Module S1

Premiers SELECTs

22.09

Dalibo SCOP

https://dalibo.com/formations

Premiers SELECTs

Module S1

TITRE : Premiers SELECTs
SOUS-TITRE : Module S1

REVISION: 22.09
DATE: 02 septembre 2022

Remerciements : Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé à :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale : Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme... Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com !

Table des Matières

1 INTRODUCTION ET PREMIERS SELECT

1.1 PRÉAMBULE

- Qu'est-ce que le standard SQL ?
- Comment lire des données
- Quels types de données sont disponibles ?

Ce module a pour but de présenter le standard SQL. Un module ne permet pas de tout voir, aussi ce module se concentrera sur la lecture de données déjà présentes en base. Cela permet d'aborder aussi la question des types de données disponibles.

1.1.1 MENU

- Principes d'une base de données
- Premières requêtes
- Connaître les types de données

1.1.2 OBJECTIFS

- Comprendre les principes
- Écrire quelques requêtes en lecture
- Connaître les différents types de données
  - et quelques fonctions très utiles

1.2 PRINCIPES D'UNE BASE DE DONNÉES

- Base de données
  - ensemble organisé d'informations
- Système de Gestion de Bases de Données
  - acronyme SGBD (DBMS en anglais)
  - programme assurant la gestion et l'accès à une base de données
  - assure la cohérence des données

Si des données sont récoltées, organisées et stockées afin de répondre à un besoin spécifique, alors on parle de base de données. Une base de données peut utiliser différents supports : papier, fichiers informatiques, etc.

Le Système de Gestion de Bases de Données (SGBD), appelé *Database Management System* (DBMS) en anglais, assure la gestion d'une base de données informatisée. Il permet l'accès aux données et assure également la cohérence des données.

1.2.1 TYPE DE BASES DE DONNÉES

- Modèle hiérarchique
- Modèle réseau
- Modèle relationnel
- Modèle objet
- Modèle relationnel-objet
- NoSQL

Au fil des années ont été développés plusieurs modèles de données, que nous allons décrire.

1.2.2 TYPE DE BASES DE DONNÉES (1)

- Modèle hiérarchique
  - structure arborescente
  - redondance des données
- Modèle réseau
  - structure arborescente, mais permettant des associations
  - ex : Bull IDS2 sur GCOS

Les modèles hiérarchiques et réseaux ont été les premiers modèles de données utilisées dans les années 60 sur les mainframes IBM ou Bull. Ils ont été rapidement supplantés par le modèle relationnel car les requêtes étaient dépendantes du modèle de données. Il était nécessaire de connaître les liens entre les différents nœuds de l'arborescence pour concevoir les requêtes. Les programmes sont donc complètement dépendants de la structure de la base de données.

Des recherches souhaitent néanmoins arriver à rendre indépendant la vue logique de l'implémentation physique de la base de données.

1.2.3 TYPE DE BASES DE DONNÉES (2)

- Modèle relationnel
  - basé sur la théorie des ensembles et la logique des prédicats
  - standardisé par la norme SQL
- Modèle objet
  - structure objet
  - pas de standard
- Modèle relationnel-objet
  - le standard SQL ajoute des concepts objets

Le modèle relationnel est issu des travaux du Docteur Edgar F. Codd qu'il a menés dans les laboratoires d'IBM à la fin des années 60. Ses travaux avaient pour but de rendre indépendant le stockage physique de la vue logique de la base de données. Et, mathématicien de formation, il s'est appuyé sur la théorie des ensembles et la logique des prédicats pour établir les fondements des bases de données relationnelles. Pour manipuler les données de façon ensembliste, le Dr Codd a mis au point le point langage SQL. Ce langage est à l'origine du standard SQL qui a émergé dans les années 80 et qui a rendu le modèle relationnel très populaire.

Le modèle objet est, quant à lui, issu de la mouvance autour des langages objets. Du fait de l'absence d'un standard avéré, le modèle objet n'a jamais été populaire et est toujours resté dans l'ombre du modèle relationnel.

Le modèle relationnel a néanmoins été étendu par la norme SQL:1999 pour intégrer des fonctionnalités objets. On parle alors de modèle relationnel-objet. PostgreSQL en est un exemple, c'est un SGBDRO (Système de Gestion de Bases de Données Relationnel-Objet).

1.2.4 TYPE DE BASES DE DONNÉES (3)

- NoSQL : *Not only SQL*
  - pas de norme de langage de requête
  - clé-valeur (Redis, Riak)
  - graphe (Neo4J)
  - document (MongoDB, CouchDB)
  - orienté colonne (HBase)
- Rapprochement relationnel/NoSQL
  - PostgreSQL permet de stocker des documents (JSON, XML)

Les bases NoSQL sont une famille de bases de données qui répondent à d'autres besoins et contraintes que les bases relationnelles. Les bases NoSQL sont souvent des bases

« sans schéma », la base ne vérifiant plus l'intégrité des données selon des contraintes définies dans le modèle de données. Chaque base de ce segment dispose d'un langage de requête spécifique, qui n'est pas normé. Une tentative de standardisation, débutée en 2011, n'a d'ailleurs abouti à aucun résultat.

Ce type de base offre souvent la possibilité d'offrir du *sharding* simple à mettre en œuvre. Le sharding consiste à répartir les données physiquement sur plusieurs serveurs. Certaines technologies semblent mieux marcher que d'autres de ce point de vue là. En contrepartie, la durabilité des données n'est pas assurée, au contraire d'une base relationnelle qui assure la durabilité dès la réponse à un `COMMIT`.

Exemple de requête SQL :

```sql
SELECT person, SUM(score), AVG(score), MIN(score), MAX(score), COUNT(*)
FROM demo
WHERE score > 0 AND person IN('bob','jake')
GROUP BY person;
```

La même requête, pour MongoDB :

```
db.demo.group({
    "key": {
        "person": true
    },
    "initial": {
        "sumscore": 0,
        "sumforaverageaveragescore": 0,
        "countforaverageaveragescore": 0,
        "countstar": 0
    },
    "reduce": function(obj, prev) {
        prev.sumscore = prev.sumscore + obj.score - 0;
        prev.sumforaverageaveragescore += obj.score;
        prev.countforaverageaveragescore++;
        prev.minimumvaluescore = isNaN(prev.minimumvaluescore) ? obj.score :
            Math.min(prev.minimumvaluescore, obj.score);
        prev.maximumvaluescore = isNaN(prev.maximumvaluescore) ? obj.score :
            Math.max(prev.maximumvaluescore, obj.score);
        if (true != null) if (true instanceof Array) prev.countstar +=
            true.length;
        else prev.countstar++;
    },
    "finalize": function(prev) {
        prev.averagescore = prev.sumforaverageaveragescore /
            prev.countforaverageaveragescore;
        delete prev.sumforaverageaveragescore;
```

```
        delete prev.countforaverageaveragescore;
    },
    "cond": {
        "score": {
            "$gt": 0
        },
        "person": {
            "$in": ["bob", "jake"]
        }
    }
});
```

Un des avantages de ces technologies, c'est qu'un modèle clé-valeur permet facilement d'utiliser des algorithmes de type MapReduce : diviser le problème en sous-problèmes traités parallèlement par différents nœuds (phase Map), puis synthétisés de façon centralisée (phase Reduce).

Les bases de données relationnelles ne sont pas incompatibles avec Map Reduce en soit. Simplement, le langage SQL étant déclaratif, il est conceptuellement opposé à la description fine des traitements qu'on doit réaliser avec MapReduce. C'est (encore une fois) le travail de l'optimiseur d'être capable d'effectuer ce genre d'opérations : la parallélisation (répartition d'une tâche sur plusieurs processeurs) est possible dans certains cas avec PostgreSQL.

1.2.5 MODÈLE RELATIONNEL

- Indépendance entre la vue logique et la vue physique
  - le SGBD gère lui-même le stockage physique
- Table ou *relation*
- Un ensemble de tables représente la vue logique

Le modèle relationnel garantit l'indépendance entre la vue logique et la vue physique. L'utilisateur ne se préoccupe que des objets logiques (pour lire ou écrire des enregistrements), et le SGBD traduit la demande exprimée avec des objets logiques en actions à réaliser sur des objets physiques.

Les objets logiques sont appelés des relations. Ce sont généralement les tables, mais il existe d'autres objets qui sont aussi des relations (les vues par exemple, mais aussi les index et les séquences).

1.2.6 CARACTÉRISTIQUES DU MODÈLE RELATIONNEL

- Théorie des ensembles
- Logique des prédicats
- Logique 3 états

Le modèle relationnel se base sur la théorie des ensembles. Chaque relation contient un ensemble de données et ces différents ensembles peuvent se joindre suivant certaines conditions.

La logique des prédicats est un sous-ensemble de la théorie des ensembles. Elle sert à exprimer des formules logiques qui permettent de filtrer les ensembles de départ pour créer de nouveaux ensembles (autrement dit, filtrer les enregistrements d'une relation).

Cependant, tout élément d'un enregistrement n'est pas forcément connu à un instant t. Les filtres et les jointures doivent donc gérer trois états lors d'un calcul de prédicat : vrai, faux ou inconnu.

1.2.7 ACID

- **Atomicité** (*Atomic*)
- **Cohérence** (*Consistent*)
- **Isolation** (*Isolated*)
- **Durabilité** (*Durable*)

Les propriétés ACID (acronyme de *Atomic Consistent Isolated Durable*) sont le fondement même de toute base de données. Il s'agit de quatre règles fondamentales que toute transaction doit respecter :

- **A** : Une transaction est entière : « tout ou rien ».
- **C** : Une transaction amène la base d'un état stable à un autre.
- **I** : Les transactions n'agissent pas les unes sur les autres.
- **D** : Une transaction validée provoque des changements permanents.

Les propriétés ACID sont quatre propriétés essentielles d'un sous-système de traitement de transactions d'un système de gestion de base de données. On considère parfois que seuls les SGBD qui respectent ces quatre propriétés sont dignes d'être considérées comme des bases de données *relationnelles*. Les SGBD de la famille des NoSQL (MongoDB, Cassandra, BigTable...) sont en effet des bases de données, mais ne respectent pas la Cohérence. Elles sont cohérentes à terme, ou en anglais *eventually consistent*, mais la cohérence en fin de transaction n'est pas garantie.

