Formation DEV0

Introduction à SQL

23.09

Table des matières

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[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

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, Pierrick Chovelon, 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, Alain Lesage, 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.

[1] mailto:formation@dalibo.com

Licence Creative Commons CC-BY-NC-SA

Marques déposées

Sur ce document

Formation	Formation DEV0
Titre	Introduction à SQL
Révision	23.09
ISBN	N/A
PDF	https://dali.bo/dev0_pdf

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode
[3] https://www.postgresql.org/about/policies/trademarks/

EPUB	https://dali.bo/dev0_epub
HTML	https://dali.bo/dev0_html
Slides	https://dali.bo/dev0_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

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 contre-partie, 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 :

```
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

Gestion transactionnelle : la force des bases de données relationnelles :

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

Les propriétés ACID sont le fondement même de toute bonne base de données. Il s'agit de l'acronyme des quatre règles que toute transaction (c'est-à-dire une suite d'ordres modifiant les données) doit respecter :

- **A** : Une transaction est appliquée en « 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 sera conservée de manière permanente.

Les bases de données relationnelles les plus courantes depuis des décennies (PostgreSQL bien sûr, mais aussi Oracle, MySQL, SQL Server, SQLite…) se basent sur ces principes, même si elles font chacune des compromis différents suivant leurs cas d'usage, les compromis acceptés à chaque époque avec la performance et les versions.

Atomicité :

Une transaction doit être exécutée entièrement ou pas du tout, et surtout pas partiellement, même si elle est longue et complexe, même en cas d'incident majeur sur la base de données. L'exemple basique est une transaction bancaire : le montant d'un virement doit être sur un compte ou un autre, et en cas de problème ne pas disparaître ou apparaître en double. Ce principe garantit que les données modifiées par des transactions valides seront toujours visibles dans un état stable, et évite nombre de problèmes fonctionnels comme techniques.

Cohérence :

Un état cohérent respecte les règles de validité définies dans le modèle, c'est-à-dire les contraintes définies dans le modèle : types, plages de valeurs admissibles, unicité, liens entre tables (clés étrangères), etc. Le non-respect de ces règles par l'applicatif entraîne une erreur et un rejet de la transaction.

Isolation :

Des transactions simultanées doivent agir comme si elles étaient seules sur la base. Surtout, elles ne voient pas les données *non validées* des autres transactions. Ainsi une transaction peut travailler sur un état stable et fixe, et durer assez longtemps sans risque de gêner les autres transactions.

Il existe plusieurs « niveaux d'isolation » pour définir précisément le comportement en cas de lectures ou écritures simultanées sur les mêmes données et pour arbitrer avec les contraintes de per-

formances ; le niveau le plus contraignant exige que tout se passe comme si toutes les transactions se déroulaient successivement.

Durabilité :

Une fois une transaction validée par le serveur (typiquement : `COMMIT` ne retourne pas d'erreur, ce qui valide la cohérence et l'enregistrement physique), l'utilisateur doit avoir la garantie que la donnée ne sera pas perdue ; du moins jusqu'à ce qu'il décide de la modifier à nouveau. Cette garantie doit valoir même en cas d'événement catastrophique : plantage de la base, perte d'un disque… C'est donc au serveur de s'assurer autant que possible que les différents éléments (disque, système d'exploitation…) ont bien rempli leur office. C'est à l'humain d'arbitrer entre le niveau de criticité requis et les contraintes de performances et de ressources adéquates (et fiables) à fournir à la base de données.

NoSQL :

À l'inverse, les outils de la mouvance (« NoSQL », par exemple MongoDB ou Cassandra), ne fournissent pas les garanties ACID. C'est le cas de la plupart des bases non-relationnelles, qui reprennent le modèle BASE[1] (*Basically Available*, *Soft State*, *Eventually Consistent*, soit succintement : disponibilité d'abord ; incohérence possible entre les réplicas ; cohérence… à terme, après un délai). Un intérêt est de débarasser le développeur de certaines lourdeurs apparentes liées à la modélisation assez stricte d'une base de données relationnelle. Cependant, la plupart des applications ont d'abord besoin des garanties de sécurité et cohérence qu'offrent un moteur transactionnel classique, et la décision d'utiliser un système ne les garantissant pas ne doit pas être prise à la légère ; sans parler d'autres critères comme la fragmentation du domaine par rapport au monde relationnel et son SQL (à peu près) standardisé. Avec le temps, les moteurs transactionnels ont acquis des fonctionnalités qui faisaient l'intérêt des bases NoSQL (en premier lieu la facilité de réplication et le stockage de JSON), et ces dernières ont tenté d'intégrer un peu plus de sécurité dans leur modèle.

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 »

[1] https://en.wikipedia.org/wiki/Eventual_consistency

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'« autocommit » 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

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

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.

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

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

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.

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.

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

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
- 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 !

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

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 :

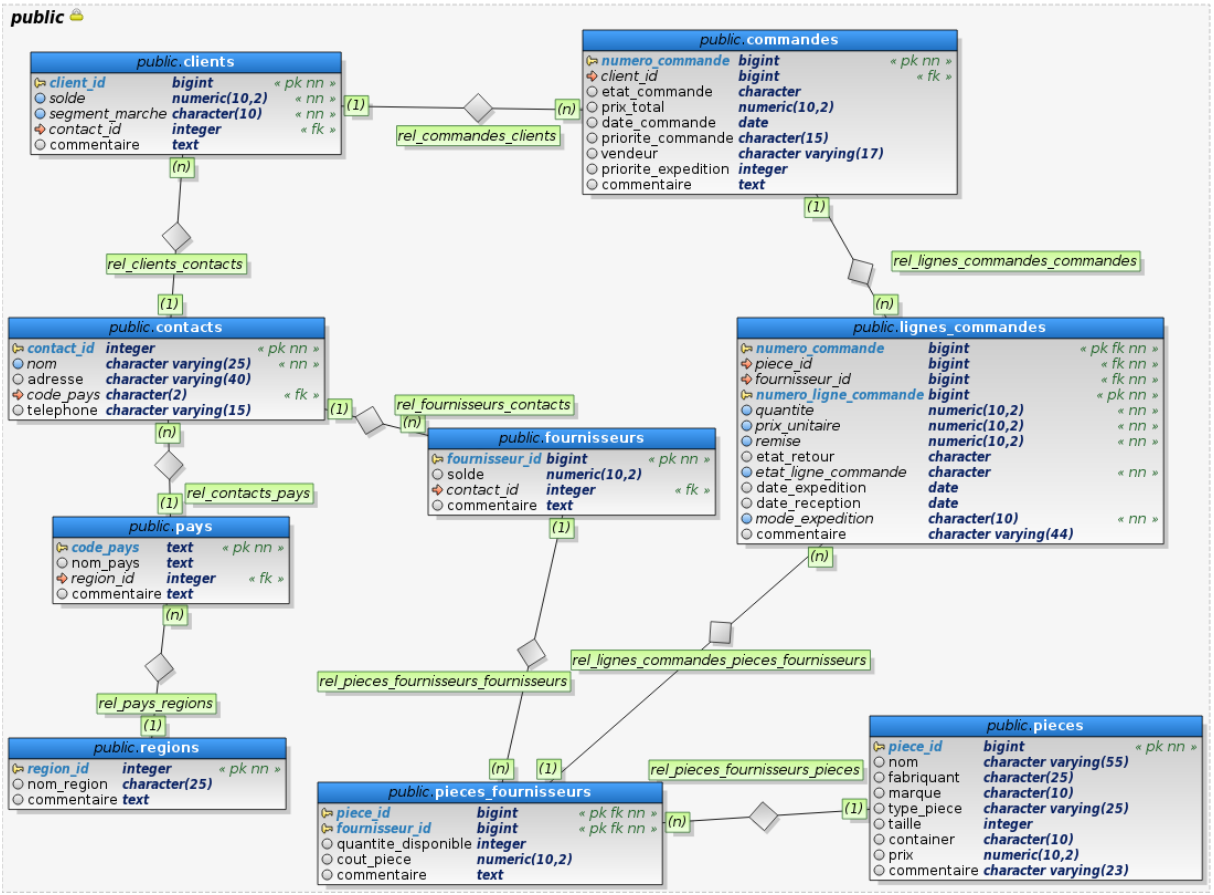


Figure 1/ .1: Schéma base tpc

> 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;
```

2/ Création d’objet et mises à jour

2.1 INTRODUCTION

- DDL, gérer les objets
- DML, écrire des données
- Gérer les transactions

Le module précédent nous a permis de voir comment lire des données à partir de requêtes SQL. Ce module a pour but de présenter la création et la gestion des objets dans la base de données (par exemple les tables), ainsi que l'ajout, la suppression et la modification de données.

Une dernière partie sera consacrée aux transactions.

2.1.1 Menu

- DDL (`Data Definition Language`)
- DML (`Data Manipulation Language`)
- TCL (`Transaction Control Language`)

2.1.2 Objectifs

- Savoir créer, modifier et supprimer des objets
- Savoir utiliser les contraintes d'intégrité
- Savoir mettre à jour les données
- Savoir utiliser les transactions

2.2 DDL

- DDL
  - `Data Definition Language`
  - langage de définition de données
- Permet de créer des objets dans la base de données

Les ordres DDL (acronyme de `Data Definition Language`) permettent de créer des objets dans la base de données et notamment la structure de base du standard SQL : les tables.

2.2.1 Objets d'une base de données

- Objets définis par la norme SQL :
  - schémas
  - séquences
  - tables
  - contraintes
  - domaines
  - vues
  - fonctions
  - triggers

La norme SQL définit un certain nombre d'objets standards qu'il est possible de créer en utilisant les ordres DDL. D'autres types d'objets existent bien entendu, comme les domaines. Les ordres DDL permettent également de créer des index, bien qu'ils ne soient pas définis dans la norme SQL.

La seule structure de données possible dans une base de données relationnelle est la table.

2.2.2 Créer des objets

- Ordre CREATE
- Syntaxe spécifique au type d'objet
- Exemple :

```
CREATE SCHEMA s1;
```

La création d'objet passe généralement par l'ordre CREATE. La syntaxe dépend fortement du type d'objet. Voici trois exemples :

```
CREATE SCHEMA s1;
CREATE TABLE t1 (c1 integer, c2 text);
CREATE SEQUENCE s1 INCREMENT BY 5 START 10;
```

Pour créer un objet, il faut être propriétaire du schéma ou de la base auquel appartiendra l'objet ou avoir le droit CREATE sur le schéma ou la base.

2.2.3 Modifier des objets

- Ordre ALTER
- Syntaxe spécifique pour modifier la définition d'un objet, exemple:
- Renommage

```
ALTER type_objet ancien_nom RENAME TO nouveau_nom ;
```

- changement de propriétaire

```
ALTER type_objet nom_objet OWNER TO proprietaire ;
```

- changement de schéma

```
ALTER type_objet nom_objet SET SCHEMA nom_schema ;
```

Modifier un objet veut dire modifier ses propriétés. On utilise dans ce cas l'ordre ALTER. Il faut être propriétaire de l'objet pour pouvoir le faire.

Deux propriétés sont communes à tous les objets : le nom de l'objet et son propriétaire. Deux autres sont fréquentes et dépendent du type de l'objet : le schéma et le tablespace. Les autres propriétés dépendent directement du type de l'objet.

2.2.4 Supprimer des objets

- Ordre DROP
- Exemples :
  - supprimer un objet :

```
DROP type_objet nom_objet ;
```

  - supprimer un objet et ses dépendances :

```
DROP type_objet nom_objet CASCADE ;
```

Seul un propriétaire peut supprimer un objet. Il utilise pour cela l'ordre DROP. Pour les objets ayant des dépendances, l'option CASCADE permet de tout supprimer d'un coup. C'est très pratique, et c'est en même temps très dangereux : il faut donc utiliser cette option à bon escient.

2.2.5 Schéma

- Identique à un espace de nommage
- Permet d'organiser les tables de façon logique
- Possibilité d'avoir des objets de même nom dans des schémas différents
- Pas d'imbrication (contrairement à des répertoires par exemple)
- Schéma `public`
  - créé par défaut dans une base de données PostgreSQL

La notion de schéma dans PostgreSQL est à rapprocher de la notion d'espace de nommage (ou *namespace*) de certains langages de programmation. Le catalogue système qui contient la définition des schémas dans PostgreSQL s'appelle d'ailleurs `pg_namespace`.

Les schémas sont utilisés pour répartir les objets de façon logique, suivant un schéma interne à l'entreprise. Ils servent aussi à faciliter la gestion des droits (il suffit de révoquer le droit d'utilisation d'un schéma à un utilisateur pour que les objets contenus dans ce schéma ne soient plus accessibles à cet utilisateur).

Un schéma `public` est créé par défaut dans toute nouvelle base de données. Tout le monde a le droit d'y créer des objets. Il est cependant possible de révoquer ce droit ou supprimer ce schéma.

2.2.6 Gestion d'un schéma

- `CREATE SCHEMA nom_schéma`
- `ALTER SCHEMA nom_schéma`
  - renommage
  - changement de propriétaire
- `DROP SCHEMA [ IF EXISTS ] nom_schéma [ CASCADE ]`

L'ordre `CREATE SCHEMA` permet de créer un schéma. Il suffit de lui spécifier le nom du schéma. `CREATE SCHEMA` offre d'autres possibilités qui sont rarement utilisées.

L'ordre `ALTER SCHEMA nom_schema RENAME TO nouveau_nom_schema` permet de renommer un schéma. L'ordre `ALTER SCHEMA nom_schema OWNER TO proprietaire` permet de donner un nouveau propriétaire au schéma.

Enfin, l'ordre `DROP SCHEMA` permet de supprimer un schéma. La clause `IF EXISTS` permet d'éviter la levée d'une erreur si le schéma n'existe pas (très utile dans les scripts SQL). La clause `CASCADE` permet de supprimer le schéma ainsi que tous les objets qui sont positionnés dans le schéma.

Exemples

Création d'un schéma `reference` :

```
CREATE SCHEMA reference;
```

Une table peut être créée dans ce schéma :

```
CREATE TABLE reference.communes (
  commune      text,
  codepostal   char(5),
  departement  text,
  codeinsee    integer
);
```

La suppression directe du schéma ne fonctionne pas car il porte encore la table `communes` :

```
DROP SCHEMA reference;
ERROR:  cannot drop schema reference because other objects depend on it
DETAIL:  table reference.communes depends on schema reference
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

L'option CASCADE permet de supprimer le schéma et ses objets dépendants :

```
DROP SCHEMA reference CASCADE;
NOTICE:  drop cascades to table reference.communes
```

2.2.7 Accès aux objets

- Nommage explicite
  - `nom_schema.nom_objet`
- Chemin de recherche de schéma
  - paramètre `search_path`
  - `SET search_path = schema1,schema2,public;`
  - par défaut : `$user, public`

Le paramètre `search_path` permet de définir un chemin de recherche pour pouvoir retrouver les tables dont le nom n'est pas qualifié par le nom de son schéma. PostgreSQL procèdera de la même façon que le système avec la variable $PATH : il recherche la table dans le premier schéma listé. S'il trouve une table portant ce nom dans le schéma, il préfixe le nom de table avec celui du schéma. S'il ne trouve pas de table de ce nom dans le schéma, il effectue la même opération sur le prochain schéma de la liste du `search_path`. S'il n'a trouvé aucune table de ce nom dans les schémas listés par `search_path`, PostgreSQL lève une erreur.

Comme beaucoup d'autres paramètres, le `search_path` peut être positionné à différents endroits. Par défaut, il est assigné à `$user, public`, c'est-à-dire que le premier schéma de recherche portera le nom de l'utilisateur courant, et le second schéma de recherche est `public`.

On peut vérifier la variable `search_path` à l'aide de la commande SHOW :

```
SHOW search_path;
  search_path
----------------
 "$user",public
(1 row)
```

Pour obtenir une configuration particulière, la variable `search_path` peut être positionnée dans le fichier `postgresql.conf` :

```
search_path = '"$user",public'
```

Cette variable peut aussi être positionnée au niveau d'un utilisateur. Chaque fois que l'utilisateur se connectera, il prendra le `search_path` de sa configuration spécifique :

```
ALTER ROLE nom_role SET search_path = "$user", public;
```

Cela peut aussi se faire au niveau d'une base de données. Chaque fois qu'un utilisateur se connectera à la base, il prendra le `search_path` de cette base, sauf si l'utilisateur a déjà une configuration spécifique :

```
ALTER DATABASE nom_base SET search_path = "$user", public;
```

La variable `search_path` peut également être positionnée pour un utilisateur particulier, dans une base particulière :

```
ALTER ROLE nom_role IN DATABASE nom_base SET search_path = "$user", public;
```

Enfin, la variable `search_path` peut être modifiée dynamiquement dans la session avec la commande SET :

```
SET search_path = "$user", public;
```

Avant la version 9.3, les requêtes préparées et les fonctions conservaient en mémoire le plan d'exécution des requêtes. Ce plan ne faisait plus référence aux noms des objets mais à leurs identifiants. Du coup, un `search_path` changeant entre deux exécutions d'une requête préparée ou d'une fonction ne permettait pas de cibler une table différente. Voici un exemple le montrant :

```
-- création des objets
CREATE SCHEMA s1;
CREATE SCHEMA s2;
CREATE TABLE s1.t1 (c1 text);
CREATE TABLE s2.t1 (c1 text);
INSERT INTO s1.t1 VALUES('schéma s1');
INSERT INTO s2.t1 VALUES('schéma s2');

SELECT * FROM s1.t1;
    c1
-----------
 schéma s1
(1 row)

SELECT * FROM s2.t1;
    c1
-----------
 schéma s2
(1 row)

-- il y a bien des données différentes dans chaque table

SET search_path TO s1;
PREPARE req AS SELECT * FROM t1;

EXECUTE req;
    c1
-----------
 schéma s1
(1 row)

SET search_path TO s2;
EXECUTE req;
    c1
-----------
 schéma s1
(1 row)

-- malgré le changement de search_path, nous en sommes toujours
-- aux données de l'autre table

b1=# SELECT * FROM t1;
    c1
-----------
 schéma s2
(1 row)
```

Dans ce cas, il est préférable de configurer le paramètre `search_path` directement au niveau de la fonction.
À partir de la version 9.3, dès que le `search_path` change, les plans en cache sont supprimés (dans le cas de la fonction) ou recréés (dans le cas des requêtes préparées).

2.2.8 Séquences

- Séquence
  - génère une séquence de nombres
- Paramètres
  - valeur minimale `MINVALUE`
  - valeur maximale `MAXVALUE`
  - valeur de départ `START`
  - incrément `INCREMENT`
  - cache `CACHE`
  - cycle autorisé `CYCLE`

Les séquences sont des objets standards qui permettent de générer des séquences de valeur. Elles sont utilisées notamment pour générer un numéro unique pour un identifiant ou, plus rarement, pour disposer d'un compteur informatif, mis à jour au besoin.

Le cache de la séquence a pour effet de générer un certain nombre de valeurs en mémoire afin de les mettre à disposition de la session qui a utilisé la séquence. Même si les valeurs pré-calculées ne sont pas consommées dans la session, elles seront consommées au niveau de la séquence. Cela peut avoir pour effet de créer des trous dans les séquences d'identifiants et de consommer très rapidement les numéros de séquence possibles. Le cache de séquence n'a pas besoin d'être ajusté sur des applications réalisant de petites transactions. Il permet en revanche d'améliorer les performances sur des applications qui utilisent massivement des numéros de séquences, notamment pour réaliser des insertions massives.

2.2.9 Création d'une séquence

```
CREATE SEQUENCE nom [ INCREMENT incrément ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
```

La syntaxe complète est donnée dans le slide.

Le mot clé TEMPORARY ou TEMP permet de définir si la séquence est temporaire. Si tel est le cas, elle sera détruite à la déconnexion de l'utilisateur.

Le mot clé INCREMENT définit l'incrément de la séquence, MINVALUE, la valeur minimale de la séquence et MAXVALUE, la valeur maximale. START détermine la valeur de départ initiale de la séquence, c'est-à-dire juste après sa création. La clause CACHE détermine le cache de séquence. CYCLE permet d'indiquer au SGBD que la séquence peut reprendre son compte à MINVALUE lorsqu'elle aura atteint MAXVALUE. La clause NO CYCLE indique que le rebouclage de la séquence est interdit, PostgreSQL lèvera alors une erreur lorsque la séquence aura atteint son MAXVALUE. Enfin, la clause OWNED BY détermine l'appartenance d'une séquence à une colonne d'une table. Ainsi, si la colonne est supprimée, la séquence sera implicitement supprimée.

Exemple de séquence avec rebouclage :

```
CREATE SEQUENCE testseq INCREMENT BY 1 MINVALUE 3 MAXVALUE 5 CYCLE START WITH 4;

SELECT nextval('testseq');
 nextval
---------
       4

SELECT nextval('testseq');
 nextval
---------
       5

SELECT nextval('testseq');
 nextval
---------
       3
```

2.2.10 Modification d'une séquence

```
ALTER SEQUENCE nom [ INCREMENT increment ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ RESTART [ [ WITH ] nouveau_début ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
```

- Il est aussi possible de modifier
  - le propriétaire
  - le schéma

Les propriétés de la séquence peuvent être modifiés avec l'ordre ALTER SEQUENCE.

La séquence peut être affectée à un nouveau propriétaire :

```
ALTER SEQUENCE [ IF EXISTS ] nom OWNER TO nouveau_propriétaire
```

Elle peut être renommée :

```
ALTER SEQUENCE [ IF EXISTS ] nom RENAME TO nouveau_nom
```

Enfin, elle peut être positionnée dans un nouveau schéma :

```
ALTER SEQUENCE [ IF EXISTS ] nom SET SCHEMA nouveau_schema
```

2.2.11 Suppression d'une séquence

```
DROP SEQUENCE nom [, ...]
```

Voici la syntaxe complète de DROP SEQUENCE :

```
DROP SEQUENCE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Le mot clé CASCADE permet de supprimer la séquence ainsi que tous les objets dépendants (par exemple la valeur par défaut d'une colonne).

2.2.12 Séquences, utilisation

- Obtenir la valeur suivante
  - `nextval('nom_sequence')`
- Obtenir la valeur courante
  - `currval('nom_sequence')`
  - mais `nextval()` doit être appelé avant dans la même session

La fonction `nextval()` permet d'obtenir le numéro de séquence suivant. Son comportement n'est pas transactionnel. Une fois qu'un numéro est consommé, il n'est pas possible de revenir dessus, malgré un ROLLBACK de la transaction. La séquence est le seul objet à avoir un comportement de ce type.

La fonction `currval()` permet d'obtenir le numéro de séquence courant, mais son usage nécessite d'avoir utilisé `nextval()` dans la session.

Il est possible d'interroger une séquence avec une requête SELECT. Cela permet d'obtenir des informations sur la séquence, dont la dernière valeur utilisée dans la colonne `last_value`. Cet usage n'est pas recommandé en production et doit plutôt être utilisé à titre informatif.

Exemples

Utilisation d'une séquence simple :

```
CREATE SEQUENCE testseq
INCREMENT BY 1 MINVALUE 10 MAXVALUE 20 START WITH 15 CACHE 1;


SELECT currval('testseq');
ERROR:  currval of sequence "testseq" is not yet defined in this session


SELECT * FROM testseq ;
- [ RECORD 1 ]-+--------
sequence_name  | testseq
last_value     | 15
start_value    | 15
increment_by   | 1
max_value      | 20
min_value      | 10
cache_value    | 5
log_cnt        | 0
is_cycled      | f
is_called      | f


SELECT nextval('testseq');
 nextval
---------
      15
(1 row)


SELECT currval('testseq');
 currval
---------
      15

SELECT nextval('testseq');
 nextval
---------
      16
(1 row)

ALTER SEQUENCE testseq RESTART WITH 5;
ERROR:  RESTART value (5) cannot be less than MINVALUE (10)

DROP SEQUENCE testseq;
```

Utilisation d'une séquence simple avec cache :

```
CREATE SEQUENCE testseq INCREMENT BY 1 CACHE 10;

SELECT nextval('testseq');
 nextval
---------
       1
```

Déconnexion et reconnexion de l'utilisateur :

```
SELECT nextval('testseq');
 nextval
---------
      11
```

Suppression en cascade d'une séquence :

```
CREATE TABLE t2 (id serial);

\d t2
                          Table "s2.t2"
 Column |  Type   |                     Modifiers
--------+---------+---------------------------------------------------
 id     | integer | not null default nextval('t2_id_seq'::regclass)

DROP SEQUENCE t2_id_seq;
ERROR:  cannot drop sequence t2_id_seq because other objects depend on it
DETAIL:  default for table t2 column id depends on sequence t2_id_seq
HINT:  Use DROP ... CASCADE to drop the dependent objects too.

DROP SEQUENCE t2_id_seq CASCADE;
NOTICE:  drop cascades to default for table t2 column id

\d t2
       Table "s2.t2"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
```

2.2.13 Type SERIAL

- Type `serial`/`bigserial`/`smallserial`
  - séquence générée automatiquement
  - valeur par défaut `nextval(...)`
- (v 10+) Préférer un entier avec `IDENTITY`

Certaines bases de données offrent des colonnes auto-incrémentées (`autoincrement` de MySQL ou `identity` de SQL Server).

PostgreSQL ne possède `identity` qu'à partir de la v 10. Jusqu'en 9.6 on pourra utiliser `serial` un équivalent qui s'appuie sur les séquences et la possibilité d'appliquer une valeur par défaut à une colonne.

Par exemple, si l'on crée la table suivante :

```
CREATE TABLE exemple_serial (
  id SERIAL PRIMARY KEY,
  valeur INTEGER NOT NULL
);
```

On s'aperçoit que table a été créée telle que demandé, mais qu'une séquence a aussi été créée. Elle porte un nom dérivé de la table associé à la colonne correspondant au type `serial`, terminé par `seq` :

```
postgres=# \d
                List of relations
 Schema |          Name           |   Type   | Owner
--------+-------------------------+----------+--------
 public | exemple_serial          | table    | thomas
 public | exemple_serial_id_seq | sequence | thomas
```

En examinant plus précisément la définition de la table, on s'aperçoit que la colonne `id` porte une valeur par défaut qui correspond à l'appel de la fonction `nextval()` sur la séquence qui a été créée implicitement :

```
postgres=# \d exemple_serial
                        Table "public.exemple_serial"
 Column |  Type   |                           Modifiers
--------+---------+---------------------------------------------------------------
 id     | integer | not null default nextval('exemple_serial_id_seq'::regclass)
 valeur | integer | not null
Indexes:
    "exemple_serial_pkey" PRIMARY KEY, btree (id)
```

`smallserial` et `bigserial` sont des variantes de `serial` s'appuyant sur des types d'entiers plus courts ou plus longs.

2.2.14 Domaines

- Permet d'associer
  - un type standard
  - et une contrainte (optionnelle)

Un domaine est utilisé pour définir un type utilisateur qui est en fait un type utilisateur standard accompagné de la définition de contraintes particulières.

Les domaines sont utiles pour ramener la définition de contraintes communes à plusieurs colonnes sur un seul objet. La maintenance en est ainsi facilitée.

L'ordre `CREATE DOMAIN` permet de créer un domaine, `ALTER DOMAIN` permet de modifier sa définition, et enfin, `DROP DOMAIN` permet de supprimer un domaine.

Exemples

Gestion d'un domaine `salaire` :

```
-- ajoutons le domaine et la table
CREATE DOMAIN salaire AS integer CHECK (VALUE > 0);
CREATE TABLE employes (id serial, nom text, paye salaire);

\d employes
                              Table « public.employes »
 Colonne |  Type   |  NULL-able |              Par défaut
---------+---------+------------+--------------------------------------
 id      | integer |  not null  | nextval('employes_id_seq'::regclass)
 nom     | text    |            |
 paye    | salaire |            |


-- insérons des données dans la nouvelle table
INSERT INTO employes (nom, paye) VALUES ('Albert', 1500);
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 0);
ERROR:  value for domain salaire violates check constraint "salaire_check"
-- erreur logique vu qu'on ne peut avoir qu'un entier strictement positif
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 1000);
INSERT 0 1
INSERT INTO employes (nom, paye) VALUES ('Bertrand', NULL);
INSERT 0 1
-- tous les employés doivent avoir un salaire
-- il faut donc modifier la contrainte, pour s'assurer
-- qu'aucune valeur NULL ne soit saisi
ALTER DOMAIN salaire SET NOT NULL;
ERROR:  column "paye" of table "employes" contains null values
-- la ligne est déjà présente, il faut la modifier
UPDATE employes SET paye=1500 WHERE nom='Bertrand';
-- maintenant, on peut ajouter la contrainte au domaine
ALTER DOMAIN salaire SET NOT NULL;
INSERT INTO employes (nom, paye) VALUES ('Delphine', NULL);
ERROR:  domain salaire does not allow null values
-- la contrainte est bien vérifiée
-- supprimons maintenant la contrainte
DROP DOMAIN salaire;
ERROR:  cannot drop type salaire because other objects depend on it
DETAIL:  table employes column paye depends on type salaire
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
-- il n'est pas possible de supprimer le domaine car il est référencé dans une
-- table.  Il faut donc utiliser l'option CASCADE
b1=# DROP DOMAIN salaire CASCADE;
NOTICE:  drop cascades to table employes column paye
```

```
DROP DOMAIN
-- le domaine a été supprimée ainsi que toutes les colonnes ayant ce type
\d employes
                                Table « public.employes »
 Colonne |  Type   | NULL-able |              Par défaut
---------+---------+-----------+--------------------------------------
 id      | integer | not null  | nextval('employes_id_seq'::regclass)
 nom     | text    |           |
```

Création et utilisation d'un domaine `code_postal_us` :

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE courrier_us (
  id_adresse SERIAL PRIMARY KEY,
  rue1 TEXT NOT NULL,
  rue2 TEXT,
  rue3 TEXT,
  ville TEXT NOT NULL,
  code_postal code_postal_us NOT NULL
);

INSERT INTO courrier_us (rue1,ville,code_postal)
VALUES ('51 Franklin Street', 'Boston, MA', '02110-1335' );

INSERT 0 1

INSERT INTO courrier_us (rue1,ville,code_postal)
VALUES ('10 rue d''Uzès','Paris','F-75002') ;

ERREUR:  la valeur pour le domaine code_postal_us viole la contrainte de
vérification « code_postal_us_check »
```

2.2.15 Tables

- Équivalent ensembliste d'une relation
- Composé principalement de
  - colonnes ordonnées
  - contraintes

La table est l'élément de base d'une base de données. Elle est composée de colonnes (à sa création) et est remplie avec des enregistrements (lignes de la table). Sa définition peut aussi faire intervenir

des contraintes, qui sont au niveau table ou colonne.

2.2.16 Création d'une table

- Définition de son nom
- Définition de ses colonnes
  - nom, type, contraintes éventuelles
- Clauses de stockage
- CREATE TABLE

Pour créer une table, il faut donner son nom et la liste des colonnes. Une colonne est définie par son nom et son type, mais aussi des contraintes optionnelles.

Des options sont possibles pour les tables, comme les clauses de stockage. Dans ce cas, on sort du contexte logique pour se placer au niveau physique.

2.2.17 CREATE TABLE

```
CREATE TABLE nom_table (
  definition_colonnes
  definition_contraintes
) clause_stockage;
```

La création d'une table passe par l'ordre CREATE TABLE. La définition des colonnes et des contraintes sont entre parenthèse après le nom de la table.

2.2.18 Définition des colonnes

```
nom_colonne type [ COLLATE collation ] [ contrainte ]
[, ...]
```

Les colonnes sont indiquées l'une après l'autre, en les séparant par des virgules.

Deux informations sont obligatoires pour chaque colonne : le nom et le type de la colonne. Dans le cas d'une colonne contenant du texte, il est possible de fournir le collationnement de la colonne. Quelque soit la colonne, il est ensuite possible d'ajouter des contraintes.

2.2.19 Valeur par défaut

- DEFAULT
  - affectation implicite
- Utiliser directement par les types sériés

La clause DEFAULT permet d'affecter une valeur par défaut lorsqu'une colonne n'est pas référencée dans l'ordre d'insertion ou si une mise à jour réinitialise la valeur de la colonne à sa valeur par défaut.

Les types sériés définissent une valeur par défaut sur les colonnes de ce type. Cette valeur est le retour de la fonction `nextval()` sur la séquence affectée automatiquement à cette colonne.

Exemples

Assignation d'une valeur par défaut :

```
CREATE TABLE valdefaut (
  id integer,
  i integer DEFAULT 0,
  j integer DEFAULT 0
);

INSERT INTO valdefaut (id, i) VALUES (1, 10);

SELECT * FROM valdefaut ;
 id | i  | j
----+----+---
  1 | 10 | 0
(1 row)
```

2.2.20 Copie de la définition d'une table

- Création d'une table à partir d'une autre table
  - `CREATE TABLE ... (LIKE table clause_inclusion)`
- Avec les valeurs par défaut des colonnes :
  - `INCLUDING DEFAULTS`
- Avec ses autres contraintes :
  - `INCLUDING CONSTRAINTS`
- Avec ses index :
  - `INCLUDING INDEXES`

L'ordre `CREATE TABLE` permet également de créer une table à partir de la définition d'une table déjà existante en utilisant la clause `LIKE` en lieu et place de la définition habituelles des colonnes. Par défaut, seule la définition des colonnes avec leur typage est repris.

Les clauses `INCLUDING` permettent de récupérer d'autres éléments de la définition de la table, comme les valeurs par défaut (`INCLUDING DEFAULTS`), les contraintes d'intégrité (`INCLUDING CONSTRAINTS`), les index (`INCLUDING INDEXES`), les clauses de stockage (`INCLUDING STORAGE`) ainsi que les commentaires (`INCLUDING COMMENTS`). Si l'ensemble de ces éléments sont repris, il est possible de résumer la clause `INCLUDING` à `INCLUDING ALL`.

La clause `CREATE TABLE` suivante permet de créer une table `archive_evenements_2010` à partir de la définition de la table `evenements` :

```
CREATE TABLE archive_evenements_2010
 (LIKE evenements
  INCLUDING DEFAULTS
  INCLUDING CONSTRAINTS
  INCLUDING INDEXES
  INCLUDING STORAGE
  INCLUDING COMMENTS
);
```

Elle est équivalente à :

```
CREATE TABLE archive_evenements_2010
 (LIKE evenements
  INCLUDING ALL
);
```

2.2.21 Modification d'une table

- `ALTER TABLE`
- Définition de la table
  - renommage de la table
  - ajout/modification/suppression d'une colonne
  - déplacement dans un schéma différent
  - changement du propriétaire
- Définition des colonnes
  - renommage d'une colonne
  - changement de type d'une colonne
- Définition des contraintes
  - ajout/suppression d'une contrainte

Pour modifier la définition d'une table (et non pas son contenu), il convient d'utiliser l'ordre `ALTER TABLE`. Il permet de traiter la définition de la table (nom, propriétaire, schéma, liste des colonnes), la définition des colonnes (ajout, modification de nom et de type, suppression… mais pas de changement au niveau de leur ordre), et la définition des contraintes (ajout et suppression).

2.2.22 Conséquences des modifications d'une table

- contention avec les verrous
- vérification des données
- performance avec une possible réécriture de la table

Suivant l'opération réalisée, les verrous posés ne seront pas les mêmes, même si le verrou par défaut sera un verrou exclusif. Par exemple, renommer une table nécessite un verrou exclusif mais changer la taille de l'échantillon statistiques bloque uniquement certaines opérations de maintenance (comme `VACUUM` et `ANALYZE`) et certaines opérations DDL. Il convient donc d'être très prudent lors de l'utilisation de la commande `ALTER TABLE` sur un serveur en production.

Certaines opérations nécessitent de vérifier les données. C'est évident lors de l'ajout d'une contrainte (comme une clé primaire ou une contrainte NOT NULL), mais c'est aussi le cas lors d'un changement de type de données. Passer une colonne du type `text` vers le type `timestamp` nécessite de vérifier que

les données de cette colonne ne contiennent que des données convertibles vers le type `timestamp`. Dans les anciennes versions, la vérification était effectuée en permanence, y compris pour des cas simples où cela n'était pas nécessaire. Par exemple, convertir une colonne du type `varchar(200)` à `varchar(100)` nécessite de vérifier que la colonne ne contient que des chaînes de caractères de longueur inférieure à 100. Mais convertir une colonne du type `varchar(100)` vers le type `varchar(200)` ne nécessite pas de vérification. Les dernières versions de PostgreSQL font la différence, ce qui permet d'éviter de perdre du temps pour une vérification inutile.

Certaines opérations nécessitent une réécriture de la table. Par exemple, convertir une colonne de type `varchar(5)` vers le type `int4` impose une réécriture de la table car il n'y a pas de compatibilité binaire entre les deux types. Ce n'est pas le cas si la modification est uniquement sur la taille d'une colonne `varchar`. Certaines optimisations sont ajoutées sur les nouvelles versions de PostgreSQL. Par exemple, l'ajout d'une colonne avec une valeur par défaut causait la réécriture complète de la table pour intégrer la valeur de cette nouvelle colonne alors que l'ajout d'une colonne sans valeur par défaut n'avait pas la même conséquence. À partir de la version 11, cette valeur par défaut est enregistrée dans la colonne `attmissingval` du catalogue système `pg_attribute` et la table n'a de ce fait plus besoin d'être réécrite.

Il convient donc d'être très prudent lors de l'utilisation de la commande `ALTER TABLE`. Elle peut poser des problèmes de performances, à cause de verrous posés par d'autres commandes, de verrous qu'elle réclame, de vérification des données, voire de réécriture de la table.

2.2.23 Suppression d'une table

- Supprimer une table :

```
DROP TABLE nom_table;
```

- Supprimer une table et tous les objets dépendants :

```
DROP TABLE nom_table CASCADE;
```

L'ordre `DROP TABLE` permet de supprimer une table. L'ordre `DROP TABLE ... CASCADE` permet de supprimer une table ainsi que tous ses objets dépendants. Il peut s'agir de séquences rattachées à une colonne d'une table, à des colonnes référençant la table à supprimer, etc.

2.2.24 Contraintes d'intégrité

- ACID
  - **C**ohérence
  - une transaction amène la base d'un état stable à un autre
- Assurent la cohérence des données
  - unicité des enregistrements
  - intégrité référentielle
  - vérification des valeurs
  - identité des enregistrements
  - règles sémantiques

Les données dans les différentes tables ne sont pas indépendantes mais obéissent à des règles sémantiques mises en place au moment de la conception du modèle conceptuel des données. Les contraintes d'intégrité ont pour principal objectif de garantir la cohérence des données entre elles, et donc de veiller à ce qu'elles respectent ces règles sémantiques. Si une insertion, une mise à jour ou une suppression viole ces règles, l'opération est purement et simplement annulée.

2.2.25 Clés primaires

- Identifie une ligne de manière unique
- Une seule clé primaire par table
- Une ou plusieurs colonnes
- À choisir parmi les clés candidates
  - parfois, utiliser une clé artificielle

Une clé primaire permet d'identifier une ligne de façon unique, il n'en existe qu'une seule par table.

Une clé primaire garantit que toutes les valeurs de la ou des colonnes qui composent cette clé sont uniques et non nulles. Elle peut être composée d'une seule colonne ou de plusieurs colonnes, selon le besoin.

La clé primaire est déterminée au moment de la conception du modèle de données.

Les clés primaires créent implicitement un index qui permet de renforcer cette contrainte.

2.2.26 Déclaration d'une clé primaire

Construction :

```
[CONSTRAINT nom_contrainte]
PRIMARY KEY ( nom_colonne [, ... ] )
```

Exemples

Définition de la table `region` :

```
CREATE TABLE region
  (
    id       serial   UNIQUE NOT NULL,
    libelle  text     NOT NULL,

    PRIMARY KEY(id)
  );

INSERT INTO region VALUES (1, 'un');
INSERT INTO region VALUES (2, 'deux');

INSERT INTO region VALUES (NULL, 'trois');
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (null, trois).

INSERT INTO region VALUES (1, 'trois');
ERROR:  duplicate key value violates unique constraint "region_pkey"
DETAIL:  Key (id)=(1) already exists.

