(25), entrees:integer,
   nomReal=>Realisateur(nom), prenomReal=>Realisateur(prenom))
2  Realisateur (#nom:varchar(25), #prenom:varchar(25), ddn:date)
```

Soit la requête suivante portant sur ce schéma implémenté sous PostgreSQL :

```
1  SELECT f.titre AS film, r.ddn AS real
2  FROM Film f, Realisateur r
3  WHERE f.nomReal=r.nom AND f.prenomReal=r.prenom
```

Question

Proposer une optimisation de cette requête sous la forme de la vue matérialisée `vTopFilms`.

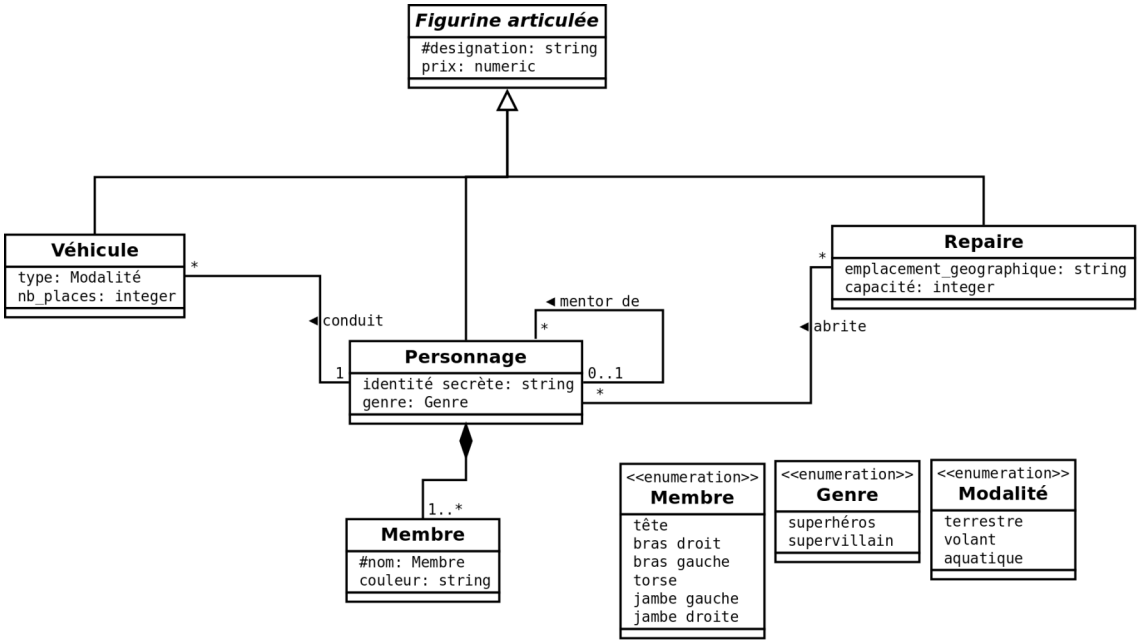
9 - http://wwwsi.supelec.fr/~yb/poly_bd/node122.html
10 - http://wwwsi.supelec.fr/~yb/poly_bd/node126.html

2. Super-lents

[15 minutes]

L'entreprise GARVEL propose des figurines de super-héros à acheter en ligne sur un site Internet adossé à sa base de données. Son catalogue a atteint cette année le million de modèles. Depuis quelques temps, et la récente forte augmentation des modèles au catalogue, les performances des consultations ont beaucoup chuté, entraînant des temps d'accès lents (plusieurs secondes) et une baisse des actes d'achat.

La requête la plus utilisée par l'application Web sert à lister tous les super-héros avec toutes leurs caractéristiques, avec leurs véhicules et leurs repaires (et également toutes leurs caractéristiques) et à la trier par ordre de prix.



Modèle UML Figurines GARVEL

Soyez **concis** et **précis** dans vos réponses ; La bonne mobilisation des concepts du domaine et la clarté de la rédaction seront appréciées.

Question 1

Expliquer pourquoi cette requête peut poser des problèmes de performance, lorsque la base comprend de nombreux enregistrements.

Question 2

Proposer et justifier une première solution d'optimisation à mettre en œuvre qui sera utile dans tous les cas et n'aura que peu d'effets indésirables.

Question 3

Proposer deux solutions alternatives qui, si l'on reste en relationnel, permettraient d'améliorer considérablement les performances. Vous en poserez les avantages, inconvénients et les mesures à prendre pour contenir ces inconvénients.

3. Explications

Fichier CSV des départements français (95 lignes)

dpt_france.csv.zip
Document 1 Fichier des départements français

Extrait des premières lignes du fichier :

```
numero,nom,pop
1,Ain,529
2,Aisne,552
3,Allier,357
4,Alpes-de-Haute-Provence,145
5,Hautes-Alpes,127
6,Alpes-Maritimes,1023
7,Ardèche,295
8,Ardennes,299
9,Ariège,143
```

Fichier CSV des villes françaises (36700 lignes)

villes_france.csv.zip

Document 2 Fichier des communes françaises

Extrait des premières lignes du fichier :

```
codeinsee,departement,nom,pop2010,pop1999
1284,1,Ozan,618,469
1123,1,Cormoranche-sur-Saône,1058,903
1298,1,Plagne,129,83
1422,1,Tossiat,1406,1111
1309,1,Pouillat,88,58
1421,1,Torcieu,698,643
1320,1,Replonges,3500,2841
1119,1,Corcelles,243,222
1288,1,Péron,2143,1578
```

Question 1

Instancier la base de données Postgres permettant de gérer ces fichiers.

VACUUM

Exécutez la commande ci-après, on observe seulement 21300 lignes collectées, là ou 36700 étaient attendues.

```
EXPLAIN
SELECT * FROM ville;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on ville  (cost=0.00..497.00 rows=21300 width=32)
```

Question 2

Expliquez pourquoi et proposez une solution.

Indice :

Collecte de statistiques pour le moteur d'optimisation

Projection

Exécutez la commande ci-après permettant de projeter le nom des villes.

```
EXPLAIN
SELECT nom FROM ville;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on ville  (cost=0.00..651.00 rows=36700 width=12)
```

Question 3

Quelle différence observez-vous avec le plan de la requête `SELECT * FROM ville` ? Expliquez.

Restriction et tri

Exécutez les commande ci-après permettant respectivement de faire une restriction et un tri sur le nom des villes.

```
1  EXPLAIN
2  SELECT nom FROM ville
3  WHERE nom='Compiègne';
```

```
1                        QUERY PLAN
2  -------------------------------------------------------
3   Seq Scan on ville  (cost=0.00..742.75 rows=1 width=12)
4     Filter: ((nom)::text = 'Compiègne'::text)
```

```
1  EXPLAIN
2  SELECT nom FROM ville
3  ORDER BY nom;
```

```
1                              QUERY PLAN
2  ---------------------------------------------------------------------
3   Sort  (cost=3433.50..3525.25 rows=36700 width=12)
4     Sort Key: nom
5     ->  Seq Scan on ville  (cost=0.00..651.00 rows=36700 width=12)
```

Question 4

Qu'observez-vous ? Proposez une solution pour améliorer ces deux requêtes. Mesurez le gain apporté par la solution. Notez les éventuels inconvénient apportés par la solution.

Jointure

Exécutez la commande ci-après permettant de faire une jointure.

```
1  EXPLAIN
2  SELECT v.nom, d.nom
3  FROM ville v JOIN dpt d
4  ON v.departement = d.numero;
```

```
1  Hash Join  (cost=3.14..1136.96 rows=34520 width=22)
2     Hash Cond: (v.departement = d.numero)
3     ->  Seq Scan on ville v  (cost=0.00..651.00 rows=36700 width=16)
4     ->  Hash  (cost=1.95..1.95 rows=95 width=13)
5           ->  Seq Scan on dpt d  (cost=0.00..1.95 rows=95 width=13)
```

Question 5

Indexez la clé étrangère `v.departement`. Observez-vous un gain ? Expliquez.

Question 6

À partir des plans précédents, calculez le gain qu'apporterait une dénormalisation à cette jointure.

III

Gestion des transactions pour la fiabilité et la concurrence

A. Cours

Les transactions sont une réponse générale aux problèmes de fiabilité et d'accès concurrents dans les BD, et en particulier dans les BD en mode client-serveur. Elles sont le fondement de toute implémentation robuste d'une BD. Un SGBDR ne fonctionne nativement qu'en mode transactionnel.

1. Principes des transactions

Objectifs

Comprendre les principes et l'intérêt des transactions

a) Problématique des pannes et de la concurrence

Une BD est un ensemble persistant de données organisées qui a en charge la préservation de la cohérence de ces données. Les données sont cohérentes si elles respectent l'ensemble des contraintes d'intégrité spécifiées pour ces données : contraintes de domaine, intégrité référentielle, dépendances fonctionnelles...

La cohérence des données peut être remise en cause par deux aspects de la vie d'une BD :

- **La défaillance**

  Lorsque le système tombe en panne alors qu'un traitement est en cours, il y a un risque qu'une partie seulement des instructions prévues soit exécutée,

ce qui peut conduire à des incohérences. Par exemple pendant une mise à jour en cascade de clés étrangères suite au changement d'une clé primaire.

- **La concurrence**

  Lorsque deux accès concurrents se font sur les données, il y a un risque que l'un des deux accès rende l'autre incohérent. Par exemple si deux utilisateurs en réseau modifient une donnée au même moment, seule une des deux mises à jour sera effectuée.

La gestion de transactions par un SGBD est à la base des mécanismes qui permettent d'assurer le maintien de la cohérence des BD. C'est à dire encore qu'il assure que toutes les contraintes de la BD seront toujours respectées, même en cas de panne et même au cours d'accès concurrents.

b) Notion de transaction

Définition : Transaction

Une transaction est une unité logique de travail, c'est à dire une séquence d'instructions, dont l'exécution assure le passage de la BD d'un état cohérent à un autre état cohérent.

Fondamental : Cohérence des exécutions incorrectes

La transaction assure le maintien de la cohérence des données que son exécution soit **correcte** ou **incorrecte**.

Exemple : Exemples d'exécutions incorrectes

L'exécution d'une transaction peut être incorrecte parce que :

- Une panne a lieu
- Un accès concurrent pose un problème
- Le programme qui l'exécute en a décidé ainsi

c) Déroulement d'une transaction

1. DEBUT
2. TRAITEMENT
   - Accès aux données en lecture
   - Accès aux données en écriture
3. FIN
   - Correcte : Validation des modifications
   - Incorrecte : Annulation des modifications

Fondamental

Tant qu'une transaction n'a pas été terminée correctement, elle doit être assimilée à une **tentative** ou une mise à jour **virtuelle**, elle reste incertaine. Une fois terminée correctement la transaction ne peut plus être annulée par aucun moyen.

d) Propriétés ACID d'une transaction

Une transaction doit respecter quatre propriétés fondamentales :

- **L'atomicité**

  Les transactions constituent l'unité logique de travail, toute la transaction est exécutée ou bien rien du tout, mais jamais une partie seulement de la transaction.

- **La cohérence**

  Les transactions préservent la cohérence de la BD, c'est à dire qu'elle transforme la BD d'un état cohérent à un autre (sans nécessairement que les états intermédiaires internes de la BD au cours de l'exécution de la transaction respectent cette cohérence)
- **L'isolation**

  Les transactions sont isolées les unes des autres, c'est à dire que leur exécution est indépendante des autres transactions en cours. Elles accèdent donc à la BD comme si elles étaient seules à s'exécuter, avec comme corollaire que les résultats intermédiaires d'une transaction ne sont jamais accessibles aux autres transactions.
- **La durabilité**

  Les transactions assurent que les modifications qu'elles induisent perdurent, même en cas de défaillance du système.

Remarque

Les initiales de Atomicité, Cohérence, Isolation et Durabilité forme le mot mnémotechnique ACID.

e) Journal des transactions

Définition : Journal

Le journal est un fichier système qui constitue un espace de stockage redondant avec la BD. Il répertorie l'ensemble des mises à jour faites sur la BD (en particulier les valeurs des enregistrements avant et après mise à jour). Le journal est donc un historique **persistant** (donc en mémoire secondaire) qui mémorise tout ce qui se passe sur la BD.

Le journal est indispensable pour la validation (COMMIT), l'annulation (ROLLBACK) et la reprise après panne de transactions.

Synonymes : Log

2. Manipulation de transactions en SQL

Objectifs

Connaître les syntaxes SQL standard, PostgreSQL, Oracle et Access pour utiliser des transactions

a) Transactions en SQL

Introduction

Le langage SQL fournit trois instructions pour gérer les transactions.

Syntaxe : Début d'une transaction

```
1 BEGIN TRANSACTION (ou BEGIN) ;
```

Cette syntaxe est optionnelle (voire inconnue de certains SGBD), une transaction étant débutée de façon **implicite** dès qu'instruction est initiée sur la BD.

Syntaxe : Fin correcte d'une transaction

```
1 COMMIT TRANSACTION (ou COMMIT) ;
```

Cette instruction SQL signale la fin d'une transaction couronnée de succès. Elle indique donc au gestionnaire de transaction que l'unité logique de travail s'est terminée dans un état cohérent est que les données peuvent effectivement être modifiées de façon durable.

Syntaxe : Fin incorrecte d'une transaction

```
1 ROLLBACK TRANSACTION (ou ROLLBACK) ;
```

Cette instruction SQL signale la fin d'une transaction pour laquelle quelque chose s'est mal passé. Elle indique donc au gestionnaire de transaction que l'unité logique de travail s'est terminée dans un état potentiellement incohérent et donc que les données ne doivent pas être modifiées en annulant les modifications réalisées au cours de la transaction.

Remarque : Programme

Un programme est généralement une séquence de plusieurs transactions.

b) Mini-TP : Transaction en SQL standard sous PostgreSQL

1. Se connecter à une base de données : `psql mydb`
2. Créer une table `test` : `CREATE TABLE test (a integer);`

Question 1

1. Commencer une transaction : `BEGIN TRANSACTION;`
2. Insérer les deux valeurs 1 et 2 dans la table : `INSERT INTO...`
3. Valider la transaction : `COMMIT`
4. Vérifier que les valeurs sont bien dans la table : `SELECT * FROM ...`

Question 2

1. Commencer une transaction : `BEGIN TRANSACTION;`
2. Insérer les deux valeurs 3 et 4 dans la table : `INSERT INTO...`
3. Annuler la transaction : `ROLLBACK`
4. Vérifier que les valeurs ne sont pas dans la table : `SELECT * FROM ...`

c) Exemple : Transaction sous Oracle en SQL

Exemple : Exemple en SQL sous Oracle

```
1 INSERT INTO test (a) VALUES (1);
2 COMMIT;
3
```

Attention : BEGIN implicite sous Oracle

La commande `BEGIN;` ou `BEGIN TRANSACTION;` ne peut pas être utilisé sous Oracle (la commande `BEGIN` est réservée à l'ouverture d'un bloc PL/SQL).

Toute commande SQL LMD (`INSERT`, `UPDATE` ou `DELETE`) démarre par défaut une transaction, la commande `BEGIN TRANSACTION` est donc implicite.

d) Exemple : Transaction sous Oracle en PL/SQL

Exemple : Exemple en PL/SQL sous Oracle

```
1  BEGIN
2    INSERT INTO test (a) VALUES (1);
3    COMMIT;
4  END;
5
```

e) Exemple : Transaction sous Access en VBA

Exemple

```
1  Sub MyTransaction
2    BeginTrans
3      CurrentDb.CreateQueryDef("", "INSERT INTO test (a) VALUES
   (1)").Execute
4    CommitTrans
5  End Sub
```

Remarque : VBA et transactions

Sous Access, il n'est possible de définir des transactions sur plusieurs objets requêtes qu'en VBA.

f) Mode Autocommit

Attention : Autocommit

La plupart des clients et langages d'accès aux bases de données proposent un mode *autocommit* permettant d'encapsuler chaque instruction dans une transaction. Ce mode revient à avoir un `COMMIT` implicite après chaque instruction.

Ce mode doit être désactivé pour permettre des transactions portant sur plusieurs instructions.

Oracle

- Sous Oracle SQL*Plus : `SET AUTOCOMMIT ON` et `SET AUTOCOMMIT OFF`
- Sous Oracle SQL Developper : Menu `Outils > Préférences > Base de données > Paramètres de feuille de calcul > Validation automatique dans une feuille de calcul`

PostgreSQL

Avec psql :

- `\set AUTOCOMMIT on`
- `\set AUTOCOMMIT off`

Si l'*autocommit* est activé, il est néanmoins possible de démarrer une transaction sur plusieurs lignes en exécutant un `BEGIN TRANSACTION` explicite : `BEGIN ; ... ; COMMIT ;`.

Ainsi deux modes sont possibles :

- *Autocommit* activé : BEGIN explicites, COMMIT implicites
- *Autocommit* désactivé : BEGIN implicites, COMMIT explicites

Access

Sous Access, toute requête portant sur plusieurs lignes d'une table est encapsulée dans une transaction.

Ainsi par exemple la requête `UPDATE Compte SET Solde=Solde*6,55957` est exécutée dans une transaction et donc, soit toutes les lignes de la table Compte seront mises à jour, soit aucune (par exemple en cas de panne pendant .

g) Exercice

Quel est le résultat de l'exécution suivante sous Oracle ?

```
1  SET AUTOCOMMIT OFF
2  CREATE TABLE T1 (A INTEGER);
3
4  INSERT INTO T1 VALUES (1);
5  INSERT INTO T1 VALUES (2);
6  INSERT INTO T1 VALUES (3);
7  COMMIT;
8  INSERT INTO T1 (4);
9  INSERT INTO T1 (5);
10 ROLLBACK;
11 SELECT SUM(A) FROM T1
```

h) Exercice

Combien de lignes renvoie l'avant-dernière instruction SELECT ?

```
1  SET AUTOCOMMIT OFF
2  CREATE TABLE T1 (A INTEGER);
3
4  INSERT INTO T1 VALUES (1);
5  INSERT INTO T1 VALUES (1);
6  INSERT INTO T1 VALUES (1);
7  SELECT * FROM T1;
8  COMMIT;
```

i) Exercice

Combien de tables sont créées par cette série d'instruction ?

```
1  SET AUTOCOMMIT OFF
2  CREATE TABLE T1 (A INTEGER);
3  CREATE TABLE T2 (A INTEGER);
4  CREATE TABLE T3 (A INTEGER);
5  INSERT INTO T1 VALUES (1);
6  INSERT INTO T2 VALUES (1);
7  INSERT INTO T3 VALUES (1);
8  ROLLBACK;
```

3. Fiabilité et transactions

Objectifs

Appréhender la gestion des pannes dans les SGBD.

Comprendre la réponse apportée par la gestion des transactions.

a) Les pannes

Une BD est parfois soumise à des défaillances qui entraînent une perturbation, voire un arrêt, de son fonctionnement.

On peut distinguer deux types de défaillances :

- **Les défaillances système**

  ou défaillances douces (*soft crash*), par exemple une coupure de courant ou une panne réseau. Ces défaillances affectent toutes les transactions en cours de traitement, mais pas la BD au sens de son espace de stockage physique.

- **Les défaillances des supports**

  ou défaillances dures (*hard crash*), typiquement le disque dur sur lequel est stockée la BD. Ces défaillances affectent également les transactions en cours (par rupture des accès aux enregistrements de la BD), mais également les données elles-mêmes.

Remarque : Annulation des transactions non terminées

Lorsque le système redémarre après une défaillance, toutes les transactions qui étaient en cours d'exécution (pas de COMMIT) au moment de la panne sont annulés (ROLLBACK imposé par le système).

Cette annulation assure le retour à un état cohérent, en vertu des propriétés ACID des transactions.

Remarque : Ré-exécution des transactions terminées avec succès

Au moment de la panne certaines transactions étaient peut-être terminées avec succès (COMMIT) mais non encore (ou seulement partiellement) enregistrées dans la BD (en mémoire volatile, tampon, etc.). Lorsque le système redémarre il doit commencer par rejouer ces transactions, qui assurent un état cohérent de la BD plus avancé.

Cette reprise des transactions après COMMIT est indispensable dans la mesure où c'est bien l'instruction COMMIT qui assure la fin de la transaction et donc la **durabilité**. Sans cette gestion, toute transaction pourrait être remise en cause.

Remarque : Unité de reprise

Les transactions sont des unités de travail, et donc également de reprise.

Remarque : Défaillance des supports

Tandis que la gestion de transactions et de journal permet de gérer les défaillances systèmes, les défaillances des supports ne pourront pas toujours être gérés par ces seuls mécanismes.

Il faudra leur adjoindre des procédures de **sauvegarde** et de restauration de la BD pour être capable au pire de revenir dans un état antérieur cohérent et au mieux de réparer complètement la BD (cas de la réplication en temps réel par exemple).

b) Point de contrôle

Définition : Point de contrôle

Un point de contrôle est une écriture dans le journal positionnée automatiquement par le système qui établit la liste de toutes les transactions en cours (au moment où le point de contrôle est posé) et force la sauvegarde des données alors en mémoire centrale dans la mémoire secondaire.

Le point de contrôle est positionné à intervalles de temps ou de nombre d'entrées.

Le dernier point de contrôle est le point de départ d'une reprise après panne, dans la mesure où c'est le dernier instant où toutes les données ont été sauvegardées en mémoire non volatile.

Synonymes : Syncpoint

c) Ecriture en avant du journal

Fondamental

On remarquera que pour que la reprise de panne soit en mesure de rejouer les transactions, la première action que doit effectuer le système au moment du COMMIT est l'écriture dans le journal de cette fin correcte de transaction. En effet ainsi la transaction pourra être rejouée, même si elle n'a pas eu le temps de mettre effectivement à jour les données dans la BD, puisque le journal est en mémoire secondaire.

d) Reprise après panne

Introduction

Le mécanisme de reprise après panne s'appuie sur le journal et en particulier sur l'état des transactions au moment de la panne et sur le dernier point de contrôle.

Le schéma ci-après illustre les cinq cas de figure possibles pour une transaction au moment de la panne.

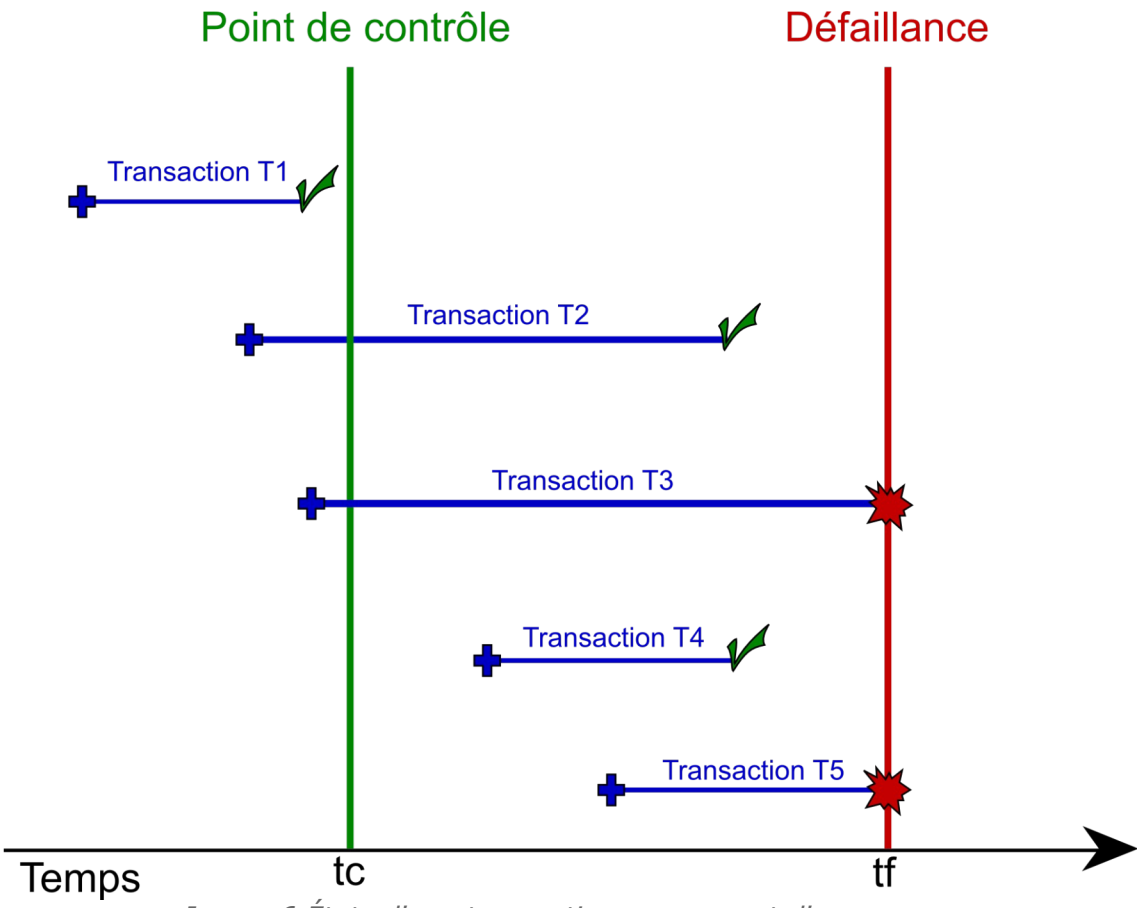


Image 6 États d'une transaction au moment d'une panne

- **Transactions de type T1**
  Elles ont débuté et se sont terminées avant tc. Elles n'interviennent pas dans le processus de reprise.
- **Transactions de type T2**
  Elles ont débuté avant tc et se sont terminées entre tc et tf. Elles devront être rejouées (il n'est pas sûr que les données qu'elles manipulaient aient été correctement inscrites en mémoire centrale, puisque après tc, or le COMMIT impose la durabilité).
- **Transactions de type T3**
  Elles ont débuté avant tc, mais n'était pas terminées à tf. Elles devront être annulées (pas de COMMIT).
- **Transactions de type T4**
  Elles ont débuté après tc et se sont terminées avant tf. Elles devront être rejouées.
- **Transactions de type T5**
  Elles ont débuté après tc et ne se sont pas terminées. Elles devront être annulées.

Remarque

Les transactions sont des unités d'intégrité.

e) Mini-TP : Simulation d'une panne sous PostgreSQL

1. Se connecter à une base de données : `psql mydb`

2. Créer une table `test` : `CREATE TABLE test (a integer);`

Question

1. Commencer une transaction : `BEGIN TRANSACTION;`
2. Insérer les deux valeurs 1 et 2 dans la table : `INSERT INTO...`
3. Vérifier que les valeurs sont bien dans la table : `SELECT * FROM ...`
4. Simuler un crash en fermant le terminal : `ROLLBACK` du système
5. Se reconnecter et vérifier que les valeurs **ne sont plus** dans la table : `SELECT * FROM ...`

f) Exemple : Transfert protégé entre deux comptes

Exemple : Transfert entre deux comptes en SQL standard sous PostgreSQL