1.2.8 LANGAGE SQL

- Norme ISO 9075
  - dernière version stable : 2016
- Langage déclaratif
  - on décrit le résultat et pas la façon de l'obtenir
  - comme Prolog
- Traitement ensembliste
  - par opposition au traitement procédural
  - « on effectue des opérations sur des relations pour obtenir des relations »

Le langage SQL a été normalisé par l'ANSI en 1986 et est devenu une norme ISO internationale en 1987. Elle a subi plusieurs évolutions dans le but d'ajouter des fonctionnalités correspondant aux attentes de l'industrie logicielle. Parmi ces améliorations, notons l'intégration de quelques fonctionnalités objets pour le modèle relationnel-objet.

1.2.9 SQL EST UN LANGAGE

- Langage
  - règles d'écriture
  - règles de formatage
  - commentaires
- Améliore la lisibilité d'une requête

Il n'y a pas de règles établies concernant l'écriture de requêtes SQL. Il faut néanmoins avoir à l'esprit qu'il s'agit d'un langage à part entière et, au même titre que ce qu'un développeur fait avec n'importe quel code source, il convient de l'écrire de façon lisible.

1.2.10 RECOMMANDATIONS D'ÉCRITURE ET DE FORMATAGE

- Écriture
  - mots clés SQL en MAJUSCULES
  - identifiants de colonnes/tables en minuscule
- Formatage
  - dissocier les éléments d'une requête
  - un prédicat par ligne
  - indentation

Quelle est la requête la plus lisible ?

celle-ci ?

```
select  groupeid,datecreationitem from itemagenda where typeitemagenda = 5  and
groupeid in(12225,12376) and datecreationitem > now() order by groupeid,
datecreationitem ;
```

ou celle-ci ?

```
SELECT groupeid, datecreationitem
  FROM itemagenda
 WHERE typeitemagenda = 5
   AND groupeid IN (12225,12376)
   AND datecreationitem > now()
 ORDER BY groupeid, datecreationitem;
```

Cet exemple est tiré du forum postgresql.fr[2] .

1.2.11 COMMENTAIRES

- Commentaire sur le reste de la ligne

```
-- commentaire
```

- Commentaire dans un bloc

```
/* bloc
 */
```

Une requête SQL peut être commentée au même titre qu'un programme standard.

Le marqueur `--` permet de signifier à l'analyseur syntaxique que le reste de la ligne est commenté, il n'en tiendra donc pas compte dans son analyse de la requête.

Un commentaire peut aussi se présenter sous la forme d'un bloc de commentaire, le bloc pouvant occuper plusieurs lignes :

```
/* Ceci est un commentaire
   sur plusieurs
   lignes
*/
```

Aucun des éléments compris entre le marqueur de début de bloc `/*` et le marqueur de fin de bloc `*/` ne sera pris en compte. Certains SGBDR propriétaires utilisent ces commentaires pour y placer des informations (appelées `hints` sur Oracle) qui permettent d'influencer le comportement de l'optimiseur, mais PostgreSQL ne possède pas ce genre de mécanisme.

[2] https://forum.postgresql.fr/viewtopic.php?id=2610

1.2.12 LES 4 TYPES D'ORDRES SQL

- DDL
  - Data Definition Language
  - définit les structures de données
- DML
  - Data Manipulation Language
  - manipule les données
- DCL
  - Data Control Language
  - contrôle l'accès aux données
- TCL
  - Transaction Control Language
  - contrôle les transactions
  - implicites si « autocommit »

Le langage SQL est divisé en quatre sous-ensembles qui ont chacun un but différent.

Les ordres DDL (pour `Data Definition Language`) permettent de définir les structures de données. On retrouve les ordres suivants :

- `CREATE` : crée un objet
- `ALTER` : modifie la définition d'un objet
- `DROP` : supprime un objet
- `TRUNCATE` : vide un objet
- `COMMENT` : ajoute un commentaire sur un objet

Les ordres DML (pour `Data Manipulation Language`) permettent l'accès et la modification des données. On retrouve les ordres suivants :

- `SELECT` : lit les données d'une ou plusieurs tables
- `INSERT` : ajoute des données dans une table
- `UPDATE` : modifie les données d'une table
- `DELETE` : supprime les données d'une table

Les ordres DCL (pour `Data Control Language`) permettent de contrôler l'accès aux données. Ils permettent plus précisément de donner ou retirer des droits à des utilisateurs ou des groupes sur les objets de la base de données :

- `GRANT` : donne un droit d'accès à un rôle sur un objet
- `REVOKE` : retire un droit d'accès d'un rôle sur un objet

Enfin, les ordres TCL (pour `Transaction Control Language`) permettent de contrôler les transactions :

- `BEGIN` : ouvre une transaction
- `COMMIT` : valide les traitements d'une transaction
- `ROLLBACK` : annule les traitements d'une transaction
- `SAVEPOINT` : crée un point de reprise dans une transaction
- `SET TRANSACTION` : modifie les propriétés d'une transaction en cours

Les ordres `BEGIN` et `COMMIT` sont souvent implicites dans le cas d'ordres isolés, si l'« auto-commit » est activé. Vous devez entrer donc manuellement `BEGIN ;` / `COMMIT ;` pour faire des transactions de plus d'un ordre. C'est en fait dépendant de l'outil client, et `psql` a un paramètre `autocommit` à `on` par défaut. Mais ce n'est pas forcément le cas sur votre configuration précise et le défaut peut être inversé sur d'autres bases de données (notamment Oracle).

Le `ROLLBACK` est implicite en cas de sortie brutale.

Noter que, contrairement à d'autres bases (et surtout Oracle), PostgreSQL n'effectue pas de `COMMIT` implicite sur certaines opérations : les ordres `CREATE TABLE`, `DROP TABLE`, `TRUNCATE TABLE`... sont transactionnels, n'effectuent aucun `COMMIT` et peuvent être annulés par `ROLLBACK`.

1.3 LECTURE DE DONNÉES

- Ordre `SELECT`
  - lecture d'une ou plusieurs tables
  - ou appel de fonctions

La lecture des données se fait via l'ordre `SELECT`. Il permet de récupérer des données d'une ou plusieurs tables (il faudra dans ce cas joindre les tables). Il permet aussi de faire appel à des fonctions stockées en base.

1.3.1 SYNTAXE DE SELECT

```
SELECT expressions_colonnes
 [ FROM elements_from ]
 [ WHERE predicats ]
 [ ORDER BY expressions_orderby ]
 [ LIMIT limite ]
 [ OFFSET offset ];
```

L'ordre `SELECT` est composé de différents éléments dont la plupart sont optionnels. L'exemple de syntaxe donné ici n'est pas complet.

La syntaxe complète de l'ordre `SELECT` est disponible dans le manuel de PostgreSQL[3] .

1.3.2 LISTE DE SÉLECTION

- Description du résultat de la requête
  - colonnes retournées
  - renommage
  - dédoublonnage

La liste de sélection décrit le format de la table virtuelle qui est retournée par l'ordre `SELECT`. Les types de données des colonnes retournées seront conformes au type des éléments donnés dans la liste de sélection.

1.3.3 COLONNES RETOURNÉES

- Liste des colonnes retournées
  - expression
  - séparées par une virgule
- Expression
  - constante
  - référence de colonne :

    `table.colonne`
- opération sur des colonnes et/ou des constantes

La liste de sélection décrit le format de la table virtuelle qui est retournée par l'ordre `SELECT`. Cette liste est composée d'expressions séparées par une virgule.

[3] https://docs.postgresql.fr/current/sql-select.html

Chaque expression peut être une simple constante, peut faire référence à des colonnes d'une table lue par la requête, et peut être un appel à une fonction.

Une expression peut être plus complexe. Par exemple, elle peut combiner plusieurs constantes et/ou colonnes à l'aide d'opérations. Parmi les opérations les plus classiques, les opérateurs arithmétiques classiques sont utilisables pour les données numériques. L'opérateur de concaténation permet de concaténer des chaînes de caractères.

L'expression d'une colonne peut être une constante :

```
SELECT 1;
 ?column?
----------
        1
(1 row)
```

Elle peut aussi être une référence à une colonne d'une table :

```
SELECT appellation.libelle
  FROM appellation;
```

Comme il n'y a pas d'ambiguïté avec la colonne `libelle`, la référence de la colonne `appellation.libelle` peut être simplifiée en `libelle` :

```
SELECT libelle
  FROM appellation;
```

Le SGBD saura déduire la table et la colonne mises en œuvre dans cette requête. Il faudra néanmoins utiliser la forme complète `table.colonne` si la requête met en œuvre des tables qui possèdent des colonnes qui portent des noms identiques.

Une requête peut sélectionner plusieurs colonnes. Dans ce cas, les expressions de colonnes sont définies sous la forme d'une liste dont chaque élément est séparé par une virgule :

```
SELECT id, libelle, region_id
  FROM appellation;
```

Le joker `*` permet de sélectionner l'ensemble des colonnes d'une table, elles apparaitront dans leur ordre physique (attention si l'ordre change !) :

```
SELECT *
  FROM appellation;
```

Si une requête met en œuvre plusieurs tables, on peut choisir de retourner toutes les colonnes d'une seule table :

```
SELECT appellation.*
  FROM appellation;
```

Enfin, on peut récupérer un tuple entier de la façon suivante :

```
SELECT appellation
  FROM appellation;
```

Une expression de colonne peut également être une opération, par exemple une addition :

```
SELECT 1 + 1;
 ?column?
----------
        2
(1 row)
```

Ou une soustraction :

```
SELECT annee, nombre - 10
  FROM stock;
```

1.3.4 ALIAS DE COLONNE

- Renommage
  - ou alias
  - `AS` :

  `expression AS alias`
- le résultat portera le nom de l'alias

Afin de pouvoir nommer de manière adéquate les colonnes du résultat d'une requête `SELECT`, le mot clé `AS` permet de définir un alias de colonne. Cet alias sera utilisé dans le résultat pour nommer la colonne en sortie :

```
SELECT 1 + 1 AS somme;
 somme
-------
     2
(1 row)
```

Cet alias n'est pas utilisable dans le reste de la requête (par exemple dans la clause `WHERE`).