INSERT INTO region VALUES (3, 'trois');

SELECT * FROM region;
 id | libelle
----+---------
  1 | un
  2 | deux
  3 | trois
(3 rows)
```

2.2.27 Contrainte d'unicité

- Garantie l'unicité des valeurs d'une ou plusieurs colonnes
- Permet plusieurs valeurs `NULL`
  - en v15, possibilité de modifier ce comportement
- Clause `UNIQUE`
- Contrainte `UNIQUE` != index `UNIQUE`

Une contrainte d'unicité permet de garantir que les valeurs de la ou des colonnes sur lesquelles porte la contrainte sont uniques. Elle autorise néanmoins d'avoir plusieurs valeurs `NULL` car elles ne sont pas considérées comme égales mais de valeur inconnue (`UNKNOWN`).

Une contrainte d'unicité peut être créée simplement en créant un index `UNIQUE` approprié. Ceci est fortement déconseillé du fait que la contrainte ne sera pas référencée comme telle dans le schéma de la base de données. Il sera donc très facile de ne pas la remarquer au moment d'une reprise du schéma pour une évolution majeure de l'application. Une colonne possédant un index `UNIQUE` peut malgré tout être référencée par une clé étrangère.

Les contraintes d'unicité créent implicitement un index qui permet de renforcer cette unicité.

2.2.28 Déclaration d'une contrainte d'unicité

Construction :

```
[ CONSTRAINT nom_contrainte]
{ UNIQUE { NULLS NOT DISTINCT } ( nom_colonne [, ... ] )
```

Voici un exemple complet.

Sans contrainte d'unicité, on peut insérer plusieurs fois la même valeur :

```
postgres=# CREATE TABLE utilisateurs(id integer);
CREATE TABLE
postgres=# INSERT INTO utilisateurs VALUES (10);
INSERT 0 1
postgres=# INSERT INTO utilisateurs VALUES (10);
INSERT 0 1
```

Ce n'est plus le cas avec une contrainte d'unicité :

```
postgres=# TRUNCATE utilisateurs;
TRUNCATE TABLE
```

```
postgres=# ALTER TABLE utilisateurs ADD UNIQUE(id);
ALTER TABLE
postgres=# INSERT INTO utilisateurs (id) VALUES (10);
INSERT 0 1
postgres=# INSERT INTO utilisateurs (id) VALUES (10);
ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(10) already exists.
postgres=# INSERT INTO utilisateurs (id) VALUES (11);
INSERT 0 1
```

Par contre, on peut insérer plusieurs valeurs NULL :

```
postgres=# INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT 0 1
postgres=# INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT 0 1
postgres=# INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT 0 1
```

Ce comportement est modifiable en version 15. Lors de la création de la contrainte, il faut préciser ce nouveau comportement :

```
postgres=# TRUNCATE utilisateurs;
TRUNCATE TABLE
postgres=# ALTER TABLE utilisateurs DROP CONSTRAINT utilisateurs_id_key;
ALTER TABLE
postgres=# ALTER TABLE utilisateurs ADD UNIQUE NULLS NOT DISTINCT(id);
ALTER TABLE
postgres=# INSERT INTO utilisateurs (id) VALUES (10);
INSERT 0 1
postgres=# INSERT INTO utilisateurs (id) VALUES (10);
ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(10) already exists.
postgres=# INSERT INTO utilisateurs (id) VALUES (11);
INSERT 0 1
postgres=# INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT 0 1
postgres=# INSERT INTO utilisateurs (id) VALUES (NULL);
ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(null) already exists.
```

2.2.29 Intégrité référentielle

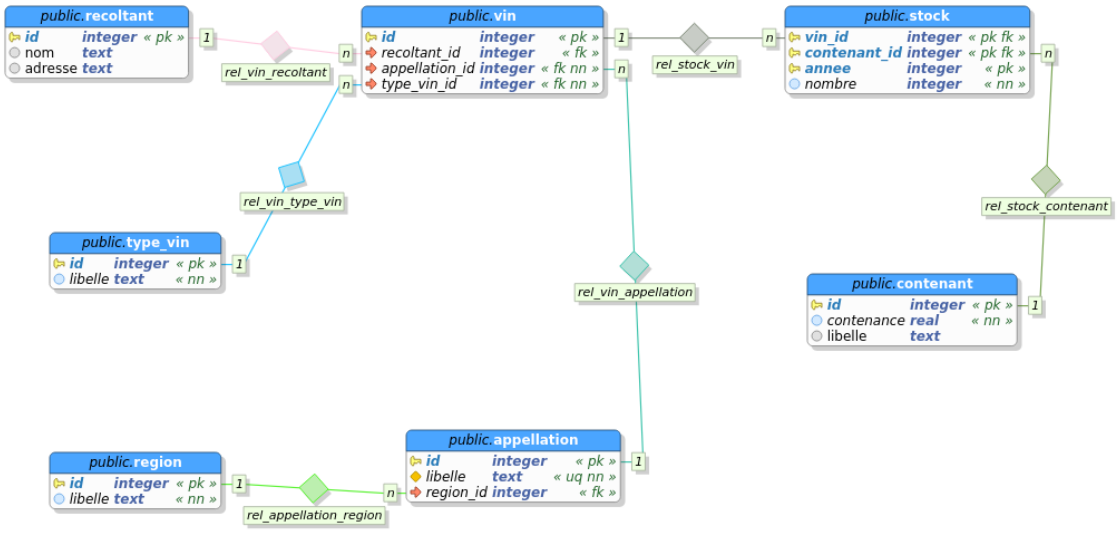
- **Contrainte d'intégrité référentielle**
  - ou **Clé étrangère**
- Référence une **clé primaire** ou un groupe de colonnes `UNIQUE` et `NOT NULL`
- Garantie l'intégrité des données
- `FOREIGN KEY`

Une clé étrangère sur une table fait référence à une clé primaire ou une contrainte d'unicité d'une autre table. La clé étrangère garantit que les valeurs des colonnes de cette clé existent également dans la table portant la clé primaire ou la contrainte d'unicité. On parle de contrainte référentielle d'intégrité : la contrainte interdit les valeurs qui n'existent pas dans la table référencée.

Ainsi, la base cave définit une table `region` et une table `appellation`. Une appellation d'origine est liée au terroir, et par extension à son origine géographique. La table `appellation` est donc liée par une clé étrangère à la table `region` : la colonne `region_id` de la table `appellation` référence la colonne `id` de la table `region`.

Cette contrainte permet d'empêcher les utilisateurs d'entrer dans la table `appellation` des identifiants de région (`region_id`) qui n'existent pas dans la table `region`.

2.2.30 Exemple



2.2.31 Déclaration d'une clé étrangère

```
[ CONSTRAINT nom_contrainte ] FOREIGN KEY ( nom_colonne [, ...] )
    REFERENCES table_reference [ (colonne_reference [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
```

Exemples

Définition de la table `stock` :

```
CREATE TABLE stock
  (
    vin_id        int    not null,
    contenant_id  int    not null,
    annee         int4   not null,
    nombre        int4   not null,

    PRIMARY KEY(vin_id,contenant_id,annee),

    FOREIGN KEY(vin_id) REFERENCES vin(id) ON DELETE CASCADE,
    FOREIGN KEY(contenant_id) REFERENCES contenant(id) ON DELETE CASCADE
  );
```

Création d'une table mère et d'une table fille. La table fille possède une clé étrangère qui référence la table mère :

```
CREATE TABLE mere (id integer, t text);

CREATE TABLE fille (id integer, mere_id integer, t text);

ALTER TABLE mere ADD CONSTRAINT pk_mere PRIMARY KEY (id);

ALTER TABLE fille
  ADD CONSTRAINT fk_mere_fille
      FOREIGN KEY (mere_id)
      REFERENCES mere (id)
         MATCH FULL
         ON UPDATE NO ACTION
         ON DELETE CASCADE;

INSERT INTO mere (id, t) VALUES (1, 'val1'), (2, 'val2');

-- l'ajout de données dans la table fille qui font bien référence
-- à la table mere fonctionne
INSERT INTO fille (id, mere_id, t) VALUES (1, 1, 'val1');
INSERT INTO fille (id, mere_id, t) VALUES (2, 2, 'val2');

-- l'ajout de données dans la table fille qui ne font pas référence
-- à la table mere est annulé
INSERT INTO fille (id, mere_id, t) VALUES (3, 3, 'val3');
ERROR:  insert or update on table "fille" violates foreign key constraint
        "fk_mere_fille"
DETAIL:  Key (mere_id)=(3) is not present in table "mere".

b1=# SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
(2 rows)

-- mettre à jour la référence dans la table mere ne fonctionnera pas
-- car la contrainte a été définie pour refuser les mises à jour
-- (ON UPDATE NO ACTION)

b1=# UPDATE mere SET id=3 WHERE id=2;
```

```
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(2) is still referenced from table "fille".

-- par contre, la suppression d'une ligne de la table mere référencée dans la
-- table fille va propager la suppression jusqu'à la table fille
-- (ON DELETE CASCADE)

b1=# DELETE FROM mere WHERE id=2;
DELETE 1

b1=# SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
(1 row)

b1=# SELECT * FROM mere;
 id |  t
----+------
  1 | val1
(1 row)
```

2.2.32 Vérification simple ou complète

- Vérification complète ou partielle d'une clé étrangère
- MATCH
  - MATCH FULL (complète)
  - MATCH SIMPLE (partielle)

La directive MATCH permet d'indiquer si la contrainte doit être entièrement vérifiée (MATCH FULL) ou si la clé étrangère autorise des valeurs NULL (MATCH SIMPLE). MATCH SIMPLE est la valeur par défaut.

Avec MATCH FULL, toutes les valeurs des colonnes qui composent la clé étrangère de la table référençant doivent avoir une correspondance dans la table référencée.

Avec MATCH SIMPLE, les valeurs des colonnes qui composent la clé étrangère de la table référençant peuvent comporter des valeurs NULL. Dans le cas des clés étrangères multi-colonnes, toutes les colonnes peuvent ne pas être renseignées. Dans le cas des clés étrangères sur une seule colonne, la contrainte autorise les valeurs NULL.

Exemples

Les exemples reprennent les tables `mere` et `fille` créées plus haut.

```
INSERT INTO fille VALUES (4, NULL, 'test');

SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
  4 |         | test
(2 rows)
```

2.2.33 Colonnes d'identité

- Identité d'un enregistrement
- `GENERATED ... AS IDENTITY`
  - `ALWAYS`
  - `BY DEFAULT`
- Préférer à `serial`
- Unicité non garantie sans contrainte explicite !

Cette contrainte permet d'avoir une colonne dont la valeur est incrémentée automatiquement, soit en permanence (clause `ALWAYS`), soit quand aucune valeur n'est saisie (clause `BY DEFAULT`). Cette technique d'auto-incrémentation correspond au standard SQL, contrairement au pseudo-type `serial` qui était utilisé jusqu'à la version 10.

De plus, elle corrige certains défauts de ce pseudo-type. Avec le type `serial`, l'utilisation de `CREATE TABLE .. LIKE` copiait la contrainte de valeur par défaut sans changer le nom de la séquence. Il n'est pas possible d'ajouter ou de supprimer un pseudo-type `serial` avec l'instruction `ALTER TABLE`. La suppression de la contrainte `DEFAULT` d'un type `serial` ne supprime pas la séquence associée. Tout ceci fait que la définition d'une colonne d'identité est préférable à l'utilisation du pseudo-type `serial`.

Il reste obligatoire de définir une clé primaire ou unique si l'on tient à l'unicité des valeurs car même une clause `GENERATED ALWAYS AS IDENTITY` peut être contournée avec une mise à jour portant la mention `OVERRIDING SYSTEM VALUE`.

Exemple :

```
CREATE table personnes (id int GENERATED ALWAYS AS IDENTITY, nom TEXT);

CREATE TABLE

INSERT INTO personnes (nom) VALUES ('Dupont') ;
INSERT 0 1
```

```
INSERT INTO personnes (nom) VALUES ('Durand') ;
INSERT 0 1

SELECT * FROM personnes ;
 id |  nom
----+--------
  1 | Dupont
  2 | Durand
(2 lignes)

INSERT INTO personnes (id,nom) VALUES (3,'Martin') ;

ERROR:  cannot insert into column "id"
DÉTAIL : Column "id" is an identity column defined as GENERATED ALWAYS.
ASTUCE : Use OVERRIDING SYSTEM VALUE to override.

INSERT INTO personnes (id,nom) OVERRIDING SYSTEM VALUE VALUES (3,'Martin') ;
INSERT 0 1

INSERT INTO personnes (id,nom) OVERRIDING SYSTEM VALUE VALUES (3,'Dupond') ;
INSERT 0 1

SELECT * FROM personnes ;
 id |  nom
----+--------
  1 | Dupont
  2 | Durand
  3 | Martin
  3 | Dupond
```

2.2.34 Mise à jour de la clé primaire

- Que faire en cas de mise à jour d'une clé primaire ?
  - les clés étrangères seront fausses
- ON UPDATE
- ON DELETE
- Définition d'une action au niveau de la clé étrangère
  - interdiction
  - propagation de la mise à jour
  - NULL
  - valeur par défaut

Si des valeurs d'une clé primaire sont mises à jour ou supprimées, cela peut entrainer des incohérences dans la base de données si des valeurs de clés étrangères font référence aux valeurs de la clé primaire touchées par le changement.

Afin de pouvoir gérer cela, la norme SQL prévoit plusieurs comportements possibles. La clause `ON UPDATE` permet de définir comment le SGBD va réagir si la clé primaire référencée est mise à jour. La clause `ON DELETE` fait de même pour les suppressions.

Les actions possibles sont :

- `NO ACTION` (ou `RESTRICT`), qui produit une erreur si une ligne référence encore le ou les lignes touchées par le changement ;
- `CASCADE`, pour laquelle la mise à jour ou la suppression est propagée aux valeurs référençant le ou les lignes touchées par le changement ;
- `SET NULL`, la valeur de la colonne devient `NULL` ;
- `SET DEFAULT`, pour lequel la valeur de la colonne prend la valeur par défaut de la colonne.

Le comportement par défaut est `NO ACTION`, ce qui est habituellement recommandé pour éviter les suppressions en chaîne mal maîtrisées.

Exemples

Les exemples reprennent les tables `mere` et `fille` créées plus haut.

Tentative d'insertion d'une ligne dont la valeur de `mere_id` n'existe pas dans la table `mere` :

```
INSERT INTO fille (id, mere_id, t) VALUES (1, 3, 'val3');
ERROR:  insert or update on table "fille" violates foreign key constraint
        "fk_mere_fille"
DETAIL:  Key (mere_id)=(3) is not present in table "mere".
```

Mise à jour d'une ligne de la table `mere` pour modifier son `id`. La clé étrangère est déclarée `ON UPDATE NO ACTION`, donc la mise à jour devrait être interdite :

```
UPDATE mere SET id = 3 WHERE id = 1;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

Suppression d'une ligne de la table `mere`. La clé étrangère sur `fille` est déclarée `ON DELETE CASCADE`, la suppression sera donc propagée aux tables qui référencent la table `mere` :

```
DELETE FROM mere WHERE id = 1;

SELECT * FROM fille ;
 id | mere_id |  t
----+---------+------
  2 |       2 | val2
(1 row)
```

2.2.35 Vérifications

- Présence d'une valeur
  - `NOT NULL`
- Vérification de la valeur d'une colonne
  - `CHECK`

La clause `NOT NULL` permet de s'assurer que la valeur de la colonne portant cette contrainte est renseignée. Dis autrement, elle doit obligatoirement être renseignée. Par défaut, la colonne peut avoir une valeur `NULL`, donc n'est pas obligatoirement renseignée.

La clause `CHECK` spécifie une expression de résultat booléen que les nouvelles lignes ou celles mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Les expressions de résultat `TRUE` ou `UNKNOWN` réussissent. Si une des lignes de l'opération d'insertion ou de mise à jour produit un résultat `FALSE`, une exception est levée et la base de données n'est pas modifiée. Une contrainte de vérification sur une colonne ne fait référence qu'à la valeur de la colonne tandis qu'une contrainte sur la table fait référence à plusieurs colonnes.

Actuellement, les expressions `CHECK` ne peuvent ni contenir des sous-requêtes ni faire référence à des variables autres que les colonnes de la ligne courante. C'est techniquement réalisable, mais non supporté.

2.2.36 Vérifications différés

- Vérifications après chaque ordre SQL
  - problèmes de cohérence
- Différer les vérifications de contraintes
  - clause `DEFERRABLE`, `NOT DEFERRABLE`
  - `INITIALLY DEFERED`, `INITIALLY IMMEDIATE`

Par défaut, toutes les contraintes d'intégrité sont vérifiées lors de l'exécution de chaque ordre SQL de modification, y compris dans une transaction. Cela peut poser des problèmes de cohérences de

données : insérer dans une table fille alors qu'on n'a pas encore inséré les données dans la table mère, la clé étrangère de la table fille va rejeter l'insertion et annuler la transaction.

Le moment où les contraintes sont vérifiées est modifiable dynamiquement par l'ordre SET CONSTRAINTS :

```
SET CONSTRAINTS { ALL | nom [, ...] } { DEFERRED | IMMEDIATE }
```

mais ce n'est utilisable que pour les contraintes déclarées comme déferrables.

Voici quelques exemples :

- avec la définition précédente des tables `mere` et `fille`

```
b1=# BEGIN;
UPDATE mere SET id=3 where id=1;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

- cette erreur survient aussi dans le cas où on demande que la vérification des contraintes soit différée pour cette transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED;
UPDATE mere SET id=3 WHERE id=1;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

- il faut que la contrainte soit déclarée comme étant différable :

```
CREATE TABLE mere (id integer, t text);
CREATE TABLE fille (id integer, mere_id integer, t text);
ALTER TABLE mere ADD CONSTRAINT pk_mere PRIMARY KEY (id);
ALTER TABLE fille
  ADD CONSTRAINT fk_mere_fille
      FOREIGN KEY (mere_id)
      REFERENCES mere (id)
         MATCH FULL
         ON UPDATE NO ACTION
         ON DELETE CASCADE
         DEFERRABLE;
INSERT INTO mere (id, t) VALUES (1, 'val1'), (2, 'val2');
INSERT INTO fille (id, mere_id, t) VALUES (1, 1, 'val1');
INSERT INTO fille (id, mere_id, t) VALUES (2, 2, 'val2');

BEGIN;
SET CONSTRAINTS all deferred;
UPDATE mere SET id=3 WHERE id=1;
SELECT * FROM mere;
 id |  t
----+------
  2 | val2
  3 | val1
```

```
(2 rows)

SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
(2 rows)

UPDATE fille SET mere_id=3 WHERE mere_id=1;
COMMIT;
```

2.2.37 Vérifications plus complexes

- Un trigger
  - si une contrainte porte sur plusieurs tables
  - si sa vérification nécessite une sous-requête
- Préférer les contraintes déclaratives

Les contraintes d'intégrités du SGBD ne permettent pas d'exprimer une contrainte qui porte sur plusieurs tables ou simplement si sa vérification nécessite une sous-requête. Dans ce cas là, il est nécessaire d'écrire un trigger spécifique qui sera déclenché après chaque modification pour valider la contrainte.

Il ne faut toutefois pas systématiser l'utilisation de triggers pour valider des contraintes d'intégrité. Cela aurait un impact fort sur les performances et sur la maintenabilité de la base de données. Il vaut mieux privilégier les contraintes déclaratives et n'envisager l'emploi de triggers que dans les cas où ils sont réellement nécessaires.

2.3 DML : MISE À JOUR DES DONNÉES

- SELECT peut lire les données d'une table ou plusieurs tables
  - mais ne peut pas les mettre à jour
- Ajout de données dans une table
  - INSERT
- Modification des données d'une table
  - UPDATE
- Suppression des données d'une table
  - DELETE

L'ordre SELECT permet de lire une ou plusieurs tables. Les mises à jours utilisent des ordres distincts.

L'ordre INSERT permet d'ajouter ou insérer des données dans une table. L'ordre UPDATE permet de modifier des lignes déjà existantes. Enfin, l'ordre DELETE permet de supprimer des lignes. Ces ordres ne peuvent travailler que sur une seule table à la fois. Si on souhaite par exemple insérer des données dans deux tables, il est nécessaire de réaliser deux INSERT distincts.

2.3.1 Ajout de données : INSERT

- Ajoute des lignes à partir des données de la requête
- Ajoute des lignes à partir d'une requête SELECT
- Syntaxe :

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    { liste_valeurs | requete }
```

L'ordre INSERT insère de nouvelles lignes dans une table. Il permet d'insérer une ou plusieurs lignes spécifiées par les expressions de valeur, ou zéro ou plusieurs lignes provenant d'une requête.

La liste des noms de colonnes est optionnelle. Si elle n'est pas spécifiée, alors PostgreSQL utilisera implicitement la liste de toutes les colonnes de la table dans l'ordre de leur déclaration, ou les N pre-

miers noms de colonnes si seules N valeurs de colonnes sont fournies dans la clause VALUES ou dans la requête. L'ordre des noms des colonnes dans la liste n'a pas d'importance particulière, il suffit de nommer les colonnes mises à jour.

Chaque colonne absente de la liste, implicite ou explicite, se voit attribuer sa valeur par défaut, s'il y en a une ou NULL dans le cas contraire. Les expressions de colonnes qui ne correspondent pas au type de données déclarées sont transtypées automatiquement, dans la mesure du possible.

2.3.2 INSERT avec liste d'expressions

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    VALUES ( { expression | DEFAULT } [, ...] ) [, ...]
```

La clause VALUES permet de définir une liste d'expressions qui va constituer la ligne à insérer dans la base de données. Les éléments de cette liste d'expression sont séparés par une virgule. Cette liste d'expression est composée de constantes ou d'appels à des fonctions retournant une valeur, pour obtenir par exemple la date courante ou la prochaine valeur d'une séquence. Les valeurs fournies par la clause VALUES ou par la requête sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Exemples

Insertion d'une ligne dans la table `stock` :

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre)
 VALUES (12, 1, 1935, 1);
```

Insertion d'une ligne dans la table `vin` :

```
INSERT INTO vin (id, recoltant_id, appellation_id, type_vin_id)
 VALUES (nextval('vin_id_seq'), 3, 6, 1);
```

2.3.3 INSERT à partir d'un SELECT

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    requête
```

L'ordre INSERT peut aussi prendre une requête SQL en entrée. Dans ce cas, INSERT va insérer autant de lignes dans la table d'arrivée qu'il y a de lignes retournées par la requête SELECT. L'ordre des

colonnes retournées par SELECT doit correspondre à l'ordre des colonnes de la liste des colonnes. Leur type de données doit également correspondre.

Exemples

Insertion dans une table `stock2` à partir d'une requête SELECT sur la table `stock1` :

```
INSERT INTO stock2 (vin_id, contenant_id, annee, nombre)
SELECT vin_id, contenant_id, annee, nombre FROM stock;
```

2.3.4 INSERT et colonnes implicites

- L'ordre physique peut changer dans le temps
  - résultats incohérents
  - requêtes en erreurs

Il est préférable de lister explicitement les colonnes touchées par l'ordre INSERT afin de garder un ordre d'insertion déterministe. En effet, l'ordre des colonnes peut changer notamment lorsque certains ETL sont utilisés pour modifier le type d'une colonne `varchar(10)` en `varchar(11)`. Par exemple, pour la colonne `username`, l'ETL Kettle génère les ordres suivants :

```
ALTER TABLE utilisateurs ADD COLUMN username_KTL VARCHAR(11);
UPDATE utilisateurs SET username_KTL=username;
ALTER TABLE utilisateurs DROP COLUMN username;
ALTER TABLE utilisateurs RENAME username_KTL TO username
```

Il génère des ordres SQL inutiles et consommateurs d'entrées/sorties disques car il doit générer des ordres SQL compris par tous les SGBD du marché. Or, tous les SGBD ne permettent pas de changer le type d'une colonne aussi simplement que dans PostgreSQL.

Exemples

Exemple de modification du schéma pouvant entrainer des problèmes d'insertion si les colonnes ne sont pas listées explicitement :

```
CREATE TABLE insere (id integer PRIMARY KEY, col1 varchar(5), col2 integer);

INSERT INTO insere VALUES (1, 'XX', 10);

ALTER TABLE insere ADD COLUMN col1_tmp varchar(6);
UPDATE insere SET col1_tmp = col1;
ALTER TABLE insere DROP COLUMN col1;
ALTER TABLE insere RENAME COLUMN col1_tmp TO col1;

INSERT INTO insere VALUES (2, 'XXX', 10);
ERROR:  invalid input syntax for integer: "XXX"
LINE 1: INSERT INTO insere VALUES (2, 'XXX', 10);
```

2.3.5 Mise à jour de données : UPDATE

- Ordre UPDATE
- Met à jour une ou plusieurs colonnes d'une même ligne
  - à partir des valeurs de la requête
  - à partir des anciennes valeurs
  - à partir d'une requête SELECT
  - à partir de valeurs d'une autre table

L'ordre de mise à jour de lignes s'appelle UPDATE.

2.3.6 Construction d'UPDATE

```
UPDATE nom_table
    SET
    {
     nom_colonne = { expression | DEFAULT }
    |
     ( nom_colonne [, ...] ) = ( { expression | DEFAULT } [, ...] )
    } [, ...]
    [ FROM liste_from ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
```

L'ordre UPDATE permet de mettre à jour les lignes d'une table.

L'ordre UPDATE ne met à jour que les lignes qui satisfont les conditions de la clause WHERE. La clause SET permet de définir les colonnes à mettre à jour. Le nom des colonne mises à jour doivent faire partie de la table mise à jour.

Les valeurs mises à jour peuvent faire référence aux valeurs avant mise à jour de la colonne, dans ce cas on utilise la forme `nom_colonne = nom_colonne`. La partie de gauche référence la colonne à mettre à jour, la partie de droite est une expression qui permet de déterminer la valeur à appliquer à la colonne. La valeur à appliquer peut bien entendu être une référence à une ou plusieurs colonnes et elles peuvent être dérivées par une opération arithmétique.

La clause FROM ne fait pas partie de la norme SQL mais certains SGBDR la supportent, notamment SQL Server et PostgreSQL. Elle permet de réaliser facilement la mise à jour d'une table à partir des valeurs d'une ou plusieurs tables annexes.

La norme SQL permet néanmoins de réaliser des mises à jour en utilisant une sous-requête, permettant d'éviter l'usage de la clause FROM.

Exemples

Mise à jour du prix d'un livre particulier :

```
UPDATE livres SET prix = 10 WHERE isbn = '978-3-8365-3872-5';
```

Augmentation de 5 % du prix des livres :

```
UPDATE livres SET prix = prix * 1.05;
```

Mise à jour d'une table `employees` à partir des données d'une table `bonus_plan` :

```
UPDATE employees e
   SET commission_rate = bp.commission_rate
  FROM bonus_plan bp
    ON (e.bonus_plan = bp.planid)
```

La même requête avec une sous-requête, conforme à la norme SQL :

```
UPDATE employees
   SET commission_rate = (SELECT commission_rate
                            FROM bonus_plan bp
                           WHERE bp.planid = employees.bonus_plan);
```

Lorsque plusieurs colonnes doivent être mises à jour à partir d'une jointure, il est possible d'utiliser ces deux écritures :

```
UPDATE employees e
   SET commission_rate = bp.commission_rate,
       commission_rate2 = bp.commission_rate2
  FROM bonus_plan bp
    ON (e.bonus_plan = bp.planid);
```

et

```
UPDATE employees e
   SET (commission_rate, commission_rate2) = (
      SELECT bp.commission_rate, bp.commission_rate2
      FROM bonus_plan bp ON (e.bonus_plan = bp.planid)
   );
```

2.3.7 Suppression de données : DELETE

- Supprime les lignes répondant au prédicat
- Syntaxe :

```
DELETE FROM nom_table [ [ AS ] alias ]
    [ WHERE condition
```

L'ordre DELETE supprime l'ensemble des lignes qui répondent au prédicat de la clause WHERE.

```
DELETE FROM nom_table [ [ AS ] alias ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
```

Exemples

Suppression d'un livre épuisé du catalogue :

```
DELETE FROM livres WHERE isbn = '978-0-8707-0635-6';
```

2.3.8 Clause RETURNING

- Spécifique à PostgreSQL
- Permet de retourner les lignes complètes ou partielles résultants de INSERT, UPDATE ou DELETE
- Syntaxe :

```
requete_sql RETURNING ( * | expression )
```

La clause RETURNING est une extension de PostgreSQL. Elle permet de retourner les lignes insérées, mises à jour ou supprimées par un ordre DML de modification. Il est également possible de dériver une valeur retournée.

L'emploi de la clause RETURNING peut nécessiter des droits complémentaires sur les objets de la base.

Exemples

Mise à jour du nombre de bouteilles en stock :

```
SELECT annee, nombre FROM stock
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967;
 annee | nombre
-------+--------
  1967 |     17
(1 row)

UPDATE stock SET nombre = nombre - 1
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967 RETURNING nombre;
 nombre
--------
     16
(1 row)
```

2.4 TRANSACTIONS

- ACID
  - **A**tomicité
  - un traitement se fait en entier ou pas du tout
- TCL pour Transaction Control Language
  - valide une transaction
  - annule une transaction
  - points de sauvegarde

Les transactions sont une partie essentielle du langage SQL. Elles permettent de rendre atomique un certain nombre de requêtes. Le résultat de toutes les requêtes d'une transaction est validée ou pas, mais on ne peut pas avoir d'état intermédiaire.

Le langage SQL définit qu'une transaction peut être validée ou annulée. Ce sont respectivement les ordres `COMMIT` et `ROLLBACK`. Il est aussi possible de faire des points de reprise ou de sauvegarde dans une transaction. Ils se font en utilisant l'ordre `SAVEPOINT`.

2.4.1 Auto-commit et transactions

- Par défaut, PostgreSQL fonctionne en auto-commit
  - à moins d'ouvrir explicitement une transaction
- Ouvrir une transaction
  - `BEGIN TRANSACTION`

PostgreSQL fonctionne en auto-commit. Autrement dit, sans `BEGIN`, une requête est considérée comme une transaction complète et n'a donc pas besoin de `COMMIT`.

Une transaction débute toujours par un `START` ou un `BEGIN`.

2.4.2 Validation ou annulation d'une transaction

- Valider une transaction
  - `COMMIT`
- Annuler une transaction
  - `ROLLBACK`
- Sans validation, une transaction est forcément annulée

Une transaction est toujours terminée par une `COMMIT` ou un `END` quand on veut que les modifications soient définitivement enregistrées, et par un `ROLLBACK` dans le cas contraire.

La transaction en cours d'une session qui se termine, quelle que soit la raison, sans `COMMIT` et sans `ROLLBACK` est considérée comme annulée.

Exemples

Avant de retirer une bouteille du stock, on vérifie tout d'abord qu'il reste suffisamment de bouteilles en stock :

```
BEGIN TRANSACTION;

SELECT annee, nombre FROM stock WHERE vin_id = 7 AND contenant_id = 1
AND annee = 1967;
 annee | nombre
-------+--------
  1967 |     17
(1 row)

UPDATE stock SET nombre = nombre - 1
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967 RETURNING nombre;
 nombre
--------
     16
(1 row)

COMMIT;
```

2.4.3 Programmation

- Certains langages implémentent des méthodes de gestion des transactions
  - PHP, Java, etc.
- Utiliser ces méthodes prioritairement

La plupart des langages permettent de gérer les transactions à l'aide de méthodes ou fonctions particulières. Il est recommandé de les utiliser.

En Java, ouvrir une transaction revient à désactiver l'auto-commit :

```
String url =
    "jdbc:postgresql://localhost/test?user=fred&password=secret&ssl=true";
Connection conn = DriverManager.getConnection(url);
conn.setAutoCommit(false);
```

La transaction est confirmée (`COMMIT`) avec la méthode suivante :

```
conn.commit();
```

À l'inverse, elle est annulée (`ROLLBACK`) avec la méthode suivante :

```
conn.rollback();
```

2.4.4 Points de sauvegarde

- Certains traitements dans une transaction peuvent être annulés
  - mais la transaction est atomique
- Définir un point de sauvegarde
  - `SAVEPOINT nom_savepoint`
- Valider le traitement depuis le dernier point de sauvegarde
  - `RELEASE SAVEPOINT nom_savepoint`
- Annuler le traitement depuis le dernier point de sauvegarde
  - `ROLLBACK TO SAVEPOINT nom_savepoint`

déroule jusqu'au bout, le point de sauvegarde pourra être relâché (`RELEASE SAVEPOINT`), confirmant ainsi les traitements. Si le traitement tombe en erreur, il suffira de revenir au point de sauvegarde (`ROLLBACK TO SAVEPOINT` pour annuler uniquement cette partie du traitement sans affecter le reste de la transaction.

Les points de sauvegarde sont des éléments nommés, il convient donc de leur affecter un nom particulier. Leur nom doit être unique dans la transaction courante.

Les langages de programmation permettent également de gérer les points de sauvegarde en utilisant des méthodes dédiées. Par exemple, en Java :

```
Savepoint save1 = connection.setSavepoint();
```

En cas d'erreurs, la transaction peut être ramener à l'état du point de sauvegarde avec :

```
connection.rollback(save1);
```

À l'inverse, un point de sauvegarde est relâché de la façon suivante :

```
connection.releaseSavepoint(save1);
```

Exemples

Transaction avec un point de sauvegarde et la gestion de l'erreur :

```
BEGIN;

INSERT INTO mere (id, val_mere) VALUES (10, 'essai');

SAVEPOINT insert_fille;

INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (1, 10, 'essai 2');
ERROR:  duplicate key value violates unique constraint "fille_pkey"
DETAIL:  Key (id_fille)=(1) already exists.

ROLLBACK TO SAVEPOINT insert_fille;

COMMIT;

SELECT * FROM mere;
 id | val_mere
----+----------
  1 | mere 1
  2 | mere 2
 10 | essai
```

2.5 CONCLUSION

- SQL : toujours un traitement d'ensembles d'enregistrements
  - c'est le côté relationnel
- Pour les définitions d'objets
  - CREATE, ALTER, DROP
- Pour les données
  - INSERT, UPDATE, DELETE

Le standard SQL permet de traiter des ensembles d'enregistrements, que ce soit en lecture, en insertion, en modification et en suppression. Les ensembles d'enregistrements sont généralement des tables qui, comme tous les autres objets, sont créées (CREATE), modifier (ALTER) et/ou supprimer (DROP).

2.5.1 Questions

N'hésitez pas, c'est le moment !

2.6 TRAVAUX PRATIQUES

Cet exercice 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
```

Pour cet exercice, les modifications de schéma doivent être effectuées par un rôle ayant suffisamment de droits pour modifier son schéma. Le rôle **tpc_admin** a les droits suffisants.

1. Ajouter une colonne `email` de type `text` à la table `contacts`. Cette colonne va permettre de stocker l'adresse e-mail des clients et des fournisseurs. Ajouter également un commentaire décrivant cette colonne dans le catalogue de PostgreSQL (utiliser la commande `COMMENT`).
2. Mettre à jour la table des contacts pour indiquer l'adresse e-mail de *Client6657* qui est `client6657@dalibo.com`.
3. Ajouter une contrainte d'intégrité qui valide que la valeur de la colonne `email` créée est bien formée (vérifier que la chaîne de caractère contient au moins le caractère @).
4. Valider la contrainte dans une transaction de test.
5. Déterminer quels sont les contacts qui disposent d'une adresse e-mail et affichez leur nom ainsi que le code de leur pays.
6. La génération des numéros de commande est actuellement réalisée à l'aide de la séquence `commandes_commande_id_seq`. Cette méthode ne permet pas de garantir que tous les numéros de commande se suivent. Proposer une solution pour sérialiser la génération des numéros de commande. Autrement dit, proposer une méthode pour obtenir un numéro de commande sans avoir de « trou » dans la séquence en cas d'échec d'une transaction.
7. Noter le nombre de lignes de la table `pieces`. Dans une transaction, majorer de 5% le prix des pièces de moins de 1500 € et minorer de 5 % le prix des pièces dont le prix actuel est égal ou supérieur à 1500 €. Vérifier que le nombre de lignes mises à jour au total correspond au nombre total de lignes de la table `pieces`.
8. Dans une même transaction, créer un nouveau client en incluant l'ajout de l'ensemble des informations requises pour pouvoir le contacter. Un nouveau client a un solde égal à 0.

2.7 TRAVAUX PRATIQUES (SOLUTIONS)

1. Ajouter une colonne `email` de type `text` à la table `contacts`. Cette colonne va permettre de stocker l'adresse e-mail des clients et des fournisseurs. Ajouter également un commentaire décrivant cette colonne dans le catalogue de PostgreSQL (utiliser la commande `COMMENT`).

```
ALTER TABLE contacts
   ADD COLUMN email text
;

COMMENT ON COLUMN contacts.email IS
   'Adresse e-mail du contact'
;
```

2. Mettre à jour la table des contacts pour indiquer l'adresse e-mail de *Client6657* qui est `client6657@dalibo.com`.

```
UPDATE contacts
   SET email = 'client6657@dalibo.com'
WHERE nom = 'Client6657'
;
```

Vérifier les résultats :

```
SELECT *
FROM contacts
WHERE nom = 'Client6657'
;
```

3. Ajouter une contrainte d'intégrité qui valide que la valeur de la colonne `email` créée est bien formée (vérifier que la chaîne de caractère contient au moins le caractère `@`).

```
ALTER TABLE contacts
ADD CONSTRAINT chk_contacts_email_valid
   CHECK (email LIKE '%@%')
;
```

Cette expression régulière est simplifiée et simpliste pour les besoins de l'exercice. Des expressions régulières plus complexes permettent de valider réellement une adresse e-mail.

Voici un exemple un tout petit peu plus évolué en utilisant une expression rationnelle simple, ici pour vérifier que la chaîne précédent le caractère `@` contient au moins un caractère, et que la chaîne le suivant est une chaîne de caractères contenant un point :

```
ALTER TABLE contacts
ADD CONSTRAINT chk_contacts_email_valid
   CHECK (email ~ '.+@.+\..+')
;
```

4. Valider la contrainte dans une transaction de test.

Démarrer la transaction :

```
BEGIN ;
```

Tenter de mettre à jour la table `contacts` avec une adresse e-mail ne répondant pas à la contrainte :

```
UPDATE contacts
   SET email = 'test'
;
```

L'ordre UPDATE retourne l'erreur suivante, indiquant que l'expression régulière est fonctionnelle :

```
ERROR:  new row for relation "contacts" violates check constraint
        "chk_contacts_email_valid"
DETAIL: Failing row contains
        (300001, Client1737, nkD, SA, 20-999-929-1440, test).
```

La transaction est ensuite annulée :

```
ROLLBACK ;
```

5. Déterminer quels sont les contacts qui disposent d'une adresse e-mail et afficher leur nom ainsi que le code de leur pays.

```
SELECT nom, code_pays
FROM contacts
WHERE email IS NOT NULL
;
```

6. La génération des numéros de commande est actuellement réalisée à l'aide de la séquence `commandes_commande_id_seq`. Cette méthode ne permet pas de garantir que tous les numéros de commande se suivent. Proposer une solution pour sérialiser la génération des numéros de commande. Autrement dit, proposer une méthode transactionnelle pour obtenir un numéro de commande, sans avoir de « trou » dans la séquence en cas d'échec d'une transaction.

La solution la plus simple pour imposer la sérialisation des numéros de commandes est d'utiliser une table de séquences. Une ligne de cette table correspondra au compteur des numéros de commande.

```
-- création de la table qui va contenir la séquence :
CREATE TABLE numeros_sequences (
   nom text NOT NULL PRIMARY KEY,
   sequence integer NOT NULL
)
;

-- initialisation de la séquence :
INSERT INTO numeros_sequences (nom, sequence)
SELECT 'sequence_numero_commande', max(numero_commande)
FROM commandes
;
```

L'obtention d'un nouveau numéro de commande sera réalisé dans la transaction de création de la commande de la façon suivante :

```
BEGIN ;

UPDATE numeros_sequences
```

```
   SET sequence = sequence + 1
WHERE nom = 'numero_commande'
RETURNING sequence
;

/* insertion d'une nouvelle commande en utilisant le numéro de commande
   retourné par la commande précédente :
   INSERT INTO commandes (numero_commande, ...)
   VALUES (<la nouvelle valeur de la séquence>, ...) ;
*/

COMMIT ;
```

L'ordre UPDATE pose un verrou exclusif sur la ligne mise à jour. Tant que la mise à jour n'aura pas été validée ou annulée par COMMIT ou ROLLBACK, le verrou posé va bloquer toutes les autres transactions qui tenteraient de mettre à jour cette ligne. De cette façon, toutes les transactions seront sérialisées.

Concernant la génération des numéros de séquence, si la transaction est annulée, alors le compteur `sequence` retrouvera sa valeur précédente et la transaction suivante obtiendra le même numéro de séquence. Si la transaction est validée, alors le compteur `sequence` est incrémenté. La transaction suivante verra alors cette nouvelle valeur et non plus l'ancienne. Cette méthode garantit qu'il n'y ait pas de rupture de séquence.

Il va de soi que les transactions de création de commandes doivent être extrêmement courtes. Si une telle transaction est bloquée, toutes les transactions suivantes seront également bloquées, paralysant ainsi tous les utilisateurs de l'application.

7. Noter le nombre de lignes de la table `pieces`. Dans une transaction, majorer de 5 % le prix des pièces de moins de 1500 € et minorer de 5 % le prix des pièces dont le prix actuel est égal ou supérieur à 1500 €. Vérifier que le nombre de lignes mises à jour au total correspond au nombre total de lignes de la table `pieces`.

```
BEGIN ;

SELECT count(*)
FROM pieces
;

UPDATE pieces
   SET prix = prix * 1.05
WHERE prix < 1500
;

UPDATE pieces
   SET prix = prix * 0.95
WHERE prix >= 1500
;
```

Au total, la transaction a mis à jour 214200 (99922+114278) lignes, soit 14200 lignes de trop mises à jour.

Annuler la mise à jour :

```
ROLLBACK ;
```

Explication : Le premier UPDATE a majoré de 5 % les pièces dont le prix est inférieur à 1500 €. Or, tous les prix supérieurs à 1428,58 € passent la barre des 1500 € après le premier UPDATE. Le second UPDATE minore les pièces dont le prix est égal ou supérieur à 1500 €, ce qui inclue une partie des prix majorés par le précédent UPDATE. Certaines lignes ont donc subies *deux* modifications au lieu d'une. L'instruction CASE du langage SQL, qui sera abordée dans le prochain module, propose une solution à ce genre de problématique :

```
UPDATE pieces
   SET prix = (
      CASE
         WHEN prix < 1500 THEN prix * 1.05
         WHEN prix >= 1500 THEN prix * 0.95
      END
   )
;
```

8. Dans une même transaction, créer un nouveau client en incluant l'ajout de l'ensemble des informations requises pour pouvoir le contacter. Un nouveau client a un solde égal à 0.

```
-- démarrer la transaction
BEGIN ;

-- créer le contact et récupérer le contact_id généré
INSERT INTO contacts (nom, adresse, telephone, code_pays)
   VALUES ('M. Xyz', '3, Rue du Champignon, 96000 Champiville',
           '+33554325432', 'FR')
RETURNING contact_id
;

-- réaliser l'insertion en utilisant le numéro de contact récupéré précédemment
INSERT INTO clients (solde, segment_marche, contact_id, commentaire)
   -- par exemple ici avec le numéro 350002
   VALUES (0, 'AUTOMOBILE', 350002, 'Client très important')
;

-- valider la transaction
COMMIT ;
```

3/ Plus loin avec SQL

3.1 PRÉAMBULE

- Après la définition des objets, leur lecture et leur écriture
- Aller plus loin dans l'écriture de requêtes avec :
  - les jointures
  - les sous-requêtes
  - les vues
  - les fonctions

Maintenant que nous avons vu comment définir des objets, comment lire des données provenant de relation et comment écrire des données, nous allons pousser vers les perfectionnements du langage SQL. Nous allons notamment aborder la lecture de plusieurs tables en même temps, que ce soit par des jointures ou par des sous-requêtes.

3.1.1 Menu

- Valeur `NULL`
- Agrégats, `GROUP BY`, `HAVING`
- Sous-requêtes
- Jointures
- Expression conditionnelle `CASE`
- Opérateurs ensemblistes : `UNION`, `EXCEPT`, `INTERSECT`

3.1.2 Menu (suite)

- Fonctions de base
- Vues
- Requêtes préparées

3.1.3 Objectifs

- Comprendre l'intérêt du NULL
- Savoir écrire des requêtes complexes

3.2 VALEUR NULL

- Comment représenter une valeur que l'on ne connaît pas ?
  - valeur NULL
- Trois sens possibles pour NULL :
  - valeur inconnue
  - valeur inapplicable
  - absence de valeur
- Logique 3 états

Le standard SQL définit très précisément la valeur que doit avoir une colonne dont on ne connaît pas la valeur. Il faut utiliser le mot clé NULL. En fait, ce mot clé est utilisé dans trois cas : pour les valeurs inconnues, pour les valeurs inapplicables et pour une absence de valeurs.

3.2.1 Avertissement

- Chris J. Date a écrit :
  - *La valeur NULL telle qu'elle est implémentée dans SQL peut poser plus de problèmes qu'elle n'en résout. Son comportement est parfois étrange et est source de nombreuses erreurs et de confusions.*
- Éviter d'utiliser NULL le plus possible
  - utiliser NULL correctement lorsqu'il le faut

Il ne faut utiliser NULL que lorsque cela est réellement nécessaire. La gestion des valeurs NULL est souvent source de confusions et d'erreurs, ce qui explique qu'il est préférable de l'éviter tant qu'on n'entre pas dans les trois cas vu ci-dessus (valeur inconnue, valeur inapplicable, absence de valeur).

3.2.2 Assignation de NULL

- Assignation de NULL pour INSERT et UPDATE
- Explicitement :
  - NULL est indiqué explicitement dans les assignations
- Implicitement :
  - la colonne n'est pas affectée par INSERT
  - et n'a pas de valeur par défaut
- Empêcher la valeur NULL
  - contrainte NOT NULL

Il est possible de donner le mot-clé NULL pour certaines colonnes dans les INSERT et les UPDATE. Si jamais une colonne n'est pas indiquée dans un INSERT, elle aura comme valeur sa valeur par défaut (très souvent, il s'agit de NULL). Si jamais on veut toujours avoir une valeur dans une colonne particulière, il faut utiliser la clause NOT NULL lors de l'ajout de la colonne. C'est le cas pour les clés primaires par exemple.