```
1  BEGIN;
2  UPDATE Compte1 SET Solde=Solde+100 WHERE Num=1;
3  UPDATE Compte2 SET Solde=Solde-100 WHERE Num=1;
4  COMMIT;
5  /
```

Exemple : Transfert entre deux comptes en PL/SQL sous Oracle

```
1  CREATE OR REPLACE PROCEDURE myTransfC1C2
2  IS
3  BEGIN
4    UPDATE Compte1 SET Solde=Solde+100 WHERE Num=1;
5    UPDATE Compte2 SET Solde=Solde-100 WHERE Num=1;
6   COMMIT;
7  END;
8  /
```

Exemple : Transfert entre deux comptes en VBA sous Access

```
1  Sub myTransfC1C2
2    BeginTrans
3      CurrentDb.CreateQueryDef("", "UPDATE Compte1 SET Solde=Solde+100
   WHERE Num=1").Execute
4      CurrentDb.CreateQueryDef("", "UPDATE Compte2 SET Solde=Solde-100
   WHERE Num=1").Execute
5    CommitTrans
6  End Sub
```

Fondamental : Transfert protégé

Pour les trois exemples ci-avant le transfert est protégé au sens où soit les deux `UPDATE` seront exécutés , soit aucun.

En cas de panne pendant la transaction, le transfert sera annulé (`ROLLBACK` système), mais en aucun cas un des deux comptes ne pourra être modifié sans que l'autre le soit (ce qui aurait entraîné une perte ou un gain sur la somme des deux comptes).

g) Exercice

Pour faire un transfert sécurisé d'un point de vue transactionnel de 100€ du compte bancaire C1 vers le compte bancaire C2 pour le compte numéro 12, quelle est la série d'instructions correcte (en mode `autocommit off`)?

○	UPDATE C1 SET Solde=Solde-100 WHERE Numero=12; ROLLBACK; UPDATE C2 SET Solde=Solde+100 WHERE Numero=12; COMMIT;
○	UPDATE C1 SET Solde=Solde-100 WHERE Numero=12 UPDATE C2 SET Solde=Solde+100 WHERE Numero=12 ROLLBACK;
○	UPDATE C1 SET Solde=Solde-100 WHERE Numero=12; COMMIT; UPDATE C2 SET Solde=Solde+100 WHERE Numero=12; COMMIT;
○	UPDATE C1 SET Solde=Solde-100 WHERE Numero=12; UPDATE C2 SET Solde=Solde+100 WHERE Numero=12; COMMIT;
○	UPDATE C1 SET Solde=Solde-100 WHERE Numero=12; ROLLBACK; UPDATE C2 SET Solde=Solde-100 WHERE Numero=12; ROLLBACK;

h) Complément : Algorithme de reprise UNDO-REDO

Introduction

L'algorithme suivant permet d'assurer une reprise après panne qui annule et rejoue les transactions adéquates.

```
1. SOIT deux listes REDO et UNDO
  1a. Initialiser la liste REDO à vide
  1b. Initialiser la liste UNDO avec toutes les transactions en cours au dernier point de contrôle

2. FAIRE une recherche en avant dans le journal, à partir du point de contrôle
  2a. SI une transaction T est commencée ALORS  ajouter T à UNDO
  2b. SI une transaction T est terminée avec succès alors déplacer T de UNDO à REDO

3. QUAND la fin du journal est atteinte
  3a. Annuler les transactions de la liste UNDO (reprise en arrière)
  3b. Rejouer les transactions de la liste REDO (reprise en avant)

4. TERMINER la reprise et redevenir disponible pour de nouvelles instructions
```

Exemple

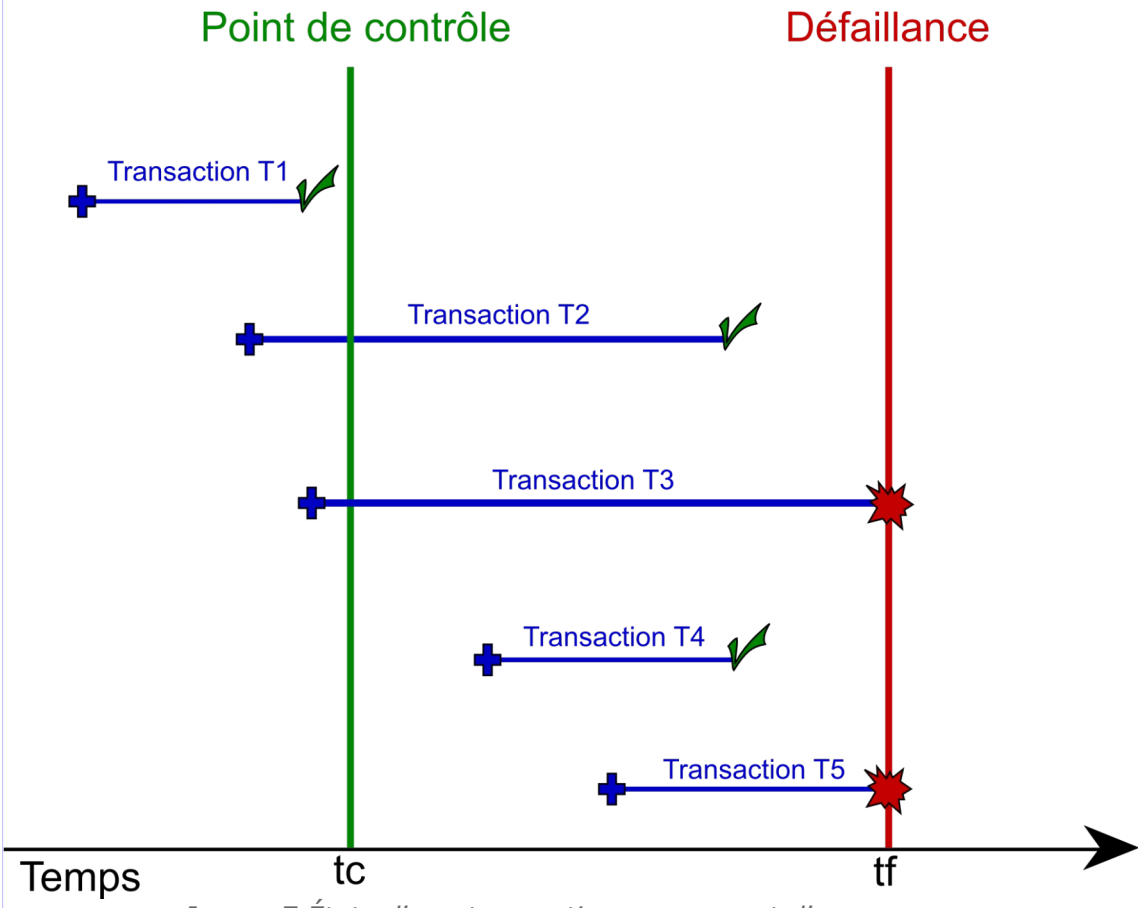


Image 7 États d'une transaction au moment d'une panne

- **Transactions de type T1**
  Non prises en compte par l'algorithme.
- **Transactions de type T2**
  Ajoutées à la liste UNDO (étape 1b) puis déplacée vers REDO (étape 2b) puis rejouée (étape 3b).
- **Transactions de type T3**
  Ajoutées à la liste UNDO (étape 1b) puis annulée (étape 3a).
- **Transactions de type T4**
  Ajoutées à la liste UNDO (étape 2a) puis déplacée vers REDO (étape 2b) puis rejouée (étape 3b).
- **Transactions de type T5**
  Ajoutées à la liste UNDO (étape 2a) puis annulée (étape 3a).

* *
*

On voit que la gestion transactionnelle est un appui important à la reprise sur panne, en ce qu'elle assure des états cohérents qui peuvent être restaurés.

4. Concurrence et transactions

Objectifs

Appréhender la gestion de la concurrence dans les SGBD.
Comprendre la réponse apportée par la gestion des transactions.

a) Trois problèmes soulevés par la concurrence.

Introduction

Nous proposons ci-dessous trois problèmes posés par les accès concurrents des transactions aux données.

i Perte de mise à jour

Temps	Transaction A	Transaction B
t1	LIRE T	
t2	...	LIRE T
t3	UPDATE T	...
t4	...	UPDATE T
t5	COMMIT	...
t4		COMMIT

Tableau 2 Problème de la perte de mise à jour du tuple T par la transaction A

Les transaction A et B accèdent au même tuple T ayant la même valeur respectivement à t1 et t2. Ils modifient chacun la valeur de T. Les modifications effectuées par A seront perdues puisqu'elle avait lu T avant sa modification par B.

Exemple

Temps	A : Ajouter 100	B : Ajouter 10
t1	LIRE COMPTE C=1000	
t2	...	LIRE COMPTE C=1000
t3	UPDATE COMPTE C=C+100=1100	...
t4	...	UPDATE COMPTE C=C+10=1010
t5	COMMIT C=1100	...
t6		COMMIT C=1010

Tableau 3 Doucle crédit d'un compte bancaire C

Dans cet exemple le compte bancaire vaut 1010 à la fin des deux transactions à la place de 1110.

ii Accès à des données non validées

Temps	Transaction A	Transaction B
t1		UPDATE T
t2	LIRE T	...
t3		ROLLBACK

Tableau 4 Problème de la lecture impropre du tuple T par la transaction A

La transaction A accède au tuple T qui a été modifié par la transaction B. B annule sa modification et A a donc accédé à une valeur qui n'aurait jamais dû exister (virtuelle) de T. Pire A pourrait faire une mise à jour de T **après** l'annulation par B, cette mise à jour inclurait la valeur **avant** annulation par B (et donc reviendrait à annuler l'annulation de B).

Exemple

Temps	A : Ajouter 10	B : Ajouter 100 (erreur)
t1		LIRE COMPTE C=1000
t2		UPDATE COMPTE C=C+100=1100
t3	LIRE COMPTE C=1100	...
t4	...	ROLLBACK C=1000
t5	UPDATE C C=C+10=1110	
t6	COMMIT C=1110	

Tableau 5 Annulation de crédit sur le compte bancaire C

Dans cet exemple le compte bancaire vaut 1110 à la fin des deux transactions à la place de 1010.

iii Lecture incohérente

Temps	Transaction A	Transaction B
t1	LIRE T	
t2	...	UPDATE T
t3	...	COMMIT
t4	LIRE T	

Tableau 6 Problème de la lecture non reproductible du tuple T par la transaction A

Si au cours d'une même transaction A accède deux fois à la valeur d'un tuple alors que ce dernier est, entre les deux, modifié par une autre transaction B, alors la

lecture de A est inconsistente. Ceci peut entraîner des incohérences par exemple si un calcul est en train d'être fait sur des valeurs par ailleurs en train d'être mises à jour par d'autres transactions.

Remarque

Le problème se pose bien que la transaction B ait été validée, il ne s'agit donc pas du problème d'accès à des données non validées.

Exemple

Temps	A : Calcul de S=C1+C2	B : Transfert de 10 de C1 à C2
t1	LIRE COMPTE1 C1=100	
t2	...	LIRE COMPTE 1 C1=100
t3	...	LIRE COMPTE 2 C2=100
t4	...	UPDATE COMPTE 1 C1=100-10=90
t5	...	UPDATE COMPTE 2 C2=100+10=110
t6	...	COMMIT
t7	LIRE COMPTE 2 C2=110	
t8	CALCUL S S=C1+C2=210	

Tableau 7 Transfert du compte C1 au compte C2 pendant une opération de calcul C1+C2

Dans cet exemple la somme calculée vaut 210 à la fin du calcul alors qu'elle devrait valoir 200.

b) Le verrouillage

Introduction

Une solution générale à la gestion de la concurrence est une technique très simple appelée verrouillage.

Définition : Verrou

Poser un verrou sur un objet (typiquement un tuple) par une transaction signifie rendre cet objet inaccessible aux autres transactions.

Synonymes : Lock

Définition : Verrou partagé S

Un verrou partagé, noté S, est posé par une transaction lors d'un accès en **lecture** sur cet objet.

Un verrou partagé interdit aux autres transaction de poser un verrou exclusif sur

cet objet et donc d'y accéder en écriture.

Synonymes : Verrou de lecture, Shared lock, Read lock

Définition : Verrou exclusif X

Un verrou exclusif, noté X, est posé par une transaction lors d'un accès en **écriture** sur cet objet.

Un verrou exclusif interdit aux autres transactions de poser tout autre verrou (partagé ou exclusif) sur cet objet et donc d'y accéder (ni en lecture, ni en écriture).

Synonymes : Verrou d'écriture, Exclusive lock, Write lock

Remarque : Verrous S multiples

Un même objet peut être verrouillé de façon partagée par plusieurs transactions en même temps. Il sera impossible de poser un verrou exclusif sur cet objet tant **qu'au moins une** transaction disposera d'un verrou S sur cet objet.

Méthode : Règles de verrouillage

Soit la transaction A voulant poser un verrou S sur un objet O

1. Si O n'est pas verrouillé alors A peut poser un verrou S
2. Si O dispose déjà d'un ou plusieurs verrous S alors A peut poser un verrou S
3. Si O dispose déjà d'un verrou X alors A ne peut pas poser de verrou S

Soit la transaction A voulant poser un verrou X sur un objet O

1. Si O n'est pas verrouillé alors A peut poser un verrou X
2. Si O dispose déjà d'un ou plusieurs verrous S ou d'un verrou X alors A ne peut pas poser de verrou X

	Verrou X présent	Verrou(s) S présent(s)	Pas de verrou présent
Verrou X demandé	Demande refusée	Demande refusée	Demande accordée
Verrou S demandé	Demande refusée	Demande accordée	Demande accordée

Tableau 8 Matrice des règles de verrouillage

Remarque : Promotion d'un verrou

Une transaction qui dispose déjà, **elle-même**, d'un verrou S sur un objet peut obtenir un verrou X sur cet objet si aucune autre transaction ne détient de verrou S sur l'objet. Le verrou est alors promu du statut partagé au statut exclusif.

c) Le déverrouillage

Définition : Déverrouillage

Lorsqu'une transaction se termine (COMMIT ou ROLLBACK) elle libère tous les verrous qu'elle a posé.

Synonymes : *Unlock*

d) Inter-blocage

Définition : Inter-blocage

L'inter-blocage est le phénomène qui apparaît quand deux transactions (ou plus, mais généralement deux) se bloquent mutuellement par des verrous posés sur les données. Ces verrous empêchent chacune des transactions de se terminer et donc

de libérer les verrous qui bloquent l'autre transaction. Un processus d'attente sans fin s'enclenche alors.

Les situations d'inter-blocage sont détectées par les SGBD et gérées, en annulant l'une, l'autre ou les deux transactions, par un ROLLBACK système. Les méthodes utilisées sont la détection de cycle dans un graphe d'attente et la détection de délai d'attente trop long.

Synonymes : *Deadlock*, Blocage, Verrou mortel

Définition : Cycle dans un graphe d'attente

Principe de détection d'un inter-blocage par détection d'un cycle dans un graphe représentant quelles transactions sont en attente de quelles transactions (par inférence sur les verrous posés et les verrous causes d'attente). Un cycle est l'expression du fait qu'une transaction A est en attente d'une transaction B qui est en attente d'une transaction A.

La détection d'un tel cycle permet de choisir une **victime**, c'est à dire une des deux transactions qui sera annulée pour que l'autre puisse se terminer.

Synonymes : Circuit de blocage

Définition : Délai d'attente

Principe de décision qu'une transaction doit être abandonnée (ROLLBACK) lorsque son délai d'attente est trop long.

Ce principe permet d'éviter les situations d'inter-blocage, en annulant une des deux transactions en cause, et en permettant donc à l'autre de se terminer.

Synonymes : *Timeout*

Remarque : Risque lié à l'annulation sur délai

Si le délai est trop court, il y a un risque d'annuler des transactions en situation d'attente longue, mais non bloquées.

Si le délai est trop long, il y a un risque de chute des performances en situation d'inter-blocage (le temps que le système réagisse).

La détection de cycle est plus adaptée dans tous les cas, mais plus complexe à mettre en œuvre.

Remarque : Relance automatique

Une transaction ayant été annulée suite à un inter-blocage (détection de cycle ou de délai d'attente) n'a pas commis de "faute" justifiant son annulation. Cette dernière est juste due aux contraintes de la gestion de la concurrence. Aussi elle n'aurait pas dû être annulée et devra donc être exécutée à nouveau.

Certains SGBD se charge de relancer automatiquement les transactions ainsi annulées.

e) Mini-TP : Simulation d'accès concurrents sous PostgreSQL

1. Se connecter deux fois à une même base de données dans deux terminaux (**term1** et **term2**) :
   - `psql mydb`
   - `psql mydb`
2. Créer une table `test` : `CREATE TABLE test (a integer);`

Question 1

1. **term1** Insérer une valeur : `INSERT ...` (`COMMIT` implicite)
2. **term2** Vérifier que la valeur est visible : `SELECT ...`

Question 2

1. **term1** Commencer une transaction : `BEGIN TRANSACTION;`
2. **term1** Insérer la valeur 2 dans la table : `INSERT INTO...`
3. **term1** Vérifier que la valeur est bien dans la table : `SELECT * FROM ...`
4. **term2** Vérifier que la valeur **n'est pas visible** : `SELECT * FROM ...`
5. **term1** Valider la transaction : `COMMIT;`
6. **term2** Vérifier que la valeur **est visible** : `SELECT * FROM ...`

Question 3

1. **term1** Commencer une transaction : `BEGIN TRANSACTION;`
2. **term1** Exécuter une mise à jour (multiplier toutes les valeurs par 2) : `UPDATE...`
3. **term2** Exécuter une mise à jour concurrente (multiplier les valeurs par 3) : `UPDATE...`
4. **term2** Observer la mise en attente
5. **term1** Valider la transaction : `COMMIT;`
6. **term2** Vérifier le déblocage et que les deux mises à jour ont bien été effectuées (a multiplié par 6) : `SELECT...`
7. **term1** Vérifier que les deux mises à jour ont bien été effectuées (a multiplié par 6) : `SELECT...`

f) Solution aux trois problèmes soulevés par la concurrence.

Introduction

Le principe du verrouillage permet d'apporter une solution aux trois problèmes classiques soulevés par les accès aux concurrents aux données par les transactions.

i Perte de mise à jour

Temps	Transaction A	Transaction B
t1	LIRE T Verrou S	
t2	...	LIRE T Verrou S
t3	UPDATE T Attente...	...
t4	... Attente...	UPDATE T Attente...
...	Inter-blocage	

Tableau 9 Problème de la perte de mise à jour du tuple T par la transaction A

Remarque

Le problème de perte de mise à jour est réglé, mais soulève ici un autre problème, celui de **l'inter-blocage**.

ii Accès à des données non validées

Temps	Transaction A	Transaction B
t1		UPDATE T Verrou X
t2	LIRE T Attente...	...
t3		ROLLBACK Libération du verrou X
t4	Verrou S	

Tableau 10 Problème de la lecture impropre du tuple T par la transaction A

iii Lecture incohérente

Temps	Transaction A	Transaction B
t1	LIRE T Verrou S	
t2	...	UPDATE T Attente...
t3	LIRE T Verrous S	...
...	... libération des verrous ...	... reprise de la transaction ...

Tableau 11 Problème de la lecture non reproductible du tuple T par la transaction A

Remarque

La lecture reste cohérente car aucune mise à jour ne peut intervenir pendant le processus de lecture d'une même transaction.

g) Exercice

Soit l'exécution concurrente des deux transactions ci-dessous (on pose qu'aucune autre transaction ne s'exécute par ailleurs).

NB : Le tableau fait état des temps auxquels les instructions sont soumises au serveur et non auxquels elles sont traitées.

Temps	Transaction 1	Transaction 2
t0		BEGIN TRANSACTION
t1	BEGIN TRANSACTION	
t2	UPDATE TABLE1 SET TABLE1.A=TABLE1.A+1	
t3		UPDATE TABLE1 SET TABLE1.A=TABLE1.A+1
t4	UPDATE TABLE1 SET TABLE1.A=TABLE1.A+1	
t5	COMMIT	
t6		?

Tableau 12 Transactions concurrentes

De combien le champ TABLE1.A a-t-il été augmenté à t6 du point de vue de la transaction 2 ?

NB : Si vous pensez que le résultat ne peut être déterminé à cause d'une perte de mise à jour ou d'un inter-blocage, répondez 0.

h) Complément : Protocole d'accès aux données.

Méthode : Règles de verrouillage avant les lectures et écritures des données

Soit la transaction A voulant lire des données d'un tuple T :

1. A demande à poser un verrou S sur T
2. Si A obtient de poser le verrou alors A lit T
3. Sinon A attend le droit de poser son verrou (et donc que les verrous qui l'en empêchent soient levés)

Soit la transaction A voulant écrire des données d'un tuple T :

1. A demande à poser un verrou X sur T
2. Si A obtient de poser le verrou alors A écrit T
3. Sinon A attend le droit de poser son verrou (et donc que les verrous qui l'en empêchent soient levés)

Soit la transaction A se terminant (COMMIT ou ROLLBACK) :

1. A libère tous les verrous qu'elle avait posé
2. Certaines transactions en attente obtiennent éventuellement le droit de poser des verrous

Remarque : Liste d'attente

Afin de rationaliser les attentes des transactions, des stratégies du type FIFO sont généralement appliquées et donc les transactions sont empilées selon leur ordre de demande.

5. Synthèse : Les transactions

Transaction

Unité logique de travail pour assurer la cohérence de la BD même en cas de pannes ou d'accès concurrents.

- Panne

  Même en cas de panne, la BD doit rester cohérente.
  - Défaillances système
    Coupure de courant, de réseau, etc.
  - Défaillances du support
    Crash disque (dans ce cas les transactions peuvent être insuffisantes).
- Concurrence

  Dimension relevant de la conception d'application.
  - Perte de mise à jour
  - Accès à des données non valides
  - Lecture incohérente
- Programmation

  Un programme peut décider de l'annulation d'une transaction.
  - ROLLBACK
    Instruction SQL d'annulation d'une transaction.

6. Bibliographie commentée sur les transactions

Complément : Synthèses

Les transactions [w_developpez.com/hcesbronlavau]

Une bonne introduction courte au principe des transactions avec un exemple très bien choisi. Des exemples d'implémentation sous divers SGBD (InterBase par exemple)

SQL2 SQL3, applications à Oracle [Delmal01]

Une bonne description des principes des transactions, avec les exemples caractéristiques, l'implémentation SQL et une étude de cas sous Oracle 8 (chapitre 5).

Tutoriel de bases de données relationnelles de l'INT Evry [w_int-evry.fr] (*http://www-inf.int-evry.fr/COURS/BD/BD_REL/SUPPORT/poly.html#RTFToC30*[11]

Un aperçu général de l'ensemble de la problématique des transactions, de la concurrence et de la fiabilité.

Programmation SQL [Mata03]

Un exemple d'exécution de transactions (pages 20-23)

B. Exercices

1. Super-héros sans tête

[15 minutes]

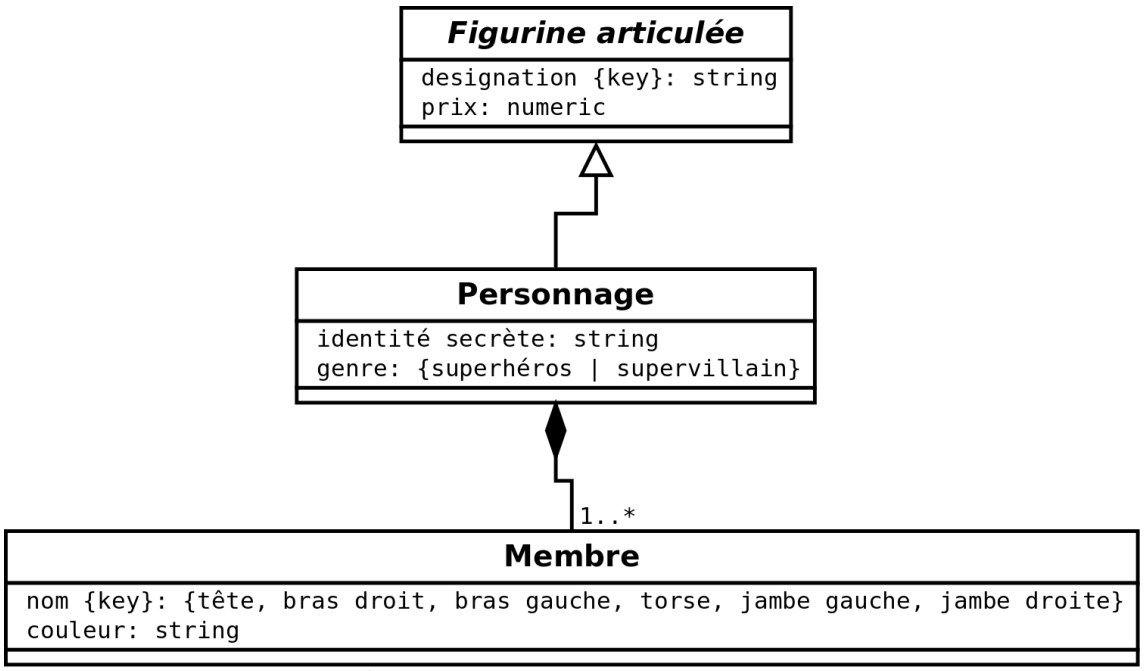
Les usines GARVEL construisent des figurines de super-héros à partir des données présentes dans la base de données PostgreSQL de l'entreprise. Un gros problème est survenu le mois dernier, lorsque l'usine en charge d'une nouvelle figurine,

11 - http://www-inf.int-evry.fr/COURS/BD/BD_REL/SUPPORT/poly.html#RTFToC30

Superchild, a livré un million d'exemplaires sans tête. À l'analyse de la base il a en effet été observé que la base contenait un tuple "Superchild" dans la table Personnage, et cinq tuples associés dans la table Membre, deux pour les bras, deux pour les jambes et un pour le torse, mais aucun pour la tête.

Le service qui a opéré la saisie du nouveau personnage assure, sans ambiguïté possible, que la tête a pourtant été saisie dans la base. En revanche, l'enquête montre des instabilités de son réseau à cette période.

L'extrait du modèle UML utile au problème est proposé ci-après, ainsi que le code SQL exécuté via le client `psql` lors de l'insertion de la nouvelle figurine.



Modèle UML Figurines GARVEL (extrait)