1.3.5 DÉDOUBLONNAGE DES RÉSULTATS

```
SELECT DISTINCT expressions_colonnes...
```

- Dédoublonnage des résultats avant de les retourner
  - à ne pas utiliser systématiquement

Par défaut, `SELECT` retourne tous les résultats d'une requête. Parfois, des doublons peuvent se présenter dans le résultat. La clause `DISTINCT` permet de les éviter en réalisant un dédoublonnage des données avant de retourner le résultat de la requête.

Il faut néanmoins faire attention à l'utilisation systématique de la clause `DISTINCT`. En effet, elle entraîne un tri systématique des données juste avant de retourner les résultats de la requête, ce qui va consommer de la ressource mémoire, voire de la ressource disque si le volume de données à trier est important. De plus, cela va augmenter le temps de réponse de la requête du fait de cette opération supplémentaire.

En règle générale, la clause `DISTINCT` devient inutile lorsqu'elle doit trier un ensemble qui contient des colonnes qui sont déjà uniques. Si une requête récupère une clé primaire, les données sont uniques par définition. Le `SELECT DISTINCT` sera alors transformé en simple `SELECT`.

1.3.6 DÉRIVATION

- SQL permet de dériver les valeurs des colonnes
  - opérations arithmétiques : `+`, `-`, `/`, `*`
  - concaténation de chaînes : `||`
  - appel de fonction

Les constantes et valeurs des colonnes peuvent être dérivées selon le type des données manipulées.

Les données numériques peuvent être dérivées à l'aide des opérateurs arithmétiques standards : `+`, `-`, `/`, `*`. Elles peuvent faire l'objet d'autres calculs à l'aide de fonctions internes et de fonctions définies par l'utilisateur.

La requête suivante permet de calculer le volume total en litres de vin disponible dans le stock du caviste :

```
SELECT SUM(c.contenance * s.nombre) AS volume_total
  FROM stock s
  JOIN contenant c
    ON (contenant_id=c.id);
```

Les données de type chaînes de caractères peuvent être concaténées à l'aide de l'opérateur dédié `||`. Cet opérateur permet de concaténer deux chaînes de caractères mais également des données numériques avec une chaîne de caractères.

Dans la requête suivante, l'opérateur de concaténation est utilisé pour ajouter l'unité. Le résultat est ainsi implicitement converti en chaîne de caractères.

```
SELECT SUM(c.contenance * s.nombre) || ' litres' AS volume_total
  FROM stock AS s
  JOIN contenant AS c
    ON (contenant_id=c.id);
```

De manière générale, il n'est pas recommandé de réaliser les opérations de formatage des données dans la base de données. La base de données ne doit servir qu'à récupérer les résultats, le formatage étant assuré par l'application.

Différentes fonctions sont également applicables aux chaînes de caractères, de même qu'aux autres types de données.

1.3.7 FONCTIONS UTILES

- Fonctions sur données temporelles :
  - date et heure courante : `now()`
  - âge : `age(timestamp)`
  - extraire une partie d'une date : `extract( 'year' FROM timestamp)`
  - ou `date_part('Y',timestamp)`
- Fonctions sur données caractères :
  - longueur d'une chaîne de caractère : `char_length(chaine)`
- Compter les lignes : `count(*)`

Parmi les fonctions les plus couramment utilisés, la fonction `now()` permet d'obtenir la date et l'heure courante. Elle ne prend aucun argument. Elle est souvent utilisée, notamment pour affecter automatiquement la valeur de l'heure courante à une colonne.

La fonction `age(timestamp)` permet de connaître l'âge d'une date par rapport à la date courante.

La fonction `char_length(varchar)` permet de connaître la longueur d'une chaîne de caractère.

Enfin, la fonction `count(*)` permet de compter le nombre de lignes. Il s'agit d'une fonction d'agrégat, il n'est donc pas possible d'afficher les valeurs d'autres colonnes sans faire appel aux capacités de regroupement des lignes de SQL.

Exemples

Affichage de l'heure courante :

```
SELECT now();
              now
-------------------------------
 2017-08-29 14:45:17.213097+02
```

Affichage de l'âge du 1er janvier 2000 :

```
SELECT age(date '2000-01-01');
          age
------------------------
 17 years 7 mons 28 days
```

Affichage de la longueur de la chaîne "Dalibo" :

```
SELECT char_length('Dalibo');
 char_length
-------------
           6
```

Affichage du nombre de lignes de la table `vin` :

```
SELECT count(*) FROM vin;
 count
-------
  6067
```

1.3.8 CLAUSE FROM

```
FROM expression_table [, expression_table ...]
```

- Description des tables mises en œuvre dans la requête
  - une seule table
  - plusieurs tables jointes
  - sous-requête

La clause `FROM` permet de lister les tables qui sont mises en œuvres dans la requêtes `SELECT`. Il peut s'agir d'une table physique, d'une vue ou d'une sous-requête. Le résultat de leur lecture sera une table du point de vue de la requête qui la met en œuvre.

Plusieurs tables peuvent être mises en œuvre, généralement dans le cadre d'une jointure.

1.3.9 ALIAS DE TABLE

- Mot-clé `AS`
  - optionnel :

`reference_table alias`

- La table sera ensuite référencée par l'alias

```
reference_table [AS] alias
reference_table AS alias (alias_colonne1, ...)
```

De la même façon qu'on peut créer des alias de colonnes, on peut créer des alias de tables. La table sera ensuite référencée uniquement par cet alias dans la requête. Elle ne pourra plus être référencée par son nom réel. L'utilisation du nom réel provoquera d'ailleurs une erreur.

Le mot clé `AS` permet de définir un alias de table. Le nom réel de la table se trouve à gauche, l'alias se trouve à droite. L'exemple suivant définie un alias `reg` sur la table `region` :

```
SELECT id, libelle
  FROM region AS reg;
```

Le mot clé `AS` est optionnel :

```
SELECT id, libelle
  FROM region reg;
```

La requête suivante montre l'utilisation d'un alias pour les deux tables mises en œuvre dans la requête. La table `stock` a pour alias `s` et la table `contenant` a pour alias `c`. Les deux tables possèdent toutes les deux une colonnes `id`, ce qui peut poser une ambiguïté dans la clause de jointure (`ON (contenant_id=c.id)`). La condition de jointure portant sur la colonne `contenant_id` de la table `stock`, son nom est unique et ne porte pas à ambiguïté. La condition de jointure porte également sur la colonne `id` de table `contenant`, il faut préciser le nom complet de la colonne en utilisant le préfixe `c` pour la nommer : `c.id`.

```
SELECT SUM(c.contenance * s.nombre) AS volume_total
  FROM stock s
  JOIN contenant c
    ON (contenant_id=c.id);
```

Enfin, la forme `reference_table AS alias (alias_colonne1, ...)` permet de définir un alias de table et définir par la même occasion des alias de colonnes. Cette forme est peu recommandé car les alias de colonnes dépendent de l'ordre physique de ces colonnes. Cet ordre peut changer dans le temps et donc amener à des erreurs :

```
SELECT id_region, nom_region
  FROM region AS reg (id_region, nom_region);
```

1.3.10 NOMMAGE DES OBJETS

- Noms d'objets convertis en minuscules
  - `Nom_Objet` devient `nom_objet`
  - certains nécessitent l'emploi de majuscules
- Le guillemet double `"` conserve la casse
  - `"Nom_Objet"`

Avec PostgreSQL, les noms des objets sont automatiquement convertis en minuscule, sauf s'ils sont englobés entre des guillemets doubles. Si jamais ils sont créés avec une casse mixte en utilisant les guillemets doubles, chaque appel à cet objet devra utiliser la bonne casse et les guillemets doubles. Il est donc conseillé d'utiliser une notation des objets ne comprenant que des caractères minuscules.

Il est aussi préférable de ne pas utiliser d'accents ou de caractères exotiques dans les noms des objets.

1.3.11 CLAUSE WHERE

- Permet d'exprimer des conditions de filtrage
  - prédicats
- Un prédicat est une opération logique
  - renvoie vrai ou faux
- La ligne est présente dans le résultat
  - si l'expression logique des prédicats est vraie

La clause `WHERE` permet de définir des conditions de filtrage des données. Ces conditions de filtrage sont appelées des prédicats.

Après le traitement de la clause `FROM`, chaque ligne de la table virtuelle dérivée est vérifiée avec la condition de recherche. Si le résultat de la vérification est positif (true), la ligne est conservée dans la table de sortie, sinon (c'est-à-dire si le résultat est faux ou nul) la ligne est ignorée.

La condition de recherche référence typiquement au moins une colonne de la table générée dans la clause `FROM` ; ceci n'est pas requis mais, dans le cas contraire, la clause `WHERE` n'aurait aucune utilité.

1.3.12 EXPRESSION ET OPÉRATEURS DE PRÉDICATS

- Comparaison
  - `=`, `<`, `>`, `<=`, `>=`, `<>`
- Négation
  - `NOT`

```
expression operateur_comparaison expression
```

Un prédicat est composé d'une expression qui est soumise à un opérateur de prédicat pour être éventuellement comparé à une autre expression. L'opérateur de prédicat retourne alors `true` si la condition est vérifiée ou `false` si elle ne l'est pas ou `NULL` si son résultat ne peut être calculé.

Les opérateurs de comparaison sont les opérateurs de prédicats les plus souvent utilisés. L'opérateur d'égalité `=` peut être utilisé pour vérifier l'égalité de l'ensemble des types de données supportés par PostgreSQL. Il faudra faire attention à ce que les données comparées soient de même type.

L'opérateur `<>` signifie « pas égal à » et peut aussi s'écrire `!=`.

L'opérateur `NOT` est une négation. Si un prédicat est vrai, l'opérateur `NOT` retournera faux. À l'inverse, si un prédicat est faux, l'opérateur `NOT` retournera vrai. La clause `NOT` se place devant l'expression entière.