Voici quelques exemples d'insertion et de mise à jour :

```
CREATE TABLE public.personnes
(
   id serial,
   nom character varying(60) NOT NULL,
   prenom character varying(60),
   date_naissance date,
   CONSTRAINT pk_personnes PRIMARY KEY (id)
) ;

INSERT INTO personnes( nom, prenom, date_naissance )
    VALUES ('Lagaffe', 'Gaston', date '1957-02-28') ;

-- assignation explicite

INSERT INTO personnes( nom, prenom, date_naissance )
    VALUES ('Fantasio', NULL, date '1938-01-01') ;

-- assignation implicite

INSERT INTO personnes ( nom, prenom )
    VALUES ('Prunelle', 'Léon') ;

-- observation des résultats
SELECT * FROM personnes ;
```

```
 id |   nom    | prenom | date_naissance
----+----------+--------+----------------
  1 | Lagaffe  | Gaston | 1957-02-28
  2 | Fantasio | (null) | 1938-01-01
  3 | Prunelle | Léon   | (null)
```

L'affichage `(null)` dans `psql` est obtenu avec la méta-commande :

```
\pset null (null)
```

3.2.3 Calculs avec NULL

- Utilisation dans un calcul
  - propagation de NULL
- NULL est inapplicable
  - le résultat vaut NULL

La valeur NULL est définie comme inapplicable. Ainsi, si elle présente dans un calcul, elle est propagée sur l'ensemble du calcul : le résultat vaudra NULL.

Exemples de calcul

Calculs simples :

```
SELECT 1 + 2 AS resultat ;
```

```
 resultat
----------
        3
```

```
SELECT 1 + 2 + NULL AS resultat ;
```

```
 resultat
----------
  (null)
```

Calcul à partir de l'âge :

```
SELECT nom, prenom,
    1 + extract('year' from age(date_naissance)) AS calcul_age
FROM personnes ;
```

```
   nom    | prenom | calcul_age
----------+--------+------------
 Lagaffe  | Gaston |         60
 Fantasio | (null) |         79
 Prunelle | Léon   |     (null)
```

Exemple d'utilisation de NULL dans une concaténation :

```
SELECT nom || ' ' || prenom AS nom_complet
FROM personnes ;

  nom_complet
----------------
 Lagaffe Gaston
 (null)
 Prunelle Léon
```

L'affichage `(null)` est obtenu avec la méta-commande `\pset null (null)` du shell psql.

3.2.4 NULL et les prédicats

- Dans un prédicat du WHERE :
  - opérateur `IS NULL` ou `IS NOT NULL`
- AND :
  - vaut `false` si `NULL AND false`
  - vaut NULL si `NULL AND true` ou `NULL AND NULL`
- OR :
  - vaut `true` si `NULL OR true`
  - vaut NULL si `NULL OR false` ou `NULL OR NULL`

Les opérateurs de comparaisons classiques ne sont pas fonctionnels avec une valeur NULL. Du fait de la logique à trois états de PostgreSQL, une comparaison avec NULL vaut toujours NULL, ainsi `expression = NULL` vaudra toujours NULL et de même pour `expression <> NULL` vaudra toujours NULL. Cette comparaison ne vaudra jamais ni vrai, ni faux.

De ce fait, il existe les opérateurs de prédicats `IS NULL` et `IS NOT NULL` qui permettent de vérifier qu'une expression est NULL ou n'est pas NULL.

Pour en savoir plus sur la logique ternaire qui régit les règles de calcul des prédicats, se conformer à la page Wikipédia sur la logique ternaire[1].

Exemples

Comparaison directe avec NULL, qui est invalide :

```
SELECT * FROM personnes WHERE date_naissance = NULL ;
```

[1] https://fr.wikipedia.org/wiki/Logique_ternaire

```
 id | nom | prenom | date_naissance
----+-----+--------+----------------
(0 rows)
```

L'opérateur `IS NULL` permet de retourner les lignes dont la date de naissance n'est pas renseignée :

```
SELECT * FROM personnes WHERE date_naissance IS NULL ;
```

```
 id |   nom    | prenom | date_naissance
----+----------+--------+----------------
  3 | Prunelle | Léon   | (null)
```

3.2.5 NULL et les agrégats

- Opérateurs d'agrégats
  - ignorent NULL
  - sauf `count(*)`

Les fonctions d'agrégats ne tiennent pas compte des valeurs NULL :

```
SELECT SUM(extract('year' from age(date_naissance))) AS age_cumule
FROM personnes ;
```

```
 age_cumule
------------
        139
```

Sauf `count(*)` et uniquement `count(*)`, la fonction `count(_expression_)` tient compte des valeurs NULL :

```
SELECT count(*) AS compte_lignes, count(date_naissance) AS compte_valeurs
  FROM (SELECT date_naissance
          FROM personnes) date_naissance ;
```

```
 compte_lignes | compte_valeurs
---------------+----------------
             3 |              2
```

3.2.6 COALESCE

- Remplacer NULL par une autre valeur
    - `COALESCE(attribut, ...);`

Cette fonction permet de tester une colonne et de récupérer sa valeur si elle n'est pas NULL et une autre valeur dans le cas contraire. Elle peut avoir plus de deux arguments. Dans ce cas, la première expression de la liste qui ne vaut pas NULL sera retournée par la fonction.

Voici quelques exemples :

Remplace les prénoms non-renseignés par la valeur X dans le résultat :

```
SELECT nom, COALESCE(prenom, 'X') FROM personnes ;
```

```
   nom    | coalesce
----------+----------
 Lagaffe  | Gaston
 Fantasio | X
 Prunelle | Léon
```

Cette fonction est efficace également pour la concaténation précédente :

```
SELECT nom || ' ' || COALESCE(prenom, '') AS nom_complet FROM personnes ;
```

```
  nom_complet
----------------
 Lagaffe Gaston
 Fantasio
 Prunelle Léon
```

3.3 AGRÉGATS

- Regroupement de données
- Calculs d'agrégats

Comme son nom l'indique, l'agrégation permet de regrouper des données, qu'elles viennent d'une ou de plusieurs colonnes. Le but est principalement de réaliser des calculs sur les données des lignes regroupées.

3.3.1 Regroupement de données

- Regroupement de données : `sql GROUP BY expression [, ...]`
- Chaque groupe de données est ensuite représenté sur une seule ligne
- Permet d'appliquer des calculs sur les ensembles regroupés
  - comptage, somme, moyenne, etc.

La clause GROUP BY permet de réaliser des regroupements de données. Les données regroupées sont alors représentées sur une seule ligne. Le principal intérêt de ces regroupements est de permettre de réaliser des calculs sur ces données.

3.3.2 Calculs d'agrégats

- Effectuent un calcul sur un ensemble de valeurs
  - somme, moyenne, etc.
- Retournent `NULL` si l'ensemble est vide
  - sauf `count()`

Nous allons voir les différentes fonctions d'agrégats disponibles.

3.3.3 Agrégats simples

- Comptage : `sql count(expression)`
- compte les lignes : `count(*)`
  - compte les valeurs renseignées : `count(colonne)`
- Valeur minimale : `sql min(expression)`
- Valeur maximale : `sql max(expression)`

La fonction `count()` permet de compter les éléments. La fonction est appelée de deux façons.

La première forme consiste à utiliser `count(*)` qui revient à transmettre la ligne complète à la fonction d'agrégat. Ainsi, toute ligne transmise à la fonction sera comptée, même si elle n'est composée que de valeurs NULL. On rencontre parfois une forme du type `count(1)`, qui transmet une valeur arbitraire à la fonction, et qui permettait d'accélérer le temps de traitement sur certains SGBD mais qui reste sans intérêt avec PostgreSQL.

La seconde forme consiste à utiliser une expression, par exemple le nom d'une colonne : `count(nom_colonne)`. Dans ce cas-là, seules les valeurs renseignées, donc non NULL, seront prises en compte. Les valeurs NULL seront exclues du comptage.

La fonction `min()` permet de déterminer la valeur la plus petite d'un ensemble de valeurs données. La fonction `max()` permet à l'inverse de déterminer la valeur la plus grande d'un ensemble de valeurs données. Les valeurs NULL sont bien ignorées. Ces deux fonctions permettent de travailler sur des données numériques, mais fonctionnent également sur les autres types de données comme les chaînes de caractères.

La documentation de PostgreSQL permet d'obtenir la liste des fonctions d'agrégats disponibles[2].

Exemples :

Différences entre count(*) et count(colonne) :

```
CREATE TABLE test (x INTEGER) ;
-- insertion de cinq lignes dans la table test
INSERT INTO test (x) VALUES (1), (2), (2), (NULL), (NULL) ;

SELECT x, count(*) AS count_etoile, count(x) AS count_x
FROM test
GROUP BY x ;
```

```
   x    | count_etoile | count_x
--------+--------------+---------
 (null) |            2 |       0
      1 |            1 |       1
      2 |            2 |       2
```

[2] http://docs.postgresql.fr/current/functions-aggregate.html

Déterminer la date de naissance de la personne la plus jeune :

```
SELECT MAX(date_naissance) FROM personnes ;
```

```
    max
------------
 1957-02-28
```

3.3.4 Calculs d'agrégats

- Moyenne : `sql avg(expression)`
- Somme : `sql sum(expression)`
- Écart-type : `sql stddev(expression)`
- Variance : `sql variance(expression)`

La fonction `avg()` permet d'obtenir la moyenne d'un ensemble de valeurs données. La fonction `sum()` permet, quant à elle, d'obtenir la somme d'un ensemble de valeurs données. Enfin, les fonctions `stddev()` et `variance()` permettent d'obtenir respectivement l'écart-type et la variance d'un ensemble de valeurs données.

Ces fonctions retournent `NULL` si aucune donnée n'est applicable. Elles ne prennent en compte que des valeurs numériques.

La documentation de PostgreSQL permet d'obtenir la liste des fonctions d'agrégats disponibles[3].

Exemples

Quel est le nombre total de bouteilles en stock par millésime ?

```
SELECT annee, sum(nombre)
FROM stock
GROUP BY annee
ORDER BY annee ;
```

```
 annee |  sum
-------+--------
  1950 | 210967
  1951 | 201977
  1952 | 202183
...
```

Calcul de moyenne avec des valeurs NULL :

```
CREATE TABLE test (a int, b int) ;

INSERT INTO test VALUES (10,10) ;
INSERT INTO test VALUES (20,20) ;
```

[3] https://docs.postgresql.fr/current/functions-aggregate.html

```
INSERT INTO test VALUES (30,30) ;
INSERT INTO test VALUES (null,0) ;

SELECT avg(a), avg(b) FROM test ;

         avg          |         avg
----------------------+---------------------
 20.0000000000000000 | 15.0000000000000000
```

3.3.5 Agrégats sur plusieurs colonnes

- Possible d'avoir plusieurs paramètres sur la même fonction d'agrégat
- Quelques exemples
  - pente : `regr_slope(Y,X)`
  - intersection avec l'axe des ordonnées : `regr_intercept(Y,X)`
  - indice de corrélation : `corr (Y,X)`

Une fonction d'agrégat peut aussi prendre plusieurs variables.

Par exemple concernant la méthode des « moindres carrés » :

- pente : `regr_slope(Y,X)`
- intersection avec l'axe des ordonnées : `regr_intercept(Y,X)`
- indice de corrélation : `corr (Y,X)`

Voici un exemple avec un nuage de points proches d'une fonction **y=2x+5** :

```
CREATE TABLE test (x real, y real) ;
INSERT INTO test VALUES (0,5.01), (1,6.99), (2,9.03) ;

SELECT regr_slope(y,x) FROM test ;

    regr_slope
------------------
 2.00999975204468

SELECT regr_intercept(y,x) FROM test ;

  regr_intercept
------------------
 5.00000015894572

SELECT corr(y,x) FROM test ;

       corr
-------------------
 0.999962873745297
```

3.3.6 Clause HAVING

- Filtrer sur des regroupements
  - HAVING
- WHERE s'applique sur les lignes lues
- HAVING s'applique sur les lignes groupées

La clause HAVING permet de filtrer les résultats sur les regroupements réalisés par la clause GROUP BY. Il est possible d'utiliser une fonction d'agrégat dans la clause HAVING.

Il faudra néanmoins faire attention à ne pas utiliser la clause HAVING comme clause de filtrage des données lues par la requête. La clause HAVING ne doit permettre de filtrer que les données traitées par la requête.

Ainsi, si l'on souhaite le nombre de vins rouge référencés dans le catalogue. La requête va donc exclure toutes les données de la table vin qui ne correspondent pas au filtre `type_vin = 3`. Pour réaliser cela, on utilisera la clause WHERE.

En revanche, si l'on souhaite connaître le nombre de vins par type de cépage si ce nombre est supérieur à 2030, on utilisera la clause HAVING.

Exemples

```
SELECT type_vin_id, count(*)
  FROM vin
 GROUP BY type_vin_id
HAVING count(*) > 2030 ;
```

```
 type_vin_id | count
-------------+-------
           1 |  2031
```

Si la colonne correspondant à la fonction d'agrégat est renommée avec la clause AS, il n'est pas possible d'utiliser le nouveau nom au sein de la clause HAVING. Par exemple :

```
SELECT type_vin_id, count(*) AS nombre
  FROM vin
 GROUP BY type_vin_id
HAVING nombre > 2030 ;
```

```
ERROR:  column "nombre" does not exist
```

3.4 SOUS-REQUÊTES

- Corrélation requête/sous-requête
- Sous-requêtes retournant une seule ligne
- Sous-requêtes retournant une liste de valeur
- Sous-requêtes retournant un ensemble
- Sous-requêtes retournant un ensemble vide ou non-vide

3.4.1 Corrélation requête/sous-requête

- Fait référence à la requête principale
- Peut utiliser une valeur issue de la requête principale

Une sous-requête peut faire référence à des variables de la requête principale. Ces variables seront ainsi transformées en constante à chaque évaluation de la sous-requête.

La corrélation requête/sous-requête permet notamment de créer des clauses de filtrage dans la sous-requête en utilisant des éléments de la requête principale.

3.4.2 Qu'est-ce qu'une sous-requête ?

- Une requête imbriquée dans une autre requête
- Le résultat de la requête principale dépend du résultat de la sous-requête
- Encadrée par des parenthèses : `(` et `)`

Une sous-requête consiste à exécuter une requête à l'intérieur d'une autre requête. La requête principale peut être une requête de sélection (`SELECT`) ou une requête de modification (`INSERT`, `UPDATE`, `DELETE`). La sous-requête est obligatoirement un `SELECT`.

Le résultat de la requête principale dépend du résultat de la sous-requête. La requête suivante effectue la sélection des colonnes d'une autre requête, qui est une sous-requête. La sous-requête effectue une lecture de la table `appellation`. Son résultat est transformé en un ensemble qui est nommé `requete_appellation` :

```
SELECT * FROM
  (SELECT libelle, region_id
   FROM appellation          )  requete_appellation ;

                  libelle                  | region_id
-------------------------------------------+-----------
 Ajaccio                                   |         1
 Aloxe-Corton                              |         2
...
```

3.4.3 Utiliser une seule ligne

- La sous-requête ne retourne qu'une seule ligne
  - sinon une erreur est levée
- Positionnée
  - au niveau de la liste des expressions retournées par `SELECT`
  - au niveau de la clause `WHERE`
  - au niveau d'une clause `HAVING`

La sous-requête peut être positionnée au niveau de la liste des expressions retournées par `SELECT`. La sous-requête est alors généralement un calcul d'agrégat qui ne donne en résultat qu'une seule colonne sur une seule ligne. Ce type de sous-requête est peu performant. Elle est en effet appelée pour chaque ligne retournée par la requête principale.

La requête suivante permet d'obtenir le cumul du nombre de bouteilles année par année.

```
SELECT annee,
       sum(nombre) AS stock,
       (SELECT sum(nombre)
          FROM stock s
         WHERE s.annee <= stock.annee) AS stock_cumule
  FROM stock
 GROUP BY annee
 ORDER BY annee ;

 annee | stock  | stock_cumule
-------+--------+--------------
  1950 | 210967 |       210967
  1951 | 201977 |       412944
  1952 | 202183 |       615127
  1953 | 202489 |       817616
  1954 | 202041 |      1019657
...
```

Une telle sous-requête peut également être positionnée au niveau de la clause `WHERE` ou de la clause `HAVING`.

Par exemple, pour retourner la liste des vins rouge :

```
SELECT *
  FROM vin
 WHERE type_vin_id = (SELECT id
                        FROM type_vin
                       WHERE libelle = 'rouge') ;
```

3.4.4 Utiliser une liste de valeurs

- La sous-requête retourne
  - plusieurs lignes
  - sur une seule colonne
- Positionnée
  - avec une clause `IN`
  - avec une clause `ANY`
  - avec une clause `ALL`

Les sous-requêtes retournant une liste de valeur sont plus fréquemment utilisées. Ce type de sous-requête permet de filtrer les résultats de la requête principale à partir des résultats de la sous-requête.

3.4.5 Clause IN

```
expression IN (sous-requete)
```

- L'expression de gauche est évaluée et vérifiée avec la liste de valeurs de droite
- `IN` vaut `true`
  - si l'expression de gauche correspond à un élément de la liste de droite
- `IN` vaut `false`
  - si aucune correspondance n'est trouvée et la liste ne contient pas `NULL`
- `IN` vaut `NULL`
  - si l'expression de gauche vaut `NULL`
  - si aucune valeur ne correspond et la liste contient `NULL`

La clause `IN` dans la requête principale permet alors d'exploiter le résultat de la sous-requête pour sélectionner les lignes dont une colonne correspond à une valeur retournée par la sous-requête.

L'opérateur `IN` retourne `true` si la valeur de l'expression de gauche est trouvée au moins une fois dans la liste de droite. La liste de droite peut contenir la valeur `NULL` dans ce cas :

```
SELECT 1 IN (1, 2, NULL) AS in ;
```

```
 in
 t
```

Si aucune correspondance n'est trouvée entre l'expression de gauche et la liste de droite, alors `IN` vaut `false` :

```
SELECT 1 IN (2, 4) AS in ;
```

```
 in
 f
```

Mais `IN` vaut `NULL` si aucune correspondance n'est trouvée et que la liste de droite contient au moins une valeur `NULL` :

```
SELECT 1 IN (2, 4, NULL) AS in ;
```

```
   in
--------
 (null)
```

`IN` vaut également `NULL` si l'expression de gauche vaut `NULL` :

```
SELECT NULL IN (2, 4) AS in ;
```

```
   in
--------
 (null)
```

Exemples

La requête suivante permet de sélectionner les bouteilles du stock de la cave dont la contenance est comprise entre 0,3 litre et 1 litre. Pour répondre à la question, la sous-requête retourne les identifiants de contenant qui correspondent à la condition. La requête principale ne retient alors que les lignes dont la colonne contenant_id correspond à une valeur d'identifiant retournée par la sous-requête.

```
SELECT *
  FROM stock
 WHERE contenant_id IN (SELECT id
                          FROM contenant
                         WHERE contenance
                       BETWEEN 0.3 AND 1.0) ;
```

3.4.6 Clause NOT IN

```
expression NOT IN (sous-requete)
```

- L'expression de droite est évaluée et vérifiée avec la liste de valeurs de gauche
- `NOT IN` vaut `true`
  - si aucune correspondance n'est trouvée et la liste ne contient pas `NULL`
- `NOT IN` vaut `false`
  - si l'expression de gauche correspond à un élément de la liste de droite
- `NOT IN` vaut `NULL`
  - si l'expression de gauche vaut `NULL`
  - si aucune valeur ne correspond et la liste contient `NULL`

À l'inverse, la clause `NOT IN` permet dans la requête principale de sélectionner les lignes dont la colonne impliquée dans la condition ne correspond pas aux valeurs retournées par la sous-requête.

La requête suivante permet de sélectionner les bouteilles du stock dont la contenance n'est pas inférieure à 2 litres.

```
SELECT *
  FROM stock
 WHERE contenant_id NOT IN (SELECT id
                              FROM contenant
                             WHERE contenance < 2.0) ;
```

Il est à noter que les requêtes impliquant les clauses `IN` ou `NOT IN` peuvent généralement être réécrites sous la forme d'une jointure.

De plus, les optimiseurs SQL parviennent difficilement à optimiser une requête impliquant `NOT IN`. Il est préférable d'essayer de réécrire ces requêtes en utilisant une jointure.

Avec `NOT IN`, la gestion des valeurs `NULL` est à l'inverse de celle de la clause `IN` :

Si une correspondance est trouvée, `NOT IN` vaut `false` :

```
SELECT 1 NOT IN (1, 2, NULL) AS notin ;

 notin
-------
 f
```

Si aucune correspondance n'est trouvée, `NOT IN` vaut `true` :

```
SELECT 1 NOT IN (2, 4) AS notin ;

 notin
-------
 t
```

Si aucune correspondance n'est trouvée mais que la liste de valeurs de droite contient au moins un `NULL`, `NOT IN` vaut `NULL` :

```
SELECT 1 NOT IN (2, 4, NULL) AS notin ;

 notin
--------
 (null)
```

Si l'expression de gauche vaut `NULL`, alors `NOT IN` vaut `NULL` également :

```
SELECT NULL IN (2, 4) AS notin ;

 notin
--------
 (null)
```

Les sous-requêtes retournant des valeurs `NULL` posent souvent des problèmes avec `NOT IN`. Il est préférable d'utiliser `EXISTS` ou `NOT EXISTS` pour ne pas avoir à se soucier des valeurs `NULL`.

3.4.7 Clause ANY

```
expression operateur ANY (sous-requete)
```

- L'expression de gauche est comparée au résultat de la sous-requête avec l'opérateur donné
- La ligne de gauche est retournée
  - si le résultat d'au moins une comparaison est vraie
- La ligne de gauche n'est pas retournée
  - si aucun résultat de la comparaison n'est vrai
  - si l'expression de gauche vaut NULL
  - si la sous-requête ramène un ensemble vide

La clause ANY, ou son synonyme SOME, permet de comparer l'expression de gauche à chaque ligne du résultat de la sous-requête en utilisant l'opérateur indiqué. Ainsi, la requête de l'exemple avec la clause IN aurait pu être écrite avec = ANY de la façon suivante :

```
SELECT *
  FROM stock
 WHERE contenant_id = ANY (SELECT id
                             FROM contenant
                            WHERE contenance
                          BETWEEN 0.3 AND 1.0) ;
```

3.4.8 Clause ALL

```
expression operateur ALL (sous-requete)
```

- L'expression de gauche est comparée à tous les résultats de la sous-requête avec l'opérateur donné
- La ligne de gauche est retournée
  - si tous les résultats des comparaisons sont vrais
  - si la sous-requête retourne un ensemble vide
- La ligne de gauche n'est pas retournée
  - si au moins une comparaison est fausse
  - si au moins une comparaison est NULL

La clause ALL permet de comparer l'expression de gauche à chaque ligne du résultat de la sous-requête en utilisant l'opérateur de comparaison indiqué.

La ligne de la table de gauche sera retournée si toutes les comparaisons sont vraies ou si la sous-requête retourne un ensemble vide. En revanche, la ligne de la table de gauche sera exclue si au moins une comparaison est fausse ou si au moins une comparaison est NULL.

La requête d'exemple de la clause NOT IN aurait pu être écrite avec <> ALL de la façon suivante :

```
SELECT *
  FROM stock
 WHERE contenant_id <> ALL (SELECT id
                              FROM contenant
                             WHERE contenance < 2.0) ;
```

3.4.9 Utiliser un ensemble

- La sous-requête retourne
  - plusieurs lignes
  - sur plusieurs colonnes
- Positionnée au niveau de la clause FROM
- Nommée avec un alias de table

La sous-requête peut être utilisée dans la clause FROM afin d'être utilisée comme une table dans la requête principale. La sous-requête devra obligatoirement être nommée avec un alias de table. Lorsqu'elles sont issues d'un calcul, les colonnes résultantes doivent également être nommées avec un alias de colonne afin d'éviter toute confusion ou comportement incohérent.

La requête suivante permet de déterminer le nombre moyen de bouteilles par année :

```
SELECT AVG(nombre_total_annee) AS moyenne
  FROM (SELECT annee, sum(nombre) AS nombre_total_annee
          FROM stock
         GROUP BY annee) stock_total_par_annee ;
```

3.4.10 Clause EXISTS

```
EXISTS (sous-requete)
```

- Intéressant avec une corrélation
- La clause EXISTS vérifie la présence ou l'absence de résultats
  - vrai si l'ensemble est non vide
  - faux si l'ensemble est vide

EXISTS présente peu d'intérêt sans corrélation entre la sous-requête et la requête principale.

Le prédicat EXISTS est en général plus performant que IN. Lorsqu'une requête utilisant IN ne peut pas être réécrite sous la forme d'une jointure, il est recommandé d'utiliser EXISTS en lieu et place de IN. Et à l'inverse, une clause NOT IN sera réécrite avec NOT EXISTS.

La requête suivante permet d'identifier les vins pour lesquels il y a au moins une bouteille en stock :

```
SELECT *
  FROM vin
 WHERE EXISTS (SELECT *
                 FROM stock
                WHERE vin_id = vin.id) ;
```

3.5 JOINTURES

- Conditions de jointure dans `JOIN` ou dans `WHERE` ?
- Produit cartésien
- Jointure interne
- Jointures externes
- Jointure ou sous-requête ?

Les jointures permettent d'écrire des requêtes qui impliquent plusieurs tables. Elles permettent de combiner les colonnes de plusieurs tables selon des critères particuliers, appelés conditions de jointures.

Les jointures permettent de tirer parti du modèle de données dans lequel les tables sont associées à l'aide de clés étrangères.

3.5.1 Conditions de jointure dans `JOIN` ou dans `WHERE` ?

- Jointure dans clause `JOIN`
  - séparation nette jointure et filtrage
  - plus lisible et maintenable
  - jointures externes propres
  - facilite le travail de l'optimiseur
- Jointure dans clause `WHERE`
  - uistorique

Bien qu'il soit possible de décrire une jointure interne sous la forme d'une requête `SELECT` portant sur deux tables dont la condition de jointure est décrite dans la clause `WHERE`, cette forme d'écriture n'est pas recommandée. Elle est essentiellement historique et se retrouve surtout dans des projets migrés sans modification.

En effet, les conditions de jointures se trouvent mélangées avec les clauses de filtrage, rendant ainsi la compréhension et la maintenance difficiles. Il arrive aussi que, noyé dans les autres conditions de filtrage, l'utilisateur oublie la configuration de jointure, ce qui aboutit à un produit cartésien, n'ayant rien à voir avec le résultat attendu, sans même parler de la lenteur de la requête.

Il est recommandé d'utiliser la syntaxe SQL:92 et d'exprimer les jointures à l'aide de la clause `JOIN`. D'ailleurs, cette syntaxe est la seule qui soit utilisable pour exprimer simplement et efficacement une jointure externe. Cette syntaxe facilite la compréhension de la requête mais facilite également le travail de l'optimiseur SQL qui peut déduire beaucoup plus rapidement les jointures qu'en analysant la clause `WHERE` pour déterminer les conditions de jointure et les tables auxquelles elles s'appliquent le cas échéant.

Comparer ces deux exemples d'une requête typique d'ERP pourtant simplifiée :

```
SELECT
    clients.numero,
    SUM(lignes_commandes.chiffre_affaire)
FROM
    lignes_commandes
    INNER JOIN commandes ON (lignes_commandes.commande_id = commandes.id)
    INNER JOIN clients   ON (commandes.client_id = clients.id)
    INNER JOIN addresses ON (clients.adresse_id = addresses.id)
    INNER JOIN pays      ON (adresses.pays_id = pays.id)
WHERE
    pays.code = 'FR'
    AND addresses.ville = 'Strasbourg'
    AND commandes.statut = 'LIVRÉ'
    AND clients.type = 'PARTICULIER'
    AND clients.actif IS TRUE
GROUP BY clients.numero ;
```

et :

```
SELECT
    clients.numero,
    SUM(lignes_commandes.chiffre_affaire)
FROM
    lignes_commandes,
    commandes,
    clients,
    addresses,
    pays
WHERE
    pays.code = 'FR'
    AND lignes_commandes.commande_id = commandes.id
    AND commandes.client_id = clients.id
    AND commandes.statut = 'LIVRÉ'
    AND clients.type = 'PARTICULIER'
    AND clients.actif IS TRUE
    AND clients.adresse_id = addresses.id
    AND adresses.pays_id = pays.id
    AND addresses.ville = 'Strasbourg'
GROUP BY clients.numero ;
```

3.5.2 Produit cartésien

- Clause `CROSS JOIN`
- Réalise toutes les combinaisons entre les lignes d'une table et les lignes d'une autre
- À éviter dans la mesure du possible
  - peu de cas d'utilisation
  - peu performant

Le produit cartésien peut être exprimé avec la clause de jointure `CROSS JOIN` :

```
-- préparation du jeu de données
CREATE TABLE t1 (i1 integer, v1 integer) ;
CREATE TABLE t2 (i2 integer, v2 integer) ;

INSERT INTO t1 (i1, v1) VALUES (0, 0), (1, 1) ;
INSERT INTO t2 (i2, v2) VALUES (2, 2), (3, 3) ;

-- requête CROSS JOIN
SELECT * FROM t1 CROSS JOIN t2 ;
```

```
 i1 | v1 | i2 | v2
----+----+----+----
  0 |  0 |  2 |  2
  0 |  0 |  3 |  3
  1 |  1 |  2 |  2
  1 |  1 |  3 |  3
```

Ou plus simplement, en listant les deux tables dans la clause `FROM` sans indiquer de condition de jointure :

```
SELECT * FROM t1, t2 ;
```

```
 i1 | v1 | i2 | v2
----+----+----+----
  0 |  0 |  2 |  2
  0 |  0 |  3 |  3
  1 |  1 |  2 |  2
  1 |  1 |  3 |  3
(4 rows)
```

Voici un autre exemple utilisant aussi un `NOT EXISTS` :

```
CREATE TABLE sondes (id_sonde int, nom_sonde text);
CREATE TABLE releves_horaires (
    id_sonde int,
    heure_releve timestamptz check
        (date_trunc('hour',heure_releve)=heure_releve),
```

```
    valeur numeric);

INSERT INTO sondes VALUES (1,'sonde 1'),
                          (2, 'sonde 2'),
                          (3, 'sonde 3') ;

INSERT INTO releves_horaires VALUES
  (1,'2013-01-01 12:00:00',10),
  (1,'2013-01-01 13:00:00',11),
  (1,'2013-01-01 14:00:00',12),
  (2,'2013-01-01 12:00:00',10),
  (2,'2013-01-01 13:00:00',12),
  (2,'2013-01-01 14:00:00',12),
  (3,'2013-01-01 12:00:00',10),
  (3,'2013-01-01 14:00:00',10) ;

-- quels sont les relevés manquants entre 12h et 14h ?

SELECT id_sonde,
       heures_releves
FROM sondes
CROSS JOIN generate_series('2013-01-01 12:00:00','2013-01-01 14:00:00',
    interval '1 hour') series(heures_releves)
WHERE NOT EXISTS
    (SELECT 1
     FROM releves_horaires
     WHERE releves_horaires.id_sonde=sondes.id_sonde
       AND releves_horaires.heure_releve=series.heures_releves) ;

 id_sonde |     heures_releves
----------+------------------------
        3 | 2013-01-01 13:00:00+01
```

3.5.3 Jointure interne

- Clause `INNER JOIN`
  - meilleure lisibilité
  - facilite le travail de l'optimiseur
- Joint deux tables entre elles
  - Selon une condition de jointure

Une jointure interne est considérée comme un produit cartésien accompagné d'une clause de jointure pour ne conserver que les lignes qui répondent à la condition de jointure. Les SGBD réalisent néanmoins l'opération plus simplement.

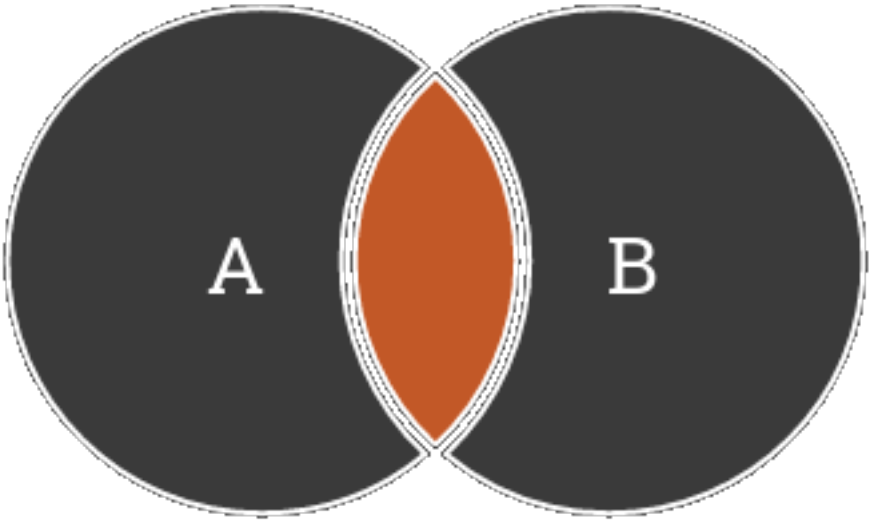


Figure 3/ .1: Schéma de jointure interne

La condition de jointure est généralement une égalité, ce qui permet d'associer entre elles les lignes de la table à gauche et de la table à droite dont les colonnes de condition de jointure sont égales.

La jointure interne est exprimée à travers la clause `INNER JOIN` ou plus simplement `JOIN`. En effet, si le type de jointure n'est pas spécifié, l'optimiseur considère la jointure comme étant une jointure interne.

3.5.4 Syntaxe d'une jointure interne

- Condition de jointure par prédicats : `sql table1 [INNER] JOIN table2 ON prédicat [...]`
- Condition de jointure implicite par liste des colonnes impliquées : `sql table1 [INNER] JOIN table2 USING (colonne [, ...])`
- Liste des colonnes de même nom (dangereux) : `sql table1 NATURAL [INNER] JOIN table2`

La clause `ON` permet d'écrire les conditions de jointures sous la forme de prédicats tels qu'on les retrouve dans une clause `WHERE`.

La clause `USING` permet de spécifier les colonnes sur lesquelles porte la jointure. Les tables jointes devront posséder toutes les colonnes sur lesquelles portent la jointure. La jointure sera réalisée en vérifiant l'égalité entre chaque colonne portant le même nom.

La clause `NATURAL` permet de réaliser la jointure entre deux tables en utilisant les colonnes qui

portent le même nom sur les deux tables comme condition de jointure. `NATURAL JOIN` est fortement déconseillé car elle peut facilement entraîner des comportements inattendus.

La requête suivante permet de joindre la table appellation avec la table `region` pour déterminer l'origine d'une appellation :

```
SELECT apl.libelle AS appellation, reg.libelle AS region
  FROM appellation apl
  JOIN region reg
    ON (apl.region_id = reg.id) ;
```

3.5.5 Jointure externe

- Jointure externe à gauche
  - ramène le résultat de la jointure interne
  - ramène l'ensemble de la table de gauche qui ne peut être joint avec la table de droite
  - les attributs de la table de droite sont alors `NULL`

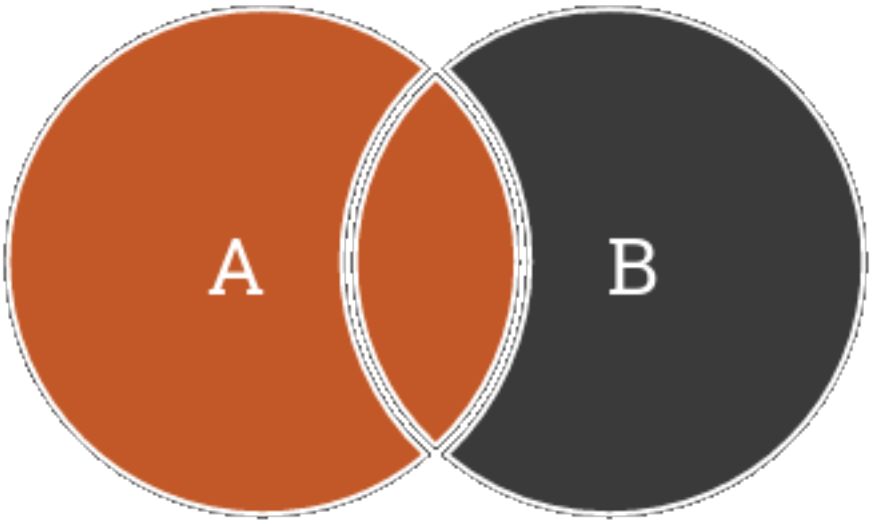


Figure 3/ .2: Schéma de jointure externe gauche

Il existe deux types de jointure externe : la jointure à gauche et la jointure à droite. Cela ne concerne que l'ordre de la jointure, le traitement en lui- même est identique.

3.5.6 Jointure externe - 2

- Jointure externe à droite
  - ramène le résultat de la jointure interne
  - ramène l'ensemble de la table de droite qui ne peut être joint avec la table de gauche
  - les attributs de la table de gauche sont alors NULL

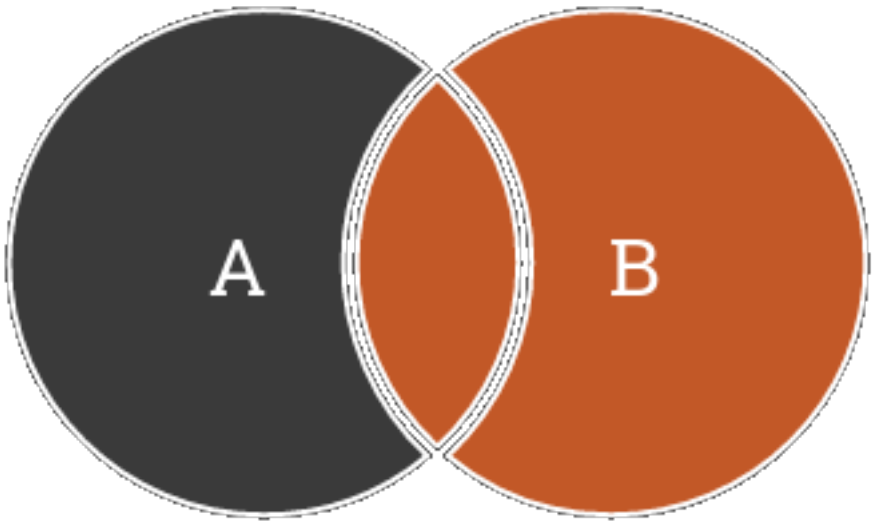


Figure 3/ .3: Schéma de jointure externe droite

3.5.7 Jointure externe complète

- Ramène le résultat de la jointure interne
- Ramène l'ensemble de la table de gauche qui ne peut être joint avec la table de droite
  - les attributs de la table de droite sont alors `NULL`
- Ramène l'ensemble de la table de droite qui ne peut être joint avec la table de gauche
  - les attributs de la table de gauche sont alors `NULL`

3.5.8 Syntaxe d'une jointure externe à gauche

- Condition de jointure par prédicats : `sql table1 LEFT [OUTER] JOIN table2 ON prédicat [...]`
- Condition de jointure implicite par liste des colonnes impliquées : `sql table1 LEFT [OUTER] JOIN table2 USING (colonne [, ...])`
- Liste des colonnes implicites : `sql table1 NATURAL LEFT [OUTER] JOIN table2`

Il existe trois écritures différentes d'une jointure externe à gauche. La clause `NATURAL` permet de réaliser la jointure entre deux tables en utilisant les colonnes qui portent le même nom sur les deux tables comme condition de jointure.

Les voici en exemple :

- par prédicat : `sql SELECT article.art_titre, auteur.aut_nom FROM article LEFT JOIN auteur ON (article.aut_id=auteur.aut_id) ;`
- par liste de colonnes : `sql SELECT article.art_titre, auteur.aut_nom FROM article LEFT JOIN auteur USING (aut_id) ;`

3.5.9 Syntaxe d'une jointure externe à droite

- Condition de jointure par prédicats : `sql table1 RIGHT [OUTER] JOIN table2 ON prédicat [...]`
- Condition de jointure implicite par liste des colonnes impliquées : `sql table1 RIGHT [OUTER] JOIN table2 USING (colonne [, ...])`
- Liste des colonnes implicites : `sql table1 NATURAL RIGHT [OUTER] JOIN table2`

Les jointures à droite sont moins fréquentes mais elles restent utilisées.

3.5.10 Syntaxe d'une jointure externe complète

- Condition de jointure par prédicats : `sql table1 FULL OUTER JOIN table2 ON prédicat [...]`
- Condition de jointure implicite par liste des colonnes impliquées : `sql table1 FULL OUTER JOIN table2 USING (colonne [, ...])`
- Liste des colonnes implicites : `sql table1 NATURAL FULL OUTER JOIN table2`

3.5.11 Jointure ou sous-requête ?

- Jointures
  - algorithmes très efficaces
  - ne gèrent pas tous les cas
- Sous-requêtes
  - parfois peu performantes
  - répondent à des besoins non couverts par les jointures

Les sous-requêtes sont fréquemment utilisées mais elles sont moins performantes que les jointures. Ces dernières permettent d'utiliser des optimisations très efficaces.

3.6 EXPRESSIONS CASE

- Équivalent à l'instruction `switch` en C ou Java
- Emprunté au langage Ada
- Retourne une valeur en fonction du résultat de tests

CASE permet de tester différents cas. Il s'utilise de la façon suivante :

```
SELECT
  CASE WHEN col1=10 THEN 'dix'
       WHEN col1>10 THEN 'supérieur à 10'
       ELSE 'inférieur à 10'
  END AS test
FROM t1;
```

3.6.1 CASE simple

```
CASE expression
     WHEN valeur THEN expression
     WHEN valeur THEN expression
     (...)
     ELSE expression
 END
```

Il est possible de tester le résultat d'une expression avec CASE. Dans ce cas, chaque clause WHEN reprendra la valeur à laquelle on souhaite associé une expression particulière :

```
CASE nom_region
  WHEN 'Afrique'  THEN 1
  WHEN 'Amérique' THEN 2
  WHEN 'Asie'     THEN 3
  WHEN 'Europe'   THEN 4
  ELSE 0
END
```

3.6.2 CASE sur expressions

```
CASE WHEN expression THEN expression
     WHEN expression THEN expression
     (...)
     ELSE expression
 END
```

Une expression peut être évaluée pour chaque clause WHEN. Dans ce cas, l'expression CASE retourne la première expression qui est vraie. Si une autre peut satisfaire la suivante, elle ne sera pas évaluée.

Par exemple :

```
CASE WHEN salaire * prime < 1300 THEN salaire * prime
     WHEN salaire * prime < 3000 THEN salaire
     WHEN salaire * prime > 5000 THEN salaire * prime
END
```

3.6.3 Spécificités de CASE

- Comportement procédural
  - les expressions sont évaluées dans l'ordre d'apparition
- Transtypage
  - le type du retour de l'expression dépend du type de rang le plus élevé de toute l'expression
- Imbrication
  - des expressions CASE à l'intérieur d'autres expressions CASE
- Clause ELSE
  - recommandé

Il est possible de placer plusieurs clauses WHEN. Elles sont évaluées dans leur ordre d'apparition.

```
CASE nom_region
  WHEN 'Afrique'  THEN 1
  WHEN 'Amérique' THEN 2
```

```
  /* l'expression suivante ne sera jamais évaluée */
  WHEN 'Afrique'  THEN 5
  WHEN 'Asie'     THEN 1
  WHEN 'Europe'   THEN 3
  ELSE 0
END
```

Le type de données renvoyé par l'instruction CASE correspond au type indiqué par l'expression au niveau des THEN et du ELSE. Ce doit être le même type. Si les types de données ne correspondent pas, alors PostgreSQL retournera une erreur :

```
SELECT *,
  CASE nom_region
    WHEN 'Afrique'  THEN 1
    WHEN 'Amérique' THEN 2
    WHEN 'Asie'     THEN 1
    WHEN 'Europe'   THEN 3
    ELSE 'inconnu'
  END
 FROM regions ;
ERROR:  invalid input syntax for integer: "inconnu"
LIGNE 7 :     ELSE 'inconnu'
```

La clause ELSE n'est pas obligatoire mais fortement recommandé. En effet, si une expression CASE ne comporte pas de clause ELSE, alors la base de données ajoutera une clause ELSE NULL à l'expression.

