

Institut Paul Lambin

Haute Ecole Léonard de Vinci

Clos Chapelle-aux-Champs 43 • 1200 Bruxelles

Tél 02 764 46 46 • Fax 02 771 40 35 • www.ipl.be • lambin@ipl.be



2^{ème} année Baccalauréat en Informatique

Cours de Bases de Données (SQL)

Année académique 2019-2020



Sommaires

1.	Introduction.....	1
1.a.	SQL de nos jours	1
1.b.	RDBMS (Relational Database Management System)	2
1.b.i.	Le modèle relationnel.....	2
1.b.ii.	L'algèbre relationnelle.....	2
1.b.iii.	L'analyse relationnelle.....	7
1.b.iv.	Exemple et exercices.	8
2.	Requêtes (SELECT).....	9
2.a.	SELECT ... FROM ... WHERE ... ORDER BY	9
2.a.i.	Exemples.....	10
2.a.ii.	Exercices	12
2.b.	Produit cartésien et jointure	13
2.b.i.	Produit cartésien	13
2.b.ii.	Jointure.....	14
2.b.iii.	Exemples.....	15
2.b.iv.	Exercices	17
2.c.	Fonctions d'aggrégation	18
2.c.i.	Exemples.....	18
2.d.	GROUP BY ... HAVING	20
2.d.i.	Exemples.....	20
2.d.ii.	Exercices	23
2.e.	SOUS-SELECT	24
2.e.i.	Exemples avec IN.....	24
2.e.ii.	Exemples avec ("=" "<>" "<" ">" "<=" ">=") ["ALL" "ANY"]	25
2.e.iii.	Exemples avec EXISTS.....	27
2.e.iv.	Exercices	30
2.f.	UNION, INTERSECTION, DIFFERENCE	31
2.f.i.	Exemples.....	31
2.g.	OUTER JOIN (Jointures externes)	32
2.g.i.	Exemples.....	32
2.g.ii.	Exercices	35
2.h.	COALESCE	36

2.h.i.	Exemple	36
2.h.ii.	Exercice.....	36
3.	Stockage de données.....	37
3.a.	Ajout/Suppression/Modification de tables.....	37
3.a.i.	Contraintes d'intégrité	37
3.a.ii.	CREATE TABLE.....	38
3.a.iii.	DROP TABLE.....	39
3.a.iv.	ALTER TABLE	39
3.b.	Normalisation	41
3.c.	Dénormalisation	42
3.c.i.	Exemple	42
3.c.ii.	Exercice.....	42
3.d.	Ajout/Suppression/Modification de tuples.....	43
3.d.i.	INSERT INTO	43
3.d.ii.	UPDATE ... SET ... WHERE	43
3.d.iii.	DELETE FROM	43
3.e.	Séquences.....	46
3.e.i.	Exemple	46
3.e.i.	Exercice.....	46
3.f.	Serial.....	47
4.	Gestion d'une base de données	48
4.a.	SQL Procédural	48
4.a.i.	CREATE FUNCTION	49
4.a.ii.	DECLARE	49
4.a.iii.	Affectation.....	49
4.a.iv.	Commentaire.....	49
4.a.v.	Structures de contrôle.....	49
4.a.vi.	Exceptions.....	50
4.a.vii.	Exemple	51
4.a.viii.	Procédure stockée renvoyant un tableau	51
4.b.	Automatisation : TRIGGER.....	53
4.b.i.	Cas typique d'utilisation	53
4.b.ii.	Référence complète de CREATE TRIGGER.....	54
4.b.iii.	Procédure trigger en PL/pgSQL	54

4.c.	Utilisateurs	57
4.c.i.	CREATE USER	57
4.d.	GRANT / REVOKE	58
4.d.i.	Exemple	58
4.d.ii.	Cas particulier des SERIAL	58
4.e.	Simplification des requêtes : VIEW	59
4.e.i.	Exemples.....	59
4.f.	Performance : INDEX	60
4.f.i.	Exemple	61
5.	Concurrence	62
5.a.	ACID	63
5.b.	Transaction en SQL.....	64
5.b.i.	Transactions en PL/pgSQL	64
5.b.ii.	Exemple	65
5.b.iii.	Transactions et trigger.....	65
5.b.iv.	Exercice.....	65
5.c.	MVCC.....	66
6.	SQL embarqué	69
6.a.	JDBC.....	69
6.b.	Driver	69
6.c.	Connection	69
6.d.	Statement.....	70
6.e.	ResultSet.....	70
6.f.	PreparedStatement	71
6.g.	Intégration entre Java et une base de données.....	73
6.h.	Bonne pratique JDBC en général.....	74
6.i.	Bonne pratique JDBC lorsque l'on minimise l'usage du SQL.....	75
6.j.	Bonne pratique JDBC lorsque l'on maximise l'usage du SQL	75
6.k.	Exemple	75
6.l.	Exercice.....	78
7.	Annexes	79
7.a.	Exercices récapitulatifs sur les requêtes	79
7.b.	Notation BNF	83
7.c.	Référence rapide PostgreSQL.....	85

7.c.i.	Liste des commandes SQL	85
7.c.ii.	Types.....	89
7.c.iii.	Fonctions d'agrégat	90
7.c.iv.	Fonctions pour le type chaîne	91
7.c.v.	Opérateurs et Fonctions pour le type date	92
7.c.vi.	Fonctions de transtypage	94
7.d.	Codes d'erreur de PostgreSQL.....	95
7.e.	Diagramme de Bachman de la base de donnée pubs2	98
8.	Références.....	99

1. Introduction

En informatique, il est très fréquent de rencontrer encore et encore le même problème. Avec l'expérience, on peut évaluer les différentes solutions qui ont été apportées et choisir celle qui tient le mieux la route. Ce choix est d'ailleurs souvent très pragmatique : c'est la solution qui survit à l'épreuve du temps qui finit par être universellement acceptée.

Ainsi dans le courant des années 1960, il existait un besoin grandissant d'automatiser la gestion des données. Non seulement il fallait pouvoir stocker ces données, mais en plus il fallait trouver un mécanisme permettant de les exploiter : récupérer les données, les faire évoluer d'une manière consistante, faire des croisements entre plusieurs informations, consolider des résultats, etc. Il y avait donc un besoin d'exprimer toutes sortes de requêtes et manipulations sur les données. Mais développer des algorithmes spécifiques à chaque besoin spécifique aurait été coûteux et contreproductif. C'est pourquoi Edgar F. Codd publia en 1970 un article intitulé « A Relational Model of Data for Large Shared Data Banks » proposant une manière de les représenter et de les manipuler. Cette approche donna naissance au langage SQL : Structured Query Language.

1.a. SQL de nos jours

SQL reste de nos jours incontestablement le standard pour gérer la persistance des données. SQL est tellement omniprésent sur Internet que probablement tous les sites web que vous fréquentez l'utilisent. Il commence aussi à avoir une grande présence sur les Smartphones pour toutes les applications ayant des données non triviales à gérer. Vous retrouverez SQL au cœur de la gestion des données de presque tous les systèmes informatiques : applications bancaires, médicales, cartographie, multimédia, etc.....

Le marché est composé de poids lourds commerciaux (Oracle, Microsoft, Sybase, ...), mais aussi de produits issus du monde du logiciel libre (MySQL, PostgreSQL, SQLite, ...). SQL est un langage qui est normalisé mais reste très complexe. Ainsi chaque vendeur n'implémente qu'un sous ensemble plus ou moins complet du langage. De plus la concurrence étant bien présente, chaque vendeur essaie de se sortir du lot par des extensions qui lui sont propres. Au final, il y a un tronc SQL commun à toutes ces bases de données (BD), mais on se retrouve assez rapidement à utiliser des instructions qui sont spécifiques à un vendeur particulier.

Dans le cadre de ce cours, nous utiliserons PostgreSQL. C'est un logiciel libre et donc gratuit. Il attache une grande importance au respect de la norme SQL. C'est un produit moderne, avec une communauté de développeur très active. Il est fréquemment utilisé en milieu professionnel et il est fort probable que vous le rencontrerez à un moment ou un autre de votre future carrière. Pour toutes ces raisons nous avons choisi ce logiciel.

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

1.b. RDBMS (Relational Database Management System)

1.b.i. Le modèle relationnel

Une base de données relationnelle est constituée d'un ensemble de «relations», souvent appelées « tables ».

Chaque table est divisée d'une part en colonnes et d'autre part en lignes.

Les colonnes sont les « **attributs** ». Elles sont désignées par leur numéro d'ordre (1,2,3,...) ou par des noms d'attributs; dans ce dernier cas, l'ordre des colonnes est sans importance. Chaque attribut peut se voir imposer un « domaine », c'est-à-dire l'ensemble de toutes les valeurs possibles de cet attribut.

Les lignes sont appelées « **tuples** »: elles correspondent plus ou moins aux « enregistrements » des fichiers classiques, à ceci près qu'une table n'est pas considérée comme une *séquence* de tuples, mais bien comme un *ensemble* (au sens mathématique) de tuples. Ceci entraîne deux conséquences :

- l'ordre des tuples est sans importance (càd : il n'y a pas d'ordre);
- deux tuples sont toujours distincts l'un de l'autre.

La définition en « intention » d'une base de données est constituée de la liste de ses tables, avec, pour chacune d'elles, la liste des attributs et de leurs domaines, ainsi que d'une série de « contraintes d'intégrité », servant à garantir la cohérence des données : cette définition, qui n'inclut pas le contenu actuel des tables, est la partie permanente, fixe de la D.B.

La définition en « extension » d'une base de données est constituée du contenu actuel de ses tables, c'est-à-dire des ensembles de tuples qu'elles contiennent.

1.b.ii. L'algèbre relationnelle.

La première façon de manipuler de telles D.B. est d'utiliser un langage algébrique basé sur plusieurs opérateurs. Ci-dessous, sont proposés 5 opérateurs de base, suivi de 4 autres (qui ne sont pas indispensables car ils peuvent s'écrire au moyen des 5 premiers, mais augmentent le confort de manipulation).

Opérateurs de base.

1. Union.

L'union de deux tables R et S, que l'on notera $R \cup S$, est l'union au sens ensembliste des tuples des deux tables. Pour que le résultat soit encore une table, il faut que R et S comportent le même nombre de colonnes, et que si les colonnes ont des noms d'attributs, ces noms soient identiques, et que les colonnes qui se correspondent dans les deux tables aient des domaines identiques (ou au moins compatibles).

2. Différence.

Il s'agit ici de la différence au sens ensembliste de tuples de deux tables. Les mêmes restrictions s'appliquent que pour l'union. On notera $R - S$ ou $R \setminus S$.

3. Produit cartésien.

Si R est une table comportant r colonnes et S une table comportant s colonnes, leur produit cartésien $R \times S$ est une table comportant r+s colonnes héritant leurs noms d'attributs de ceux de R et S, et contenant pour tuples tous les tuples que l'on peut obtenir en **juxtaposant un tuple de R et un tuple de S**. Si R contient n tuples et que S en contient p, $R \times S$ en contiendra n x p. Si des attributs de R et S ont même noms, on les distinguera en mentionnant la table d'origine : R.i, S.i.

4. Projection.

Si i,j, ...,k sont des numéros de colonnes ou des noms d'attributs d'une table R, l'opérateur unaire de projection sur i,j,...,k est défini comme suit : la table $\pi_{i,j,...,k}(R)$ est obtenue à partir de R en ne conservant que les colonnes i,j,...,k, et bien sûr en supprimant du résultats les éventuels tuples qui seraient identiques à d'autres.

5. Sélection.

Soit R une table, et F une formule logique constituée

- de numéros ou noms d'attributs
- de constantes
- d'opérateurs arithmétiques ou autres s'appliquant aux constantes et aux domaines des attributs
- d'opérateurs de relation (=, <, >, etc.)
- d'opérateurs logiques : NOT, AND, OR.

Alors l'opération de sélection est définie comme suit : la table $\sigma_F(R)$ est la table comportant les mêmes attributs que R, et contenant les tuples de R qui satisfont à la condition F (c'est-à-dire qui donnent la valeur TRUE à F si on y remplace les *noms* d'attributs par les *valeurs* prises par ces attributs dans le tuple concerné).

Opérateurs additionnels.

6. Intersection.

L'intersection $R \cap S$ est définie de façon similaire à l'union. Rappelons que : $R \cap S = R - (R - S)$.

7. Quotient.

Soient R et S deux tables contenant respectivement r et s colonnes, avec $s < r$. Supposons en outre que les noms d'attributs de S forment un sous-ensemble de ceux de R et que S ne soit pas vide. On définira le quotient $R : S$ comme étant la table comportant r-s attributs (ayant pour noms ceux de la table R qui sont absents de la table S), et contenant comme tuples t ceux pour lesquels, pour tout tuple u de S, la combinaison « tu » soit dans R.

On peut définir cet opérateur à partir des 5 premiers de la façon suivante : si i,j,...,k sont les attributs présents dans R mais absents de S, on a :

$$R : S = \pi_{i,j,...,k}(R) - \pi_{i,j,...,k}((\pi_{i,j,...,k}(R) \times S) - R).$$

8. Jointure.

Si i est un attribut de R et j un attribut de S , la jointure $R \text{ JOIN}_{i=j} S$ est la table obtenue en prenant le produit cartésien de R et S , dont on ne conserve que les tuples pour lesquels les valeurs de l'attribut $R.i$ et de l'attribut $S.j$ sont égales. On peut de même définir des jointures basées sur d'autres opérateurs de relation (par exemple: $R \text{ JOIN}_{i<j} S$) ou sur des conditions combinées (par exemple: $R \text{ JOIN}_{(i=j \text{ and } k \neq l)} S$).

Le cas particulier où l'opérateur d'égalité « = » est le seul présent est appelé « équijointure ».

9. Jointure naturelle.

La jointure naturelle de deux tables R et S comportant un ou plusieurs attributs identiques (mêmes noms et mêmes domaines) est la table $R \text{ JOIN } S$ obtenue en effectuant tout d'abord une équijointure selon les attributs communs, puis une projection ayant pour effet de supprimer du résultat les attributs provenant de S qui sont des duplicatas d'attributs provenant de R .

Exemples.

R

Nom	Destination	Code-dépl
Dufour	Paris	321
Dufour	Milan	325
Durand	Paris	360
Dutoit	Paris	322
Dutoit	Paris	312
Dutoit	Oslo	319

D

Nom
Dutoit
Dufour

S

Nom	Destination	Code-dépl
Dufour	Paris	321
Dufour	Milan	325
Durand	Paris	588
Janssens	Prague	322

T

Nom	Rembours
Dufour	2
Dutoit	4
Janssens	0
Albrecht	2
Fanuel	3

R ∪ S

Nom	Destination	Code-dépl
Dufour	Paris	321
Dufour	Milan	325
Durand	Paris	360
Dutoit	Paris	322
Dutoit	Paris	312
Dutoit	Oslo	319
Durand	Paris	588
Janssens	Prague	322

R - S

Nom	Destination	Code-dépl
Durand	Paris	360
Dutoit	Paris	322
Dutoit	Paris	312
Dutoit	Oslo	319

R x T

R.Nom	Destination	Code-dépl	T.Nom	Rembours
Dufour	Paris	321	Dufour	2
Dufour	Paris	321	Dutoit	4

Dufour	Paris	321	Janssens	0
Dufour	Paris	321	Albrecht	2
Dufour	Paris	321	Fanuel	3
Dufour	Milan	325	Dufour	2
...
Dutoit	Oslo	319	Fanuel	3

R JOIN T

Nom	Destination	Code-dépl	Rembours
Dufour	Paris	321	2
Dufour	Milan	325	2
Dutoit	Paris	322	4
Dutoit	Paris	312	4
Dutoit	Oslo	319	4

 $\pi_{\text{Nom, Destination}}(R)$

Nom	Destination
Dufour	Paris
Dufour	Milan
Durand	Paris
Dutoit	Paris
Dutoit	Oslo

 $\sigma_{\text{Rembours} < 3}(T)$

Nom	Rembours
Dufour	2
Janssens	0
Albrecht	2

1.b.iii. **L'analyse relationnelle.**

Les notations ensemblistes permettent d'atteindre les mêmes résultats. Ainsi, pour les exemples qui précèdent, on pourrait utiliser les notations équivalentes suivantes :

$$R \cup S = \{ t \mid t \in R \text{ ou } t \in S \}$$

$$R - S = \{ t \mid t \in R \text{ et } t \notin S \}$$

$$R \times T = \{ (R.Nom, Destination, Code-dépl, T.Nom, Rembours) \mid \\ (R.Nom, Destination, Code-dépl) \in R \text{ et } (T.Nom, Rembours) \in T \}$$

$$R \text{ JOIN } T = \{ (Nom, Destination, Code-dépl, Rembours) \mid \\ (Nom, Destination, Code-dépl) \in R \text{ et } (Nom, Rembours) \in T \}$$

ou encore

$$R \text{ JOIN } T = \{ (Nom, Destination, Code-dépl, Rembours) \mid \\ (R.Nom, Destination, Code-dépl) \in R \\ \text{et} \quad (T.Nom, Rembours) \in T \\ \text{et} \quad R.Nom = T.Nom \}$$

$$\pi_{Nom, Destination}(R) = \{ (Nom, Destination) \mid \exists \text{ Code-dépl} : (Nom, Destination, Code-dépl) \in R \}$$

$$\sigma_{Rembours < 3}(T) = \{ (Nom, Rembours) \in T \mid Rembours < 3 \}$$

Ces notations ensemblistes ont directement inspiré le langage SQL, comme le montrent les exemples suivants :

$R \text{ JOIN } T$ s'obtient par l'instruction

```
SELECT R.Nom, Destination, Code-dépl, Rembours
FROM R,T
WHERE R.Nom = T.Nom;
```

$\sigma_{Rembours < 3}(T)$ s'obtient par l'instruction

```
SELECT Nom, Rembours
FROM T
WHERE Rembours < 3 ;
```

1.b.iv. **Exemple et exercices.**

Soit la DB à 3 tables, bien utile aux fidèles des bistrots du coin :

- 1) « Freq » est la table des fréquentations, ses deux attributs sont : Buveur et Bar.
Un tuple (Buveur, Bar) est dans la table si ce buveur fréquente ce bar.
- 2) « Sert » est la liste des bières servies dans les bars; ses 2 attributs sont : Bar et Bière. Par exemple (café-de-la-gare, jupiler) sera un tuple de la table si on sert bien cette marque de bière dans ce bistrot.
- 3) « Aime » est la table qui reprend les goûts des buveurs. Les tuples sont formés d'un buveur (attribut Buveur) et d'une marque de bière (attribut Bière).

Ecrivez un exemple de contenu de ces tables, puis répondez aux questions suivantes.

Ex.1. Donnez le contenu précis des tables suivantes, après avoir reformulé les requêtes en français :

- a) $\pi_{\text{Buveur}}(\text{Freq})$
- b) Freq JOIN Sert
- c) $(\text{Freq JOIN Sert}) \text{ JOIN Aime}$
- d) $(\text{Freq JOIN Sert}) \text{ JOIN}_{\text{Freq.Buveur}=\text{Aime.buveur}} \text{ Aime}$
- e) $(\text{Freq JOIN Sert}) \text{ JOIN}_{\text{Freq.Buveur} \neq \text{Aime.buveur}} \text{ Aime}$
- f) $\sigma_{\text{Bar} \neq \text{Café-de-la-gare}} \text{ Freq}$
- g) $\pi_{\text{Buveur}} \sigma_{\text{Bar} \neq \text{Café-de-la-gare}} \text{ Freq}$
- h) $(\pi_{\text{Buveur}} \text{ Freq}) \cup (\pi_{\text{Buveur}} \text{ Aime})$
- i) $\{ \text{Buveur} \mid (\exists \text{ Bar}) ((\text{Buveur}, \text{Bar}) \in \text{Freq} \wedge (\text{Bar}, 'Stella') \in \text{Sert}) \}$
- j) $\{ (\text{Buveur}, \text{Bière}) \mid (\neg \exists \text{ Bar}) ((\text{Buveur}, \text{Bar}) \in \text{Freq} \wedge (\text{Bar}, \text{Bière}) \notin \text{Sert}) \}$

Ex.2. Exprimez les requêtes suivantes dans chacune des notations vues:

- a) Quelles bières sert-on au Café de la Gare ?
- b) Quels sont les buveurs qui fréquentent un bar où l'on sert de la Jupiler?
- c) Quels sont les bars où l'on ne sert pas de Jupiler ?
- d) Quels sont les bars où l'on ne sert que de la Jupiler ?
- e) Quels sont les buveurs qui fréquentent le Café de la Poste et aiment la Stella ?
- f) Quels sont les buveurs qui fréquentent (au moins) un bar où l'on sert (au moins) une bière qu'ils aiment ?
- g) Quels sont les buveurs qui ne fréquentent que des bars où l'on sert au moins une bière qu'ils aiment ? (Difficile!)
- h) Quels sont les buveurs ne fréquentant aucun bar servant une bière qu'ils aiment ? (Difficile!)

2. Requêtes (SELECT)

Dans le cadre de ce cours nous illustrerons la théorie et les exercices à l'aide de la base de données pubs2. Le diagramme de Bachman de cette B.D. se trouve en annexe.

2.a. SELECT ... FROM ... WHERE ... ORDER BY ...

```
SELECT [ ALL | DISTINCT ]
      * | nom_colonne [ [ AS ] nom_d_affichage ] [, ...]
[ FROM nom_table ]
[ WHERE condition ]
[ ORDER BY nom_colonne [ ASC | DESC ] [, ...] ]
```

- *condition* est une expression dont le résultat est de type boolean. Toute ligne qui ne satisfait pas cette condition est éliminée de la sortie. Une ligne satisfait la condition si elle retourne vrai quand les valeurs réelles de la ligne sont substituées à toute référence de variable.

```
condition =
  conditionélémentaire { ( "AND" | "OR" ) conditionélémentaire } .
conditionélémentaire =
  ( [ "NOT" ] nomattribut ( "=" | "<>" [ "<" | ">" | "<=" | ">=" ] constante
    | nom_attribut ["NOT"] "IN" "(" constante { "," constante } ")"
    | nom_attribut ["NOT"] "LIKE" modèle_de_chaine_de_caractères_like
    | nom_attribut ["NOT"] "SIMILAR TO" modèle_de_chaine_de_caractères_similar
    | nom_attribut ["NOT"] "BETWEEN" constante "AND" constante
    | nom_attribut "IS" [ "NOT" ] "NULL"
    | [ "NOT" ] "(" condition ")"
  )
```

- **DISTINCT** élimine les lignes dupliquées du résultat. ALL (la valeur par défaut) renvoie toutes les lignes candidates, y compris les lignes dupliquées.
- **ORDER BY** impose le tri des lignes de résultat suivant les expressions spécifiées. Si deux lignes sont identiques suivant l'expression la plus à gauche, elles sont comparées avec l'expression suivante et ainsi de suite. Si elles sont identiques pour toutes les expressions de tri, elles sont renvoyées dans un ordre dépendant de l'implantation.

Chaque nom_colonne peut être le nom ou le numéro ordinal d'une colonne en sortie (élément de la liste SELECT).

ASC (par défaut) spécifie que le tri est ascendant, DESC que le tri est descendant.

- **modèle_de_chaine_de_caractères_like peut contenir :**
 - le caractère "%", qui remplace n'importe quelle séquence (même vide) de caractères.
 - le caractère "_", qui remplace exactement un caractère
 - n'importe quel autre caractère, qui se représente lui-même.

Exemple : WHERE nom LIKE '_m%' impose qu'il y ait un "m" en 2ème position.

Attention une chaîne de caractères est toujours entourée par des apostrophes simples. Les double guillemets peuvent être utilisés pour délimiter le nom des tables ou colonnes, mais pas pour délimiter les chaînes de caractères.