Exemples

Sélection de la région dont l'identifiant est égal à 3 (et ensuite différent de 3) :

```
SELECT *
  FROM region
 WHERE id = 3;

SELECT *
  FROM region
 WHERE NOT id = 3;
```

1.3.13 COMBINER DES PRÉDICATS

- OU logique
  - `predicat OR predicat`
- ET logique
  - `predicat AND predicat`

Les opérateurs logiques `OR` et `AND` permettent de combiner plusieurs prédicats dans la clause `WHERE`.

L'opérateur `OR` est un OU logique. Il retourne vrai si au moins un des deux prédicats combinés est vrai. L'opérateur `AND` est un ET logique. Il retourne vrai si et seulement si les deux prédicats combinés sont vrais.

Au même titre qu'une multiplication ou une division sont prioritaires sur une addition ou une soustraction dans un calcul, l'évaluation de l'opérateur `AND` est prioritaire sur celle de l'opérateur `OR`. Et, tout comme dans un calcul, il est possible de protéger les opérations prioritaires en les encadrant de parenthèses.

Exemples

Dans le stock, affiche les vins dont le nombre de bouteilles est inférieur à 2 ou supérieur à 16 :

```
SELECT *
  FROM stock
 WHERE nombre < 2
    OR nombre > 16;
```

1.3.14 CORRESPONDANCE DE MOTIF

- Comparaison de motif

  `chaine LIKE motif ESCAPE 'c'`
- `%` : toute chaîne de 0 à plusieurs caractères
  - `_` : un seul caractère
- Expression régulière POSIX

  `chaine ~ motif`

L'opérateur `LIKE` permet de réaliser une recherche simple sur motif. La chaîne exprimant le motif de recherche peut utiliser deux caractères joker : `_` et `%`. Le caractère `_` prend la place d'un caractère inconnu, qui doit toujours être présent. Le caractère `%` est un joker qui permet d'exprimer que PostgreSQL doit trouver entre 0 et plusieurs caractères.

Exploiter la clause `LIKE` avec un motif sans joker ne présente pas d'intérêt. Il est préférable dans ce cas d'utiliser l'opérateur d'égalité.

Le mot clé `ESCAPE 'c'` permet de définir un caractère d'échappement pour protéger les caractères `_` et `%` qui font légitimement partie de la chaîne de caractère du motif évalué. Lorsque PostgreSQL rencontre le caractère d'échappement indiqué, les caractères `_` et `%` seront évalués comme étant les caractères `_` et `%` et non comme des jokers.

L'opérateur `LIKE` dispose d'une déclinaison qui n'est pas sensible à la casse. Il s'agit de l'opérateur `ILIKE`.

Exemples

Création d'un jeu d'essai :

```
CREATE TABLE motif (chaine varchar(30));
INSERT INTO motif (chaine) VALUES ('Durand'), ('Dupont'), ('Dupond'),
    ('Dupon'), ('Dupuis');
```

Toutes les chaînes commençant par la suite de caractères `Dur` :

```
SELECT * FROM motif WHERE chaine LIKE 'Dur%';
 chaine
--------
 Durand
```

Toutes les chaînes terminant par `d` :

```
SELECT * FROM motif WHERE chaine LIKE '%d';
 chaine
--------
 Durand
 Dupond
```

Toutes les chaînes qui commencent par `Dupon` suivi d'un caractère inconnu. La chaîne `Dupon` devrait être ignorée :

```
SELECT * FROM motif WHERE chaine LIKE 'Dupon_';
   chaine
------------
 Dupont
 Dupond
```

1.3.15 LISTES ET INTERVALLES

- Liste de valeurs

```
expression IN (valeur1 [, ...])
```

- Chevauchement d'intervalle de valeurs

```
expression BETWEEN expression AND expression
```

- Chevauchement d'intervalle de dates

```
(date1, date2) OVERLAPS (date3, date4)
```

La clause `IN` permet de vérifier que l'expression de gauche est égale à une valeur présente dans l'expression de droite, qui est une liste d'expressions. La négation peut être utilisée en utilisant la construction `NOT IN`.

L'opérateur `BETWEEN` permet de vérifier que la valeur d'une expression est comprise entre deux bornes. Par exemple, l'expression `valeur BETWEEN 1 AND 10` revient à exprimer la condition suivante : `valeur >= 1 AND valeur<= 10`. La négation peut être utilisée en utilisant la construction `NOT BETWEEN`.

Exemples

Recherche les chaînes qui sont présentes dans la liste `IN` :

```
SELECT * FROM motif WHERE chaine IN ('Dupont', 'Dupond', 'Ducobu');
 chaine
--------
 Dupont
 Dupond
```

1.3.16 TRIS

- SQL ne garantit pas l'ordre des résultats
  - tri explicite requis
- Tris des lignes selon des expressions

```
ORDER BY expression  [ ASC | DESC | USING opérateur ]
                     [ NULLS { FIRST | LAST } ] [, ...]
```

- ordre du tri : `ASC` ou `DESC`
  - placement des valeurs `NULL` : `NULLS FIRST` ou `NULLS LAST`
  - ordre de tri des caractères : `COLLATE collation`

La clause `ORDER BY` permet de trier les lignes du résultat d'une requête selon une ou plusieurs expressions combinées.

L'expression la plus simple est le nom d'une colonne. Dans ce cas, les lignes seront triées selon les valeurs de la colonne indiquée, et par défaut dans l'ordre ascendant, c'est-à-dire

de la valeur la plus petite à la plus grande pour une donnée numérique ou temporelle, et dans l'ordre alphabétique pour une donnée textuelle.

Les lignes peuvent être triées selon une expression plus complexe, par exemple en dérivant la valeur d'une colonne.

L'ordre de tri peut être modifié à l'aide de la clause `DESC` qui permet un tri dans l'ordre descendant, donc de la valeur la plus grande à la plus petite (ou alphabétique inverse le cas échéant).

La clause `NULLS` permet de contrôler l'ordre d'apparition des valeurs `NULL`. La clause `NULLS FIRST` permet de faire apparaître d'abord les valeurs `NULL` puis les valeurs non `NULL` selon l'ordre de tri. La clause `NULLS LAST` permet de faire apparaître d'abord les valeurs non `NULL` selon l'ordre de tri suivies par les valeurs NULL. Si cette clause n'est pas précisée, alors PostgreSQL utilise implicitement `NULLS LAST` dans le cas d'un tri ascendant (`ASC`, par défaut) ou `NULLS FIRST` dans le cas d'un tri descendant (`DESC`, par défaut).

Exemples

Tri de la table `region` selon le nom de la région :

```
SELECT *
  FROM region
 ORDER BY libelle;
```

Tri de la table `stock` selon le nombre de bouteille, dans l'ordre décroissant :

```
SELECT *
  FROM stock
 ORDER BY nombre DESC;
```

Enfin, la clause `COLLATE` permet d'influencer sur l'ordre de tri des chaînes de caractères.

1.3.17 LIMITER LE RÉSULTAT

- Obtenir des résultats à partir de la ligne n
  - `OFFSET n`
- Limiter le nombre de lignes à n lignes
  - `FETCH {FIRST | NEXT} n ROWS ONLY`
  - `LIMIT n`
- Opérations combinables
  - `OFFSET` doit apparaitre avant `FETCH`
- Peu d'intérêt sur des résultats non triés

La clause `OFFSET` permet d'exclure les `n` premières lignes du résultat. Toutes les autres lignes sont ramenées.

La clause `FETCH` permet de limiter le résultat d'une requête. La requête retournera au maximum `n` lignes de résultats. Elle en retournera moins, voire aucune, si la requête ne peut ramener suffisamment de lignes. La clause `FIRST` ou `NEXT` est obligatoire mais le choix de l'une ou l'autre n'a aucune conséquence sur le résultat.

La clause `FETCH` est synonyme de la clause `LIMIT`. Mais `LIMIT` est une clause propre à PostgreSQL et quelques autres SGBD. Il est recommandé d'utiliser `FETCH` pour se conformer au standard.

Ces deux opérations peuvent être combinées. La norme impose de faire apparaître la clause `OFFSET` avant la clause `FETCH`. PostgreSQL permet néanmoins d'exprimer ces clauses dans un ordre différent, mais la requête ne pourra pas être portée sur un autre SGBD sans transformation.

Il faut faire attention au fait que ces fonctions ne permettent pas d'obtenir des résultats stables si les données ne sont pas triées explicitement. En effet, le standard SQL ne garantie en aucune façon l'ordre des résultats à moins d'employer la clause `ORDER BY`.

Exemples

La fonction `generate_series` permet de générer une suite de valeurs numériques. Par exemple, une suite comprise entre 1 et 10 :

```
SELECT * FROM generate_series(1, 10);
 generate_series
-----------------
               1
(...)
              10
(10 rows)
```

La clause `FETCH` permet donc de limiter le nombre de lignes du résultats :

```
SELECT * FROM generate_series(1, 10) FETCH FIRST 5 ROWS ONLY;
 generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

La clause `LIMIT` donne un résultat équivalent :

```
SELECT * FROM generate_series(1, 10) LIMIT 5;
 generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

La clause `OFFSET 4` permet d'exclure les quatre premières lignes et de retourner les autres lignes du résultat :

```
SELECT * FROM generate_series(1, 10) OFFSET 4;
 generate_series
-----------------
               5
               6
               7
               8
               9
              10
(6 rows)
```

Les clauses `LIMIT` et `OFFSET` peuvent être combinées pour ramener les deux lignes en excluant les quatre premières :

```
SELECT * FROM generate_series(1, 10) OFFSET 4 LIMIT 2;
 generate_series
-----------------
               5
               6
(2 rows)
```

1.3.18 UTILISER PLUSIEURS TABLES

- Clause `FROM`
  - liste de tables séparées par `,`
- Une table est combinée avec une autre
  - jointure
  - produit cartésien

Il est possible d'utiliser plusieurs tables dans une requête `SELECT`. Lorsque c'est le cas, et sauf cas particulier, on fera correspondre les lignes d'une table avec les lignes d'une autre

table selon certains critères. Cette mise en correspondance s'appelle une jointure et les critères de correspondances s'appellent une condition de jointure.