Ainsi l'expression suivante :

```
CASE
  WHEN salaire < 1000 THEN 'bas'
  WHEN salaire > 3000 THEN 'haut'
END
```

Sera implicitement transformée de la façon suivante :

```
CASE
  WHEN salaire < 1000 THEN 'bas'
  WHEN salaire > 3000 THEN 'haut'
  ELSE NULL
END
```

3.7 OPÉRATEURS ENSEMBLISTES

- UNION
- INTERSECT
- EXCEPT

3.7.1 Regroupement de deux ensembles

- Regroupement avec dédoublonnage :

```
requete_select1 UNION requete_select2
```

- Regroupement sans dédoublonnage :

```
requete_select1 UNION ALL requete_select2
```

L'opérateur ensembliste UNION permet de regrouper deux ensembles dans un même résultat.

Le dédoublonnage peut être particulièrement coûteux car il implique un tri des données.

Exemples

La requête suivante assemble les résultats de deux requêtes pour produire le résultat :

```
SELECT *
  FROM appellation
 WHERE region_id = 1
UNION ALL
SELECT *
  FROM appellation
 WHERE region_id = 3 ;
```

3.7.2 Intersection de deux ensembles

- Intersection de deux ensembles avec dédoublonnage :

```
requete_select1 INTERSECT requete_select2
```

- Intersection de deux ensembles sans dédoublonnage :

```
requete_select1 INTERSECT ALL requete_select2
```

L'opérateur ensembliste INTERSECT permet d'obtenir l'intersection du résultat de deux requêtes.

Le dédoublonnage peut être particulièrement coûteux car il implique un tri des données.

Exemples

L'exemple suivant n'a pas d'autre intérêt que de montrer le résultat de l'opérateur INTERSECT sur deux ensembles simples :

```
SELECT *
  FROM region
INTERSECT
SELECT *
  FROM region
  WHERE id = 3 ;
 id | libelle
----+---------
  3 | Alsace
```

3.7.3 Différence entre deux ensembles

- Différence entre deux ensembles avec dédoublonnage :

```
requete_select1 EXCEPT requete_select2
```

- Différence entre deux ensembles sans dédoublonnage :

```
requete_select1 EXCEPT ALL requete_select2
```

L'opérateur ensembliste EXCEPT est l'équivalent de l'opérateur MINUS d'Oracle. Il permet d'obtenir la différence entre deux ensembles : toutes les lignes présentes dans les deux ensembles sont exclues du résultat.

Le dédoublonnage peut être particulièrement coûteux car il implique un tri des données.

Exemples :

L'exemple suivant n'a pas d'autre intérêt que de montrer le résultat de l'opérateur EXCEPT sur deux ensembles simples. La première requête retourne l'ensemble des lignes de la table `region` alors que la seconde requête retourne la ligne qui correspond au prédicat `id = 3`. Cette ligne est ensuite retirée du résultat car elle est présente dans les deux ensembles de gauche et de droite :

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

```
 id |          libelle
----+---------------------------
 11 | Cotes du Rhone
 12 | Provence produit a Cassis.
 10 | Beaujolais
 19 | Savoie
  7 | Languedoc-Roussillon
  4 | Loire
  6 | Provence
 16 | Est
  8 | Bordeaux
 14 | Lyonnais
 15 | Auvergne
  2 | Bourgogne
 17 | Forez
  9 | Vignoble du Sud-Ouest
 18 | Charente
 13 | Champagne
  5 | Jura
  1 | Provence et Corse
```

3.8 FONCTIONS DE BASE

- Transtypage
- Manipulation de chaines
- Manipulation de types numériques
- Manipulation de dates
- Génération de jeu de données

PostgreSQL propose un nombre conséquent de fonctions permettant de manipuler les différents types de données disponibles. Les étudier de façon exhaustive n'est pas l'objet de ce module. Néanmoins, le manuel de PostgreSQL établit une liste complète des fonctions disponibles dans le SGBD[4].

3.8.1 Transtypage

- Conversion d'un type de données vers un autre type de données
- `CAST (expression AS type)`
- `expression::type`

Les opérateurs de transtypages permettent de convertir une donnée d'un type particulier vers un autre type. La conversion échoue si les types de données sont incompatibles.

Exemples

Transtypage incorrect d'une chaîne de caractères vers un entier :

```
SELECT 3 + '3.5'::integer ;
ERROR:  invalid input syntax for type integer: "3.5"
LIGNE 1 : select 3 + '3.5'::integer ;
                     ^
```

PostgreSQL est volontairement peu flexible pour éviter certaines erreurs subtiles liées à une conversion trop hâtive.

[4] https://docs.postgresql.fr/current/functions.html

3.8.2 Opérations simples sur les chaînes

- Concaténation : `chaîne1 || chaîne2`
- Longueur de la chaîne : `char_length(chaîne)`
- Conversion en minuscules : `lower(chaîne)`
- Conversion en majuscules : `upper(chaîne)`

L'opérateur de concaténation permet de concaténer deux chaînes de caractères :

```
SELECT 'Bonjour' || ', Monde!' ;
-----------------
 Bonjour, Monde!
```

Il permet aussi de concaténer une chaîne de caractères avec d'autres type de données :

```
SELECT 'Texte ' || 1::integer ;
---------
 Texte 1
```

La fonction `char_length()` permet de connaître la longueur d'une chaîne de caractères :

```
SELECT char_length('Texte' || 1::integer) ;
-------------
           6
```

Les fonctions `lower` et `upper` permettent de convertir une chaîne respectivement en minuscule et en majuscule :

```
SELECT lower('Bonjour, Monde!') ;
-----------------
 bonjour, monde!

SELECT upper('Bonjour, Monde!') ;
-----------------
 BONJOUR, MONDE!
```

3.8.3 Manipulations de chaînes

- Extrait une chaîne à partir d'une autre : `substring(chaîne [from int] [for int])`
- Emplacement d'une sous-chaîne : `position(sous-chaîne in chaîne)`

La fonction `substring` permet d'extraire une chaîne de caractère à partir d'une chaîne en entrée. Il faut lui indiquer, en plus de la chaîne source, la position de départ, et la longeur de la sous-chaîne. Par exemple :

```
SELECT substring('Bonjour, Monde' from 5 for 4) ;

 substring
-----------
 our,
```

Notez que vous pouvez aussi utiliser un appel de fonction plus standard :

```
SELECT substring('Bonjour, Monde', 5, 4) ;

 substring
-----------
 our,
```

La fonction `position` indique la position d'une chaîne de caractère dans la chaîne indiquée. Par exemple :

```
SELECT position (',' in 'Bonjour, Monde') ;

 position
----------
        8
```

La combinaison des deux est intéressante :

```
SELECT version() ;

                             version
--------------------------------------------------------------------
 PostgreSQL 14.2 (Ubuntu 14.2-1.pgdg20.04+1) on x86_64-pc-linux-…

SELECT substring(version() from 1 for position(' on' in version()));
    substring
-----------------
 PostgreSQL 14.2
```

3.8.4 Manipulation de types numériques

- Opérations arithmétiques
- Manipulation de types numériques
- Génération de données

3.8.5 Opérations arithmétiques

- Addition : +
- Soustraction : −
- Multiplication : *
- Division : /
  - entière si implique des entiers !
- Reste (modulo) : %

Ces opérateurs sont classiques. L'ensemble des opérations arithmétiques disponibles sont documentées dans le manuel[5].

La principale surprise vient de / qui est par défaut une division entière si des entiers sont seuls impliqués :

```
# SELECT 100  / 101     AS div_entiere,
         100 * 1.0 / 101 AS div_non_entiere ;

 div_entiere |    div_non_entiere
-------------+------------------------
           0 | 0.99009900990099009901
```

3.8.6 Fonctions numériques courantes

- Arrondi : `round(numeric)`
- Troncature : `trunc(numeric [, precision])`
- Entier le plus petit : `floor(numeric)`
- Entier le plus grand : `ceil(numeric)`

Ces fonctions sont décrites dans le manuel[6].

[5] https://docs.postgresql.fr/current/functions-math.html
[6] https://docs.postgresql.fr/current/functions-math.html

3.8.7 Génération de données

- Générer une suite d'entiers : `generate_series(borne_debut, borne_fin, intervalle)`
- Générer un nombre aléatoire : `random()`

La fonction `generate_series(n, m)` est spécifique à PostgreSQL et permet de générer une suite d'entiers compris entre une borne de départ et une borne de fin, en suivant un certain intervalle :

```
SELECT generate_series(1, 4) ;
```

```
 generate_series
-----------------
               1
               2
               3
               4
(4 rows)
```

La déclinaison `generate_series(n, m, interval)` permet de spécifier un incrément pour chaque itération :

```
SELECT generate_series(1, 10, 4) ;
```

```
 generate_series
-----------------
               1
               5
               9
(3 rows)
```

Quant à la fonction `random()`, elle génère un nombre aléatoire, de type `numeric`, compris entre 0 et 1.

```
SELECT random() ;
```

```
      random
-------------------
 0.381810061167926
```

Pour générer un entier compris entre 0 et 100, il suffit de réaliser la requête suivante :

```
SELECT round(100*random())::integer ;
```

```
 round
-------
    74
```

Il est possible de contrôler la graine du générateur de nombres aléatoires en positionnant le paramètre de session SEED :

```
SET SEED = 0.123 ;
```

ou à l'aide de la fonction `setseed()` :

```
SELECT setseed(0.123) ;
```

La graine est un flottant compris entre -1 et 1.

Ces fonctions sont décrites dans le manuel de PostgreSQL[7].

3.8.8 Manipulation de dates

- Obtenir la date et l'heure courante
- Manipuler des dates
- Opérations arithmétiques
- Formatage de données

3.8.9 Date et heure courante

- Retourne la date courante : `current_date`
- Retourne l'heure courante : `current_time`
- Retourne la date et l'heure courante : `current_timestamp` / `now()`

Les fonctions `current_date` et `current_time` permettent d'obtenir respectivement la date courante et l'heure courante. La première fonction retourne le résultat sous la forme d'un type `date` et la seconde sous la forme d'un type `time with time zone`.

Préférer `current_timestamp` et son synonyme `now()` pour obtenir la date et l'heure courante, le résultat étant de type `timestamp with time zone`.

Exceptionnellement, les fonctions `current_date`, `current_time` et `current_timestamp` n'ont pas besoin d'être invoquée avec les parenthèses ouvrantes et fermantes typiques de l'appel d'une fonction. En revanche, l'appel de la fonction `now()` requiert ces parenthèses.

```
SELECT current_date ;
```

```
 current_date
--------------
 2017-10-04
```

[7] https://docs.postgresql.fr/current/functions-math.html

```
SELECT current_time ;

    current_time
--------------------
 16:32:47.386689+02

SELECT current_timestamp ;

       current_timestamp
-------------------------------
 2017-10-04 16:32:50.314897+02

SELECT now();
              now
-------------------------------
 2017-10-04 16:32:53.684813+02
```

Il est possible d'utiliser ces variables comme valeur par défaut d'une colonne :

```
CREATE TABLE test (
   id int       GENERATED ALWAYS AS IDENTITY,
   dateheure    timestamp with time zone DEFAULT current_timestamp,
   valeur varchar
) ;

INSERT INTO test (valeur) VALUES ('Bonjour, monde!');

SELECT * FROM test ;

 id |           dateheure           |     valeur
----+-------------------------------+-----------------
  1 | 2020-01-30 18:34:34.067738+01 | Bonjour, monde!
```

3.8.10 Manipulation des données

- Âge
  - Par rapport à la date courante : `age(timestamp)`
  - Par rapport à une date de référence : `age(timestamp, timestamp)`

La fonction `age(timestamp)` permet de déterminer l'âge de la date donnée en paramètre par rapport à la date courante. L'âge sera donné sous la forme d'un type `interval`.

La forme `age(timestamp, timestamp)` permet d'obtenir l'âge d'une date par rapport à une autre date, par exemple pour connaître l'âge de Gaston Lagaffe au 5 janvier 1997 :

```
SELECT age(date '1997-01-05', date '1957-02-28') ;

          age
------------------------
 39 years 10 mons 5 days
```

3.8.11 Tronquer et extraire

- Troncature d'une date : `date_trunc(text, timestamp)`
- Exemple : `date_trunc('month' from date_naissance)`
- Extrait une composante de la date : `extract(text, timestamp)`
- Exemple : `extract('year' from date_naissance)`

La fonction `date_trunc(text, timestamp)` permet de tronquer la date à une précision donnée. La précision est exprimée en anglais, et autorise les valeurs suivantes :

- microseconds
- milliseconds
- second
- minute
- hour
- day
- week
- month
- quarter
- year
- decade
- century
- millennium

La fonction `date_trunc()` peut agir sur une donnée de type `timestamp`, `date` ou `interval`. Par exemple, pour *arrondir* l'âge de Gaston Lagaffe de manière à ne représenter que le nombre d'années :

```
SELECT date_trunc('year',
    age(date '1997-01-05', date '1957-02-28')) AS age_lagaffe ;

 age_lagaffe
-------------
 39 years
```

La fonction `extract(text from timestamp)` permet d'extraire uniquement une composante donnée d'une date, par exemple l'année. Elle retourne un type de données flottant `double precision`.

```
SELECT extract('year' from
    age(date '1997-01-05', date '1957-02-28')) AS age_lagaffe ;

 age_lagaffe
-------------
          39
```

3.8.12 Arithmétique sur les dates

- Opérations arithmétiques sur `timestamp`, `time` ou `date`
    - `date/time - date/time = interval`
    - `date/time + time = date/time`
    - `date/time + interval = date/time`
- Opérations arithmétiques sur `interval`
    - `interval * numeric  = interval`
    - `interval / numeric  = interval`
    - `interval + interval = interval`

La soustraction de deux types de données représentant des dates permet d'obtenir un intervalle qui représente le délai écoulé entre ces deux dates :

```
SELECT timestamp '2012-01-01 10:23:10' - date '0001-01-01' AS soustraction ;

     soustraction
----------------------
 734502 days 10:23:10
```

L'addition entre deux types de données est plus restreinte. En effet, l'expression de gauche est obligatoirement de type `timestamp` ou `date` et l'expression de droite doit être obligatoirement de type `time`. Le résultat de l'addition permet d'obtenir une donnée de type `timestamp`, avec ou sans information sur le fuseau horaire selon que cette information soit présente ou non sur l'expression de gauche.

```
SELECT timestamp '2001-01-01 10:34:12' + time '23:56:13' AS addition ;

      addition
---------------------
 2001-01-02 10:30:25

SELECT date '2001-01-01' + time '23:56:13' AS addition ;

      addition
---------------------
 2001-01-01 23:56:13
```

L'addition d'une donnée datée avec une donnée de type `interval` permet d'obtenir un résultat du même type que l'expression de gauche :

```
SELECT timestamp with time zone '2001-01-01 10:34:12' +
    interval '1 day 1 hour' AS addition ;
```

```
        addition
------------------------
 2001-01-02 11:34:12+01
```

```
SELECT date '2001-01-01'  + interval '1 day 1 hour' AS addition ;
```

```
      addition
---------------------
 2001-01-02 01:00:00
```

```
SELECT time '10:34:24' + interval '1 day 1 hour' AS addition ;
```

```
 addition
----------
 11:34:24
```

Une donnée de type `interval` peut subir des opérations arithmétiques. Le résultat sera de type `interval` :

```
SELECT interval '1 day 1 hour' * 2 AS multiplication ;
```

```
 multiplication
-----------------
 2 days 02:00:00
```

```
SELECT interval '1 day 1 hour' / 2 AS division ;
```

```
 division
----------
 12:30:00
```

```
SELECT interval '1 day 1 hour' + interval '2 hour' AS addition ;
```

```
    addition
---------------
 1 day 03:00:00
```

```
SELECT interval '1 day 1 hour' - interval '2 hour' AS soustraction ;
  soustraction
-----------------
 1 day -01:00:00
```

3.8.13 Date vers chaîne

- Conversion d'une date en chaîne de caractères : `to_char(timestamp, text)`
- Exemple : `to_char(current_timestamp, 'DD/MM/YYYY HH24:MI:SS')`

La fonction `to_char()` permet de restituer une date selon un format donné :

```
SELECT current_timestamp ;

       current_timestamp
-------------------------------
 2017-10-04 16:35:39.321341+02

SELECT to_char(current_timestamp, 'DD/MM/YYYY HH24:MI:SS');

       to_char
---------------------
 04/10/2017 16:35:43
```

3.8.14 Chaîne vers date

- Conversion d'une chaîne de caractères en date : `to_date(text, text)` `to_date('05/12/2000', 'DD/MM/YYYY')`
- Conversion d'une chaîne de caractères en timestamp : `to_timestamp(text, text)` `to_timestamp('05/12/2000 12:00:00', 'DD/MM/YYYY HH24:MI:SS')`
- Paramètre `datestyle`

Quant à la fonction `to_date()`, elle permet de convertir une chaîne de caractères dans une donnée de type `date`. La fonction `to_timestamp()` permet de réaliser la même mais en donnée de type `timestamp`.

```
SELECT to_timestamp('04/12/2000 12:00:00', 'DD/MM/YYYY HH24:MI:SS') ;

to_timestamp
------------------------
 2000-12-04 12:00:00+01
```

Ces fonctions sont détaillées dans la section concernant les fonctions de formatage de données du manuel[8].

Le paramètre `DateStyle` contrôle le format de saisie et de restitution des dates et heures. 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 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 ;
```

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

```
              now
-------------------------------
 2017-10-04 16:36:38.189973+02

SET datestyle = 'SQL, DMY' ;

SELECT current_timestamp ;

              now
---------------------------------
 04/10/2017 16:37:04.307034 CEST
```

3.8.15 Génération de données

- Générer une suite de timestamp :
  - `generate_series (timestamp_debut, timestamp_fin, intervalle)`

La fonction `generate_series(date_debut, date_fin, interval)` permet de générer des séries de dates :

```
SELECT generate_series(date '2012-01-01',date '2012-12-31',interval '1 month') ;

    generate_series
------------------------
 2012-01-01 00:00:00+01
 2012-02-01 00:00:00+01
 2012-03-01 00:00:00+01
 2012-04-01 00:00:00+02
 2012-05-01 00:00:00+02
 2012-06-01 00:00:00+02
 2012-07-01 00:00:00+02
 2012-08-01 00:00:00+02
 2012-09-01 00:00:00+02
 2012-10-01 00:00:00+02
 2012-11-01 00:00:00+01
 2012-12-01 00:00:00+01
(12 rows)
```

3.9 VUES

- Tables virtuelles
  - définies par une requête SELECT
  - définition stockée dans le catalogue de la base de données
- Objectifs
  - masquer la complexité d'une requête
  - masquer certaines données à l'utilisateur
- Vues ≠ vues matérialisées

Les vues sont des « tables virtuelles » qui permettent d'obtenir le résultat d'une requête SELECT. Sa définition est stockée dans le catalogue système de la base de données. Le SELECT est exécuté quand la vue est appelée.

De cette façon, il est possible de créer une vue à destination de certains utilisateurs pour combler différents besoins :

- permettre d'interroger facilement une vue qui exécute une requête complexe ou lourde à écrire ;
- masquer certaines lignes ou certaines colonnes aux utilisateurs, pour amener un niveau de sécurité complémentaire ;
- rendre les données plus intelligibles, en nommant mieux les colonnes d'une vue et/ou en simplifiant la structure de données ;
- assurer la compatibilité avec d'anciennes requêtes après des modifications…

En plus de cela, les vues permettent d'obtenir facilement des valeurs dérivées d'autres colonnes. Ces valeurs dérivées pourront alors être utilisées simplement en appelant la vue plutôt qu'en réécrivant systématiquement le calcul de dérivation à chaque requête qui le nécessite.

Les vues classiques équivalent à exécuter un SELECT. Il existe des « vues matérialisées », qui ne seront pas développées ici, qui de vraies tables créées à partir d'une requête (et rafraîchies uniquement sur demande explicitement).

3.9.1 Création d'une vue

- Une vue porte un nom au même titre qu'une table
  - elle sera nommée avec les mêmes règles
- Ordre de création d'une vue : `CREATE VIEW vue (colonne ...) AS SELECT ...`

Bien qu'une vue n'ait pas de représentation physique directe, elle est accédée au même titre qu'une table avec `SELECT` et dans certains cas avec `INSERT`, `UPDATE` et `DELETE`. La vue logique ne distingue pas les accès à une vue des accès à une table. De cette façon, une vue doit utiliser les mêmes conventions de nommage qu'une table.

Une vue est créée avec l'ordre SQL `CREATE VIEW` :

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW nom
    [ ( nom_colonne [, ...] ) ]
    [ WITH ( nom_option_vue [= valeur_option_vue] [, ... ] ) ]
    AS requete
```

Le mot clé `CREATE VIEW` permet de créer une vue. Si elle existe déjà, il est possible d'utiliser `CREATE OR REPLACE VIEW` qui aura pour effet de créer la vue si elle n'existe pas ou de remplacer la définition de la vue si elle existe déjà. Attention, dans ce dernier cas, les colonnes et les types de données retournés par la vue ne doivent pas changer.

La clause `nom` permet de nommer la vue. La clause `nom_colonne, ...` permet lister explicitement les colonnes retournées par une vue, cette clause est optionnelle mais recommandée pour mieux documenter la vue.

La clause `requete` correspond simplement à la requête `SELECT` exécutée lorsqu'on accède à la vue.

Exemples

```
CREATE TABLE phone_data (person text, phone text, private boolean) ;

CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data ;

GRANT SELECT ON phone_number TO secretary ;
```

3.9.2 Lecture d'une vue

- Une vue est lue comme une table
  - `SELECT * FROM vue;`

Une vue est lue de la même façon qu'une table. On utilisera donc l'ordre SELECT pour le faire. L'optimiseur de PostgreSQL remplacera l'appel à la vue par la définition de la vue pendant la phase de réécriture de la requête. Le plan d'exécution prendra alors compte des particularités de la vue pour optimiser les accès aux données.

Exemples

```
CREATE TABLE phone_data (person text, phone text, private boolean);

CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data ;

INSERT INTO phone_data (person, phone, private)
  VALUES ('Titi', '0123456789', true) ;

INSERT INTO phone_data (person, phone, private)
  VALUES ('Rominet', '0123456788', false) ;

SELECT person, phone FROM phone_number ;

 person  |   phone
---------+------------
 Titi    |
 Rominet | 0123456788
```

3.9.3 Sécurisation d'une vue

- Sécuriser une vue
  - droits avec GRANT et REVOKE
- Utiliser les vues comme moyen de filtrer les lignes est dangereux
  - option `security_barrier`

Il est possible d'accorder (ou de révoquer) à un utilisateur les mêmes droits sur une vue que sur une table :

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

Le droit SELECT autorise un utilisateur à lire une table. Les droits INSERT, UPDATE et DELETE permettent de contrôler les accès en mise à jour à une vue.

Les droits TRUNCATE et REFERENCES n'ont pas d'utilité avec une vue. Ils ne sont tout simplement pas supportés car TRUNCATE n'agit que sur une table et une clé étrangère ne peut être liée d'une table qu'à une autre table.

Les vues sont parfois utilisées pour filtrer les lignes pouvant être lues par l'utilisateur. Cette protection peut être contournée si l'utilisateur a la possibilité de créer une fonction. L'option `security_barrier` permet d'éviter ce problème.

Exemples

```
postgres=# CREATE TABLE elements (id serial, contenu text, prive boolean) ;
CREATE TABLE
postgres=# INSERT INTO elements (contenu, prive)
VALUES ('a', false), ('b', false), ('c super prive', true),
       ('d', false), ('e prive aussi', true) ;
INSERT 0 5

postgres=# SELECT * FROM elements ;

 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
```

La table `elements` contient cinq lignes, dont trois considérés comme privés. Nous allons donc créer une vue ne permettant de voir que les lignes publiques.

```
postgres=# CREATE OR REPLACE VIEW elements_public AS
SELECT * FROM elements
WHERE CASE WHEN current_user='postgres' THEN TRUE ELSE NOT prive END ;

CREATE VIEW

postgres=# SELECT * FROM elements_public ;

 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
```

```
postgres=# CREATE USER u1;
CREATE ROLE

postgres=# GRANT SELECT ON elements_public TO u1;
GRANT

postgres=# \c - u1
You are now connected to database "postgres" as user "u1".

postgres=> SELECT * FROM elements ;
ERROR:  permission denied for relation elements

postgres=> SELECT * FROM elements_public ;

 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

L'utilisateur `u1` n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Mais le contenu peut être trahi par une simple fonction qui trahit le contenu des lignes que l'exécuteur rencontre avant de les filtrer :

```
postgres=> CREATE OR REPLACE FUNCTION abracadabra(integer, text, boolean)
           RETURNS bool AS $$
           BEGIN
             -- afficher chaque ligne rencontrée
             RAISE NOTICE '% - % - %', $ 1, $ 2, $ 3 ;
             RETURN true ;
           END$$
           LANGUAGE plpgsql
           -- désigner un coût bas pour exécuter cette fonction
           -- avant le filtre dans la vue
           COST 0.000000000000000000001 ;
CREATE FUNCTION

postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive) ;

NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  3 - c super prive - t
NOTICE:  4 - d - f
NOTICE:  5 - e prive aussi - t
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

Que s'est-il passé ? pour comprendre, il suffit de regarder l'`EXPLAIN` de cette requête :

```
postgres=> EXPLAIN SELECT * FROM elements_public
WHERE abracadabra(id, contenu, prive) ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on elements  (cost=0.00..28.15 rows=202 width=37)
   Filter: (abracadabra(id, contenu, prive) AND
           CASE WHEN ("current_user"() = 'u1'::name)
                THEN (NOT prive) ELSE true END)
```

La requête contient deux filtres : celui dans la vue, celui dans la fonction `abracadra`. On a déclaré un coût si faible pour cette dernière que PostgreSQL, pour optimiser,l'exécute avant le filtre de la vue. Du coup, la fonction voit toutes les lignes de la table et peut trahir leur contenu.

Seul moyen d'échapper à cette optimisation du planificateur, utiliser l'option `security_barrier` :

```
postgres=> \c - postgres
You are now connected to database "postgres" as user "postgres".

postgres=# CREATE OR REPLACE VIEW elements_public
           WITH (security_barrier)
           AS
           SELECT * FROM elements
           WHERE CASE WHEN current_user='postgres' THEN true ELSE NOT prive END ;

CREATE VIEW

postgres=# \c - u1
You are now connected to database "postgres" as user "u1".

postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);

NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  4 - d - f
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f

postgres=> EXPLAIN SELECT * FROM elements_public WHERE
abracadabra(id, contenu, prive) ;

                                QUERY PLAN
---------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..34.20 rows=202 width=37)
   Filter: abracadabra(elements_public.id, elements_public.contenu,
           elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.15 rows=605 width=37)
         Filter: CASE WHEN ("current_user"() = 'u1'::name)
                 THEN (NOT prive) ELSE true END
```

Il peut y avoir un impact en performance : le filtre de la vue s'applique d'abord, donc force peut forcer l'optimiseur à s'écarter du chemin optimal.

Voir aussi cet article de blog[9].

[9] https://rhaas.blogspot.com/2012/03/security-barrier-views.html

3.9.4 Mise à jour des vues

- *Updatable view*
- `WITH CHECK OPTION`
- Mises à jour non triviales : trigger `INSTEAD OF`

Depuis PostgreSQL 9.3, le moteur permet de mettre à jour des vues simples, ou plus exactement de mettre à jour les tables sous-jacentes au travers de la vue. Les critères déterminant si une vue peut être mise à jour ou non sont assez simples à résumer : la vue doit reprendre la définition de la table avec éventuellement une clause `WHERE` pour restreindre les résultats.

La clause `WITH CHECK OPTION` (à partie de PostgreSQL 9.4) empêche l'utilisateur d'insérer des données qui ne satisfont pas les critères de filtrage de la vue. Sans elle, il est possible d'insérer un numéro de téléphone privé alors que la vue ne permet pas d'afficher les numéros privés. Cette option doit être demandée explicitement.

Pour gérer les cas plus complexes, PostgreSQL permet de créer des triggers `INSTEAD OF` sur des vues. Une alternative est d'utiliser le système de règles (`RULES`) mais cette pratique est peu recommandée en raison de la difficulté de débogage et de maintenance.

Un trigger `INSTEAD OF` permet de déclencher une fonction utilisateur lorsqu'une opération de mise à jour est déclenchée sur une vue. Le code de la fonction sera exécuté en lieu et place de la mise à jour.

Exemples

```
CREATE TABLE phone_data (person text, phone text, private boolean);

CREATE VIEW maj_phone_number (person, phone, private) AS
    SELECT person, phone, private
      FROM phone_data
     WHERE private = false ;

-- On peut insérer des données car les colonnes de la vue correspondent aux
-- colonnes de la table
INSERT INTO maj_phone_number VALUES ('Titi', '0123456789', false) ;

-- On parvient même à insérer des données qui ne pourront pas être affichées
-- par la vue. Ça peut être gênant.
INSERT INTO maj_phone_number VALUES ('Loulou', '0123456789', true) ;

SELECT * FROM maj_phone_number ;

 person |   phone    | private
--------+------------+---------
 Titi   | 0123456789 | f

-- L'option WITH CHECK OPTION rajoute une sécurité
CREATE OR REPLACE VIEW maj_phone_number (person, phone, private) AS
```

```
    SELECT person, phone, private
    FROM phone_data
    WHERE private = false
    WITH CHECK OPTION ;

INSERT INTO maj_phone_number VALUES ('Lili', '9993456789', true);
ERROR:  new row violates check option for view "maj_phone_number"
DETAIL : Failing row contains (Lili, 9993456789, t).


-- Cas d'une vue avec un champ calculé
CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data ;


-- On ne peut pas insérer de données car les colonnes de la vue ne
-- correspondent pas à celles de la table
INSERT INTO phone_number VALUES ('Fifi', '0123456789');
ERROR:  cannot insert into column "phone" of view "phone_number"
DETAIL:  View columns that are not columns of their base relation are not updatable.


CREATE OR REPLACE FUNCTION phone_number_insert_row()
  RETURNS TRIGGER
  LANGUAGE plpgsql
AS $function$
BEGIN
  INSERT INTO phone_data (person, phone, private)
    VALUES (NEW.person, NEW.phone, false);
  RETURN NEW ;
END ;
$function$;


CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON phone_number
    FOR EACH ROW
    EXECUTE PROCEDURE phone_number_insert_row();


-- Avec le trigger, c'est maintenant possible.
INSERT INTO phone_number VALUES ('Rominet', '0123456788');

SELECT * FROM phone_number ;


 person  |   phone
---------+------------
 Titi    | 0123456789
 Loulou  |
 Rominet | 0123456788
```

3.9.5 Mauvaises utilisations des vues

- Prolifération des vues
  - créer une vue doit se justifier
  - ne pas créer une vue par table
- Vues trop complexes utilisées comme interface
- Vues empilées

La création d'une vue doit être pensée préalablement et doit se justifier du point de vue de l'application ou d'une règle métier. Toute vue créée doit être documentée, au moins en plaçant un commentaire sur la vue pour la documenter.

Bien qu'une vue n'ait pas de représentation physique, elle occupe malgré tout un peu d'espace disque. En effet, le catalogue système comporte une entrée pour chaque vue créée, autant d'entrées qu'il y a de colonnes à la vue, etc. Trop de vues entraîne donc malgré tout l'augmentation de la taille du catalogue système, donc une empreinte mémoire plus importante car ce catalogue reste en général systématiquement présent en cache.

Un problème fréquent est celui de vues complexes calculant beaucoup de choses pour le confort de l'usager… au prix des performances quand l'utilisateur n'a pas besoin de ces informations. L'optimiseur ne peut pas forcément tout élaguer.

Pour cette raison, et pour des raisons de facilité de maintenance, il faut aussi éviter d'empiler les vues.

3.10 REQUÊTES PRÉPARÉES

- Exécution en deux temps
  - préparation du plan d'exécution de la requête
  - exécution de la requête en utilisant le plan préparé
- Objectif :
  - éviter simplement les injections SQL
  - améliorer les performances

Les *requêtes préparées*, aussi appelées *requêtes paramétrées*, permettent de séparer la phase de préparation du plan d'exécution de la phase d'exécution. Le plan d'exécution qui est alors généré est générique car les paramètres de la requêtes sont inconnus à ce moment là.

L'exécution est ensuite commandée par l'application, en passant l'ensemble des valeurs des paramètres de la requête. De plus, ces paramètres sont passés de façon à éviter les injections SQL.

L'exécution peut être ensuite commandée plusieurs fois, sans avoir à préparer le plan d'exécution. Cela permet un gain important en terme de performances car l'étape d'analyse syntaxique et de recherche du plan d'exécution optimal n'est plus à faire.

L'utilisation de requêtes préparées peut toutefois être contre-performant si les sessions ne sont pas maintenues et les requêtes exécutées qu'une seule fois. En effet, l'étape de préparation oblige à un premier aller-retour entre l'application et la base de données et l'exécution oblige à un second aller-retour, ajoutant ainsi une surcharge qui peut devenir significative.

3.10.1 Utilisation

- PREPARE, préparation du plan d'exécution d'une requête
- EXECUTE, passage des paramètres de la requête et exécution réelle
- L'implémentation dépend beaucoup du langage de programmation utilisé
  - le connecteur JDBC supporte les requêtes préparées
  - le connecteur PHP/PDO également

L'ordre PREPARE permet de préparer le plan d'exécution d'une requête. Le plan d'exécution pren-

dra en compte le contexte courant de l'utilisateur au moment où la requête est préparée, et notamment le `search_path`. Tout changement ultérieur de ces variables ne sera pas pris en compte à l'exécution.

L'ordre EXECUTE permet de passer les paramètres de la requête et de l'exécuter.

La plupart des langages de programmation mettent à disposition des méthodes qui permettent d'employer les mécanismes de préparation de plans d'exécution directement. Les paramètres des requêtes seront alors transmis un à un à l'aide d'une méthode particulière.

Voici comment on prépare une requête :

```
PREPARE req1 (text) AS
    SELECT person, phone FROM phone_number WHERE person = $1;
```

Le test suivant montre le gain en performance qu'on peut attendre d'une requête préparée :

- préparation de la table :

```
CREATE TABLE t1 (c1 integer primary key, c2 text);
INSERT INTO t1 select i, md5(random()::text)
FROM generate_series(1, 1000000) AS i ;
```

- préparation de deux scripts SQL, une pour les requêtes standards, l'autre pour les requêtes préparées :

```
$ for i in $(seq 1 100000); do
    echo "SELECT * FROM t1 WHERE c1=$i;";
done > requetes_std.sql
echo "PREPARE req AS SELECT * FROM t1 WHERE c1=\$1;" > requetes_prep.sql
for i in $(seq 1 100000); do echo "EXECUTE req($i);"; done >> requetes_prep.sql
```

- exécution du test (deux fois pour s'assurer que les temps d'exécution sont réalistes) :

```
$ time psql -f requetes_std.sql postgres >/dev/null

real  0m12.742s
user  0m2.633s
sys 0m0.771s
$ time psql -f requetes_std.sql postgres >/dev/null

real  0m12.781s
user  0m2.573s
sys 0m0.852s
$ time psql -f requetes_prep.sql postgres >/dev/null

real  0m10.186s
user  0m2.500s
sys 0m0.814s
$ time psql -f requetes_prep.sql postgres >/dev/null

real  0m10.131s
user  0m2.521s
sys 0m0.808s
```

Le gain est de 16 % dans cet exemple. Il peut être bien plus important. En lisant 500 000 lignes (et non pas 100 000), on arrive à 25 % de gain.

3.11 CONCLUSION

- Possibilité d'écrire des requêtes complexes
- C'est là où PostgreSQL est le plus performant

Le standard SQL va bien plus loin que ce que les requêtes simplistes laissent penser. Utiliser des requêtes complexes permet de décharger l'application d'un travail conséquent et le développeur de coder quelque chose qui existe déjà. Cela aide aussi la base de données car il est plus simple d'optimiser une requête complexe qu'un grand nombre de requêtes simplistes.

3.11.1 Questions

N'hésitez pas, c'est le moment !

3.12 TRAVAUX PRATIQUES 1 (ÉNONCÉS)

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 :

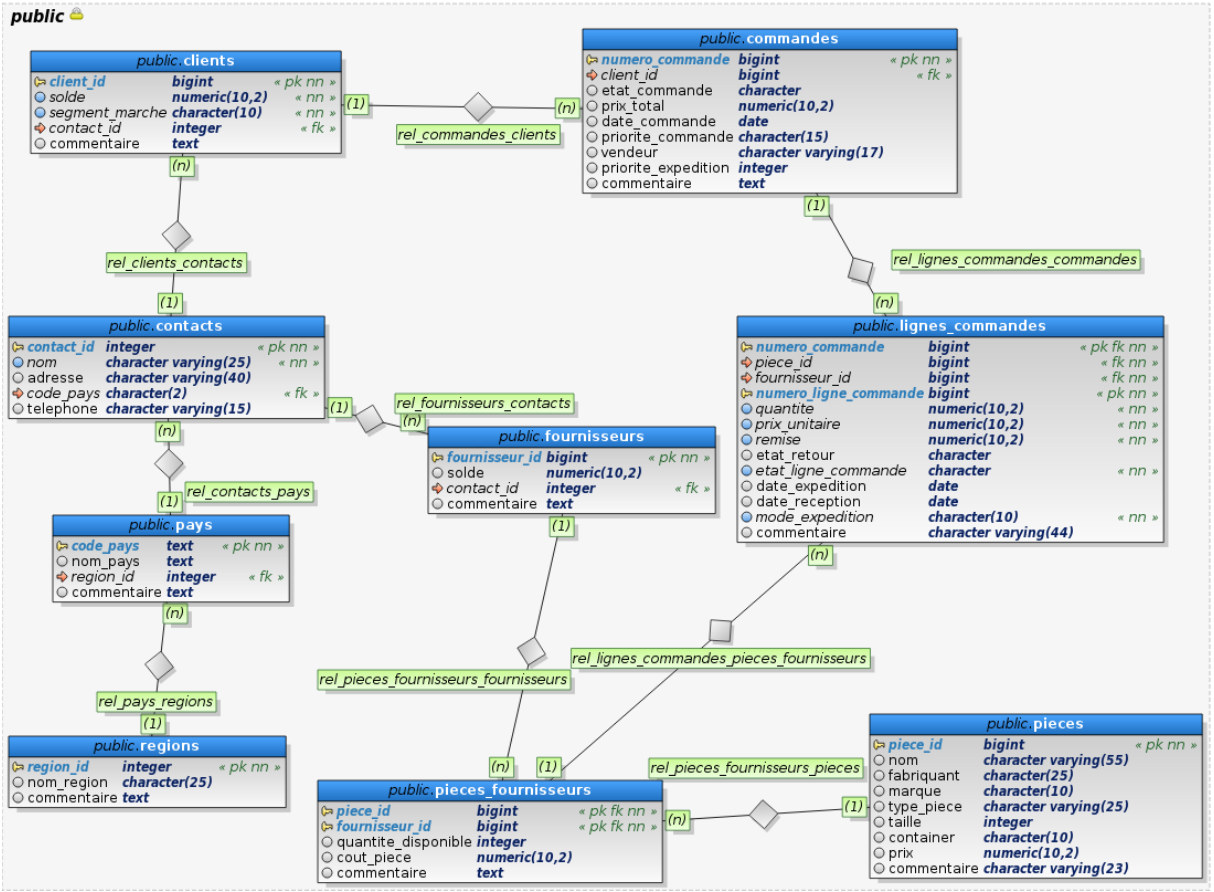


Figure 3/ .4: Schéma base tpc

1. Affichez, par pays, le nombre de fournisseurs.

Sortie attendue :

```
           nom_pays             | nombre
--------------------------------+--------
 ARABIE SAOUDITE                |    425
 ARGENTINE                      |    416
 (...)
```

2. Affichez, par continent (`regions`), le nombre de fournisseurs.

Sortie attendue :

```
        nom_region          | nombre
----------------------------+--------
 Afrique                    |   1906
 Moyen-Orient               |   2113
 Europe                     |   2094
 Asie                       |   2002
 Amérique                   |   1885
```

3. Affichez le nombre de commandes trié selon le nombre de lignes de commandes au sein de chaque commande.

Sortie attendue :

```
 num | count
-----+-------
   1 | 13733
   2 | 27816
   3 | 27750
   4 | 27967
   5 | 27687
   6 | 27876
   7 | 13895
```

4. Pour les 30 premières commandes (selon la date de commande), affichez le prix total de la commande, en appliquant la remise accordée sur chaque article commandé. La sortie sera triée de la commande la plus chère à la commande la moins chère.

Sortie attendue :

```
 numero_commande | prix_total
-----------------+------------
               3 |  259600.00
              40 |  258959.00
               6 |  249072.00
              69 |  211330.00
              70 |  202101.00
               4 |  196132.00
 (...)
```

5. Affichez, par année, le total des ventes. La date de commande fait foi. La sortie sera triée par année.

Sortie attendue :

```
 annee |  total_vente
-------+---------------
  2005 | 3627568010.00
  2006 | 3630975501.00
  2007 | 3627112891.00
(...)
```

6. Pour toutes les commandes, calculez le temps moyen de livraison, depuis la date d'expédition. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

Sortie attendue :

```
temps_moyen_livraison
----------------------
 8 jour(s)
```

7. Pour les 30 commandes les plus récentes (selon la date de commande), calculez le temps moyen de livraison de chaque commande, depuis la date de commande. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

Sortie attendue :

```
temps_moyen_livraison
----------------------
 38 jour(s)
```

8. Déterminez le taux de retour des marchandises (l'état à R indiquant qu'une marchandise est retournée).

Sortie attendue :

```
taux_retour
-------------
       24.29
```

9. Déterminez le mode d'expédition qui est le plus rapide, en moyenne.

Sortie attendue :

```
 mode_expedition |       delai
-----------------+--------------------
 AIR             | 7.4711070230494535
```

10. Un bug applicatif est soupçonné, déterminez s'il existe des commandes dont la date de commande est postérieure à la date d'expédition des articles.

Sortie attendue :

```
count
-------
     2
```

11. Écrivez une requête qui corrige les données erronés en positionnant la date de commande à la date d'expédition la plus ancienne des marchandises. Vérifiez qu'elle soit correcte. Cette requête permet de corriger des calculs de statistiques sur les délais de livraison.

12. Écrivez une requête qui calcule le délai total maximal de livraison de la totalité d'une commande donnée, depuis la date de la commande.

Sortie attendue pour la commande n°1 :

```
delai_max
-----------
       102
```

13. Écrivez une requête pour déterminer les 10 commandes dont le délai de livraison, entre la date de commande et la date de réception, est le plus important, pour l'année 2011 uniquement.

Sortie attendue :

```
 numero_commande | delai
-----------------+-------
          413510 |   146
          123587 |   143
          224453 |   143
(...)
```

14. Un autre bug applicatif est détecté. Certaines commandes n'ont pas de lignes de commandes. Écrivez une requête pour les retrouver.

```
-[ RECORD 1 ]------------------------
numero_commande     | 91495
client_id           | 93528
etat_commande       | P
prix_total          |
date_commande       | 2007-07-07
priorite_commande   | 5-NOT SPECIFIED
vendeur             | Vendeur 000006761
priorite_expedition | 0
commentaire         | xxxxxxxxxxxx
```

15. Écrivez une requête pour supprimer ces commandes. Vérifiez le travail avant de valider.