L'opérateur de concaténation des chaînes de caractères est la double barre verticale : ||

- **modèle_de_chaine_de_caractères_similar** est une extension de **modèle_de_chaine_de_caractères_like**. Il supporte en plus :
 - les parenthèses "()" peuvent être utilisées pour grouper
 - le caractère "|" représente une alternative
 - le caractère "*" représente une répétition de zéro, une ou plusieurs fois le caractère précédent

- le caractère "+" représente une répétition de une ou plusieurs fois le caractère précédent
- les caractères "[]" représentent un caractère unique parmi ceux contenus entre les crochets.

2.a.i. Exemples

Afficher la table complète des auteurs (tous les attributs, tous les tuples).

```
SELECT * FROM Authors ;
```

ou

```
SELECT Authors.* FROM Authors;
```

ou

```
SELECT au_id, au_lname, au_fname, address, city, state, country  
FROM Authors ;
```

ou

```
SELECT Authors.au_id, Authors.au_name, Authors.au_fname,  
        Authors.address, Authors.city, Authors.state, Authors.country  
FROM Authors ;
```


Afficher la liste des auteurs californiens (tous les tuples).

```
SELECT *  
FROM Authors  
WHERE state = 'CA' ;
```

ou, s'il y a risque d'ambiguïté :

```
SELECT *  
FROM Authors  
WHERE state = 'CA' AND country = 'USA' ;
```

Afficher la liste des noms et prénoms des auteurs, triée par le nom et puis le prénom.

```
SELECT au_lname, au_fname  
FROM Authors  


```
ORDER BY 1 ASC, au_fname ASC;
```


```

Afficher la liste des villes des auteurs.

```
SELECT city  
FROM Authors;
```

Afficher la liste des villes où habite au moins un auteur.

```
SELECT DISTINCT city  
FROM Authors;
```

Quelles sont les villes des USA où habite au moins un auteur dont le nom commence par "A" ?

```
SELECT DISTINCT city
FROM Authors
WHERE country = 'USA' AND au_fname LIKE 'A%';
```

Même question, mais pour les noms commençant par "A", "B" ou "C" ?

```
SELECT DISTINCT city
FROM Authors
WHERE country = 'USA' AND au_fname SIMILAR TO '[ABC]%';
```

Afficher les titres des livres dont le prix est entre 10 et 20 \$ (les deux compris).

```
SELECT title
FROM Titles
WHERE price BETWEEN 10 AND 20 ;
```

ou

```
SELECT title
FROM Titles
WHERE price >= 10 AND price <= 20 ;
```

Quels sont les titres et les prix des livres pour lesquels le total des ventes est inconnu ou nul ?

```
SELECT title, price
FROM Titles
WHERE total_sales IS NULL OR total_sales = 0 ;
```

Quels sont les titres et les prix des livres pour lesquels le total des ventes est connu et le prix situé entre 10 et 20 \$ ou supérieur à 250 \$, mais dont le titre ne contient pas "Anger" ?

```
SELECT title, price
FROM Titles
WHERE total_sales IS NOT NULL
      AND (price BETWEEN 10 AND 20 OR price >250) AND title NOT LIKE '%Anger%';
```

Quels sont les auteurs qui habitent en Californie ou en Oregon ou au Texas ?

```
SELECT *
FROM Authors
WHERE state IN ( 'CA', 'OR', 'TX' );
```


2.a.ii. **Exercices**

1. Quels sont les noms des auteurs habitant la ville de Oakland ?
2. Donnez les noms et adresses des auteurs dont le prénom commence par la lettre "A".
3. Donnez les noms et adresses complètes des auteurs qui n'ont pas de numéro de téléphone.
4. Y a-t-il des auteurs californiens dont le numéro de téléphone ne commence pas par "415" ?
5. Quels sont les auteurs habitant au Bénélux ?
6. Donnez les identifiants des éditeurs ayant publié au moins un livre de type "psychologie" ?
7. Donnez les identifiants des éditeurs ayant publié au moins un livre de type "psychologie", si l'on omet tous les livres dont le prix est compris entre 10 et 25 \$?
8. Donnez la liste des villes de Californie où l'on peut trouver un (ou plusieurs) auteur(s) dont le prénom est Albert ou dont le nom finit par "er".
9. Donnez tous les couples Etat-pays ("state" - "country") de la table des auteurs, pour lesquels l'Etat est fourni, mais le pays est autre que "USA".
10. Pour quels types de livres peut-on trouver des livres de prix inférieur à 15 \$?

2.b. Produit cartésien et jointure

```
SELECT [ ALL | DISTINCT ]
      * | expression [ [ AS ] nom_d_affichage ] [, ...]
[ FROM éléments_from [, ...] ]
[ WHERE condition ]
[ ORDER BY order_expression [ ASC | DESC ] [, ...] ]
```

- avec *éléments_from* qui peut être :

```
nom_table [ * ] [ [ AS ] alias [ ( alias_colonne [, ...] ) ] ]
[,éléments_from]
```

- condition* =
conditionélémentaire { ("AND" | "OR") *conditionélémentaire* } .
conditionélémentaire =
 (["NOT"] **expression** ("=" | "<>" ["<" | ">" | "<=" | ">="]) **expression**
 | *nom_attribut* ["NOT"] "IN" "(" *constante* { "," *constante* } ")"
 | *nom_attribut* ["NOT"] "LIKE" *modèle_de_chaine_de_caractères*
 | *nom_attribut* ["NOT"] "BETWEEN" *constante* "AND" *constante*
 | *nom_attribut* "IS" ["NOT"] "NULL"
 | ["NOT"] "(" *condition* ")"
)
- Une **expression** est le plus souvent un simple nom d'attribut, précédé éventuellement du nom de la table ou de son synonyme. Le nom de la table (ou le synonyme) est obligatoire dans le cas où il y a ambiguïté, c'est-à-dire dans le cas où ce nom d'attribut se retrouve dans plusieurs des tables utilisées dans ce SELECT.
 Mais une "expression" peut être plus complexe que cela : on peut avoir une expression arithmétique "classique", constituée d'opérateurs (+, -, *, /) et d'opérandes qui peuvent être :
 - des noms d'attributs (avec éventuellement nom de table ou synonyme)
 - des constantes
 - des fonctions agrégées (cf. annexe : SUM, COUNT, MIN, MAX, AVG) (confer 2.c)
 - des sous-expressions entre parenthèses
 - des fonctions de conversion d'un type à un autre.
 Similairement, il existe des expressions de type chaîne de caractères, dates, etc. (confer annexes).
- L'*order_expression* peut être le nom ou le numéro ordinal d'une colonne en sortie ou une valeur arbitraire formée à partir de valeurs des colonnes.

2.b.i. Produit cartésien

	titre	num_éditeur		num_éditeur	nom_éditeur
Livres			Editeurs		
	Les bleus	2		1	Bordas
	Les rouges	3		2	Springer
	Les verts	2		3	Larousse
	Les blancs	2			

```
SELECT Livres.*, Editeurs.*
FROM Livres, Editeurs ;
```

titre	Lives.num_éditeur	Editeurs.num_éditeur	nom_éditeur
Les bleus	2	1	Bordas
Les rouges	3	1	Bordas
Les verts	2	1	Bordas
Les blancs	2	1	Bordas
Les bleus	2	2	Springer
Les rouges	3	2	Springer
Les verts	2	2	Springer
Les blancs	2	2	Springer
Les bleus	2	3	Larousse
Les rouges	3	3	Larousse
Les verts	2	3	Larousse
Les blancs	2	3	Larousse

Chaque tuple de la table des livres est accolé à chaque tuple de la table des éditeurs. Cette opération est rarement utile. La plupart du temps, on ne désire accoler à chaque tuple d'une table que certains tuples de l'autre. On parlera dans ce cas de jointure. Dans l'exemple ci-dessus, on souhaitera normalement faire correspondre à chaque livre son éditeur. On accolera donc uniquement les paires de tuples des deux tables présentant le même numéro d'éditeur. En outre, on omettra de répéter deux fois le numéro d'éditeur dans le résultat. Cela donne l'exemple suivant.

2.b.ii. **Jointure**

```
SELECT Livres.*, Editeurs.nom_éditeur
FROM Livres, Editeurs
WHERE Livres.num_éditeur = Editeurs.num_éditeur;
```

titre	Lives.num_éditeur	nom_éditeur
Les bleus	2	Springer
Les verts	2	Springer
Les blancs	2	Springer
Les rouges	3	Larousse

La norme SQL2 propose des alternatives à cette écriture, utilisant le mot-clé JOIN, par ex. :

```
SELECT * FROM Livres JOIN Editeurs ;
```

2.b.iii. Exemples

Afficher la liste des livres en donnant pour chacun son titre, son prix et le nom de l'éditeur.

```
SELECT title, price, pub_name
FROM Titles, Publishers
WHERE Titles.pub_id = Publishers.pub_id ;
```

ou encore, en utilisant des "synonymes" :

```
SELECT title, price, pub_name
FROM Titles T, Publishers P
WHERE T.pub_id = P.pub_id ;
```

ou

```
SELECT T.title, T.price, P.pub_name
FROM Titles T, Publishers P
WHERE T.pub_id = P.pub_id ;
```

Afficher la liste des livres de psychologie édité par un éditeur de Massachusetts, en donnant pour chacun son titre, son prix et le nom de l'éditeur.

```
SELECT T.title, T.price, P.pub_name
FROM Titles T, Publishers P
WHERE T.pub_id = P.pub_id AND T.type = 'psychology' AND P.state = 'MA' ;
```

Pour chaque auteur de chaque livre de moins de 25 \$, donner le nom de l'auteur, son identifiant, l'identifiant et le titre du livre, et enfin le prix de celui-ci.

```
SELECT A.au_name, A.au_id, T.title_id, T.title, T.price
FROM Authors A, Titleauthor TA, Titles T
WHERE A.au_id = TA.au_id AND TA.title_id = T.title_id AND T.price < 25 ;
```

Trouvez les auteurs américains ayant fait publier un livre chez un éditeur américain habitant un autre état qu'eux. Pour chacune de ces paires auteur-éditeur, donnez les deux noms et les deux états.

```
SELECT DISTINCT A.au_name, A.state, P.pub_name, P.state
FROM Authors A, Titleauthor TA, Titles T, Publishers P
WHERE A.au_id = TA.au_id
AND TA.title_id = T.title_id
AND T.pub_id = P.pub_id
AND A.country = 'USA'
AND P.country = 'USA'
AND P.state <> A.state ;
```

Donnez les livres publiés par "Algodata Infosystems", en mentionnant pour chacun : son titre, son prix en \$, son prix converti en francs belges, le nombre d'exemplaires vendus, le montant total de cette vente en \$, ce même montant total converti en francs belges, tout en vous limitant aux livres pour lesquels ce dernier montant dépasse les 150.000 BEF. On suppose que 1 \$ = 35 BEF.

```
SELECT title, price, price * 35, total_sales, total_sales * price, total_sales * price * 35
FROM Titles T, Publishers P
WHERE T.pub_id = P. pub_id
      AND P.pub_name = 'Algodata Infosystems'
      AND total_sales * price * 35 > 100000 ;
```

Cas particulier : jointure d'une table avec elle-même

Y a-t-il des auteurs qui habitent dans la même ville ?

```
SELECT A1.au_id, A1.au_lname, A2.au_id, A2.au_lname, A1.city
FROM Authors A1, Authors A2
WHERE A1.city = A2.city;
```

Cette solution est imparfaite : en effet, tout auteur sera affiché comme habitant dans la même ville que lui-même! Voici une amélioration évitant cet inconvénient :

```
SELECT A1.au_id, A1.au_lname, A2.au_id, A2.au_lname, A1.city
FROM Authors A1, Authors A2
WHERE A1.city = A2.city
      AND A1.au_id <> A2.au_id;
```

Il reste un léger inconvénient : si X et Y habitent la même ville, on trouvera deux fois la paire, d'une part dans l'ordre X Y, et d'autre part dans l'ordre YX. Evitons cela :

```
SELECT A1.au_id, A1.au_lname, A2.au_id, A2.au_lname, A1.city
FROM Authors A1, Authors A2
WHERE A1.city = A2.city
      AND A1.au_id < A2.au_id;
```

2.b.iv. Exercices

1. Affichez la liste de tous les livres, en indiquant pour chacun son titre, son prix et le nom de son éditeur.
2. Affichez la liste de tous les livres de psychologie, en indiquant pour chacun son titre, son prix et le nom de son éditeur.
3. Quels sont les auteurs qui ont effectivement écrit un (des) livre(s) présent(s) dans la DB ? Donnez leurs noms et prénoms.
4. Dans quels Etats y a-t-il des auteurs qui ont effectivement écrit un (des) livre(s) présent(s) dans la DB ?
5. Donnez les noms et adresses des magasins qui ont commandé des livres en novembre 1991.
6. Quels sont les livres de psychologie de moins de 20 \$ édités par des éditeurs dont le nom ne commence pas par "Algo" ?
7. Donnez les titres des livres écrits par (au moins) un auteur californien (state = "CA").
8. Quels sont les auteurs qui ont écrit un livre (au moins) publié par un éditeur californien ?
9. Quels sont les auteurs qui ont écrit un livre (au moins) publié par un éditeur localisé dans leur Etat ?
10. Quels sont les éditeurs dont on a vendu des livres entre le 1/11/1990 et le 1/3/1991 ?
11. Quels magasins ont vendu des livres contenant le mot "cook" (ou "Cook") dans leur titre ?
12. Y a-t-il des paires de livres publiés par le même éditeur à la même date ?
13. Y a-t-il des auteurs n'ayant pas publié tous leurs livres chez le même éditeur ?
14. Y a-t-il des livres qui ont été vendus avant leur date de parution ?
15. Quels sont les magasins où l'on a vendu des livres écrits par Anne Ringer ?
16. Quels sont les Etats où habite au moins un auteur dont on a vendu des livres en Californie en février 1991 ?
17. Y a-t-il des paires de magasins situés dans le même Etat, où l'on a vendu des livres du même auteur ?
18. Trouvez les paires de co-auteurs.
19. Pour chaque détail de vente, donnez le titre du livre, le nom du magasin, le prix unitaire, le nombre d'exemplaires vendus, le montant total et le montant de l'éco-tax totale (qui s'élève à 2% du chiffre d'affaire).

2.c. Fonctions d'agrégation

```
SELECT [ ALL | DISTINCT ]
      * | expression [ [ AS ] nom_d_affichage ] [, ...]
[ FROM éléments_from [, ...] ]
[ WHERE condition ]
[ ORDER BY order_expression [ ASC | DESC ] [, ...] ]
```

- Comme vu en 2.b, une **expression** peut être plusieurs choses. Entre autres, elle peut être un des fonctions agrégées suivantes :
 - **COUNT** ([ALL | DISTINCT] colonne)
 - **SUM** ([ALL | DISTINCT] colonne)
 - **MIN** ([ALL | DISTINCT] colonne)
 - **MAX** ([ALL | DISTINCT] colonne)
 - **AVG** ([ALL | DISTINCT] colonne)
 - **COUNT** (*)

Le rôle d'une fonction d'agrégation est de calculer une **valeur unique pour la colonne spécifiée d'un ensemble de tuples**. Sous la forme décrite ici, ce sont tous les tuples concernés par le SELECT qui font partie de cet ensemble.

Ainsi SUM(colonne) fera la somme de tous les nombres contenus dans la colonne. COUNT comptera le nombre de tuples, MIN retournera la valeur la plus petite, MAX la plus grande et AVG la moyenne numérique.

Lorsqu'une des expressions du SELECT est une fonction d'agrégation, toutes les autres expressions doivent l'être aussi. En effet la fonction d'agrégation retourne un résultat unique, il faut donc que les autres expressions retournent aussi un résultat unique afin que le SELECT puisse retourner un tuple unique.

2.c.i. Exemples

Quel est le prix le plus élevé et quel est le prix le plus bas parmi les livres de psychologie ?

```
SELECT MAX ( price ), MIN ( price )
FROM Titles
WHERE type = 'psychology' ;
```

Quel est le prix moyen des livres édités par "Algodata Infosystems" ?

```
SELECT AVG ( T.price )
FROM Titles T, Publishers P
WHERE T.pub_id = P.pub_id
      AND P.pub_name = 'Algodata Infosystems' ;
```

Donnez le nombre de co-auteurs californiens du livre "Les bleus".

```
SELECT COUNT ( * )
FROM Authors A, Titleauthor TA, Titles T
WHERE A.au_id = TA.au_id
      AND TA.title_id = T.title_id
      AND A.state = 'CA'
      AND T.title = 'Les bleus';
```

...variante :

```
SELECT COUNT ( A.au_id )  
FROM ... (idem)
```

Dans combien d'états des Etats-Unis y a-t-il des auteurs répertoriés dans notre DB ?

```
SELECT COUNT(DISTINCT state)  
FROM Authors  
WHERE country = 'USA' ;
```

Quel est le montant total des ventes effectuées en Californie en mai 1989 ?

```
SELECT SUM ( SD.qty * T.price )  
FROM Titles T, Salesdetail SD, Sales S, Stores St  
WHERE T.title_id = SD.title_id  
AND SD.stor_id = S.stor_id  
AND SD.ord_num = S.ord_num  
AND S.stor_id = St.stor_id  
AND St.state = 'CA'  
AND S.date BETWEEN '1989/05/01' AND '1989/05/31';
```


2.d. GROUP BY ... HAVING ...

```
SELECT [ ALL | DISTINCT ]
      * | expression [ [ AS ] nom_d_affichage ] [, ...]
[ FROM éléments_from [, ...] ]
[ WHERE condition ]
[ GROUP BY group_expression [, ...] ]
[ HAVING having_condition [, ...] ]
[ ORDER BY order_expression [ ASC | DESC ] [, ...] ]
```

- **GROUP BY** rassemble les tuples en groupes déterminés par `group_expression`. `group_expression` peut être le nom d'une ou plusieurs colonnes séparés par des virgules.

Pour chaque groupe, le `SELECT` retournera un tuple unique. Chaque champ de ce tuple pourra être :

- Soit une fonction d'agrégation (`COUNT`, `SUM`, `MAX`, `MIN`, `AVG` confer 2.c) qui porte alors sur tous les tuples de ce groupe particulier.
 - Soit d'une ou plusieurs des colonnes utilisées pour regrouper. En effet ces colonnes possèdent toutes la même valeur au sein du groupe et c'est cette valeur unique qui est retournée par ce groupe.
- **HAVING** filtre les tuples représentant les groupes suivant `having_condition`. Cette condition porte sur des groupes entiers, au contraire de `WHERE` qui filtre sur les tuples individuels avant même leurs formations en groupes.

`having_condition` doit aussi porter soit sur des fonctions d'agrégation, soit sur des colonnes utilisées pour regrouper.

2.d.i. Exemples

Pour chaque type de livre, donnez le nom du type et le nombre de livres de ce type.

```
SELECT type, COUNT ( * )
FROM Titles
GROUP BY type;
```

Chaque groupe de tuples de la table `Titles` possédant la même valeur de l'attribut "type" donnera lieu à une seule ligne dans le résultat.

On ne peut forcément afficher que deux sortes d'éléments :

- la valeur d'un attribut ayant servi à grouper ("type" dans l'exemple)
- une fonction d'agrégation (comme `COUNT(*)` dans l'exemple)

(ou une combinaison de tels éléments), car tout autre élément n'aurait pas une valeur unique pour tout le groupe. Une instruction telle que :

```
SELECT type, price, COUNT ( * )
FROM Titles
GROUP BY type ;
```

n'a donc aucun sens ; elle doit être refusée par la machine SQL!

Pour chaque type de livre, donnez le nom du type, le nombre de livres de plus de 15 \$ de ce type ainsi que la moyenne des prix de ces livres.

```
SELECT type, COUNT ( * ), AVG ( price )
FROM Titles
WHERE price > 15
GROUP BY type ;
```

Remarquons que les types pour lesquels aucun livre ne dépasse les 15 \$ ne figureront même pas dans la table-résultat ! Ceci pourra être corrigé par une opération d'UNION.

Pour chaque type de livre, donnez le nom du type et le nombre de livres de ce type. Limitez- vous aux types pour lesquels il y a au moins 4 livres !

```
SELECT type, COUNT ( * )
FROM Titles
GROUP BY type
HAVING COUNT ( * ) >= 4 ;
```

La clause "HAVING" est donc une condition sur les groupes, tout comme la clause "WHERE" était une condition sur les tuples.

Pour chaque éditeur et chaque type de livre, donnez l'identifiant de l'éditeur, le nom du type, et le nombre total d'exemplaires vendus.

```
SELECT pub_id, type, SUM ( total_sales )
FROM Titles
GROUP BY pub_id, type
```

Dormez la liste des auteurs californiens ayant écrit des livres de psychologie, avec, en regard de chacun, le nombre de livres de psychologie et le prix du plus cher d'entre eux.

```
SELECT A.au_lname, A.au_fname, COUNT ( * ), MAX ( T.price )
FROM Authors A, Titleauthor TA, Titles T
WHERE A.au_id = TA.au_id
AND TA.title_id = T.title_id
AND A.state = 'CA' AND T.type = 'psychology'
GROUP BY A.au_id;
```

Remarquons que la requête sélectionne les attributs A.au_lname et A.au_fname dans la liste des colonnes à afficher. Pour un A.au_id donné, le A.au_lname et A.au_fname vont rester constant. Ainsi le regroupement par identifiant d'auteur (A.au_id) est suffisant.

D'autre part, on pourrait vouloir plutôt regrouper sur A.au_lname et A.au_fname plutôt que sur A.au_id. D'apparence correcte, cette solution présente toutefois un danger : si par hasard deux auteurs distincts étaient parfaitement homonymes (même nom et même prénom), leurs livres seraient comptabilisés ensemble, ce qui constituerait une erreur !

Y a-t-il des villes où habitent plus de 3 auteurs? Lesquelles et combien d'auteurs y trouve-t-on?

```
SELECT city, COUNT (*)  
FROM Authors  
GROUP BY city  
HAVING COUNT (*) > 3
```

Pour chaque type de livre, combien y a-t-il d'auteurs qui y ont touché ?

```
SELECT T.type, COUNT (DISTINCT TA.au_id)  
FROM Titles T, Titleauthor TA  
WHERE T.title_id = TA.title_id  
GROUP BY T.type ;
```

2.d.ii. **Exercices**

1. Quel est le prix moyen des livres édités par "Algodata Infosystems" ?
2. Quel est le prix moyen des livres écrits par chaque auteur ? (Pour chaque auteur, donnez son nom, son prénom et le prix moyen de ses livres.)
3. Pour chaque livre édité par "Algodata Infosystems", donnez le prix du livre et le nombre d'auteurs.
4. Pour chaque livre, donnez son titre, son prix, et le nombre de magasins différents où il a été vendu.
5. Quels sont les livres qui ont été vendus dans plusieurs magasins ?
6. Pour chaque type de livre, donnez le nombre total de livres de ce type ainsi que leur prix moyen.
7. Pour chaque livre, le "total_sales" devrait normalement être égal au nombre total des ventes enregistrées pour ce livre, c'est-à-dire à la somme de toutes les "qty" des détails de vente relatifs à ce livre. Vérifiez que c'est bien le cas en affichant pour chaque livre ces deux valeurs côte à côte, ainsi que l'identifiant du livre.
8. Même question, mais en n'affichant que les livres pour lesquels il y a erreur.
9. Quels sont les livres ayant été écrits par au moins 3 auteurs ?
10. Combien d'exemplaires de livres d'auteurs californiens édités par des éditeurs californiens a-t-on vendus dans des magasins californiens ? (Attention, il y a un piège : si vous le détectez, vous devrez peut-être attendre un chapitre ultérieur avant de pouvoir résoudre correctement cet exercice...)