Si aucune condition de jointure n'est donnée, chaque ligne de la première table est mise en correspondance avec toutes les lignes de la seconde table. C'est un produit cartésien. En général, un produit cartésien n'est pas souhaitable et est généralement le résultat d'une erreur de conception de la requête.

Exemples

Création d'un jeu de données simple :

```
CREATE TABLE mere (id integer PRIMARY KEY, val_mere text);
CREATE TABLE fille (
    id_fille integer PRIMARY KEY,
    id_mere integer REFERENCES mere(id),
    val_fille text
);

INSERT INTO mere (id, val_mere) VALUES (1, 'mere 1');
INSERT INTO mere (id, val_mere) VALUES (2, 'mere 2');

INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (1, 1, 'fille 1');
INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (2, 1, 'fille 2');
```

Pour procéder à une jointure entre les tables `mere` et `fille`, les identifiants `id_mere` de la table `fille` doivent correspondre avec les identifiants `id` de la table `mere` :

```
SELECT * FROM mere, fille
 WHERE mere.id = fille.id_mere;
 id | val_mere | id_fille | id_mere | val_fille
----+----------+----------+---------+-----------
  1 | mere 1   |        1 |       1 | fille 1
  1 | mere 1   |        2 |       1 | fille 2
(2 rows)
```

Un produit cartésien est créé en omettant la condition de jointure, le résultat n'a plus de sens :

```
SELECT * FROM mere, fille;
 id | val_mere | id_fille | id_mere | val_fille
----+----------+----------+---------+-----------
  1 | mere 1   |        1 |       1 | fille 1
  1 | mere 1   |        2 |       1 | fille 2
  2 | mere 2   |        1 |       1 | fille 1
  2 | mere 2   |        2 |       1 | fille 2
```

```
(4 rows)
```

1.4 TYPES DE DONNÉES

- Type de données
  - du standard SQL
  - certains spécifiques PostgreSQL

PostgreSQL propose l'ensemble des types de données du standard SQL, à l'exception du type `BLOB` qui a toutefois un équivalent. Mais PostgreSQL a été conçu pour être extensible et permet de créer facilement des types de données spécifiques. C'est pourquoi PostgreSQL propose un certain nombre de types de données spécifiques qui peuvent être intéressants.

1.4.1 QU'EST-CE QU'UN TYPE DE DONNÉES ?

- Le système de typage valide les données
- Un type détermine
  - les valeurs possibles
  - comment les données sont stockées
  - les opérations que l'on peut appliquer

On utilise des types de données pour représenter une information de manière pertinente. Les valeurs possibles d'une donnée vont dépendre de son type. Par exemple, un entier long ne permet par exemple pas de coder des valeurs décimales. De la même façon, un type entier ne permet pas de représenter une chaîne de caractère, mais l'inverse est possible.

L'intérêt du typage des données est qu'il permet également à la base de données de valider les données manipulées. Ainsi un entier `integer` permet de représenter des valeurs comprises entre -2,147,483,648 et 2,147,483,647. Si l'utilisateur tente d'insérer une donnée qui dépasse les capacités de ce type de données, une erreur lui sera retournée. On retrouve ainsi la notion d'intégrité des données. Comme pour les langages de programmation fortement typés, cela permet de détecter davantage d'erreurs, plus tôt : à la compilation dans les langages typés, ou ici des la première exécution d'une requête, plutôt que plus tard, quand une chaîne de caractère ne pourra pas être convertie à la volée en entier par exemple.

Le choix d'un type de données va également influencer la façon dont les données sont

représentées. En effet, toute donnée à une représentation textuelle et une représentation en mémoire et sur disque. Ainsi, un `integer` est représenté sous la forme d'une suite de 4 octets, manipulables directement par le processeur, alors que sa représentation textuelle est une suite de caractères. Cela a une implication forte sur les performances de la base de données.

Le type de données choisi permet également de déterminer les opérations que l'on pourra appliquer. Tous les types de données permettent d'utiliser des opérateurs qui leur sont propres. Ainsi il est possible d'additionner des entiers, de concaténer des chaînes de caractères, etc. Si une opération ne peut être réalisée nativement sur le type de données, il faudra utiliser des conversions coûteuses. Vaut-il mieux additionner deux entiers issus d'une conversion d'une chaîne de caractère vers un entier ou additionner directement deux entiers ? Vaut-il mieux stocker une adresse IP avec un `varchar` ou avec un type de données dédié ?

Il est à noter que l'utilisateur peut contrôler lui-même certains types de données paramétrés. Le paramètre représente la longueur ou la précision du type de données. Ainsi, un type `varchar(15)` permettra de représenter des chaînes de caractères de 15 caractères maximum.

1.4.2 TYPES DE DONNÉES

- Types standards SQL
- Types dérivés
- Types spécifiques à PostgreSQL
- Types utilisateurs

Les types de données standards permettent de traiter la plupart des situations qui peuvent survenir. Dans certains cas, il peut être nécessaire de faire appel aux types spécifiques à PostgreSQL, par exemple pour stocker des adresses IP avec le type spécifique et bénéficier par la même occasion de toutes les classes d'opérateurs qui permettent de manipuler simplement ce type de données.

Et si cela ne s'avère pas suffisant, PostgreSQL permet à l'utilisateur de créer lui-même ses propres types de données, ainsi que les classes d'opérateurs et fonctions permettant d'indexer ces données.

1.4.3 TYPES STANDARDS (1)

- Caractère
  - `char`, `varchar`
- Numérique
  - `integer`, `smallint`, `bigint`
  - `real`, `double precision`
  - `numeric`, `decimal`
- Booléen
  - `boolean`

Le standard SQL propose des types standards pour stocker des chaînes de caractères (de taille fixe ou variable), des données numériques (entières, à virgule flottante) et des booléens.

1.4.4 TYPES STANDARDS (2)

- Temporel
  - `date`, `time`
  - `timestamp`
  - `interval`
- Chaînes de bit
  - `bit`, `bit varying`
- Formats validés
  - JSON
  - XML

Le standard SQL propose également des types standards pour stocker des éléments temporels (date, heure, la combinaison des deux avec ou sans fuseau horaire, intervalle).

D'utilisation plus rare, SQL permet également de stocker des chaînes de bit et des données validées au format XML. Le format JSON est de plus en plus courant.

1.4.5 CARACTÈRES

- `char(n)`
  - longueur fixe
  - de *n* caractères
  - complété à droite par des espaces si nécessaire
- `varchar(n)`
  - longueur variable
  - maximum *n* caractères
  - *n* optionnel

Le type `char(n)` permet de stocker des chaînes de caractères de taille fixe, donnée par l'argument `n`. Si la chaîne que l'on souhaite stocker est plus petite que la taille donnée à la déclaration de la colonne, elle sera complétée par des espaces à droite. Si la chaîne que l'on souhaite stocker est trop grande, une erreur sera levée.

Le type `varchar(n)` permet de stocker des chaînes de caractères de taille variable. La taille maximale de la chaîne est donnée par l'argument `n`. Toute chaîne qui excèdera cette taille ne sera pas prise en compte et génèrera une erreur. Les chaînes de taille inférieure à la taille limite seront stockées sans altérations.

La longueur de chaîne est mesurée en nombre de caractères sous PostgreSQL. Ce n'est pas forcément le cas dans d'autres SGBD.

1.4.6 REPRÉSENTATION DONNÉES CARACTÈRES

- Norme SQL
  - chaîne encadrée par `'`
  - `'chaîne de caractères'`
- Chaînes avec échappement du style C
  - chaîne précédée par `E` ou `e`
  - `E'chaîne de caractères'`
- Chaînes avec échappement Unicode
  - chaîne précédée par `U&`
  - `U&'chaîne de caractères'`

La norme SQL définit que les chaînes de caractères sont représentées encadrées de guillemets simples (caractère `'`). Le guillemet double (caractère `"`) ne peut être utilisé car il sert à protéger la casse des noms d'objets. PostgreSQL interprétera alors la chaîne comme un nom d'objet et générera une erreur.

Une représentation correcte d'une chaîne de caractères est donc de la forme suivante :

```
'chaîne de caractères'
```

Les caractères `'` doivent être doublés s'ils apparaissent dans la chaîne :

```
'J''ai acheté des croissants'
```

Une extension de la norme par PostgreSQL permet d'utiliser les méta-caractères des langages tels que le C, par exemple `\n` pour un retour de ligne, `\t` pour une tabulation, etc. :

```
E'chaîne avec un retour \nde ligne et une \ttabulation'
```

1.4.7 NUMÉRIQUES

- Entier
  - `smallint`, `integer`, `bigint`
  - signés
- Virgule flottante
  - `real`, `double precision`
  - valeurs inexactes
- Précision arbitraire
  - `numeric(precision, echelle)`, `decimal(precision, echelle)`
  - valeurs exactes

Le standard SQL propose des types spécifiques pour stocker des entiers signés. Le type `smallint` permet de stocker des valeurs codées sur 2 octets, soit des valeurs comprises entre -32768 et +32767. Le type `integer` ou `int`, codé sur 4 octets, permet de stocker des valeurs comprises entre -2147483648 et +2147483647. Enfin, le type `bigint`, codé sur 8 octets, permet de stocker des valeurs comprises entre -9223372036854775808 et 9223372036854775807. Le standard SQL ne propose pas de stockage d'entiers non signés.

Le standard SQL permet de stocker des valeurs décimales en utilisant les types à virgules flottantes. Avant de les utiliser, il faut avoir à l'esprit que ces types de données ne permettent pas de stocker des valeurs exactes, des différences peuvent donc apparaître entre la donnée insérée et la donnée restituée. Le type `real` permet d'exprimer des valeurs à virgules flottantes sur 4 octets, avec une précision relative de six décimales. Le type `double precision` permet d'exprimer des valeurs à virgules flottantes sur huit octets, avec une précision relative de 15 décimales.