16. Écrivez une requête pour déterminer les 20 pièces qui ont eu le plus gros volume de commande.

Sortie attendue :

```
                     nom                      |  sum
----------------------------------------------+--------
 lemon black goldenrod seashell plum          | 461.00
 brown lavender dim white indian              | 408.00
 burlywood white chiffon blanched lemon       | 398.00
(...)
```

17. Affichez les fournisseurs des 20 pièces qui ont été le plus commandées sur l'année 2011.

Sortie attendue :

```
     nom      | piece_id
--------------+----------
 Supplier4395 |   191875
 Supplier4397 |   191875
 Supplier6916 |   191875
 Supplier9434 |   191875
 Supplier4164 |    11662
 Supplier6665 |    11662
(...)
```

18. Affichez le pays qui a connu, en nombre, le plus de commandes sur l'année 2011.

Sortie attendue :

```
    nom_pays     | count
-----------------+-------
 ARABIE SAOUDITE |  1074
```

19. Affichez pour les commandes passées en 2011, la liste des continents (`régions`) et la marge brute d'exploitation réalisée par continents, triés dans l'ordre décroissant.

Sortie attendue :

```
        nom_region          |   benefice
----------------------------+---------------
 Moyen-Orient               | 2008595508.00
(...)
```

20. Affichez le nom, le numéro de téléphone et le pays des fournisseurs qui ont un commentaire contenant le mot clé `Complaints` :

Sortie attendue :

```
 nom_fournisseur |    telephone    |           nom_pays
-----------------+-----------------+-------------------------------
 Supplier3873    | 10-741-199-8614 | IRAN, RÉPUBLIQUE ISLAMIQUE D'
(...)
```

21. Déterminez le top 10 des fournisseurs ayant eu le plus long délai de livraison, entre la date de commande et la date de réception, pour l'année 2011 uniquement.

Sortie attendue :

```
 fournisseur_id | nom_fournisseur | delai
----------------+-----------------+-------
           9414 | Supplier9414    |   146
(...)
```

3.13 TRAVAUX PRATIQUES 2 (ÉNONCÉS)

Ce TP s'exécute sur la base **tpc** :

1. Ajouter une adresse mail à chaque contact avec la concaténation du nom avec le texte « @dalibo.com ».

2. Concaténer nom et adresse mail des contacts français sous la forme « nom <mail> ».

3. Même demande mais avec le nom en majuscule et l'adresse mail en minuscule.

4. Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`. Mettre à jour la colonne `prix_total` des commandes avec les informations des lignes de la table `lignes_commandes`.

5. Récupérer le montant total des commandes par mois pour l'année 2010. Les montants seront arrondis à deux décimales.

6. Supprimer les commandes de mai 2010.

7. Ré-exécuter la requête trouvée au point 5.

8. Qu'observez-vous ?

9. Corriger le problème rencontré.

10. Créer une vue calculant le prix total de chaque commande.

11. Réécrire la requête du point 5 pour utiliser la vue créée au point 10.

3.14 TRAVAUX PRATIQUES 1 (SOLUTIONS)

1. Affichez, par pays, le nombre de fournisseurs.

```
SELECT p.nom_pays, count(*)
FROM fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
GROUP BY p.nom_pays ;
```

2. Affichez, par continent (`regions`), le nombre de fournisseurs.

```
SELECT r.nom_region, count(*)
FROM fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
   JOIN regions r ON p.region_id = r.region_id
GROUP BY r.nom_region ;
```

3. Affichez le nombre de commandes trié selon le nombre de lignes de commandes au sein de chaque commande.

```
SELECT
   nombre_lignes_commandes,
   count(*) AS nombre_total_commandes
FROM (
   /* cette sous-requête permet de compter le nombre de lignes de commande de
      chaque commande, et remonte cette information à la requête principale */
   SELECT count(numero_ligne_commande) AS nombre_lignes_commandes
   FROM lignes_commandes
   GROUP BY numero_commande
   ) comm_agg
/* la requête principale aggrège et trie les données sur ce nombre de lignes
   de commandes pour compter le nombre de commandes distinctes ayant le même
   nombre de lignes de commandes */
GROUP BY nombre_lignes_commandes
ORDER BY nombre_lignes_commandes DESC ;
```

4. Pour les 30 premières commandes (selon la date de commande), affichez le prix total de la commande, en appliquant la remise accordée sur chaque article commandé. La sortie sera triée de la commande la plus chère à la commande la moins chère.

```
SELECT c.numero_commande, sum(quantite * prix_unitaire - remise) prix_total
FROM (
   SELECT numero_commande, date_commande
   FROM commandes
   ORDER BY date_commande
   LIMIT 30
   ) c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
GROUP BY c.numero_commande
ORDER BY sum(quantite * prix_unitaire - remise) DESC ;
```

5. Affichez, par année, le total des ventes. La date de commande fait foi. La sortie sera triée par année.

```
SELECT
   extract ('year' FROM date_commande),
   sum(quantite * prix - remise) AS prix_total
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
   JOIN pieces p ON lc.piece_id = p.piece_id
GROUP BY extract ('year' FROM date_commande)
ORDER BY extract ('year' FROM date_commande) ;
```

6. Pour toutes les commandes, calculez le temps moyen de livraison, depuis la date d'expédition. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

```
SELECT ceil(avg(date_reception - date_expedition))::text || ' jour(s)'
FROM lignes_commandes lc ;
```

7. Pour les 30 commandes les plus récentes (selon la date de commande), calculez le temps moyen de livraison de chaque commande, depuis la date de commande. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

```
SELECT count(*), ceil(avg(date_reception - date_commande))::text || ' jour(s)'
FROM (
   SELECT numero_commande, date_commande
   FROM commandes
   ORDER BY date_commande DESC
   LIMIT 30
   ) c
JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande ;
```

Note : la colonne `date_commande` de la table `commandes` n'a pas de contrainte `NOT NULL`, il est donc possible d'avoir des commandes sans date de commande renseignée. Dans ce cas, ces commandes vont remonter par défaut en haut de la liste, puisque la clause `ORDER BY` renvoie les `NULL` après les valeurs les plus grandes, et que l'on inverse le tri. Pour éviter que ces commandes ne faussent les résultats, il faut donc les exclure de la sous-requête, de la façon suivante :

```
   SELECT numero_commande, date_commande
   FROM commandes
   WHERE date_commande IS NOT NULL
   ORDER BY date_commande DESC
   LIMIT 30
```

8. Déterminez le taux de retour des marchandises (l'état à R indiquant qu'une marchandise est retournée).

```
SELECT
   round(
      sum(
         CASE etat_retour
```

```
            WHEN 'R' THEN 1.0
            ELSE 0.0
         END
      ) / count(*)::numeric * 100,
      2
   )::text || ' %' AS taux_retour
FROM lignes_commandes ;
```

La clause `FILTER` des fonctions d'agrégation permet d'écrire une telle requête plus facilement :

```
SELECT
   round(
      count(*) FILTER (WHERE etat_retour = 'R') / count(*)::numeric * 100,
      2
   )::text || ' %' AS taux_retour
FROM lignes_commandes ;
```

9. Déterminez le mode d'expédition qui est le plus rapide, en moyenne.

```
SELECT mode_expedition, avg(date_reception - date_expedition)
FROM lignes_commandes lc
GROUP BY mode_expedition
ORDER BY avg(date_reception - date_expedition) ASC
LIMIT 1 ;
```

10. Un bug applicatif est soupçonné, déterminez s'il existe des commandes dont la date de commande est postérieure à la date d'expédition des articles.

```
SELECT count(*)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
      AND c.date_commande > lc.date_expedition ;
```

11. Écrivez une requête qui corrige les données erronés en positionnant la date de commande à la date d'expédition la plus ancienne des marchandises. Vérifiez qu'elle soit correcte. Cette requête permet de corriger des calculs de statistiques sur les délais de livraison.

Afin de se protéger d'une erreur de manipulation, on ouvre une transaction :

```
BEGIN;

UPDATE commandes c_up
SET date_commande = (
   SELECT min(date_expedition)
   FROM commandes c
      JOIN lignes_commandes lc ON lc.numero_commande = c.numero_commande
         AND c.date_commande > lc.date_expedition
   WHERE c.numero_commande = c_up.numero_commande
   )
WHERE EXISTS (
   SELECT 1
   FROM commandes c2
      JOIN lignes_commandes lc ON lc.numero_commande = c2.numero_commande
```

```
        AND c2.date_commande > lc.date_expedition
      WHERE c_up.numero_commande  = c2.numero_commande
   GROUP BY 1
   ) ;
```

La requête réalisée précédemment doit à présent retourner 0 :

```
SELECT count(*)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
      AND c.date_commande > lc.date_expedition ;
```

Si c'est le cas, on valide la transaction :

```
COMMIT ;
```

Si ce n'est pas le cas, il doit y avoir une erreur dans la transaction, on l'annule :

```
ROLLBACK ;
```

> 12. Écrivez une requête qui calcule le délai total maximal de livraison de la totalité d'une commande donnée, depuis la date de la commande.

Par exemple pour la commande dont le numéro de commande est le 1 :

```
SELECT max(date_reception - date_commande)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
WHERE c.numero_commande = 1 ;
```

> 13. Écrivez une requête pour déterminer les 10 commandes dont le délai de livraison, entre la date de commande et la date de réception, est le plus important, pour l'année 2011 uniquement.

```
SELECT
   c.numero_commande,
   max(date_reception - date_commande)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
    AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY c.numero_commande
ORDER BY max(date_reception - date_commande) DESC
LIMIT 10 ;
```

> 14. Un autre bug applicatif est détecté. Certaines commandes n'ont pas de lignes de commandes. Écrivez une requête pour les retrouver.

Pour réaliser cette requête, il faut effectuer une jointure spéciale, nommée « Anti-jointure ». Il y a plusieurs façons d'écrire ce type de jointure. Les différentes méthodes sont données de la moins efficace à la plus efficace.

La version la moins performante est la suivante, avec `NOT IN` :

```
SELECT c.numero_commande
FROM commandes
WHERE numero_commande NOT IN (
   SELECT numero_commande
   FROM lignes_commandes
   ) ;
```

Il n'y a aucune corrélation entre la requête principale et la sous-requête. PostgreSQL doit donc vérifier pour chaque ligne de `commandes` que `numero_commande` n'est pas présent dans l'ensemble retourné par la sous-requête. Il est préférable d'éviter cette syntaxe.

Autre écriture possible, avec `LEFT JOIN` :

```
SELECT c.numero_commande
FROM commandes c
   LEFT JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
/* c'est le filtre suivant qui permet de ne conserver que les lignes de la
   table commandes qui n'ont PAS de correspondance avec la table
   numero_commandes */
WHERE lc.numero_commande IS NULL ;
```

Enfin, l'écriture généralement préférée, tant pour la lisibilité que pour les performances, avec `NOT EXISTS` :

```
SELECT c.numero_commande
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   ) ;
```

15. Écrivez une requête pour supprimer ces commandes. Vérifiez le travail avant de valider.

Afin de se protéger d'une erreur de manipulation, on ouvre une transaction :

```
BEGIN ;
```

La requête permettant de supprimer ces commandes est dérivée de la version `NOT EXISTS` de la requête ayant permis de trouver le problème :

```
DELETE
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   )
-- on peut renvoyer directement les numeros de commande qui ont été supprimés :
-- RETURNING numero_commande
;
```

Pour vérifier que le problème est corrigé :

```
SELECT count(*)
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   ) ;
```

Si la requête ci-dessus remonte `0`, alors la transaction peut être validée :

```
COMMIT ;
```

16. Écrivez une requête pour déterminer les 20 pièces qui ont eu le plus gros volume de commande.

```
SELECT p.nom,
   sum(quantite)
FROM pieces p
   JOIN lignes_commandes lc ON p.piece_id = lc.piece_id
GROUP BY p.nom
ORDER BY sum(quantite) DESC
LIMIT 20 ;
```

17. Affichez les fournisseurs des 20 pièces qui ont été le plus commandées sur l'année 2011.

```
SELECT co.nom, max_p.piece_id, total_pieces
FROM (
   /* cette sous-requête est sensiblement la même que celle de l'exercice
      précédent, sauf que l'on remonte cette fois l'id de la piece plutôt
      que son nom pour pouvoir faire la jointure avec pieces_fournisseurs, et
      que l'on ajoute une jointure avec commandes pour pouvoir filtrer sur
      l'année 2011 */
   SELECT
      p.piece_id,
      sum(quantite) AS total_pieces
   FROM pieces p
      JOIN lignes_commandes lc ON p.piece_id = lc.piece_id
      JOIN commandes c ON c.numero_commande = lc.numero_commande
   WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
      AND to_date('31/12/2011', 'DD/MM/YYYY')
   GROUP BY p.piece_id
   ORDER BY sum(quantite) DESC
   LIMIT 20
   ) max_p
   /* il faut passer par la table de liens pieces_fournisseurs pour récupérer
      la liste des fournisseurs d'une piece */
   JOIN pieces_fournisseurs pf ON max_p.piece_id = pf.piece_id
   JOIN fournisseurs f ON f.fournisseur_id = pf.fournisseur_id
   -- la jointure avec la table contact permet d'afficher le nom du fournisseur
   JOIN contacts co ON f.contact_id = co.contact_id ;
```

18. Affichez le pays qui a connu, en nombre, le plus de commandes sur l'année 2011.

```
SELECT nom_pays,
   count(c.numero_commande)
```

```
FROM commandes c
   JOIN clients cl ON (c.client_id = cl.client_id)
   JOIN contacts co ON (cl.contact_id = co.contact_id)
   JOIN pays p ON (co.code_pays = p.code_pays)
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY p.nom_pays
ORDER BY count(c.numero_commande) DESC
LIMIT 1 ;
```

19. Affichez pour les commandes passées en 2011, la liste des continents (`régions`) et la marge brute d'exploitation réalisée par continents, triés dans l'ordre décroissant.

```
SELECT
   nom_region,
   round(sum(quantite * prix - remise) - sum(quantite * cout_piece), 2)
    AS marge_brute
FROM
   commandes c
   JOIN lignes_commandes lc ON lc.numero_commande = c.numero_commande
   /* il faut passer par la table de liens pieces_fournisseurs pour récupérer
      la liste des fournisseurs d'une piece - attention, la condition de
      jointure entre lignes_commandes et pieces_fournisseurs porte sur deux
      colonnes ! */
   JOIN pieces_fournisseurs pf ON lc.piece_id = pf.piece_id
      AND lc.fournisseur_id = pf.fournisseur_id
   JOIN pieces p ON p.piece_id = pf.piece_id
   JOIN fournisseurs f ON f.fournisseur_id = pf.fournisseur_id
   JOIN clients cl ON c.client_id = cl.client_id
   JOIN contacts co ON cl.contact_id = co.contact_id
   JOIN pays pa ON co.code_pays = pa.code_pays
   JOIN regions r ON r.region_id = pa.region_id
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY nom_region
ORDER BY sum(quantite * prix - remise) - sum(quantite * cout_piece) DESC ;
```

20. Affichez le nom, le numéro de téléphone et le pays des fournisseurs qui ont un commentaire contenant le mot clé `Complaints` :

```
SELECT
   nom,
   telephone,
   nom_pays
FROM
   fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
WHERE f.commentaire LIKE '%Complaints%' ;
```

21. Déterminez le top 10 des fournisseurs ayant eu le plus long délai de livraison, entre la date de commande et la date de réception, pour l'année 2011 uniquement.

```
SELECT
   f.fournisseur_id,
```

```
   co.nom,
   max(date_reception - date_commande)
FROM
   lignes_commandes lc
   JOIN commandes c ON c.numero_commande = lc.numero_commande
   JOIN pieces_fournisseurs pf ON lc.piece_id = pf.piece_id
      AND lc.fournisseur_id = pf.fournisseur_id
   JOIN fournisseurs f ON pf.fournisseur_id = f.fournisseur_id
   JOIN contacts co ON f.contact_id = co.contact_id
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY f.fournisseur_id, co.nom
ORDER BY max(date_reception - date_commande) DESC
LIMIT 10 ;
```

3.15 TRAVAUX PRATIQUES 2 (SOLUTIONS)

1. Ajouter une adresse mail à chaque contact avec la concaténation du nom avec le texte « @dalibo.com ».

```
BEGIN ;

UPDATE contacts
SET email = nom||'@dalibo.com' ;

COMMIT ;
```

Note : pour éviter de mettre à jour les contacts ayant déjà une adresse mail, il suffit d'ajouter une clause WHERE :

```
UPDATE contacts
SET email = nom||'@dalibo.com'
WHERE email IS NULL ;
```

2. Concaténer nom et adresse mail des contacts français sous la forme « nom <mail> ».

```
SELECT nom||' <'||email||'>'
FROM contacts
;
```

3. Même demande mais avec le nom en majuscule et l'adresse mail en minuscule.

```
SELECT upper(nom)||' <'||lower(email)||'>'
FROM contacts
;
```

4. Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`. Mettre à jour la colonne `prix_total` des commandes avec les informations des lignes de la table `lignes_commandes`.

```
ALTER TABLE commandes ADD COLUMN prix_total numeric(10,2) ;

BEGIN ;

UPDATE commandes c
SET prix_total= (
   /* cette sous-requête fait une jointure entre lignes_commandes et la table
      commandes à mettre à jour pour calculer le prix par commande */
   SELECT SUM(quantite * prix_unitaire - remise)
   FROM lignes_commandes lc
   WHERE lc.numero_commande=c.numero_commande
)
-- on peut récupérer le détail de la mise à jour directement dans la requête :
-- RETURNING numero_commande, prix_total
;
COMMIT ;
```

Une autre variante de cette requête serait :

```
UPDATE commandes c SET prix_total=prix_calc
FROM (
    SELECT numero_commande, SUM(quantite * prix_unitaire - remise) AS prix_calc
        FROM lignes_commandes
       GROUP BY numero_commande
) as prix_detail
WHERE prix_detail.numero_commande = c.numero_commande ;
```

Bien que cette dernière variante soit moins lisible, elle est bien plus rapide sur un gros volume de données.

5. Récupérer le montant total des commandes par mois pour l'année 2010. Les montants seront arrondis à deux décimales.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(prix_total),2) AS montant_total
FROM commandes
WHERE date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1 ;
```

Attention : il n'y a pas de contrainte `NOT NULL` sur le champ `date_commande`, donc s'il existe des commandes sans date de commande, celles-ci seront agrégées à part des autres, puisque `extract()` renverra `NULL` pour ces lignes.

6. Supprimer les commandes de mai 2010.

```
BEGIN;

/* en raison de la présence de clés étrangères, il faut en premier leur
   supprimer les lignes de la table lignes_commandes correspondant aux
   commandes à supprimer */
DELETE
FROM lignes_commandes
WHERE numero_commande IN (
   SELECT numero_commande
   FROM commandes
   WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
      AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
   );

-- ensuite seulement on peut supprimer les commandes
DELETE
FROM commandes
WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY') ;

COMMIT ;
```

Le problème de l'approche précédente est d'effectuer l'opération en deux temps. Il est possible de réaliser la totalité des suppressions dans les deux tables `lignes_commandes` et `commandes` en une seule requête en utilisant une *CTE* :

```
WITH del_lc AS (
   /* ici on déclare la CTE qui va se charger de supprimer les lignes dans la
      table lignes_commandes et retourner les numeros de commande supprimés */
   DELETE
   FROM lignes_commandes
   WHERE numero_commande IN (
      SELECT numero_commande
      FROM commandes
      WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
         AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
      )
   RETURNING numero_commande
)
/* requête principale, qui supprime les commandes dont les numéros
   correspondent aux numéros de commandes remontés par la CTE */
DELETE
FROM commandes c
WHERE EXISTS (
   SELECT 1
   FROM del_lc
   WHERE del_lc.numero_commande = c.numero_commande
   ) ;
```

7. Ré-exécuter la requête trouvée au point 5.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(prix_total),2) AS montant_total
FROM commandes
GROUP BY 1
ORDER BY 1 ;
```

8. Qu'observez-vous ?

La ligne correspondant au mois de mai a disparu.

9. Corriger le problème rencontré.

```
SELECT numero_mois, round(coalesce(sum(prix_total), 0.0),2) AS montant_total
  /* la fonction generate_series permet de générer une pseudo-table d'une
     colonne contenant les chiffres de 1 à 12 */
FROM generate_series(1, 12) AS numero_mois
  /* le LEFT JOIN entre cette pseudo-table et la table commandes permet de
     s'assurer que même si aucune commande n'a eu lieu sur un mois, la ligne
     correspondante sera tout de même présente */
LEFT JOIN commandes ON extract('month' from date_commande) = numero_mois
      AND date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
      AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1 ;
```

Notez l'utilisation de la fonction `coalesce()` dans le `SELECT`, afin d'affecter la valeur `0.0` aux lignes « ajoutées » par le `LEFT JOIN` qui n'ont par défaut aucune valeur (`NULL`).

10. Créer une vue calculant le prix total de chaque commande.

```
CREATE VIEW commande_montant AS
SELECT
   numero_commande,
   sum(quantite * prix_unitaire - remise) AS total_commande
FROM lignes_commandes
GROUP BY numero_commande ;
```

11. Réécrire la requête du point 5 pour utiliser la vue créée au point 10.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(total_commande),2) AS montant_total
FROM commandes c
JOIN commande_montant cm ON cm.numero_commande = c.numero_commande
WHERE date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1 ;
```

4/ SQL avancé pour le transactionnel

4.0.1 Préambule

- SQL et PostgreSQL proposent de nombreuses possibilités avancées
  - normes SQL:99, 2003, 2008 et 2011
  - parfois, extensions propres à PostgreSQL

La norme SQL a continué d'évoluer et a bénéficié d'un grand nombre d'améliorations. Beaucoup de requêtes qu'il était difficile d'exprimer avec les premières incarnations de la norme sont maintenant faciles à réaliser avec les dernières évolutions.

Ce module a pour objectif de voir les fonctionnalités pouvant être utiles pour développer une application transactionnelle.

4.0.2 Menu

- `LIMIT/OFFSET`
- Jointures `LATERAL`
- UPSERT : INSERT ou UPDATE
- *Common Table Expressions*
- Serializable Snapshot Isolation

4.0.3 Objectifs

- Aller au-delà de SQL:92
- Concevoir des requêtes simples pour résoudre des problèmes complexes

4.1 LIMIT

- Clause `LIMIT`
- ou syntaxe en norme SQL : `FETCH FIRST xx ROWS`
- Utilisation :
  - limite le nombre de lignes du résultat

La clause `LIMIT`, ou sa déclinaison normalisée par le comité ISO `FETCH FIRST xx ROWS`, permet de limiter le nombre de lignes résultant d'une requête SQL. La syntaxe normalisée vient de DB2 d'IBM et va être amenée à apparaître sur la plupart des bases de données. La syntaxe `LIMIT` reste néanmoins disponible sur de nombreux SGBD et est plus concise.

4.1.1 LIMIT : exemple

```
SELECT *
  FROM employes
 LIMIT 2;

 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
(2 lignes)
```

L'exemple ci-dessous s'appuie sur le jeu d'essai suivant :

```
SELECT *
  FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   |             | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

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 garantit en aucune façon l'ordre

des résultats à moins d'employer la clause `ORDER BY`, et que l'ensemble des champs sur lequel on trie soit unique et non null.

Si une ligne était modifiée, changeant sa position physique dans la table, le résultat de la requête ne serait pas le même. Par exemple, en réalisant une mise à jour fictive de la ligne correspondant au matricule `00000001` :

```
UPDATE employes
   SET nom = nom
 WHERE matricule = '00000001';
```

L'ordre du résultat n'est pas garanti :

```
SELECT *
  FROM employes
 LIMIT 2;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+---------
 00000004  | Fantasio | Courrier    | 4500.00
 00000006  | Prunelle | Publication | 4000.00
(2 lignes)
```

L'application d'un critère de tri explicite permet d'obtenir la sortie souhaitée :

```
SELECT *
  FROM employes
 ORDER BY matricule
 LIMIT 2;
 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
```

4.1.2 OFFSET

- Clause OFFSET
  - à utiliser avec `LIMIT`
- Utilité :
  - pagination de résultat
  - sauter les *n* premières lignes avant d'afficher le résultat

Ainsi, en reprenant le jeu d'essai utilisé précédemment :

```
SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
```

```
-----------+----------+-------------+----------
 00000001  | Dupuis   |             | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

4.1.3 OFFSET : exemple (1/2)

- Sans offset :

```
SELECT *
  FROM employes
 LIMIT 2
 ORDER BY matricule;

 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
```

4.1.4 OFFSET : exemple (2/2)

- En sautant les deux premières lignes :

```
SELECT *
  FROM employes
 ORDER BY matricule
 LIMIT 2
 OFFSET 2;

 matricule |   nom    |   service   | salaire
-----------+----------+-------------+---------
 00000006  | Prunelle | Publication | 4000.00
 00000020  | Lagaffe  | Courrier    | 3000.00
```

4.1.5 OFFSET : problèmes

- OFFSET est problématique
  - beaucoup de données lues
  - temps de réponse dégradés
- Alternative possible
  - utilisation d'un index sur le critère de tri
  - critère de filtrage sur la page précédente
- Article sur le sujet[1]

Cependant, sur un jeu de données conséquent et une pagination importante, ce principe de fonctionnement peut devenir contre-performant. En effet, la base de données devra lire malgré tout les enregistrements qui n'apparaîtront pas dans le résultat de la requête, simplement dans le but de les compter.

Soit la table `posts` suivante (téléchargeable sur https://dali.bo/tp_posts, à laquelle on ajoute un index sur `(id_article, id_post)`):

```
\d posts
                              Table « public.posts »
  Colonne   |           Type           | Collationnement | NULL-able | Par défaut
------------+--------------------------+-----------------+-----------+------------
 id_article | integer                  |                 |           |
 id_post    | integer                  |                 |           |
 ts         | timestamp with time zone |                 |           |
 message    | text                     |                 |           |
Index :
    "posts_id_article_id_post" btree (id_article, id_post)
    "posts_ts_idx" btree (ts)
```

Si l'on souhaite récupérer les 10 premiers enregistrements :

```
SELECT  *
FROM    posts
WHERE   id_article =12
ORDER BY id_post
LIMIT   10 ;
```

on obtient le plan d'exécution[2] suivant :

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.43..18.26 rows=10 width=115)
```

[2] https://explain.dalibo.com/plan/xEs

```
        (actual time=0.043..0.053 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                 (cost=0.43..1745.88 rows=979 width=115)
               (actual time=0.042..0.051 rows=10 loops=1)
         Index Cond: (id_article = 12)
 Planning Time: 0.204 ms
 Execution Time: 0.066 ms
```

La requête est rapide car elle profite d'un index bien trié et elle ne lit que peu de données, ce qui est bien.

En revanche, si l'on saute un nombre conséquent d'enregistrements grâce à la clause `OFFSET`, la situation devient problématique :

```
SELECT  *
FROM    posts
WHERE   id_article = 12
ORDER BY id_post
LIMIT   10
OFFSET  900 ;
```

Le plan[3] n'est plus le même :

```
Limit  (cost=1605.04..1622.86 rows=10 width=115)
        (actual time=0.216..0.221 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                  (cost=0.43..1745.88 rows=979 width=115)
                  (actual time=0.018..0.194 rows=910 loops=1)
         Index Cond: (id_article = 12)
 Planning Time: 0.062 ms
 Execution Time: 0.243 ms
```

Pour répondre à la requête, PostgreSQL choisit la lecture de l'ensemble des résultats, puis leur tri, pour enfin appliquer la limite. En effet, `LIMIT` et `OFFSET` ne peuvent s'opérer que sur le résultat trié : il faut lire les 910 posts avant de pouvoir choisir les 10 derniers.

Le problème de ce plan est que, plus le jeu de données sera important, plus les temps de réponse seront importants. Ils seront encore plus importants si le tri n'est pas utilisable dans un index, ou si l'on déclenche un tri sur disque. Il faut donc trouver une solution pour les minimiser.

Les problèmes de l'utilisation de la clause `OFFSET` sont parfaitement expliqués dans cet article[4].

Dans notre cas, le principe est d'abord de créer un index qui contient le critère ainsi que le champ qui fixe la pagination (l'index existant convient). Puis on mémorise à quel `post_id` la page précédente s'est arrêtée, pour le donner comme critère de filtrage (ici `12900`). Il suffit donc de récupérer les 10 articles pour lesquels `id_article = 12` et `id_post > 12900` :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article = 12
AND     id_post> 12900
```

[3] https://explain.dalibo.com/plan/V05
[4] https://use-the-index-luke.com/fr/no-offset

```
ORDER BY id_post
LIMIT   10 ;

                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=0.43..18.29 rows=10 width=115)
        (actual time=0.018..0.024 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                         (cost=0.43..1743.02 rows=976 width=115)
                         (actual time=0.016..0.020 rows=10 loops=1)
         Index Cond: ((id_article = 12) AND (id_post > 12900))
 Planning Time: 0.111 ms
 Execution Time: 0.039 ms
```

4.2 RETURNING

- Clause RETURNING
- Utilité :
  - récupérer les enregistrements modifiés
  - avec INSERT
  - avec UPDATE
  - avec DELETE

La clause RETURNING permet de récupérer les valeurs modifiées par un ordre DML. Ainsi, la clause RETURNING associée à l'ordre INSERT permet d'obtenir une ou plusieurs colonnes des lignes insérées.

4.2.1 RETURNING : exemple

```
CREATE TABLE test_returning (id serial primary key, val integer);

INSERT INTO test_returning (val)
  VALUES (10)
RETURNING id, val;

 id | val
----+-----
  1 |  10
(1 ligne)
```

Cela permet par exemple de récupérer la valeur de colonnes portant une valeur par défaut, comme la valeur affectée par une séquence, comme sur l'exemple ci-dessus.

La clause RETURNING permet également de récupérer les valeurs des colonnes mises à jour :

```
UPDATE test_returning
   SET val = val + 10
 WHERE id = 1
 RETURNING id, val;
 id | val
----+-----
  1 |  20
(1 ligne)
```

Associée à l'ordre DELETE, il est possible d'obtenir les lignes supprimées :

```
DELETE FROM test_returning
 WHERE val < 30
 RETURNING id, val;
 id | val
----+-----
  1 |  20
(1 ligne)
```

4.3 UPSERT

- INSERT ou UPDATE ?
  - INSERT ... ON CONFLICT DO { NOTHING | UPDATE }
  - à partir de la version 9.5
- Utilité :
  - mettre à jour en cas de conflit sur un INSERT
  - ne rien faire en cas de conflit sur un INSERT

L'implémentation de l'UPSERT peut poser des questions sur la concurrence d'accès. L'implémentation de PostgreSQL de ON CONFLICT DO UPDATE est une opération atomique, c'est-à-dire que PostgreSQL garantit qu'il n'y aura pas de conditions d'exécution qui pourront amener à des erreurs. L'utilisation d'une contrainte d'unicité n'est pas étrangère à cela, elle permet en effet de pouvoir vérifier que la ligne n'existe pas, et si elle existe déjà, de verrouiller la ligne à mettre à jour de façon atomique.

En comparaison, plusieurs approches naïves présentent des problèmes de concurrences d'accès. Les différentes approches sont décrites dans cet article de depesz[5]. Elle présente toutes des problèmes de *race conditions* qui peuvent entraîner des erreurs. Une autre possibilité aurait été d'utiliser une CTE en écriture, mais elle présente également les problèmes de concurrence d'accès décrits dans l'article.

Sur des traitements d'intégration de données, il s'agit d'un comportement qui n'est pas toujours souhaitable. La norme SQL propose l'ordre MERGE pour palier à des problèmes de ce type, mais il est peu probable de le voir rapidement implémenté dans PostgreSQL[6]. L'ordre INSERT s'est toutefois vu étendu avec PostgreSQL 9.5 pour gérer les conflits à l'insertion.

Les exemples suivants s'appuient sur le jeu de données suivant :

```
\d employes
       Table "public.employes"
  Column   |     Type     | Modifiers
-----------+--------------+-----------
 matricule | character(8) | not null
 nom       | text         | not null
 service   | text         |
 salaire   | numeric(7,2) |
Indexes:
    "employes_pkey" PRIMARY KEY, btree (matricule)
```

[5] https://www.depesz.com/2012/06/10/why-is-upsert-so-complicated/

[6] La solution actuelle semble techniquement meilleure et la solution actuelle a donc été choisie. Le wiki du projet PostgreSQL montre que l'ordre MERGE a été étudié et qu'un certain nombre d'aspects cruciaux n'ont pas été spécifiés, amenant le projet PostgreSQL à utiliser sa propre version. Voir la documentation : https://wiki.postgresql.org/wiki/UPSERT#MERGE_disadvantages.

```
SELECT * FROM employes ;
```

```
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
```

4.3.1 UPSERT : problème à résoudre

- Insérer une ligne déjà existante provoque une erreur :

```
INSERT INTO employes (matricule, nom, service, salaire)
 VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00);
ERROR:  duplicate key value violates unique constraint
        "employes_pkey"
DETAIL:  Key (matricule)=(00000001) already exists.
```

Si l'on souhaite insérer une ligne contenant un matricule déjà existant, une erreur de clé dupliquée est levée et toute la transaction est annulée.

4.3.2 ON CONFLICT DO NOTHING

- la clause `ON CONFLICT DO NOTHING` évite d'insérer une ligne existante :

```
=# INSERT INTO employes (matricule, nom, service, salaire)
   VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00)
   ON CONFLICT DO NOTHING;
INSERT 0 0
```

Les données n'ont pas été modifiées :

```
=# SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
```

```
 00000001  | Dupuis   | Direction   | 10000.00
(5 rows)
```

La transaction est toujours valide.

4.3.3 ON CONFLICT DO NOTHING : syntaxe

```
INSERT ....
ON CONFLICT
  DO NOTHING;
```

Il suffit d'indiquer à PostgreSQL de ne rien faire en cas de conflit sur une valeur dupliquée avec la clause `ON CONFLICT DO NOTHING` placée à la fin de l'ordre INSERT qui peut poser problème.

Dans ce cas, si une rupture d'unicité est détectée, alors PostgreSQL ignorera l'erreur, silencieusement. En revanche, si une erreur apparaît sur une autre contrainte, l'erreur sera levée.

En prenant l'exemple suivant :

```
CREATE TABLE test_upsert (
  i serial PRIMARY KEY,
  v text UNIQUE,
  x integer CHECK (x > 0)
);

INSERT INTO test_upsert (v, x) VALUES ('x', 1);
```

L'insertion d'une valeur dupliquée provoque bien une erreur d'unicité :

```
INSERT INTO test_upsert (v, x) VALUES ('x', 1);
ERROR:  duplicate key value violates unique constraint "test_upsert_v_key"
```

L'erreur d'unicité est bien ignorée si la ligne existe déjà, le résultat est `INSERT 0 0` qui indique qu'aucune ligne n'a été insérée :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 1)
 ON CONFLICT DO NOTHING;
INSERT 0 0
```

L'insertion est aussi ignorée si l'on tente d'insérer des lignes rompant la contrainte d'unicité mais ne comportant pas les mêmes valeurs pour d'autres colonnes :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 4)
 ON CONFLICT DO NOTHING;
INSERT 0 0
```

Si l'on insère une valeur interdite par la contrainte CHECK, une erreur est bien levée :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 0)
 ON CONFLICT DO NOTHING;
ERROR:  new row for relation "test_upsert" violates check constraint
        "test_upsert_x_check"
DETAIL:  Failing row contains (4, x, 0).
```

4.3.4 ON CONFLICT DO UPDATE

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'M. Pirate', 'Direction', 0.00)
ON CONFLICT (matricule)
   DO UPDATE SET salaire = employes.salaire,
                 nom = excluded.nom
RETURNING *;

 matricule |    nom    |   service   | salaire
-----------+-----------+-------------+----------
 00000001  | M. Pirate | Direction   | 50000.00
```

La clause ON CONFLICT permet de déterminer une colonne sur laquelle le conflit peut arriver. Cette colonne ou ces colonnes doivent porter une contrainte d'unicité ou une contrainte d'exclusion, c'est à dire une contrainte portée par un index. La clause DO UPDATE associée fait référence aux valeurs rejetées par le conflit à l'aide de la pseudo-table `excluded`. Les valeurs courantes sont accessibles en préfixant les colonnes avec le nom de la table. L'exemple montre cela.

Avec la requête de l'exemple, on voit que le salaire du directeur n'a pas été modifié, mais son nom l'a été :

```
SELECT * FROM employes ;
 matricule |    nom    |   service   | salaire
-----------+-----------+-------------+----------
 00000004  | Fantasio  | Courrier    |  4500.00
 00000006  | Prunelle  | Publication |  4000.00
 00000020  | Lagaffe   | Courrier    |  3000.00
 00000040  | Lebrac    | Publication |  3000.00
 00000001  | M. Pirate | Direction   | 10000.00
(5 rows)
```

La clause ON CONFLICT permet également de définir une contrainte d'intégrité sur laquelle on réagit en cas de conflit :

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00)
ON CONFLICT ON CONSTRAINT employes_pkey
   DO UPDATE SET salaire = excluded.salaire;
```

On remarque que seul le salaire du directeur a changé :

```
SELECT * FROM employes ;
 matricule |   nom     |   service    | salaire
-----------+-----------+--------------+----------
 00000004  | Fantasio  | Courrier     |  4500.00
 00000006  | Prunelle  | Publication  |  4000.00
 00000020  | Lagaffe   | Courrier     |  3000.00
 00000040  | Lebrac    | Publication  |  3000.00
 00000001  | M. Pirate | Direction    | 50000.00
(5 rows)
```

4.3.5 ON CONFLICT DO UPDATE

- Avec plusieurs lignes insérées :

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000002', 'Moizelle Jeanne', 'Publication', 3000.00),
       ('00000040', 'Lebrac', 'Publication', 3100.00)
ON CONFLICT (matricule)
   DO UPDATE SET salaire = employes.salaire,
                 nom = excluded.nom
RETURNING *;

 matricule |       nom       |   service    | salaire
-----------+-----------------+--------------+----------
 00000002  | Moizelle Jeanne | Publication  |  3000.00
 00000040  | Lebrac          | Publication  |  3000.00
```

Bien sûr, on peut insérer plusieurs lignes, INSERT ON CONFLICT réagira uniquement sur les doublons :

La nouvelle employée, *Moizelle Jeanne* a été intégrée dans la tables des employés, et *Lebrac* a été traité comme un doublon, en appliquant la règle de mise à jour vue plus haut : seul le nom est mis à jour et le salaire est inchangé.

```
SELECT * FROM employes ;
 matricule |       nom       |   service    | salaire
-----------+-----------------+--------------+----------
 00000004  | Fantasio        | Courrier     |  4500.00
 00000006  | Prunelle        | Publication  |  4000.00
 00000020  | Lagaffe         | Courrier     |  3000.00
 00000001  | M. Pirate       | Direction    | 50000.00
 00000002  | Moizelle Jeanne | Publication  |  3000.00
 00000040  | Lebrac          | Publication  |  3000.00
(6 rows)
```

À noter que la clause `SET salaire = employes.salaire` est inutile, c'est ce que fait PostgreSQL implicitement.

4.3.6 ON CONFLICT DO UPDATE : syntaxe

- Colonne(s) portant(s) une contrainte d'unicité
- Pseudo-table *excluded*

```
INSERT ....
ON CONFLICT (<colonne clé>)
  DO UPDATE
        SET colonne_a_modifier = excluded.colonne,
            autre_colonne_a_modifier = excluded.autre_colonne,
            ...;
```

Si l'on choisit de réaliser une mise à jour plutôt que de générer une erreur, on utilisera la clause `ON CONFLICT DO UPDATE`. Il faudra dans ce cas préciser la ou les colonnes qui portent une contrainte d'unicité. Cette contrainte d'unicité permettra de détecter la duplication de valeur, PostgreSQL pourra alors appliquer la règle de mise à jour édictée.

La règle de mise à jour permet de définir très finement les colonnes à mettre à jour et les colonnes à ne pas mettre à jour. Dans ce contexte, la pseudo-table *excluded* représente l'ensemble rejeté par l'`INSERT`. Il faudra explicitement indiquer les colonnes dont la valeur sera mise à jour à partir des valeurs que l'on tente d'insérer, reprise de la pseudo-table *excluded* :

```
ON CONFLICT (...)
  DO UPDATE
  SET colonne = excluded.colonne,
      autre_colonne = excluded.autre_colonne,
      ...
```

En alternative, il est possible d'indiquer un nom de contrainte plutôt que le nom d'une colonne portant une contrainte d'unicité :

```
INSERT ....
ON CONFLICT ON CONSTRAINT nom_contrainte
  DO UPDATE
        SET colonne_a_modifier = excluded.colonne,
            autre_colonne_a_modifier = excluded.autre_colonne,
            ...;
```

De plus amples informations quant à la syntaxe sont disponibles dans la documentation[7].

[7] https://www.postgresql.org/docs/current/static/sql-insert.html

4.4 LATERAL

- Jointures LATERAL
  - SQL:99
  - PostgreSQL 9.3
  - équivalent d'une boucle `foreach`
- Utilisations
  - top-N à partir de plusieurs tables
  - jointure avec une fonction retournant un ensemble

LATERAL apparaît dans la révision de la norme SQL de 1999. Elle permet d'appliquer une requête ou une fonction sur le résultat d'une table.

4.4.1 LATERAL : avec une sous-requête

- Jointure LATERAL
  - équivalent de *foreach*
- Utilité :
  - Top-N à partir de plusieurs tables
  - exemple : *afficher les 5 derniers messages des 5 derniers sujets actifs d'un forum*

La clause LATERAL existe dans la norme SQL depuis plusieurs années. L'implémentation de cette clause dans la plupart des SGBD reste cependant relativement récente.

Elle permet d'utiliser les données de la requête principale dans une sous-requête. La sous-requête sera appliquée à chaque enregistrement retourné par la requête principale.

4.4.2 LATERAL : exemple

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets,
  LATERAL(SELECT date_publication,
                 substr(message, 0, 100) AS extrait
            FROM messages
           WHERE sujets.sujet_id = messages.sujet_id
           ORDER BY date_publication DESC
           LIMIT 5) top_5_messages
 ORDER BY sujets.date_modification DESC,
           top_5_messages.date_publication DESC
 LIMIT 25;
```

L'exemple ci-dessus montre comment afficher les 5 derniers messages postés sur les 5 derniers sujets actifs d'un forum avec la clause LATERAL.