2.e. SOUS-SELECT

```

SELECT [ ALL | DISTINCT ]
      * | expression [ [ AS ] nom_d_affichage ] [, ...]
      [ FROM éléments_from [, ...] ]
      [ WHERE condition ]
      [ GROUP BY group_expression [, ...] ]
      [ HAVING having_condition [, ...] ]
      [ ORDER BY order_expression [ ASC | DESC ] [, ...] ]

• condition =
  conditionélémentaire { ( "AND" | "OR" ) conditionélémentaire } .
conditionélémentaire =
  ( [ "NOT" ] expression ( "=" | "<>" [ "<" | ">" | "<=" | ">=" ] expression
    | [ "NOT" ] expression ( "=" | "<" | ">" | "<=" | ">=" )
      [ "ALL" | "ANY" ] "(" instruction_select_à_1_colonne ")"
    | nom_attribut ["NOT"] "IN" "(" constante { "," constante } ")"
    | expression ["NOT"] "IN" "(" instruction_select_à_1_colonne ")"
    | nom_attribut ["NOT"] "LIKE" modèle_de_chaine_de_caractères
    | nom_attribut ["NOT"] "BETWEEN" constante "AND" constante
    | nom_attribut "IS" ["NOT"] "NULL"
    | [ "NOT" ] "EXISTS" "(" instruction_select ")"
    | [ "NOT" ] "(" condition ")"
  )

```

2.e.i. Exemples avec IN

Quels sont les auteurs habitant dans le même Etat qu'au moins un éditeur ?

```

SELECT au_id, au_lname, au_fname
FROM Authors
WHERE state IN ( SELECT State
                  FROM Publishers );

```

Si le sous-select donne pour résultat la table à une colonne contenant les valeurs 'CA', 'OR', 'TX', ce select est équivalent à :

```

SELECT au_id, au_lname, au_fname
FROM Authors
WHERE state IN ( 'CA', 'OR', 'TX' );

```

Les éventuelles répétitions de valeurs n'ont aucun impact : un DISTINCT dans le sous- select ne change rien au résultat.

Quels sont les auteurs habitant dans un Etat où n'habite aucun éditeur ?

```

SELECT au_id, au_lname, au_fname
FROM Authors
WHERE state NOT IN ( SELECT State
                     FROM Publishers );

```

Quels sont les auteurs habitant dans le même Etat qu'au moins un éditeur ayant publié un livre de business ?

```

SELECT au_id, au_lname, au_fname
FROM Authors
WHERE state IN ( SELECT P.state
                  FROM Publishers P, Titles T
                  WHERE P.pub_id = T.pub_id
                    AND T.type = 'business' );

```

ou

```

SELECT au_id, au_lname, au_fname
FROM Authors
WHERE state IN ( SELECT P.state
                  FROM Publishers P,
                  WHERE P.pub_id IN ( SELECT pub_id
                                      FROM titles T
                                      WHERE T.type = 'business' ));

```

...sans oublier la solution avec "**grande jointure**" :

```

SELECT DISTINCT A.au_id, A.au_lname, A.au_fname
FROM Authors A, Publishers P, Titles T
WHERE P.pub_id = T.pub_id
      AND A.state = P.state
      AND T.type = 'business' ;

```

Comme on peut le constater, **les jointures** et les **conditions avec IN** sont souvent **interchangeables**.
Ce n'est par contre pas le cas du "NOT IN" !

2.e.ii. **Exemples avec ("=" | "<>" | "<" | ">" | "<=" | ">=") ["ALL" | "ANY"]**

Quel est le livre le plus cher ?

```

SELECT title_id, title
FROM Titles
WHERE price = ( SELECT MAX(price)
                FROM Titles );

```

Attention, cette requête n'est pas strictement équivalente à :

```

SELECT title_id, title
FROM Titles
WHERE price >= ALL ( SELECT price
                     FROM Titles );

```

En effet dans ce second cas, l'opérateur >= doit comparer toutes les valeurs de price, y compris les null pour lequel il retourne faux ! Comme il y a des titres pour lesquels le price est null, cette requête retourne donc un résultat vide. Pour éviter ce problème, on devrait gérer le null explicitement :

```

SELECT title_id, title
FROM Titles
WHERE price >= ALL ( SELECT price
                     FROM Titles
                     WHERE price IS NOT NULL ) ;

```

*Quels sont les auteurs habitant dans le même Etat **qu'au moins** un éditeur ?*

```

SELECT au_id, au_lname, au_fname
FROM Authors
WHERE state = ANY ( SELECT state
                   FROM Publishers ) ;

```

Quels sont les livres coûtant plus cher que "Life Without Fear" ?

```

SELECT title_id, title
FROM Titles
WHERE price > ( SELECT price
               FROM Titles
               WHERE title = 'Life Without Fear' ) ;

```

L'absence de "ALL" et de "ANY" **n'est acceptable que lorsque le sous-select ne renvoie qu'une seule valeur** (table résultat à 1 ligne et 1 colonne). Dans l'exemple ci-dessus, il faut espérer qu'il n'y ait pas plusieurs livres intitulés "Life Without Fear"! Par contre, l'exemple donné plus haut, avec "MAX(price) ", est plus sûr...

Autre solution, avec auto-jointure :

```

SELECT T1.title_id, T1.title
FROM Titles T1, Titles T2
WHERE T1.price > T2.price
      AND T2 .title = 'Life Without Fear' ;

```

Quels sont les auteurs qui ont écrit exactement 2 livres ?

```

SELECT A.au_id, A.au_lname, A. au_fname
FROM Authors A
WHERE 2 = ( SELECT COUNT ( * )
           FROM Titleauthor TA
           WHERE TA.au_id = A.au_id ) ;

```

*Ce dernier exemple introduit une nouvelle notion : le sous-select fait référence à une table du **select principal**.*

L'effet est le suivant : pour chaque tuple de la table Authors, lorsqu'on veut savoir si la condition du WHERE est remplie, on réévalue le sous-select en utilisant la valeur de "A.au_id" figurant dans ce tuple-là.

Quels sont les livres coûtant plus cher que la moyenne de tous les livres ?

```
SELECT title_id, title, price
FROM Titles
WHERE price > ( SELECT AVG(price)
                FROM Titles );
```

Quels sont les livres coûtant plus cher que la moyenne de tous les livres du même type ?

```
SELECT T1.title_id, T1.title, T1.price
FROM Titles T1
WHERE T1.price > ( SELECT AVG ( T2.price )
                  FROM Titles T2
                  WHERE T2.type = T1.type );
```

2.e.iii. Exemples avec EXISTS

Quels sont les auteurs habitant dans le même Etat qu'au moins un éditeur ?

```
SELECT au_id, au_lname, au_fname
FROM Authors A
WHERE EXISTS ( SELECT *
               FROM Publishers P
               WHERE P.state = A.state );
```

Quels sont les auteurs habitant un Etat où n'habite aucun éditeur ?

```
SELECT au_id, au_lname, au_fname
FROM Authors A
WHERE NOT EXISTS ( SELECT *
                  FROM Publishers P
                  WHERE P.state = A.state );
```

Quels sont les auteurs qui n'ont écrit aucun livre publié par "Algodata Infosystems" ?

```
SELECT au_id, au_lname, au_fname
FROM Authors A
WHERE NOT EXISTS ( SELECT *
                  FROM Titleauthor TA, Titles T, Publishers P
                  WHERE TA.title_id = T.title_id
                  AND T.pub_id = P.pub_id
                  AND P.name = 'Algodata Infosystems' );
```



```

AND P.pub_name = 'Algodata Infosystems'
AND TA.au_id = A.au_id);

```

ou

```

SELECT au_id, au_lname, au_fname
FROM Authors A
WHERE au_id NOT IN ( SELECT TA.au_id
                     FROM Titleauthor TA, Titles T, Publishers P
                     WHERE TA.title_id = T.title_id
                           AND T.pub_id = P.pub_id
                           AND P.pub_name = 'Algodata Infosystems' );

```

ou

```

SELECT au_id, au_lname, au_fname
FROM Authors A
WHERE au_id NOT IN
  ( SELECT TA.au_id
    FROM Titleauthor TA
    WHERE title_id IN
      ( SELECT title_id
        FROM Titles
        WHERE pub_id =
          ( SELECT pub_id
            FROM Publishers
            WHERE pub_name = 'Algodata Infosystems' ) ));

```

Quels sont les magasins où l'on a vendu tous les livres édités par "Algodata Infosystems" ?

On peut reformuler cette question de la façon suivante :

Quels sont les magasins pour lesquels il n'existe aucun livre édité par "Algodata Infosystems" que l'on ne retrouve pas dans leurs ventes ?

```

SELECT St.stor_id, St.stor_name
FROM Stores St
WHERE NOT EXISTS
  ( SELECT *
    FROM Titles T, Publishers P
    WHERE T.pub_id = P.pub_id
          AND P.pub_name = 'Algodata Infosystems'
          AND NOT EXISTS
            ( SELECT *
              FROM Salesdetail SD
              WHERE SD.title_id = T.title_id
                    AND SD.stor_id = St.stor_id ));

```

ou

```

SELECT St.stor_id, St.stor_name
FROM Stores St
WHERE NOT EXISTS

```

```
( SELECT *  
FROM Titles T, Publishers P  
WHERE T.pub_id = P.pub_id  
AND P.pub_name = 'Algodata Infosystems'  
AND T.title_id NOT IN  
  ( SELECT SD.title_id  
    FROM Salesdetail SD  
    WHERE SD.stor_id = St.stor_id ));
```

2.e.iv. **Exercices**

1. Quel est le livre le plus cher publié par l'éditeur "Algodata Infosystems" ?
2. Quels sont les livres qui ont été vendus dans plusieurs magasins ?
3. Quels sont les livres dont le prix est supérieur à une fois et demi le prix moyen des livres du même type ?
4. Quels sont les auteurs qui ont écrit un livre (au moins), publié par un éditeur localisé dans le même état ?
5. Quels sont les éditeurs qui n'ont rien édité ?
6. Quel est l'éditeur qui a édité le plus grand nombre de livres ?
7. Quels sont les éditeurs dont on n'a vendu aucun livre ?
8. Quels sont les différents livres écrits par des auteurs californiens, publiés par des éditeurs californiens, et qui n'ont été vendus *que* dans des magasins californiens ?
9. Quel est le titre du livre vendu le plus récemment ? (S'il a des ex-aequo, donnez-les tous.)
10. Quels sont les magasins où l'on a vendu (au moins) tous les livres vendus par le magasin "Bookbeat" ?
11. Quelles sont les villes de Californie où l'on peut trouver un auteur, mais aucun magasin ?
12. Quels sont les éditeurs localisés dans la ville où il y a le plus d'auteurs ?
13. Donnez les titres des livres dont tous les auteurs sont californiens.
14. Quels sont les livres qui n'ont été écrits par aucun auteur californien ?
15. Quels sont les livres qui n'ont été écrits que par un seul auteur ?
16. Quels sont les livres qui n'ont qu'un auteur, et tels que cet auteur soit californien ?

2.f. UNION, INTERSECTION, DIFFERENCE

```
SELECT [ ALL | DISTINCT ]
      * | expression [ [ AS ] nom_d_affichage ] [, ...]
[ FROM éléments_from [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY order_expression [ ASC | DESC ] [, ...] ]
```

- L'opérateur UNION calcule l'union ensembliste des lignes renvoyées par les instructions SELECT impliquées. Une ligne est dans l'union de deux ensembles de résultats si elle apparaît dans au moins un des ensembles. Les deux instructions SELECT qui représentent les opérandes directes de l'UNION doivent produire le même nombre de colonnes et les colonnes correspondantes doivent être d'un type de données compatible.

Sauf lorsque l'option ALL est spécifiée, il n'y a pas de doublons dans le résultat de UNION. ALL empêche l'élimination des lignes dupliquées. UNION ALL est donc significativement plus rapide qu'UNION, et sera préféré.

Si une instruction SELECT contient plusieurs opérateurs UNION, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent.

- L'opérateur INTERSECT calcule l'intersection des lignes renvoyées par les instructions SELECT impliquées. Une ligne est dans l'intersection des deux ensembles de résultats si elle apparaît dans chacun des deux ensembles.

Le résultat d'INTERSECT ne contient aucune ligne dupliquée sauf si l'option ALL est spécifiée. Dans ce cas, une ligne dupliquée m fois dans la table gauche et n fois dans la table droite apparaît min(m,n) fois dans l'ensemble de résultats.

Si une instruction SELECT contient plusieurs opérateurs INTERSECT, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent. INTERSECT a une priorité supérieure à celle d'UNION. C'est-à-dire que A UNION B INTERSECT C est lu comme A UNION (B INTERSECT C).

- L'opérateur EXCEPT calcule l'ensemble de lignes qui appartiennent au résultat de l'instruction SELECT de gauche mais pas à celui de droite.

Le résultat d'EXCEPT ne contient aucune ligne dupliquée sauf si l'option ALL est spécifiée. Dans ce cas, une ligne dupliquée m fois dans la table gauche et n fois dans la table droite apparaît max(m-n,0) fois dans l'ensemble de résultats.

Si une instruction SELECT contient plusieurs opérateurs EXCEPT, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent. EXCEPT a la même priorité qu'UNION.

2.f.i. Exemples

Quelles sont les villes où l'on peut trouver un auteur et/ou un éditeur ?

```
(SELECT DISTINCT city FROM Authors) UNION
(SELECT DISTINCT city FROM Publishers);
```

Pour chaque éditeur, donnez son nom et le nombre de livres de psychologie qu'il a édité. Classez le tout par ordre décroissant de nombre de livres.

```
( SELECT pub_name, COUNT ( title_id )
  FROM Publishers P, Titles T
```

```

WHERE P.pub_id = T.pub_id
AND type = 'psychology' GROUP BY P.pub_id, pub_name )
UNION
( SELECT pub_name,0
FROM Publishers
WHERE pub_id NOT IN
( SELECT pub_id
FROM Titles
WHERE type = 'psychology' ) )
ORDER BY 2 DESC;

```

2.g. OUTER JOIN (Jointures externes)

```

SELECT [ ALL | DISTINCT ]
* | expression [ [ AS ] nom_d_affichage ] [, ...]
[ FROM éléments_from [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY order_expression [ ASC | DESC ] [, ...] ]

```

- avec `éléments_from` qui peut être :

```

nom_table [ * ] [ [ AS ] alias [ ( alias_colonne [, ...] ) ] ]
[,éléments_from]
éléments_from [ NATURAL ] type_jointure éléments_from [ ON condition_jointure
| USING ( colonne_jointure [, ...] ) ]

```

- avec `type_jointure` qui peut être :
 - [INNER] JOIN** : équivalent à la jointure habituelle où la condition de jointure est exprimée dans la partie **WHERE** du **SELECT**.
 - LEFT [OUTER] JOIN** : tous les tuples de la table de gauche font partie du résultat ; ils sont joints à la table de droite quand c'est possible de par la condition de jointure. Sinon les colonnes correspondantes à la table de droite sont remplies de **null**.
 - RIGHT [OUTER] JOIN** : identique à **LEFT [OUTER] JOIN** mais en inversant les tables de gauche et de droite.
 - FULL [OUTER] JOIN** : tous les tuples de la table de gauche et de droite font partie du résultat, joints quand c'est possible sinon les colonnes manquantes sont remplies de **null**.

Une condition de jointure doit être choisie entre :

- ON** `condition_jointure` permet d'exprimer une condition de jointure sous une forme similaire à la condition du **WHERE**
- USING** (`a, b, ...`) est un raccourci pour **ON** `table_gauche.a = table_droite.a AND table_gauche.b = table_droite.b`
- NATURAL** est un raccourci pour une liste **USING** qui mentionne toutes les colonnes de même nom dans les deux tables.

2.g.i. Exemples

titre	num_éditeur	num_éditeur	nom_éditeur
Les bleus	2	1	Bordas

Livres	Les rouges	3	Editeurs	2	Springer
	Les verts	2		3	Larousse
	Les blancs	2			
	Les noirs	5			

```
SELECT titre, num_éditeur, nom_éditeur
FROM Livres NATURAL FULL OUTER JOIN Editeurs ;
```

titre	Lives.num_éditeur	nom_éditeur
Les bleus	2	Springer
Les verts	2	Springer
Les blancs	2	Springer
Les rouges	3	Larousse
Les noirs	5	NULL
NULL	1	Bordas

```
SELECT titre, num_éditeur, nom_éditeur
FROM Livres NATURAL LEFT OUTER JOIN Editeurs ;
```

titre	Lives.num_éditeur	nom_éditeur
Les bleus	2	Springer
Les verts	2	Springer
Les blancs	2	Springer
Les rouges	3	Larousse
Les noirs	5	NULL

```
SELECT titre, num_éditeur, nom_éditeur
FROM Livres NATURAL RIGHT OUTER JOIN Editeurs ;
```

titre	Lives.num_éditeur	nom_éditeur
Les bleus	2	Springer

Les verts	2	Springer
Les blancs	2	Springer
Les rouges	3	Larousse
NULL	1	Bordas

2.g.ii. **Exercices**

- 1 Donnez la liste des magasins, en ordre alphabétique, en mentionnant pour chacun le chiffre d'affaire total.
- 2 Donnez la liste des magasins, en mentionnant pour chacun le chiffre d'affaire total. Classez cette liste par ordre décroissant de chiffre d'affaire.
- 3 Donnez la liste des livres de plus de \$20, classés par type, en donnant pour chacun son type, son titre, le nom de son éditeur et son prix.
- 4 Donnez la liste des livres de plus de \$20, classés par type, en donnant pour chacun son type, son titre, les noms de ses auteurs et son prix.
- 5 Quelles sont les villes de Californie où l'on peut trouver un auteur et/ou un éditeur, mais aucun magasin ?
6. Donnez la liste des auteurs en indiquant pour chacun, outre son nom et son prénom, le nombre de livres de plus de 20 \$ qu'il a écrits. Classez cela par ordre décroissant de nombre de livres, et, en cas d'ex aequo, par ordre alphabétique. N'oubliez pas les auteurs qui n'ont écrit aucun livre de plus de 20 \$!

2.h. COALESCE

Dans une requête, il est possible d'utiliser l'expression COALESCE pour remplacer une valeur NULL par une valeur conventionnelle (par exemple la valeur 0 ou la chaîne vide). La particularité de cette fonction est d'accepter autant de paramètres que l'on veut. Elle renvoie la première expression évaluable (donc différente de NULL) dans l'ordre d'écriture (de gauche à droite).

Sa syntaxe est la suivante :

```
COALESCE(expression1 [, expression2 [, expression3 [ ... ] ] ])
```

2.h.i. Exemple

```
SELECT au_id, au_fname, au_lname, COALESCE(country, state, 'etat et pays inconnus')  
FROM authors
```

Pour chaque auteur, cette requête affichera quatre champs. Les trois premiers champs seront l'id, le prénom et le nom de l'auteur. Si son pays est différent de NULL, le quatrième champ sera son pays. Si son pays est NULL mais que son état est différent de NULL, alors le quatrième champ sera son état. Si son pays et son état sont NULL, alors le quatrième champ sera la chaîne de caractère 'etat et pays inconnus'.

2.h.ii. Exercice

1. Donnez la liste complète des couples auteur-livre avec, pour chaque couple, le nom et le prénom de l'auteur, son adresse complète, ainsi que l'identifiant et le titre du livre. Les auteurs qui n'ont rien écrit doivent aussi figurer dans le résultat (avec "aucun livre" comme titre de livre). A l'affichage, les tuples relatifs au même livre doivent se suivre.

3. Stockage de données

Grâce au chapitre précédent nous sommes capables d'interroger une base de données. Nous allons maintenant nous intéresser à créer cette base de donnée et en faire évoluer le contenu et la structure.

3.a. Ajout/Suppression/Modification de tables

3.a.i. Contraintes d'intégrité

Avant de rentrer dans le vif du sujet, il y a un aspect important à aborder. Une base de données fournit toute une série de services à l'application qui l'utilise : écrire des données, les récupérer, les modifier etc. En tant que point central de la gestion de ses données, la B.D. est un bon candidat pour fournir des services supplémentaires dont on aurait de toute façon besoin d'une manière récurrente.

Un de ces services est de vérifier qu'une donnée est du bon type. Par exemple un nombre d'occurrences devra être un entier. Un nom devra être une chaîne de caractère avec une certaine longueur maximale. Cette information fera donc partie de la définition même de la base de données.

Un autre de ces services porte sur l'aspect relationnel de la base de données. Par exemple une commande doit être en relation avec le vendeur et l'acheteur. Il est donc nécessaire que ces champs aient une valeur permettant d'effectuer ce lien. De plus on a besoin d'être sûr de pointer le bon vendeur et le bon acheteur. Par exemple pour l'acheteur on ne peut pas se baser uniquement sur son nom et son prénom car il y a un risque d'avoir des homonymes (Jean Dupont par exemple). On a donc besoin en interne d'avoir une manière d'identifier uniquement l'acheteur. Ces informations portent donc sur l'intégrité même des données de la BD, et cette dernière permet de spécifier des contraintes sur ce qui peut ou pas être présent : les contraintes d'intégrité. Il en existe de plusieurs types :

- Clef primaire : un ou plusieurs champs d'un tuple. La contrainte garantit que ces champs seront non null et unique au sein de la table. On pourra donc les utiliser comme identifiant unique de ce tuple.
- Clef étrangère : un ou plusieurs champs d'un tuple mis en relation avec la clef primaire d'une autre table. La contrainte garantit qu'à cette clef étrangère la clef primaire correspondante existe dans l'autre table.
- Pas null : un ou plusieurs champs d'un tuple. La contrainte garantit que tous ces champs contiennent une valeur réelle (pas null).
- Unicité : un ou plusieurs champs d'un tuple. La contrainte garantit que la combinaison de ces champs est unique au sein de la table.

3.a.ii. CREATE TABLE

```
CREATE TABLE nom_table ( [
  { nom_colonne type_donnees [ DEFAULT default_expr ] [ contrainte_colonne [...] ]
    | contrainte_table }
  [, ... ]
] )
```

- type_donnees peut être : character [(n)], character varying [(n)], smallint, integer, numeric [(p, s)], double precision, timestamp, ou tout autre type tel que listé dans les annexes.

- contrainte_colonne peut être :

```
[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL |
  UNIQUE |
  PRIMARY KEY |
  CHECK ( expression ) |
  REFERENCES table_reference [ ( colonne_reference ) ]
}
```

- contrainte_table peut être :

```
[ CONSTRAINT nom_contrainte ]
{ UNIQUE ( nom_colonne [, ... ] ) |
  PRIMARY KEY ( nom_colonne [, ... ] ) |
  CHECK ( expression ) |
  FOREIGN KEY ( nom_colonne [, ...
  ] ) REFERENCES table_reference [ (
  colonne_reference [, ... ] ) ] }
```

Exemples

CREATE TABLE titles

```
(
  title_id character varying(6) NOT NULL,
  title character varying(80) NOT NULL,
  "type" character(12) NOT NULL,
  pub_id character(4),
  price numeric(8,2),
  advance numeric(8,2),
  total_sales integer,
  notes character varying(200),
  pubdate timestamp NOT NULL DEFAULT now(),
  contract bit(1) NOT NULL,
  CONSTRAINT titles_pkey PRIMARY KEY (title_id),
  CONSTRAINT titles_pub_id_fkey FOREIGN KEY (pub_id)
    REFERENCES publishers (pub_id)
)
```

CREATE TABLE titleauthor

```
(
  au_id character varying(11) NOT NULL,
  title_id character varying(6) NOT NULL,
  au_ord smallint,
  royaltyper integer,
```

```

CONSTRAINT titleauthor_pkey PRIMARY KEY (au_id, title_id),
CONSTRAINT titleauthor_au_id_fkey FOREIGN KEY (au_id)
    REFERENCES authors (au_id),
CONSTRAINT titleauthor_title_id_fkey FOREIGN KEY (title_id)
    REFERENCES titles (title_id)
)

```

Exercices

1. Ecrivez la commande permettant de créer la table salesdetail
2. Ecrivez la commande permettant de créer la table authors
3. Ecrivez la commande permettant de créer la table stores

3.a.iii. DROP TABLE

```
DROP TABLE nom
```

3.a.iv. ALTER TABLE

```

ALTER TABLE nom [ * ]
    action [, ... ]
ALTER TABLE nom [ * ]
    RENAME [ COLUMN ] colonne TO nouvelle_colonne
ALTER TABLE nom
    RENAME TO nouveau_nom

```

- action peut être :


```

      ADD [ COLUMN ] colonne type [ contrainte_colonne [ ... ] ]
      DROP [ COLUMN ] colonne
      ALTER [ COLUMN ] colonne [ SET DATA ] TYPE type [ USING expression ]
      ALTER [ COLUMN ] colonne SET DEFAULT expression
      ALTER [ COLUMN ] colonne DROP DEFAULT
      ALTER [ COLUMN ] colonne { SET | DROP } NOT NULL
      ADD contrainte_table
      DROP CONSTRAINT nom_contrainte
      
```

Exemples

Ajouter une colonne de type varchar à une table :

```

ALTER TABLE publishers
ADD COLUMN continent varchar(30);

```

Changer le type de la colonne :

```

ALTER TABLE publishers
ALTER COLUMN continent
TYPE varchar(20);

```

Supprimer la colonne :

```

ALTER TABLE publishers
DROP COLUMN continent;

```

Exercices

1. Ajouter un ISBN à la table titles, de type varchar(15).
2. Modifier le champ ISBN pour qu'il soit unique.
3. Finalement supprimer ce champ.