```
1  \set AUTOCOMMIT on
2  INSERT INTO Personnage (designation, prix, identite_secrete, genre)
   VALUES ('Superchild','12','Jordy','superhéros') ;
3  INSERT INTO Membre (propriétaire, nom, couleur) VALUES
   ('Superchild','bras droit','bleu') ;
4  INSERT INTO Membre (propriétaire, nom, couleur) VALUES
   ('Superchild','bras gauche','bleu') ;
5  INSERT INTO Membre (propriétaire, nom, couleur) VALUES
   ('Superchild','jambe gauche','bleu') ;
6  INSERT INTO Membre (propriétaire, nom, couleur) VALUES
   ('Superchild','jambe droite','bleu') ;
7  INSERT INTO Membre (propriétaire, nom, couleur) VALUES
   ('Superchild','torse','rouge') ;
8  INSERT INTO Membre (propriétaire, nom, couleur) VALUES
   ('Superchild','tete','rose') ;
```

Question 1

Expliquer la nature du problème qui est probablement survenu. Proposer une solution générale pour que le problème ne se renouvelle pas, en expliquant pourquoi.

Soyez **concis** et **précis** : La bonne mobilisation des concepts du domaine et la clarté de la rédaction seront appréciées.

Question 2

Illustrer la solution proposée en corrigeant le code SQL de l'insertion de "Superchild".

2. Exercice : Films en concurrence

Soit la table Film suivante définie en relationnel permettant d'enregistrer le nombre d'entrées des films identifiés par leur ISAN.

```
1 Film(#isan:char(33),entrees:integer)
```

Soit l'exécution concurrente de deux transactions TR1 et TR2 visant à ajouter chacune une entrée au film '123' :

Temps	Transaction TR1	Transaction TR2
t0	BEGIN	
t1		BEGIN
t2	UPDATE Film SET entrees=entrees+1 WHERE isan='123'	
t3		
t4		UPDATE Film SET entrees=entrees+1 WHERE isan='123'
t5		
t6	COMMIT	
t7		
t8		COMMIT

Tableau 13 Transaction parallèles TR1 et TR2 sous PostgreSQL

NB :

- *Les instructions sont reportées au moment où elles sont transmises au serveur*
- *Aucune autre transaction n'est en cours d'exécution entre t0 et t8.*

Exercice

*De combien les entrées du film 123 ont-t-elles été augmentées à **t3** du point de vue de la transaction **TR1** ?*

Exercice

*De combien les entrées du film 123 ont-t-elles été augmentées à **t3** du point de vue de la transaction **TR2** ?*

Exercice

*De combien les entrées du film 123 ont-t-elles été augmentées à **t5** du point de vue de la transaction **TR1** ?*

Exercice

*De combien les entrées du film 123 ont-t-elles été augmentées à **t5** du point de vue de la transaction **TR2** ?*

Exercice

*De combien les entrées du film 123 ont-t-elles été augmentées à **t7** du point de vue de la transaction **TR1** ?*

Exercice

*De combien les entrées du film 123 ont-t-elles été augmentées à **t7** du point de vue de la transaction **TR2** ?*

IV

Implémentation de bases de données relationnelles avec PostgreSQL sous Linux

A. Cours

1. Introduction à PostgreSQL : présentation, installation et utilisation du client

a) Présentation de PostgreSQL

PostgreSQL est :

- un SGBDR
- libre (licence BSD)
- multi-plate-formes (Unix, Linux, Windows, MacOS, ...)
- puissant
- très respectueux du standard
- très bien documenté

Fondamental : Documentation de PostgreSQL

- *www.postgresql.org/docs*[12]

12 - https://www.postgresql.org/docs/current

- en français : *docs.postgresqlfr.org*[13]

Fonctionnement général

PostgreSQL comme la grande majorité des SGBD est un logiciel client-serveur. Il faut donc pour l'utiliser un serveur (programme *postgresql* sous Linux) et un client.

Il existe plusieurs clients, les deux plus utilisés sont le client textuel *psql* et le client graphique *PgAdmin III*.

Complément

- *www.postgresql.org*[14]
- *www.postgresql.fr*[15]

b) Le client textuel "psql"

Définition : psql

`psql` est le client textuel de PostgreSQL.

Méthode : Connexion

```
1 psql -h tuxa.sme.utc -U nf17a001 -d dbnf17a001
```

Cette commande *psql* essaye de se connecter sur la machine *tuxa.sme.utc* avec un utilisateur PostgreSQL nommé *nf17a001* à la base de données *dbnf17a001*.

- Un serveur PostgreSQL doit tourner sur la machine distante *tuxa.sme.utc* (sur le port standard 5432).
- Un utilisateur *nf17a001* doit exister sur PostgreSQL et avoir un mot de passe défini.
- Une base de données *dbnf17a001* doit exister sur PostgreSQL et l'utilisateur *nf17a001* doit avoir le droit d'y accéder.
- Le mot de passe de l'utilisateur *nf17a001* sera demandé.

Syntaxe : Écrire une instruction SQL

```
1 dbnf17p015=> SELECT * FROM matable ;
```

Syntaxe : Écrire une instruction SQL sur plusieurs lignes

Une instruction SQL peut s'écrire sur une ou plusieurs lignes, le `retour chariot` n'a pas d'incidence sur la requête, c'est le `;` qui marque la fin de l'instruction SQL et provoque son exécution.

```
1 dbnf17p015=> SELECT *
2 dbnf17p015-> FROM matable
3 dbnf17p015-> ;
```

On notera dans `psql` la différence entre les caractères `=>` et `->` selon que l'on a ou pas effectué un retour chariot.

13 - https://docs.postgresqlfr.org/current/
14 - http://www.postgresql.org/
15 - http://www.postgresql.fr/

Fondamental : Commandes de base : aide

`\?` : Liste des commandes `psql`

`\h` : Liste des instructions SQL

`\h CREATE TABLE` : Description de l'instruction SQL `CREATE TABLE`

Fondamental : Commandes de base : catalogue

`\d` : Liste des relations (catalogue de données)

`\d maTable` : Description de la relation `maTable`

Fondamental : Commandes de base : quitter

`\q` : Quitter `psql`

Complément

http://www.postgresql.org/docs/current/static/app-psql.html[16]

2. Éléments de base pour l'utilisation de PostgreSQL

a) Exécuter des instructions SQL depuis un fichier

Il est souvent intéressant d'exécuter un fichier contenant une liste de commandes SQL, plutôt que de les entrer une par une dans le terminal. Cela permet en particulier de recréer une base de données à partir du script de création des tables.

Syntaxe

Pour exécuter un fichier contenant du code SQL utiliser la commande PostgreSQL `\i chemin/fichier.sql`

- `chemin` désigne le répertoire dans lequel est le fichier `fichier.sql`
- le dossier de travail de `psql` est le dossier dans lequel il a été lancé, le script peut être lancé à partir de son dossier `home` pour en être indépendant (`~/.../fichier.sql`)
- chaque commande doit être terminée par un `;`

```
1  dbnf17p015=> \i /home/me/bdd.sql
```

Méthode : Programmer une base de données avec PostgreSQL

Pour programmer une base de données sous PostgreSQL, voici une méthode générale :

- Rendez-vous dans un répertoire de travail : `cd /home/me/bdd1`
- Créez ou ouvrez un fichier texte avec un éditeur colorisant : `atom fichier.sql`
- Ouvrez un terminal et exécutez votre client *psql*
- Écrivez votre code SQL, et testez-le au fur et à mesure : `\i fichier.sql`

16 - http://www.postgresql.org/docs/current/static/app-psql.html

Conseil : Tester votre code régulièrement

Afin de tester régulièrement votre base de données, pensez à insérer en début de script des instruction de destruction des tables.

On supprime les tables dans l'ordre inverse de leur création, on peut ajouter la clause `IF EXISTS` afin d'éviter les erreurs lorsqu'une exécution précédente avait déjà échoué.

```
1  DROP TABLE IF EXISTS t2 ;
2  DROP TABLE IF EXISTS t1 ;
3  CREATE TABLE t1 (a VARCHAR PRIMARY KEY);
4  CREATE TABLE t2 (a VARCHAR REFERENCES t1(a));
```

b) Types de données de PostgreSQL

Types standards

On retrouve dans PostgreSQL la plupart des types standard SQL :

- numériques : integer (smallint, bigint), real (double precision)
- dates : date (time, timestamp)
- chaînes : char, varchar
- autres : boolean

Autres types

On notera les autres types suivants (non standard) :

- identifiant : serial, uuid
- numériques : money
- chaînes : text
- non SQL : xml, json

Complément : Fonctions de formatage des types de données

https://docs.postgresql.fr/current/functions-formatting.html[17]

Complément : Documentation

https://docs.postgresqlfr.org/current/datatype.html[18]

c) Importer un fichier CSV sous PostgreSQL

Syntaxe

```
1  \copy nom_table (att1, att2, ...) FROM 'fichier.csv' WITH CSV
   DELIMITER ';' QUOTE '"'
```

- `WITH` introduit les options de l'import
- `CSV` indique qu'il s'agit d'un fichier CSV
- `DELIMITER 'c'` indique que le caractère **c** est utilisé comme délimiteur de champ (en général **;** ou **,**)
- `QUOTE 'c'` indique que le caractère **c** est utilisé comme délimiteur de chaîne (en général **"**)

17 - https://docs.postgresql.fr/current/functions-formatting.html
18 - https://docs.postgresqlfr.org/current/datatype.html

Remarque

- La table `nom_table` doit déjà exister
- Le nombre de colonnes spécifié doit correspondre au nombre de colonnes du fichier CSV
- Les types doivent être compatibles

Remarque

Ajouter l'option `HEADER` après `WITH CSV` si le fichier CSV contient une ligne s'entête.

```
1  \copy nom_table (att1, att2, ...) FROM 'fichier.csv' WITH CSV HEADER
   DELIMITER ';' QUOTE '"'
```

Méthode : Localisation du fichier CSV depuis psql

Par défaut, la commande `\copy` prendra le chemin du répertoire courant au moment où la commande `psql` a été lancée.

Sous `psql`, vous pouvez utiliser les commandes :

- `dbnf17p007=> \! pwd`

  Pour exécuter la commande *shell* `pwd` et obtenir le répertoire courant
- `dbnf17p007=> \cd directory`

  Pour changer le répertoire courant

Complément

PostgreSQL sous Linux
Fichier CSV

d) Notion de schéma sous PostgreSQL

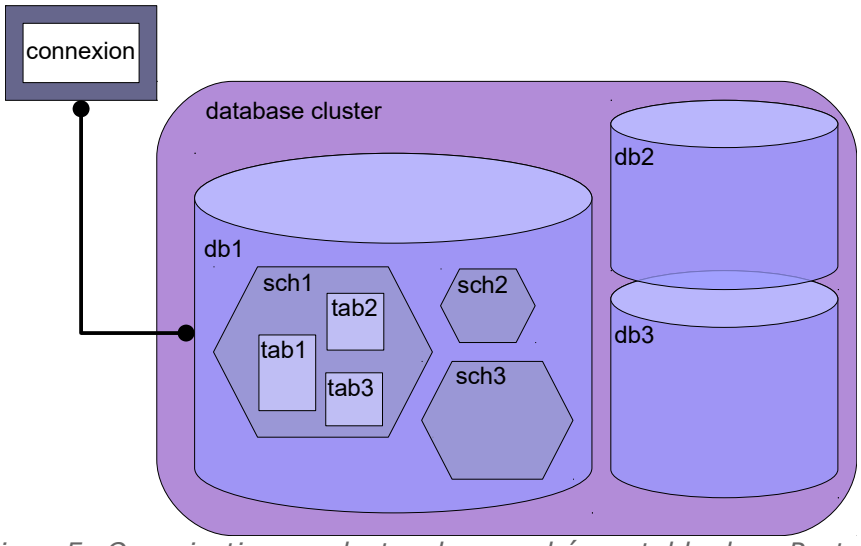
Définition

A PostgreSQL database cluster contains one or more named databases. Users and groups of users are shared across the entire cluster, but no other data is shared across databases. Any given client connection to the server can access only the data in a single database, the one specified in the connection request.

A database contains one or more named schemas, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both schema1 and myschema can contain tables named mytable. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database he is connected to, if he has privileges to do so.

http://www.postgresql.org/docs/8.4/static/ddl-schemas.html[19]

19 - http://www.postgresql.org/docs/8.4/static/ddl-schemas.html



Graphique 5 Organisation en cluster, base, schéma, table dans PostgreSQL

Syntaxe : Créer un schéma

```
1  CREATE SCHEMA myschema;
```

Syntaxe : Créer une table dans un schéma

```
1  CREATE TABLE myschema.mytable (
2  ...
3  );
```

Syntaxe : Requêter dans un schéma

```
1  SELECT ...
2  FROM myschema.mytable
```

Exemple

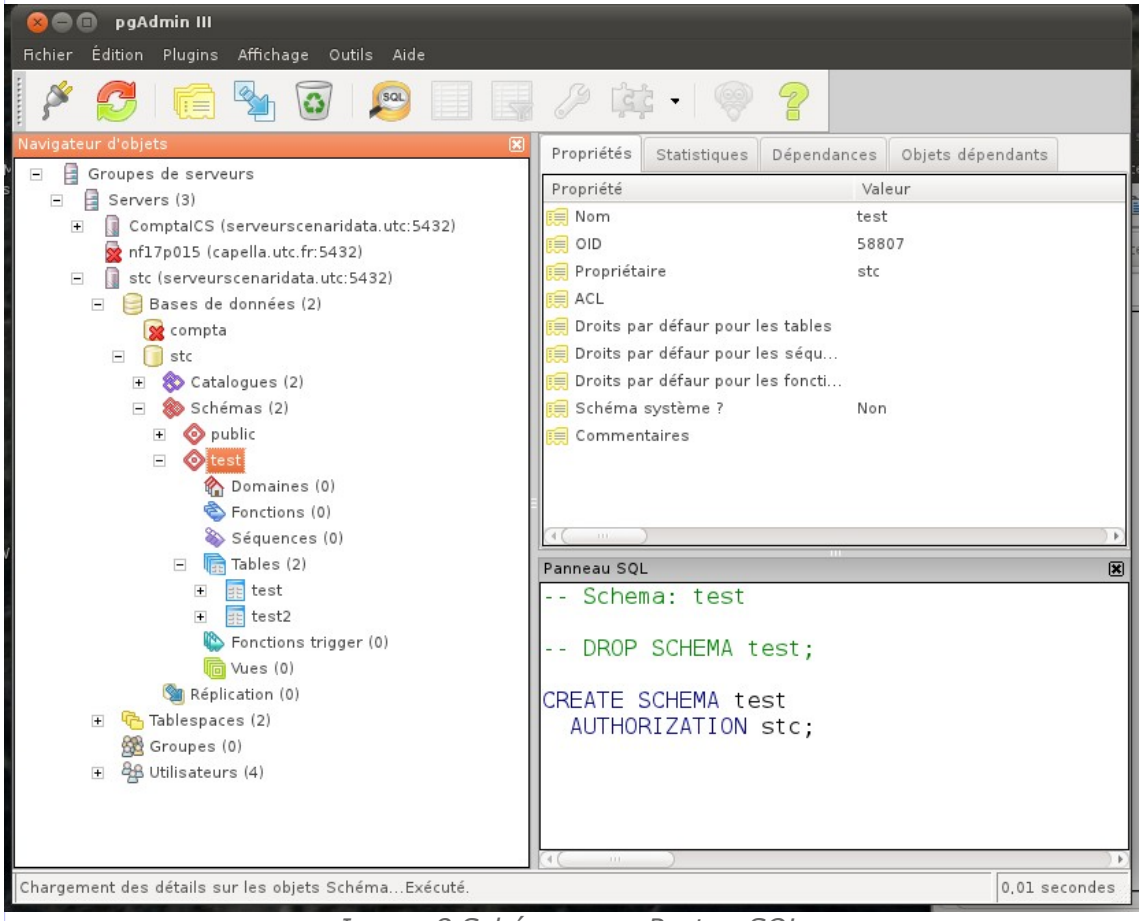


Image 8 Schéma sous PostgreSQL

Syntaxe : Catalogue : schémas

`\dn` : Liste des schémas de ma base de données

Complément : Schéma par défaut : search_path

Afin d'alléger la syntaxe il est possible de définir un schéma par défaut, dans lequel seront créer les tables non-préfixées et un ou plusieurs schémas par défaut dans lesquels seront requêtées les tables non-préfixées.