Une autre forme d'écriture emploie le mot clé JOIN, inutile dans cet exemple. Il peut avoir son intérêt si l'on utilise une jointure externe (LEFT JOIN par exemple si un sujet n'impliquait pas forcément la présence d'un message) :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets
  JOIN LATERAL(SELECT date_publication, substr(message, 0, 100) AS extrait
           FROM messages
          WHERE sujets.sujet_id = messages.sujet_id
          ORDER BY date_publication DESC
          LIMIT 5) top_5_messages
    ON (true) -- condition de jointure toujours vraie
 ORDER BY sujets.date_modification DESC, top_5_messages.date_publication DESC
 LIMIT 25;
```

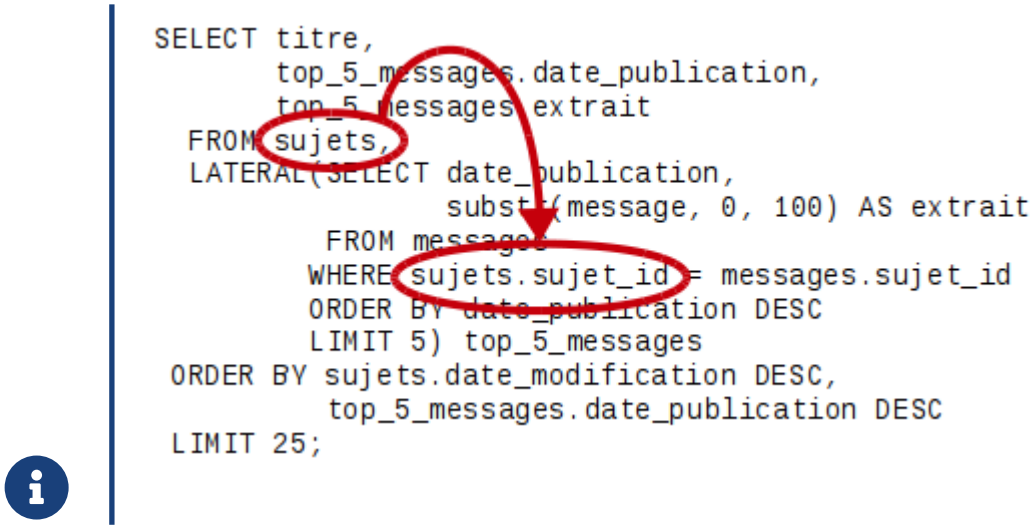
Il aurait été possible de réaliser cette requête par d'autres moyens, mais LATERAL permet d'obtenir la requête la plus performante. Une autre approche quasiment aussi performante aurait été de faire appel à une fonction retournant les 5 enregistrements souhaités.

À noter qu'une colonne `date_modification` a été ajouté à la table `sujets` afin de déterminer rapidement les derniers sujets modifiés. Sans cela, il faudrait parcourir l'ensemble des sujets, récupérer la date de publication des derniers messages avec une jointure LATERAL et récupérer les 5 derniers sujets actifs. Cela nécessite de lire beaucoup de données. Un trigger positionné sur la table `messages` permettra d'entretenir la colonne `date_modification` sur la table `sujets` sans difficulté. Il s'agit donc ici d'une entorse aux règles de modélisation en vue d'optimiser les traitements.

Un index sur les colonnes `sujet_id` et `date_publication` permettra de minimiser les accès pour cette requête :

```
CREATE INDEX ON messages (sujet_id, date_publication DESC);
```

4.4.3 LATERAL : principe



Si nous n'avions pas la clause LATERAL, nous pourrions être tentés d'écrire la requête suivante :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets
  JOIN (SELECT date_publication, substr(message, 0, 100) AS extrait
          FROM messages
         WHERE sujets.sujet_id = messages.sujet_id
         ORDER BY date_message DESC
         LIMIT 5) top_5_messages
 ORDER BY sujets.date_modification DESC
 LIMIT 25;
```

Cependant, la norme SQL interdit une telle construction, il n'est pas possible de référencer la table principale dans une sous-requête. Mais avec la clause LATERAL, la sous-requête peut faire appel à la table principale.

4.4.4 LATERAL : avec une fonction

- Utilisation avec une fonction retournant un ensemble
  - clause LATERAL optionnelle
- Utilité :
  - extraire les données d'un tableau ou d'une structure JSON sous la forme tabulaire
  - utiliser une fonction métier qui retourne un ensemble X selon un ensemble Y fourni

L'exemple ci-dessous montre qu'il est possible d'utiliser une fonction retournant un ensemble (SRF pour *Set Returning Functions*).

4.4.5 LATERAL : exemple avec une fonction

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets,
       get_top_5_messages(sujet_id) AS top_5_messages
 ORDER BY sujets.date_modification DESC
 LIMIT 25;
```

La fonction `get_top_5_messages` est la suivante :

```
CREATE OR REPLACE FUNCTION get_top_5_messages (p_sujet_id integer)
RETURNS TABLE (date_publication timestamp, extrait text)
AS $PROC$
BEGIN
  RETURN QUERY SELECT date_publication, substr(message, 0, 100) AS extrait
    FROM messages
   WHERE messages.sujet_id = p_sujet_id
   ORDER BY date_publication DESC
   LIMIT 5;
END;
$PROC$ LANGUAGE plpgsql;
```

La clause LATERAL n'est pas obligatoire, mais elle s'utiliserait ainsi :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets, LATERAL get_top_5_messages(sujet_id) AS top_5_messages
 ORDER BY sujets.date_modification DESC LIMIT 25;
```

4.5 COMMON TABLE EXPRESSIONS

- Common Table Expressions
  - clauses `WITH` et `WITH RECURSIVE`
- Utilité :
  - factoriser des sous-requêtes

4.5.1 CTE et SELECT

- Utilité
  - factoriser des sous-requêtes
  - améliorer la lisibilité d'une requête

Les CTE permettent de factoriser la définition d'une sous-requête qui pourrait être appelée plusieurs fois.

Une CTE est exprimée avec la clause `WITH`. Cette clause permet de définir des vues éphémères qui seront utilisées les unes après les autres et au final utilisées dans la requête principale.

Avant la version 12, une CTE était forcément matérialisée. À partir de la version 12, ce n'est plus le cas. Le seul moyen de s'en assurer revient à ajouter la clause `MATERIALIZED`.

4.5.2 CTE et SELECT : exemple

```
WITH resultat AS (
   /* requête complexe */
)
SELECT *
  FROM resultat
 WHERE nb < 5;
```

On utilise principalement une CTE pour factoriser la définition d'une sous-requête commune, comme dans l'exemple ci-dessus.

Un autre exemple un peu plus complexe :

```
WITH resume_commandes AS (
SELECT c.numero_commande, c.client_id, quantite*prix_unitaire AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
)
SELECT type_client, NULL AS pays, SUM(montant) AS montant_total_commande
  FROM resume_commandes
  JOIN clients
    ON (resume_commandes.client_id = clients.client_id)
 GROUP BY type_client
UNION ALL
SELECT NULL, code_pays AS pays, SUM(montant)
  FROM resume_commandes r
  JOIN clients cl
    ON (r.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 GROUP BY code_pays;
```

Le plan d'exécution de la requête montre que la vue `resume_commandes` est exécutée une seule fois et son résultat est utilisé par les deux opérations de regroupements définies dans la requête principale :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=244618.50..323855.66 rows=12 width=67)
   CTE resume_commandes
     ->  Hash Join  (cost=31886.90..174241.18 rows=1216034 width=26)
           Hash Cond: (l.numero_commande = c.numero_commande)
           ->  Seq Scan on lignes_commandes l
                    (cost=0.00..73621.47 rows=3141947 width=18)
           ->  Hash  (cost=25159.00..25159.00 rows=387032 width=16)
                 ->  Seq Scan on commandes c
                        (cost=0.00..25159.00 rows=387032 width=16)
                      Filter: ((date_commande >= '2014-01-01'::date)
                               AND (date_commande <= '2014-12-31'::date))
   ->  HashAggregate  (cost=70377.32..70377.36 rows=3 width=34)
         Group Key: clients.type_client
         ->  Hash Join  (cost=3765.00..64297.15 rows=1216034 width=34)
               Hash Cond: (resume_commandes.client_id = clients.client_id)
               ->  CTE Scan on resume_commandes
                        (cost=0.00..24320.68 rows=1216034 width=40)
               ->  Hash  (cost=2026.00..2026.00 rows=100000 width=10)
                     ->  Seq Scan on clients
                            (cost=0.00..2026.00 rows=100000 width=10)
   ->  HashAggregate  (cost=79236.89..79237.00 rows=9 width=35)
         Group Key: co.code_pays
         ->  Hash Join  (cost=12624.57..73156.72 rows=1216034 width=35)
               Hash Cond: (r.client_id = cl.client_id)
```

```
->  CTE Scan on resume_commandes r
      (cost=0.00..24320.68 rows=1216034 width=40)
->  Hash  (cost=10885.57..10885.57 rows=100000 width=11)
      ->  Hash Join
            (cost=3765.00..10885.57 rows=100000 width=11)
          Hash Cond: (co.contact_id = cl.contact_id)
          ->  Seq Scan on contacts co
                (cost=0.00..4143.05 rows=110005 width=11)
          ->  Hash (cost=2026.00..2026.00 rows=100000 width=16)
                ->  Seq Scan on clients cl
                      (cost=0.00..2026.00 rows=100000 width=16)
```

Si la requête avait été écrite sans CTE, donc en exprimant deux fois la même sous-requête, le coût d'exécution aurait été multiplié par deux car il aurait fallu exécuter la sous-requête deux fois au lieu d'une.

On utilise également les CTE pour améliorer la lisibilité des requêtes complexes, mais cela peut poser des problèmes d'optimisations, comme cela sera discuté plus bas.

4.5.3 CTE et SELECT : syntaxe

```
WITH nom_vue1 AS [ [ NOT ] MATERIALIZED ] (
 <requête pour générer la vue 1>
)
SELECT *
  FROM nom_vue1;
```

La syntaxe de définition d'une vue est donnée ci-dessus.

On peut néanmoins enchaîner plusieurs vues les unes à la suite des autres :

```
WITH nom_vue1 AS (
 <requête pour générer la vue 1>
), nom_vue2 AS (
 <requête pour générer la vue 2, pouvant utiliser la vue 1>
)
<requête principale utilisant vue 1 et/ou vue2>;
```

4.5.4 CTE et barrière d'optimisation

- Attention, une CTE est une barrière d'optimisation !
  - pas de transformations
  - pas de propagation des prédicats
- Sauf à partir de la version 12
  - clause `MATERIALIZED` pour obtenir cette barrière

Il faut néanmoins être vigilant car l'optimiseur n'inclut pas la définition des CTE dans la requête principale quand il réalise les différentes passes d'optimisations.

Par exemple, sans CTE, si un prédicat appliqué dans la requête principale peut être remonté au niveau d'une sous-requête, l'optimiseur de PostgreSQL le réalisera :

```
EXPLAIN
SELECT MAX(date_embauche)
  FROM (SELECT * FROM employes WHERE num_service = 4) e
 WHERE e.date_embauche < '2006-01-01';
```

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Aggregate  (cost=1.21..1.22 rows=1 width=4)
   ->  Seq Scan on employes  (cost=0.00..1.21 rows=2 width=4)
         Filter: ((date_embauche < '2006-01-01'::date) AND (num_service = 4))
(3 lignes)
```

Les deux prédicats `num_service = 4` et `date_embauche < '2006-01-01'` ont été appliqués en même temps, réduisant ainsi le jeu de données à considérer dès le départ. En anglais, on parle de *predicate push-down*.

Une requête équivalente basée sur une CTE ne permet pas d'appliquer le filtre au plus tôt : ici le filtre inclus dans la CTE est appliqué, pas le second.

```
EXPLAIN
WITH e AS
  (SELECT * FROM employes WHERE num_service = 4)
SELECT MAX(date_embauche)
 FROM e
WHERE e.date_embauche < '2006-01-01';
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=1.29..1.30 rows=1 width=4)
   CTE e
     ->  Seq Scan on employes  (cost=0.00..1.18 rows=5 width=43)
           Filter: (num_service = 4)
```

```
   ->  CTE Scan on e  (cost=0.00..0.11 rows=2 width=4)
         Filter: (date_embauche < '2006-01-01'::date)
```

On peut se faire piéger également en voulant calculer trop de choses dans les CTE. Dans cet autre exemple, on cherche à afficher les 7 commandes d'un client donné, le cumul des valeurs des lignes par commande étant réalisé dans un CTE :

```
EXPLAIN ANALYZE
WITH nos_commandes AS
(
  SELECT c.numero_commande, c.client_id, SUM(quantite*prix_unitaire) AS montant
  FROM   commandes c
  JOIN   lignes_commandes l
  ON     (c.numero_commande = l.numero_commande)
  GROUP BY 1,2
)
SELECT clients.client_id, type_client, nos_commandes.*
FROM   nos_commandes
INNER JOIN clients
ON     (nos_commandes.client_id = clients.client_id)
WHERE clients.client_id = 6845
;
```

```
                          QUERY PLAN
-----------------------------------------------------------------------
Nested Loop  (cost=154567.68..177117.90 rows=5000 width=58)
           (actual time=7.757..5526.148 rows=7 loops=1)
   CTE nos_commandes
     ->  GroupAggregate  (cost=3.51..154567.39 rows=1000000 width=48)
                  (actual time=0.043..5076.121 rows=1000000 loops=1)
           Group Key: c.numero_commande
           ->  Merge Join  (cost=3.51..110641.89 rows=3142550 width=26)
                           (actual time=0.017..2511.385 rows=3142632 loops=1)
                 Merge Cond: (c.numero_commande = l.numero_commande)
                 ->  Index Scan using commandes_pkey on commandes c
                           (cost=0.42..16290.72 rows=1000000 width=16)
                           (actual time=0.008..317.547 rows=1000000 loops=1)
                 ->  Index Scan using lignes_commandes_pkey on lignes_commandes l
                           (cost=0.43..52570.08 rows=3142550 width=18)
                           (actual time=0.006..1030.420 rows=3142632 loops=1)
   ->  Index Scan using clients_pkey on clients
                         (cost=0.29..0.51 rows=1 width=10)
                         (actual time=0.009..0.009 rows=1 loops=1)
         Index Cond: (client_id = 6845)
   ->  CTE Scan on nos_commandes  (cost=0.00..22500.00 rows=5000 width=48)
                                  (actual time=7.746..5526.128 rows=7 loops=1)
         Filter: (client_id = 6845)
         Rows Removed by Filter: 999993
```

Notez que la construction de la CTE fait un calcul sur l'intégralité des 5000 commandes et brasse un million de lignes. Puis, une fois connu le `client_id`, PostgreSQL parcourt cette CTE pour en récupérer une seule ligne. C'est évidemment extrêmement coûteux et dure plusieurs secondes.

Alors que sans la CTE, l'optimiseur se permet de faire la jointure avec les tables, donc à filtrer sur le client demandé, et ne fait la somme des lignes qu'après, en quelques millisecondes.

```
EXPLAIN ANALYZE
SELECT clients.client_id, type_client, nos_commandes.*
FROM
(
  SELECT c.numero_commande, c.client_id, SUM(quantite*prix_unitaire) AS montant
  FROM   commandes c
  JOIN   lignes_commandes l
  ON     (c.numero_commande = l.numero_commande)
  GROUP BY 1,2
) AS nos_commandes
INNER JOIN clients
ON    (nos_commandes.client_id = clients.client_id)
WHERE clients.client_id = 6845
;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=12.83..13.40 rows=11 width=58)
              (actual time=0.113..0.117 rows=7 loops=1)
   ->  Index Scan using clients_pkey on clients  (cost=0.29..0.51 rows=1 width=10)
                                                 (actual time=0.007..0.007 rows=1 loops=1)
         Index Cond: (client_id = 6845)
   ->  HashAggregate  (cost=12.54..12.67 rows=11 width=48)
                      (actual time=0.106..0.108 rows=7 loops=1)
         Group Key: c.numero_commande
         ->  Nested Loop  (cost=0.85..12.19 rows=35 width=26)
                          (actual time=0.028..0.087 rows=23 loops=1)
               ->  Index Scan using commandes_clients_fkey on commandes c
                                    (cost=0.42..1.82 rows=11 width=16)
                                    (actual time=0.022..0.028 rows=7 loops=1)
                     Index Cond: (client_id = 6845)
               ->  Index Scan using lignes_commandes_pkey on lignes_commandes l
                                    (cost=0.43..0.89 rows=5 width=18)
                                    (actual time=0.006..0.007 rows=3 loops=7)
                     Index Cond: (numero_commande = c.numero_commande)
```

En plus d'améliorer la lisibilité et d'éviter la duplication de code, le mécanisme des CTE est aussi un moyen contourner certaines limitations de l'optimiseur de PostgreSQL en vue de contrôler précisément le plan d'exécution d'une requête.

Ce principe de fonctionnement a changé avec la version 12 de PostgreSQL. Par défaut, il n'y a pas de matérialisation mais celle-ci peut être forcée avec l'option `MATERIALIZED`.

4.5.5 CTE en écriture

- CTE avec des requêtes en modification
  - avec INSERT/UPDATE/DELETE
  - et éventuellement RETURNING
  - obligatoirement exécuté sur PostgreSQL
- Exemple d'utilisation :
  - archiver des données
  - partitionner les données d'une table
  - débugger une requête complexe

4.5.6 CTE en écriture : exemple

```
WITH donnees_a_archiver AS (
DELETE FROM donnes_courantes
 WHERE date < '2015-01-01'
 RETURNING *
)
INSERT INTO donnes_archivees
SELECT * FROM donnees_a_archiver;
```

La requête d'exemple permet d'archiver des données dans une table dédiée à l'archivage en utilisant une CTE en écriture. L'emploi de la clause RETURNING permet de récupérer les lignes purgées.

Le même principe s'applique pour une table que l'on vient de partitionner. Les enregistrements se trouvent initialement dans la table mère, il faut les répartir sur les différentes partitions. On utilisera une requête reposant sur le même principe que la précédente. L'ordre INSERT visera la table principale si l'on souhaite utiliser le trigger de partition pour répartir les données. Il pourra également viser une partition donnée afin d'éviter le surcoût du trigger de partition.

En plus de ce cas d'usage simple, il est possible d'utiliser cette fonctionnalité pour débugger une requête complexe.

```
WITH sous-requete1 AS (

),
debug_sous-requete1 AS (
INSERT INTO debug_sousrequete1
```

```
SELECT * FROM sous-requete1
), sous-requete2 AS (
SELECT ...
  FROM sous-requete1
  JOIN ....
 WHERE ....
 GROUP BY ...
),
debug_sous-requete2 AS (
INSERT INTO debug_sousrequete2
SELECT * FROM sous-requete2
)
SELECT *
  FROM sous-requete2;
```

On peut également envisager une requête CTE en écriture pour émuler une requête MERGE pour réaliser une intégration de données complexe, là où l'UPSERT ne serait pas suffisant. Il faut toutefois avoir à l'esprit qu'une telle requête présente des problèmes de concurrences d'accès, pouvant entraîner des résultats inattendus si elle est employée alors que d'autres sessions modifient les données. On se contentera d'utiliser une telle requête dans des traitements batchs.

Il est important de noter que sur PostgreSQL, chaque sous-partie d'une CTE qui exécute une opération de mise à jour sera exécutée, même si elle n'est pas explicitement appelée. Par exemple :

```
WITH del AS (DELETE FROM nom_table),
fonction_en_ecriture AS (SELECT * FROM fonction_en_ecriture())
SELECT 1;
```

supprimera l'intégralité des données de la table `nom_table`, mais n'appellera pas la fonction `fonction_en_ecriture()`, même si celle-ci effectue des écritures.

4.5.7 CTE récursive

- SQL permet d'exprimer des récursions
  - WITH RECURSIVE
- Utilité :
  - récupérer une arborescence de menu hiérarchique
  - parcourir des graphes (réseaux sociaux, etc.)

Le langage SQL permet de réaliser des récursions avec des CTE récursives. Son principal intérêt est de pouvoir parcourir des arborescences, comme par exemple des arbres généalogiques, des arborescences de service ou des entrées de menus hiérarchiques.

Il permet également de réaliser des parcours de graphes, mais les possibilités en SQL sont plus limitées de ce côté-là. En effet, SQL utilise un algorithme de type *Breadth First* (parcours en largeur) où PostgreSQL produit tout le niveau courant, et approfondit ensuite la récursion. Ce fonctionnement est à l'opposé d'un algorithme *Depth First* (parcours en profondeur) où chaque branche est explorée à fond individuellement avant de passer à la branche suivante. Ce principe de fonctionnement de l'implémentation dans SQL peut poser des problèmes sur des recherches de types réseaux sociaux où des bases de données orientées graphes, tel que Neo4J, seront bien plus efficaces. À noter que l'extension pgRouting implémente des algorithmes de parcours de graphes plus efficace. Cela permet de rester dans PostgreSQL mais nécessite un certain formalisme et il faut avoir conscience que pgRouting n'est pas l'outil le plus efficace car il génère un graphe en mémoire à chaque requête à résoudre, qui est perdu après l'appel.

4.5.8 CTE récursive : exemple (1/2)

```
WITH RECURSIVE suite AS (
SELECT 1 AS valeur
UNION ALL
SELECT valeur + 1
  FROM suite
 WHERE valeur < 10
)
SELECT * FROM suite;
```

Voici le résultat de cette requête :

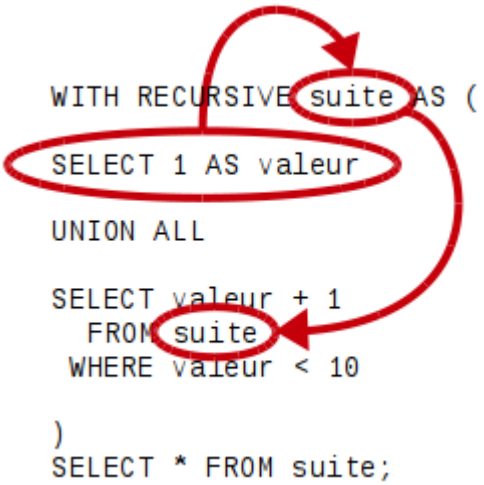
```
valeur
--------
      1
      2
      3
      4
      5
      6
      7
      8
      9
     10
```

L'exécution de cette requête commence avec le `SELECT 1 AS valeur` (la requête avant le `UNION ALL`), d'où la première ligne avec la valeur 1. Puis PostgreSQL exécute le `SELECT valeur+1 FROM suite WHERE valeur < 10` tant que cette requête renvoie des lignes. À la première exécution, il additionne 1 avec la valeur précédente (1), ce qui fait qu'il renvoie 2. A la deuxième exécution, il additionne 1 avec la valeur précédente (2), ce qui fait qu'il renvoie 3. Etc. La récursivité s'arrête quand la requête ne renvoie plus de ligne, autrement dit quand la colonne vaut 10.

Cet exemple n'a aucun autre intérêt que de présenter la syntaxe permettant de réaliser une récursion en langage SQL.

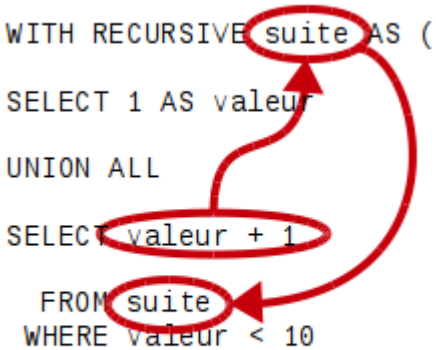
4.5.9 CTE récursive : principe

- 1ère étape : initialisation de la récursion



4.5.10 CTE récursive : principe

- récursion : la requête s'appelle elle-même



```
WITH RECURSIVE suite AS (

SELECT 1 AS valeur

UNION ALL

SELECT valeur + 1

  FROM suite
 WHERE valeur < 10

)
SELECT * FROM suite;
```

4.5.11 CTE récursive : exemple (2/2)

```
WITH RECURSIVE parcours_menu AS (
SELECT menu_id, libelle, parent_id,
       libelle AS arborescence
  FROM entrees_menu
 WHERE libelle = 'Terminal'
   AND parent_id IS NULL
UNION ALL
SELECT menu.menu_id, menu.libelle, menu.parent_id,
       arborescence || '/' || menu.libelle
  FROM entrees_menu menu
  JOIN parcours_menu parent
    ON (menu.parent_id = parent.menu_id)
)
SELECT * FROM parcours_menu;
```

Cet exemple suivant porte sur le parcours d'une arborescence de menu hiérarchique.

Une table `entrees_menu` est créée :

```
CREATE TABLE entrees_menu (menu_id serial primary key, libelle text not null,
                           parent_id integer);
```

Elle dispose du contenu suivant :

```
SELECT * FROM entrees_menu;
 menu_id |          libelle           | parent_id
---------+----------------------------+-----------
       1 | Fichier                    |
       2 | Edition                    |
       3 | Affichage                  |
       4 | Terminal                   |
       5 | Onglets                    |
       6 | Ouvrir un onglet           |         1
       7 | Ouvrir un terminal         |         1
       8 | Fermer l'onglet            |         1
       9 | Fermer la fenêtre          |         1
      10 | Copier                     |         2
      11 | Coller                     |         2
      12 | Préférences                |         2
      13 | Général                    |        12
      14 | Apparence                  |        12
      15 | Titre                      |        13
      16 | Commande                   |        13
      17 | Police                     |        14
      18 | Couleur                    |        14
      19 | Afficher la barre d'outils |         3
      20 | Plein écran                |         3
      21 | Modifier le titre          |         4
      22 | Définir l'encodage         |         4
      23 | Réinitialiser              |         4
      24 | UTF-8                      |        22
      25 | Europe occidentale         |        22
      26 | Europe centrale            |        22
      27 | ISO-8859-1                 |        25
      28 | ISO-8859-15                |        25
      29 | WINDOWS-1252               |        25
      30 | ISO-8859-2                 |        26
      31 | ISO-8859-3                 |        26
      32 | WINDOWS-1250               |        26
      33 | Onglet précédent           |         5
      34 | Onglet suivant             |         5
(34 rows)
```

Nous allons définir une CTE récursive qui va afficher l'arborescence du menu *Terminal*. La récursion va donc commencer par chercher la ligne correspondant à cette entrée de menu dans la table `entrees_menu`. Une colonne calculée arborescence est créée, elle servira plus tard dans la récursion :

```
SELECT menu_id, libelle, parent_id, libelle AS arborescence
  FROM entrees_menu
 WHERE libelle = 'Terminal'
   AND parent_id IS NULL
```

La requête qui réalisera la récursion est une jointure entre le résultat de l'itération précédente, obtenu par la vue `parcours_menu` de la CTE, qui réalisera une jointure avec la table `entrees_menu` sur

la colonne `entrees_menu.parent_id` qui sera jointe à la colonne `menu_id` de l'itération précédente.

La condition d'arrêt de la récursion n'a pas besoin d'être exprimée. En effet, les entrées terminales des menus ne peuvent pas être jointes avec de nouvelles entrées de menu, car il n'y a pas d'autre correspondance avec `parent_id`).

On obtient ainsi la requête CTE récursive présentée ci-dessus.

À titre d'exemple, voici l'implémentation du jeu des six degrés de Kevin Bacon en utilisant pgRouting :

```
WITH dijkstra AS (
SELECT seq, id1 AS node, id2 AS edge, cost
  FROM pgr_dijkstra('
  SELECT f.film_id AS id,
         f.actor_id::integer AS source,
         f2.actor_id::integer AS target,
         1.0::float8 AS cost
    FROM film_actor f
    JOIN film_actor f2
      ON (f.film_id = f2.film_id and f.actor_id <> f2.actor_id)'
  , 29539, 29726, false, false)
)
SELECT *
  FROM actors
  JOIN dijkstra
  on (dijkstra.node = actors.actor_id) ;

 actor_id |   actor_name    | seq | node  | edge | cost
----------+-----------------+-----+-------+------+------
    29539 | Kevin Bacon     |   0 | 29539 | 1330 |    1
    29625 | Robert De Niro  |   1 | 29625 |   53 |    1
    29726 | Al Pacino       |   2 | 29726 |   -1 |    0
(3 lignes)
```

4.6 CONCURRENCE D'ACCÈS

- Problèmes pouvant se poser :
  - UPDATE perdu
  - lecture non répétable
- Plusieurs solutions possibles
  - versionnement des lignes
  - `SELECT FOR UPDATE`
  - `SERIALIZABLE`

Plusieurs problèmes de concurrences d'accès peuvent se poser quand plusieurs transactions modifient les mêmes données en même temps.

Tout d'abord, des UPDATE peuvent être perdus, dans le cas où plusieurs transactions lisent la même ligne, puis la mettent à jour sans concertation. Par exemple, si la transaction 1 ouvre une transaction et effectue une lecture d'une ligne donnée :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004';
```

La transaction 2 effectue les mêmes traitements :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004';
```

Après un traitement applicatif, la transaction 1 met les données à jour pour noter l'augmentation de 5 % du salarié. La transaction est validée dans la foulée avec `COMMIT` :

```
UPDATE employes
   SET salaire = <valeur récupérée préalablement * 1.05>
 WHERE matricule = '00000004';
COMMIT;
```

Après un traitement applicatif, la transaction 2 met également les données à jour pour noter une augmentation exceptionnelle de 100 € :

```
UPDATE employes
   SET salaire = <valeur récupérée préalablement + 100>
 WHERE matricule = '00000004';
COMMIT;
```

Le salarié a normalement droit à son augmentation de 100 € ET l'augmentation de 5 %, l'augmentation de 5 % a été perdue car écrasée par la transaction n°2. Ce problème aurait pu être évité de trois façons différentes :

- en effectuant un UPDATE utilisant la valeur lue par l'ordre UPDATE,
- en verrouillant les données lues avec SELECT FOR UPDATE,
- en utilisant le niveau d'isolation SERIALIZABLE.

La première solution n'est pas toujours envisageable, il faut donc se tourner vers les deux autres solutions.

Le problème des lectures sales (*dirty reads*) ne peut pas se poser car PostgreSQL n'implémente pas le niveau d'isolation READ UNCOMMITTED. Si ce niveau d'isolation est sélectionné, PostgreSQL utilise alors le niveau READ COMMITTED.

4.6.1 SELECT FOR UPDATE

- SELECT FOR UPDATE
- Utilité :
  - « réserver » des lignes en vue de leur mise à jour
  - éviter les problèmes de concurrence d'accès

L'ordre SELECT FOR UPDATE permet de lire des lignes tout en les réservant en posant un verrou dessus en vue d'une future mise à jour. Le verrou permettra une lecture parallèle, mais mettra toute mise à jour en attente.

Reprenons l'exemple précédent et utilisons SELECT FOR UPDATE pour voir si le problème de concurrence d'accès peut être résolu.

session 1

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE;
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

La requête SELECT a retourné les données souhaitées.

session 2

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE;
```

La requête SELECT ne rend pas la main, elle est mise en attente.

session 3

Une troisième session effectue une lecture, sans poser de verrou explicite :

```
SELECT * FROM employes WHERE matricule = '00000004';
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

Le SELECT n'a pas été bloqué par la session 1. Seule la session 2 est bloquée car elle tente d'obtenir le même verrou.

session 1

L'application a effectué ses calculs et met à jour les données en appliquant l'augmentation de 5 % :

```
UPDATE employes
   SET salaire = 4725
 WHERE matricule = '00000004';
```

Les données sont vérifiées :

```
SELECT * FROM employes WHERE matricule = '00000004';
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4725.00
(1 row)
```

Enfin, la transaction est validée :

```
COMMIT;
```

session 2

La session 2 a rendu la main, le temps d'attente a été important pour réaliser ces calculs complexes :

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4725.00
(1 row)

Time: 128127,105 ms
```

Le salaire obtenu est bien le salaire mis à jour par la session 1. Sur cette base, l'application applique l'augmentation de 100 € :

```
UPDATE employes
   SET salaire = 4825.00
 WHERE matricule = '00000004';

SELECT * FROM employes WHERE matricule = '00000004';
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4825.00
```

La transaction est validée :

```
COMMIT;
```

Les deux transactions ont donc été effectuée de manière sérialisée, l'augmentation de 100 € ET l'augmentation de 5 % ont été accordées à Fantasio. En contre-partie, l'une des deux transactions concurrentes a été mise en attente afin de pouvoir sérialiser les transactions. Cela implique de penser les traitements en verrouillant les ressources auxquelles on souhaite accéder.

L'ordre SELECT FOR UPDATE dispose également d'une option NOWAIT qui permet d'annuler la transaction courante si un verrou ne pouvait être acquis. Si l'on reprend les premières étapes de l'exemple précédent :

session 1

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE NOWAIT;
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

Aucun verrou préalable n'avait été posé, la requête SELECT a retourné les données souhaitées.

session 2

On effectue la même chose sur la session n°2 :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE NOWAIT;
ERROR:  could not obtain lock on row in relation "employes"
```

Comme la session n°1 possède déjà un verrou sur la ligne qui nous intéresse, l'option NOWAIT sur le SELECT a annulé la transaction.

Il faut maintenant effectuer un ROLLBACK explicite pour pouvoir recommencer les traitements au risque d'obtenir le message suivant :

```
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
```

4.6.2 SKIP LOCKED

- SELECT FOR UPDATE SKIP LOCKED
  - PostgreSQL 9.5
- Utilité :
  - implémente des files d'attentes parallélisables

Une dernière fonctionnalité intéressante de SELECT FOR UPDATE, apparue avec PostgreSQL 9.5, permet de mettre en oeuvre différents workers qui consomment des données issues d'une table re-

présentant une file d'attente. Il s'agit de la clause SKIP LOCKED, dont le principe de fonctionnement est identique à son équivalent sous Oracle.

En prenant une table représentant la file d'attente suivante, peuplée avec des données générées :

```
CREATE TABLE test_skiplocked (id serial primary key, val text);
INSERT INTO test_skiplocked (val) SELECT md5(i::text)
FROM generate_series(1, 1000) i;
```

Une première transaction est ouverte et tente d'obtenir un verrou sur les 10 premières lignes :

```
BEGIN TRANSACTION;

SELECT *
  FROM test_skiplocked
 LIMIT 10
   FOR UPDATE SKIP LOCKED;

 id |               val
----+----------------------------------
  1 | c4ca4238a0b923820dcc509a6f75849b
  2 | c81e728d9d4c2f636f067f89cc14862c
  3 | eccbc87e4b5ce2fe28308fd9f2a7baf3
  4 | a87ff679a2f3e71d9181a67b7542122c
  5 | e4da3b7fbbce2345d7772b0674a318d5
  6 | 1679091c5a880faf6fb5e6087eb1b2dc
  7 | 8f14e45fceea167a5a36dedd4bea2543
  8 | c9f0f895fb98ab9159f51fd0297e236d
  9 | 45c48cce2e2d7fbdea1afc51c7c6ad26
 10 | d3d9446802a44259755d38e6d163e820
(10 rows)
```

Si on démarre une seconde transaction en parallèle, avec la première transaction toujours ouverte, le fait d'exécuter la requête SELECT FOR UPDATE sans la clause SKIP LOCKED aurait pour effet de la mettre en attente. L'ordre SELECT rendra la main lorsque la transaction #1 se terminera.

Avec la clause SKIP LOCKED, les 10 premières verrouillées par la transaction n°1 seront passées et ce sont les 10 lignes suivantes qui seront verrouillées et retournées par l'ordre SELECT :

```
BEGIN TRANSACTION;

SELECT *
  FROM test_skiplocked
 LIMIT 10
   FOR UPDATE SKIP LOCKED;

 id |               val
----+----------------------------------
 11 | 6512bd43d9caa6e02c990b0a82652dca
 12 | c20ad4d76fe97759aa27a0c99bff6710
 13 | c51ce410c124a10e0db5e4b97fc2af39
 14 | aab3238922bcc25a6f606eb525ffdc56
 15 | 9bf31c7ff062936a96d3c8bd1f8f2ff3
 16 | c74d97b01eae257e44aa9d5bade97baf
 17 | 70efdf2ec9b086079795c442636b55fb
 18 | 6f4922f45568161a8cdf4ad2299f6d23
```

```
 19 | 1f0e3dad99908345f7439f8ffabdffc4
 20 | 98f13708210194c475687be6106a3b84
(10 rows)
```

Ensuite, la première transaction supprime les lignes verrouillées et valide la transaction :

```
DELETE FROM test_skiplocked
 WHERE id IN (...);
COMMIT;
```

De même pour la seconde transaction, qui aura traité d'autres lignes en parallèle de la transaction #1.

4.7 SERIALIZABLE SNAPSHOT ISOLATION

SSI : Serializable Snapshot Isolation (9.1+)

- Chaque transaction est seule sur la base
- Si on ne peut maintenir l'illusion
  - une des transactions en cours est annulée
- Sans blocage
- On doit être capable de rejouer la transaction
- Toutes les transactions impliquées doivent être `serializable`
- `default_transaction_isolation=serializable` dans la configuration

PostgreSQL fournit depuis la version 9.1 un mode d'isolation appelé `SERIALIZABLE`. Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base. Dès que cette garantie ne peut plus être apportée, une des transactions est annulée.

Toute transaction non déclarée comme `SERIALIZABLE` peut en théorie s'exécuter n'importe quand, ce qui rend inutile le mode `SERIALIZABLE` sur les autres. C'est donc un mode qui doit être mis en place globalement.

Voici un exemple.

Dans cet exemple, il y a des enregistrements avec une colonne couleur contenant 'blanc' ou 'noir'. Deux utilisateurs essayent simultanément de convertir tous les enregistrements vers une couleur unique, mais chacun dans une direction opposée. Un veut passer tous les blancs en noir, et l'autre tous les noirs en blanc.

L'exemple peut être mis en place avec ces ordres :

```
create table points
  (
    id int not null primary key,
    couleur text not null
  );
insert into points
  with x(id) as (select generate_series(1,10))
  select id, case when id % 2 = 1 then 'noir'
    else 'blanc' end from x;
```

Session 1 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'noir'
where couleur = 'blanc';
```

Session 2 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'blanc'
where couleur = 'noir';
```

À ce moment, une des deux transaction est condamnée à mourir.

Session 2 :

```
commit;
```

Le premier à valider gagne.

```
select * from points order by id;
```

```
 id | couleur
----+-------
  1 | blanc
  2 | blanc
  3 | blanc
  4 | blanc
  5 | blanc
  6 | blanc
  7 | blanc
  8 | blanc
  9 | blanc
 10 | blanc
(10 rows)
```

Session 1 : Celle-ci s'est exécutée comme si elle était seule.

```
commit;
```

```
ERROR:  could not serialize access
        due to read/write dependencies
        among transactions
DETAIL:  Cancelled on identification
         as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Une erreur de sérialisation. On annule et on réessaye.

```
rollback;
begin;
update points set couleur = 'noir'
  where couleur = 'blanc';
commit;
```

Il n'y a pas de transaction concurrente pour gêner.

```
select * from points order by id;
```

```
 id | couleur
----+-------
  1 | noir
  2 | noir
  3 | noir
```

```
  4 | noir
  5 | noir
  6 | noir
  7 | noir
  8 | noir
  9 | noir
 10 | noir
(10 rows)
```

La transaction s'est exécutée seule, après l'autre.

Le mode SERIALIZABLE permet de s'affranchir des SELECT FOR UPDATE qu'on écrit habituellement, dans les applications en mode READ COMMITTED. Toutefois, il fait bien plus que ça, puisqu'il réalise du verrouillage de prédicats. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation. Il permet aussi de gérer les problèmes ci-dessus avec plus de deux sessions.

Pour des exemples plus complets, le mieux est de consulter la documentation officielle[8].

[8] https://wiki.postgresql.org/wiki/SSI/fr

4.8 CONCLUSION

- SQL est un langage très riche
- Connaître les nouveautés des versions de la norme depuis 20 ans permet de
  - gagner énormément de temps de développemment
  - mais aussi de performance

4.9 TRAVAUX PRATIQUES

Jointure latérale

Cette série de question utilise la base de TP **magasin**. La base **magasin** peut être téléchargée depuis https://dali.bo/tp_magasin (dump de 96 Mo, pour 667 Mo sur le disque au final). Elle s'importe de manière très classique (une erreur sur le schéma **public** déjà présent est normale), ici dans une base nommée aussi `magasin` :

```
curl -L https://dali.bo/tp_magasin -o magasin.dump
pg_restore -d magasin  magasin.dump
```

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

> Afficher les 10 derniers articles commandés.

> Pour chacune des 10 dernières commandes passées, afficher le premier article commandé.

CTE récursive

La table `genealogie` peut être téléchargée depuis https://dali.bo/tp_genealogie et restaurée à l'aide de `pg_restore` :

```
createdb genealogie
pg_restore -O -d genealogie genealogie.dump
```

Voici la description de la table `genealogie` qui sera utilisée :

```
\d genealogie
                          Table "public.genealogie"
     Column     |  Type   |                Modifiers
----------------+---------+-----------------------------------------
 id             | integer | not null default                       +
                |         | nextval('genealogie_id_seq'::regclass)
 nom            | text    |
 prenom         | text    |
 date_naissance | date    |
 pere           | integer |
 mere           | integer |
Indexes:
    "genealogie_pkey" PRIMARY KEY, btree (id)
```

> À partir de la table `genealogie`, déterminer qui sont les descendants de Fernand DEVAUX.

> À l'inverse, déterminer qui sont les ancêtres de Adèle TAILLANDIER

Réseau social

La table `socialnet` peut être téléchargée depuis https://dali.bo/tp_socialnet et restaurée à l'aide de `pg_restore` :

```
createdb socialnet
pg_restore -O -d socialnet socialnet.dump
```

Cet exercice est assez similaire au précédent et propose de manipuler des arborescences.

Les tableaux et la fonction `unnest()` peuvent être utiles pour résoudre plus facilement ce problème.

La table `personnes` contient la liste de toutes les personnes d'un réseau social.

```
                        Table "public.personnes"
 Column |  Type   |                       Modifiers
--------+---------+-------------------------------------------------------
 id     | integer | not null default nextval('personnes_id_seq'::regclass)
 nom    | text    | not null
 prenom | text    | not null
Indexes:
    "personnes_pkey" PRIMARY KEY, btree (id)
```

La table `relation` contient les connexions entre ces personnes.

```
   Table "public.relation"
 Column |  Type   | Modifiers
--------+---------+-----------
 gauche | integer | not null
 droite | integer | not null
Indexes:
    "relation_droite_idx" btree (droite)
    "relation_gauche_idx" btree (gauche)
```

Déterminer le niveau de connexions entre Sadry Luettgen et Yelsi Kerluke et afficher le chemin de relation le plus court qui permet de les connecter ensemble.

Dépendance de vues

Les dépendances entre objets est un problème classique dans les bases de données :

- dans quel ordre charger des tables selon les clés étrangères ?
- dans quel ordre recréer des vues ?
- etc.

Le catalogue de PostgreSQL décrit l'ensemble des objets de la base de données. Deux tables vont nous intéresser pour mener à bien cet exercice :

- `pg_depend` liste les dépendances entre objets
- `pg_rewrite` stocke les définitions des règles de réécritures des vues (RULES)
- `pg_class` liste les objets que l'on peut interroger comme une table, hormis les fonctions retournant des ensembles

La définition d'une vue peut être obtenue à l'aide de la fonction `pg_get_viewdef`.

Pour plus d'informations sur ces tables, se référer à la documentation :

- Catalogue pg_depend[9]
- Catalogue pg_rewrite[10]
- Catalogue pg_class[11]
- Fonction d'information du catalogue système[12]

L'objectif de se TP consiste à récupérer l'ordre de suppression et de recréation des vues de la base `brno2015` en fonction du niveau de dépendances entre chacune des vues. Brno est une ville de Tchéquie, dans la région de Moravie-du-Sud. Le circuit Brno-Masaryk est situé au nord-ouest de la ville. Le Grand Prix moto de Tchéquie s'y déroule chaque année.

La table `brno2015` peut être téléchargée depuis https://dali.bo/tp_brno2015 et restaurée à l'aide de `pg_restore` :

```
createdb brno2015
pg_restore -O -d brno2015 brno2015.dump
```

> Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de recréation des vues.

[9] https://www.postgresql.org/docs/current/static/catalog-pg-depend.html
[10] https://www.postgresql.org/docs/current/static/catalog-pg-rewrite.html
[11] https://www.postgresql.org/docs/current/static/catalog-pg-class.html
[12] https://www.postgresql.org/docs/current/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE

4.10 TRAVAUX PRATIQUES (SOLUTIONS)

Jointure latérale

> Afficher les 10 derniers articles commandés.

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma **magasin** :

```
SET search_path = magasin;
```

On commence par afficher les 10 dernières commandes :

```
SELECT *
  FROM commandes
 ORDER BY numero_commande DESC
 LIMIT 10;
```

Une simple jointure nous permet de retrouver les 10 derniers articles commandés :

```
SELECT lc.produit_id, p.nom
  FROM commandes c
  JOIN lignes_commandes lc
    ON (c.numero_commande = lc.numero_commande)
  JOIN produits p
    ON (lc.produit_id = p.produit_id)
 ORDER BY c.numero_commande DESC, numero_ligne_commande DESC
 LIMIT 10;
```

> Pour chacune des 10 dernières commandes passées, afficher le premier article commandé.

La requête précédente peut être dérivée pour répondre à la question demandée. Ici, pour chacune des dix dernières commandes, nous voulons récupérer le nom du dernier article commandé, ce qui sera transcrit sous la forme d'une jointure latérale :

```
SELECT numero_commande, produit_id, nom
  FROM commandes c,
  LATERAL (SELECT p.produit_id, p.nom
             FROM lignes_commandes lc
             JOIN produits p
               ON (lc.produit_id = p.produit_id)
            WHERE (c.numero_commande = lc.numero_commande)
            ORDER BY numero_ligne_commande ASC
            LIMIT 1
 ) premier_article_par_commande
 ORDER BY c.numero_commande DESC
 LIMIT 10;
```

CTE récursive

Cet exercice propose de manipuler des données généalogiques.

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma **genealogie** :

```
SET search_path = genealogie;
```

Voici la description de la table `genealogie` qui sera utilisée :

```
\d genealogie
                          Table "public.genealogie"
     Column     |  Type   |                Modifiers
----------------+---------+-----------------------------------------
 id             | integer | not null default                       +
                |         | nextval('genealogie_id_seq'::regclass)
 nom            | text    |
 prenom         | text    |
 date_naissance | date    |
 pere           | integer |
 mere           | integer |
Indexes:
    "genealogie_pkey" PRIMARY KEY, btree (id)
```

> À partir de la table genealogie, déterminer qui sont les descendants de Fernand DEVAUX.