3.b. Normalisation

Maintenant que l'on est capable de créer une table, il faut décider où placer les données. Il faut donc trouver une combinaison de tables possédant des relations entre-elles et permettant d'exploiter les données. Ceci s'appelle un schéma.

Pour tout jeu de données, il existe une multitude de schémas possibles permettant de les organiser. Cependant de part la nature même du modèle relationnel, certains schémas s'avèrent meilleurs que d'autres. Par exemple certains schémas évitent de dupliquer des données, enlevant ainsi un risque d'incohérence (la même donnée qui devrait être dupliquée à l'identique finit par avoir des valeurs différentes). On dit qu'un schéma est normalisé à une certaine forme lorsqu'il respecte les critères de cette forme. La normalisation est l'acte consistant à transformer un schéma pour atteindre une forme normalisée. Dans le cadre de ce cours nous verrons uniquement les trois premiers niveaux.

1FN - première forme normale :

Relation dont tous les attributs :

- contiennent une valeur atomique (les valeurs ne peuvent pas être divisées en plusieurs sous-valeurs dépendant également individuellement de la clé primaire)
- contiennent des valeurs non répétitives (le cas contraire consiste à mettre une liste dans un seul attribut).
- sont constants dans le temps (utiliser par exemple la date de naissance plutôt que l'âge).

Le non respect de deux premières conditions de la 1FN rend la recherche parmi les données plus lente parce qu'il faut analyser le contenu des attributs. La troisième condition quant à elle évite qu'on doive régulièrement mettre à jour les données.

2FN - deuxième forme normale

Respecte la deuxième forme normale, la relation respectant la première forme normale et dont :

- Tous les attributs non-clés sont totalement dépendants fonctionnellement de la totalité de la clé primaire.

Le non respect de la 2FN entraîne une redondance des données qui encombrant alors inutilement la mémoire et l'espace disque.

3FN - troisième forme normale

Respecte la troisième forme normale, la relation respectant la seconde forme normale et dont :

- Tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé.

Le non respect de la 3FN peut également entraîner une redondance des données.

3.c. Dénormalisation

La normalisation permet de garantir la cohérence des données et permet d'éviter des problèmes inutiles. On cherchera donc en général à normaliser son schéma complètement. Il peut cependant y avoir des exceptions pour lesquelles on dénormalisera le schéma :

- On pourrait avoir dans un schéma unique des tables appartenant logiquement à des schémas séparés. L'exemple typique est la table d'archivage ou d'audit qui duplique l'information se trouvant dans les autres tables en les mettant complètement à plat (un tuple contient toute l'information sans devoir effectuer une jointure avec une autre table). Cette table n'est évidemment pas du tout normalisée de par sa nature même.
- La normalisation force à multiplier les tables, ce qui impose une pénalité à l'exécution. Pour résoudre les problèmes de performance uniquement, on accepte d'enlever la normalisation. Cependant il existe quand même une technique permettant de s'assurer de la consistance des données, confier TRIGGER au chapitre suivant.

3.c.i. Exemple

Dans la base de données pubs2, il y a un champ total_sales dans la table titles : ce champ ne dépend pas uniquement de la clé primaire de titles, il dépend aussi directement du contenu de la table salesdetail. Cependant c'est une information dont on a fréquemment besoin dans le cadre de l'application utilisant pubs2, et pour garder de bonnes performances ce champ est dénormalisé. Comme il est dénormalisé, il y a un risque qu'il ne soit pas synchronisé correctement par rapport à la valeur réelle qu'il doit prendre.

3.c.ii. Exercice

1. Créer une table archivage qui contiendra les données des ventes effectuées il y a plus de 6 mois. Inutile d'y placer des données pour le moment. Cette table contiendra d'une manière dénormalisée toutes les informations concernant une vente.

3.d. Ajout/Suppression/Modification de tuples

3.d.i. INSERT INTO

```
INSERT INTO table [ ( colonne [, ...] ) ]
{ DEFAULT VALUES
| VALUES ( { expression | DEFAULT } [, ...] ) [, ...]
| RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] }
```

- INSERT insère de nouvelles lignes dans une table.

L'ordre des noms des colonnes n'a pas d'importance. Si aucune liste de noms de colonnes n'est donnée, toutes les colonnes de la table sont utilisées dans l'ordre de leur déclaration (les N premiers noms de colonnes si seules N valeurs de colonnes sont fournies dans la clause VALUES ou dans la requête). Les valeurs fournies par la clause VALUES sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Chaque colonne absente de la liste, implicite ou explicite, des colonnes se voit attribuer sa valeur par défaut, s'il y en a une, ou NULL dans le cas contraire.

Un transtypage automatique est entrepris lorsque l'expression d'une colonne ne correspond pas au type de donnée déclaré.

La clause RETURNING optionnelle fait que INSERT calcule et renvoie les valeurs basées sur chaque ligne en cours d'insertion. C'est principalement utile pour obtenir les valeurs qui ont été fournies par défaut, comme un numéro de séquence (confer 0). Néanmoins, toute expression utilisant les colonnes de la table est autorisée. La syntaxe de la liste RETURNING est identique à celle de la commande SELECT.

3.d.ii. UPDATE ...SET ...WHERE ...

```
UPDATE table
SET { colonne = { expression | DEFAULT } |
    ( colonne [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
[ FROM liste_from ]
[ WHERE condition ]
[ RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] ]
```

- UPDATE modifie les valeurs des colonnes spécifiées pour toutes les lignes qui satisfont la condition. Seules les colonnes à modifier doivent être mentionnées dans la clause SET ; les autres colonnes conservent leur valeur.

La clause RETURNING optionnelle fait que UPDATE calcule et renvoie les valeurs basées sur chaque ligne en cours de mise à jour. Toute expression utilisant les colonnes de la table et/ou les colonnes d'autres tables mentionnées dans FROM peut être calculée. La syntaxe de la liste RETURNING est identique à celle de la commande SELECT.

3.d.iii. DELETE FROM ...

```
DELETE FROM table
[ WHERE condition ]
[ RETURNING * | expression_sortie [ [ AS ] output_name ] [, ...] ]
```

- DELETE supprime de la table spécifiée les lignes qui satisfont la clause WHERE. Si la clause WHERE est absente, toutes les lignes de la table sont supprimées. Le résultat est une table valide, mais vide.

La clause RETURNING optionnelle fait que DELETE calcule et renvoie les valeurs basées sur chaque ligne en cours de suppression. La syntaxe de la liste RETURNING est identique à celle de la commande SELECT.

Exemples

Ajouter l'auteur Jimmy Carter, d'adresse inconnue et d'identifiant égal à 2895.

```
INSERT INTO Authors (au_lname, au_fname, au_id )
VALUES ( 'Carter', 'Jimmy', 2895 );
```

Créer une nouvelle table avec les numéros et noms des auteurs californiens uniquement.

```
CREATE TABLE Auteurs_Californiens
( au_id          INTEGER NOT NULL
  CONSTRAINT cle_prim_au_calif PRIMARY KEY,
  Au_lname       VARCHAR(30)      );

INSERT INTO Auteurs_Californiens
SELECT au_id, au_lname
FROM Authors
WHERE state = 'CA' ;
```

Supprimer les auteurs s'appellant "Ringer".

```
DELETE FROM Authors
WHERE au_lname = 'Ringer' ;
```

Supprimer tous les auteurs.

```
DELETE FROM Authors ;
```

Remarquons que cette instruction ne peut s'effectuer sans problème que si la table TitleAuthors est elle-même vide (ou absente) : en effet, la clé étrangère "au_id" de cette table impose la présence de tuples ayant même identifiant d'auteur dans la table Authors ! Une remarque analogue aurait pu être énoncée pour l'exemple précédent : les auteurs s'appellant Ringer ne peuvent avoir écrit aucun livre.

Augmenter de 5% le prix de tous les livres de psychologie.

```
UPDATE Titles
SET price = price* 1.05
WHERE type = 'psychology' ;
```

Exercices

1. Supprimez tous les auteurs qui n'ont écrit aucun livre.
2. Mettez à jour l'attribut "total_sales" de la table "Titles" à partir des valeurs de "qty" de la table "Salesdetail".
3. Supprimez les ventes de livres dont la date de vente est antérieure à la date d'édition.
4. Videz toutes les tables de la DB. Attention à l'ordre de ces suppressions : les contraintes d'intégrité doivent être respectées à tout moment!
5. Certains livres peuvent être regroupés en "Collections". Ajoutez une table des "Collections", avec un identifiant, un libellé et un identifiant d'éditeur; ajoutez un attribut "identifiant de Collection" à la table des livres. Créez enfin deux collections et quelques livres qui en font partie.
6. Supprimez toutes les ventes (et les détails de vente associés) dont la date est antérieure au 1/1/1988.

3.e. Séquences

```
CREATE SEQUENCE nom [ INCREMENT [ BY ] incrément ]
  [ MINVALUE valeurmin | NO MINVALUE ]
  [ MAXVALUE valeurmax | NO MAXVALUE ]
  [ START [ WITH ] début ]
```

- `CREATE SEQUENCE` crée un nouveau générateur de séquence de nombres. Cela implique la création et l'initialisation d'une nouvelle table à une seule ligne nommée `nom`.

La séquence est surtout utile pour laisser la base de données générer elle-même des identifiants uniques pour les tuples. Pour réaliser cela, il faut créer une séquence d'un certain nom (par exemple 'ABC') et dire que la valeur par défaut de la colonne est `nextval('ABC')`.

En PostgreSQL on utilisera donc une séquence pour générer des clefs primaires automatiquement¹. Dans ce cas lors de l'insertion d'un nouveau tuple on ne spécifie pas de valeur pour la clef primaire. Par contre on pourra vouloir directement récupérer sa valeur via la partie `RETURNING` de l'`INSERT`.

3.e.i. Exemple

```
CREATE TABLE "ExempleAutoPk"
(
  id integer NOT NULL DEFAULT nextval('seq_pk') PRIMARY KEY,
  "données" character varying(100),
)
```

```
CREATE SEQUENCE seq_pk
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
```

```
INSERT INTO "ExempleAutoPk"
  VALUES (DEFAULT, 'Donatien')
  RETURNING id;
```

Retournera 1.

```
INSERT INTO "ExempleAutoPk"
  VALUES (DEFAULT, 'Brigitte')
  RETURNING id;
```

Retournera 2 et ainsi de suite.

3.e.i. Exercice

1. Gérer `ord_num` de la table `sales` via un numéro de séquence : créez une séquence, et utilisez là comme défaut de cette colonne.

¹ D'autres base de données utilisent d'autres approches pour générer des clefs primaires automatiquement. On parle parfois d'AUTO-INCREMENT ou d'IDENTITY

3.f. Serial

Une séquence est implicitement créée lorsque l'on déclare la création d'une table avec une colonne de type SERIAL. La colonne se transformera en type integer mais sa valeur par défaut suivra l'ordre d'une séquence.

Exemple :

```
CREATE TABLE nom_de_table (  
    nom_de_colonne SERIAL  
);
```

Cette dernière instruction est équivalente à :

```
CREATE SEQUENCE nom_de_table_nom_de_colonne_seq;  
CREATE TABLE nom_de_table (  
    nom_de_colonne integer DEFAULT nextval('nom_de_table_nom_de_colonne_seq') NOT  
    NULL  
);
```

Dans la plupart des cas, vous voudrez aussi ajouter une contrainte PRIMARY KEY pour préciser que cet attribut est la clé primaire, mais ce n'est pas automatique.

4. Gestion d'une base de données

A ce stade du syllabus, nous sommes capables de créer les tables d'une base de données, d'y mettre des données et de l'interroger. Cela ne couvre cependant pas tous les besoins d'une base de données en situation réelle. Ce chapitre va aborder des besoins fréquents.

4.a. SQL Procédural

Au-delà de fournir des instructions SQL manipulant le schéma et les tuples d'une base de donnée, on peut aussi vouloir donner plus d'intelligence au serveur.

- Comme le serveur est le dernier intervenant avant le stockage physique des données, c'est un bon candidat pour valider la cohérence des données avant d'écrire n'importe quoi. Ceci est d'ailleurs déjà l'intention des contraintes d'intégrité. Mais au-delà de ces contraintes, il existera toujours des cas particuliers qu'elles ne peuvent pas valider. Par exemple on pourrait vouloir vérifier qu'un numéro de carte de banque respecte la validation numérique qu'elle est sensée avoir².
- Le serveur est aussi un bon candidat pour gérer automatiquement la cohérence des données dénormalisées. Il pourra s'assurer que ces données seront mises à jour correctement automatiquement dès que nécessaire. Par exemple dans pubs2 le champ total_sales de la table titles correspond à un calcul sur les ventes de ce livre. On pourra configurer le serveur pour effectuer la mise à jour de ce champ dès qu'une nouvelle vente est insérée.
- On pourra aussi vouloir abstraire de la complexité sous-jacente des données en fournissant des fonctions de haut niveau. Par exemple à la place de laisser un client manipuler différentes tables pour réaliser une opération complexe, le serveur fournira une fonction qui se charge de réaliser l'intégralité de la tâche. Si le schéma de la base de données change, c'est uniquement les fonctions du serveur qui devront être mises à jour, le client pourra rester inchangé.

Il faut donc un langage de programmation complet³ pour réaliser toutes cette intelligence. Les systèmes de base de données modernes (dont fait partie PostgreSQL) permettent même de choisir le langage de programmation que l'on souhaite. Historiquement on utilisait des langages procéduraux qui sont des extensions du SQL de base. Comme ces langages datent des années 70, les concepts présents dans les langages modernes n'en font pas partie. En particulier, il n'y a pas de notion d'objets. Dans le cadre de ce cours nous utiliseront PL/pgSQL. Cette section introduit brièvement ce langage, veuillez vous référer à la documentation en ligne de PostgreSQL pour plus de détails (<http://docs.postgresqlfr.org/>).

² Les deux derniers chiffres sont le modulo 97 du restant

³ Au sens de Turing : <http://fr.wikipedia.org/wiki/Turing-complet>

4.a.i. CREATE FUNCTION

```
CREATE FUNCTION nomFonction(type1, type2,..., typeX) RETURNS typeOut AS $$
    corpsFonction
$$ LANGUAGE plpgsql;
```

Définit une fonction qui prends X paramètres qui sont de type type1, type2, ..., typeX et qui retourne une valeur de type typeOut. corpsFonction contient la définition de la fonction, voir ci-dessous. Une fonction ainsi créée fait partie intégrante de la base de données, c'est pour quoi on l'appelle parfois « procédure stockée ». La commande DROP FUNCTION nomFonction permet d'effacer la fonction.

Attention les délimiteurs \$\$ et l'instruction LANGUAGE plpgsql font partie intégrante de la définition de la fonction.

Une fois définie, une fonction se comporte comme une expression. Ainsi on utilisera la commande SELECT nomFonction(param1, param2, ..., paramX) pour afficher le résultat de son exécution.

4.a.ii. DECLARE

```
CREATE FUNCTION nomFonction(type1, type2,..., typeX) RETURNS typeOut AS $$
DECLARE
    nomParam1 ALIAS FOR $1 ;
    nomParam2 ALIAS FOR $2 ;
    ...
    nomParamX ALIAS FOR $X ;
    nomVar1 typeVar1;
    nomVar2 typeVar2;
    ...
    nomVarY typeVarY;
BEGIN
    corpsDeclare ;
END ;
$$ LANGUAGE plpgsql;
```

DECLARE définit les variables internes à la fonction. ALIAS FOR permet de donner un nom explicite à un paramètre particulier. Pour des variables locales, il faut d'abord donner leur nom suivi de leur type.

4.a.iii. Affectation

```
variable := expression ;
```

4.a.iv. Commentaire

```
-- tout ce qui suit -- est ignoré jusqu'à la fin de la ligne
```

4.a.v. Structures de contrôle

```
RETURN expression ;
```

Termine l'exécution de la fonction en renvoyant la valeur calculée par expression.

```
IF ... THEN ... END IF ;
```

```
IF ... THEN ... ELSE ... END IF ;
```

```
IF ... THEN ... ELIF ... THEN ... ELSE ... END IF ;
```

```
FOR record IN instructionSelect LOOP ... END LOOP ;
```

record doit être une variable de type RECORD. L'instruction FOR fera un parcours successif des tuples retournées par l'instructionSelect. On pourra accéder aux différents champs du tuple en cours par record.nomChamp.

Exemple

```
CREATE FUNCTION compteSalesDetailQty() RETURNS INTEGER AS $$
DECLARE
```

```

        i integer := 0;
        record RECORD;
    BEGIN
        FOR record IN SELECT * FROM Salesdetail LOOP
            i := i + record.qty;
        END LOOP;
        RETURN i;
    END;
$$ LANGUAGE plpgsql;

```

```
SELECT compteSalesDetailQty();
```

Ceci est équivalent à :

```
SELECT SUM(qty) from Salesdetail;
```

Remarque importante : il n’y a aucune valeur ajoutée à implémenter soi-même ce qui devrait en fait être une requête SQL. Le code ne sera jamais aussi performant que la requête équivalente. Cela prend toujours plus de temps d’écrire une implémentation plutôt que d’écrire la requête, et le risque d’erreur est plus élevé. Dans le cadre de ce cours, ceci est donc considéré comme une faute et sanctionné comme tel.

4.a.vi. Exceptions

```

CREATE FUNCTION nomFonction(type1, type2,..., typeX) RETURNS typeOut AS $$
DECLARE
    ...
BEGIN
    corpsDeclare ;
EXCEPTION
WHEN condition [ OR condition ... ] THEN
    instructions_gestion_erreurs
[ WHEN condition [ OR condition ... ] THEN
    instructions_gestion_erreurs
... ]
END ;

```

Si une exception est levée pendant l’exécution de corpsDeclare, la partie EXCEPTION l’attrapera. Pour lever une exception manuellement, il faut utiliser l’instruction RAISE EXCEPTION nom_exception (confer les annexes ou la documentation de PostgreSQL pour la liste des nom_exception possibles et leur signification).

Exemple

```

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- ignore l'erreur
END;

```

4.a.vii. **Exemple**

Ecrire une fonction qui effectue une vente, en créant tous les tuples nécessaires dans toutes les tables.

```
CREATE FUNCTION sell(varchar(6), character(4), varchar(20), smallint)
  RETURNS BOOLEAN AS $$
DECLARE
  v_title_id ALIAS FOR $1 ;
  v_stor_id ALIAS FOR $2 ;
  v_ord_num ALIAS FOR $3 ;
  v_qty ALIAS FOR $4 ;
  price integer;
BEGIN
  SELECT price FROM titles WHERE title_id = v_title_id INTO price;
  IF price IS NULL THEN
    RETURN false;
  ELSE
    INSERT INTO sales
      VALUES (v_stor_id, v_ord_num) ;
    INSERT INTO salesdetail
      VALUES (v_stor_id, v_ord_num, v_title_id, v_qty) ;
    UPDATE titles
      SET total_sales =
        ((SELECT total_sales
          FROM titles WHERE title_id=v_title_id)
        + v_qty)
      WHERE title_id=v_title_id ;
    RETURN true;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

4.a.viii. **Procédure stockée renvoyant un tableau**

Une procédure stockée peut renvoyer un tableau. Elle s'utilise alors comme une table dans la commande SELECT :

```
CREATE OR REPLACE FUNCTION listeAuthorsLivres() RETURNS SETOF RECORD AS $$
DECLARE
  sep VARCHAR;
  texte VARCHAR;
  sortie RECORD;
  author RECORD;
  title RECORD;
BEGIN
  FOR author IN SELECT * FROM authors LOOP
    texte:='';
    sep:='';
    FOR title IN SELECT * FROM titles t, titleauthor ta WHERE
      t.title_id=ta.title_id AND ta.au_id=author.au_id LOOP
      texte:=texte || sep || title.title;
      sep:=', ';
    END LOOP;
    SELECT author.au_fname, author.au_lname, texte INTO sortie;
    RETURN NEXT sortie;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

Notez le type de retour : **SETOF** signifie plusieurs lignes tandis que **RECORD** signifie plusieurs colonnes. L'affectation à une variable de type **RECORD** se fait par la commande `SELECT valeur1, ... valeurX INTO record`. L'ajout d'une ligne en retour de l'appel de la procédure stockée se fait par la commande `RETURN NEXT`. La commande `RETURN` simple se contentant alors de terminer l'exécution

de la procédure stockée. L'appel de la procédure ci-dessous se fait par un `SELECT`, mais il faut préciser la structure des colonnes ainsi que leurs noms :

```
SELECT * FROM listeAuthorsLivres() t(fname VARCHAR(20), lname VARCHAR(40),
titles VARCHAR);
```

Comme il n'est pas très pratique d'embarquer la définition des colonnes dans le `SELECT`, il est possible d'utiliser un type personnalisé pour que ceci se fasse au niveau de la procédure stockée. Dans ce cas, l'appel à la procédure stockée devient identique au `SELECT` habituel.

```
CREATE TYPE listesAuthorsLivresReturn
AS (fname VARCHAR(20), lname VARCHAR(40), titles VARCHAR);

CREATE OR REPLACE FUNCTION listeAuthorsLivres() RETURNS SETOF
listesAuthorsLivresReturn AS $$
DECLARE
    sep VARCHAR;
    texte VARCHAR;
    sortie RECORD;
    author RECORD;
    title RECORD;
BEGIN
    FOR author IN SELECT * FROM authors LOOP
        texte:='';
        sep:='';
        FOR title IN SELECT * FROM titles t, titleauthor ta WHERE
t.title_id=ta.title_id AND ta.au_id=author.au_id LOOP
            texte:=texte || sep || title.title;
            sep:=', ';
        END LOOP;
        SELECT author.au_fname, author.au_lname, texte INTO sortie;
        RETURN NEXT sortie;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';

SELECT * FROM listeAuthorsLivres();
```

4.b. Automatisation : TRIGGER

Revenons à la problématique de la dénormalisation. Pour fixer les idées, nous regardons le problème particulier du champ `total_sales` de la table `titles` qui est en fait la somme de toutes les ventes de ce livre. Ce champ n'est pas en 3^{ème} forme normale : il dépend en effet d'autre chose qu'uniquement sa clef primaire puisqu'il dépend du contenu de la table `Salesdetail`.

Sa présence est justifiée dans la base de données pour des raisons de performance : si cette information est fréquemment demandée, il sera beaucoup plus rapide qu'elle soit précalculée dans la base de données plutôt que de devoir refaire ce calcul à chaque fois.

Mais il y a donc un risque de désynchronisation dès qu'il se passe quelque chose avec une vente de ce livre (ajout, suppression ou modification). Les bases de données offrent un mécanisme pour être sûr de garder cette synchronisation : les triggers. Un trigger est placé sur une table pour réagir en cas d'ajout, de suppression ou de modification. Dès qu'un de ces événements arrive, le trigger exécute une procédure stockée qui met à jour les données devant rester synchronisées.