```
1 SET search_path TO myschema,public;
```

Cette instruction définit le schéma `myschema` comme schéma par défaut pour la création de table et le requêtage, puis `public` pour le requêtage, le premier étant prioritaire sur le second :

- `CREATE mytable` créera ma `mytable` dans le schéma `mychema`.
- `SELECT FROM mytable` cherchera la table dans la schéma `mychema`, puis dans le schéma `public` si la table n'existe pas dans le premier schéma.

Remarque : Schéma "public"

Le schéma `public` est un schéma créé par défaut à l'initialisation de la base, et qui sert de schéma par défaut en l'absence de toute autre spécification.

Remarque : Catalogue : \d

La liste des tables retournée par `\d` dépend du *search_path*. Pour que `\d` retourne les tables de `schema1`, il faut donc exécuter l'instruction :

```
SET search_path TO public, schema1
```

Complément : Pour aller plus loin

http://www.postgresql.org/docs/8.4/static/ddl-schemas.html[20]

e) PostgreSQL sous Linux

Syntaxe : Exécuter une instruction PostgreSQL depuis Linux

```
1  psql -c "instruction psql"
```

```
1  psql -h localhost -U user -d db -c "instruction psql"
```

Exemple : Exécuter un script PostgreSQL depuis Linux

```
1  #!/bin/sh
2  psql -c "DROP DATABASE mydb"
3  psql -c "CREATE DATABASE mydb"
4  psql -c "GRANT ALL PRIVILEGES ON DATABASE mydb TO user1"
```

Complément : psql : exécuter une commande Linux (Linux sous Postgres)

- `\!` : permet d'exécuter certaines commandes du *shell* Linux depuis le client *psql*.

f) Les clients graphiques pgAdminIII et phpPgAdmin

pgAdminIII

Un client graphique une interface graphique permettant d'effectuer les mêmes opérations qu'avec le client `psql`.

- Le client graphique pgAdminIII est un client lourd qui fonctionne très bien sous Linux et sous Windows.
- Le client graphique phpPgAdmin est un client léger (qui tourne dans un navigateur Web donc).

Déclarer une connexion dans pgAdminIII

1. Sélectionner `Fichier > Ajouter un serveur`
2. Saisissez les informations de connexion à un serveur PostgreSQL existant :
   - Hôte : tuxa.sme.utc
   - Port : 5432 (port standard de PostgreSQL)
   - Base : dbnf17p...
   - Nom : nf17p...
   - Mot de passe : ...

Ouvrir un terminal SQL dans pgAdminIII

1. Sélectionner sa base de données dans la liste `Bases de données`

20 - http://www.postgresql.org/docs/8.4/static/ddl-schemas.html

2. Sélectionner `Outils > Éditeur de requêtes` (ou `CTRL+E`)

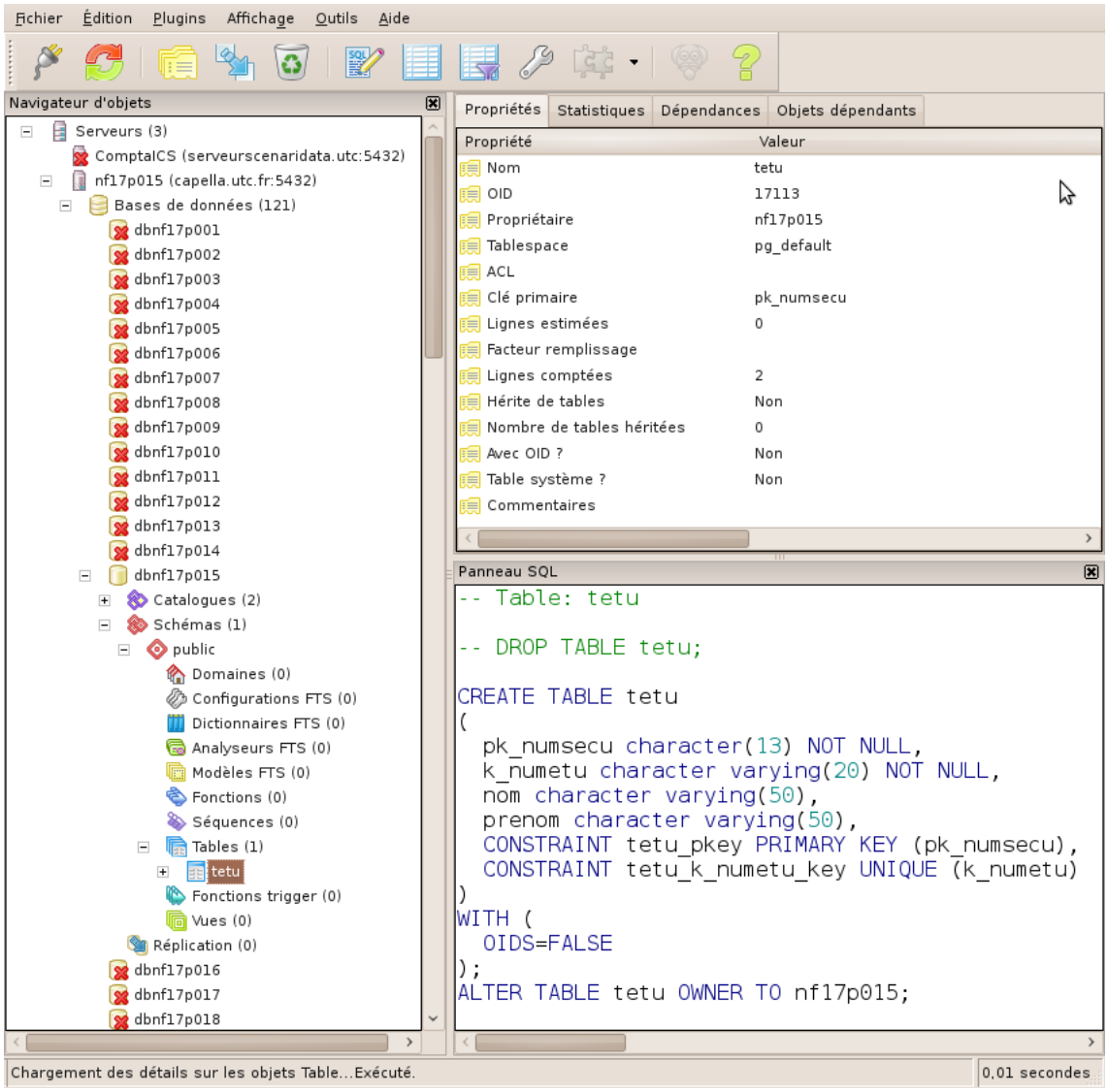


Image 9 pgAdminIII

Complément : phpPgAdmin

http://phppgadmin.sourceforge.net[21]

g) Compléments

Complément : Héritage (clause INHERITS)

http://www.postgresql.org/docs/current/static/sql-createtable.html[22]

Complément : PL/pgSQL

http://www.postgresql.org/docs/current/static/plpgsql.html[23]

21 - http://phppgadmin.sourceforge.net
22 - http://www.postgresql.org/docs/current/static/sql-createtable.html
23 - http://www.postgresql.org/docs/current/static/plpgsql.html

Complément : Autres langages procéduraux (PL)

- PL/Tcl
- PL/Perl
- PL/Python
- ...

http://www.postgresql.org/docs/current/static/xplang.html[24]
http://www.postgresql.org/docs/current/static/server-programming.html[25]

Complément : Triggers

http://www.postgresql.org/docs/current/static/sql-createtrigger.html[26]
(*http://www.postgresql.org/docs/current/static/triggers.html*[27])

B. Exercices

1. Découverte d'un SGBDR avec PostgreSQL

a) Configuration technique pour réaliser les exercices

Fondamental : Serveur Linux + Client Linux

La meilleure configuration pour réaliser ces exercices est de disposer :

1. d'un serveur Linux hébergeant un serveur PostgreSQL ;
2. d'une base de données déjà créée (cela peut être la base par défaut *postgres*) ;
3. d'un utilisateur ayant les pleins droits sur cette base de données (cela peut être l'utilisateur par défaut *postgres*) ;
4. d'un client *psql* sous Linux connecté à la base de données.

Serveur Linux + Client Windows

Si un serveur est disponible sous Linux mais que le client est sous Windows, il y a deux solutions :

- **Se connecter en SSH au serveur Linux avec Putty (on est ramené au cas précédent).**
- Installer le client *psql* sous Windows, mais cette solution est déconseillée :
  - le terminal est bien moins confortable que sous Linux,
  - une partie des questions sont relatives aux systèmes Linux et ne pourra être réalisée.

Complément : Serveur Windows + Client Windows

Cette configuration permettra de faire une partie des exercices, avec des adaptations.

24 - http://www.postgresql.org/docs/current/static/xplang.html
25 - http://www.postgresql.org/docs/current/static/server-programming.html
26 - http://www.postgresql.org/docs/current/static/sql-createtrigger.html
27 - http://www.postgresql.org/docs/current/static/triggers.html

b) Connexion à une base de données PostgreSQL

Cet exercice commence alors que vous êtes connecté à une base Oracle avec le client *psql*.

```
psql (9.x.x)
Saisissez « help » pour l'aide.

mydb=>
```

Demander de l'aide

Demander de l'aide en testant :

- help
- \?
- \h
- \h CREATE TABLE
- \h SELECT

c) Créer une base de données avec PostgreSQL

Créer une table

Exécuter les instructions suivantes.

```
CREATE TABLE etu (
pknumsecu CHAR(13) PRIMARY KEY,
knumetu VARCHAR(20) UNIQUE NOT NULL,
nom VARCHAR(50),
prenom VARCHAR(50));
```

```
INSERT INTO etu (pknumsecu, knumetu, nom, prenom)
VALUES ('1800675001066', 'AB3937098X', 'Dupont', 'Pierre');
INSERT INTO etu (pknumsecu, knumetu, nom, prenom)
VALUES ('2820475001124', 'XGB67668', 'Durand', 'Anne');
```

```
CREATE TABLE uv (
pkcode CHAR(4) NOT NULL,
fketu CHAR(13) NOT NULL,
PRIMARY KEY (pkcode, fketu),
FOREIGN KEY (fketu) REFERENCES etu(pknumsecu));
```

```
INSERT INTO uv (pkcode, fketu)
VALUES ('NF17', '1800675001066');
INSERT INTO uv (pkcode, fketu)
VALUES ('NF26', '1800675001066');
INSERT INTO uv (pkcode, fketu)
VALUES ('NF29', '1800675001066');
```

Question 1

Utiliser le catalogue pour vérifier la création de la table.

Question 2

Utiliser deux instructions SELECT pour vérifier le contenu de la table.

d) Import de données depuis un fichier CSV

Nous allons à présent réinitialiser la base avec des données contenues dans un fichier.

Question 1

Exécuter les instructions nécessaires afin de **supprimer** les données existantes dans les tables (instruction DELETE du SQL LMD). Vérifier que les tables sont vides.

Question 2

Créer les deux fichiers de données suivants, respectivement *etus.csv* et *uvs.csv*. Regarder le contenu des fichiers. Pourquoi les appelle-t-on des fichiers CSV ?

```
1;A;Dupont;Pierre
2;B;Durand;Georges
3;C;Duchemin;Paul
4;D;Dugenou;Alain
5;E;Dupied;Albert
```

```
1;NF17
1;NF18
1;NF19
1;NF20
1;LA13
1;PH01
2;NF17
2;NF18
2;NF19
2;TN01
2;LA14
2;PH01
3;NF17
3;NF18
3;NF19
3;NF21
3;LA14
3;PH01
4;NF17
4;NF20
4;NF21
4;GE10
4;LA14
4;PH01
5;NF17
5;NF18
5;NF20
5;GE10
5;PH01
5;PH02
```

Indice :

Fichier CSV

Question 3

Insérer les données en complétant les instructions suivantes.

```
\copy ... (...) FROM '...' WITH CSV DELIMITER '...'
\copy ... (...) FROM '...' WITH CSV DELIMITER '...'
```

Indice :

Importer un fichier CSV

Question 4

Écrivez les requêtes permettant d'obtenir le nombre d'UV suivies par un étudiant et le nombre d'étudiants inscrits par UV.

e) Manipulation de schémas

Question 1

Visualiser la liste des schémas existants dans votre base de données.

Indice :

Notion de schéma

Question 2

Créer un second schéma *loisir*.

Question 3

Créer une table *sport* dans le schéma *loisir* ; l'objectif est d'indiquer pour chaque étudiant, la liste des sports qu'il pratique, par exemple : Tennis, Karaté, Aviron...

Vérifier votre création dans le dictionnaire.

f) Exécution de fichiers SQL et de scripts Linux

Reporter dans un fichier SQL l'ensemble des instruction permettant de supprimer, de créer et d'instancier les tables de la base de données.

Question 1

Exécuter ce fichier depuis *psql*.

Indice :

Exécuter des instructions SQL depuis un fichier

Question 2

Exécuter ce fichier depuis un script Linux.

Indice :

PostgreSQL sous Linux

g) Test de pgAdminIII

Connexion depuis un PC avec pgAdmin III

Installer si nécessaire, puis lancer et tester le programme `pgAdminIII`.

Exécuter une ou deux requêtes permettant de se familiariser avec son fonctionnement.

V

Application de bases de données, principes et exemples avec Python

A. Cours

1. Applications et bases de données

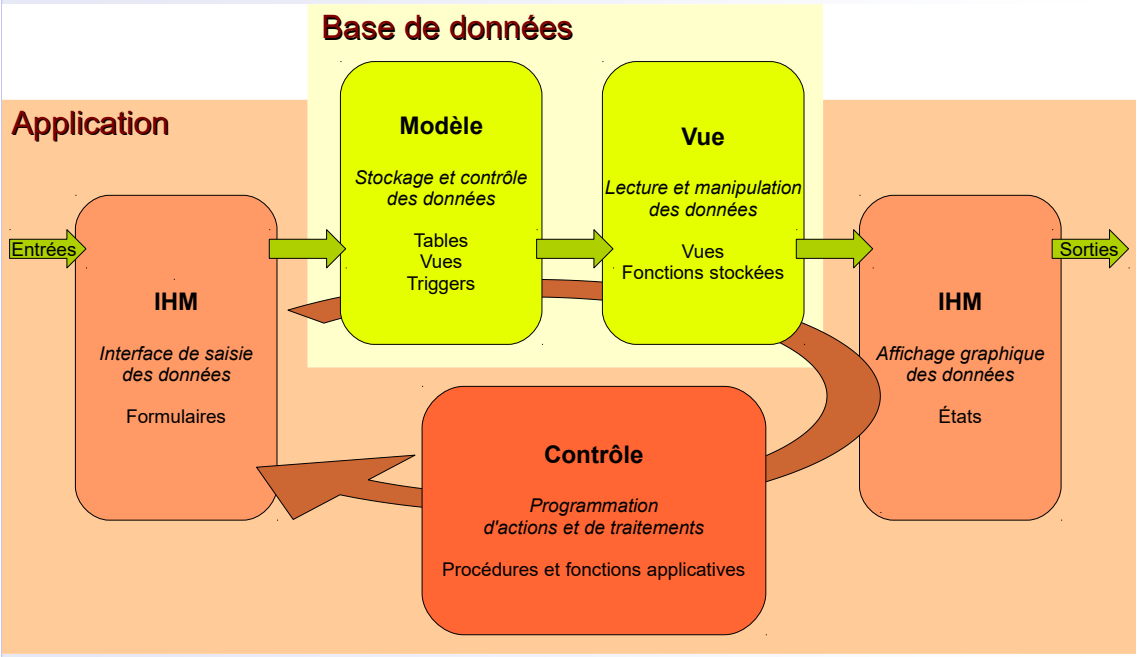
a) Architecture générale d'une application de base de données

Définition

Une application de base de données comporte :

1. Une base de données : Elle pose le modèle de données, stocke les données, définit des vues sur les données (préparation des modalités de lecture)
2. Une application : Elle définit les interfaces homme-machine pour écrire et lire les données et contient le programme informatique de traitement des données avant insertion dans la base ou présentation à l'utilisateur.

Fondamental



Graphique 6 Architecture générale d'une application de base de données

Méthode : Procédure générale de développement

1. Développement de la BD : Conception en UML ou EA, puis traduction en R (ou NoSQL), puis implémentation SQL
   - Créer les **tables**
   - Ajouter des **vues** et **triggers** (lorsque le SGBD le permet, comme Oracle ou PostgreSQL) pour implémenter les contraintes complexes non exprimables en SQL
   - Ajouter les **vues** utiles pour simplifier l'accès aux données en lecture
   - Implémenter des **fonctions stockées** (lorsque le SGBD le permet) permettant de renvoyer les valeurs calculées prévues par le modèle conceptuel
2. Développement de l'application : traitements, formulaires et états
   - Les traitements sont réalisés dans le langage applicatif choisi (PHP, Python, Java, C++...)
   - Les IHM sont parfois réalisées avec le même langage, mais de plus en plus souvent, on privilégie des IHM Web (HTML, CSS, JavaScript)

Exemple : Exemple de technologies

- BD :
  - SQL pour les tables et les vues
  - PL/SQL (Oracle), PL/pgSQL (PostgreSQL) pour les triggers et les fonctions stockées
- Formulaires
  - PHP, HTML (balise `<form>`), HTTP/POST (Web)
  - Formulaires Access
  - Input Python
- États
  - PHP , HTML (`<table>`), HTTP/GET (Web)
  - États Access

- Print Python

- Contrôle :
  - Python, PHP, JavaScript, Java, C++, Macros & VBA (Access)

b) Exemple d'application de base de données

Exemple : Modèle

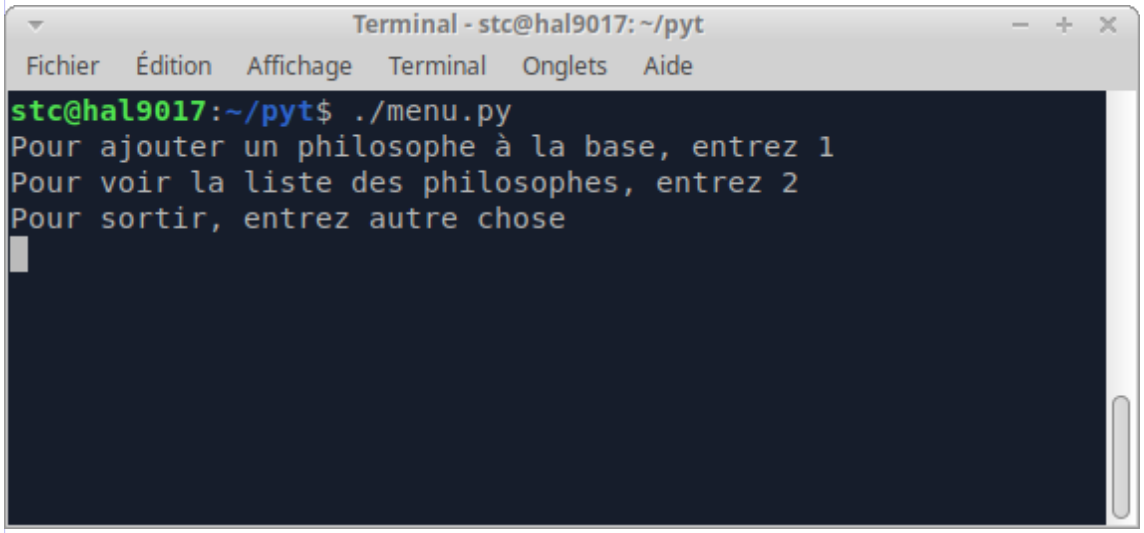
```
1  CREATE TABLE philosopher (
2  surname TEXT PRIMARY KEY,
3  givenname TEXT,
4  century INTEGER);
```

Exemple : Vue

```
1  CREATE VIEW v_philosopher (name, century) AS
2  SELECT
3    COALESCE(givenname,'') ||
4    CASE WHEN givenname IS NOT NULL THEN ' ' ELSE '' END
5    || surname,
6    century || 'e'
7  FROM philosopher;
```

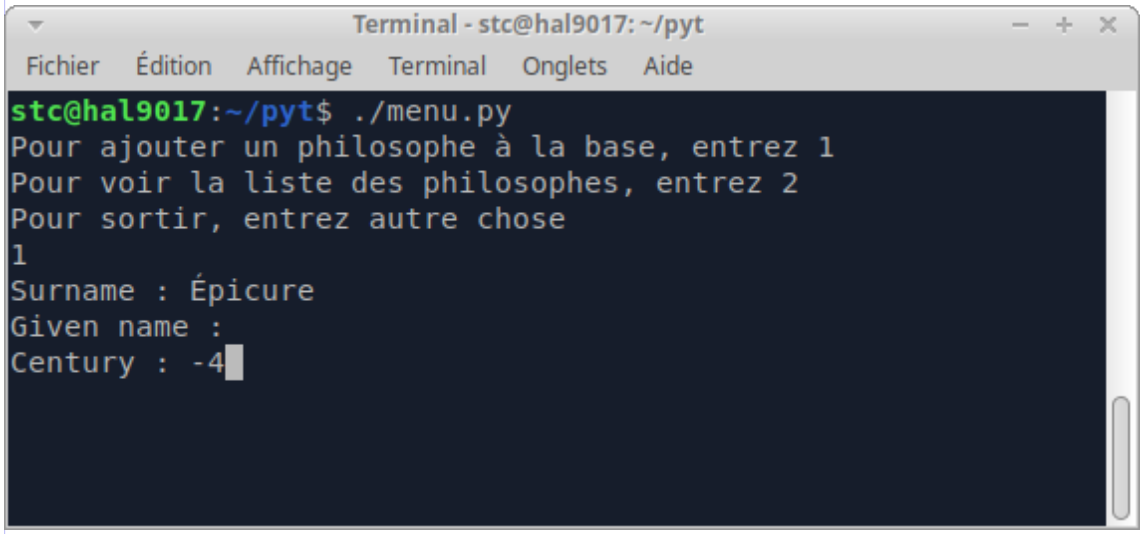
Exemple : Contrôle (menu.py)

```
1  #!/usr/bin/python3
2
3  import psycopg2
4  import forms
5  import reports
6
7  HOST = "localhost"
8  USER = "me"
9  PASSWORD = "secret"
10 DATABASE = "mydb"
11
12 conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
    (HOST, DATABASE, USER, PASSWORD))
13
14 choice = '1'
15 while choice == '1' or choice == '2':
16   print ("Pour ajouter un philosophe à la base, entrez 1")
17   print ("Pour voir la liste des philosophes, entrez 2")
18   print ("Pour sortir, entrez autre chose")
19   choice = input()
20   if choice == '1':
21     forms.addPhilo(conn)
22   if choice == '2':
23     reports.printPhilo(conn)
24   print(choice)
25
26 conn.close()
```



Exemple : Formulaire

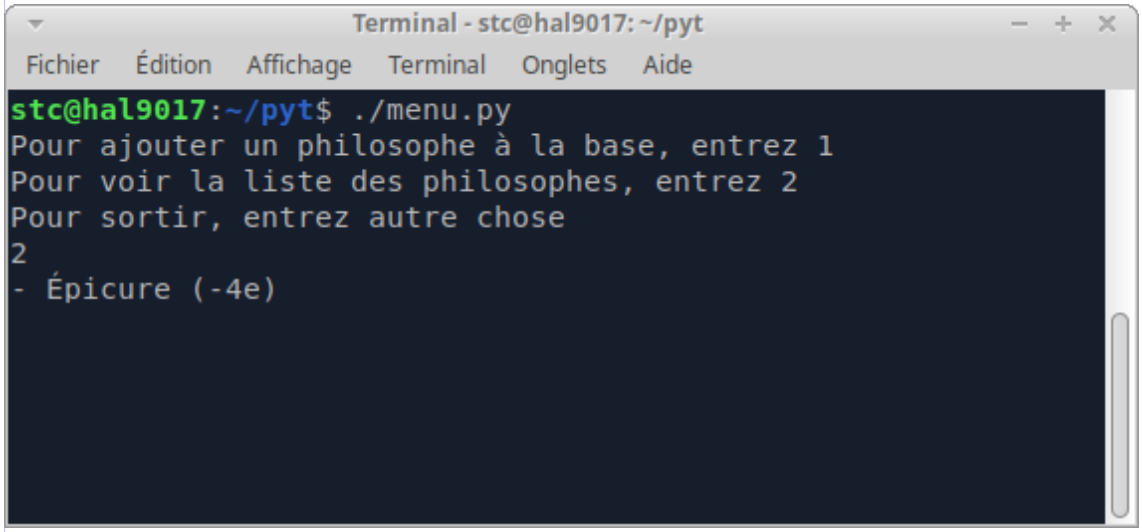
```
1  #!/usr/bin/python3
2
3  def quote(s):
4    if s:
5      return '\'%s\'' % s
6    else:
7      return 'NULL'
8
9  def addPhilo(conn):
10   surname = quote(input("Surname : "))
11   givenname = quote(input("Given name : "))
12   century = int(input("Century : "))
13   # Connect, execute SQL, close
14   cur = conn.cursor()
15   sql = "INSERT INTO philosopher VALUES (%s, %s, %i)" % (surname,
     givenname, century)
16   cur.execute(sql)
17   conn.commit()
```



Exemple : État (report.py)