Beaucoup d'applications, notamment les applications financières, ne se satisfont pas de valeurs inexactes. Pour cela, le standard SQL propose le type `numeric`, ou son synonyme

`decimal`, qui permet de stocker des valeurs exactes, selon la précision arbitraire donnée. Dans la déclaration `numeric(precision, echelle)`, la partie `precision` indique combien de chiffres significatifs sont stockés, la partie `echelle` exprime le nombre de chiffres après la virgule. Au niveau du stockage, PostgreSQL ne permet pas d'insérer des valeurs qui dépassent les capacités du type déclaré. En revanche, si l'échelle de la valeur à stocker dépasse l'échelle déclarée de la colonne, alors sa valeur est simplement arrondie.

On peut aussi utiliser `numeric` sans aucune contrainte de taille, pour stocker de façon exacte n'importe quel nombre.

1.4.8 REPRÉSENTATION DE DONNÉES NUMÉRIQUES

- Chiffres décimaux : 0 à 9
- Séparateur décimal : `.`
- *chiffres*
- *chiffres.[chiffres][*`e`*[+-]chiffres]*
- *[chiffres].chiffres[*`e`*[+-]chiffres]*
- *chiffres*`e`*[+-]chiffres*
- Conversion
  - `TYPE 'chaine'`

Au moins un chiffre doit être placé avant ou après le point décimal, s'il est utilisé. Au moins un chiffre doit suivre l'indicateur d'exponentiel (caractère `e`), s'il est présent. Il peut ne pas y avoir d'espaces ou d'autres caractères imbriqués dans la constante. Notez que tout signe plus ou moins en avant n'est pas forcément considéré comme faisant part de la constante ; il est un opérateur appliqué à la constante.

Les exemples suivants montrent différentes représentations valides de constantes numériques :

```
42
3.5
4.
.001
5e2
1.925e-3
```

Une constante numérique contenant soit un point décimal soit un exposant est tout d'abord présumée du type `integer` si sa valeur est contenue dans le type integer (4 octets). Dans le cas contraire, il est présumé de type `bigint` si sa valeur entre dans un type `bigint` (8 octets). Dans le cas contraire, il est pris pour un type `numeric`. Les constantes contenant des points décimaux et/ou des exposants sont toujours présumées de type `numeric`.

Le type de données affecté initialement à une constante numérique est seulement un point de départ pour les algorithmes de résolution de types. Dans la plupart des cas, la constante sera automatiquement convertie dans le type le plus approprié suivant le contexte. Si nécessaire, vous pouvez forcer l'interprétation d'une valeur numérique sur un type de données spécifiques en la convertissant. Par exemple, vous pouvez forcer une valeur numérique à être traitée comme un type `real` (`float4`) en écrivant :

```
REAL '1.23'
```

1.4.9 BOOLÉENS

- `boolean`
- 3 valeurs possibles
  - `TRUE`
  - `FALSE`
  - `NULL` (ie valeur absente)

Le type `boolean` permet d'exprimer des valeurs booléennes, c'est-à-dire une valeur exprimant vrai ou faux. Comme tous les types de données en SQL, une colonne booléenne peut aussi ne pas avoir de valeur, auquel cas sa valeur sera `NULL`.

Un des intérêts des types booléens est de pouvoir écrire :

```
SELECT * FROM ma_table WHERE valide;
SELECT * FROM ma_table WHERE not consulte;
```

1.4.10 TEMPOREL

- Date
  - `date`
- Heure
  - `time`
  - avec ou sans fuseau horaire
- Date et heure
  - `timestamp`
  - avec ou sans fuseau horaire
- Intervalle de temps
  - `interval`

Le type `date` exprime une date. Ce type ne connaît pas la notion de fuseau horaire.

Le type `time` exprime une heure. Par défaut, il ne connaît pas la notion de fuseau horaire. En revanche, lorsque le type est déclaré comme `time with time zone`, il prend en compte un fuseau horaire. Mais cet emploi n'est pas recommandé. En effet, une heure convertie d'un fuseau horaire vers un autre pose de nombreux problèmes. En effet, le décalage horaire dépend également de la date : quand il est 6h00, heure d'été, à Paris, il est 21H00 sur la côte Pacifique aux États-Unis mais encore à la date de la veille.

Le type `timestamp` permet d'exprimer une date et une heure. Par défaut, il ne connaît pas la notion de fuseau horaire. Lorsque le type est déclaré `timestamp with time zone`, il est adapté aux conversions d'heure d'un fuseau horaire vers un autre car le changement de date sera répercuté dans la composante date du type de données. Il est précis à la microseconde.

Le format de saisie et de restitution des dates et heures dépend du paramètre `DateStyle`. La documentation de ce paramètre permet de connaître les différentes valeurs possibles. Il reste néanmoins recommandé d'utiliser les fonctions de formatage de date qui permettent de rendre l'application indépendante de la configuration du SGBD.

La norme ISO (ISO-8601) impose le format de date « année-mois-jour ». La norme SQL est plus permissive et permet de restituer une date au format « jour/mois/année » si DateStyle est égal à `'SQL, DMY'`.

```
SET datestyle = 'ISO, DMY';

SELECT current_timestamp;
              now
-------------------------------
 2017-08-29 16:11:58.290174+02

SET datestyle = 'SQL, DMY';

SELECT current_timestamp;
              now
-------------------------------
 29/08/2017 16:12:25.650716 CEST
```

1.4.11 REPRÉSENTATION DES DONNÉES TEMPORELLES

- Conversion explicite
  - `TYPE 'chaine'`
- Format d'un timestamp
  - `'YYYY-MM-DD HH24:MI:SS.ssssss'`
  - `'YYYY-MM-DD HH24:MI:SS.ssssss+fuseau'`
  - `'YYYY-MM-DD HH24:MI:SS.ssssss' AT TIME ZONE 'fuseau'`
- Format d'un intervalle
  - `INTERVAL 'durée interval'`

Expression d'une date, forcément sans gestion du fuseau horaire :

```
DATE '2017-08-29'
```

Expression d'une heure sans fuseau horaire :

```
TIME '10:20:10'
```

Ou, en spécifiant explicitement l'absence de fuseau horaire :

```
TIME WITHOUT TIME ZONE '10:20:10'
```

Expression d'une heure, avec fuseau horaire invariant. Cette forme est déconseillée :

```
TIME WITH TIME ZONE '10:20:10' AT TIME ZONE 'CEST'
```

Expression d'un timestamp sans fuseau horaire :

```
TIMESTAMP '2017-08-29 10:20:10'
```

Ou, en spécifiant explicitement l'absence de fuseau horaire :

```
TIMESTAMP WITHOUT TIME ZONE '2017-08-29 10:20:10'
```

Expression d'un timestamp avec fuseau horaire, avec microseconde :

```
TIMESTAMP WITH TIME ZONE '2017-08-29 10:20:10.123321'
AT TIME ZONE 'Europe/Paris'
```

Expression d'un intervalle d'une journée :

```
INTERVAL '1 day'
```

Il est possible de cumuler plusieurs expressions :

```
INTERVAL '1 year 1 day'
```

Les valeurs possibles sont :

- `YEAR` pour une année ;

- `MONTH` pour un mois ;
- `DAY` pour une journée ;
- `HOUR` pour une heure ;
- `MINUTE` pour une minute ;
- `SECOND` pour une seconde.

1.4.12 GESTION DES FUSEAUX HORAIRES

- Paramètre `timezone`
- Session : `SET TIME ZONE`
- Expression d'un fuseau horaire
  - nom complet : `'Europe/Paris'`
  - nom abbrégé : `'CEST'`
  - décalage : `'+02'`

Le paramètre `timezone` du `postgresql.conf` permet de positionner le fuseau horaire de l'instance PostgreSQL. Elle est initialisée par défaut en fonction de l'environnement du système d'exploitation.

Le fuseau horaire de l'instance peut également être défini au cours de la session à l'aide de la commande `SET TIME ZONE`.

La France utilise deux fuseaux horaires normalisés. Le premier, `CET`, correspond à *Central European Time* ou autrement dit à l'heure d'hiver en Europe centrale. Le second, `CEST`, correspond à *Central European Summer Time*, c'est-à-dire l'heure d'été en Europe centrale.

La liste des fuseaux horaires supportés est disponible dans la table système `pg_timezone_names` :

```
SELECT * FROM pg_timezone_names ;
               name               | abbrev | utc_offset | is_dst
----------------------------------+--------+------------+--------
 GB                               | BST    | 01:00:00   | t
 ROK                              | KST    | 09:00:00   | f
 Greenwich                        | GMT    | 00:00:00   | f
(...)
```

Il est possible de positionner le fuseau horaire au niveau de la session avec l'ordre `SET TIME ZONE` :

```
SET TIME ZONE "Europe/Paris";

SELECT now();
              now
-------------------------------
```

```
 2017-08-29 10:19:56.640162+02
```

```
SET TIME ZONE "Europe/Kiev";
```

```
SELECT now();
              now
-------------------------------
 2017-08-29 11:20:17.199983+03
```

Conversion implicite d'une donnée de type `timestamp` dans le fuseau horaire courant :

```
SET TIME ZONE "Europe/Kiev";
```

```
SELECT TIMESTAMP WITH TIME ZONE '2017-08-29 10:20:10 CEST';
      timestamptz
------------------------
 2017-08-29 11:20:10+03
```

Conversion explicite d'une donnée de type `timestamp` dans un autre fuseau horaire :

```
SELECT '2017-08-29 06:00:00' AT TIME ZONE 'US/Pacific';
      timezone
--------------------
 28/08/2017 21:00:00
```

1.4.13 CHAÎNES DE BITS

- Chaînes de bits
  - `bit(n)`, `bit varying(n)`

Les types `bit` et `bit varying` permettent de stocker des masques de bits. Le type `bit(n)` est à longueur fixe alors que le type `bit varying(n)` est à longueur variable mais avec un maximum de `n` bits.