4.b.i. Cas typique d'utilisation

```
CREATE TRIGGER nom AFTER { INSERT | UPDATE | DELETE }
    ON table FOR EACH ROW EXECUTE PROCEDURE nomfonc
```

La procédure `nomfonc` doit être préalablement créée grâce à la commande `CREATE FUNCTION`. Elle ne doit recevoir aucun paramètre mais retourner obligatoirement un type `TRIGGER`. Cependant la valeur de retour de ce paramètre sera ignorée avec la commande `CREATE TRIGGER` ci-dessus. La procédure pourra donc se contenter de `RETURN NULL`.

A chaque insertion (`INSERT`), mise à jour (`UPDATE`) ou effacement (`DELETE`), la procédure `nomfonc` sera appelée pour chaque tuple inséré, modifié ou supprimé. De plus deux variables seront automatiquement disponibles lors de son exécution :

- `OLD` est un `RECORD` contenant les anciennes valeurs du tuple modifié pour un `UPDATE`, effacé pour un `DELETE` ou `NULL` un `INSERT`.
- `NEW` est un `RECORD` contenant la nouvelle valeur du tuple dans la `TABLE` pour un `INSERT` ou un `UPDATE`, ou `NULL` pour un `DELETE`.

Exemple

Cet exemple crée un trigger pour mettre le champ `total_sales` de la table `titles` automatiquement à jour lors de l'ajout d'un tuple dans `sales_detail` :

```
CREATE OR REPLACE FUNCTION total_sales() RETURNS TRIGGER AS $$
    DECLARE
        total INTEGER;
    BEGIN
        SELECT SUM(sd.qty) FROM salesdetail sd WHERE sd.title_id=NEW.title_id INTO
        total;
        UPDATE titles SET total_sales = total WHERE title_id=NEW.title_id;
        RETURN NULL;
    END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER sales_detail_trigger AFTER INSERT ON SALESDETAIL FOR EACH ROW
EXECUTE PROCEDURE total_sales();
```

4.b.ii. Référence complète de CREATE TRIGGER

```
CREATE TRIGGER nom { BEFORE | AFTER } { evenement [ OR ... ] }
ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE nomfonc ( arguments )
```

- CREATE TRIGGER crée un nouveau déclencheur. Le déclencheur est associé à la table spécifiée et exécute la fonction nomfonc lorsqu'un de ces trois événements survient : INSERT, UPDATE ou DELETE.

L'appel du déclencheur peut avoir lieu avant que l'opération ne soit tentée sur une ligne (avant la vérification des contraintes et la tentative d'INSERT, UPDATE ou DELETE) ou une fois que l'opération est terminée (après la vérification des contraintes et la fin de la commande INSERT, UPDATE ou DELETE). Si le déclencheur est lancé avant l'événement, le déclencheur peut ignorer l'opération sur la ligne courante ou modifier la ligne en cours d'insertion (uniquement pour les opérations INSERT et UPDATE). Si le déclencheur est activé après l'événement, toute modification, dont la dernière insertion, mise à jour ou suppression, est « visible » par le déclencheur.

Un déclencheur marqué FOR EACH ROW est appelé pour chaque ligne que l'opération modifie. Par exemple, un DELETE affectant dix lignes entraîne dix appels distincts de tout déclencheur ON DELETE sur la relation cible, une fois par ligne supprimée. Au contraire, un déclencheur marqué FOR EACH STATEMENT ne s'exécute qu'une fois pour une opération donnée, quelque soit le nombre de lignes modifiées (en particulier, une opération qui ne modifie aucune ligne résulte toujours en l'exécution des déclencheurs FOR EACH STATEMENT applicables).

Si plusieurs déclencheurs du même genre sont définis pour le même événement, ils sont déclenchés suivant l'ordre alphabétique de leur nom.

SELECT ne modifie aucune ligne ; la création de déclencheurs sur SELECT n'est donc pas possible.

nomfonc est le nom d'une fonction qui a été créée préalablement. Cette fonction n'a aucun paramètre et doit retourner un type TRIGGER. Lorsque la fonction est appelée avant l'événement **et** tuple par tuple (BEFORE et FOR EACH ROW), elle a l'occasion d'annuler l'opération ou de modifier le tuple (ce dernier cas n'est possible que pour INSERT et UPDATE).

- Pour modifier le tuple inséré ou modifié, il faut retourner une variable de type TRIGGER contenant les nouvelles données à utiliser.
- Pour laisser l'opération s'exécuter normalement, il faut soit retourner NEW pour les événements INSERT et UPDATE, soit retourner OLD pour l'événement DELETE.
- Pour annuler l'opération, il faut retourner NULL. **ATTENTION CECI EST GÉNÉRALEMENT UNE ERREUR :**
 - Cette technique est dangereuse car par exemple un INSERT peut sembler avoir réussi alors que le trigger l'a silencieusement annulé. Lorsque ce problème surgit, il est excessivement difficile d'en trouver l'origine et donc de la corriger.
 - Nous verrons plus tard la mécanique permettant d'annuler une opération correctement (confer 5.b page 64).
 - Il est donc interdit dans le cadre de ce cours d'utiliser un trigger ON BEFORE qui retourne NULL.

Dans le cas d'un TRIGGER appelé après l'opération (AFTER) ou bien d'un TRIGGER de niveau instruction (FOR EACH STATEMENT), le retour de la fonction est ignoré et peut aussi bien être NULL. On peut cependant toujours annuler complètement l'opération en envoyant une erreur.

4.b.iii. Procédure trigger en PL/pgSQL

Une procédure trigger est créée grâce à la commande CREATE FUNCTION utilisée comme fonction sans arguments ayant un type de retour trigger. Notez que la fonction doit être déclarée avec aucun

argument même si elle s'attend à recevoir les arguments spécifiés dans `CREATE TRIGGER` -- les arguments trigger sont passés via `TG_ARGV`, comme décrit plus loin.

Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Ce sont :

- `NEW` : Type de données `RECORD` ; variable contenant la nouvelle ligne de base de données pour les opérations `INSERT/UPDATE` dans les triggers de niveau ligne. Cette variable est `NULL` dans un trigger de niveau instruction et pour les opérations `DELETE`.
- `OLD` : Type de données `RECORD` ; variable contenant l'ancienne ligne de base de données pour les opérations `UPDATE/DELETE` dans les triggers de niveau ligne. Cette variable est `NULL` dans les triggers de niveau instruction et pour les opérations `INSERT`.
- `TG_NAME` : Type de données `name` ; variable qui contient le nom du trigger réellement lancé.
- `TG_WHEN` : Type de données `text` ; une chaîne, soit `BEFORE` soit `AFTER`, selon la définition du trigger.
- `TG_LEVEL` : Type de données `text` ; une chaîne, soit `ROW` soit `STATEMENT`, selon la définition du trigger.
- `TG_OP` : Type de données `text` ; une chaîne, `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE` indiquant pour quelle opération le trigger a été lancé.
- `TG_RELID` : Type de données `oid` ; l'`ID` de l'objet de la table qui a causé le déclenchement du trigger.
- `TG_TABLE_NAME` : Type de données `name` ; le nom de la table qui a déclenché le trigger.
- `TG_NARGS` : Type de données `integer` ; le nombre d'arguments donnés à la procédure trigger dans l'instruction `CREATE TRIGGER`.
- `TG_ARGV[]` : Type de donnée `text` ; les arguments de l'instruction `CREATE TRIGGER`. L'index débute à 0. Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à `tg_nargs`) auront une valeur `NULL`.

Exemple

Une procédure d'audit par trigger en PL/pgSQL. Cet exemple de trigger nous assure que toute insertion, modification ou suppression d'une ligne dans la table `emp` est enregistrée dans la table `emp_audit`. L'heure et le nom de l'utilisateur sont conservés dans la ligne avec le type d'opération réalisé.

```
CREATE TABLE emp (
    nom_employe    text NOT NULL,
    salaire        integer
);

CREATE TABLE emp_audit(
    operation       char(1)    NOT NULL,
    tampon          timestamp NOT NULL,
    id_utilisateur  text       NOT NULL,
    nom_employe    text       NOT NULL,
    salaire        integer
);

CREATE FUNCTION audit_employe() RETURNS TRIGGER AS $ $
BEGIN
    --
    -- Ajoute une ligne dans emp_audit pour refléter l'opération réalisée
    -- sur emp,
```

```
-- utilise la variable spéciale TG_OP pour cette opération.
--
IF (TG_OP = 'DELETE') THEN
    INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
END IF;
RETURN NULL; -- le résultat est ignoré car il s'agit d'un trigger AFTER
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE audit_employe();
```

Exercice

1. Créer un trigger pour mettre le champ total_sales de la table titles automatiquement à jour.
2. Créer une table d'audit dénormalisant les données d'une vente. Créer un trigger pour insérer des tuples automatiquement dans cette table à chaque vente.

4.c. Utilisateurs

La base de données gère aussi les utilisateurs. Ainsi sur un serveur unique il est possible de créer plusieurs bases de données, et d'attribuer des droits d'accès et d'écriture différents en fonction des utilisateurs.

4.c.i. CREATE USER

```
CREATE USER nom [ [ WITH ] option [ ... ] ]
```

- où *option* peut être :
 SUPERUSER | NOSUPERUSER
 | CREATEDB | NOCREATEDB
 | CREATEROLE | NOCREATEROLE
 | CREATEUSER | NOCREATEUSER
 | INHERIT | NOINHERIT
 | LOGIN | NOLOGIN
 | CONNECTION LIMIT *limite_connexion*
 | [ENCRYPTED | UNENCRYPTED] PASSWORD '*motdepasse*'
 | VALID UNTIL '*dateheure*'
 | IN ROLE *nomrole* [, ...]
 | IN GROUP *nomrole* [, ...]
 | ROLE *nomrole* [, ...]
 | ADMIN *nomrole* [, ...]
 | USER *nomrole* [, ...]
 | SYSID *uid*

Comme on le constate, il est possible de spécifier toutes sortes de droits globaux au niveau du serveur. Typiquement on se contentera de créer un utilisateur comme ceci pour un utilisateur non administrateur de la base de données :

```
CREATE USER nom PASSWORD 'motdepasse'
```

4.d. GRANT / REVOKE

GRANT accorde des droits à un utilisateur, **REVOKE** en supprime. Il existe de nombreuses variantes de ces instructions, nous en présentons ici une seule, veuillez consulter la documentation en ligne pour plus d'informations :

```
GRANT { { SELECT | INSERT | UPDATE | DELETE }
        [, ...] | ALL [ PRIVILEGES ] }
ON [ TABLE ] nomtable [, ...]
TO { nomrole | PUBLIC } [, ...]
```

```
REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE }
  [, ...] | ALL [ PRIVILEGES ] }
ON [ TABLE ] nom_table [, ...]
FROM { nom_role | PUBLIC } [, ...]
```

4.d.i. Exemple

Georges est l'employé qui enregistre les ventes sur pubs2. Il n'a donc besoin d'insérer des tuples que dans les tables sales et salesdetail. Il doit par contre être capable de consulter le contenu de toutes les tables.

```
CREATE USER georges PASSWORD 'jaimelipl' ;
GRANT CONNECT ON DATABASE pubs2 TO georges;
GRANT SELECT ON sales, salesdetail, titles, publishers, store,
        titleauthors TO georges ;
GRANT INSERT ON TABLE sales, salesdetail TO georges ;
```

Si pubs2 utilisait un schéma, il aurait fallu également donner les droits d'utilisation du schéma :

```
GRANT USAGE ON SCHEMA pubs2 TO georges;
```

4.d.ii. Cas particulier des SERIAL

Lorsqu'on veut donner les droits d'insert sur une table contenant un serial, il faut également donner les droits sur les séquences implicitement créées :

```
GRANT USAGE, SELECT ON SEQUENCE id_seq TO georges;
```

4.e. Simplification des requêtes : VIEW

```
CREATE VIEW nom [ ( nom_colonne [, ...] ) ]
AS requête
```

```
DROP VIEW nom
```

Un autre inconvénient de la normalisation est la décomposition des données en de multiples tables. Lors d'un `SELECT` il faudra donc effectuer la jointure de toutes les tables permettant d'accéder aux données dont on a besoin. Dans les situations complexes, cela force les requêtes à être artificiellement compliquées et nuit fortement à leur lisibilité. SQL propose une solution très élégante à ce problème : les vues (`VIEW`). Une vue est une table virtuelle (car inexistante physiquement dans la base de données⁴) correspondant au résultat d'un `SELECT`. Cette table est elle-même utilisable en consultation pour tout autre `SELECT` au même titre qu'une table physique réelle.

Les vues permettent aussi de changer le schéma d'une base de données plus facilement. En effet, tout code qui utilise l'ancien schéma pourra rester inchangé si l'on crée des vues le respectant.

Finalement les vues permettent de gérer la visibilité des données en configurant la base de données pour empêcher l'accès direct aux tables mais en permettant celui aux vues juste nécessaires à l'utilisateur.

4.e.i. Exemples

```
CREATE VIEW LivresDePsychologiePasChers (numéro, titre, editeur, prix )
AS SELECT T.title_id, T.title, P.pub_name, T.price
   FROM titles T, Publishers P
  WHERE T.pub_id = P.pub_id
        AND T.type = 'psychology'
        AND T.price < 20 ;

SELECT titre, prix
FROM LivresDePsychologiePasChers
WHERE editeur = 'New Age Books' ;
```

Cette instruction est traduite par SQL en :

```
SELECT T.title, t.price
FROM titles T, Publishers P
WHERE T.pub_id=P.pub_id
      AND T.type = 'psychology'
      AND T.price <20
      AND P.pub_name = 'New Age Books' ;

CREATE VIEW TotalVenteParMagasin (stor_id, stor_name, total_qty, total_price)
AS SELECT St.stor_id, St.stor_name, SUM ( SD.qty ), SUM (SD.qty * T.price)
   FROM Stores St, Salesdetail SD, Titles T
  WHERE St.stor_id = SD.stor_id AND SD.title_id = T.title_id
 GROUP BY St.stor_id;

CREATE VIEW TotalVenteParMagasinBis
AS SELECT stor_id, total_qty
   FROM TotalVenteParMagasin
  WHERE total_qty > 5000 ;
```

⁴ Les données se trouvant physiquement dans les tables de la base de données

4.f. Performance : INDEX

Admettons que notre commerce devient aussi prolifique que celui d'Amazon, et que pubs2 soit rempli de millions de livres différents. Seulement les utilisateurs se plaignent des mauvaises performances de notre application et commencent à partir à la concurrence. Quand on observe de plus près ce qu'il se passe, on se rend compte que ces utilisateurs recherchent des livres principalement sur base de leur type (par exemple tous les livres de psychologie) et qu'avec le temps on a maintenant des milliers de type différents. A chacune de ces requêtes, notre serveur doit donc parcourir l'intégralité de la table titles afin de trouver ceux correspondant au type recherché.

Le même problème pourrait apparaître pour trouver un title_id particulier lors d'une jointure, cependant là on n'observe pas de ralentissement du serveur. La raison est qu'à chaque clef primaire le serveur crée automatiquement un index sur cette clef afin d'accélérer les jointures. Mais on garde la possibilité de créer des index supplémentaires si on en a besoin.

```
CREATE [ UNIQUE ] INDEX nom ON table [ USING méthode ]
( { colonne | ( expression ) } [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [,
...] )
```

- `CREATE INDEX` construit un index nom sur la table spécifiée. Les index sont principalement utilisés pour améliorer les performances de la base de données (bien qu'une utilisation inappropriée puisse produire l'effet inverse).

Les champs clé pour l'index sont spécifiés à l'aide de noms des colonnes ou par des expressions écrites entre parenthèses. Plusieurs champs peuvent être spécifiés si la méthode d'indexation supporte les index multi-colonnes.

Un champ d'index peut être une expression calculée à partir des valeurs d'une ou plusieurs colonnes de la ligne de table. Cette fonctionnalité peut être utilisée pour obtenir un accès rapide à des données obtenues par transformation des données basiques. Par exemple, un index calculé sur `upper(col)` autorise la clause `WHERE upper(col) = 'JIM'` à utiliser un index.

PostgreSQL™ fournit les méthodes d'indexation B-tree (NDT : arbres balancés), hash (NDT : hachage), GiST (NDT : arbres de recherche généralisés) et GIN.

- `UNIQUE` : Le système vérifie la présence de valeurs dupliquées dans la table à la création de l'index (si des données existent déjà) et à chaque fois qu'une donnée est ajoutée. Les tentatives d'insertion ou de mises à jour qui résultent en des entrées dupliquées engendrent une erreur.
- `nom` : Le nom de l'index à créer. On pourra supprimer l'index par la commande `DROP INDEX nom`.
- `table` : Le nom de la table à.
- `méthode` : Le nom de la méthode à utiliser pour l'index. Les choix sont `btree`, `hash`, `gist` et `gin`. La méthode par défaut est `btree`.
- `colonne` : Le nom d'une colonne de la table.
- `expression` : Une expression basée sur une ou plusieurs colonnes de la table. L'expression doit habituellement être écrite entre parenthèses, comme la syntaxe le précise. Néanmoins, les parenthèses peuvent être omises si l'expression a la forme d'un appel de fonction.
- `ASC` : Spécifie un ordre de tri ascendant (valeur par défaut).
- `DESC` : Spécifie un ordre de tri descendant.

- `NULLS FIRST` : Spécifie que les valeurs `NULL` sont présentées avant les valeurs non `NULL`. Ceci est la valeur par défaut quand `DESC` est indiqué.
- `NULLS LAST` : Spécifie que les valeurs `NULL` sont présentées après les valeurs non `NULL`. Ceci est la valeur par défaut quand `ASC` est indiqué.

4.f.i. **Exemple**

Créer un index sur le champ `type` de la table `titles`.

```
CREATE INDEX titles_type
ON titles
USING btree(type);
```

5. Concurrency

En situation réelle, les bases de données sont consultées et modifiées simultanément par un grand nombre d'utilisateurs. Par exemple un site web d'enchères devra répondre simultanément à toutes les requêtes de tous les utilisateurs qui le consultent à un instant donné. Il devra prendre en compte correctement une enchère par un utilisateur et ce même si un autre utilisateur essaie simultanément de faire la même chose. On appelle concurrence le fait qu'une donnée unique soit accédée ou modifiée simultanément par plusieurs instances (ici des personnes différentes). La concurrence introduit une complexité supplémentaire aux opérations que l'on effectue. Par exemple, si l'enchère permet un achat immédiat (au-delà d'un certain montant, l'objet est automatiquement gagné par l'enchéreur et celle-ci se conclut donc immédiatement), que se passe-t-il si deux utilisateurs effectuent cette opération simultanément ?

Utilisateur A	Utilisateur B
Sélectionne achat immédiat	Sélectionne achat immédiat
Le système vérifie que l'enchère est toujours ouverte	Le système vérifie que l'enchère est toujours ouverte
⇒ OUI	⇒ OUI
Le système clôture l'enchère et marque que l'acheteur est A	Le système clôture l'enchère et marque que l'acheteur est B
Le système notifie A qu'il a gagné l'enchère	Le système notifie B qu'il a gagné l'enchère
Le vendeur est notifié qu'il a vendu l'objet à A	Le vendeur est notifié qu'il a vendu l'objet à B

Les deux utilisateurs penseront avoir gagné l'enchère ! Comme il n'y a qu'un seul champ acheteur le système écrira donc d'abord A et puis B, mais cela aurait aussi bien pu être B et puis A. Dans les deux cas l'information de la première vente est écrasée et donc perdue. Le vendeur est notifié deux fois de la vente de son objet. Il doit choisir à qui envoyer son objet, mais ne sait pas qui est mis comme acheteur dans la base de données (A qui a écrasé B ou l'inverse).

Il va sans dire que cette situation n'est pas admissible et il faut donc une gestion explicite de la concurrence.

5.a. ACID

Les bases de données fournissent un mécanisme permettant de gérer la concurrence : les transactions. Une transaction est une séquence d'opérations qui fait passer une base de données d'un état A cohérent à un état B cohérent. Les transactions respectent les propriétés ACID : Atomique, Cohérent, Isolé et Durable.

- Atomique: l'ensemble des opérations sera réalisé complètement ou pas du tout. Une transaction termine soit par une opération de réussite (commit) ou par une opération d'échec (rollback). Lors d'un commit, toutes les opérations sont effectuées. Lors d'un rollback, aucune de ces opérations n'est effectué : c'est exactement comme si la transaction n'avait jamais eu lieu.
- Cohérent: le résultat de l'exécution d'une transaction doit être cohérent, même si chaque opération de la transaction exécuté séparément ne donne pas un résultat cohérent. Un résultat incohérent entraînera l'échec et l'annulation des opérations de la transaction.
- Isolé: lorsque deux transactions A et B sont exécutées en même temps, les modifications effectuées par A ne sont pas visibles par B tant que la transaction A n'est pas terminée et validée (commit).
- Durable: une transaction terminée ne peut pas être annulée ou recouverte. Lorsque deux transactions sont exécutées en même temps, le résultat de la première transaction ne pourra pas être recouvert par la deuxième. Toute tentative de recouvrement entraînera l'annulation des opérations de la transaction fautive.

Reprenons l'exemple de l'enchère, et plaçons l'opération d'achat immédiat au sein d'une transaction ACID.

- La transaction commence lorsqu'un utilisateur sélectionne l'option achat immédiat et se termine lorsque l'enchère est clôturée et les parties notifiées correctement. L'atomicité garantit que tout ceci se fait complètement, ou pas du tout.
- Lorsque l'enchère est clôturée, l'acheteur doit en être notifié et le vendeur doit être notifié de cet acheteur. La cohérence garantit que ces différentes opérations seront cohérentes entre elles.
- Admettons que l'acheteur A et B commencent leur transaction simultanément. L'isolation garantit que ce qui se passe dans la transaction A n'influencera pas ce qui se passe dans la transaction B. En d'autres mots, quand A marque l'enchère comme étant clôturée, B ne le voit pas du tout ! C'est seulement quand A termine sa transaction que cette modification devient visible pour B. Mais alors le système se rendra compte par la cohérence qu'il y a un problème de concurrence entre les deux transactions et annulera celle de B. De par l'atomicité, la transaction de B sera complètement annulée comme si elle n'avait jamais eu lieu.
- Une fois que A termine sa transaction (commit), alors cette enchère est marquée durablement dans la base de donnée. Par exemple il n'y aura plus jamais de cas où cette enchère sera encore considérée ouverte.

5.b. Transaction en SQL

- `START TRANSACTION` débute une transaction.
- `COMMIT` valide la transaction en cours
- `ROLLBACK` annule la transaction en cours

Attention un bloc de transaction ne correspond pas à un bloc d'exécution de l'application. En particulier il ne faut pas confondre ce comportement avec une méthode Java où le fait de `return` quitte immédiatement la méthode. La paire `START TRANSACTION ... COMMIT` ne forme donc pas un bloc d'exécution terminé immédiatement si l'on rencontre un `ROLLBACK` au milieu !

En réalité, les requêtes SQL se passent toujours dans le contexte d'une transaction. Ceci semble contradictoire avec tout ce que l'on a fait jusqu'ici : on n'a jamais utilisé `START TRANSACTION` et `COMMIT`, et pourtant les données ont manifestement été mises à jours et on a bien obtenu des réponses à nos requêtes. En réalité en dehors d'une transaction explicite, toutes les requêtes SQL sont implicitement exécutées dans leur propre transaction qui est directement `COMMIT`. C'est exactement comme si chacune de ces instructions était précédée par `START TRANSACTION` et suivie par `COMMIT`. On appelle cela l'auto-commit.

On utilisera une transaction explicite dans les situations suivantes :

- On a besoin d'effectuer plusieurs opérations successives, qui ont des dépendances entre elles. Le cas le plus fréquent est quand on doit mettre plusieurs tuples à jour dans plusieurs tables et que ces modifications doivent toutes être effectuées pour garder la cohérence globale. Un autre exemple est quand on effectue des vérifications sur le contenu de la base de données avant d'effectuer une mise à jour. Par exemple on vérifie qu'il reste un livre en stock avant de le vendre à un client. C'est l'atomicité de la transaction qui permet d'effectuer cela d'une manière cohérente.
- La création d'une transaction requiert du travail de la part du serveur. Si l'on exécute beaucoup d'instructions SQL (qui peuvent être indépendantes), on gagnera grandement en temps d'exécution si l'on exécute le tout dans une unique transaction explicite. Pour rappel, en l'absence de transaction explicite, l'auto-commit créera une transaction implicite pour chacune des instructions SQL, consommant inutilement les ressources du serveur.