```
#!/usr/bin/python3

def printPhilo(conn):
  # Connect and retrieve data
  cur = conn.cursor()
  sql = "SELECT name, century FROM v_philosopher"
  cur.execute(sql)
  # Fetch data line by line
  raw = cur.fetchone()
  while raw:
    print ("- %s (%s)" % (raw[0], raw[1]))
    raw = cur.fetchone()
```



c) Méthode générale d'accès à une BD en écriture par un langage de programmation

Méthode

1. Connexion à la base de données et récupération d'un identifiant de connexion
2. Écriture de la requête d'insertion ou de mise à jour de données
3. Exécution de la requête sur la connexion ouverte (et éventuelle récupération d'un résultat d'exécution, par exemple : le nombre de lignes modifiés, ou un booléen selon que la requête s'est exécutée ou non avec succès)
4. Test du résultat et dialogue avec l'utilisateur an cas d'erreur ou de résultat incorrect
5. Clôture de la connexion

Méthode : Pseudo-code

```
// Connexion à la base de données
host = "foo.fr"
port = "6666"
data  = "myDatabase"
user = "me"
pass = "secret"
conn = CONNECT(host, port, data, user, pass)
```

```
// Connexion à la base de données
host = "foo.fr"
port = "6666"
data  = "myDatabase"
user = "me"
pass = "secret"
conn = CONNECT(host, port, data, user, pass)
```

```
// Écriture de la requête
sql= "INSERT INTO t (a) VALUES (1) ;"
```

```
// Exécution de la requête
result = QUERY(conn, sql)
```

```
// Test du résultat
IF (NOT result) THEN MESSAGE("Échec de l'exécution")
```

```
// Clôture de la connexion
CLOSE (conn)
```

d) Exemple d'accès à une BD en écriture par un langage de programmation

Exemple : Python (sans gestion d'erreur)

```
#!/usr/bin/python3

import psycopg2

HOST = "localhost"
USER = "me"
PASSWORD = "secret"
DATABASE = "mydb"

# Open connection
conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
 (HOST, DATABASE, USER, PASSWORD))

# Open a cursor to send SQL commands
cur = conn.cursor()

# Execute a SQL INSERT command
sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
cur.execute(sql)
conn.commit()

# Close connection
conn.close()
```

Remarque

Ce code n'intègre aucune gestion d'erreur. En Python on pourra introduire une gestion d'exceptions pour traiter les cas d'erreur (tentative d'insertion d'un doublon dans une clé par exemple).

e) Méthode générale d'accès à une BD en lecture par un langage de programmation

Définition : Pointeur sur un résultat de requête

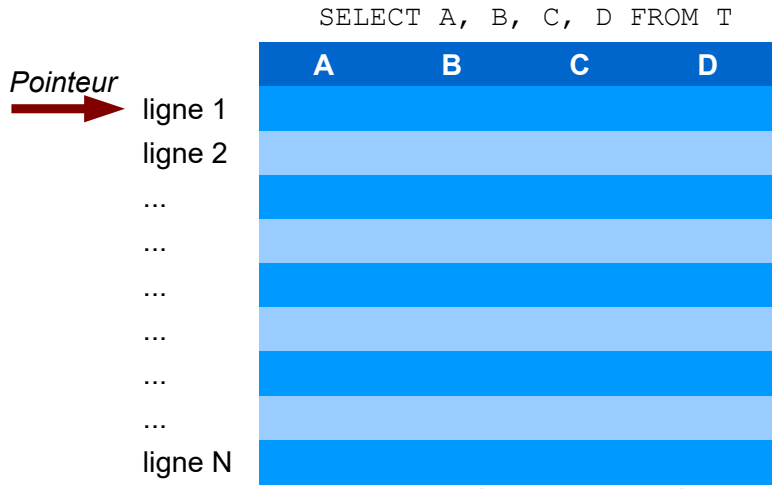
Lorsqu'un programme exécute une requête dans une BD, il récupère un pointeur sur un résultat de requête permettant de parcourir le tableau résultant de l'exécution de la requête ligne par ligne.

En effet :

- Une requête SQL retourne toujours un tableau (même si parfois il n'a qu'une ligne et une colonne), c'est le fondement du modèle relationnel
- En général il n'est pas souhaitable que ce tableau soit transmis directement sous la forme d'une structure en mémoire (`array`), car le résultat d'une requête peut être très volumineux (et ne pas tenir en mémoire primaire)
- La solution générale est donc que la BD fournisse le résultat ligne par ligne

Le pointeur permet de :

- Retourner la ligne pointée sous la forme d'un vecteur (tableau à une seule ligne)
- Passer à la ligne suivante
- Savoir si l'on a atteint la fin du tableau (ligne N)



Graphique 7 Pointeur sur un résultat de requête

Méthode

1. Connexion à la base de données et récupération d'un identifiant de connexion
2. Écriture de la requête de sélection
3. Exécution de la requête sur la connexion ouverte et récupération d'un pointeur sur un résultat de requête
4. Parcours du pointeur dans une boucle permettant de rapporter et traiter (afficher par exemple) le tableau ligne par ligne
5. Clôture de la connexion

Méthode : Pseudo-code

```
1  // Connexion à la base de données
2  host = "foo.fr"
3  port = "6666"
4  data  = "myDatabase"
```

```
5  user = "me"
6  pass = "secret"
7  conn = CONNECT(host, port, data, user, pass)
```

```
1  // Écriture de la requête
2  sql = "SELECT a, b FROM  t;"
```

```
1  // Exécution de la requête
2  pointeur = QUERY (conn, sql)
```

```
1  // Traitement du résultat
2  WHILE (pointeur)
3      FETCH pointeur IN result
4      PRINT result[1]
5      PRINT result[2]
6      NEXT pointeur
7  END WHILE
```

```
1  // Clôture de la connexion
2  CLOSE (conn)
```

f) Exemples d'accès à une BD en lecture par un langage de programmation

Exemple : Python (sans gestion d'erreur)

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  HOST = "localhost"
6  USER = "me"
7  PASSWORD = "secret"
8  DATABASE = "mydb"
9
10 # Open connection
11 conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
    (HOST, DATABASE, USER, PASSWORD))
12
13 # Open a cursor to send SQL commands
14 cur = conn.cursor()
15
16 # Execute a SQL SELECT command
17 sql = "SELECT name, century FROM v_philosopher"
18 cur.execute(sql)
19
20 # Fetch data line by line
21 raw = cur.fetchone()
22 while raw:
23   print (raw[0], raw[1])
24   raw = cur.fetchone()
```

Remarque

Ce code n'intègre aucune gestion d'erreur. En Python on pourra introduire une gestion d'exceptions pour traiter les cas d'erreur (requête SELECT mal formée par exemple).

2. Application avec Python et PostgreSQL

a) Psycopg : module Python pour PostgreSQL

Attention : Pilote de SGBD

Un langage de programmation a besoin d'un pilote (*driver*) spécifique pour interagir avec un SGBD. L'usage de ce pilote varie selon les langages et les pilotes, mais c'est une opération simple consistant à ajouter un librairie à son programme.

Définition

Pour connecter Python et PostgreSQL, il existe plusieurs pilotes (modules Python), on utilisera dans le cadre de ce cours le pilote *Psycopg2*.

Psycopg is a PostgreSQL adapter for the Python programming language. It is a wrapper for the libpq, the official PostgreSQL client library.

Syntaxe

```
1 #!/usr/bin/python3
2 import psycopg2
```

Fondamental : Documentation

https://www.psycopg.org/docs[28]

Méthode : Installation

```
1 pip3 install --user psycopg2-binary
```

Complément : Pour mettre à jour pip3

```
1 pip3 install --user --upgrade pip
```

Complément : Pour installer pip3 sous Debian

```
1 apt-get install python3-pip
```

28 - https://www.psycopg.org/docs

b) Connexion Python-PostgreSQL

Syntaxe

```
1  #!/usr/bin/python3
2  import psycopg2
3
4  # Connect to an existing database
5  conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
   ("localhost", "database", "user", "password"))
```

Exemple : Préparation de la base PostgreSQL

Préparer une base de données PostgreSQL *mydb* appartenant à l'utilisateur *me* avec le mot de passe *secret*. Ajouter une table *t* avec les attributs *a* et *b*.

```
1  #/usr/bin/bash
2  psql -c "CREATE USER me WITH PASSWORD 'secret';"
3  psql -c "CREATE DATABASE mydb OWNER=me";
4  psql -h localhost -U me -d mydb -c "CREATE TABLE t (a VARCHAR,b
   INTEGER)";
```

Exemple : Connexion à la base de données en Python (connect.py)

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  HOST = "localhost"
6  USER = "me"
7  PASSWORD = "secret"
8  DATABASE = "mydb"
9
10 # Connect to an existing database
11 conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
   (HOST, DATABASE, USER, PASSWORD))
12 print (conn)
13
14 # Close connection
15 conn.close()
16 print (conn)
```

```
1  $ ./connect.py
2  <connection object at 0x7f752e56fe88; dsn: 'user=me password=xxx
   dbname=mydb host=localhost', closed: 0>
3  <connection object at 0x7f752e56fe88; dsn: 'user=me password=xxx
   dbname=mydb host=localhost', closed: 1>
```

Complément : Documentation du module psycopg2

https://www.psycopg.org/docs/usage.html[29]

29 - https://www.psycopg.org/docs/usage.html

c) Écriture dans PostgreSQL avec Python

Exemple : Insertion (insert.py)

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  HOST = "localhost"
6  USER = "me"
7  PASSWORD = "secret"
8  DATABASE = "mydb"
9
10 # Open connection
11 conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
   (HOST, DATABASE, USER, PASSWORD))
12
13 # Open a cursor to send SQL commands
14 cur = conn.cursor()
15
16 # Execute a SQL INSERT command
17 sql = "INSERT INTO t VALUES ('Hello',1)"
18 cur.execute(sql)
19
20 # Commit (transactionnal mode is by default)
21 conn.commit()
22
23 # Close connection
24 conn.close()
```

```
1  $ ./insert.py
2  [('Hello', 1)]
```

Complément : Documentation du module psycopg2

https://www.psycopg.org/docs/usage.html[30]

d) Exercice

Soit une base de données composée d'une unique table t, contenant un unique attribut a de type chaîne de caractère.

```
1  CREATE TABLE t (a TEXT PRIMARY KEY);
```

Compléter le programme Python permettant d'ajouter la valeur something dans la base de données implémentée sous PosgreSQL.

#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" % ("localhost", "mydb", "me", "secret"))

cur = conn.cursor()

sql = "INSERT INTO VALUES ( )"

.execute( )

conn.

conn.

30 - https://www.psycopg.org/docs/usage.html

e) Lecture de PostgreSQL avec Python

Exemple : Sélection (select.py)

```
#!/usr/bin/python3

# http://initd.org/psycopg/docs/usage.html

import psycopg2

HOST = "localhost"
USER = "me"
PASSWORD = "secret"
DATABASE = "mydb"

# Open connection
conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
 (HOST, DATABASE, USER, PASSWORD))

# Open a cursor to send SQL commands
cur = conn.cursor()

# Execute a SQL SELECT command
sql = "SELECT * FROM t"
cur.execute(sql)

# Fetch data line by line
raw = cur.fetchone()
while raw:
    print (raw[0])
    print (raw[1])
    raw = cur.fetchone()

# Close connection
conn.close()
```

```
$ ./select.py
Hello
1
```

Complément : Documentation du module psycopg2

https://www.psycopg.org/docs/usage.html[31]

f) Exercice

Soit une base de données composée d'une unique table t, contenant un unique attribut a.

```
CREATE TABLE t (a TEXT PRIMARY KEY);
```

Compléter le programme Python permettant d'interroger la base de données implémentée sous PosgreSQL.

#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" % ("localhost", "mydb", "me", "secret"))

cur = conn.cursor()

31 - https://www.psycopg.org/docs/usage.html

```
sql = "SELECT          FROM          "
          .execute(          )
data =
while          :
  print(          )
  data =
conn.
```

B. Exercice

1. Country Python I

[45 min]

Soit la base de données *country* définie par le code SQL ci-après.

```sql
CREATE TABLE country (
name VARCHAR,
gdp INTEGER,
PRIMARY KEY (name)
);

CREATE TABLE region (
country VARCHAR,
name VARCHAR,
population INTEGER,
area INTEGER,
PRIMARY KEY (country, name),
FOREIGN KEY (country) REFERENCES country(name)
);

CREATE TABLE log (
id INTEGER,
country VARCHAR NOT NULL,
region VARCHAR NOT NULL,
date DATE NOT NULL,
temperature DECIMAL(3,1) NOT NULL,
humidity DECIMAL(3,1) NOT NULL,
PRIMARY KEY (id),
UNIQUE (country, region, date),
FOREIGN KEY (country, region) REFERENCES region (country, name)
);
```

```sql
INSERT INTO country VALUES ('France', 38128);
INSERT INTO country VALUES ('Norway', 70392);
INSERT INTO country VALUES ('Spain', 26609);

INSERT INTO region VALUES ('France','Hauts-de-France', 5973098,
31813);
INSERT INTO region VALUES ('France','Île-de-France', 12005077,
12012);
INSERT INTO region VALUES ('France','Normandie', 3322757, 29906);

INSERT INTO log VALUES (1, 'France','Hauts-de-France',
TO_DATE('23052017','DDMMYYYY'),21.1,51.4);
INSERT INTO log VALUES (2, 'France','Hauts-de-France',
TO_DATE('24052017','DDMMYYYY'),23.1,62.4);
INSERT INTO log VALUES (3, 'France','Hauts-de-France',
```

```
    TO_DATE('25052017','DDMMYYYY'),17.5,71.1);
```

Question 1

Écrire un programme Python qui affiche la liste des régions de la base de données, en les triant par ordre inverse de densité de population (la densité est la population divisée par la surface).

Question 2

Proposez un programme Python qui affiche *Yes* s'il existe un pays qui commence par *Fr*, et *No* sinon.

Question 3

Proposez un programme Python qui affiche tous les pays qui commencent par *Fr*.

Application de bases de données avec Python

VI

A. Cours

1. Gestion des erreurs SQL

a) Notion d'exception

Définition : Exception

Une exception ou erreur est générée automatiquement par le programme Python lorsque que quelque chose se passe mal, par exemple :

- lorsqu'on essaye de se connecter à une base de données qui n'existe pas
- lorsqu'on essaie d'insérer une clé qui existe déjà.
- lorsqu'on insère une valeur du mauvais type
- ...

Par défaut une exception interrompt immédiatement le programme.

Exemple : Insertion de valeur du mauvais type

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
   password='secret'")
6  cur = conn.cursor()
7
8  sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, 'Quatrième
   avant JC')"
9  cur.execute(sql)
10 conn.commit()
11
```

```
12 print("Fin du programme")
13
14 conn.close()
```

```
1 Traceback (most recent call last):
2   File "./sortie1.py", line 14, in <module>
3     cur.execute(sql)
4 psycopg2.DataError: invalid input syntax for integer: "Premier"
5 LINE 1: INSERT INTO philosopher VALUES ('Épicure', NULL, 'Premier')
```

Attention

La ligne `print("Fin du programme")` n'est jamais atteinte, l'exception a entraîné l'interruption du programme.

b) Gestion d'exception

Méthode

Il est possible d'interception une exception et ainsi :

- empêcher le programme de s'arrêter ;
- et faire des traitements personnalisés en cas d'erreur.

Syntaxe : Try/Except

```
1 try:
2   # code pouvant générer une erreur
3 except type_de_l_erreur as e:
4   # traitements spécifiques
5   print(e) # affichage de l'erreur
```

Exemple : Insertion de valeur du mauvais type (psycopg2.DataError)

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
   password='secret'")
6  cur = conn.cursor()
7
8  try:
9    sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, 'Premier')"
10   cur.execute(sql)
11   conn.commit()
12 except psycopg2.DataError as e:
13   print("Message personnalisé : Contrainte non respectée")
14   print("Message système :", e)
15
16 print("Fin du programme")
17
18 conn.close()
```

```
1 Message personnalisé : Contrainte non respectée
2 Message système : invalid input syntax for integer: "Premier"
3 LINE 1: INSERT INTO philosopher VALUES ('Épicure', NULL, 'Premier')
4                                                          ^
5 Fin du programme
```

c) Exceptions et transactions

Rappel

Transactions en SQL

Exemple : Duplication de clé

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
   password='secret'")
6  cur = conn.cursor()
7
8  sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
9  cur.execute(sql)
10 conn.commit()
11
12 sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
13 cur.execute(sql)
14 conn.commit()
15
16 print("Fin du programme")
17
18 conn.close()
```

```
1  Traceback (most recent call last):
2    File "./sortie1.py", line 18, in <module>
3      cur.execute(sql)
4  psycopg2.IntegrityError: duplicate key value violates unique
   constraint "philosopher_pkey"
5  DETAIL:  Key (surname)=(Épicure) already exists.
```

Exemple : Duplication de clé (gestion de l'exception)

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
   password='secret'")
6  cur = conn.cursor()
7
8  try:
9    sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
10   cur.execute(sql)
11   conn.commit()
12 except psycopg2.IntegrityError as e:
13   print("Message système :", e)
14
15 try:
16   sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
17   cur.execute(sql)
18   conn.commit()
19 except psycopg2.IntegrityError as e:
20   print("Message système :", e)
21
22 print("Fin du programme")
23
24 conn.close()
```

```
1  Message système : duplicate key value violates unique constraint
   "philosopher_pkey"
2  DETAIL:  Key (surname)=(Épicure) already exists.
3
4  Traceback (most recent call last):
5    File "./sortie1.py", line 22, in <module>
6      cur.execute(sql)
7  psycopg2.InternalError: current transaction is aborted, commands
   ignored until end of transaction block
```

Attention

L'erreur de duplication est bien gérée, mais une seconde exception est levée car lors de la seconde insertion, la première transaction est encore en cours, car l'instruction `conn.commit()` ne s'est pas exécutée.

Méthode

On va ajouter un ordre de *rollback* pour terminer la transaction en cas d'erreur.

Syntaxe : Try/Except

```
try:
  # code pouvant générer une erreur
except type_de_l_erreur as e:
  # traitements spécifiques
  print(e) # affichage de l'erreur
  conn.rollback()
```

Exemple

```
#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
 password='secret'")
cur = conn.cursor()

try:
  sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
  cur.execute(sql)
  conn.commit()
except psycopg2.IntegrityError as e:
  conn.rollback()
  print("Message système :", e)

try:
  sql = "INSERT INTO philosopher VALUES ('Épicure', NULL, -4)"
  cur.execute(sql)
  conn.commit()
except psycopg2.IntegrityError as e:
  conn.rollback()
  print("Message système :", e)

print("Fin du programme")

conn.close()
```

2. Bonne pratiques

a) Insérer des entiers

Méthode

La fonction *input* retourne une chaîne de caractère.
Afin d'obtenir des entiers on utilise la fonction de conversion de type `int()`.

Syntaxe : Saisie d'un entier

```
#!/usr/bin/python3

i = int(input("Entrez un entier : "))
```

Exemple

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
   password='secret'")
6  cur = conn.cursor()
7
8  century = int(input("Entrez un entier : "))
9
10 try:
11   sql = "INSERT INTO philosopher VALUES ('Platon', NULL, %i)" %
   century
12   cur.execute(sql)
13   conn.commit()
14 except psycopg2.IntegrityError as e:
15   conn.rollback()
16   print("Message système :", e)
17
18 conn.close()
```

Attention

Si la valeur entrée n'est pas convertible en entier, alors le programme lève une exception (et s'interrompt si l'exception n'est pas gérée).

3. Compléments

a) INSERT, DELETE, UPDATE avec valeur de retour

La clause SQL RETURNING à la fin d'une instruction INSERT, UPDATE ou DELETE, permet de retourner un résultat de requête (qui peut être lors traité comme le résultat d'un SELECT).

Syntaxe

```
1  sql = "INSERT INTO t (a, b, c) VALUES (1, 2, 3) RETURNING a"
2  cur.execute(sql)
3  res = cur.fetchone()[0]
```

Exemple

```
1  #!/usr/bin/python3
2
3  import psycopg2
4
5  conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
   password='secret'")
6  cur = conn.cursor()
7
8  try:
9    sql = "INSERT INTO philosopher VALUES ('Aristote', NULL, -4)
   RETURNING surname"
10   cur.execute(sql)
11   res = cur.fetchone()[0]
12   print(res)
13   conn.commit()
```

```
14  except psycopg2.IntegrityError as e:
15    conn.rollback()
16    print("Message système :", e)
17
18  conn.close()
```

Méthode

Cette fonction est très utile pour récupérer la valeur d'une clé artificielle générée automatiquement (par une séquence typiquement).

b) Créer un menu

Exemple

```
#!/usr/bin/python3

import psycopg2
import forms
import reports

HOST = "localhost"
USER = "me"
PASSWORD = "secret"
DATABASE = "mydb"

conn = psycopg2.connect("host=%s dbname=%s user=%s password=%s" %
 (HOST, DATABASE, USER, PASSWORD))

choice = '1'
while choice == '1' or choice == '2':
  print ("Pour ajouter un philosophe à la base, entrez 1")
  print ("Pour voir la liste des philosophes, entrez 2")
  print ("Pour sortir, entrez autre chose")
  choice = input()
  if choice == '1':
    forms.addPhilo(conn)
  if choice == '2':
    reports.printPhilo(conn)
  print(choice)

conn.close()
```

c) Fetchall

La méthode *Fetchall* est une alternative à *Fetchone*, qui permet de récupérer d'un coup l'ensemble du résultat d'une requête. Lorsque le volume attendu est suffisamment faible pour tenir en mémoire, c'est une alternative intéressante, le résultat est plus facile à manipuler, et les performances sont améliorées (suppression des multiples accès pour accéder aux données une par une).

Exemple

```
#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
 password='secret'")
cur = conn.cursor()

sql = "SELECT name, century FROM v_philosopher"
cur.execute(sql)
res = cur.fetchall()

print(res)
```

```
[('Aristote', '-4e'), ('Epicure', '-4e'), ('Platon', '-5e')]
```

Exemple

```
#!/usr/bin/python3
```

```
2
3   import psycopg2
4
5   conn = psycopg2.connect("host='localhost' dbname='mydb' user='me'
    password='secret'")
6   cur = conn.cursor()
7
8   sql = "SELECT name, century FROM v_philosopher"
9   cur.execute(sql)
10  res = cur.fetchall()
11
12  for raw in res:
13    print (raw[0], raw[1])
```

```
1   Aristote -4e
2   Epicure -4e
3   Platon -5e
```

B. Exercice

1. Country Python II

Question

Améliorer le programme Python ci-après :

- Intégrer une gestion d'exception pour la phase de connexion
- Utiliser *fetchall* à la place de *fetchone*
- Afficher – `Île-de-France (d=999)` à la place de `Île-de-France 999`