1.4.14 REPRÉSENTATION DES CHAÎNES DE BITS

- Représentation binaire
  - chaîne de caractères précédée de la lettre `B`
  - `B'01010101'`
- Représentation hexadécimale
  - chaîne de caractères précédée de la lettre `X`
  - `X'55'`

1.4.15 XML

- Type validé
  - `xml`
- Chaîne de caractères
  - validation du document XML

Le type `xml` permet de stocker des documents XML. Par rapport à une chaîne de caractères simple, le type `xml` apporte la vérification de la structure du document XML ainsi que des fonctions de manipulations spécifiques (voir la documentation officielle[4]).

1.4.16 JSON

- Type `json` : texte, avec validation du format JSON
- Préférer le type `jsonb` (binaire)
- Fonctions de manipulation

Les types `json` et `jsonb` permettent de stocker des documents JSON. Ces deux types permettent de vérifier la structure du document JSON ainsi que des fonctions de manipulations spécifiques (voir la documentation officielle[5]). On préférera de loin le type `jsonb` pour son stockage optimisé (en binaire), et ses fonctionnalités supplémentaires, notamment en terme d'indexation.

---

[4] https://docs.postgresql.fr/current/functions-xml.html
[5] https://docs.postgresql.fr/current/functions-json.html

1.4.17 TYPES DÉRIVÉS

- Types spécifiques à PostgreSQL
- Sériés
  - principe de l'« autoincrement »
  - `serial`
  - `smallserial`
  - `bigserial`
  - équivalent à un type entier associé à une séquence et avec une valeur par défaut
  - (v 10+) préférer un type entier + la propriété `IDENTITY`
- Caractères
  - `text`

Les types `smallserial`, `serial` et `bigserial` permettent d'obtenir des fonctionnalités similaires aux types `autoincrement` rencontrés dans d'autres SGBD.

Néanmoins, ces types restent assez proches de la norme car ils définissent au final une colonne qui utilise un type et des objets standards. Selon le type dérivé utilisé, la colonne sera de type `smallint`, `integer` ou `bigint`. Une séquence sera également créée et la colonne prendra pour valeur par défaut la prochaine valeur de cette séquence.

Il est à noter que la notion d'identité apparaît en version 10 et qu'il est préférable de passer par cette contrainte que par ces types dérivés.

Attention : ces types n'interdisent pas l'insertion manuelle de doublons. Une contrainte de clé primaire explicite reste nécessaire pour les éviter.

Le type `text` est l'équivalent du type `varchar` mais sans limite de taille de la chaîne de caractère.

1.4.18 TYPES ADDITIONNELS NON SQL

- `bytea`
- `array`
- `enum`
- `cidr`, `inet`, `macaddr`
- `uuid`
- `json`, `jsonb`, `hstore`
- `range`

Les types standards ne sont pas toujours suffisants pour représenter certaines données.

À l'instar d'autres SGBDR, PostgreSQL propose des types de données pour répondre à certains besoins.

On notera le type `bytea` qui permet de stocker des objets binaires dans une table. Le type `array` permet de stocker des tableaux et `enum` des énumérations.

Les types `json` et `hstore` permettent de stocker des documents non structurés dans la base de données. Le premier au format JSON, le second dans un format de type clé/-valeur. Le type `hstore` est d'ailleurs particulièrement efficace car il dispose de méthodes d'indexation et de fonctions de manipulations performantes. Le type `json` a été completé par `jsonb` qui permet de stocker un document JSON binaire et optimisé, et d'accéder à une propriété sans désérialiser intégralement le document.

Le type `range` permet de stocker des intervalles de données. Ces données sont ensuite manipulables par un jeu d'opérateurs dédiés et par le biais de méthodes d'indexation permettant d'accélérer les recherches.

1.4.19 TYPES UTILISATEURS

- Types utilisateurs
  - composites
  - énumérés (`enum`)
  - intervalles (`range`)
  - scalaires
  - tableau

```
CREATE TYPE
```

PostgreSQL permet de créer ses propres types de données. Les usages les plus courants consistent à créer des types composites pour permettre à des fonctions de retourner des données sous forme tabulaire (retour de type `SETOF`).

L'utilisation du type énuméré (`enum`) nécessite aussi la création d'un type spécifique. Le type sera alors employé pour déclarer les objets utilisant une énumération.

Enfin, si l'on souhaite étendre les types intervalles (`range`) déjà disponibles, il est nécessaire de créer un type spécifique.

La création d'un type scalaire est bien plus marginale. Elle permet en effet d'étendre les types fournis par PostgreSQL mais nécessite d'avoir des connaissances fines des mécanismes de PostgreSQL. De plus, dans la majeure partie des cas, les types standards suffisent en général à résoudre les problèmes qui peuvent se poser à la conception.

Quant aux types tableaux, ils sont créés implicitement par PostgreSQL quand un utilisateur crée un type personnalisé.

Exemples

Utilisation d'un type `enum` :

```
CREATE TYPE arc_en_ciel AS ENUM (
    'red', 'orange', 'yellow', 'green', 'blue', 'purple'
);

CREATE TABLE test (id integer, couleur arc_en_ciel);

INSERT INTO test (id, couleur) VALUES (1, 'red');

INSERT INTO test (id, couleur) VALUES (2, 'pink');
ERROR:  invalid input value for enum arc_en_ciel: "pink"
LINE 1: INSERT INTO test (id, couleur) VALUES (2, 'pink');
```

Création d'un type interval `float8_range` :

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

1.5 CONCLUSION

- SQL : traitement d'ensembles d'enregistrements
- Pour les lectures : `SELECT`
- Nom des objets en minuscules
- Des types de données simples et d'autres plus complexes

Le standard SQL permet de traiter des ensembles d'enregistrements. Un enregistrement correspond à une ligne dans une relation. Il est possible de lire ces relations grâce à l'ordre `SELECT`.

1.5.1 BIBLIOGRAPHIE

- *Bases de données - de la modélisation au SQL* (Laurent Audibert)
- *SQL avancé : programmation et techniques avancées* (Joe Celko)
- *SQL : Au coeur des performances* (Markus Winand)
- *The Manga Guide to Databases* (Takahashi, Mana, Azuma, Shoko)
- *The Art of SQL* (Stéphane Faroult)

Bases de données - de la modélisation au SQL

- Auteur : Laurent Audibert
- Éditeur : Ellipses
- ISBN : 978-2729851200

Ce livre présente les notions essentielles pour modéliser une base de données et utiliser le langage SQL pour utiliser les bases de données créées. L'auteur appuie ses exercices sur PostgreSQL.

SQL avancé : programmation et techniques avancées

- Auteur : Joe Celko
- Editeur : Vuibert
- ISBN : 978-2711786503

Ce livre est écrit par une personne ayant participé à l'élaboration du standard SQL. Il a souhaité montré les bonnes pratiques pour utiliser le SQL pour résoudre un certain nombre de problèmes de tous les jours. Le livre s'appuie cependant sur la norme SQL-92, voire SQL-89. L'édition anglaise *SQL for Smarties* est bien plus à jour. Pour les anglophones, la lecture de l'ensemble des livres de Joe Celko est particulièrement recommandée.

SQL : Au coeur des performances

- Auteur : Markus Winand
- Éditeur : auto-édité
- ISBN : 978-3950307832
- site Internet[6]

Il s'agit du livre de référence sur les performances en SQL. Il dresse un inventaire des différents cas d'utilisation des index par la base de données, ce qui permettra de mieux prévoir l'indexation dès la conception. Ce livre s'adresse à un public avancé.

The Manga Guide to Databases

- Auteur : Takahashi, Mana, Azuma, Shoko

[6] https://use-the-index-luke.com/fr

- Éditeur : No Starch Press
- ASIN : B00BUFN70E

The Art of SQL

- Auteur : Stéphane Faroult
- Éditeur : O'Reilly
- ISBN : 978-0-596-00894-9
- ISBN : 978-0-596-15971-9 (e-book)

Ce livre s'adresse également à un public avancé. Il présente également les bonnes pratiques lorsque l'on utilise une base de données.

1.5.2 QUESTIONS

N'hésitez pas, c'est le moment !

1.6 TRAVAUX PRATIQUES

Ce TP utilise la base **tpc**. La base **tpc** peut être téléchargée depuis https://dali.bo/tp_tpc (dump de 31 Mo, pour 267 Mo sur le disque au final). Auparavant créer les utilisateurs depuis le script sur https://dali.bo/tp_tpc_roles.

```
$ psql < tpc_roles.sql           # Exécuter le script de création des rôles
$ createdb --owner tpc_owner tpc # Création de la base
$ pg_restore -d tpc tpc.dump     # Une erreur sur un schéma 'public' existant est normale
```

Les mots de passe sont dans le script. Pour vous connecter :

```
$ psql -U tpc_admin -h localhost -d tpc
```

Le schéma suivant montre les différentes tables de la base :

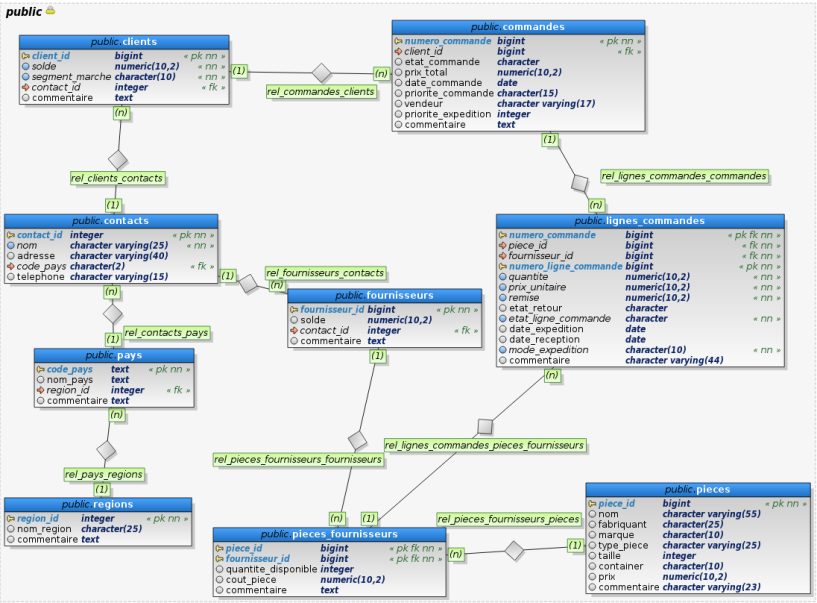

Figure 1: Schéma base tpc

> Afficher l'heure courante, au méridien de Greenwich.