5.b.i. Transactions en PL/pgSQL

N'oublions pas qu'il faut faire la distinction entre le pur SQL et le SQL procédural. Avec PostgreSQL, la gestion des transactions est différente entre les deux. En PL/pgSQL :

- Une procédure stockée est en effet toujours exécutée au sein d'une unique transaction. Si elle est appelée en dehors d'une transaction explicite, c'est le mécanisme d'auto-commit qui en crée une implicitement.
- `START TRANSACTION ... COMMIT` n'a donc pas d'effet en PL/pgSQL.
- Si la procédure stockée lance une exception, alors la transaction dans laquelle elle est exécutée effectue un `ROLLBACK`.
- Si la procédure stockée fini son exécution normalement, alors la transaction dans laquelle elle est exécutée continue normalement jusqu'à un éventuel `COMMIT` ou `ROLLBACK` futur.

- Si la procédure est exécutée en mode auto-commit et si elle termine normalement alors elle est directement suivie d'un `COMMIT`.

5.b.ii. Exemple

Voici une procédure stockée chargée de gérer complètement une vente. Notez l'usage de la transaction afin de garantir que les différentes étapes de la vente seront bien effectuées correctement.

```
CREATE FUNCTION sell(varchar(6), character(4), varchar(20), smallint)
    RETURNS BOOLEAN AS $$
DECLARE
    v_title_id ALIAS FOR $1 ;
    v_stor_id ALIAS FOR $2 ;
    v_ord_num ALIAS FOR $3 ;
    v_qty ALIAS FOR $4 ;
    price integer;
BEGIN
    price := (SELECT price FROM titles WHERE title_id = v_title_id) ;
    IF price IS NULL THEN
        RAISE no_data
    ELSE
        INSERT INTO sales
            VALUES (v_stor_id, v_ord_num) ;
        INSERT INTO salesdetail
            VALUES (v_stor_id, v_ord_num, v_title_id, v_qty) ;
        UPDATE titles
            SET total_sales =
                ((SELECT total_sales
                    FROM titles WHERE title_id=v_title_id)
                + v_qty)
            WHERE title_id=v_title_id ;
        RETURN true;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

5.b.iii. Transactions et trigger

Lorsqu'une opération résulte en l'exécution d'un (ou plusieurs) triggers, ceux-ci sont exécutés dans la même transaction que l'opération. En particulier si la procédure trigger jette une exception, c'est toute la transaction qui est annulée comprenant donc aussi l'opération. Ceci permet de programmer des contraintes d'intégrité plus intelligentes que ce qui est normalement permis : à l'insertion ou la modification le trigger valide les données et jette une exception si elles ne sont pas acceptables.

5.b.iv. Exercice

1. Créez une table commande. Cette table correspond à une commande groupée pour laquelle chaque tuple comporte :
 - Un identifiant de commande groupée unique.
 - Un identifiant de titre (`title_id`)
 - Une quantité (`qty`)
 - Un identifiant de magasin (`stor_id`)
 - Un identifiant de commande (`ord_num`)
2. Ecrivez une fonction qui prend un identifiant de commande groupée et exécute toutes les commandes qu'elle contient. Ceci doit s'effectuer dans une transaction.

3. Ecrivez une fonction d'archivage des ventes. Cette fonction transfère dans une table archive (que vous devez créer vous-même) toutes ventes qui ont plus d'un mois. Un tuple de la table archive représente toutes les données d'une vente mise à plat : chaque tuple contient l'intégralité des données d'une vente et il n'y a pas besoin d'effectuer une jointure pour récupérer une partie de l'information de la vente. Les ventes archivées disparaissent des tables habituelles. Ceci doit s'effectuer dans une transaction.

5.c. MVCC

PostgreSQL (ainsi que de nombreuses autres bases de données) réalise le mécanisme de transaction par la technique de Multiversion Concurrency Control (MVCC) :

- Chaque requête ne voit que les transactions terminées avant qu'elle n'ait commencé.
- A chaque transaction est associé un compteur unique (timestamp) permettant de les ordonner dans le temps.
- Chaque tuple est annoté par deux meta-informations : un compteur de transaction de création et un compteur de transaction d'expiration.
- Au démarrage d'une requête, PostgreSQL retient :
 - Le compteur de cette transaction
 - Tous les compteurs des transactions en cours d'exécution
- Dans une transaction couvrant plusieurs instructions SQL, les instructions précédentes de la transaction sont visibles au sein même de la transaction.
- Lorsqu'il y a insertion ou modification d'un tuple dans une transaction, le tuple original reste dans la table. Un tuple supplémentaire est ajouté, annoté en création par le compteur de la transaction.
- Lorsqu'il y a suppression d'un tuple dans une transaction, le tuple original est annoté en expiration par le compteur de la transaction.
- Les tuples visibles au sein d'une transaction doivent être annotés en création par un compteur de transaction tel que :
 - Cette transaction a été commitée.
 - Ce compteur est inférieur au compteur de la transaction en cours.*et*
Ce compteur ne faisait pas partie d'une autre transaction en cours au démarrage de celle-ci.
- Les tuples visibles au sein d'une transaction doivent aussi être annotés en expiration par un compteur de transaction tel que :
 - Il est vide ou annulé.
 - Ce compteur est supérieur au compteur de la transaction en cours.*ou*
Ce compteur faisait partie d'une autre transaction en cours au démarrage de celle-ci.

Ainsi la base de donnée est capable d'exécuter les transactions en parallèle plutôt que séquentiellement. Pour chaque transaction, à tout instant le système possède suffisamment d'information pour pouvoir inférer quels sont les tuples logiquement visibles au sein de cette transaction. A chaque requête, la base de donnée effectue un parcours séquentiel de la table et en fonction des annotations et des critères ci-dessus décide si le tuple est visible ou non.

Il existe un inconvénient à cette technique : physiquement le système n'est capable que d'ajouter de l'information, jamais de la supprimer. Et chaque modification ajoute un tuple supplémentaire plutôt que d'écraser l'ancien. Ce n'est donc qu'une question de temps avant que la base de données grandisse au-delà de la taille disque disponible. Il existe donc une commande permettant de nettoyer les tuples périmés afin de récupérer leur espace physique. C'est la commande `VACUUM` qui s'en charge. PostgreSQL est aussi capable de déterminer automatiquement quand cette commande devrait être utilisée et de le faire sans intervention extérieure.

La technique MVCC est affinée en différents niveaux de performances et de garanties. En général plus on demande des garanties sur la transaction, moins on a un niveau élevé de performance et inversement.


```
START TRANSACTION [ mode_transaction [, ...] ]
```

- où mode_transaction fait partie de :

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ  
                  | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY
```

Les caractéristiques de transaction disponibles sont le niveau d'isolation et le mode d'accès de la transaction (lecture/écriture ou lecture seule).

Le niveau d'isolation détermine les données que la transaction peut voir quand d'autres transactions fonctionnent concurrentiellement :

- o READ COMMITTED : Une instruction ne peut voir que les lignes validées avant qu'elle ne commence. C'est la valeur par défaut.
- o SERIALIZABLE : Toute instruction de la transaction en cours ne peut voir que les lignes validées avant que la première requête ou instruction de modification de données soit exécutée dans cette transaction.

Le standard SQL définit deux niveaux supplémentaires, READ UNCOMMITTED et REPEATABLE READ. Dans PostgreSQL™, READ UNCOMMITTED est traité comme READ COMMITTED alors que REPEATABLE READ est traité comme SERIALIZABLE.

La méthode d'accès de la transaction détermine si elle est en lecture/écriture ou en lecture seule. Lecture/écriture est la valeur par défaut. Quand une transaction est en lecture seule, les commandes SQL modifiant le contenu de la base de données interdites.

6. SQL embarqué

Nous sommes donc maintenant capables de créer une base de données, d'y insérer/modifier/supprimer des données, d'automatiser la mise à jour de certains champs, de créer des vues, de gérer la concurrence, etc. Il ne reste donc maintenant plus qu'à écrire l'application qui va avec ces données ! Alors que la base de données est installée sur un serveur joignable par Internet, l'application elle-même s'exécute généralement chez le client. Cette application se doit d'avoir une belle interface utilisateur, de gérer des aspects supplémentaires que juste les données, etc. C'est pourquoi on écrira cette application dans un langage de programmation adapté à cette tâche. Nous utiliserons Java ici. Quand le programme Java aura besoin d'interagir avec la base de données, on y embarquera les commandes SQL correspondantes. On appelle donc cela le SQL embarqué (embedded SQL en anglais).

6.a. JDBC

Le module Java gérant le SQL embarqué s'appelle JDBC et est inclus en standard avec Java. JDBC est en fait le portage d'ODBC qui est la technologie à l'origine du SQL embarqué. Cette technologie date du début des années 1990, et à l'époque ses créateurs comptaient encore à partir de 1, alors que tout bon informaticien moderne se doit de compter à partir de zéro⁵. Par héritage du passé, JDBC requiert donc aussi de compter à partir de 1 !

6.b. Driver

Le module JDBC requiert un driver pour fonctionner avec une base de données. Dans le cas de PostgreSQL, ce driver est disponible à l'adresse : <http://jdbc.postgresql.org/download.html>

Ce driver doit être placé dans le Build Path du projet Java qui l'utilise. Dans Eclipse, il faut placer le jar (par exemple postgresql-8.4-701.jdbc4.jar) dans un répertoire du projet, cliquer-droit dessus et sélectionner Build Path/Add to Build Path. Ceci permet à votre application d'accéder à ce driver pendant la compilation et son exécution.

En plus, il faut dire à l'application de charger ce driver en mémoire, ce qui se fait par cette commande :

```
Class.forName("org.postgresql.Driver");
```

6.c. Connection

Une fois le driver chargé, il faut établir une connexion avec la base de données :

```
String url =  
"jdbc:postgresql://postgresql.ipl.be/pubs2?user=public&password=public";  
Connection conn = DriverManager.getConnection(url);
```

L'URL est composée de :

- `jdbc:postgresql` : signale quel driver utiliser.
- `//postgresql.ipl.be` : URL du serveur, peut se terminer par `:port` pour préciser un numéro de port.
- `/pubs2` : nom de la base de données
- `?user=public` : nom de l'utilisateur à connecter

⁵ Cet avis n'engage que l'auteur de ce texte !

- `&password=public` : mot de passe

Notez que le format d'encodage des caractères spéciaux des URLs est d'application⁶. Vous trouverez d'autres méthodes statiques dans la classe `DriverManager` qui ne requièrent pas d'encoder le login et mot de passe.

Sur base d'une connexion, on pourra envoyer autant de commandes SQL qu'on le souhaite, et recevoir tout autant de résultats. Chaque connexion prend un certain temps à s'établir et occupe de la ressource auprès du client et du serveur. On veillera donc à n'ouvrir une connexion qu'en cas de réel besoin. En général, on aura deux stratégies viables :

- Pour une application mono-utilisateur, l'application ouvre une connexion au démarrage et la garde active pendant toute sa durée de vie. Toutes les opérations DB passent par cette connexion.
- Pour une application multi-utilisateur (typiquement une application Web), plusieurs connexions sont ouvertes et fermées suivant les besoins en cours. Ainsi, quand moins d'utilisateurs sont actifs, les connexions sont fermées. Quand plus d'utilisateurs sont actifs, des connexions sont ouvertes en plus. En général on utilise une librairie pour effectuer cette maintenance de connexions.

Pour fermer manuellement une connexion, il faut appeler la méthode `close` :

```
conn.close();
```

6.d. Statement

A partir d'une connexion, on peut créer un `Statement` : une instruction SQL.

```
try {
    Statement st=conn.createStatement();
    st.execute("DELETE FROM Store");
} catch (SQLException e) {
    e.printStackTrace();
}
```

La méthode `execute(String sql)` se charge d'exécuter cette instruction. Cette méthode déclare lancer l'exception `SQLException` que l'on est obligé d'attraper. En fait toutes les méthodes de JDBC qui interagissent avec le serveur sont susceptibles de lancer cette exception. On se retrouve donc assez rapidement à devoir écrire des blocs `try {} catch` en de multiples endroits, ce qui rend le code plutôt difficile à lire.

6.e. ResultSet

En plus d'envoyer des instructions SQL, on a besoin de récupérer le résultat obtenu. Ceci se fait grâce aux `ResultSet` :

```
ResultSet rs=null;
try {
    Statement st=conn.createStatement();
    try (ResultSet rs= st.executeQuery("SELECT au_fname, au_lname " +
                                     "FROM Authors")) {
```

⁶ <https://en.wikipedia.org/wiki/Percent-encoding>

```

        while (rs.next()) {
            String r=rs.getString(1) + " " + rs.getString(2);
            System.out.println(r);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

La méthode `executeQuery` d'un `Statement` est similaire à la méthode `execute` (elle exécute une commande SQL) mais en plus elle retourne un `ResultSet` permettant d'accéder au résultat du query.

Le `ResultSet` est un curseur sur les résultats, similaire à un `Iterator` Java. La méthode `next()` permet de déplacer ce curseur vers l'avant ; elle retourne `false` quand on est arrivé à la fin des tuples, `true` sinon. Il y a toute une série de getters permettant de récupérer les différents champs du tuple pointé par le curseur. Chacun de ces getters retourne un type Java particulier ; il faut choisir le getter qui a un sens par rapport au type du champ dans la base de données. Le paramètre des getters spécifie quel champ doit être retourné ; comme dit au début du chapitre, le premier champ est le numéro 1.

Un `ResultSet` doit être fermé pour relâcher les ressources JDBC le concernant. La bonne pratique consiste à utiliser un `try` d'allocation de ressource : la variable déclarée dans la parenthèse suivant le `try` sera automatiquement clôturée à la fin de ce dernier. Ainsi la base de données et le driver JDBC sont mis au courant que le curseur du `ResultSet` peut disparaître et éviter ainsi une fuite de mémoire.

6.f. PreparedStatement

On a fréquemment besoin de paramétrer les requêtes en fonction de données provenant de l'application elle-même, par exemple en fonction d'une entrée de l'utilisateur. Il serait alors tentant d'utiliser un `Statement` en construisant à l'exécution la chaîne de caractères spécifiant la requête SQL :

```

public void showNames(String state) {
    try {
        Statement st=conn.createStatement();
        try (ResultSet rs=st.executeQuery("SELECT au_fname, au_lname " +
            "FROM Authors " +
            "WHERE state='"+state+"'")) {
            while (rs.next()) {
                String r=rs.getString(1) + " " + rs.getString(2) ;
                System.out.println(r) ;
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

Cette manière de faire est en fait très dangereuse ! Imaginons que le paramètre `state` provient d'une interface utilisateur permettant à l'utilisateur d'y mettre tout ce qu'il veut. S'il rentre :

```
XX' ; DROP TABLE SalesDetail ; SELECT 'a', 'a
```

La query devient :

```
SELECT au_fname, au_lname FROM Authors WHERE state='XX' ; DROP TABLE
SalesDetail ; SELECT 'a', 'a' ;
```

Nous venons juste de perdre la table `SalesDetail`. La query quant à elle retourne le tuple ('a','a') et le programme continue de s'exécuter comme si de rien n'était.

Il ne faut surtout pas croire que c'est un cas hypothétique et tordu qui n'arrive pas en réalité : ceci s'appelle une attaque par injection de SQL et est probablement le numéro un des failles de sécurité des sites web !

Pour éviter ce genre de problème, il faut utiliser `PreparedStatement` à la place de `Statement` :

```
public void showNames(String state) {
    try {
        PreparedStatement st=
            conn.prepareStatement("SELECT au_fname, au_lname " +
                                "FROM Authors " +
                                "WHERE state= ?") ;

        st.setString(1, state);
        try (ResultSet rs=st.executeQuery()) {
            while (rs.next()) {
                String r=rs.getString(1) + " " + rs.getString(2) ;
                System.out.println(r) ;
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

La méthode `prepareStatement` prend une instruction SQL paramétrée par zéro, un ou plusieurs ?. Chaque ? représente une place logique à remplir plus tard. C'est le `PreparedStatement` qui se charge de ce remplissage, d'une manière qui est garantie correcte. Par exemple pour un champ textuel, il ne faudra pas placer d'apostrophes autours du ?, c'est le `prepareStatement` qui se chargera de le faire.

Un `PreparedStatement` ainsi créé peut être configuré à tout moment à l'aide de setters. Tout comme pour les `ResultSet`, les setters existent sous différentes formes portant sur des types Java différents qu'il faut choisir en accord avec le type du champ correspondant de la base de données. Le premier paramètre du setter est le numéro du ? qu'il faut remplir, en comptant de nouveau à partir de 1. Le second paramètre est la valeur à placer. Une fois que tous les ? ont été settés, `executeQuery()` retourne le `ResultSet` correspondant.

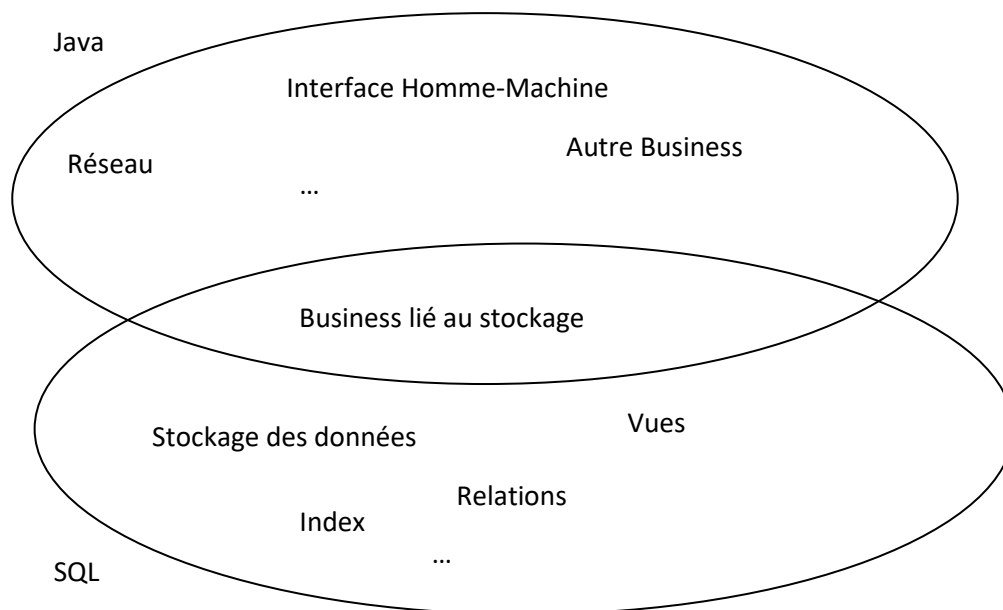
Avec cette technique on évite tous les problèmes d'injection de SQL. Ainsi dans l'exemple ci-dessus, le `PreparedStatement` recherchera juste un état dont le nom est bien toute la chaîne « XX' ; DROP TABLE SalesDetail ; SELECT 'a', 'a' ».

Un `PreparedStatement` peut être utilisé plusieurs fois dans des requêtes successives paramétrées indépendamment les unes des autres. Ceci permet de mieux factoriser son code et a aussi l'avantage d'être plus performant. En effet, chaque instruction SQL doit être traduite par la base de données. Avec un `PreparedStatement`, la base de données traduit l'instruction (incomplète) une et une seule fois. A l'exécution du `PreparedStatement`, la base de données n'aura qu'à compléter les parties manquantes de l'instruction pour pouvoir l'exécuter.

Remarque importante : Puisque les PreparedStatement sont plus efficaces et plus sûrs que les Statement simples, il faudra toujours les préférer. Dans le cadre de ce cours, l'utilisation d'un Statement est donc considérée comme une faute, et sanctionnée comme tel.

6.g. Intégration entre Java et une base de données

Comme nous l'avons vu, il est possible de stocker des procédures au niveau de la base de données et d'utiliser des triggers pour les appeler automatiquement. Alternativement, on pourrait réaliser une fonctionnalité similaire en Java : repérer les requêtes qui exigent des mises à jour supplémentaires et alors appeler la méthode Java qui effectue cette dernière. On a donc une frontière qui se superpose entre les deux mondes.



Pour la partie commune, il y a des raisons diverses pour préférer Java à SQL ou inversement.

- D'un point de vue monétaire, les grands vendeurs de base de données pousseront toujours à préférer SQL. Non seulement parce que cela fait partie de leur cœur de métier, mais surtout parce que cela lie fortement le logiciel avec leur produit et empêche ainsi de migrer vers la concurrence. En effet tous les vendeurs proposent des extensions de SQL qui leur sont spécifiques, et dès que l'on commence à les exploiter il devient difficile voire impossible de porter ensuite la base de données vers un autre produit. Si l'on veut éviter ce piège, il devient important de se limiter au strict minimum de SQL possible en favorisant au plus Java. Souvent en entreprise, ceci est une décision stratégique sur laquelle le développeur n'aura pas d'influence. Les petites entreprises préfèrent en général ne pas se lier à un vendeur pour limiter leurs coûts, tandis que les grosses entreprises préfèrent se lier à Oracle ou à Microsoft afin de bénéficier d'un support préférentiel.
- D'un point de vue pragmatique, on peut s'interroger sur la nécessité de reproduire en Java des fonctionnalités déjà existantes en SQL. Surtout que ces fonctionnalités SQL sont sûrement des solutions beaucoup plus matures et optimisées. Ainsi ce courant de pensée pousse à utiliser la base de données dans toutes ses possibilités. La partie Java doit être la

moins concernée possible par le stockage des données, c'est la base de données qui a cette responsabilité.

- L'inconvénient de l'approche précédente est que maintenant le code gérant le stockage des données est en petite partie en Java et en grande partie en SQL. Le développeur doit avoir une bonne maîtrise technique des deux langages pour avoir une chance d'y résoudre un bug. Il lui faudra deux débogueurs dans deux environnements de développement différents. Pire, le mécanisme des triggers s'exécute automatiquement et modifie des données « en cachette » du code que l'on pense être en train de déboguer. Il devient difficile de prédire le comportement de l'application. Il faut donc être excessivement attentif à d'abord trouver les triggers concernés et ensuite à bien voir quand et comment ils seront exécutés. Si par contre on se restreint au Java, on n'a plus qu'un seul environnement de développement et il n'y a plus de triggers mais bien des méthodes qui s'appellent entre elles. Il devient plus simple de comprendre le comportement de l'application et donc d'y résoudre les éventuels problèmes.

Le choix d'une approche ou d'une autre dépend donc d'abord de la présence ou pas d'une politique d'entreprise à ce niveau. En son absence, il est souvent préférable de limiter autant que possible l'usage du SQL. Ainsi on se garde la possibilité de migrer vers un autre système et on limite la connaissance technique nécessaire à la compréhension et maintenance de l'application. On diminue aussi le travail effectué par le serveur et on augmente donc ses performances. On perd évidemment l'avantage des services avancés que la base de données est capable de fournir.

Dans le cadre de ce cours, nous nous concentrons en priorité sur l'approche maximisant l'usage du SQL puisqu'après tout c'est le thème même de ce cours. Les chapitres suivants décriront une partie des bonnes pratiques de développement.

6.h. Bonne pratique JDBC en général

En gardant à l'esprit que le code SQL est dépendant de la base de données, on a un intérêt fort à isoler autant que possible toute la partie du code Java qui interagit avec JDBC. Ainsi on aura un seul endroit à mettre à jour en cas de migration. Il y a des raisons supplémentaires (architecture, modularisation, pattern) pour vouloir isoler cette partie au sein de l'application, vous verrez cela dans d'autres cours. La bonne pratique consistera donc à :

- Créer un package qui contiendra toute la partie de l'application Java interagissant directement avec la base de données.
- Au sein de ce package, créer une classe de gestion de la connexion JDBC. Au démarrage de l'application, cette classe sera instanciée et elle établira alors la connexion physique avec la base de données.
- Au sein de ce package, créer des classes gérant un ensemble de requêtes. Sans vouloir aller trop loin sur la manière d'écrire ces classes, une première approche consiste à regrouper des requêtes logiquement proches ensembles. Ainsi on regroupera au sein d'une classe toutes les requêtes concernant la gestion d'une authentification d'une personne par exemple. On peut évidemment utiliser des sous-packages pour continuer à organiser ces classes.
- Lorsqu'une classe qui gère des requêtes est instanciée, elle crée autant de `PreparedStatement` que nécessaire. Lorsque des méthodes sont appelées sur cette instance, ces méthodes n'utilisent que des `PreparedStatement` pour communiquer avec la base de données.

- Utiliser des `try` avec ressource pour s'assurer que les `ResultSet` sont bien fermés correctement.
- Fermer la connexion lorsque l'application se termine.

6.i. Bonne pratique JDBC lorsque l'on minimise l'usage du SQL

Dans cette approche, on essaie de dépendre du minimum possible de SQL. Comme la création de la base de données s'effectue une et une seule fois, on créera un script SQL (un simple fichier texte de commandes SQL) pour la création de la base de données. On se passera complètement des triggers et des procédures stockées : si ce mécanisme est nécessaire au fonctionnement de l'application, il faudra reproduire son équivalent en Java. On gardera par contre les index pour des raisons de performance. On évitera tout ce qui est spécifique à la base de données, comme les index auto-générés et les fonctionnalités non standard. On pourra se contenter d'un seul rôle (public) sachant que c'est au niveau du code Java que la sécurité d'accès à la base de données sera implémentée.

- Au niveau du code Java, les requêtes seront des commandes `SELECT`, `UPDATE`, `DELETE` explicites.

6.j. Bonne pratique JDBC lorsque l'on maximise l'usage du SQL

Dans cette approche, on va s'abstraire autant que possible au niveau du Java du SQL sous-jacent. Ceci permettra de faire évoluer la base de données en minimisant l'impact sur la partie Java :

- Éviter les requêtes portant directement sur des tables. A la place, écrire des procédures stockées au niveau du serveur et n'utiliser que cela au niveau du client.
- Si le schéma de la base de donnée change, utiliser les vues pour garder constant les données retournées au client. Par exemple si on ajoute une colonne à la table `titles`, créer une vue qui ignore cette colonne.

De plus on peut utiliser les fonctionnalités de la base de données pour amener plus de garanties de correction :

- Utiliser des séquences afin de générer automatiquement des identifiants unique pour les tuples (des clefs primaires). Comme c'est le serveur qui gère cela, on a la garantie qu'il n'y aura jamais de collision de clefs primaires (deux clients pourraient souhaiter s'approprier la même clef au même moment).
- Créer des triggers pour gérer automatiquement la bonne cohérence des données.
- Créer des utilisateurs dans la base de données correspondant au rôle logique dont a besoin l'application. Par exemple une application de consultation uniquement se connectera au serveur en spécifiant un utilisateur possédant uniquement le droit d'effectuer des `SELECT`. Par contre l'application d'administration elle se connectera avec un utilisateur administrateur de la base de données.

Exercice

1. Porter la base de données `pubs2` sur un serveur MySQL. Quel sont les différences entre le script créant cette base sur PostgreSQL et le script la créant sur MySQL ?

6.k. Exemple

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```