```
1   #!/usr/bin/python3
2
3   import psycopg2
4
5   conn = psycopg2.connect("dbname='mydb' user='me' host='localhost'
    password='secret'")
6
7   sql = "SELECT name, population/area AS density FROM region ORDER BY
    density DESC";
8
9   resultat = conn.cursor()
10  resultat.execute(sql)
11
12  ligne = resultat.fetchone()
13  while ligne:
14          name = ligne[0]
15          density = ligne[1]
16          print(name, density)
17          ligne = resultat.fetchone()
```

Introduction aux bases de données non-relationnelles

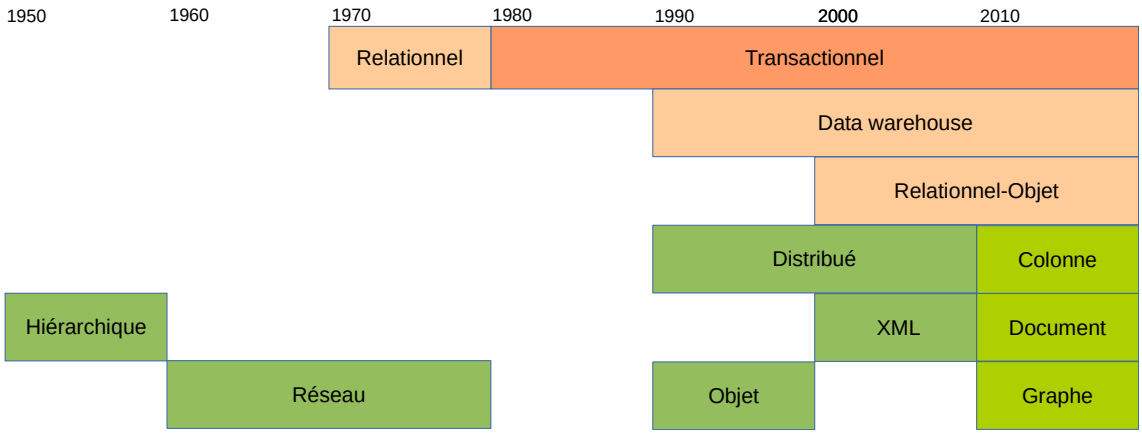
VII

A. Cours

« Un couple de concepteurs de bases de données entre dans un restaurant NoSQL. Une heure après, ils ressortent sans avoir réussi à manger ; ils n'ont jamais réussi à trouver une seule table. »

1. Perspective technologique et historique : forces et faiblesses du relationnel

a) Relationnel et non-relationnel



Les BD NoSQL remettent en cause l'hégémonie des SGBDR telle qu'elle s'est constitué dans les années 1980.

Les BD NoSQL sont essentiellement un retour à des modèles de données antérieurs à cette hégémonie, telles que les représentations hiérarchique ou réseau qui existaient dans les années 60.

b) Domination du relationnel

Fondamental

La première fonction d'une base de données est de permettre de **stocker** et **retrouver** l'information.

Relationnel et contrôle de l'intégrité des données

À la naissance de l'informatique, plusieurs modèles de stockage de l'information sont explorés, comme les modèles hiérarchique ou réseau.

Mais c'est finalement le modèle relationnel qui l'emporte dans les années 1970 car c'est lui qui permet de mieux assurer le contrôle de l'intégrité des données, grâce à un modèle théorique puissant et simple.

On notera en particulier :

- Le **schéma** : on peut exprimer des règles de cohérence a priori et déléguer leur contrôle au système.
- La **normalisation** : on peut contrôler des contraintes (*via* le contrôle de la redondance) par un mécanisme de décomposition et retrouver l'information consolidée par les jointures.
- La **transaction** : le système assure le maintien d'états cohérents au sein d'environnements concurrents et susceptibles de pannes.

Relationnel et performance en contexte transactionnel

La représentation relationnelle se fonde sur la décomposition de l'information ce qui minimise les entrées/sorties (accès disques, transfert réseau) et permet d'être très performant pour répondre à des questions et des mises à jour ciblées (qui concernent peu de données parmi un ensemble qui peut être très grand). C'est donc une bonne solution dans un contexte transactionnel qui comprend de nombreux accès ciblés à la base.

En revanche ce n'est plus une bonne solution pour des accès globaux à la base (puisqu'il faut alors effectuer beaucoup de jointures pour reconsolider l'ensemble de l'information). C'est par exemple le problème posé notamment par le décisionnel.

2. Au delà des bases de données relationnelles : Data warehouse, XML et NoSQL

a) Problème de l'agrégat et des data warehouses

Fondamental : Problème posé par le décisionnel et résolu par les data warehouses

- Décision vs Gestion
- Agrégat vs Accès ciblé
- Historisation vs Transaction

Définition

Un système décisionnel est une application informatique destinée à effectuer des exploitations statistiques sur la base des données existantes dans une organisation

dans le but d'aider à la prise de décision.

- Le modèle relationnel est peu performant pour les agrégats qui portent sur de nombreuses tables car il est nécessaire de faire des jointures qui sont coûteuses.
- La rigueur du modèle relationnel n'est pas nécessaire pour des traitement statistiques qui sont tolérants aux incohérences isolées.
- Les volumes de données peuvent devenir importants si l'on conserve l'historique de toutes les transactions.

Exemple : Agrégat

Une étude statistique peut rapidement concerner plusieurs dizaines de tables dans une base de données relationnelles et donc exiger autant de jointures.

Exemple : Tolérance aux incohérences

Il n'est pas acceptable de perdre des données en contexte transactionnel (je ne sais pas si une personne existe ou pas), mais ce n'est pas important si je travaille sur une moyenne (l'âge moyen des personnes dans mon système ne sera pas impacté s'il me manque un enregistrement).

Exemple : Volume de données

Si le système produit 1.000 enregistrements chaque jour et que je les conserve pendant 3 ans, j'ai 1.000.000 de lignes (mon système change d'ordre de grandeur).

- Un data warehouse est une base de données dédiée à un système décisionnel.
- Les problèmes d'agrégat, de tolérance aux incohérences et de volumes de données sont adressés par les data warehouses.
- Pour cela les data warehouses se basent sur des modèles **fortement redondants** et **potentiellement localement incohérents**.

Complément

Décisionnel

Différence entre un DW et un système transactionnel

Objectifs du modèle dimensionnel

b) Problème de l'imbrication et des structures arborescentes (XML ou Json)

Si on manipule des structures de données imbriquées dans une application, comme par exemple des structures arborescentes exprimées en XML ou en Json ou encore des documents HTML, il est difficile de stocker les informations dans une base de données relationnelle.

Fondamental : Imbrication

La représentation d'un arbre profond en relationnel implique la création de nombreuses tables et donc la création d'un modèle complexe (éloigné de la réalité modélisée), et pouvant poser des problèmes de performance.

Exemple : Comparaison représentation XML versus Relationnel

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module>
    <titre>Introduction au non-relationnel</titre>
    <publication>10 avril 2018</publication>
    <licence>CC BY-SA</licence>
    <activite>
        <titre>Cours</titre>
        <section>
            <titre>Au delà du relationnel</titre>
            <page>
                <titre>Problème de l'imbrication</titre>
                <entete>
                    <auteur>Stéphane Crozat</auteur>
                    <date>11 avril 2018</date>
                </entete>
                <introduction>
                    <paragraphe>Si on manipule des structures de données imbriquées dans une
                        application...</paragraphe>
                    <paragraphe>En effet la représentation d'un arbre profond...</paragraphe>
                </introduction>
                <exemple>
                    <xml>
                        <fichier>exemple01.xml</fichier>
                    </xml>
                </exemple>
            </page>
        </section>
    </activite>
</module>

```

```
Module (#titre:varchar, publication:date, licence:varchar)
Activite (#titre:varchar, #module=>Module)
Section (#titre:varchar, #activite=>Activite)
Page (#titre:varchar, #Section=>Section)
Entete (#page=>Page, auteur:varchar, date:date)
Bloc (#id:int, page=>Page, type:{introduction|exemple})
Paragraphe (#id:int, bloc=>Bloc, texte:CLOB)
Xml (#id:int, bloc=>Bloc, fichier:varchar)
```

Fondamental : Impedance mismatch

Par ailleurs les données étant manipulées en mémoire sous une certaine structure et en base sous une autre, il faut "jongler" entre plusieurs représentations et écrire du code de conversion ce qui est source de complexification et de risque d'erreur dans le processus de programmation.

c) Problème de l'identification et des structures en graphe

Si on manipule des données représentées en graphe, alors l'identification par les données implique de faire une requête pour chaque recherche dans le graphe, ce qui est potentiellement coûteux.

Exemple

Soit une application Web reposant sur un graphe d'utilisateurs et qui souhaite afficher sur chaque page consultée (une page concerne un utilisateur) les utilisateurs qui lui sont connectés.

Si l'application repose sur une base de données relationnelle, elle devra à chaque

accès à une page faire une requête au serveur pour chercher qui sont les utilisateurs connectés par le graphe.

1. J'accède à Alice : je fais une requête qui remonte les données relatives à Alice et aux utilisateurs qui lui sont liés (Bob et Charlie)
2. J'accède à Bob : je dois faire une nouvelle requête pour obtenir les données relatives aux utilisateurs liés à Bob

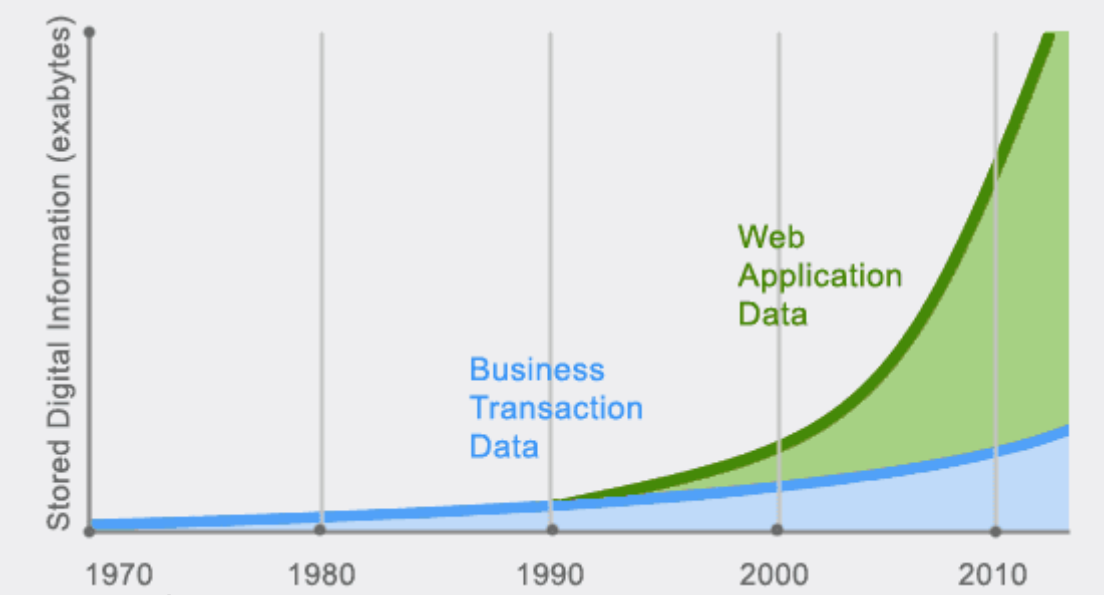
```
SELECT *
FROM links l JOIN users u
ON l.user2=u.id
WHERE l.user1='bob'
```

Fondamental

Si les données étaient identifiables selon une autre méthode, par exemple ici, si chaque *utilisateur* disposait d'un pointeur sur l'adresse physique de chacun de ses *utilisateurs* liés, alors on pourrait obtenir ces données beaucoup facilement.

d) Problème des gros volumes et de la distribution

Big data



Évolution des volumes de données d'entreprise versus web

Fondamental : Distribution

Passage du serveur central (*main frame*) à des grappes de machines modestes :

- pour gérer l'explosion des données
- à cause de l'évolution du *hardware*

Problème de la performance

La performance de jointures exécutées sur plusieurs machines est mauvaise, or la normalisation augmente les jointures.

Problème de la gestion du contrôle d'intégrité et des transactions

il faut intervenir sur plusieurs machines en réseau à chaque modification des données et synchroniser les informations entre ces machines.

Problème de l'identification unique par les données

Comment savoir qu'une autre instance n'est pas en train d'affecter une même donnée clé.

Attention

Ces questions doivent être traitées sachant que les machines tombent en panne et que les réseaux sont victimes de coupures : est-ce que tout est mis en attente chaque fois qu'un élément du système dysfonctionne ?

- Si oui, plus le système est grand plus il est fragile, ce n'est pas tenable.
- Si non, on aura des incohérences, est-ce tenable ?

Fondamental

Il y a une équation à résoudre entre les paramètres de distribution, performance et cohérence.

e) Synthèse : De la 1NF à la NFNF (NF²)

Rappel : 1NF

La 1NF (*first normal form*) pose :

1. Les tables ont une clé.
2. Les attributs sont atomiques.

Fondamental : Identification : les tables ont une clé

La première règle de la 1NF fixe le système d'identification des données en relationnel : ce sont les **données** qui permettent d'identifier les données.

C'est à dire que c'est un sous ensemble des données (une clé) qui permet d'identifier un enregistrement.

Or :

- Il existe d'autres solutions, par exemple l'identification par adresse mémoire (pointeur) ou disque, par URI, par UUID.
- Les clés artificielles sont déjà un échappatoire à cette règle puisqu'elles permettent l'identification avec une donnée spécifiquement créée pour l'identification, extérieure aux données.

Fondamental : Imbrication : les attributs sont atomiques.

La seconde règle de la 1NF fixe le système de représentation des données en relationnel : la seule structure reconnue est la **case** d'un tableau.

C'est à dire qu'une case ne peut pas avoir de structure propre.

Or :

- Il existe d'autres structures de données : les arbres, les tableaux...
- Il existe déjà des types structuré ajouté au SQL depuis la version 1, comme la date introduite dès SQL2 en 1992.

Fondamental : Non First Normal Form (NFNF ou NF²)

Les système en *Non First Normal Form* ou NF² sont des systèmes qui rompent volontairement avec la 1NF pour dépasser les limites qu'elle impose.

- Les attributs ne sont plus atomiques

- L'identification ne se fait plus par les données

Exemple

Les systèmes relationnels-objets et NoSQL sont des systèmes NF².

3. Bases de données NoSQL

a) Définition du mouvement NoSQL

Définition

Le NoSQL regroupe de nombreuses bases de données, récentes pour la plupart, qui se caractérisent par une logique de représentation de données non relationnelle et qui n'offrent donc pas une interface de requêtes en SQL.

http://blog.xebia.fr/2010/04/21/nosql-europe-tour-dhorizon-des-bases-de-donnees-nosql/[32]

Attention

NoSQL signifie *Not Only SQL* et non pas *No SQL*, il s'agit de compléments aux SGBDR pour des besoins spécifiques et non de solutions de remplacement.

Exemple

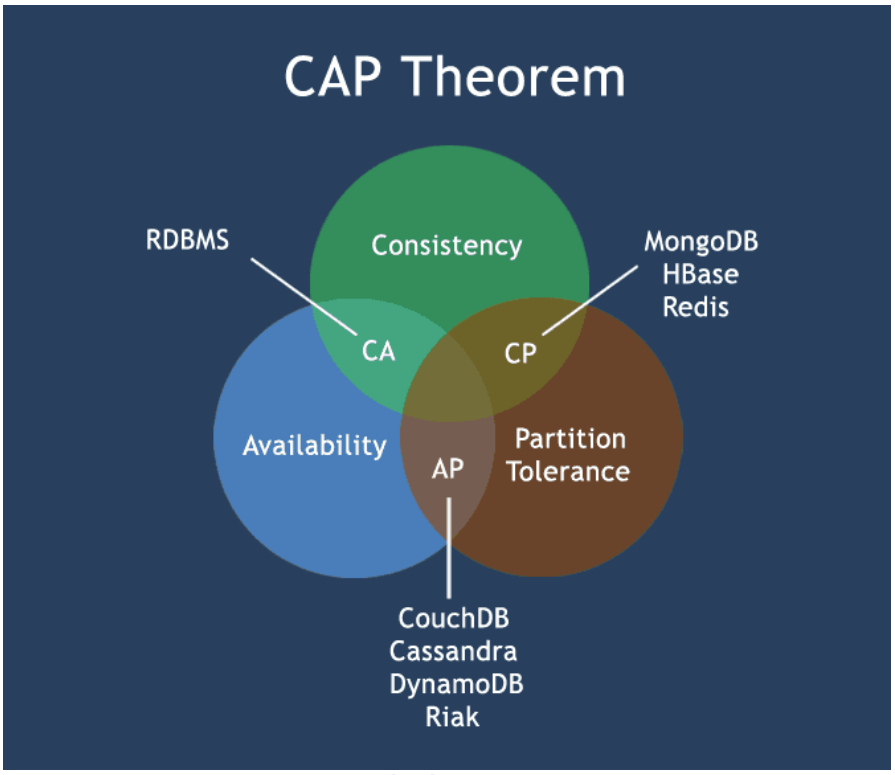
- BD orientée clé-valeur
- BD orientée graphe
- BD orientée colonne
- BD orientée document

Complément

http://blog.xebia.fr/2010/04/21/nosql-europe-tour-dhorizon-des-bases-de-donnees-nosql/[33]

32 - http://blog.xebia.fr/2010/04/21/nosql-europe-tour-dhorizon-des-bases-de-donnees-nosql/
33 - http://blog.xebia.fr/2010/04/21/nosql-europe-tour-dhorizon-des-bases-de-donnees-nosql/

b) "Théorème" CAP



Théorème CAP

Fondamental : Le "commerce" de la 3NF

On échange de la performance contre de la souplesse ou contre de la cohérence.

c) Fondamentaux des modèles NoSQL : Clé-valeur, distribution, imbrication, schema-less

Fondamental

- Logique de dépôt uniquement (stocker et retrouver) ;
- c'est la couche applicative qui fait tout le travail (traitement, cohérence...).

Modèle de stockage clé-valeur

Les données sont des valeurs auxquelles on accède par une clé artificielle.

Les données ne sont plus des lignes stockées dans des tables.

Identification

Les clés d'identification sur des clés artificielles, en général unique à l'échelle de la BD, voire du monde :

- Object Identifiers (OID)
- Universally Unique IDentifier (UUID)
- Uniform Resource Identifier (URI)

Pointeurs physiques

On stocke des pointeurs physiques pour relier les données.

Imbrication

Structure des valeurs stockées connue par le serveur (RO, JSON, XML, structure interne de type colonne...)

Partitionnement et distribution (sharding)

Les données sont partitionnées horizontalement et distribuées sur les nœuds d'un cluster.

On utilise par exemple une clé de hashage pour distribuer les données.

Schema-less

Les bases NoSQL se fondent sur une approche dite *schema-less*, c'est à dire sans schéma logique défini a priori.

L'équivalent du `CREATE TABLE` en SQL n'est soit pas nécessaire, soit même pas possible ; on peut directement faire l'équivalent de `INSERT INTO`.

Cela apporte de la souplesse et de la rapidité, mais se paye avec moins de contrôle et donc de cohérence des données.

Le mouvement NoSQL tend à réintégrer des fonctions de schématisation a priori, à l'instar de ce qui se fait en XML : le schéma est optionnel, mais conseillé en contexte de contrôle de cohérence.

d) Illustration des modèles NoSQL

Exemple : Représentation de ventes en relationnel

Table Sales

#ticket	#date	#book
1	01/01/16	2212121504
1	01/01/16	2212141556
2	01/01/16	2212141556

Table Book

#isbn	#title	#author
2212121504	Scenari	1
2212141556	NoSQL	2

Table Author

#id	surname	firstname
1		
2	Bruchez	Rudi

Exemple : Représentation de ventes en colonne

Family Sales

#ticket	date	books
1	01/01/16	2212121504 2212141556
2	01/01/16	2212141556

Family Book

#isbn	title	a-surname	a-firstname
2212121504	Scenari		
2212141556	NoSQL	Bruchez	Rudi

Exemple : Représentation de ventes en document

Collection Sales

#oid	
4d0407660766b236450b45a3	"ticket" : 1 "date" : "01/01/16" "books" : ["_id" : 2212121504 "_id" : 2212141556]
4d0407660766b236450b45a4	"ticket" : 2 "date" : "01/01/16" "books" : ["_id" : 2212141556]

Collection Book

#oid	
4d0407660766b236450b45a5	"isbn" : 2212121504 "title" : "Scenari"

4d0407660766b236450b45a6	"isbn" : 2212141556
	"title" : "NoSQL"
	"author" : {
	"surname" : Bruchez
	"firstname" : Rudi
	}

Exemple : Représentation de ventes en document (avec imbrication redondante)

Collection Sales

#oid	

4d0407660766b236450b45a3	"ticket" : 1
	"date" : "01/01/16"
	"books" : [
	{
	"isbn" : 2212121504
	"title" : "Scenari"
	}
	{
	"isbn" : 2212141556
	"title" : "NoSQL"
	"author" : {
	"surname" : Bruchez
	"firstname" : Rudi
	}
	}
	]

4d0407660766b236450b45a4	"ticket" : 2
	"date" : "01/01/16"
	"books" : [
	{
	"isbn" : 2212141556
	"title" : "NoSQL"
	"author" : {
	"surname" : Bruchez
	"firstname" : Rudi
	}
	}
	]

Exemple : Représentation de ventes en graphe

Classe Sales

#oid		
4d0407660766b236450b45a3	*property*	ticket : 1
	property	date : 01/01/16
	relation	book : 4d0407660766b236450b45a5
	relation	book : 4d0407660766b236450b45a6
4d0407660766b236450b45a4	*property*	ticket : 2
	property	date : 01/01/16
	relation	book : 4d0407660766b236450b45a6

Classe Book

#oid		
4d0407660766b236450b45a5	*property*	title : Scenari
4d0407660766b236450b45a6	*property*	title : NoSQL
	relation	author : 4d0407660766b236450b45a8

Classe Author

#oid		
4d0407660766b236450b45a8	*property*	surname : Bruchez
	property	firstnam : Rudi

4. Un exemple : Modélisation logique arborescente et objet en JSON

a) JavaScript Object Notation

Définition : Introduction

JSON est un format de représentation logique de données, hérité de la syntaxe de création d'objets en JavaScript.

C'est un format réputé léger (il ne contient pas trop de caractères de structuration), assez facilement lisible par les humains, facilement *parsable* par les machines, et indépendant des langages qui l'utilisent (sa seule fonction est de décrire des données, qui sont ensuite utilisées différemment pour chaque cas suivant le contexte).

Exemple : Un fichier JSON simple

```
1 {
2     "nom" : "Norris",
3     "prenom" : "Chuck",
4     "age" : 73,
5     "etat" : "Oklahoma"
6 }
```

Attention : JSON est Indépendant de tout langage

Bien que JSON puise sa syntaxe du JavaScript, il est **indépendant de tout langage de programmation**. Il peut ainsi être interprété par tout langage à l'aide d'un *parser*.

Complément : Extension

Un fichier au format JSON a pour extension "**.json**".

b) La syntaxe JSON en bref

Syntaxe : Règles syntaxiques

- Il ne doit exister qu'un seul élément père par document contenant tous les autres : **un élément racine**.
- Tout **fichier JSON bien formé** doit être :
  - **soit un objet** commençant par { et se terminant par },
  - **soit un tableau** commençant par [et terminant par].

  Cependant ils peuvent être vides, ainsi [] et {} sont des JSON valides.
- Les **séparateurs** utilisés entre deux paires/valeurs sont des **virgules**.
- Un objet JSON peut contenir d'autres objets JSON.
- Il ne peut pas y avoir d'éléments croisés.

Fondamental : Éléments du format JSON

Il existe deux types d'éléments :

- Des couples de type **"nom": valeur**, comme l'on peut en trouver dans les tableaux associatifs.
- Des listes de valeurs, comme les tableaux utilisés en programmation.

Définition : Valeurs possibles

- Primitifs : nombre, booléen, chaîne de caractères, null.
- Tableaux : liste de valeurs (tableaux et objets aussi autorisés) entrées entre crochets, séparées par des virgules.
- Objets : listes de couples "nom": valeur (tableaux et objets aussi autorisés) entrés entre accolades, séparés par des virgules.

Exemple

```
{
   "nom cours" : "NF29",
   "theme" : "ingenierie documentaire",
   "etudiants" : [
                  {
                    "nom" : "Norris",
                    "prenom" : "Chuck",
                    "age" : 73,
                    "pays" : "USA"
                  },
                  {
                    "nom" : "Doe",
                    "prenom" : "Jane",
                    "age" : 45,
                    "pays" : "Angleterre"
```

```
                    },
                    {
                      "nom" : "Ourson",
                      "prenom" : "Winnie",
                      "age" : 10,
                      "pays" : "France"
                    }
                ]
}
```

c) Principaux usages de JSON

Chargements asynchrones

Avec la montée en flèche des chargements asynchrones tels que l'AJAX (Asynchronous JavaScript And XML) dans le web actuel (qui ne provoquent pas le rechargement total de la page), il est devenu de plus en plus important de pouvoir charger des données organisées, de manière rapide et efficace.