```
WITH RECURSIVE arbre_genealogique AS (
SELECT id, nom, prenom, date_naissance, pere, mere
  FROM genealogie
 WHERE nom = 'DEVAUX'
   AND prenom = 'Fernand'
UNION ALL
SELECT g.*
  FROM arbre_genealogique ancetre
  JOIN genealogie g
    ON (g.pere = ancetre.id OR g.mere = ancetre.id)
)
SELECT id, nom, prenom, date_naissance
  FROM arbre_genealogique;
```

> À l'inverse, déterminer qui sont les ancètres de Adèle TAILLANDIER

```
WITH RECURSIVE arbre_genealogique AS (
SELECT id, nom, prenom, date_naissance, pere, mere
  FROM genealogie
 WHERE nom = 'TAILLANDIER'
   AND prenom = 'Adèle'
UNION ALL
SELECT ancetre.id, ancetre.nom, ancetre.prenom, ancetre.date_naissance,
       ancetre.pere, ancetre.mere
  FROM arbre_genealogique descendant
  JOIN genealogie ancetre
    ON (descendant.pere = ancetre.id OR descendant.mere = ancetre.id)
)
SELECT id, nom, prenom, date_naissance
  FROM arbre_genealogique;
```

Réseau social

Cet exercice est assez similaire au précédent.

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma **socialnet** :

```
SET search_path = socialnet;
```

Les tableaux et la fonction unnest peuvent être utiles pour résoudre plus facilement ce problème

La table `personnes` contient la liste de toutes les personnes d'un réseau social.

```
                        Table "public.personnes"
 Column |  Type   |                        Modifiers
--------+---------+-----------------------------------------------------------
 id     | integer | not null default nextval('personnes_id_seq'::regclass)
 nom    | text    | not null
 prenom | text    | not null
Indexes:
    "personnes_pkey" PRIMARY KEY, btree (id)
```

La table `relation` contient les connexions entre ces personnes.

```
   Table "public.relation"
 Column |  Type   | Modifiers
--------+---------+-----------
 gauche | integer | not null
 droite | integer | not null
Indexes:
    "relation_droite_idx" btree (droite)
    "relation_gauche_idx" btree (gauche)
```

> Déterminer le niveau de connexions entre Sadry Luettgen et Yelsi Kerluke et afficher le chemin de relation le plus court qui permet de les connecter ensemble.

La requête suivante permet de répondre à cette question :

```
WITH RECURSIVE connexions AS (
SELECT gauche, droite, ARRAY[gauche] AS personnes_connectees,0::integer AS level
  FROM relation
 WHERE gauche = 1
UNION ALL
SELECT p.gauche, p.droite, personnes_connectees || p.gauche, level + 1 AS level
  FROM connexions c
  JOIN relation p ON (c.droite = p.gauche)
 WHERE level < 4
   AND p.gauche <> ANY (personnes_connectees)
), plus_courte_connexion AS (
SELECT *
  FROM connexions
 WHERE gauche = (
    SELECT id FROM personnes WHERE nom = 'Kerluke' AND prenom = 'Yelsi'
 )
 ORDER BY level ASC
 LIMIT 1
)
```

```
SELECT list.id, p.nom, p.prenom, list.level - 1 AS level
  FROM plus_courte_connexion,
       unnest(personnes_connectees) WITH ORDINALITY AS list(id, level)
  JOIN personnes p on (list.id = p.id)
 ORDER BY list.level;
```

Cet exemple fonctionne sur une faible volumétrie, mais les limites des bases relationnelles sont rapidement atteintes sur de telles requêtes.
Une solution consisterait à implémenter un algorithme de parcours de graphe avec pgRouting[13], mais cela nécessitera de présenter les données sous une forme particulière.
Pour les problématiques de traitement de graphe, notamment sur de grosses volumétries, une base de données orientée graphe comme Neo4J sera probablement plus adaptée.

Dépendance de vues

Les dépendances entre objets est un problème classique dans les bases de données :

- dans quel ordre charger des tables selon les clés étrangères ?
- dans quel ordre recréer des vues ?
- etc.

Le catalogue de PostgreSQL décrit l'ensemble des objets de la base de données. Deux tables vont nous intéresser pour mener à bien cet exercice :

- `pg_depend` liste les dépendances entre objets
- `pg_rewrite` stocke les définitions des règles de réécritures des vues (RULES)
- `pg_class` liste les objets que l'on peut interroger comme une table, hormis les fonctions retournant des ensembles

La définition d'une vue peut être obtenue à l'aide de la fonction `pg_get_viewdef`.

Pour plus d'informations sur ces tables, se référer à la documentation :

- Catalogue pg_depend[14]
- Catalogue pg_rewrite[15]
- Catalogue pg_class[16]
- Fonction d'information du catalogue système[17]

Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de recréation des vues.

[14] http://www.postgresql.org/docs/current/static/catalog-pg-depend.html
[15] http://www.postgresql.org/docs/current/static/catalog-pg-rewrite.html
[16] http://www.postgresql.org/docs/current/static/catalog-pg-class.html
[17] http://www.postgresql.org/docs/current/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma **brno2015** :

```
SET search_path = brno2015;
```

Si la jointure entre `pg_depend` et `pg_rewrite` est possible pour l'objet de départ, alors il s'agit probablement d'une vue. En discriminant sur les objets qui référencent la vue `pilotes_brno`, nous arrivons à la requête de départ suivante :

```
SELECT DISTINCT pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass;
```

La présence de doublons nous oblige à utiliser la clause DISTINCT.

Nous pouvons donc créer un graphe de dépendances à partir de cette requête de départ, transformée en requête récursive :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
)
SELECT * FROM graph;
```

Il faut maintenant résoudre les OID pour déterminer les noms des vues et leur schéma. Pour cela, nous ajoutons une vue `resolved` telle que :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
),
resolved AS (
  SELECT n.nspname AS dependent_schema, d.relname as dependent,
         n2.nspname AS dependee_schema, d2.relname as dependee,
         depth
  FROM graph
  JOIN pg_class d ON d.oid = objid
  JOIN pg_namespace n ON d.relnamespace = n.oid
```

```
  JOIN pg_class d2 ON d2.oid = refobjid
  JOIN pg_namespace n2 ON d2.relnamespace = n2.oid
)
SELECT * FROM resolved;
```

Nous pouvons maintenant présenter les ordres de suppression et de recréation des vues, dans le bon ordre. Les vues doivent être supprimées selon le numéro d'ordre décroissant et recrées selon le numéro d'ordre croissant :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
),
resolved AS (
  SELECT n.nspname AS dependent_schema, d.relname as dependent,
         n2.nspname AS dependee_schema, d2.relname as dependee,
         d.oid as dependent_oid,
         depth
  FROM graph
  JOIN pg_class d ON d.oid = objid
  JOIN pg_namespace n ON d.relnamespace = n.oid
  JOIN pg_class d2 ON d2.oid = refobjid
  JOIN pg_namespace n2 ON d2.relnamespace = n2.oid
)
(SELECT 'DROP VIEW ' || dependent_schema || '.' || dependent || ';'
   FROM resolved
  GROUP BY dependent_schema, dependent
  ORDER BY max(depth) DESC)
UNION ALL
(SELECT 'CREATE OR REPLACE VIEW ' || dependent_schema || '.' || dependent ||
        ' AS ' || pg_get_viewdef(dependent_oid)
   FROM resolved
  GROUP BY dependent_schema, dependent, dependent_oid
  ORDER BY max(depth));
```

5/ Types de base

- PostgreSQL offre un système de typage complet
  - types standards
  - types avancés propres à PostgreSQL

5.0.1 Préambule

- SQL possède un typage fort
  - le type employé décrit la donnée manipulée
  - garantit l'intégrité des données
  - primordial au niveau fonctionnel
  - garantit les performances

5.0.2 Menu

- Qu'est-ce qu'un type ?
- Les types SQL standards
  - numériques
  - temporels
  - textuels et binaires
- Les types avancés de PostgreSQL

5.0.3 Objectifs

- Comprendre le système de typage de PostgreSQL
- Savoir choisir le type adapté à une donnée
- Être capable d'utiliser les types avancés à bon escient

5.1 LES TYPES DE DONNÉES

- Qu'est-ce qu'un type ?
- Représentation physique
- Impacts sur l'intégrité
- Impacts fonctionnels

5.1.1 Qu'est-ce qu'un type ?

- Un type définit :
  - les valeurs que peut prendre une donnée
  - les opérateurs applicables à cette donnée

5.1.2 Impact sur les performances

- Choisir le bon type pour :
  - optimiser les performances
  - optimiser le stockage

5.1.3 Impacts sur l'intégrité

- Le bon type de données garantit l'intégrité des données :
  - la bonne représentation
  - le bon intervalle de valeur

Le choix du type employé pour stocker une donnée est primordial pour garantir l'intégrité des données.

Par exemple, sur une base de données mal conçue, il peut arriver que les dates soient stockées sous la forme d'une chaîne de caractère. Ainsi, une date malformée ou invalide pourra être enregistrée dans la base de données, passant outre les mécanismes de contrôle d'intégrité de la base de données. Si une date est stockée dans une colonne de type `date`, alors ces problèmes ne se posent pas :

```
postgres=# create table test_date (dt date);
CREATE TABLE

postgres=# insert into test_date values ('2015-0717');
ERROR:  invalid input syntax for type date: "2015-0717"
LINE 1: insert into test_date values ('2015-0717');
                                      ^
postgres=# insert into test_date values ('2015-02-30');
ERROR:  date/time field value out of range: "2015-02-30"
LINE 1: insert into test_date values ('2015-02-30');

postgres=# insert into test_date values ('2015-07-17');
INSERT 0 1
```

5.1.4 Impacts fonctionnels

- Un type de données offre des opérateurs spécifiques :
  - comparaison
  - manipulation
- Exemple: une date est-elle comprise entre deux dates données ?

5.2 TYPES NUMÉRIQUES

- Entiers
- Flottants
- Précision fixée

5.2.1 Types numériques : entiers

- 3 types entiers :
  - `smallint` : 2 octets
  - `integer` : 4 octets
  - `bigint` : 8 octets
- Valeur exacte
- Signé
- Utilisation :
  - véritable entier
  - clé technique

5.2.2 Types numériques : flottants

- 2 types flottants :
  - `real/float4`
  - `double precision/float8`
- Données numériques « floues »
  - valeurs non exactes
- Utilisation :
  - stockage des données issues de capteurs

5.2.3 Types numériques : numeric

- 1 type
  - `numeric(.., ..)`
- Type exact
  - mais calcul lent
- Précision choisie : totale, partie décimale
- Utilisation :
  - données financières
  - calculs exacts
- Déconseillé pour :
  - clés primaires
  - données non exactes (ex : résultats de capteurs)

5.2.4 Opérations sur les numériques

- Indexable : >, >=, =, <=,<
- +, −, /, *, modulo (%), puissance (^)
- Pour les entiers :
  - AND, OR, XOR (&, |, #)
  - décalage de bits (*shifting*): >>, <<
- Attention aux conversions (*casts*) / promotions !

Tout les types numériques sont indexables avec des indexes standards btree, permettant la recherche avec les opérateurs d'égalité / inégalité. Pour les entiers, il est possible de réaliser des opérations bit-à-bit :

```
postgres=# select 2 | 4;
 ?column?
----------
        6
(1 ligne)
```

```
postgres=# select 7 & 3;
 ?column?
----------
        3
(1 ligne)
```

Il faut toutefois être vigilant face aux opérations de cast implicites et de promotions des types numériques. Par exemple, les deux requêtes suivantes ramèneront le même résultat, mais l'une sera capable d'utiliser un éventuel index sur `id`, l'autre non :

```
postgres=# explain select * from t1 where id = 10::int4;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=4.67..52.52 rows=50 width=4)
   Recheck Cond: (id = 10)
   ->  Bitmap Index Scan on t1_id_idx  (cost=0.00..4.66 rows=50 width=0)
         Index Cond: (id = 10)
(4 lignes)

postgres=# explain select * from t1 where id = 10::numeric;
                      QUERY PLAN
-------------------------------------------------
 Seq Scan on t1  (cost=0.00..195.00 rows=50 width=4)
   Filter: ((id)::numeric = 10::numeric)
(2 lignes)
```

Cela peut paraître contre-intuitif, mais le cast est réalisé dans ce sens pour ne pas perdre d'information. Par exemple, si la valeur numérique cherchée n'est pas un entier. Il faut donc faire spécialement attention aux types utilisés côté applicatif. Avec un ORM tel Hibernate, il peut être tentant de faire correspondre un `BigInt` à un numeric côté SQL, ce qui engendrera des casts implicites, et potentiellement des indexes non utilisés.

5.2.5 Choix d'un type numérique

- `integer` ou `biginteger` :
  - identifiants (clés primaires et autre)
  - nombres entiers
- `numeric` :
  - valeurs décimales exactes
  - performance non critique
- `float`, `real` :
  - valeurs flottantes, non exactes
  - performance demandée : `SUM()`, `AVG()`, etc.

Pour les identifiants, il est préférable d'utiliser des entiers ou grands entiers. En effet, il n'est pas nécessaire de s'encombrer du bagage technique et de la pénalité en performance dû à l'utilisation de `numeric`. Contrairement à d'autres SGBD, PostgreSQL ne transforme pas un `numeric` sans partie décimale en entier, et celui-ci souffre donc des performances inhérentes au type `numeric`.

De même, lorsque les valeurs sont entières, il faut utiliser le type adéquat.

Pour les nombres décimaux, lorsque la performance n'est pas critique, préférer le type `numeric` : il est beaucoup plus simple de raisonner sur ceux-ci et leur précision que de garder à l'esprit les subtilités du standard IEEE 754 définissant les opérations sur les flottants. Dans le cas de données décimales nécessitant une précision exacte, il est impératif d'utiliser le type `numeric`.

Les nombres flottants (`float` et `real`) ne devraient être utilisés que lorsque les implications en terme de perte de précision sont intégrées, et que la performance d'un type `numeric` devient gênante. En pratique, cela est généralement le cas lors d'opérations d'agrégations.

Pour bien montrer les subtilités des types `float`, et les risques auquels ils nous exposent, considérons l'exemple suivant, en créant une table contenant 25000 fois la valeur `0.4`, stockée soit en `float` soit en `numeric` :

```
postgres=# create table t_float as (
    select 0.04::float as cf,
           0.04::numeric as cn
    from generate_series(1, 25000)
);
SELECT 25000
postgres=# select sum(cn), sum(cf) from t_float ;
   sum   |       sum
---------+-----------------
 1000.00 | 999.99999999967
(1 ligne)
```

Si l'on considère la performance de ces opérations, on remarque des temps d'exécution bien différents :

```
postgres=# select sum(cn) from t_float ;
   sum
---------
 1000.00
(1 ligne)

Temps : 10,611 ms
postgres=# select sum(cf) from t_float ;
       sum
-----------------
 999.99999999967
(1 ligne)

Temps : 6,434 ms
```

Pour aller (beaucoup) plus loin, le document suivant détaille le comportement des flottants selon le standard IEEE : https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

5.3 TYPES TEMPORELS

- Date
- Date & heure
  - …avec ou sans fuseau

5.3.1 Types temporels : date

- `date`
  - représente une date, sans heure
  - affichage format ISO : `YYYY-MM-DD`
- Utilisation :
  - stockage d'une date lorsque la composante heure n'est pas utilisée
- Cas déconseillés :
  - stockage d'une date lorsque la composante heure est utilisée

```
# SELECT now()::date ;

    now
------------
 2019-11-13
```

5.3.2 Types temporels : time

- `time`
  - représente une heure sans date
  - affichage format ISO `HH24:MI:SS`
- Peu de cas d'utilisation
- À éviter :
  - stockage d'une date et de la composante heure dans deux colonnes

```
# SELECT now()::time ;

        now
-----------------
 15:19:39.947677
```

5.3.3 Types temporels : timestamp

- `timestamp` (`without time zone` !)
  - représente une date et une heure
  - fuseau horaire non précisé
- Utilisation :
  - stockage d'une date et d'une heure

```
# SELECT now()::timestamp ;

             now
----------------------------
 2019-11-13 15:20:54.222233
```

Le nom réel est `timestamp without time zone`. Comme on va le voir, il faut lui préférer le type `timestamptz`.

5.3.4 Types temporels : timestamp with time zone

- `timestamp with time zone` = `timestamptz`
  - représente une date et une heure
  - fuseau horaire inclus
  - affichage : `2019-11-13 15:33:00.824096+01`
- Utilisation :
  - stockage d'une date et d'une heure, cadre mondial
  - à préférer à `timestamp without time zone`

Ces deux exemples ont été exécutés à quelques secondes d'intervalle sur des instances en France (heure d'hiver) et au Brésil :

```
# SHOW timezone;

   TimeZone
--------------
 Europe/Paris

# SELECT  now() ;
              now
-------------------------------
 2019-11-13 15:32:09.615455+01

# SHOW timezone;

  TimeZone
-------------
 Brazil/West
(1 ligne)

# SELECT now() ;
              now
-------------------------------
 2019-11-13 10:32:39.536972-04

# SET timezone to 'Europe/Paris' ;

# SELECT now() ;
              now
-------------------------------
 2019-11-13 15:33:00.824096+01
```

On préférera presque tout le temps le type `timestamptz` à `timestamp` (sans fuseau horaire), ne serait-ce qu'à cause des heures d'été et d'hiver. Les deux types occupent 8 octets.

5.3.5 Types temporels : interval

- `interval`
  - représente une durée
- Utilisation :
  - exprimer une durée
  - dans une requête, pour modifier une date/heure existante

5.3.6 Choix d'un type temporel

- Préférer les types avec *timezone*
  - toujours plus simple à gérer au début qu'à la fin
- Considérer les types `range` pour tout couple « début / fin »
- Utiliser `interval` /`generate_series`

De manière générale, il est beaucoup plus simple de gérer des dates avec timezone côté base. En effet, dans le cas où une seule timezone est gérée, les clients ne verront pas la différence. Si en revanche les besoins évoluent, il sera beaucoup plus simple de gérer les différentes timezones à ce moment là.

Les points suivants concernent plus de la modélisation que des types de données à proprement parler, mais il est important de considérer les types ranges dès lors que l'on souhaite stocker un couple « date de début / date de fin ». Nous aurons l'occasion de revenir sur ces types dans la suite de ce module.

Enfin, une problématique assez commune consiste à vouloir effectuer des jointures contre une table de dates de références. Une (mauvaise) solution à ce problème consiste à stocker ces dates dans une table. Il est beaucoup plus avantageux en terme de maintenance de ne pas stocker ces dates, mais de les générer à la volée. Par exemple, pour générer tous les jours de janvier 2015 :

```
postgres=# select * from generate_series(
    '2015-01-01',
    '2015-01-31',
    '1 day'::interval
);
    generate_series
------------------------
 2015-01-01 00:00:00+01
```

```
2015-01-02 00:00:00+01
2015-01-03 00:00:00+01
2015-01-04 00:00:00+01
2015-01-05 00:00:00+01
2015-01-06 00:00:00+01
2015-01-07 00:00:00+01
2015-01-08 00:00:00+01
2015-01-09 00:00:00+01
2015-01-10 00:00:00+01
2015-01-11 00:00:00+01
2015-01-12 00:00:00+01
2015-01-13 00:00:00+01
2015-01-14 00:00:00+01
2015-01-15 00:00:00+01
2015-01-16 00:00:00+01
2015-01-17 00:00:00+01
2015-01-18 00:00:00+01
2015-01-19 00:00:00+01
2015-01-20 00:00:00+01
2015-01-21 00:00:00+01
2015-01-22 00:00:00+01
2015-01-23 00:00:00+01
2015-01-24 00:00:00+01
2015-01-25 00:00:00+01
2015-01-26 00:00:00+01
2015-01-27 00:00:00+01
2015-01-28 00:00:00+01
2015-01-29 00:00:00+01
2015-01-30 00:00:00+01
2015-01-31 00:00:00+01
```

5.4 TYPES CHAÎNES

- Texte à longueur variable
- Binaires

En général on choisira une chaîne de longueur variable. Nous ne parlerons pas ici du type `char` (à tailel fixe), d'utilisation très restreinte.

5.4.1 Types chaînes : caractères

- `varchar(_n_),text`
- Représentent une chaîne de caractères
- Valident l'encodage
- Valident la longueur maximale de la chaîne (contrainte !)
- Utilisation :
  - stocker des chaînes de caractères non binaires

5.4.2 Types chaînes : binaires

- `bytea`
- Stockage de données binaires
  - encodage en hexadécimal ou séquence d'échappement
- Utilisation :
  - stockage de courtes données binaires
- Cas déconseillés :
  - stockage de fichiers binaires

Le type `bytea` permet de stocker des données binaires dans une base de données PostgreSQL.

5.4.3 Quel type choisir ?

- `varchar` (sans limite) ou `text` (non standard)
- Implémenter la limite avec une contrainte
  - plus simple à modifier

```
CREATE TABLE t1 (c1 varchar CHECK (length(c1) < 10))
```

En règle générale, il est recommandé d'utiliser un champ de type `varchar` tout court, et de vérifier la longueur au niveau d'une contrainte. En effet, il sera plus simple de modifier celle-ci par la suite, en modifiant uniquement la contrainte. De plus, la contrainte permet plus de possibilités, comme par exemple d'imposer une longueur minimale.

5.4.4 Collation

- L'ordre de tri dépend des langues & de conventions variables
- Collation par colonne / index / requête
- `SELECT * FROM mots ORDER BY t COLLATE "C" ;`
- `CREATE TABLE messages ( id int, fr TEXT COLLATE "fr_FR.utf8", de TEXT COLLATE "de_DE.utf8" );`

L'ordre de tri des chaînes de caractère (« collation ») peut varier suivant le contenu d'une colonne. Rien que parmi les langues européennes, il existe des spécificités propres à chacune, et même à différents pays pour une même langue. Si l'ordre des lettres est une convention courante, il existe de nombreuses variations propres à chacune (comme é, à, æ, ö, ß, å, ñ…), avec des règles de tri propres. Certaines lettres peuvent être assimilées à une combinaison d'autres lettres. De plus, la place relative des majuscules, celles des chiffres, ou des caractères non alphanumérique est une pure affaire de convention.

La collation dépend de l'encodage (la manière de stocker les caractères), de nos jours généralement UTF8[1] (standard Unicode). PostgreSQL utilise par défaut UTF8 et il est chaudement conseillé de ne pas changer cela.

[1] https://fr.wikipedia.org/wiki/UTF-8

La collation par défaut dans une base est définie à sa création, et est visible avec `\l` (ci-dessous pour une installation en français). Le type de caractères est généralement identique.

```
# \l
                      Liste des bases de données
Nom        | Propriétaire | Encodage | Collationnement | Type caract. |...
-----------+--------------+----------+-----------------+--------------+
 pgbench   | pgbench      | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 postgres  | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 template0 | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  | ...
 template1 | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  | ...
```

Parmi les collations que l'on peut rencontrer, il y a par exemple `en_US.UTF-8` (la collation par défaut de beaucoup d'installations), ou C, basée sur les caractères ASCII et les valeurs des octets. De vieilles installations peuvent encore contenir `fr_FR.iso885915@euro`.

Si le tri par défaut ne convient pas, on peut le changer à la volée dans la requête SQL, au besoin après avoir créé la collation.

Exemple avec du français :

```
CREATE TABLE mots (t text) ;

INSERT INTO mots
VALUES ('A'),('a'),('aa'),('z'),('ä'),('å'),('Å'),('aa'),('æ'),('ae'),('af'), ('ß'),
 ↪  ('ss') ;

SELECT * FROM mots ORDER BY t ; -- sous-entendu, ordre par défaut en français ici

 t
 a
 A
 å
 Å
 ä
 aa
 aa
 ae
 æ
 af
 ss
 ß
 z
```

Noter que les caractères « æ » et « ß » sont correctement assimilés à « ae » et « ss ». (Ce serait aussi le cas avec `en_US.utf8` ou `de_DE.utf8`).

Avec la collation C, l'ordre est plus basique, soit celui des codes UTF-8 :

```
SELECT * FROM mots ORDER BY t COLLATE "C" ;

 t
 A
 a
 aa
 aa
```

```
ae
af
ss
z
Å
ß
ä
å
æ
```

Un intérêt de la collation C est qu'elle est plus simple et se repose sur la glibc du système, ce qui lui permet d'être souvent plus rapide qu'une des collations ci-dessus. Il suffit donc parfois de remplacer `ORDER BY champ_texte` par `ORDER BY champ_text COLLATE "C"`, à condition bien sûr que l'ordre ASCII convienne.

Il est possible d'indiquer dans la définition de chaque colonne quelle doit être sa collation par défaut :

Pour du danois :

```
-- La collation doit exister sur le système d'exploitation
CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

ALTER TABLE mots ALTER COLUMN t TYPE text COLLATE "da_DK" ;

SELECT * FROM mots ORDER BY t ; -- ordre danois
```

```
t
A
a
ae
af
ss
ß
z
æ
ä
Å
å
aa
```

Dans cette langue, les majuscules viennent traditionnellement avant les minuscules, et « å» et « aa » viennent après le « z ».

Avec une collation précisée dans la requête, un index peut ne pas être utilisable. En effet, par défaut, il est trié sur disque dans l'ordre de la collation de la colonne. Un index peut cependant se voir affecter une collation différente de celle de la colonne, par exemple pour un affichage ou une interrogation dans plusieurs langues :

```
CREATE INDEX ON mots (t);                       -- collation par défaut de la colonne
CREATE INDEX ON mots (t COLLATE "de_DE.utf8"); -- tri allemand
```

La collation n'est pas qu'une question d'affichage. Le tri joue aussi dans la sélection quand il y a des inégalités, et le français et le danois revoient ici des résultats différents :

```
SELECT * FROM mots WHERE t > 'z' COLLATE "fr_FR";

 t
(0 ligne)

SELECT * FROM mots WHERE t > 'z' COLLATE "da_DK";

 t
 aa
 ä
 å
 Å
 aa
 æ
(6 lignes)
```

5.4.5 Collation & sources

Source des collations :

- le système : installations séparées nécessaires, différences entre OS
- (>= v 10) : librairie externe ICU

```
CREATE COLLATION danois (provider = icu, locale = 'da-x-icu')
;
```

Des collations comme `en_US.UTF-8` ou `fr_FR.UTF-8` sont dépendantes des locales installées sur la machine. Cela implique qu'elles peuvent subtilement différer entre deux systèmes, même entre deux versions d'un même système d'exploitation ! De plus, la locale voulue n'est pas forcément présente, et son mode d'installation dépend du système d'exploitation et de sa distribution…

Pour éliminer ces problèmes tout en améliorant la flexibilité, PostgreSQL 10 a introduit les collations ICU, c'est-à-dire standardisées et versionnées dans une librairie séparée. En pratique, les paquets des distributions l'installent automatiquement avec PostgreSQL. Les collations linguistiques sont donc immédiatement disponibles via ICU :

```
CREATE COLLATION danois (provider = icu, locale = 'da-x-icu') ;
```

La librairie ICU fournit d'autres collations plus spécifiques liées à un contexte, par exemple l'ordre d'un annuaire ou l'ordre suivant la casse. Par exemple, cette collation très pratique tient compte de la valeur des chiffres (« tri naturel ») :

```
CREATE COLLATION nombres (provider = icu, locale = 'fr-u-kn-kr-latn-digit');

SELECT * FROM
  (VALUES ('1 sou'),('01 sou'),('02 sous'),('2 sous'),
  ('10 sous'),('0100 sous') ) AS n(n)
ORDER BY n COLLATE nombres ;
```

```
     n
-----------
 01 sou
 1 sou
 02 sous
 2 sous
 10 sous
 0100 sous
```

Alors que, par défaut, « 02 » précéderait « 1 » :

```
SELECT * FROM
  (VALUES ('1 sou'),('01 sou'),('02 sous'),('2 sous'),
  ('10 sous'),('0100 sous') ) AS n(n)
ORDER BY n ;  -- tri avec la locale par défaut
```

```
     n
-----------
 0100 sous
 01 sou
 02 sous
 10 sous
 1 sou
 2 sous
```

Pour d'autres exemples et les détails, voir ce billet de Peter Eisentraut[2] et la documentation officielle[3].

Pour voir les collations disponibles, consulter `pg_collation` :

```
  SELECT collname, collcollate, collprovider, collversion
  FROM pg_collation WHERE collname LIKE 'fr%' ;
```

```
  collname   | collcollate | collprovider | collversion
-------------+-------------+--------------+-------------
 fr-BE-x-icu | fr-BE       | i            | 153.80
 fr-BF-x-icu | fr-BF       | i            | 153.80
 fr-CA-x-icu | fr-CA       | i            | 153.80.32.1
 fr-x-icu    | fr          | i            | 153.80
 ...
 fr_FR       | fr_FR.utf8  | c            | ¤
 fr_FR.utf8  | fr_FR.utf8  | c            | ¤
 fr_LU       | fr_LU.utf8  | c            | ¤
 fr_LU.utf8  | fr_LU.utf8  | c            | ¤
(57 lignes)
```

Les collations installées dans la base sont visibles avec `\dO` sous `psql` :

```
=# \dO
                Liste des collationnements
 Schéma |     Nom      |    Collationnement    | … | Fournisseur | …
--------+--------------+-----------------------+---+-------------+----
 public | belge        | fr-BE-x-icu           | … | icu         | …
 public | chiffres_fin | fr-u-kn-kr-latn-digit | … | icu         | …
```

[2] https://blog.2ndquadrant.com/icu-support-postgresql-10/
[3] https://docs.postgresql.fr/current/collation.html

```
public | da_DK         | da_DK.utf8           | … | libc        | …
public | danois        | da-x-icu             | … | icu         | …
public | de_DE         | de_DE.utf8           | … | libc        | …
public | de_phonebook  | de-u-co-phonebk      | … | icu         | …
public | es_ES         | es_ES.utf8           | … | libc        | …
public | espagnol      | es-x-icu             | … | icu         | …
public | fr_FR         | fr_FR.utf8           | … | libc        | …
public | français      | fr-FR-x-icu          | … | icu         | …
```

5.5 TYPES AVANCÉS

- PostgreSQL propose des types plus avancés
- De nombreuses extensions !
  - faiblement structurés (JSON…)
  - intervalle
  - géométriques
  - tableaux

5.5.1 Types faiblement structurés

- PostgreSQL propose plusieurs types faiblement structurés :
  - `hstore` (clé/valeur historique)
  - JSON
  - XML

5.5.2 JSON

- `json`
  - stockage sous forme d'une chaîne de caractère
  - valide un document JSON sans modification
- `jsonb` (PG > 9.4)
  - stockage binaire optimisé
  - beaucoup plus de fonctions (dont jsonpath en v12)
  - à préférer

5.5.3 XML

- `xml`
  - stocke un document XML
  - valide sa structure
- Quelques opérateurs disponibles

5.6 TYPES INTERVALLE DE VALEURS

- Représentation d'intervalle
  - utilisable avec plusieurs types : entiers, dates, timestamps, etc.
  - contrainte d'exclusion

5.6.1 range

- représente un intervalle de valeurs continues
  - entre deux bornes
  - incluses ou non
- plusieurs types natifs
  - `int4range`, `int8range`, `numrange`
  - `daterange`, `tsrange`, `tstzrange`

Les intervalles de valeurs (*range*) représentent un ensemble de valeurs continues comprises entre deux bornes. Ces dernières sont entourées par des crochets [et] lorsqu'elles sont incluses, et par des parenthèses (et) lorsqu'elles sont exclues. L'absence de borne est admise et correspond à l'infini.

- `[0,10]` : toutes les valeurs comprises entre 0 à 10 ;
- `(100,200]` : toutes les valeurs comprises entre 100 et 200, 100 exclu ;
- `[2021-01-01,)` : toutes les dates supérieures au 1er janvier 2021 inclus ;
- `empty` : aucune valeur ou intervalle vide.

Le type abstrait `anyrange` se décline en `int4range` (`int`), `int8range` (`bigint`), `numrange` (`numeric`), `daterange` (`date`), `tsrange` (`timestamp without timezone`), `tstzrange` (`timestamp with timezone`).

5.6.2 Manipulation

- opérateurs spécifiques *, &&, <@ ou @>
- indexation avec GiST ou SP-GiST
- types personnalisés

Les opérateurs d'inclusion <@ et @> déterminent si une valeur ou un autre intervalle sont contenus dans l'intervalle de gauche ou de droite.

```
SELECT produit, date_validite FROM produits
 WHERE date_validite @> '2020-01-01'::date;
```

```
               produit                |      date_validite
--------------------------------------+-------------------------
 a0fd7a5a-6deb-4454-b7a7-9cd38eef53a4 | [2012-07-12,)
 79eb3a63-eb76-43b9-b1d6-f9f82dd77460 | [2019-07-31,2021-04-01)
 e4edaac4-33f1-426d-b2b0-4ea3b1c6caec | (,2020-01-02)
```

L'opérateur de chevauchement && détermine si deux intervalles du même type disposent d'au moins une valeur commune.

```
SELECT produit, date_validite FROM produits
 WHERE date_validite && '[2021-01-01,2021-12-31]'::daterange
```

```
               produit                |      date_validite
--------------------------------------+-------------------------
 8791d13f-bdfe-46f8-afc6-8be33acdbfc7 | [2012-07-12,)
 000a72d5-a90f-4030-aa15-f0a05e54b701 | [2019-07-31,2021-04-01)
```

L'opérateur d'intersection * reconstruit l'intervalle des valeurs continues et communes entre deux intervalles.

```
SELECT '[2021-01-01,2021-12-31]'::daterange
     * '[2019-07-31,2021-04-01)'::daterange AS intersection;
```

```
      intersection
-------------------------
 [2021-01-01,2021-04-01)
```

Pour garantir des temps de réponse acceptables sur les recherches avancées avec les opérateurs ci-dessus, il est nécessaire d'utiliser les index GiST ou SP-GiST. La syntaxe est la suivante :

```
CREATE INDEX ON produits USING gist (date_validite);
```

Enfin, il est possible de créer ses propres types `range` personnalisés à l'aide d'une fonction de différence. L'exemple ci-dessous permet de manipuler l'intervalle de données pour le type `time`. La fonction `time_subtype_diff()` est tirée de la documentation[4].

[4] https://docs.postgresql.fr/current/rangetypes.html#RANGETYPES-DEFINING

```
-- fonction utilitaire pour le type personnalisé "timerange"
CREATE FUNCTION time_subtype_diff(x time, y time)
RETURNS float8 AS
  'SELECT EXTRACT(EPOCH FROM (x - y))'
LANGUAGE sql STRICT IMMUTABLE;

-- définition du type "timerange", basé sur le type "time"
CREATE TYPE timerange AS RANGE (
  subtype = time,
  subtype_diff = time_subtype_diff
);

-- Exemple
SELECT '[11:10, 23:00]'::timerange;

      timerange
---------------------
 [11:10:00,23:00:00]
```

5.6.3 Contraintes d'exclusion

- Utilisation :
  - éviter le chevauchement de deux intervalles (*range*)
- Performance :
  - s'appuie sur un index

```
CREATE TABLE vendeurs (
  nickname varchar NOT NULL,
  plage_horaire timerange NOT NULL,
  EXCLUDE USING GIST (plage_horaire WITH &&)
);
```

Une contrainte d'exclusion s'apparente à une contrainte d'unicité, mais pour des intervalles de valeurs. Le principe consiste à identifier les chevauchements entre deux lignes pour prévenir l'insertion d'un doublon sur un intervalle commun.

Dans l'exemple suivant, nous utilisons le type personnalisé `timerange`, présenté ci-dessus. La table `vendeurs` reprend les agents de vente d'un magasin et leurs plages horaires de travail, valables pour tous les jours ouvrés de la semaine.

```
CREATE TABLE vendeurs (
  nickname varchar NOT NULL,
  plage_horaire timerange NOT NULL,
  EXCLUDE USING GIST (plage_horaire WITH &&)
);
```

```
INSERT INTO vendeurs (nickname, plage_horaire)
VALUES
  ('john', '[09:00:00,11:00:00)'::timerange),
  ('bobby', '[11:00:00,14:00:00)'::timerange),
  ('jessy', '[14:00:00,17:00:00)'::timerange),
  ('thomas', '[17:00:00,20:00:00]'::timerange);
```

Un index GiST est créé automatiquement pour la colonne `plage_horaire`.

```
postgres=# \x on
postgres=# \di+

List of relations
-[ RECORD 1 ]-+---------------------------
Schema        | public
Name          | vendeurs_plage_horaire_excl
Type          | index
Owner         | postgres
Table         | vendeurs
Persistence   | permanent
Access method | gist
Size          | 8192 bytes
Description   |
```

L'ajout d'un nouveau vendeur pour une plage déjà couverte par l'un de ces collègues est impossible, avec une violation de contrainte d'exclusion, gérée par l'opérateur de chevauchement &&.

```
INSERT INTO vendeurs (nickname, plage_horaire)
VALUES ('georges', '[10:00:00,12:00:00)'::timerange);

ERROR:  conflicting key value violates exclusion constraint
        "vendeurs_plage_horaire_excl"
DETAIL:  Key (plage_horaire)=([10:00:00,12:00:00)) conflicts
         with existing key (plage_horaire)=([09:00:00,11:00:00)).
```

Il est aussi possible de mixer les contraintes d'unicité et d'exclusion grâce à l'extension `btree_gist`. Dans l'exemple précédent, nous imaginons qu'un nouveau magasin ouvre et recrute de nouveaux vendeurs. La contrainte d'exclusion doit évoluer pour prendre en compte une nouvelle colonne, `magasin_id`.

```
CREATE EXTENSION btree_gist;
ALTER TABLE vendeurs
  DROP CONSTRAINT IF EXISTS vendeurs_plage_horaire_excl,
  ADD COLUMN magasin_id int NOT NULL DEFAULT 1,
  ADD EXCLUDE USING GIST (magasin_id WITH =, plage_horaire WITH &&);

INSERT INTO vendeurs (magasin_id, nickname, plage_horaire)
VALUES (2, 'georges', '[10:00:00,12:00:00)'::timerange);
```

En cas de recrutement pour une plage horaire déjà couverte par le nouveau magasin, la contrainte d'exclusion lèvera toujours une erreur, comme attendu.

```
INSERT INTO vendeurs (magasin_id, nickname, plage_horaire)
VALUES (2, 'laura', '[09:00:00,11:00:00)'::timerange);
```

```
ERROR:  conflicting key value violates exclusion constraint
        "vendeurs_magasin_id_plage_horaire_excl"
DETAIL:  Key (magasin_id, plage_horaire)=(2, [09:00:00,11:00:00)) conflicts
         with existing key (magasin_id, plage_horaire)=(2, [10:00:00,12:00:00)).
```

5.7 TYPES GÉOMÉTRIQUES

- Plusieurs types natifs 2D :
  - point, ligne, segment, polygone, cercle
- Utilisation :
  - stockage de géométries simples, sans référentiel de projection
- Pour la géographie :
  - extension PostGIS

5.8 TYPES UTILISATEURS

- Plusieurs types définissables par l'utilisateur
  - types composites
  - domaines
  - enums

5.8.1 Types composites

- Regroupe plusieurs attributs
  - la création d'une table implique la création d'un type composite associé
- Utilisation :
  - déclarer un tableau de données composites
  - en PL/pgSQL, déclarer une variable de type enregistrement

Les types composites sont assez difficiles à utiliser, car ils nécessitent d'adapter la syntaxe spécifiquement au type composite. S'il ne s'agit que de regrouper quelques attributs ensemble, autant les lister simplement dans la déclaration de la table.

En revanche, il peut être intéressant pour stocker un tableau de données composites dans une table.

5.8.2 Type énumération

- Ensemble fini de valeurs possibles
  - uniquement des chaînes de caractères
  - 63 caractères maximum
- Équivalent des énumérations des autres langages
- Utilisation :
  - listes courtes figées (statuts…)
  - évite des jointures

Référence :

- Type énumération[5]

[5] https://docs.postgresql.fr/current/datatype-enum.html

6/ SQL pour l’analyse de données

6.1 PRÉAMBULE

- Analyser des données est facile avec PostgreSQL
  - opérations d'agrégation disponibles
  - fonctions OLAP avancées

6.1.1 Menu

- Agrégation de données
- Clause FILTER
- Fonctions window
- GROUPING SETS, ROLLUP, CUBE
- WITHIN GROUPS

6.1.2 Objectifs

- Écrire des requêtes encore plus complexes
- Analyser les données en amont
  - pour ne récupérer que le résultat

6.2 AGRÉGATS

- SQL dispose de fonctions de calcul d'agrégats
- Utilité :
  - calcul de sommes, moyennes, valeur minimale et maximale
  - nombreuses fonctions statistiques disponibles

À l'aide des fonctions de calcul d'agrégats, on peut réaliser un certain nombre de calculs permettant d'analyser les données d'une table.

La plupart des exemples utilisent une table `employes` définie telle que :

```
CREATE TABLE employes (
  matricule char(8) primary key,
  nom       text    not null,
  service   text,
  salaire   numeric(7,2)
);

INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000001', 'Dupuis', 'Direction', 10000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000004', 'Fantasio', 'Courrier', 4500.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000006', 'Prunelle', 'Publication', 4000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000020', 'Lagaffe', 'Courrier', 3000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000040', 'Lebrac', 'Publication', 3000.00);

SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

Ainsi, on peut déduire le salaire moyen avec la fonction `avg()`, les salaires maximum et minimum versés par la société avec les fonctions `max()` et `min()`, ainsi que la somme totale des salaires versés avec la fonction `sum()` :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
```

```
       sum(salaire) AS somme_salaires
  FROM employes;
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 4900.0000000000000000 |        10000.00 |         3000.00 |       24500.00
```

La base de données réalise les calculs sur l'ensemble des données de la table et n'affiche que le résultat du calcul.