```

public class ProgrammePrincipal {

    private String url="jdbc:postgresql://postgres.ipl.be/pubs2" +
        "?user=public&password=public";
    private PreparedStatement listeAuteurs;
    private PreparedStatement listeAuteursAvecNom;
    private Connection conn=null;

    public ProgrammePrincipal() {
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("Driver PostgreSQL manquant !");
            System.exit(1);
        }
        try {
            conn=DriverManager.getConnection(url);
        } catch (SQLException e) {
            System.out.println("Impossible de joindre le server !");
            System.exit(1);
        }
        try {
            listeAuteurs=
                conn.prepareStatement("SELECT au_fname, au_lname" +
                    " FROM Authors");
            listeAuteursAvecNom=
                conn.prepareStatement("SELECT au_lname" +
                    " FROM Authors" +
                    " WHERE au_fname LIKE ?");
        } catch (SQLException e) {
            System.out.println("Erreur avec les requêtes SQL !");
            System.exit(1);
        }
    }

    private void listeAuteursAvecNom(String name) {
        try {
            listeAuteursAvecNom.setString(1,name);
            try (ResultSet rs=listeAuteursAvecNom.executeQuery()) {
                while(rs.next()) {
                    System.out.println("Prénom "+rs.getString(1));
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void listeAuteurs() {
        try {
            try (ResultSet rs=listeAuteurs.executeQuery()) {
                while(rs.next()) {
                    System.out.println("Nom " + rs.getString(1) +
                        " Prénom " + rs.getString(2));
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

```
    }

    public void close() {
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ProgrammePrincipal pp=new ProgrammePrincipal();
        System.out.println("Liste des auteurs");
        pp.listeAuteurs();
        System.out.println("Liste des auteurs dont le nom est Sheryl");
        pp.listeAuteursAvecNom("Sheryl");
        pp.close();
    }
}
```

6.1.Exercice

1. Compléter le programme précédent : ajouter une méthode permettant d'ajouter un livre.
2. Compléter le programme précédent : ajouter une méthode permettant d'ajouter un auteur.
3. Compléter le programme précédent : ajouter une méthode permettant d'ajouter un magasin.
4. Compléter le programme précédent : ajouter une méthode permettant d'ajouter un éditeur.
5. Compléter le programme précédent : ajouter une méthode permettant d'ajouter une vente.
6. Compléter le programme précédent : créer un interface textuelle permettant à l'utilisateur a) de choisir quelle type de donnée il veut ajouter (1=livre, 2=auteur, 3=magasin, 4=éditeur, 5=vente) et b) de saisir chacun des champs nécessaire à la création d'un tuple de ce type. Finalement c) le programme devra appeler la bonne méthode d'ajout.
7. Compléter le programme précédent : ajouter une méthode permettant lister les livres.
8. Compléter le programme précédent : ajouter une méthode permettant d'afficher tous les détails d'un livre particulier.
9. Compléter le programme précédent : ajouter une méthode permettant d'afficher tous les détails d'un auteur particulier.
10. Compléter le programme précédent : ajouter une méthode permettant lister les magasins.
11. Compléter le programme précédent : ajouter une méthode permettant d'afficher tous les détails d'un magasin particulier.
12. Compléter le programme précédent : ajouter une méthode permettant lister les éditeurs.
13. Compléter le programme précédent : ajouter une méthode permettant d'afficher tous les détails d'un éditeur particulier.
14. Compléter le programme précédent : ajouter une méthode permettant lister les ventes.
15. Compléter le programme précédent : ajouter une méthode permettant d'afficher tous les détails d'une vente particulière.
16. Compléter le programme précédent : étendre l'interface textuelle pour permettre à l'utilisateur de lister les différentes informations de la base de données. Utiliser les méthodes écrites ci-dessus.
17. Compléter le programme précédent : étendre l'interface textuelle pour permettre à l'utilisateur d'afficher les détails d'une information spécifique de la base de données. Utiliser les méthodes écrites
18. Créer des fonctions au niveau du serveur pour remplacer toutes les requêtes utilisées par le client. Adapter le code du client pour n'utiliser que ces fonctions.

7. Annexes

7.a. Exercices récapitulatifs sur les requêtes

1. Quelles sont les villes où habite (au moins) un auteur dont le nom commence par la lettre B ou par la lettre C ?
2. Quelles sont les villes où habitent au moins 3 auteurs dont le nom commence par la lettre B ou par la lettre C ?
3. Quels sont les auteurs (nom, prénom) qui ont écrit (au moins) un livre de plus de 21\$? Affichez-les en ordre alphabétique.
4. Quels sont les auteurs (nom, prénom) dont on a vendu (au moins) un livre de plus de 15\$ au mois de juin 1990 dans un magasin californien ?
5. Quels sont les livres (titres) qui ont été vendus en Californie, mais pas en Oregon ?
6. Pour chaque éditeur, donnez son nom, son adresse, ainsi que le prix moyen des livres qu'il a publiés et qui ont été écrits par plusieurs auteurs.
7. Quel est l'éditeur qui a publié le plus de livres d'auteurs californiens ? (Donnez son nom.) (On appelle "livres d'auteurs californiens" ceux qui ont au moins un auteur californien.)
8. Y a-t-il des livres de types différents publiés chez le même éditeur ? Si oui, fournissez le numéro et le nom de l'éditeur, ainsi que les deux types.
9. Quels sont les types des livres comprenant au moins un livre publié avant 1992 ?
10. Quels sont les magasins (nom, adresse) où l'on a vendu (au moins) un livre dont le nom contient la chaîne de caractères "Data" ?
11. Quels sont les auteurs (nom, prénom) ayant écrit exactement 2 livres publiés par "New Age Books" ? Donnez-les en ordre alphabétique. (N.B. Ces auteurs pourraient avoir aussi publié d'autres livres chez d'autres éditeurs! Comparez avec l'exercice suivant.)
12. Quels sont les auteurs (nom, prénom) ayant écrit exactement 2 livres, et tels que ces livres soient tous deux publiés par "New Age Books" ?
13. Quels sont les magasins (nom, adresse) où l'on a vendu, au mois de juin 1990, (au moins) un livre de plus de 15\$ écrit par un auteur californien ?
14. Y a-t-il des paires d'auteurs ayant le même nom ? Pour chaque paire, donnez le nom commun et les deux prénoms.
15. Quel est le nom de l'éditeur qui a publié le moins de livres de plus de 15\$ vendus en Californie ?
16. Pour chaque livre de psychologie, donnez son titre, le nombre d'auteurs et la moyenne des quantités répertoriées pour ce livre dans la table "Salesdetail".

17. Quels sont les Etats ("state") d'Amérique ("country" : USA) où l'on peut trouver au moins un magasin répertorié dans la DB ?
18. Quels sont les livres (titres) que l'on n'a vendu que dans un seul magasin ?
19. Quels sont les auteurs (nom, prénom) n'ayant pas publié tous leurs livres chez le même éditeur ?
20. Pour chaque magasin, donnez son nom, son adresse, ainsi que son montant total de vente (obtenu en additionnant les montants de détail, que l'on trouve en multipliant les quantités par les prix unitaires). Classez-les en ordre décroissant du montant total.
21. Quels sont les éditeurs (nom, adresse) dont on a vendu, au mois d'août 1988, (au moins) un livre écrit par un auteur dont le numéro de téléphone commence par "415" ?
22. Y a-t-il des paires d'auteurs vivant dans la même ville mais qui ont des codes postaux différents ? Pour chaque paire, donnez la ville commune, les deux codes postaux, les deux noms.
23. Quel est l'auteur qui a écrit le plus de livres publiés par "Algodata Infosystems" avant 1989?
24. Quels sont les magasins où l'on a vendu au moins un livre écrit par un auteur californien mais aucun livre publié en 1990 ?
25. Quels sont les livres qui coûtent plus cher que la moyenne des livres de leur type?
26. Quels sont les auteurs qui n'ont écrit aucun livre ayant été publié en même temps (même date) et chez le même éditeur qu'un livre écrit par Anne Ringer ?
27. Donnez les titres et noms d'éditeurs de tous les livres écrits par un auteur de nom "Simenon" et de prénom commençant par la lettre "G".
28. Même question, mais on se limite aux livres écrits uniquement par un certain "G. Simenon", en excluant donc les livres qui auraient un co-auteur. (Exemple : on exclut cette fois le livre "Maigret contre Hercule Poirot", écrit par G. Simenon et A. Christie.)
29. On s'intéresse aux ventes des livres édités par l'éditeur "Algodata Infosystems". On souhaite obtenir la liste des magasins (stor_id + stor_name) qui ont vendu certains de ces livres, avec, pour chacun d'eux, le nombre total d'exemplaires vendus (la somme des valeurs de l'attribut "qty").
30. Même question, mais sans afficher les magasins où ce nombre total est inférieur à 10.
31. Affichez la liste des numéros et titres des livres de psychologie.
32. Affichez la liste des numéros et titres des livres de psychologie, ainsi que, pour chacun d'eux, le nom de l'éditeur.
33. Affichez la liste des numéros et titres des livres de psychologie, ainsi que, pour chacun d'eux, le nom de l'éditeur et le nombre d'auteurs.

34. Affichez la liste des numéros et titres des livres de psychologie qui ont été vendus en 1000 exemplaires au moins. (Pour connaître le nombre d'exemplaires vendus, additionnez les "qty" de la table "salesdetail".)
35. Affichez la liste des numéros des livres de psychologie qui ont été vendus dans des magasins de plusieurs villes différentes. (Attention : deux villes d'états différents ont parfois le *même nom*, mais il s'agit bien sûr de villes distinctes ! Ex. : Washington.)
36. Comme la question précédente. Mais on vous demande en outre d'afficher les noms de ces villes et de leurs états, en regard du livre concerné.
37. Donnez les numéros et les titres des livres qui ont été vendus en 1991 mais pas en 1990.
38. Quel est l'éditeur dont on a vendu le plus d'exemplaires de livres en Californie en 1991 ?
39. Quels sont les Etats où l'on a vendu des livres qui ont été écrits par un seul auteur ?
40. Est-ce que l'attribut "total_sales" est bien égal, pour chaque livre, à la somme des valeurs de l'attribut "qty" des détails de vente relatifs à ce livre ? Pour chaque cas d'erreur, affichez le numéro du livre, son titre, et les deux nombres qui auraient dû être égaux.
41. Quels sont les livres (numéros, titres) que l'on a déjà vendus moins d'un an après leur date de parution ?
42. Quels sont les éditeurs (numéros, noms) ayant publié un (des) livre(s) d'auteur(s) de Californie, mais aucun livre d'auteur(s) de l'Oregon ?
43. Donnez les noms et villes des magasins où l'on n'a vendu aucun des livres qui ont été vendus par le magasin "Bookbeat" de la ville de Portland.
44. Donnez les noms et villes des magasins où l'on a vendu tous les livres qui ont été vendus par le magasin "Bookbeat" de la ville de Portland.
45. Classez les villes de Californie selon le nombre d'auteurs qui y habitent, en commençant par celles où il y en a le plus. Ignorez les villes où habitent moins de 2 auteurs.
46. Donnez le prix du livre le plus cher de chaque type.
47. Donnez le prix et le titre du livre le plus cher de chaque type.
48. Pour chaque livre de psychologie, donnez son numéro, le début de son titre, son prix, le nom de son éditeur, le nombre de magasins différents où il a été vendu, la date de parution et la première date de vente. Classez-les en ordre alphabétique d'éditeur, et, parmi ceux qui ont le même éditeur, en ordre chronologique de parution.
49. Quels sont les co-auteurs d'Anne Ringer ? (C'est-à-dire les auteurs ayant écrit au moins un livre en commun avec Anne Ringer.)
50. Quels sont les magasins situés dans le même pays et le même état qu'au moins un auteur ?
51. Quels sont les magasins situés dans le même pays et le même état qu'exactly un auteur ?

52. Quels sont les magasins situés dans un pays et un état où n'habite aucun un auteur ?
53. Quel est l'auteur qui a écrit le plus de livres ? Donnez son numéro, son nom, son prénom.
54. Quel est l'auteur qui a écrit le plus de livres vendus en Oregon? Donnez son numéro, son nom, son prénom.
55. Quel est l'auteur qui a écrit seul (sans co-auteur) le plus de livres ? Donnez son numéro, son nom, son prénom.
56. Trouvez les éventuelles paires d'auteurs habitant le même état et ayant écrit des livres de même type. Pour chaque paire, affichez les noms et prénoms des deux auteurs, leur état et le type commun.
57. Y a-t-il des co-auteurs (des paires d'auteurs du même livre) qui habitaient dans le même état mais pas dans la même ville ? Donnez les 2 noms et prénoms, l'état, les 2 villes.

7.b. Notation BNF

Les éléments syntaxiques de ce syllabus sont décrits à l'aide de "Backus-Naur Forms" (BNF), selon les conventions suivantes :

- Toute règle syntaxique, définissant un symbole terminal ou non-terminal, se présente sous la forme

nom-du-symbole = définition .

où les signes "=" et "." sont des métasymboles du métalangage BNF.

- Toute séquence de caractères placée entre guillemets doubles figurera telle quelle dans le langage.
- Tout identificateur non placé entre guillemets est un symbole non-terminal : il représente son développement, qui sera décrit dans une autre règle.
- Le métasymbole "|" représente un "ou exclusif".
- Les métasymboles "[" et "]" encadrent une partie facultative.
- Les métasymboles "{" et "}" encadrent une partie qui doit être répétée 0, 1 ou plusieurs fois.
- La simple juxtaposition d'éléments dans le métalangage signifie une simple juxtaposition de leurs développements dans le langage décrit.
- Le métasymbole "|" a une priorité moins élevée que la juxtaposition, mais les priorités peuvent être modifiées par l'emploi des parenthèses arrondies.

Exemple:

texte = phrase {phrase } .

phrase = prénom verbe {complément} ". "

prénom = "Pierre" | "Jean".

verbe = "voit" | "entend" | "ignore" | "sourit".

complément = prénom | "dans" ("le" "train" | "la" ["belle"] "maison") | "toujours" | "maintenant".

Un "texte" répondant à cette syntaxe serait :

Pierre voit Jean dans le train. Jean ignore toujours Pierre. Pierre sourit. Jean entend Jean toujours
Jean Jean Pierre dans le train Pierre.

(La dernière phrase de l'exemple montre que la syntaxe proposée n'est pas assez stricte pour engendrer un sous-ensemble de la langue française! Exercice : corrigez-la, puis enrichissez-la.)

7.c. Référence rapide PostgreSQL

La référence complète en ligne se trouve ici : <http://docs.postgresql.fr/>

Cette section présente une sélection partielle de la référence complète qui devrait couvrir la majorité de vos besoins pour ce cours.

7.c.i. Liste des commandes SQL

ABORT — Interrompre la transaction en cours

ALTER AGGREGATE — Modifier la définition d'une fonction d'agrégat

ALTER CONVERSION — Modifier la définition d'une conversion

ALTER DATABASE — Modifier une base de données

ALTER DOMAIN — Modifier la définition d'un domaine

ALTER FOREIGN DATA WRAPPER — modifier la définition d'un wrapper de données distantes

ALTER FUNCTION — Modifier la définition d'une fonction

ALTER GROUP — Modifier le nom d'un rôle ou la liste de ses membres

ALTER INDEX — Modifier la définition d'un index

ALTER LANGUAGE — Modifier la définition d'un langage procédural

ALTER OPERATOR — Modifier la définition d'un opérateur

ALTER OPERATOR CLASS — Modifier la définition d'une classe d'opérateur

ALTER OPERATOR FAMILY — Modifier la définition d'une famille d'opérateur

ALTER ROLE — Modifier un rôle de base de données

ALTER SCHEMA — Modifier la définition d'un schéma

ALTER SEQUENCE — Modifier la définition d'un générateur de séquence

ALTER SERVER — modifier la définition d'un serveur distant

ALTER TABLE — Modifier la définition d'une table

ALTER TABLESPACE — Modifier la définition d'un tablespace

ALTER TEXT SEARCH CONFIGURATION — modifier la définition d'une configuration de recherche plein texte

ALTER TEXT SEARCH DICTIONARY — modifier la définition d'un dictionnaire de recherche plein texte

ALTER TEXT SEARCH PARSER — modifier la définition d'un analyseur de recherche plein texte

ALTER TEXT SEARCH TEMPLATE — modifier la définition d'un modèle de recherche plein texte

ALTER TRIGGER — Modifier la définition d'un déclencheur

ALTER TYPE — Modifier la définition d'un type

ALTER USER — Modifier un rôle de la base de données

ALTER USER MAPPING — change la définition d'une correspondance d'utilisateurs (user mapping)

ALTER VIEW — modifier la définition d'une vue

ANALYZE — Collecter les statistiques d'une base de données

BEGIN — Débuter un bloc de transaction

CHECKPOINT — Forcer un point de vérification dans le journal des transactions

CLOSE — Fermer un curseur

CLUSTER — Réorganiser une table en fonction d'un index

COMMENT — Définir ou modifier le commentaire associé à un objet

COMMIT — Valider la transaction en cours

COMMIT PREPARED — Valider une transaction préalablement préparée en vue d'une validation en deux phases

COPY — Copier des données depuis/vers un fichier vers/depuis une table

CREATE AGGREGATE — Définir une nouvelle fonction d'agrégat

CREATE CAST — Définir un transtypage

CREATE CONSTRAINT TRIGGER — Définir un nouveau déclencheur sur contrainte

CREATE CONVERSION — Définir une nouvelle conversion d'encodage

CREATE DATABASE — Créer une nouvelle base de données

CREATE DOMAIN — Définir un nouveau domaine

CREATE FOREIGN DATA WRAPPER — définit un nouveau wrapper de données distantes

CREATE FUNCTION — Définir une nouvelle fonction

CREATE GROUP — Définir un nouveau rôle de base de données

CREATE INDEX — Définir un nouvel index

CREATE LANGUAGE — Définir un nouveau langage procédural

CREATE OPERATOR — Définir un nouvel opérateur

CREATE OPERATOR CLASS — Définir une nouvelle classe d'opérateur

CREATE OPERATOR FAMILY — définir une nouvelle famille d'opérateur

CREATE ROLE — Définir un nouveau rôle de base de données

CREATE RULE — Définir une nouvelle règle de réécriture

CREATE SCHEMA — Définir un nouveau schéma

CREATE SEQUENCE — Définir un nouveau générateur de séquence

CREATE SERVER — Définir un nouveau serveur distant

CREATE TABLE — Définir une nouvelle table

CREATE TABLE AS — Définir une nouvelle table à partir des résultats d'une requête

CREATE TABLESPACE — Définir un nouvel tablespace

CREATE TEXT SEARCH CONFIGURATION — définir une nouvelle configuration de recherche plein texte

CREATE TEXT SEARCH DICTIONARY — définir un dictionnaire de recherche plein texte

CREATE TEXT SEARCH PARSER — définir un nouvel analyseur de recherche plein texte

CREATE TEXT SEARCH TEMPLATE — définir un nouveau modèle de recherche plein texte

CREATE TRIGGER — Définir un nouveau déclencheur

CREATE TYPE — Définir un nouveau type de données

CREATE USER — Définir un nouveau rôle de base de données

CREATE USER MAPPING — Définir une nouvelle correspondance d'utilisateur (user mapping) pour un serveur distant

CREATE VIEW — Définir une vue

DEALLOCATE — Désaffecter (libérer) une instruction préparée

DECLARE — Définir un curseur

DELETE — Supprimer des lignes d'une table

DISCARD — Annuler l'état de la session

DROP AGGREGATE — Supprimer une fonction d'agrégat

DROP CAST — Supprimer un transtypage

DROP CONVERSION — Supprimer une conversion

DROP DATABASE — Supprimer une base de données

DROP DOMAIN — Supprimer un domaine

DROP FOREIGN DATA WRAPPER — Supprimer un wrapper de données distantes

DROP FUNCTION — Supprimer une fonction

DROP GROUP — Supprimer un rôle de base de données

DROP INDEX — Supprimer un index

DROP LANGUAGE — Supprimer un langage procédural

DROP OPERATOR — Supprimer un opérateur

DROP OPERATOR CLASS — Supprimer une classe d'opérateur

DROP OPERATOR FAMILY — Supprimer une famille d'opérateur

DROP OWNED — Supprimer les objets de la base possédés par un rôle

DROP ROLE — Supprimer un rôle de base de données

DROP RULE — Supprimer une règle de réécriture

DROP SCHEMA — Supprimer un schéma

DROP SEQUENCE — Supprimer une séquence

DROP SERVER — Supprimer un descripteur de serveur distant

DROP TABLE — Supprimer une table

DROP TABLESPACE — Supprimer un tablespace

DROP TEXT SEARCH CONFIGURATION — Supprimer une configuration de recherche plein texte