Avec XML, le format JSON s'est montré adapté à ce type de besoins.

Les APIs

Des sociétés telles que Twitter, Facebook ou LinkedIn, offrent essentiellement des services basés sur l'échange d'informations, et font preuve d'un intérêt grandissant envers les moyens possibles pour distribuer ces données à des tiers.

Alors qu'il n'y a pas de domination totale d'un des deux formats (JSON ou XML) dans le domaine des APIs, on constate toutefois que JSON est en train de prendre le pas là où le format XML avait été pionnier.

Exemple : APIs retournant des données au format JSON

Twitter : *https://dev.twitter.com/rest/public*[34] : récupération de données du réseau social.

Netatmo : *https://dev.netatmo.com/doc/publicapi*[35] : récupération de données météo

Les bases de données

Le JSON est très utilisé dans le domaine des bases de données NoSQL (MongoDB, CouchDB, Riak...).

On notera également qu'il est possible de soumettre des requêtes à des SGBDR et de récupérer une réponse en JSON.

Exemple : Fonctions JSON de Postgres et MySQL

Fonctions PostgreSQL : *http://www.postgresql.org/docs/9.3/static/functions-json.html*[36]

Fonctions MySQL : *https://dev.mysql.com/doc/refman/5.7/en/json.html*[37]

d) Exercice

Exercice

34 - https://dev.twitter.com/rest/public
35 - https://dev.netatmo.com/doc/publicapi
36 - http://www.postgresql.org/docs/9.3/static/functions-json.html
37 - https://dev.mysql.com/doc/refman/5.7/en/json.html

Qu'est-ce que le JSON ?

- ○ Un langage de programmation orientée objet
- ○ Un type de pages web
- ○ Un format qui permet de décrire des données structurées
- ○ Une catégorie de documents générés par un traitement de texte

Exercice

A quel domaine le JSON n'est-il pas appliqué (à l'heure actuelle) ?

- ○ Le Big Data
- ○ Les chargements asynchrones
- ○ Les APIs
- ○ La mise en forme de pages web

Exercice

Un fichier JSON doit forcément être traité via du code JavaScript.

- ○ Vrai
- ○ Faux

Exercice

On peut utiliser des tableaux en JSON et en XML.

- ○ Vrai
- ○ Faux, on ne peut le faire qu'en XML
- ○ Faux, on ne peut le faire qu'en JSON
- ○ Faux, on ne peut le faire ni en XML ni en JSON

e) Un programme de traitement de JSON

Soit le fichier HTML et le fichier JavaScript ci-après.

```
1  <html>
2  <head>
3  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
4  <title>Démo JSON/JavaScript</title>
5  <script type="text/javascript" src="query.js"></script>
6  </head>
7  <body>
8  <input type="file" id="myfile" onchange="query()"/>
9  <div id="mydiv"/>
10 </body>
11 </html>
```

```
1  function query() {
2  // File
```

```
let vFile = document.getElementById("myfile").files[0];
// Reader
let vReader = new FileReader();
vReader.readAsText(vFile);
vReader.onload = function(pEvent) {
    // String Input
    let vContent = pEvent.target.result;
    // JSON to object
    let vJson = JSON.parse(vContent);
    // Query
    let vResult = vJson.prenom + " " + vJson.nom + " (" + vJson.age + ")";
    // Output

document.getElementById("mydiv").appendChild(document.createTextNode(vResult));
};
}
```

Question 1

Écrire un fichier JSON permettant de représenter une personne avec les attributs suivants :

- un nom
- un prénom
- un age
- une liste d'adresses mail

Indice :

La syntaxe JSON

Question 2

Expliquer ce que fait le programme. Tester le programme avec le fichier JSON proposé.

Question 3

Compléter la fonction *query()* afin d'afficher la liste des adresses mail (en plus des information actuelles).

Indice :

On parcours le tableau JSON à l'aide d'une boucle.

```
for (let i=0;i<vJson.adresse.length;i++) {
  ...
}
```

B. Exercice

1. Modélisation orientée document avec JSON

On souhaite réaliser une base de données orientée documents pour gérer des cours et des étudiants, étant données les informations suivantes :

- Un cours est décrit par les attributs code, titre, description, crédits et prérequis.
- Les prérequis sont d'autres cours.

- Un étudiant est décrit par les attributs nom, prénom, adresse.
- Les adresses sont composées d'un numéro de rue, d'une rue, d'une ville et d'un code postal.
- Un étudiant suit plusieurs cours et les cours sont suivis par plusieurs étudiants.

Question 1

Réaliser le MCD en UML de ce problème.

Question 2

Proposer un exemple JSON équivalent à ce que l'on aurait fait en relationnel normalisé (sachant que ce type de solution ne sera généralement pas favorisé en BD orientée document).

Indice :

Représentation d'un cours par un objet JSON.

```
{
"code":"api04",
"titre":"DW et NOSQL"
...
}
```

Question 3

Proposer un exemple JSON basé sur l'imbrication.

Quel est le principal défaut de cette solution ?

Est-il toujours possible d'avoir une solution ne reposant que sur l'imbrication ?

Question 4

Proposer un exemple JSON basé sur les références.

Quelles sont les principales différences avec un système relationnel ?

Question 5

Sachant que l'objectif de l'application est de visualiser une liste des étudiants avec les cours que chacun suit, et d'accéder aux détails des cours uniquement lorsque l'on clique sur son code ou son titre.

Proposer une solution adaptée à ce problème mobilisant référence et imbrication.

VIII

Imbrication avec Json et Mongo (base de données orientée document)

A. Cours

1. Exemple de base de données orientée document avec MongoDB

a) Présentation de MongoDB

MongoBD est une base de données *open source* NoSQL orientée document. Elle stocke des données au format JSON (en fait BSON, qui est une version binaire de JSON).

Le serveur MongoDB est organisé en plusieurs *databases* :

- Chaque *database* contient des *collections*.
- Chaque *collection* contient des *documents*.
- Chaque *document* est au format JSON et contient donc des propriétés.

SQL	MongoDB
base de données et/ou schéma	base de données
table	collection
enregistrement	document
attribut (atomique)	propriété (chaîne, entier, tableau, structure)

Tableau 14 Comparaison SQL / MongoDB

Schema-less

C'est une base *schema-less*, aussi une collection peut contenir des documents de structures différentes et il n'est pas possible de définir la structure a priori d'une collection. La structure d'une collection n'est donc définie que par les document qui la compose, et elle peut évoluer dynamiquement au fur et à mesure des insertions et suppressions.

Identification clé / valeur

Chaque document est identifié par un identifiant nommé *_id* unique pour une collection, fonctionnant comme une **clé primaire artificielle**.

Architecture

MongoDB fonctionne a minima sous la forme d'un serveur auquel il est possible de se connecter avec un client textuel (*mongo shell*).

MongoDB peut être distribuée sur plusieurs serveurs (partitionnement horizontal ou *sharding*) et accédée à travers de multiples couches applicatives (langages, API...)

Complément

https://docs.mongodb.org/manual[38]

b) Installation et démarrage de MongoDB

MongoDB est disponible sur Windows, Mac OS X et Linux : *https://docs.mongodb.com/manual/installation*[39]

L'installation présentée ici est uniquement destinée à un contexte d'apprentissage, elle permet l'installation d'un serveur sur une machine Linux (ou Mac OS X) sans privilèges utilisateurs et l'exploitation de ce serveur avec un client textuel CLI situé sur la même machine.

Installation du serveur et du client

Installer *MongoDB Community Edition* sur son système.

Exemple sous Debian : `apt-get install mongodb-org` après avoir déclaré le dépôt MongoDB.

https://docs.mongodb.com/manual/tutorial/install-mongodb-on-debian/[40]

Démarrage d'un serveur Mongo

Créer un espace de stockage en lecture écriture pour la base de données (exemple : `mkdir ~/mongodata`)

Lancer le serveur MongoDB sur le port standard 27017 : `mongod --dbpath ~/mongodata`

Démarrage du client CLI Mongo (mongo shell)

Pour se connecter à un serveur MongoDB sur le port standard : `mongo --host nom-du-serveur` (par exemple : `mongo --host localhost`)

38 - https://docs.mongodb.org/manual
39 - https://docs.mongodb.com/manual/installation
40 - https://docs.mongodb.com/manual/tutorial/install-mongodb-on-debian/

Test

Une fois connecté exécuter le code suivant pour vérifier le bon fonctionnement de la base :

```
1  db.test.insert({ "test":"Hello World !" })
2  db.test.find({}, {_id:0})
```

Le résultat attendu est le suivant, la clé générée étant bien entendu différente :

```
{ "test" : "Hello World !" }
```

Complément

https://docs.mongodb.com/manual/mongo/[41]

c) Créer des bases de données et des collections

Fondamental : Créer une base et une collection

MongoDB est une base *schema-less*, la création des bases et des collections est dynamique lors d'une première **insertion** de données.

Méthode

Pour créer une base de données il faut exécuter une instruction `use` sur une nouvelle base de données, puis donner un ordre d'insertion d'un premier document JSON avec `insert`.

Exemple

```
use db1
db.col1.insert( { "x":1 } )
```

Syntaxe : Catalogue de données

- On peut voir la liste des bases de données avec :
  `show dbs`
- On peut voir la liste des collections de la base de données en cours avec :
  `show collections`
- On peut voir le contenu d'une collection avec :
  `db.col1.find()`

Complément

`https://docs.mongodb.com/manual/mongo`

d) Insertion des documents

L'insertion de données dans une base MongoDB se fait avec l'instruction `db.collection.insert(`*`Document JSON`*`)`.

Si la collection n'existe pas, elle est créée dynamiquement.

L'insertion associe un identifiant (*_id*) à chaque document de la collection.

41 - https://docs.mongodb.com/manual/mongo

Exemple

```
db.Cinema.insert(
{
   "nom":"Honkytonk Man",
   "realisateur":{
      "nom":"Eastwood",
      "prenom":"Clint"
   },
   "annee":1982,
   "acteurs":[
      {
         "nom":"Eastwood",
         "prenom":"Kyle"
      },
      {
         "nom":"Eastwood",
         "prenom":"Clint"
      }
   ]
}
)
```

Complément

https://docs.mongodb.org/manual/core/crud[42]
https://docs.mongodb.com/manual/tutorial/insert-documents[43]

e) Trouver des documents

La recherche de données dans une base MongoDB se fait avec l'instruction `db.collection.find(Document JSON, document JSON)`, avec :

- le premier document JSON définit une restriction ;
- le second document JSON définit une projection (ce second argument est optionnel).

Exemple : Restriction

```
1 db.Cinema.find({"nom":"Honkytonk Man"})
```

retourne les document JSON tels qu'ils ont à la racine un attribut "nom" avec la valeur "Honkytonk Man".

Exemple : Restriction et projection

```
1 db.Cinema.find({"nom":"Honkytonk Man"}, {"nom":1, "realisateur":1} )
```

retourne les document JSON tels qu'ils ont à la racine un attribut "nom" avec la valeur "Honkytonk Man", et seul les attributs situés à la racine "nom" et "realisateur" sont projetés (en plus de l'attribut "_id" qui est projeté par défaut).

42 - https://docs.mongodb.org/manual/core/crud/
43 - https://docs.mongodb.com/manual/tutorial/insert-documents/

Complément

https://docs.mongodb.org/manual/tutorial/query-documents/[44]
https://docs.mongodb.com/manual/reference/sql-comparison/[45]

2. Interroger Mongo en JavaScript

a) Interroger Mongo en JavaScript

Le console `mongo` permet d'exécuter des programme JavaScript avec instruction `load`.

```
1  //test.js
2  print("Hello world");
```

```
1  > load("test.js")
```

Parcours d'un résultat de requête Mongo

```
1  //query.js
2  conn = new Mongo();
3  db = conn.getDB("db1");
4
5  recordset = db.User.find({"liked":{$elemMatch:{"star":3}}}, {"_id":0,
   "liked.film":1})
6
7  while ( recordset.hasNext() ) {
8     printjson( recordset.next() );
9  }
```

44 - https://docs.mongodb.org/manual/tutorial/query-documents/
45 - https://docs.mongodb.com/manual/reference/sql-comparison/

Complément

https://docs.mongodb.com/manual/tutorial/write-scripts-for-the-mongo-shell/[46]

B. Exercice

1. Au ciné avec Mongo I

[1h30]

Soit les données suivantes représentant des films de cinéma.

```
db.Cinema.drop()

db.Cinema.insert(
{
nom:"Goodfellas",
annee:1990,
realisateur:{nom:"Scorsese", prenom:"Martin"},
acteurs:
 [
 {nom:"De Niro", prenom:"Robert"},
 {nom:"Liotta", prenom:"Ray"},
 {nom:"Pesci", prenom:"Joe"}
 ]
})

db.Cinema.insert(
{
nom:"The Godfather",
annee:1972,
realisateur:{nom:"Coppola", prenom:"Francis Ford"},
acteurs:
 [
 {nom:"Pacino", prenom:"Al"},
 {nom:"Brando", prenom:"Marlon"},
 {nom:"Duvall", prenom:"Robert"}
 ]
})

db.Cinema.insert(
{
nom:"Million Dollar Baby",
realisateur:{nom:"Eastwood", prenom:"Clint"},
acteurs:
 [
 {nom:"Swank", prenom:"Hilary"},
 {nom:"Eastwood", prenom:"Clint"}
 ]
})

db.Cinema.insert(
{
nom:"Gran Torino",
annee:2008,
realisateur:{nom:"Eastwood", prenom:"Clint"},
acteurs:
 [
 {nom:"Vang", prenom:"Bee"},
 {nom:"Eastwood", prenom:"Clint"}
```

46 - https://docs.mongodb.com/manual/tutorial/write-scripts-for-the-mongo-shell/

```
 ]
})

db.Cinema.insert(
{
nom:"Unforgiven",
realisateur:{nom:"Eastwood", prenom:"Clint"},
acteurs:
 [
 {nom:"Hackman", prenom:"Gene"},
 {nom:"Eastwood", prenom:"Clint"}
 ]
})

db.Cinema.insert(
{
nom:"Mystic River",
realisateur:{nom:"Eastwood", prenom:"Clint"},
acteurs:
 [
  {nom:"Penn", prenom:"Sean"},
  {nom:"Bacon", prenom:"Kevin"}
 ]
})

db.Cinema.insert(
{
nom:"Honkytonk Man",
realisateur:{nom:"Eastwood", prenom:"Clint"},
annee:1982,
acteurs:
 [
  {nom:"Eastwood", prenom:"Kyle"},
  {nom:"Bloom", prenom:"Verna"}
 ]
})

db.Cinema.find()
```

L'objectif est d'initialiser une base MongoDB avec ce script, puis d'écrire les requêtes MongoDB permettant de répondre aux questions suivantes.

Question 1

Créer une nouvelle base MongoDB et exécuter le script. Nommez votre base par votre nom de famille ou votre login sur la machine par exemple.

Indices :

Pour créer une base de données, utiliser l'instruction `use myNewDatabase`*, puis exécuter au moins une instruction d'insertion.*

Créer des bases de données et des collections

Question 2

Quels sont les films sortis en 1990 ?

Indices :

Trouver des documents

`db.Cinema.find(document JSON)`

La syntaxe JSON

`db.col.find({"attribute":"value"})`

Question 3

Quels sont les films sortis avant 2000 ?

Indices :

https://docs.mongodb.com/manual/tutorial/query-documents[47]

On utilisera l'objet `{$lt:value}` *à la place de la valeur de l'attribut à tester ($lt pour lesser than).*

Question 4

Quels sont les films réalisés par Clint Eastwood ?

Indice :

On utilisera un objet comme valeur.

Question 5

Quels sont les films réalisés par quelqu'un prénommé Clint ?

Indice :

Utiliser le navigateur de propriété des objets point : object.attribute.

Question 6

Quels sont les films réalisés par quelqu'un prénommé Clint avant 2000 ?

Indice :

Utiliser une liste de conditions attribut:valeur pour spécifier un AND (et logique) :

```
db.col.find({"attribute1":"value1", "attribute2":"value2"})
```

Question 7

Quels sont les films dans lesquels joue Clint Eastwood ?

Indice :

https://docs.mongodb.com/manual/tutorial/query-array-of-documents/[48]

Question 8

Quels sont les films dans lesquels joue un Eastwood ?

Question 9

Quels sont les noms des films dans lesquels joue un Eastwood ?

Indices :

Pour gérer la ***projection****, utiliser un second argument de la clause find() :*

```
db.Cinema.find({document JSON de sélection }, {document JSON de projection})
```

avec document JSON de projection de la forme : `{"attribut1":1, "attribut2":1...}`

Les identifiants sont toujours affichés par défaut, si on veut les supprimer, on peut ajouter la clause `_id:0` *dans le document de projection.*

47 - https://docs.mongodb.com/manual/tutorial/query-documents
48 - https://docs.mongodb.com/manual/tutorial/query-array-of-documents/

Question 10

Compléter le programme JavaScript suivant afin d'afficher les titre selon le format suivant :

```
- Million Dollar Baby
- Gran Torino
- Unforgiven
- Honkytonk Man
```

Indice :

```
conn = new Mongo();
db = conn.getDB("...");

recordset = ...

while ( recordset.hasNext() ) {
   film = recordset.next() ;
   print("- ", ...);
}
```

On veut à présent ajouter une nouvelle collection permettant de gérer des utilisateurs et leurs préférences. Pour chaque utilisateur on gérera un pseudonyme, et une liste de films préférés avec une note allant de une à trois étoiles.

Question 11

Ajouter trois utilisateurs préférant chacun un ou deux films.

On utilisera les identifiants des films pour les référencer.

Indices :

Insertion des documents

https://docs.mongodb.com/manual/reference/method/ObjectId[49]

2. Au ciné avec Mongo II

[30 min]

Soit les données suivantes représentant des films de cinéma et des avis d'utilisateurs concernant ces films.

```
db.Cinema.drop()

db.Cinema.insert(
{
nom:"Goodfellas",
annee:1990,
realisateur:{nom:"Scorsese", prenom:"Martin"},
})

db.Cinema.insert(
{
nom:"The Godfather",
annee:1972,
realisateur:{nom:"Coppola", prenom:"Francis Ford"},
})

db.Cinema.insert(
{
nom:"Million Dollar Baby",
```

49 - https://docs.mongodb.com/manual/reference/method/ObjectId/

```
realisateur:{nom:"Eastwood", prenom:"Clint"},
})

db.Cinema.insert(
{
nom:"Gran Torino",
annee:2008,
realisateur:{nom:"Eastwood", prenom:"Clint"},
})

db.Cinema.find()
```

```
db.User.drop()

db.User.insert(
{
 "pseudo":"Stph",
 "liked" :
   [
     {"film":ObjectId("590c366d70f50381c920ca71"),"star":3},
     {"film":ObjectId("590c366d70f50381c920ca72"),"star":1}
   ]
}
)

db.User.insert(
{
 "pseudo":"Luke",
 "liked" :
   [
     {"film":ObjectId("590c366d70f50381c920ca71"),"star":2}
   ]
}
)

db.User.insert(
{
 "pseudo":"Tuco",
 "liked" :
   [
     {"film":ObjectId("590c366d70f50381c920ca73"),"star":3}
   ]
}
)

db.User.find()
```

Question 1

Critiquer cette insertion mobilisant les identifiants des films ? Pourquoi n'est ce pas reproductible ? Imaginez deux solutions.

Question 2

Pourquoi trouver les titres de ces films n'est pas trivial avec Mongo (si, comme dans l'énoncé initial on a pas stocké le nom du film comme clé de référencement dans la collection User) ?

Question 3

On cherche à présent les identifiants des films qui sont aimés au moins une fois avec 3 étoiles.

On essaie cette requête : `db.User.find({"liked.star":3}, {_id:0, "liked.film":1})`, mais elle ne renvoie pas le résultat escompté.

Expliquez pourquoi ? Proposez des solutions.

Relationnel-JSON avec PostgreSQL

IX

A. Cours

1. Introduction à la manipulation JSON sous PostgreSQL

a) Type JSON sous PostgreSQL

PostgreSQL propose un nouveau type de données qui permet d'insérer du JSON dans une table.

Il s'agit d'une solution pour coupler de l'imbrication NF² avec une base relationnelle.

Rappel

La syntaxe JSON en bref

Syntaxe : Créer une table avec des attributs JSON

```
1  CREATE TABLE t (
2  ...
3  jsonatt JSON,
4  ...
5  );
6
```

Syntaxe : Insérer du JSON dans une table

```
1  INSERT INTO t
2  VALUES (
3  ...
4  '{ "key1": "value1", "key2": "value2"}',
5  ...
6  );
```

Exemple

```
CREATE TABLE cours (
titre TEXT PRIMARY KEY,
auteur JSON NOT NULL,
chapitres JSON NOT NULL,
nbpages INTEGER
);
```

```
INSERT INTO cours (titre, auteur, chapitres, nbpages)
VALUES (
'Bases de données',
'{"nom":"Crozat","prenom":"Stéphane"}',
'["UML","R","SQL","Normalisation","RO"]',
98
);
INSERT INTO cours (titre, auteur, chapitres, nbpages)
VALUES (
'Ingénierie documentaire',
'{"nom":"Crozat","prenom":"Stéphane"}',
'["XML","JSON"]',
45
);
```

```
          titre          |                auteur                |
chapitres                | nbpages
-------------------------+--------------------------------------
+---------------------------------------+---------
 Bases de données        | {"nom":"Crozat","prenom":"Stéphane"} |
["UML","R","SQL","Normalisation","RO"] |      98
 Ingénierie documentaire | {"nom":"Crozat","prenom":"Stéphane"} |
["XML","JSON"]                         |      45
```

Fondamental

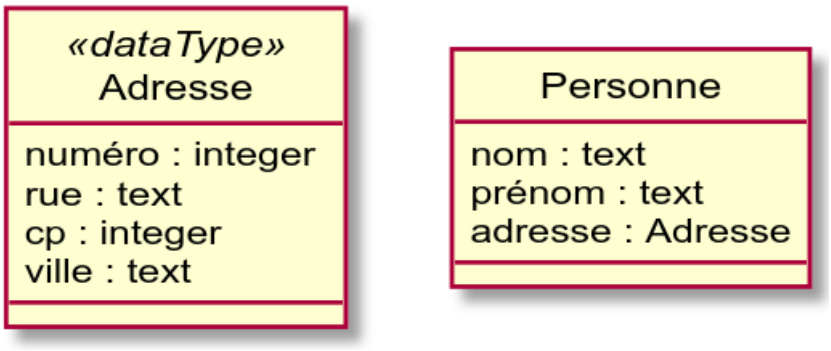
Pour manipuler les données JSON intégrées dans une base PosgreSQL, on dispose d'opérateur complémentaires permettant de projeter les représentations JSON en relationnel.