> Afficher la date et l'heure qu'il sera dans 1 mois et 1 jour.

> Ajouter 1 au nombre de type réel '1.42'. Pourquoi ce résultat ? Quel type de données permet d'obtenir un résultat correct ?

> Afficher le contenu de la table `pays` en classant les pays dans l'ordre alphabétique.

> Afficher les pays contenant la lettre `a`, majuscule ou minuscule. Plusieurs solutions sont possibles.

> Afficher le nombre lignes de commandes dont la quantité commandée est comprise entre 5 et 10.

> Pour chaque pays, afficher son nom et la région du monde dont il fait partie.

```
nom_pays            | nom_region
--------------------+-----------
 ALGÉRIE            | Afrique
(...)
```

> Afficher le nombre total de clients français et allemands.

Sortie attendue :

```
count
-------
 12418
```

> Afficher le numéro de commande et le nom du client ayant passé la commande. Seul un sous-ensemble des résultats sera affiché : les 20 premières lignes du résultat seront exclues et seules les 20 suivantes seront affichées. Il faut penser à ce que le résultat de cette requête soit stable entre plusieurs exécutions.

Sortie attendue :

```
numero_commande  |  nom_client
-----------------+-------------
              67 | Client112078
              68 | Client33842
(...)
```

> Afficher les noms et codes des pays qui font partie de la région « Europe ».

Sortie attendue :

```
nom_pays        | code_pays
----------------+-----------
ALLEMAGNE       | DE
(...)
```

> Pour chaque pays, afficher une chaîne de caractères composée de son nom, suivi entre parenthèses de son code puis, séparé par une virgule, du nom de la région dont il fait partie.

Sortie attendue :

```
detail_pays
----------------------
 ALGÉRIE (DZ), Afrique
(...)
```

> Pour les clients ayant passé des commandes durant le mois de janvier 2011, affichez les identifiants des clients, leur nom, leur numéro de téléphone et le nom de leur pays.

Sortie attendue :

```
client_id |     nom      |    telephone    |           nom_pays
----------+--------------+-----------------+------------------------------
    83279 | Client83279  | 12-835-574-2048 | JAPON
```

> Pour les dix premières commandes de l'année 2011, afficher le numéro de la commande, la date de la commande ainsi que son âge.

Sortie attendue :

```
numero_commande | date_commande |            age
----------------+---------------+--------------------------
          11364 | 2011-01-01    | 1392 days 15:25:19.012521
(...)
```

1.7 TRAVAUX PRATIQUES (SOLUTIONS)

> Afficher l'heure courante, au méridien de Greenwich.

```
SELECT now() AT TIME ZONE 'GMT';
```

> Afficher la date et l'heure qu'il sera dans 1 mois et 1 jour.

```
SELECT now() + INTERVAL '1 month 1 day';
```

> Ajouter 1 au nombre de type réel '1.42'. Pourquoi ce résultat ? Quel type de données permet d'obtenir un résultat correct ?

```
SELECT REAL '1.42' + 1 AS resultat;
     resultat
------------------
 2.41999995708466
(1 row)
```

Le type de données `real` est un type numérique à virgule flottante, codé sur 4 octets. Il n'offre pas une précision suffisante pour les calculs précis. Son seul avantage est la vitesse de calcul. Pour effectuer des calculs précis, il vaut mieux privilégier le type de données `numeric`.

> Afficher le contenu de la table `pays` en classant les pays dans l'ordre alphabétique.

```
SELECT * FROM pays ORDER BY nom_pays;
```

> Afficher les pays contenant la lettre `a`, majuscule ou minuscule. Plusieurs solutions sont possibles.

```
SELECT * FROM pays WHERE lower(nom_pays) LIKE '%a%';

SELECT * FROM pays WHERE nom_pays ILIKE '%a%';

SELECT * FROM pays WHERE nom_pays LIKE '%a%' OR nom_pays LIKE '%A%';
```

En terme de performances, la seconde variante sera plus rapide sur un volume de données important si l'on dispose du bon index. La taille de la table `pays` ne permet pas d'observer de différence significative sur cette requête.

> Afficher le nombre lignes de commandes dont la quantité commandée est comprise entre 5 et 10.

```
SELECT count(*)
  FROM lignes_commandes
 WHERE quantite BETWEEN 5 AND 10;
```

Autre écriture possible :

```
SELECT count(*)
  FROM lignes_commandes
 WHERE quantite >= 5
   AND quantite <= 10;
```

> Pour chaque pays, afficher son nom et la région du monde dont il fait partie.

```
SELECT nom_pays, nom_region
  FROM pays p, regions r
 WHERE p.region_id = r.region_id;
```

> Afficher le nombre total de clients français et allemands.

```
SELECT count(*)
  FROM clients cl, contacts cn, pays p
 WHERE cl.contact_id = cn.contact_id
   AND cn.code_pays = p.code_pays
   AND p.nom_pays IN ('FRANCE', 'ALLEMAGNE');
```

À noter que cette syntaxe est obsolète, il faut utiliser la clause `JOIN`, plus lisible et plus complète, qui sera vue plus loin :

```
SELECT count(*)
  FROM clients cl
  JOIN contacts cn ON (cl.contact_id = cn.contact_id)
  JOIN pays p ON (cn.code_pays = p.code_pays)
 WHERE p.nom_pays IN ('FRANCE', 'ALLEMAGNE');
```

En connaissant les codes de ces pays, il est possible d'éviter la lecture de la table `pays` :

```
SELECT count(*)
  FROM clients cl, contacts cn
 WHERE cl.contact_id = cn.contact_id
   AND cn.code_pays IN ('FR', 'DE');
```

L'équivalent avec la syntaxe `JOIN` serait :

```
SELECT count(*)
  FROM clients cl
  JOIN contacts cn ON (cl.contact_id = cn.contact_id)
 WHERE cn.code_pays IN ('FR', 'DE');
```

> Afficher le numéro de commande et le nom du client ayant passé la commande. Seul un sous-ensemble des résultats sera affiché : les 20 premières lignes du résultat seront exclues et seules les 20 suivantes seront affichées. Il faut penser à ce que le résultat de cette requête soit stable entre plusieurs exécutions.

La syntaxe normalisée SQL impose d'écrire la requête de la façon suivante. La stabilité du résultat de la requête est garantie par un tri explicite, s'il n'est pas précisé, la base de données va retourner les lignes dans l'ordre physique qui est susceptible de changer entre deux exécutions :

```
SELECT numero_commande, nom AS nom_client
  FROM commandes cm, clients cl, contacts cn
 WHERE cm.client_id = cl.client_id
   AND cl.contact_id = cn.contact_id
 ORDER BY numero_commande
 FETCH FIRST 20 ROWS ONLY
 OFFSET 20;
```

Mais PostgreSQL supporte également la clause `LIMIT` :

```
SELECT numero_commande, nom AS nom_client
  FROM commandes cm, clients cl, contacts cn
 WHERE cm.client_id = cl.client_id
   AND cl.contact_id = cn.contact_id
 ORDER BY numero_commande
 LIMIT 20
 OFFSET 20;
```

Et l'équivalent avec la syntaxe `JOIN` serait :

```
SELECT numero_commande, nom AS nom_client
  FROM commandes cm
  JOIN clients cl ON (cm.client_id = cl.client_id)
  JOIN contacts cn ON (cl.contact_id = cn.contact_id)
 ORDER BY numero_commande
 LIMIT 20
 OFFSET 20;
```

> Afficher les noms et codes des pays qui font partie de la région « Europe ».

```
SELECT nom_pays, code_pays
  FROM regions r, pays p
 WHERE r.region_id = p.region_id
   AND r.nom_region = 'Europe';
```

Et l'équivalent avec la syntaxe `JOIN` serait :

```
SELECT nom_pays, code_pays
  FROM regions r
  JOIN pays p ON (r.region_id = p.region_id)
 WHERE r.nom_region = 'Europe';
```

> Pour chaque pays, afficher une chaîne de caractères composée de son nom, suivi entre parenthèses de son code puis, séparé par une virgule, du nom de la région dont il fait partie.

```
SELECT nom_pays || ' (' || code_pays || '), ' || nom_region
  FROM regions r, pays p
 WHERE r.region_id = p.region_id;
```

Et l'équivalent avec la syntaxe `JOIN` serait :

```
SELECT nom_pays || ' (' || code_pays || '), ' || nom_region
  FROM regions r
  JOIN pays p ON (r.region_id = p.region_id);
```

> Pour les clients ayant passé des commandes durant le mois de janvier 2011, affichez les identifiants des clients, leur nom, leur numéro de téléphone et le nom de leur pays.

```
SELECT cl.client_id, nom, telephone, nom_pays
  FROM clients cl, commandes cm, contacts cn, pays p
 WHERE cl.client_id = cm.client_id
   AND cl.contact_id = cn.contact_id
   AND cn.code_pays = p.code_pays
   AND date_commande BETWEEN '2011-01-01' AND '2011-01-31';
```

Le troisième module de la formation abordera les jointures et leurs syntaxes. À l'issue de ce prochain module, la requête de cet exercice pourrait être écrite de la façon suivante :

```
SELECT cl.client_id, nom, telephone, nom_pays
  FROM clients cl
```

```
  JOIN commandes cm
       USING (client_id)
  JOIN contacts co
       USING (contact_id)
  JOIN pays p
       USING (code_pays)
 WHERE date_commande BETWEEN '2011-01-01' AND '2011-01-31';
```

> Pour les dix premières commandes de l'année 2011, afficher le numéro de la commande, la date de la commande ainsi que son âge.

```
SELECT numero_commande, date_commande, now() - date_commande AS age
  FROM commandes
 WHERE date_commande BETWEEN '2011-01-01' AND '2011-12-31'
 ORDER BY date_commande
 LIMIT 10;
```

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.