DROP TEXT SEARCH DICTIONARY — Supprimer un dictionnaire de recherche plein texte

DROP TEXT SEARCH PARSER — Supprimer un analyseur de recherche plein texte

DROP TEXT SEARCH TEMPLATE — Supprimer un modèle de recherche plein texte

DROP TRIGGER — Supprimer un déclencheur

DROP TYPE — Supprimer un type de données

DROP USER — Supprimer un rôle de base de données

DROP USER MAPPING — Supprimer une correspondance d'utilisateur pour un serveur distant

DROP VIEW — Supprimer une vue

END — Valider la transaction en cours

EXECUTE — Exécuter une instruction préparée

EXPLAIN — Afficher le plan d'exécution d'une instruction

FETCH — Récupérer les lignes d'une requête à l'aide d'un curseur

GRANT — Définir les droits d'accès

INSERT — Insérer de nouvelles lignes dans une table

LISTEN — Attendre une notification

LOAD — Charger une bibliothèque partagée

LOCK — verrouiller une table

MOVE — positionner un curseur

NOTIFY — engendrer une notification

PREPARE — prépare une instruction pour exécution

PREPARE TRANSACTION — prépare la transaction en cours pour une validation en deux phases

REASSIGN OWNED — Modifier le propriétaire de tous les objets de la base appartenant à un rôle spécifique

REINDEX — reconstruit les index

RELEASE SAVEPOINT — détruit un point de sauvegarde précédemment défini

RESET — réinitialise un paramètre d'exécution à sa valeur par défaut

REVOKE — supprime les droits d'accès

ROLLBACK — annule la transaction en cours

ROLLBACK PREPARED — annule une transaction précédemment préparée en vue d'une validation en deux phases

ROLLBACK TO SAVEPOINT — annule les instructions jusqu'au point de sauvegarde

SAVEPOINT — définit un nouveau point de sauvegarde à l'intérieur de la transaction en cours

SELECT — récupère des lignes d'une table ou d'une vue

SELECT INTO — définit une nouvelle table à partir des résultats d'une requête

SET — change un paramètre d'exécution

SET CONSTRAINTS — initialise le mode de vérification de contrainte de la transaction en cours

SET ROLE — initialise l'identifiant utilisateur courant de la session en cours

SET SESSION AUTHORIZATION — Initialise l'identifiant de session de l'utilisateur et l'identifiant de l'utilisateur actuel de la session en cours

SET TRANSACTION — initialise les caractéristiques de la transaction actuelle

SHOW — affiche la valeur d'un paramètre d'exécution

START TRANSACTION — débute un bloc de transaction

TRUNCATE — vide une table ou un ensemble de tables

UNLISTEN — arrête l'écoute d'une notification

UPDATE — mettre à jour les lignes d'une table

VACUUM — récupère l'espace inutilisé et, optionnellement, analyse une base

VALUES — calcule un ensemble de lignes

7.c.ii. **Types**

Nom	Alias	Description
bigint	int8	Entier signé sur 8 octets
bigserial	serial8	Entier sur 8 octets à incrémentation automatique
bit [(n)]	varbit	Suite de bits de longueur fixe
bit varying [(n)]		Suite de bits de longueur variable
boolean	bool	Booléen (Vrai/Faux)
box	varchar [(n)]	Boîte rectangulaire dans le plan
bytea		Donnée binaire (« tableau d'octets »)
character varying [(n)]		Chaîne de caractères de longueur variable de taille maximum n
character [(n)]		Chaîne de caractères de longueur fixe égale à n
cidr	char [(n)]	Adresse réseau IPv4 ou IPv6
circle	float8	Cercle dans le plan
date		Date du calendrier (année, mois, jour)
double precision		Nombre à virgule flottante de double précision (sur huit octets)
inet		Adresse d'ordinateur IPv4 ou IPv6
integer	int, int4	Entier signé sur 4 octets
interval [champs] [(p)]	decimal [(p, s)]	Intervalle de temps
line		Droite (infinie) dans le plan
lseg		Segment de droite dans le plan
macaddr		Adresse MAC (pour <i>Media Access Control</i>)
money	float4	Montant monétaire
numeric [(p, s)]		Nombre exact dont la précision peut être précisée
path		Chemin géométrique dans le plan
point		Point géométrique dans le plan
polygon	int2	Chemin géométrique fermé dans le plan
real		Nombre à virgule flottante de simple précision (sur quatre octets)
smallint		Entier signé sur 2 octets
serial	serial4	Entier sur 4 octets à incrémentation automatique
text	timetz	Chaîne de caractères de longueur variable
time [(p)] [without time zone]		Heure du jour (pas du fuseau horaire)
time [(p)] with time zone		Heure du jour, avec fuseau horaire
timestamp [(p)] [without time zone]		Date et heure (pas du fuseau horaire)
timestamp [(p)] with time zone	timestamptz	Date et heure, avec fuseau horaire
tsquery	timestamptz	requête pour la recherche plein texte
tsvector		document pour la recherche plein texte
txid_snapshot		image de l'identifiant de transaction au niveau utilisateur
uuid		identifiant unique universel
xml		données XML

7.c.iii. Fonctions d'agrégat

Fonction	Type d'argument	Type de retour	Description
<code>array_agg(expression)</code>	any	tableau du type de l'argument	les valeurs entrées concaténées dans un tableau
<code>avg(expression)</code>	smallint, int, bigint, real, double precision, numeric ou interval	numeric pour tout argument de type entier, double precision pour tout argument en virgule flottante, sinon identique au type de données de l'argument	la moyenne arithmétique de toutes les valeurs en entrée
<code>bit_and(expression)</code>	smallint, int, bigint ou bit	identique au type de données de l'argument	le AND bit à bit de toutes les valeurs non NULL en entrée ou NULL s'il n'y en a pas
<code>bit_or(expression)</code>	smallint, int, bigint ou bit	identique au type de données de l'argument	le OR bit à bit de toutes les valeurs non NULL en entrée ou NULL s'il n'y en a pas
<code>bool_and(expression)</code>	bool	bool	true si toutes les valeurs en entrée valent true, false sinon
<code>bool_or(expression)</code>	bool	bool	true si au moins une valeur en entrée vaut true, false sinon
<code>count(*)</code>		bigint	nombre de lignes en entrée
<code>count(expression)</code>	tout type	bigint	nombre de lignes en entrée pour lesquelles l' <i>expression</i> n'est pas NULL
<code>max(expression)</code>	tout type array, numeric, string ou date/time	identique au type en argument	valeur maximale de l' <i>expression</i> pour toutes les valeurs en entrée
<code>min(expression)</code>	tout type array, numeric, string ou date/time	identique au type en argument	valeur minimale de l' <i>expression</i> pour toutes les valeurs en entrée
<code>sum(expression)</code>	smallint, int, bigint, real, double precision, numeric ou interval	bigint pour les arguments de type smallint ou int, numeric pour les arguments de type bigint, double precision pour les arguments en virgule flottante, sinon identique au type de données de l'argument	somme de l' <i>expression</i> pour toutes les valeurs en entrée

7.c.iv. Fonctions pour le type chaîne

Fonction	Type renvoyé	Description	Exemple	Résultat
<i>chaîne</i> <i>chaîne</i>	text	Concaténation de chaînes	'Post' 'greSQL'	PostgreSQL
<i>chaîne</i> <i>autre-que-chaîne</i> ou <i>autre-que-chaîne</i> <i>chaîne</i>	text	Concaténation de chaînes avec un argument non-chaîne	'Value: ' 42	Value: 42
char_length(<i>chaîne</i>) ou character_length(<i>chaîne</i>)	int	Nombre de caractères de la chaîne	char_length('jose')	4
lower(<i>chaîne</i>)	text	Convertit une chaîne en minuscule	lower('TOM')	tom
octet_length(<i>chaîne</i>)	int	Nombre d'octets de la chaîne	octet_length('jose')	4
overlay(<i>chaîne</i> placing <i>chaîne</i> from int [for int])	text	Remplace la sous-chaîne	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(substring in <i>chaîne</i>)	int	Emplacement de la sous-chaîne indiquée	position('om' in 'Thomas')	3
substring(<i>chaîne</i> [from int] [for int])	text	Extrait une sous-chaîne	substring('Thomas' from 2 for 3)	hom
trim([leading trailing both] [<i>caractères</i>] from <i>chaîne</i>)	text	Supprime la plus grande chaîne qui ne contient que les <i>caractères</i> (une espace par défaut) à partir du début, de la fin ou des deux extrémités (respectivement leading, trailing, both) de la <i>chaîne</i> .	trim(both 'x' from 'xTomxx')	Tom
upper(<i>chaîne</i>)	text	Convertit une chaîne en majuscule	upper('tom')	TOM

7.c.v. **Opérateurs et Fonctions pour le type date**

Opérateur	Exemple	Résultat
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (jours)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day - 01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Fonction	Code de retour	Description	Exemple	Resultat
<code>age(timestamp, timestamp)</code>	interval	Soustrait les arguments, ce qui produit un resultat « symbolique » en années, mois, jours	<code>age(timestamp '2001-04-10', timestamp '1957-06-13')</code>	43 years 9 mons 2 7 days
<code>age(timestamp)</code>	interval	Soustrait a la date courante (<code>current_date</code> a minuit)	<code>age(timestamp '1957-06-13')</code>	43 years 8 mons 3 days
<code>clock_timestamp()</code>	timestamp with time zone	Date et heure courantes (change pendant l'execution de l'instruction)		
<code>current_date</code>	date	Date courante		1
<code>current_time</code>	time with time zone	Heure courante		
<code>current_timestamp</code>	timestamp with time zone	Date et heure courantes (debut de la transaction en cours)		
<code>date_part(text, timestamp)</code>	double precision	Obtenir un sous-champ	<code>date_part('hour', timestamp '2001-02-16 20:38:40')</code>	20
<code>date_part(text, interval)</code>	double precision	Obtenir un sous-champ (equivalent a extract)	<code>date_part('month', interval '2 years 3 months')</code>	3
<code>date_trunc(text, timestamp)</code>	timestamp	Tronquer a la precision indiquee	<code>date_trunc('hour', timestamp '2001-02-16 20:38:40')</code>	2001-02-16 20:00:00
<code>extract(field from timestamp)</code>	double precision	Obtenir un sous-champ	<code>extract(hour from timestamp '2001-02-16 20:38:40')</code>	20
<code>extract(field from interval)</code>	double precision	Obtenir un sous-champ	<code>extract(month from interval '2 years 3 months')</code>	3
<code>isfinite(date)</code>	boolean	Test si la date est finie (donc different de +/- infinity)	<code>isfinite(date '2001-02-16')</code>	true
<code>isfinite(timestamp)</code>	boolean	Teste si l'estampille temporelle est finie (donc different de +/- infinity)	<code>isfinite(timestamp '2001-02-16 21:28:30')</code>	true
<code>isfinite(interval)</code>	boolean	Teste si l'intervalle est fini	<code>isfinite(interval '4 hours')</code>	true
<code>justify_days(interval)</code>	interval	Ajuste l'intervalle pour que les periodes de 30 jours soient representees comme des mois	<code>justify_days(interval '35 days')</code>	1 mon 5 days
<code>justify_hours(interval)</code>	interval	Ajuste l'intervalle pour que les periodes de 24 heures soient representees comme des jours	<code>justify_hours(interval '27 hours')</code>	1 day 03:00:00
<code>justify(interval)</code>	interval	Ajuste l'intervalle en utilisant <code>justify_days</code> et <code>justify_hours</code> , avec des signes supplementaires d'ajustement	<code>justify interval(interval '29 days 23:00:00')</code>	29 days 23:00:00
<code>localtime</code>	time	Heure du jour courante		

localtimestamp	timestamp	Date et heure courantes (debut de la transaction)		
now ()	timestamp with time zone	Date et heure courantes (debut de la transaction)		
statement_timestamp ()	timestamp with time zone	Date et heure courantes (debut de l'instruction en cours)		
timeofday ()	text	Date et heure courantes (comme clock_timestamp mais avec une chaîne de type text)		
transaction_timestamp ()	timestamp with time zone	Date et heure courantes (debut de la transaction en cours)		

7.c.vi. Fonctions de transtypage

Fonction	Type en retour	Description	Exemple
to_char(timestamp, text)	text	convertit un champ de type timestamp en chaîne	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	convertit un champ de type interval en chaîne	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	convertit un champ de type integer en chaîne	to_char(125, '999')
to_char(double precision, text)	text	convertit un champ de type real/double precision en chaîne	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	convertit un champ de type numeric en chaîne	to_char(-125.8, '999D99S')
to_date(text, text)	date	convertit une chaîne en date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convertit une chaîne en champ de type numeric	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	convertit une chaîne string en champ de type timestamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(double precision)	timestamp with time zone	convertit une valeur de type epoch UNIX en valeur de type timestamp	to_timestamp(1284352323)

7.d. Codes d'erreur de PostgreSQL

Class 00 -- Succès de l'opération

- successful_completion

Class 01 -- Avertissement

- warning
- dynamic_result_sets_returned
- implicit_zero_bit_padding
- null_value_eliminated_in_set_function
- privilege_not_granted
- privilege_not_revoked
- string_data_right_truncation
- deprecated_feature

Class 02 -- Pas de données (également une classe d'avertissement selon le standard SQL)

- no_data
- no_additional_dynamic_result_sets_returned

Class 03 -- Instruction SQL pas encore terminée

- sql_statement_not_yet_complete

Class 08 -- Problème de connexion

- connection_exception
- connection_does_not_exist
- connection_failure
- sqlclient_unable_to_establish_sqlconnection
- sqlserver_rejected_establishment_of_sqlconnection
- transaction_resolution_unknown
- protocol_violation

Class 09 -- Problème d'action déclenchée

- triggered_action_exception

Class 0A -- Fonctionnalité non supportée

- feature_not_supported

Class 0B -- Initialisation de transaction invalide

- invalid_transaction_initiation

Classe 0F -- Problème de pointeur (Locator)

- locator_exception
- invalid_locator_specification

Classe 0L -- Granteur invalide

- invalid_grantor
- invalid_grant_operation

Classe 0P -- Spécification de rôle invalide

- invalid_role_specification

Class 20 -- Case Not Found

- case_not_found

Class 21 -- Violation de cardinalité

- cardinality_violation

Class 22 -- Problème de données

- data_exception
- array_subscript_error
- character_not_in_repertoire

- datetime_field_overflow
- division_by_zero
- error_in_assignment
- escape_character_conflict
- indicator_overflow
- interval_field_overflow
- invalid_argument_for_logarithm
- invalid_argument_for_ntile_function
- invalid_argument_for_nth_value_function
- invalid_argument_for_power_function
- invalid_argument_for_width_bucket_function
- invalid_character_value_for_cast
- invalid_datetime_format
- invalid_escape_character
- invalid_escape_octet
- invalid_escape_sequence
- nonstandard_use_of_escape_character
- invalid_indicator_parameter_value
- invalid_parameter_value
- invalid_regular_expression
- invalid_row_count_in_limit_clause
- invalid_row_count_in_result_offset_clause
- invalid_time_zone_displacement_value
- invalid_use_of_escape_character
- most_specific_type_mismatch
- null_value_not_allowed
- null_value_no_indicator_parameter
- numeric_value_out_of_range
- string_data_length_mismatch
- string_data_right_truncation
- substring_error
- trim_error
- unterminated_c_string
- zero_length_character_string
- floating_point_exception
- invalid_text_representation
- invalid_binary_representation
- bad_copy_file_format
- untranslatable_character
- not_an_xml_document
- invalid_xml_document
- invalid_xml_content
- invalid_xml_comment
- invalid_xml_processing_instruction

Class 23 -- Violation de contrainte d'intégrité

- integrity_constraint_violation
- restrict_violation
- not_null_violation
- foreign_key_violation
- unique_violation
- check_violation

Classe 24 -- État de curseur invalide

- invalid_cursor_state

Class 25 -- État de transaction invalide

- invalid_transaction_state
- active_sql_transaction
- branch_transaction_already_active

- held_cursor_requires_same_isolation_level
- inappropriate_access_mode_for_branch_transaction
- inappropriate_isolation_level_for_branch_transaction
- no_active_sql_transaction_for_branch_transaction
- read_only_sql_transaction
- schema_and_data_statement_mixing_not_supported
- no_active_sql_transaction
- in_failed_sql_transaction

Classe 26 -- Nom d'instruction SQL invalide

- invalid_sql_statement_name

Classe 27 -- Violation de modification de donnée déclenchée

- triggered_data_change_violation

Classe 28 -- Spécification d'autorisation invalide

- invalid_authorization_specification

Classe 2B -- Descripteurs de privilège dépendant toujours existant

- dependent_privilege_descriptors_still_exist
- dependent_objects_still_exist

Classe 2D -- Fin de transaction invalide

- invalid_transaction_termination

Classe 2F -- Exception dans une routine SQL

- sql_routine_exception
- function_executed_no_return_statement
- modifying_sql_data_not_permitted
- prohibited_sql_statement_attempted
- reading_sql_data_not_permitted

Classe 34 -- Nom de curseur invalide

- invalid_cursor_name

Classe 38 -- Exception de routine externe

- external_routine_exception
- containing_sql_not_permitted
- modifying_sql_data_not_permitted
- prohibited_sql_statement_attempted
- reading_sql_data_not_permitted

Classe 39 -- Exception dans l'appel d'une routine externe

- external_routine_invocation_exception
- invalid_sqlstate_returned
- null_value_not_allowed
- trigger_protocol_violated
- srf_protocol_violated

Classe 3B -- Exception dans un point de retournement

- savepoint_exception
- invalid_savepoint_specification
- invalid_catalog_name

Classe 3D -- Nom de catalogue invalide

- invalid_schema_name

Classe 3F -- Nom de schéma invalide

- transaction_rollback

Classe 40 -- Annulation de transaction

- transaction_integrity_constraint_violation
- serialization_failure
- statement_completion_unknown
- deadlock_detected

Classe 42 -- Erreur de syntaxe ou violation de règle d'accès

- syntax_error_or_access_rule_violation
- syntax_error
- insufficient_privilege
- cannot_coerce
- grouping_error
- windowing_error
- invalid_recursion
- invalid_foreign_key
- invalid_name
- name_too_long
- reserved_name
- datatype_mismatch
- indeterminate_datatype
- wrong_object_type
- undefined_column
- undefined_function
- undefined_table
- undefined_parameter
- undefined_object
- duplicate_column
- duplicate_cursor
- duplicate_database
- duplicate_function
- duplicate_prepared_statement
- duplicate_schema
- duplicate_table
- duplicate_alias
- duplicate_object
- ambiguous_column
- ambiguous_function
- ambiguous_parameter
- ambiguous_alias
- invalid_column_reference
- invalid_column_definition
- invalid_cursor_definition
- invalid_database_definition
- invalid_function_definition
- invalid_prepared_statement_definition
- invalid_schema_definition
- invalid_table_definition
- invalid_object_definition

Classe 44 -- Violation de WITH CHECK OPTION

- with_check_option_violation

Classe 53 -- Ressources insuffisantes

- insufficient_resources
- disk_full
- out_of_memory
- too_many_connections

Classe 54 -- Limite du programme dépassée

- program_limit_exceeded
- statement_too_complex
- too_many_columns
- too_many_arguments

Classe 55 -- L'objet n'est pas l'état prérequis

- object_not_in_prerequisite_state
- object_in_use
- cant_change_runtime_param
- lock_not_available

Classe 57 -- Intervention d'un opérateur

- operator_intervention
- query_canceled
- admin_shutdown
- crash_shutdown
- cannot_connect_now

Class 58 -- Erreur système (erreurs externes à PostgreSQL™)

- io_error
- undefined_file
- duplicate_file

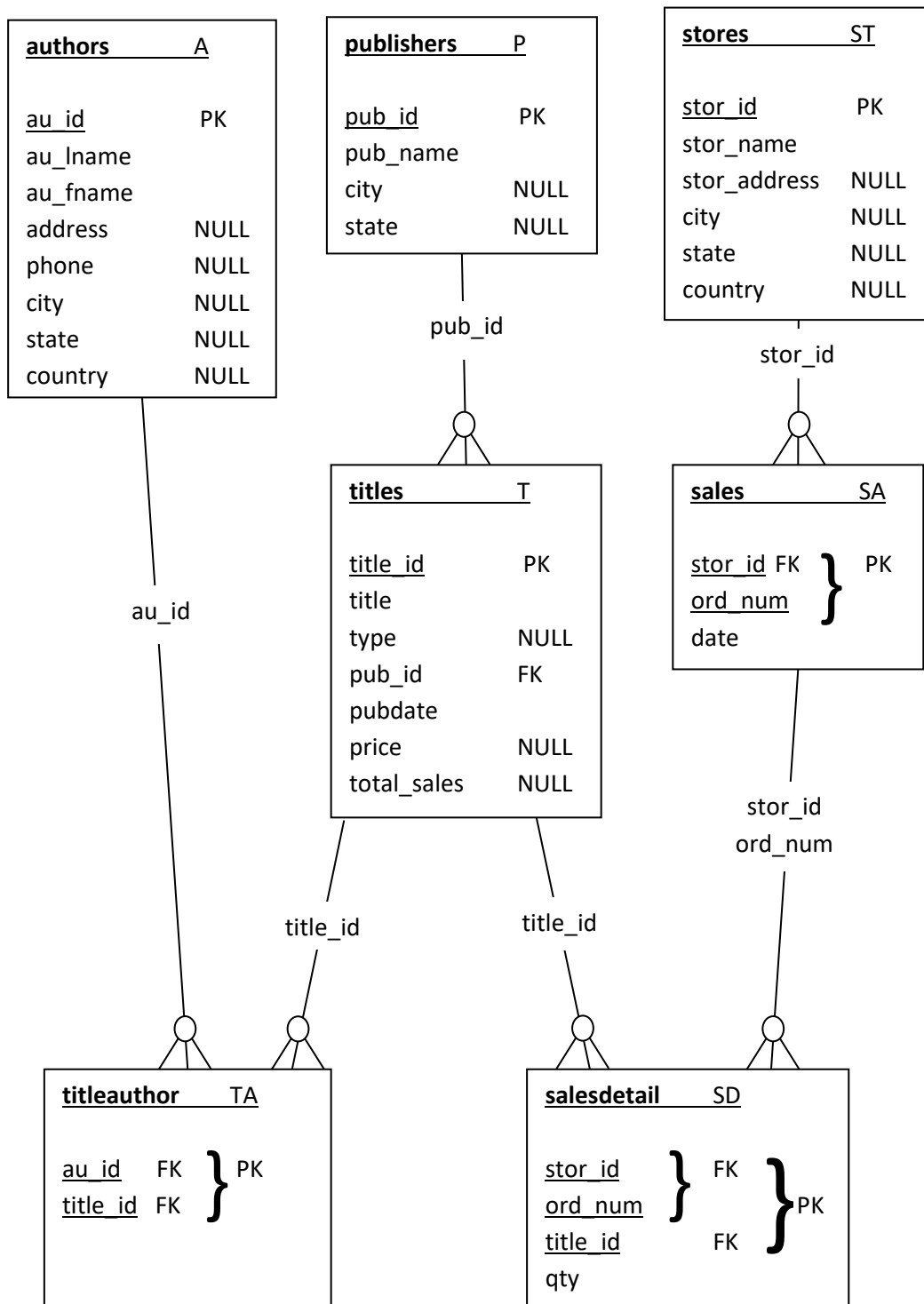
Classe F0 -- Erreur dans le fichier de configuration

- config_file_error
- lock_file_exists

Classe P0 -- Erreur PL/pgSQL

- plpgsql_error
- raise_exception
- no_data_found
- too_many_rows
- internal_error
- data_corrupted
- index_corrupted

7.e. Diagramme de Bachman de la base de donnée pubs2



8. Références

Ce texte collationne des informations provenant de diverses origines :

- L'ancien syllabus de SQL de Benoît Dupuis
- Le manuel utilisateur de PostgreSQL
- La documentation en ligne de Java
- Wikipedia
- Un supplément de texte original de Donatien Grolaux
- Une mise-à-jour de Christophe Damas et Stéphanie Ferneeuw