Par exemple :

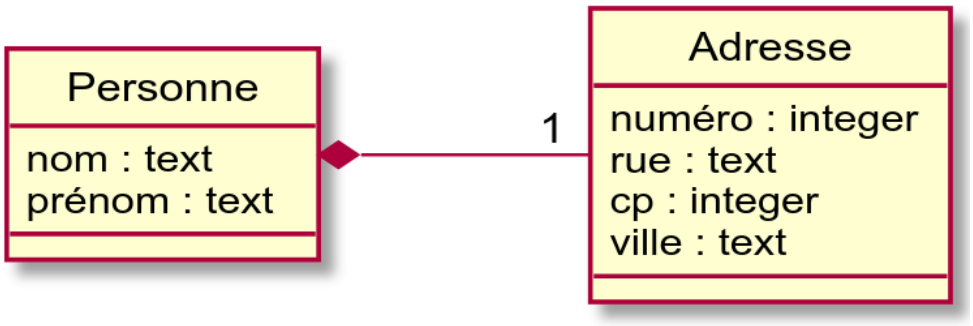
- ->> permet de projeter une valeur JSON
- -> permet de parcourir un arbre JSON
- `JSON_ARRAY_ELEMENTS` permet de convertir un tableau de valeurs JSON en relationnel
- `JSON_TO_RECORDSET` permet de convertir un tableau d'objets JSON en relationnel

b) Attributs composés (datatype) en R-JSON (->>)

Exemple


«dataType»
Adresse
numéro : integer
rue : text
cp : integer
ville : text
Personne
nom : text
prénom : text
adresse : Adresse

Rappel : Représentation équivalente



Une représentation avec datatype est équivalente à une composition 1:1.

Exemple

```
1  CREATE TABLE Personne (
2  sk TEXT PRIMARY KEY,
3  nom TEXT NOT NULL,
4  prénom TEXT NOT NULL,
5  adresse JSON NOT NULL
6  );
```

```
1  INSERT INTO Personne
2  VALUES (
3  1,
4  'Holmes',
5  'Sherlock',
6  '{"numéro":221,"rue":"rue du boulanger", "cp":60200,
   "ville":"Compiègne"}'
7  );
```

```
1   sk |  nom   |  prénom  |                                        adresse
2  ----+--------+----------
   +-------------------------------------------------------------------------
   ------
3   1  | Holmes | Sherlock | {"numéro":221,"rue":"rue du boulanger",
   "cp":60200, "ville":"Compiègne"}
```

Syntaxe : Obtenir une valeur d'un objet JSON : opérateur ->>('key')

```
1  SELECT jsonatt->>'key' FROM t
```

Renvoie la valeur correspondant à la clé `key` située à la racine du JSON.

Exemple

```
1  SELECT nom, prénom, adresse->>'ville' AS ville
2  FROM Personne;
```

```
1    nom   |  prénom  |   ville
2  --------+----------+-----------
3   Holmes | Sherlock | Compiègne
```

Attention : Forcer les types (CAST)

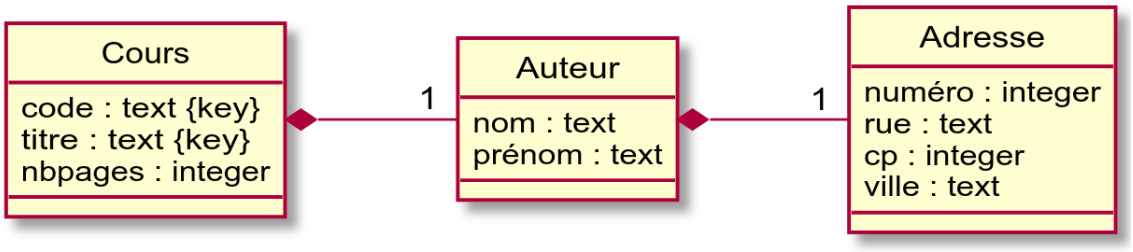
Les attributs sont retournés comme des chaînes, pour obtenir d'autre types, il faut utiliser la fonction CAST.

```
1  SELECT nom, prénom, CAST(adresse->>'numéro' AS INTEGER) AS numéro,
   adresse->>'rue' AS rue
2  FROM Personne;
```

```
1    nom    |  prénom   | numéro |        rue
2  ---------+-----------+--------+-------------------
3   Holmes | Sherlock |    221 | rue du boulanger
```

c) Arborescences en R-JSON (->)

Exemple



Exemple

```
1  CREATE TABLE cours (
2  code TEXT PRIMARY KEY,
3  titre TEXT UNIQUE NOT NULL,
4  nbpages INTEGER NOT NULL,
5  auteur JSON NOT NULL
6  );
```

```
1  INSERT INTO cours (code, titre, nbpages, auteur)
2  VALUES (
3  'NF17',
4  'Bases de données',
5  98,
6  '{"nom":"Crozat","prenom":"Stéphane","adresse":{"num":18,"rue":"rue
   St Fiacre","cp":60200,"ville":"Compiègne"}}'
7  );
8  INSERT INTO cours (code, titre, nbpages, auteur)
9  VALUES (
10 'NF29',
11 'Ingénierie documentaire',
12 45,
13 '{"nom":"Crozat","prenom":"Stéphane","adresse":{"num":18,"rue":"rue
   St Fiacre","cp":60200,"ville":"Compiègne"}}'
14 );
```

```
1  INSERT INTO cours (code, titre, nbpages, auteur)
2  VALUES (
3  'NF29',
4  'Ingénierie documentaire',
5  45,
6  '{"nom":"Crozat","prenom":"Stéphane","adresse":{"num":4,"rue":"rue St
```

```
    Germain","cp":60300,"ville":"Senlis"}}'
);
```

Syntaxe : Parcourir un JSON (opérateur ->)

```
SELECT jsonatt->'key1'->>'key2' FROM t;
```

Renvoie la valeur associée à `key2` qui se trouve dans `key1`.

Exemple

```
SELECT
titre,
auteur->>'nom' AS nom_auteur,
auteur->>'prenom' AS prenom_auteur,
auteur->'adresse'->>'ville' AS ville_auteur
FROM cours;
```

```
          titre          | nom_auteur | prenom_auteur | ville_auteur
-------------------------+------------+---------------+--------------
 Bases de données        | Crozat     | Stéphane      | Compiègne
 Ingénierie documentaire | Crozat     | Stéphane      | Senlis
```

Attention

- -> Est un opérateur de parcours intermédiaire, qui renvoie du JSON
- ->> Est un opérateur final qui renvoie une valeur.

Complément : Opérateurs

Operator	Right Operand Type	Description	Example	Example Result
->	int	Get JSON array element (indexed from zero, negative integers count from the end)	'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2	{"c":"baz"}
->	text	Get JSON object field by key	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	Get JSON array element as text	'[1,2,3]'::json->>2	3
->>	text	Get JSON object field as text	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	Get JSON object at specified path	'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	Get JSON object at specified path as text	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'	3

json Operators

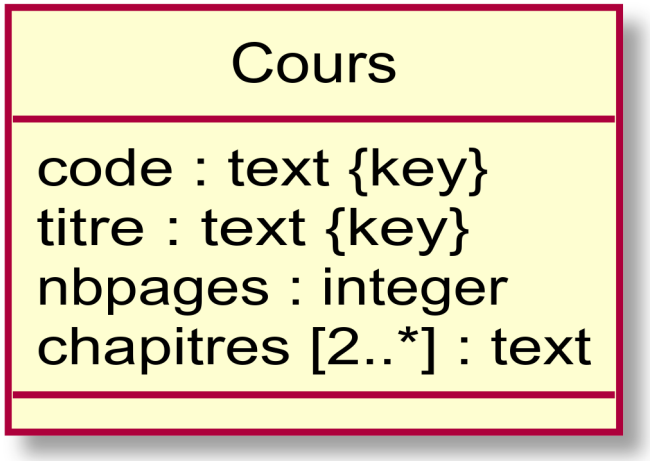
Complément

https://www.postgresql.org/docs/current/functions-json.html[50]

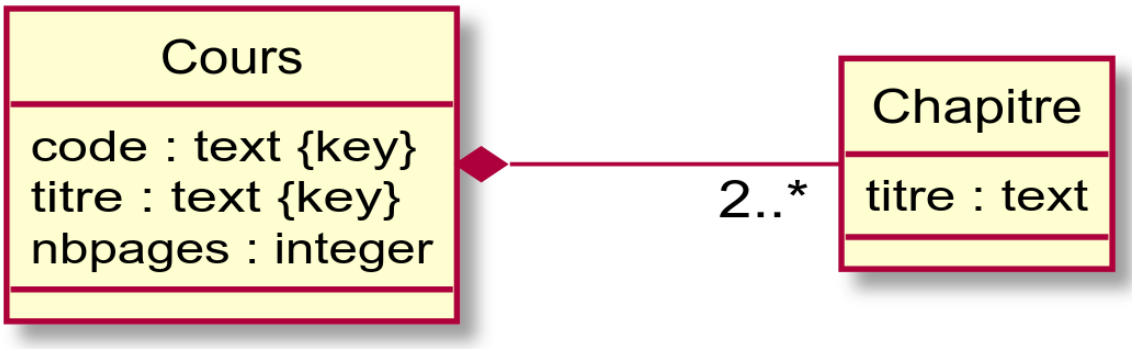
d) Attributs multivalués en R-JSON (JSON_ARRAY_ELEMENTS)

50 - https://www.postgresql.org/docs/current/functions-json.html

Exemple



Rappel : Représentation équivalente



Une représentation avec attribut multivalué est équivalente à une composition dont le composite ne comporte qu'un seul attribut.

Exemple

```
1  CREATE TABLE cours (
2  code TEXT PRIMARY KEY,
3  titre TEXT UNIQUE NOT NULL,
4  nbpages INTEGER NOT NULL,
5  chapitres JSON NOT NULL
6  );
```

```
1  INSERT INTO cours (code, titre, nbpages, chapitres)
2  VALUES (
3  'NF17',
4  'Bases de données',
5  98,
6  '["UML","R","SQL","Normalisation","RO"]'
7  );
8  INSERT INTO cours (code, titre, nbpages, chapitres)
9  VALUES (
10 'NF29',
11 'Ingénierie documentaire',
12 45,
```

```
'["XML","JSON"]'
);
```

```
 code |          titre          | nbpages |                chapitres
------+-------------------------+---------
     +-------------------------------------
 NF17 | Bases de données        |      98 |
["UML","R","SQL","Normalisation","RO"]
 NF29 | Ingénierie documentaire |      45 | ["XML","JSON"]
```

Syntaxe : Convertir un tableau de scalaires JSON en relationnel : opérateur JSON_ARRAY_ELEMENTS

```
SELECT a.* FROM t, JSON_ARRAY_ELEMENTS(t.jsonatt) a
```

Exemple

```
SELECT co.titre, ch.*
FROM cours co, JSON_ARRAY_ELEMENTS(co.chapitres) ch;
```

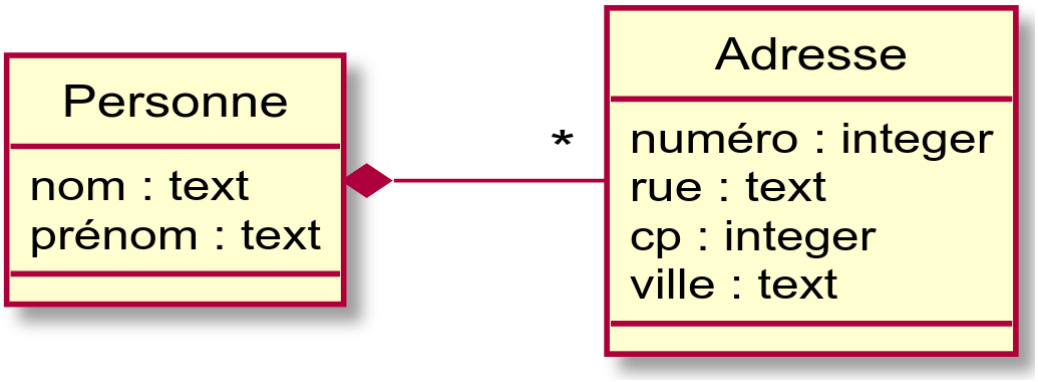
```
          titre          |      value
-------------------------+-----------------
 Bases de données        | "UML"
 Bases de données        | "R"
 Bases de données        | "SQL"
 Bases de données        | "Normalisation"
 Bases de données        | "RO"
 Ingénierie documentaire | "XML"
 Ingénierie documentaire | "JSON"
```

Complément

Complément : Attributs multivalués : opérateurs complémentaires

e) Compositions en R-JSON (JSON_ARRAY_ELEMENTS et ->>)

Exemple



Exemple

```
CREATE TABLE Personne (
sk TEXT PRIMARY KEY,
nom TEXT NOT NULL,
prénom TEXT NOT NULL,
adresse JSON
);
```

```
INSERT INTO Personne
VALUES (
1,
'Holmes',
'Sherlock',
'[
{"numéro":221,"rue":"rue du boulanger", "cp":60200,
"ville":"Compiègne"},
{"numéro":0,"rue":"rue Secrète", "cp":60200, "ville":"Compiègne"}
]'
);

INSERT INTO Personne
VALUES (
2,
'Watson',
'John',
'[
{"numéro":221,"rue":"rue du boulanger", "cp":60200,
"ville":"Compiègne"}
]'
);
```

```
 sk |  nom    |  prénom  |                                        adresse
----+--------+----------
+-------------------------------------------------------------------------
-------
 1  | Holmes | Sherlock | [
+
    |        |          | {"numéro":221,"rue":"rue du boulanger",
"cp":60200, "ville":"Compiègne"},+
    |        |          | {"numéro":0,"rue":"rue Secrète",
"cp":60200, "ville":"Compiègne"}        +
    |        |          | ]
 2  | Watson | John     | [
+
    |        |          | {"numéro":221,"rue":"rue du boulanger",
"cp":60200, "ville":"Compiègne"} +
    |        |          | ]
```

Syntaxe : Convertir un tableau d'objets JSON en relationnel : opérateurs JSON_ARRAY_ELEMENTS et ->>

```
SELECT a->>a1, a->>a2... FROM t, JSON_ARRAY_ELEMENTS(t.jsonatt) a
```

Exemple

```
SELECT p.nom, p.prénom, CAST(ad->>'numéro' AS INTEGER) AS numéro, ad-
>>'rue' AS rue, ad->>'ville' AS ville
FROM personne p, JSON_ARRAY_ELEMENTS(p.adresse) ad
```

```
  nom   |  prénom  | numéro |       rue        |   ville
```

```
--------+----------+--------+-----------------+-----------
 Holmes | Sherlock |    221 | rue du boulanger | Compiègne
 Holmes | Sherlock |      0 | rue Secrète      | Compiègne
 Watson | John     |    221 | rue du boulanger | Compiègne
```

Rappel

Il est nécessaire d'utiliser la fonction CAST pour obtenir un type autre que chaîne de caractère.

Attributs composés (datatype) en R-JSON (->>)

Complément

Complément : Compositions avec JSON_TO_RECORDSET

2. Éléments complémentaires pour la manipulation des compositions

a) Optionnalité avec JSON_ARRAY_ELEMENTS

Exemple : LEFT JOIN JSON_ARRAY_ELEMENTS

```
INSERT INTO Personne
VALUES (
1,
'Holmes',
'Sherlock',
'[
{"numéro":221,"rue":"rue du boulanger", "cp":60200,
"ville":"Compiègne"},
{"numéro":0,"rue":"rue Secrète", "cp":60200, "ville":"Compiègne"}
]'
);

INSERT INTO Personne
VALUES (
2,
'Watson',
'John'
);
```

```
 sk |  nom   |  prénom  |                                        adresse
----+--------+----------
+-----------------------------------------------------------------------
-------
 1  | Holmes | Sherlock | [
+
    |        |          | {"numéro":221,"rue":"rue du boulanger",
"cp":60200, "ville":"Compiègne"},+
    |        |          | {"numéro":0,"rue":"rue Secrète",
"cp":60200, "ville":"Compiègne"}        +
    |        |          | ]
 2  | Watson | John     |

```

Remarque

```
SELECT a->>a1, a->>a2... FROM t, JSON_ARRAY_ELEMENTS(t.jsonatt) a
```

équivalent à

```
SELECT a->>a1, a->>a2... FROM t JOIN JSON_ARRAY_ELEMENTS(t.jsonatt) a
ON TRUE
```

Attention : L'opérateur JSON_ARRAY_ELEMENTS agit comme une jointure

```
SELECT p.nom, p.prénom, CAST(ad->>'numéro' AS INTEGER) AS numéro, ad-
>>'rue' AS rue, ad->>'ville' AS ville
FROM personne p JOIN JSON_ARRAY_ELEMENTS(p.adresse) ad ON TRUE
```

```
  nom    |  prénom   | numéro |       rue        |   ville
---------+-----------+--------+------------------+-----------
 Holmes  | Sherlock  |    221 | rue du boulanger | Compiègne
 Holmes  | Sherlock  |      0 | rue Secrète      | Compiègne
```

Syntaxe : Convertir un tableau optionnel d'objets JSON en relationnel : opérateurs LEFT JOIN JSON_ARRAY_ELEMENTS et ->>

```
SELECT a->>a1, a->>a2... FROM t LEFT JOIN
JSON_ARRAY_ELEMENTS(jsonatt) a on TRUE
```

Exemple : L'opérateur JSON_ARRAY_ELEMENTS agit comme une jointure

```
SELECT p.nom, p.prénom, CAST(ad->>'numéro' AS INTEGER) AS numéro, ad-
>>'rue' AS rue, ad->>'ville' AS ville
FROM personne p LEFT JOIN JSON_ARRAY_ELEMENTS(p.adresse) ad ON TRUE
```

```
  nom    |  prénom   | numéro |       rue        |   ville
---------+-----------+--------+------------------+-----------
 Holmes  | Sherlock  |    221 | rue du boulanger | Compiègne
 Holmes  | Sherlock  |      0 | rue Secrète      | Compiègne
 Watson  | John      |        |                  |
```

b) Complément : Compositions avec JSON_TO_RECORDSET

Complément : Convertir un tableau d'objets JSON en relationnel : opérateur JSON_TO_RECORDSET (à partir de Postgres 11)

```
SELECT a.* FROM JSON_TO_RECORDSET(jsonatt) a (a1 type1, a2, type2...)
```

Les attributs JSON à projeter en attribut relationnel doivent être explicitement mentionnés et typés.

```
SELECT p.nom, p.prénom, ad.*
FROM personne p, JSON_TO_RECORDSET(adresse) ad (numéro INTEGER, rue
TEXT, ville TEXT);
```

```
  nom    |  prénom   | numéro |       rue        |   ville
---------+-----------+--------+------------------+-----------
 Holmes  | Sherlock  |    221 | rue du boulanger | Compiègne
 Holmes  | Sherlock  |      0 | rue Secrète      | Compiègne
 Watson  | John      |    221 | rue du boulanger | Compiègne
```

Complément

https://www.postgresql.org/docs/current/functions-json.html[51]

c) Complément : Opérateurs complémentaires pour les tableaux JSON

Complément : Obtenir une valeur d'un tableau JSON : opérateur ->>(integer)

```
1  SELECT jsonatt->>N FROM t
```

Renvoie le Nième élément du tableau situé à la racine du JSON (en comptant à partir de 0).

```
1  SELECT
2  titre,
3  chapitres->>0 AS chapitre1
4  FROM cours;
```

```
1            titre           | chapitre1
2  --------------------------+-----------
3   Bases de données         | UML
4   Ingénierie documentaire  | XML
```

Complément : Combinaison arborescence et tableau

```
1  SELECT jsonatt->'key1'->>0 FROM t;
```

Renvoie la première valeur du tableau associé dans `key1`.

```
1  INSERT INTO cours (code, titre, nbpages, chapitres)
2  VALUES (
3  'NF26',
4  'Datawarehouses',
5  67,
6  '{"sections":["Principes","Modélisation","ETL"]}'
7  );
8
9  SELECT
10 titre,
11 chapitres->'sections'->>0 AS chapitre1
12 FROM cours;
```

```
1            titre           | chapitre1
2  --------------------------+-----------
3   Bases de données         |
4   Ingénierie documentaire  |
5   Datawarehouses           | Principes
```

Complément

https://www.postgresql.org/docs/current/functions-json.html[52]

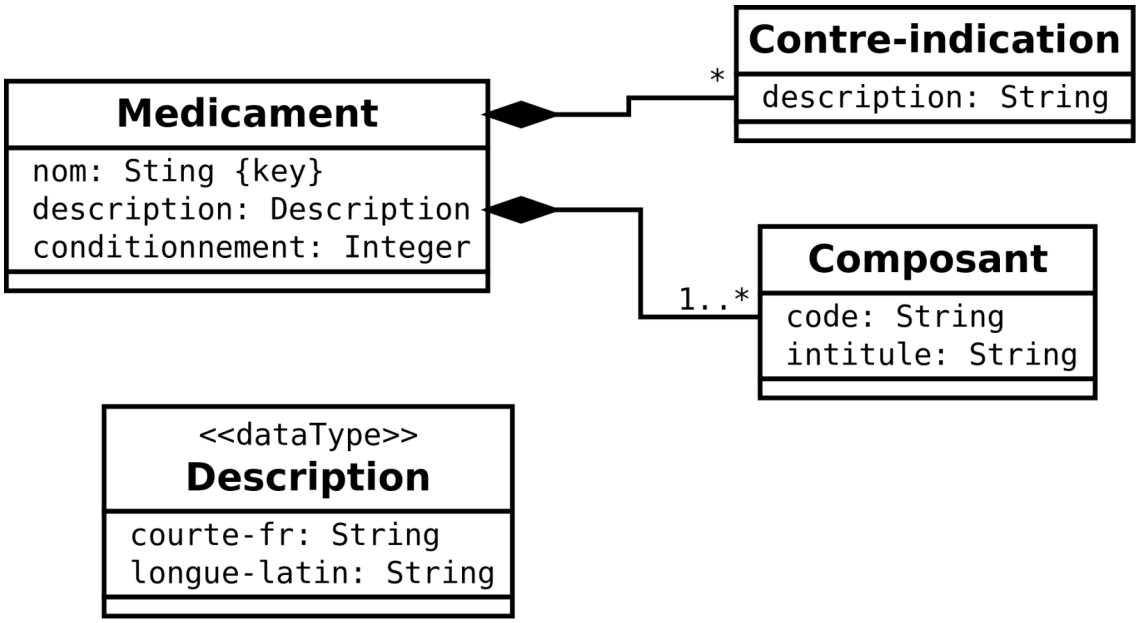
51 - https://www.postgresql.org/docs/current/functions-json.html
52 - https://www.postgresql.org/docs/current/functions-json.html

B. Exercice

1. Lab VIII

30 min

Soit le modèle UML ci-après.




```
1  Le Chourix a pour description courte "Médicament contre la chute des
   choux" et pour description longue "Vivamus fermentum semper porta.
   Nunc diam velit, adipiscing ut tristique vitae, sagittis vel odio.
   Maecenas convallis ullamcorper ultricies. Curabitur ornare.". Il est
   conditionné en boîte de 13.
2  Ses contre-indications sont :
3  - Le Chourix ne doit jamais être pris après minuit.
4  - Le Chourix ne doit jamais être mis au contact avec de l'eau.
5  Ses composants sont le HG79 ("Vif-argent allégé") et le SN50 ("Pur
   étain").
6
7  Le Tropas a pour description courte "Médicament contre les
   dysfonctionnements intellectuels" et pour description longue
   "Suspendisse lectus leo, consectetur in tempor sit amet, placerat
   quis neque. Etiam luctus porttitor lorem, sed suscipit est rutrum
   non.". Il est conditionné en boîte de 42.
8  Ses contre-indications sont :
9  - Le Tropas doit être gardé à l'abri de la lumière du soleil.
10 Son unique composant est le HG79 ("Vif-argent allégé").
```

Question 1

Réaliser l'implémentation R-JSON de ce modèle UML et insérer des données d'exemple.

Question 2

Proposez 3 vues permettant de visualkiser :

- La liste des noms des médicaments avec leurs contre-indications

- La liste des des noms des médicaments avec leurs composants
- La liste des noms des médicaments avec leur description courte, leur conditionnement, le nombre de leurs contre-indications et le nombre de leurs composants

Index