Si l'on applique un filtre sur les données, par exemple pour ne prendre en compte que le service *Courrier*, alors PostgreSQL réalise le calcul uniquement sur les données issues de la lecture :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
  FROM employes
 WHERE service = 'Courrier';
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 3750.0000000000000000 |         4500.00 |         3000.00 |        7500.00
(1 ligne)
```

En revanche, il n'est pas possible de référencer d'autres colonnes pour les afficher à côté du résultat d'un calcul d'agrégation à moins de les utiliser comme critère de regroupement :

```
SELECT avg(salaire), nom FROM employes;
ERROR:  column "employes.nom" must appear in the GROUP BY clause or be used in
        an aggregate function
LIGNE 1 : SELECT avg(salaire), nom FROM employes;
                               ^
```

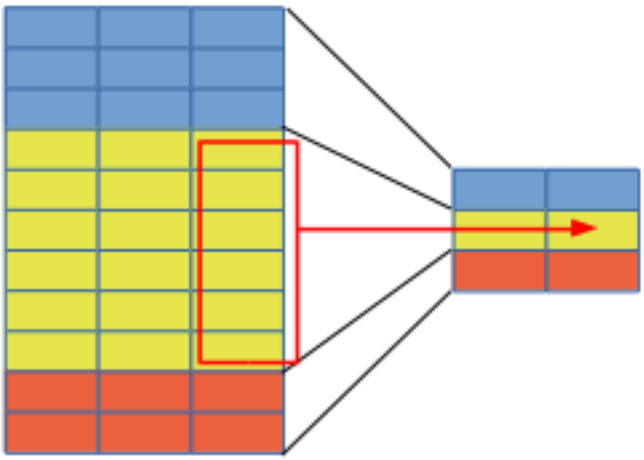
6.2.1 Agrégats avec GROUP BY

- agrégat + GROUP BY
- Utilité
  - effectue des calculs sur des regroupements : moyenne, somme, comptage, etc.
  - regroupement selon un critère défini par la clause GROUP BY
  - exemple : calcul du salaire moyen de chaque service

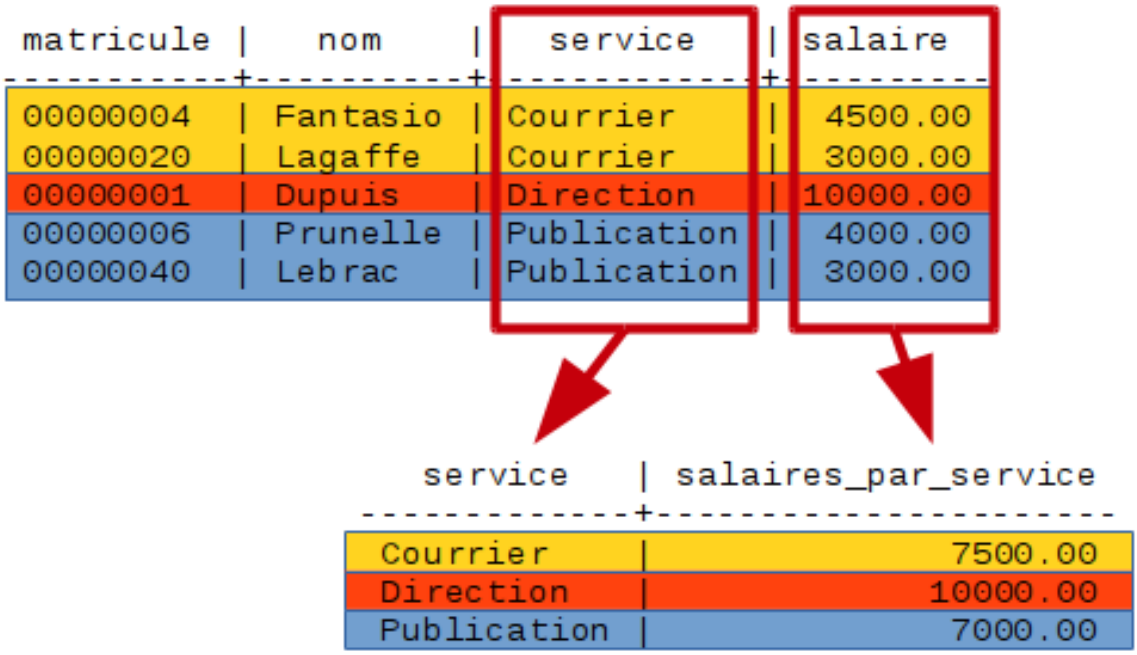
L'opérateur d'agrégat GROUP BY indique à la base de données que l'on souhaite regrouper les données selon les mêmes valeurs d'une colonne.

matricule	nom	service	salaire
00000004	Fantasio	Courrier	4500.00
00000020	Lagaffe	Courrier	3000.00
00000001	Dupuis	Direction	10000.00
00000006	Prunelle	Publication	4000.00
00000040	Lebrac	Publication	3000.00

Des calculs pourront être réalisés sur les données agrégées selon le critère de regroupement donné. Le résultat sera alors représenté en n'affichant que les colonnes de regroupement puis les valeurs calculées par les fonctions d'agrégation :



6.2.2 GROUP BY : principe



L'agrégation est ici réalisée sur la colonne `service`. En guise de calcul d'agrégation, une somme est réalisée sur les salaires payés dans chaque service.

6.2.3 GROUP BY : exemples

```
SELECT service,
       sum(salaire) AS salaires_par_service
  FROM employes
 GROUP BY service;

   service    | salaires_par_service
--------------+----------------------
 Courrier     |              7500.00
 Direction    |             10000.00
 Publication  |              7000.00
(3 lignes)
```

SQL permet depuis le début de réaliser des calculs d'agrégation. Pour cela, la base de données observe les critères de regroupement définis dans la clause GROUP BY de la requête et effectue l'opération sur l'ensemble des lignes qui correspondent au critère de regroupement.

On peut bien entendu combiner plusieurs opérations d'agrégations :

```
SELECT service,
       sum(salaire) salaires_par_service,
       avg(salaire) AS salaire_moyen_service
  FROM employes
 GROUP BY service;
```

```
   service    | salaires_par_service | salaire_moyen_service
--------------+----------------------+-----------------------
 Courrier     |              7500.00 |  3750.0000000000000000
 Direction    |             10000.00 | 10000.0000000000000000
 Publication  |              7000.00 |  3500.0000000000000000
(3 lignes)
```

On peut combiner le résultat de deux requêtes d'agrégation avec `UNION ALL`, si les ensembles retournées sont de même type :

```
SELECT service,
       sum(salaire) AS salaires_par_service
  FROM employes GROUP BY service
UNION ALL
SELECT 'Total' AS service,
       sum(salaire) AS salaires_par_service
  FROM employes;
```

```
   service    | salaires_par_service
--------------+----------------------
 Courrier     |              7500.00
 Direction    |             10000.00
 Publication  |              7000.00
 Total        |             24500.00
(4 lignes)
```

On le verra plus loin, cette dernière requête peut être écrite plus simplement avec les `GROUPING SETS`, mais qui nécessitent au minimum PostgreSQL 9.5.

6.2.4 Agrégats et ORDER BY

- Extension propriétaire de PostgreSQL
  - ORDER BY dans la fonction d'agrégat
- Utilité :
  - ordonner les données agrégées
  - surtout utile avec `array_agg`, `string_agg` et `xmlagg`

Les fonctions `array_agg`, `string_agg` et `xmlagg` permettent d'agréger des éléments dans un tableau, dans une chaîne ou dans une arborescence XML. Autant l'ordre dans lequel les données sont utilisées n'a pas d'importance lorsque l'on réalise un calcul d'agrégat classique, autant cet ordre va influencer la façon dont les données seront produites par les trois fonctions citées plus haut. En effet, le tableau généré par `array_agg` est composé d'éléments ordonnés, de même que la chaîne de caractères ou l'arborescence XML.

6.2.5 Utiliser ORDER BY avec un agrégat

```
SELECT service,
       string_agg(nom, ', ' ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |  liste_employes
--------------+-------------------
 Courrier     | Fantasio, Lagaffe
 Direction    | Dupuis
 Publication | Lebrac, Prunelle
(3 lignes)
```

La requête suivante permet d'obtenir, pour chaque service, la liste des employés dans un tableau, trié par ordre alphabétique :

```
SELECT service,
       string_agg(nom, ', ' ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |  liste_employes
--------------+-------------------
 Courrier     | Fantasio, Lagaffe
 Direction    | Dupuis
 Publication | Lebrac, Prunelle
(3 lignes)
```

Il est possible de réaliser la même chose mais pour obtenir un tableau plutôt qu'une chaîne de caractère :

```
SELECT service,
       array_agg(nom ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |    liste_employes
--------------+---------------------
 Courrier     | {Fantasio,Lagaffe}
 Direction    | {Dupuis}
 Publication | {Lebrac,Prunelle}
```

6.3 CLAUSE FILTER

- Clause `FILTER`
- Utilité :
  - filtrer les données sur les agrégats
  - évite les expressions CASE complexes
- SQL:2003
- Intégré dans la version 9.4

La clause `FILTER` permet de remplacer des expressions complexes écrites avec CASE et donc de simplifier l'écriture de requêtes réalisant un filtrage dans une fonction d'agrégat.

6.3.1 Filtrer avec CASE

- La syntaxe suivante était utilisée :

```
SELECT count(*) AS compte_pays,
       count(CASE WHEN r.nom_region='Europe' THEN 1
                  ELSE NULL
             END) AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

Avec cette syntaxe, dès que l'on a besoin d'avoir de multiples filtres ou de filtres plus complexes, la requête devient très rapidement peu lisible et difficile à maintenir. Le risque d'erreur est également élevé.

6.3.2 Filtrer avec FILTER

- La même requête écrite avec la clause `FILTER` :

```
SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe')
                AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

L'exemple suivant montre l'utilisation de la clause `FILTER` et son équivalent écrit avec une expression CASE :

```
sql=# SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe') AS compte_pays_europeens,
       count(CASE WHEN r.nom_region='Europe' THEN 1 END)
           AS oldschool_compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
 compte_pays | compte_pays_europeens | oldschool_compte_pays_europeens
-------------+-----------------------+---------------------------------
          25 |                     5 |                               5
(1 ligne)
```

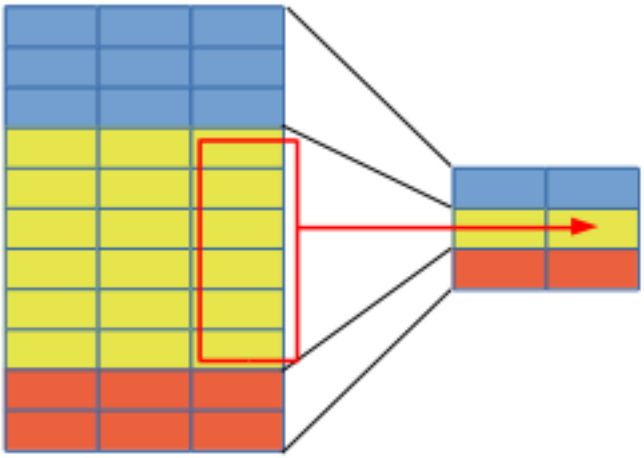
6.4 FONCTIONS DE FENÊTRAGE

- Fonctions *window*
  - travaille sur des ensembles de données regroupés et triés indépendamment de la requête principale
- Utilisation :
  - utiliser plusieurs critères d'agrégation dans la même requête
  - utiliser des fonctions de classement
  - faire référence à d'autres lignes de l'ensemble de données

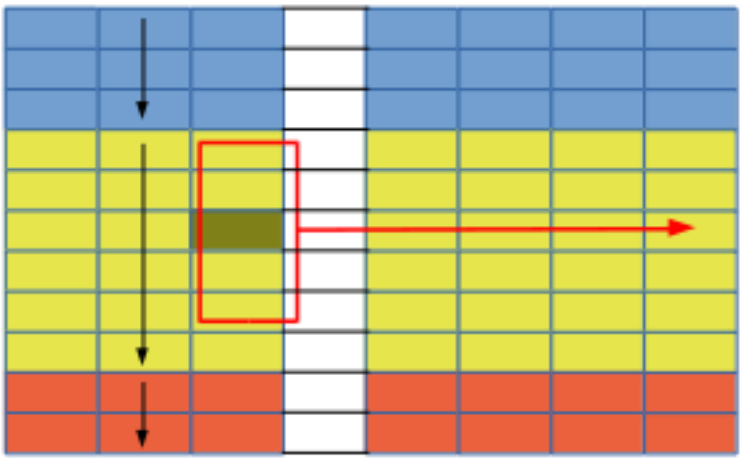
PostgreSQL supporte les fonctions de fenêtrage depuis la version 8.4. Elles apportent des fonctionnalités analytiques à PostgreSQL, et permettent d'écrire beaucoup plus simplement certaines requêtes.

Prenons un exemple.

```
SELECT service, AVG(salaire)
FROM employe
GROUP BY service
```



```
SELECT service, id_employe, salaire,
  AVG(salaire) OVER (
    PARTITION BY service
    ORDER BY age
    ROWS BETWEEN 2 PRECEEDING AND 2 FOLLOWING
  )
FROM employes
```



6.4.1 Regroupement

- Regroupement
  - clause `OVER (PARTITION BY ...)`
- Utilité :
  - plusieurs critères de regroupement différents
  - avec des fonctions de calcul d'agrégats

La clause `OVER` permet de définir la façon dont les données sont regroupées - uniquement pour la colonne définie - avec la clause `PARTITION BY`.

Les exemples vont utiliser cette table `employes` :

```
exemple=# SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

6.4.2 Regroupement : exemple

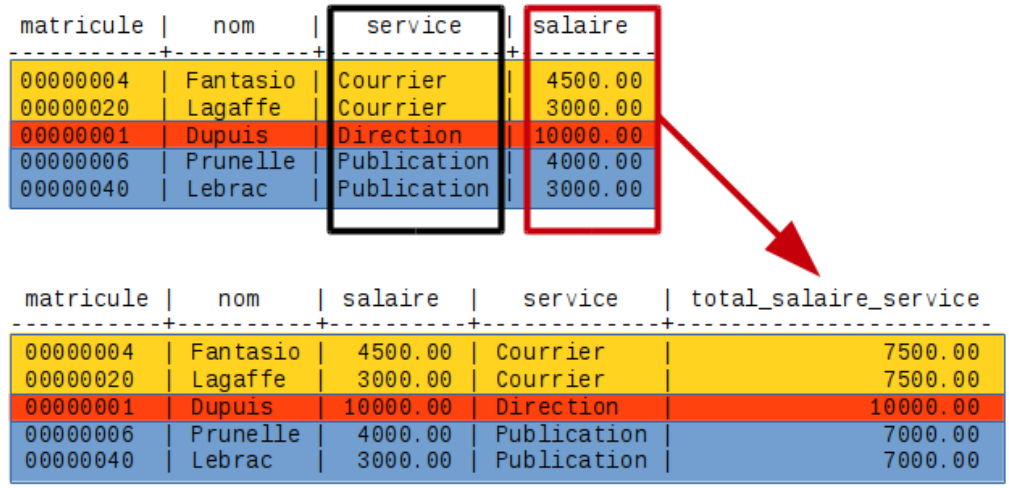
```
SELECT matricule, salaire, service,
       SUM(salaire) OVER (PARTITION BY service)
                 AS total_salaire_service
  FROM employes;
```

```
 matricule | salaire  |   service   | total_salaire_service
-----------+----------+-------------+-----------------------
 00000004  |  4500.00 | Courrier    |               7500.00
 00000020  |  3000.00 | Courrier    |               7500.00
 00000001  | 10000.00 | Direction   |              10000.00
 00000006  |  4000.00 | Publication |               7000.00
 00000040  |  3000.00 | Publication |               7000.00
```

Les calculs réalisés par cette requête sont identiques à ceux réalisés avec une agrégation utilisant GROUP BY. La principale différence est que l'on évite de ici de perdre le détail des données tout en disposant des données agrégées dans le résultat de la requête.

6.4.3 Regroupement : principe

```
SUM(salaire) OVER (PARTITION BY service)
```



Entouré de noir, le critère de regroupement et entouré de rouge, les données sur lesquelles sont appliqués le calcul d'agrégat.

6.4.4 Regroupement : syntaxe

```
SELECT ...
 agregation OVER (PARTITION BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Le terme `PARTITION BY` permet d'indiquer les critères de regroupement de la fenêtre sur laquelle on souhaite travailler.

6.4.5 Tri

- Tri
  - `OVER (ORDER BY …)`
- Utilité :
  - numéroter les lignes : `row_number()`
  - classer des résultats : `rank()`, `dense_rank()`
  - faire appel à d'autres lignes du résultat : `lead()`, `lag()`

6.4.6 Tri : exemple

- Pour numéroter des lignes :

```
SELECT row_number() OVER (ORDER BY matricule),
       matricule, nom
  FROM employes;

 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          3 | 00000006  | Prunelle
          4 | 00000020  | Lagaffe
          5 | 00000040  | Lebrac
(5 lignes)
```

La fonction `row_number()` permet de numéroter les lignes selon un critère de tri défini dans la clause OVER.

L'ordre de tri de la clause OVER n'influence pas l'ordre de tri explicite d'une requête :

```
SELECT row_number() OVER (ORDER BY matricule),
       matricule, nom
  FROM employes
 ORDER BY nom;
 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          4 | 00000020  | Lagaffe
          5 | 00000040  | Lebrac
          3 | 00000006  | Prunelle
(5 lignes)
```

On dispose aussi de fonctions de classement, pour déterminer par exemple les employés les moins bien payés :

```
SELECT matricule, nom, salaire, service,
       rank() OVER (ORDER BY salaire),
       dense_rank() OVER (ORDER BY salaire)
  FROM employes ;
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

La fonction de fenêtrage `rank()` renvoie le classement en autorisant des trous dans la numérotation, et `dense_rank()` le classement sans trous.

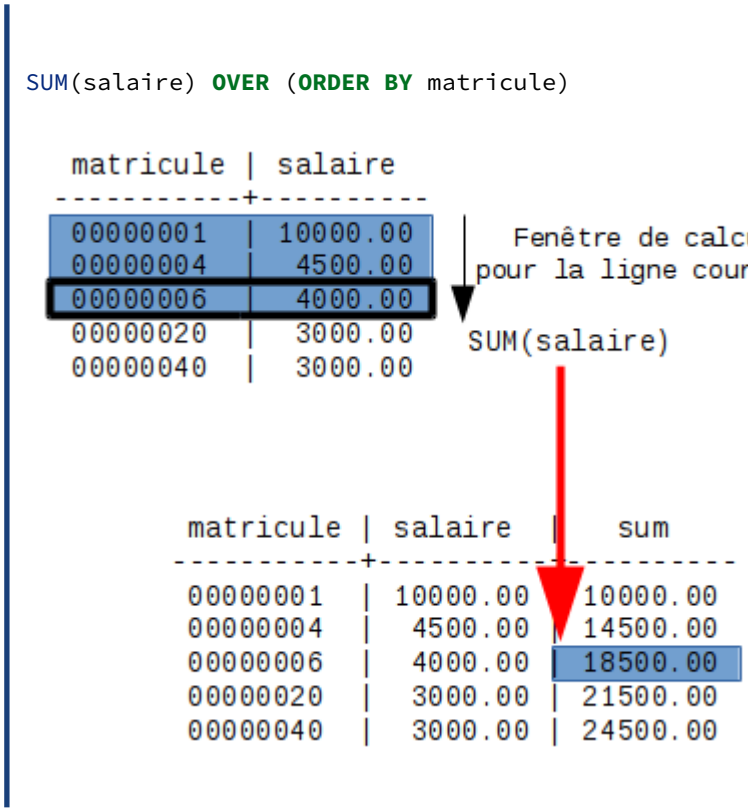
6.4.7 Tri : exemple avec une somme

- Calcul d'une somme glissante :

```
SELECT matricule, salaire,
       SUM(salaire) OVER (ORDER BY matricule)
  FROM employes;

 matricule | salaire  |   sum
-----------+----------+----------
 00000001  | 10000.00 | 10000.00
 00000004  |  4500.00 | 14500.00
 00000006  |  4000.00 | 18500.00
 00000020  |  3000.00 | 21500.00
 00000040  |  3000.00 | 24500.00
```

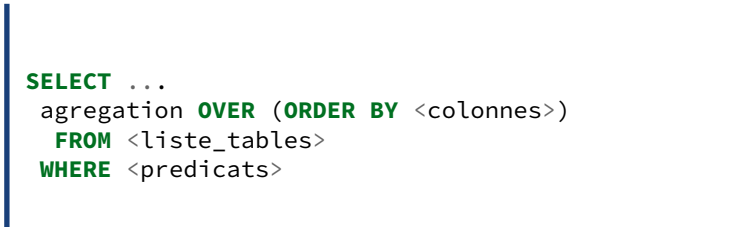
6.4.8 Tri : principe



Lorsque l'on utilise une clause de tri, la portion de données visible par l'opérateur d'agrégat correspond aux données comprises entre la première ligne examinée et la ligne courante. La fenêtre est définie selon le critère `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Nous verrons plus loin que nous pouvons modifier ce comportement.

6.4.9 Tri : syntaxe



```
SELECT ...
 agregation OVER (ORDER BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Le terme ORDER BY permet d'indiquer les critères de tri de la fenêtre sur laquelle on souhaite travailler.

6.4.10 Regroupement et tri

- On peut combiner les deux
  - `OVER (PARTITION BY .. ORDER BY ..)`
- Utilité :
  - travailler sur des jeux de données ordonnés et isolés les uns des autres

Il est possible de combiner les clauses de fenêtrage `PARTITION BY` et `ORDER BY`. Cela permet d'isoler des jeux de données entre eux avec la clause `PARTITION BY`, tout en appliquant un critère de tri avec la clause `ORDER BY`. Beaucoup d'applications sont possibles si l'on associe à cela les nombreuses fonctions analytiques disponibles.

6.4.11 Regroupement et tri : exemple

```
SELECT continent, pays, population,
       rank() OVER (PARTITION BY continent
                    ORDER BY population DESC)
              AS rang
  FROM population;
```

```
    continent     |       pays          | population | rang
------------------+---------------------+------------+------
 Afrique          | Nigéria             |      173.6 |    1
 Afrique          | Éthiopie            |       94.1 |    2
 Afrique          | Égypte              |       82.1 |    3
 Afrique          | Rép. dém. du Congo  |       67.5 |    4
(...)
 Amérique du Nord | États-Unis          |      320.1 |    1
 Amérique du Nord | Canada              |       35.2 |    2
(...)
```

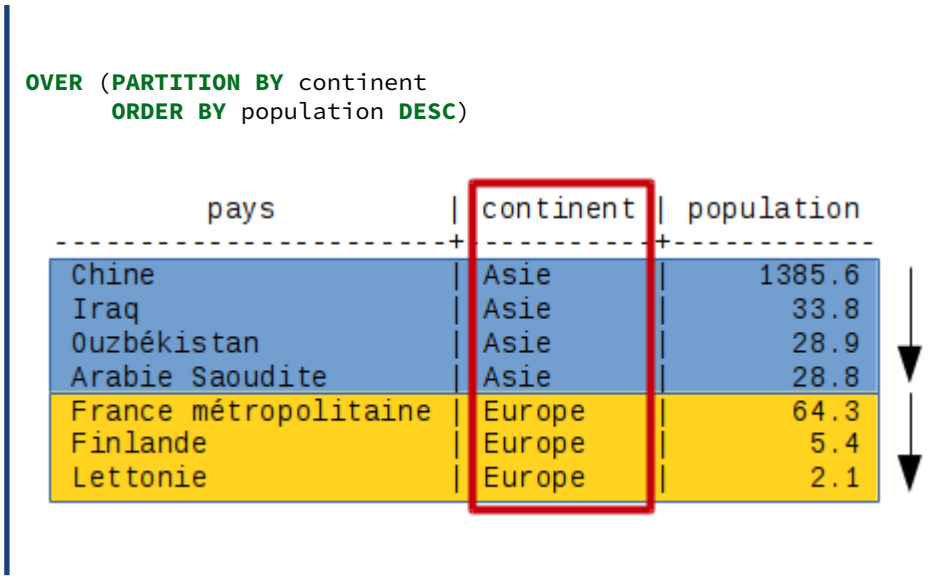
Si l'on applique les deux clauses `PARTITION BY` et `ORDER BY` à une fonction de fenêtrage, alors le critère de tri est appliqué dans la partition et chaque partition est indépendante l'une de l'autre.

Voici un extrait plus complet du résultat de la requête présentée ci-dessus :

```
         continent          |         pays        | population | rang_pop
----------------------------+---------------------+------------+----------
```

```
 Afrique                   | Nigéria            |      173.6 |        1
 Afrique                   | Éthiopie           |       94.1 |        2
 Afrique                   | Égypte             |       82.1 |        3
 Afrique                   | Rép. dém. du Congo |       67.5 |        4
 Afrique                   | Afrique du Sud     |       52.8 |        5
 Afrique                   | Tanzanie           |       49.3 |        6
 Afrique                   | Kenya              |       44.4 |        7
 Afrique                   | Algérie            |       39.2 |        8
 Afrique                   | Ouganda            |       37.6 |        9
 Afrique                   | Maroc              |       33.0 |       10
 Afrique                   | Ghana              |       25.9 |       11
 Afrique                   | Mozambique         |       25.8 |       12
 Afrique                   | Madagascar         |       22.9 |       13
 Afrique                   | Côte-d'Ivoire      |       20.3 |       14
 Afrique                   | Niger              |       17.8 |       15
 Afrique                   | Burkina Faso       |       16.9 |       16
 Afrique                   | Zimbabwe           |       14.1 |       17
 Afrique                   | Soudan             |       14.1 |       17
 Afrique                   | Tunisie            |       11.0 |       19
 Amérique du Nord          | États-Unis         |      320.1 |        1
 Amérique du Nord          | Canada             |       35.2 |        2
 Amérique latine. Caraïbes | Brésil             |      200.4 |        1
 Amérique latine. Caraïbes | Mexique            |      122.3 |        2
 Amérique latine. Caraïbes | Colombie           |       48.3 |        3
 Amérique latine. Caraïbes | Argentine          |       41.4 |        4
 Amérique latine. Caraïbes | Pérou              |       30.4 |        5
 Amérique latine. Caraïbes | Venezuela          |       30.4 |        5
 Amérique latine. Caraïbes | Chili              |       17.6 |        7
 Amérique latine. Caraïbes | Équateur           |       15.7 |        8
 Amérique latine. Caraïbes | Guatemala          |       15.5 |        9
 Amérique latine. Caraïbes | Cuba               |       11.3 |       10
(...)
```

6.4.12 Regroupement et tri : principe



```
OVER (PARTITION BY continent
      ORDER BY population DESC)
```

pays	continent	population
Chine	Asie	1385.6
Iraq	Asie	33.8
Ouzbékistan	Asie	28.9
Arabie Saoudite	Asie	28.8
France métropolitaine	Europe	64.3
Finlande	Europe	5.4
Lettonie	Europe	2.1

6.4.13 Regroupement et tri : syntaxe

```
SELECT ...
 <agregation> OVER (PARTITION BY <colonnes>
                    ORDER BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Cette construction ne pose aucune difficulté syntaxique. La norme impose de placer la clause PARTITION BY avant la clause ORDER BY, c'est la seule chose à retenir au niveau de la syntaxe.

6.4.14 Fonctions analytiques

- PostgreSQL dispose d'un certain nombre de fonctions analytiques
- Utilité :
  - faire référence à d'autres lignes du même ensemble
  - évite les auto-jointures complexes et lentes

Sans les fonctions analytiques, il était difficile en SQL d'écrire des requêtes nécessitant de faire appel à des données provenant d'autres lignes que la ligne courante.

Par exemple, pour renvoyer la liste détaillée de tous les employés ET le salaire le plus élevé du service auquel il appartient, on peut utiliser la fonction `first_value()` :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
 matricule |   nom    | salaire  |   service   | salaire_maximum_service
-----------+----------+----------+-------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier    |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier    |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction   |                10000.00
 00000006  | Prunelle |  4000.00 | Publication |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication |                 4000.00
(5 lignes)
```

Il existe également les fonctions suivantes :

- `last_value(colonne)` : renvoie la dernière valeur pour la colonne ;
- `nth(colonne, n)` : renvoie la n-ème valeur (en comptant à partir de **1**) pour la colonne ;
- `lag(colonne, n)` : renvoie la valeur située en n-ème position **avant** la ligne en cours pour la colonne ;
- `lead(colonne, n)` : renvoie la valeur située en n-ème position **après** la ligne en cours pour la colonne ;

  - pour ces deux fonctions, le n est facultatif et vaut **1** par défaut ;
  - ces deux fonctions acceptent un 3ème argument facultatif spécifiant la valeur à renvoyer si aucune valeur n'est trouvée en n-ème position avant ou après. Par défaut, `NULL` sera renvoyé.

6.4.15 lead() et lag()

- `lead(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **après** la ligne courante
- `lag(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **avant** la ligne courante

La construction `lead(colonne)` est équivalente à `lead(colonne, 1)`. De même, la construction `lag(colonne)` est équivalente à `lag(colonne, 1)`. Il s'agit d'un raccourci pour utiliser la valeur précédente ou la valeur suivante d'une colonne dans la fenêtre définie.

6.4.16 lead() et lag() : exemple

```
SELECT pays, continent, population,
       lag(population) OVER (PARTITION BY continent
                             ORDER BY population DESC)
  FROM population;
         pays           | continent | population |  lag
------------------------+-----------+------------+--------
 Chine                  | Asie      |     1385.6 |
 Iraq                   | Asie      |       33.8 | 1385.6
 Ouzbékistan            | Asie      |       28.9 |   33.8
 Arabie Saoudite        | Asie      |       28.8 |   28.9
 France métropolitaine  | Europe    |       64.3 |
 Finlande               | Europe    |        5.4 |   64.3
 Lettonie               | Europe    |        2.1 |    5.4
```

La requête présentée en exemple ne s'appuie que sur un jeu réduit de données afin de montrer un résultat compréhensible.

6.4.17 lead() et lag() : principe

```
lag(population) OVER (PARTITION BY continent
                      ORDER BY population DESC)
```

pays	continent	population	lag
Chine	Asie	1385.6	
Iraq	Asie	33.8	1385.6
Ouzbékistan	Asie	28.9	33.8
Arabie Saoudite	Asie	28.8	28.9
France métropolitaine	Europe	64.3	
Finlande	Europe	5.4	64.3
Lettonie	Europe	2.1	5.4

NULL est renvoyé lorsque la valeur n'est pas accessible dans la fenêtre de données, comme par exemple si l'on souhaite utiliser la valeur d'une colonne appartenant à la ligne précédant la première ligne de la partition.

6.4.18 first/last/nth_value

- `first_value(colonne)`
  - retourne la première valeur pour la colonne
- `last_value(colonne)`
  - retourne la dernière valeur pour la colonne
- `nth_value(colonne, n)`
  - retourne la n-ème valeur (en comptant à partir de 1) pour la colonne

Utilisé avec `ORDER BY` et `PARTITION BY`, la fonction `first_value()` permet par exemple d'obtenir le salaire le plus élevé d'un service :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
```

```
 matricule |   nom    | salaire  |   service    | salaire_maximum_service
-----------+----------+----------+--------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier     |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier     |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction    |                10000.00
 00000006  | Prunelle |  4000.00 | Publication  |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication  |                 4000.00
(5 lignes)
```

6.4.19 first/last/nth_value : exemple

```
SELECT pays, continent, population,
       first_value(population)
           OVER (PARTITION BY continent
                 ORDER BY population DESC)
  FROM population;
```

```
       pays       | continent | population | first_value
------------------+-----------+------------+-------------
 Chine            | Asie      |     1385.6 |      1385.6
 Iraq             | Asie      |       33.8 |      1385.6
 Ouzbékistan      | Asie      |       28.9 |      1385.6
 Arabie Saoudite  | Asie      |       28.8 |      1385.6
 France           | Europe    |       64.3 |        64.3
 Finlande         | Europe    |        5.4 |        64.3
 Lettonie         | Europe    |        2.1 |        64.3
```

Lorsque que la clause ORDER BY est utilisée pour définir une fenêtre, la fenêtre visible depuis la ligne courante commence par défaut à la première ligne de résultat et s'arrête à la ligne courante.

Par exemple, si l'on exécute la même requête en utilisant `last_value()` plutôt que `first_value()`, on récupère à chaque fois la valeur de la colonne sur la ligne courante :

```
SELECT pays, continent, population,
       last_value(population) OVER (PARTITION BY continent
                         ORDER BY population DESC)
  FROM population;
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Chine                 | Asie      |     1385.6 |     1385.6
 Iraq                  | Asie      |       33.8 |       33.8
 Ouzbékistan           | Asie      |       28.9 |       28.9
```

```
 Arabie Saoudite       | Asie      |       28.8 |       28.8
 France métropolitaine | Europe    |       64.3 |       64.3
 Finlande              | Europe    |        5.4 |        5.4
 Lettonie              | Europe    |        2.1 |        2.1
(7 rows)
```

Il est alors nécessaire de redéfinir le comportement de la fenêtre visible pour que la fonction se comporte comme attendu, en utilisant `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` - cet aspect sera décrit dans la section sur les possibilités de modification de la définition de la fenêtre.

6.4.20 Clause WINDOW

- Pour factoriser la définition d'une fenêtre :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
 WINDOW w AS (ORDER BY salaire);
```

Il arrive que l'on ait besoin d'utiliser plusieurs fonctions de fenêtrage au sein d'une même requête qui utilisent la même définition de fenêtre (même clause `PARTITION BY` et/ou `ORDER BY`). Afin d'éviter de dupliquer cette clause, il est possible de définir une fenêtre nommée et de l'utiliser à plusieurs endroits de la requête. Par exemple, l'exemple précédent des fonctions de classement pourrait s'écrire :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
  WINDOW w AS (ORDER BY salaire);
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

À noter qu'il est possible de définir de multiples définitions de fenêtres au sein d'une même requête, et qu'une définition de fenêtre peut surcharger la clause `ORDER BY` si la définition parente ne l'a pas définie. Par exemple, la requête SQL suivante est correcte :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w_asc,
```

```
      dense_rank() OVER w_desc
FROM employes
WINDOW w AS (PARTITION BY service),
       w_asc AS (w ORDER BY salaire),
       w_desc AS (w ORDER BY salaire DESC);
```

6.4.21 Clause WINDOW : syntaxe

```
SELECT fonction_agregat OVER nom,
       fonction_agregat_2 OVER nom ...
       ...
  FROM <liste_tables>
 WHERE <predicats>
 WINDOW nom AS (PARTITION BY ... ORDER BY ...)
```

6.4.22 Définition de la fenêtre

- La fenêtre de travail par défaut est :

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

- Trois modes possibles :
  - RANGE
  - ROWS
  - GROUPS (v11+)
- Nécessite une clause ORDER BY

6.4.23 Définition de la fenêtre : RANGE

- Indique un intervalle à bornes *flou*
- Borne de départ :
  - `UNBOUNDED PRECEDING`: depuis le début de la partition
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `UNBOUNDED FOLLOWING` : jusqu'à la fin de la partition
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
      ORDER BY ...
      RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

6.4.24 Définition de la fenêtre : ROWS

- Indique un intervalle borné par un nombre de ligne défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx valeurs devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx valeurs derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
      ORDER BY ...
      ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

6.4.25 Définition de la fenêtre : GROUPS

- Indique un intervalle borné par un groupe de lignes de valeurs identiques défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx groupes de valeurs identiques devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante ou le premier élément identique dans le tri réalisé par ORDER BY
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx groupes de valeurs identiques derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante ou le dernier élément identique dans le tri réalisé par ORDER BY

```
OVER (PARTITION BY ...
      ORDER BY ...
      GROUPS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

Ceci n'est disponible que depuis la version 11.

6.4.26 Définition de la fenêtre : EXCLUDE

- Indique des lignes à exclure de la fenêtre de données (v11+)
- `EXCLUDE CURRENT ROW` : exclut la ligne courante
- `EXCLUDE GROUP` : exclut la ligne courante et le groupe de valeurs identiques dans l'ordre
- `EXCLUDE TIES` exclut et le groupe de valeurs identiques à la ligne courante dans l'ordre mais pas la ligne courante
- `EXCLUDE NO OTHERS` : pas d'exclusion (valeur par défaut)

Ceci n'est disponible que depuis la version 11.

6.4.27 Définition de la fenêtre : exemple

```
SELECT pays, continent, population,
       last_value(population)
        OVER (PARTITION BY continent ORDER BY population
              RANGE BETWEEN UNBOUNDED PRECEDING
                        AND UNBOUNDED FOLLOWING)
  FROM population;
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Arabie Saoudite       | Asie      |       28.8 |     1385.6
 Ouzbékistan           | Asie      |       28.9 |     1385.6
 Iraq                  | Asie      |       33.8 |     1385.6
 Chine (4)             | Asie      |     1385.6 |     1385.6
 Lettonie              | Europe    |        2.1 |       64.3
 Finlande              | Europe    |        5.4 |       64.3
 France métropolitaine | Europe    |       64.3 |       64.3
```

6.5 WITHIN GROUP

- WITHIN GROUP
  - PostgreSQL 9.4
- Utilité :
  - calcul de médianes, centiles

La clause WITHIN GROUP est une nouvelle clause pour les agrégats utilisant des fonctions dont les données doivent être triées. Quelques fonctions ont été ajoutées pour profiter au mieux de cette nouvelle clause.

6.5.1 WITHIN GROUP : exemple

```
SELECT continent,
  percentile_disc(0.5)
    WITHIN GROUP (ORDER BY population) AS "mediane",
  percentile_disc(0.95)
    WITHIN GROUP (ORDER BY population) AS "95pct",
  ROUND(AVG(population), 1) AS moyenne
FROM population
 GROUP BY continent;

         continent          | mediane | 95pct  | moyenne
----------------------------+---------+--------+---------
 Afrique                    |    33.0 |  173.6 |    44.3
 Amérique du Nord           |    35.2 |  320.1 |   177.7
 Amérique latine. Caraïbes  |    30.4 |  200.4 |    53.3
 Asie                       |    53.3 | 1252.1 |   179.9
 Europe                     |     9.4 |   82.7 |    21.8
```

Cet exemple permet d'afficher le continent, la médiane de la population par continent et la population du pays le moins peuplé parmi les 5% de pays les plus peuplés de chaque continent.

Pour rappel, la table contient les données suivantes :

```
postgres=# SELECT * FROM population ORDER BY continent, population;
         pays           | population | superficie | densite | continent
------------------------+------------+------------+---------+----------
```

```
 Tunisie                |       11.0 |        164 |      67 | Afrique
 Zimbabwe               |       14.1 |        391 |      36 | Afrique
 Soudan                 |       14.1 |        197 |      72 | Afrique
 Burkina Faso           |       16.9 |        274 |      62 | Afrique
(...)
```

En ajoutant le support de cette clause, PostgreSQL améliore son support de la norme SQL 2008 et permet le développement d'analyses statistiques plus élaborées.

6.6 GROUPING SETS

- `GROUPING SETS/ROLLUP/CUBE`
- Extension de `GROUP BY`
- PostgreSQL 9.5
- Utilité :
  - présente le résultat de plusieurs agrégations différentes
  - réaliser plusieurs agrégations différentes dans la même requête

Les `GROUPING SETS` permettent de définir plusieurs clauses d'agrégation `GROUP BY`. Les résultats seront présentés comme si plusieurs requêtes d'agrégation avec les clauses `GROUP BY` mentionnées étaient assemblées avec `UNION ALL`.

6.6.1 GROUPING SETS : jeu de données

stock		
piece	region	quantite
ecrous	est	50
ecrous	ouest	0
ecrous	sud	40
clous	est	70
clous	nord	40
vis	ouest	50
vis	sud	50
vis	nord	60

stock				
piece/region	est	ouest	sud	nord
ecrous	50	0	40	
clous	70			0
vis		50	50	60

```
CREATE TABLE stock AS SELECT * FROM (
    VALUES ('ecrous', 'est',   50),
           ('ecrous', 'ouest',  0),
```

```
            ('ecrous', 'sud',   40),
            ('clous',  'est',   70),
            ('clous',  'nord',   0),
            ('vis',    'ouest', 50),
            ('vis',    'sud',   50),
            ('vis',    'nord',  60)
) AS VALUES(piece, region, quantite);
```

6.6.2 GROUPING SETS : exemple visuel

sum (quantite) … grouping sets (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total	120	50	90	60	

6.6.3 GROUPING SETS : exemple ordre sql

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
 piece  | region | sum
--------+--------+-----
 clous  |        |  70
 ecrous |        |  90
 vis    |        | 160
        | est    | 120
        | nord   |  60
        | ouest  |  50
        | sud    |  90
```

6.6.4 GROUPING SETS : équivalent

- On peut se passer de la clause GROUPING SETS
  - mais la requête sera plus lente

```
SELECT piece,NULL as region,sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region,sum(quantite)
  FROM STOCK
  GROUP BY region;
```

Le comportement de la clause GROUPING SETS peut être émulée avec deux requêtes utilisant chacune une clause GROUP BY sur les colonnes de regroupement souhaitées.

Cependant, le plan d'exécution de la requête équivalente conduit à deux lectures et peut être particulièrement coûteux si le jeu de données sur lequel on souhaite réaliser les agrégations est important :

```
EXPLAIN SELECT piece,NULL as region,sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region,sum(quantite)
  FROM STOCK
  GROUP BY region;

                                QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=1.12..2.38 rows=7 width=44)
   ->  HashAggregate  (cost=1.12..1.15 rows=3 width=45)
         Group Key: stock.piece
         ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=9)
   ->  HashAggregate  (cost=1.12..1.16 rows=4 width=44)
         Group Key: stock_1.region
         ->  Seq Scan on stock stock_1  (cost=0.00..1.08 rows=8 width=8)
```

La requête utilisant la clause GROUPING SETS propose un plan bien plus efficace :

```
EXPLAIN SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
                            QUERY PLAN
-------------------------------------------------------------------
 GroupAggregate  (cost=1.20..1.58 rows=14 width=17)
   Group Key: piece
   Sort Key: region
     Group Key: region
   ->  Sort  (cost=1.20..1.22 rows=8 width=13)
```

```
        Sort Key: piece
        ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=13)
```

6.6.5 ROLLUP

- ROLLUP
- PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête

La clause ROLLUP est une fonctionnalité d'analyse type OLAP du langage SQL. Elle s'utilise dans la clause GROUP BY, tout comme GROUPING SETS

6.6.6 ROLLUP : exemple visuel

sum (quantite) … ROLLUP (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total					320

6.6.7 ROLLUP : exemple ordre sql

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY ROLLUP (piece,region);
```

Cette requête est équivalente à la requête suivante utilisant `GROUPING SETS` :

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY GROUPING SETS ((),(piece),(piece,region));
```

Sur une requête un peu plus intéressante, effectuant des statistiques sur des ventes :

```
SELECT type_client, code_pays, SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           |           |  111557177.00
(...)
 P           | RU        |  287605812.99
 P           | US        |  296424154.49
 P           |           | 4692152751.08
             |           | 5217862160.65
```

Une fonction `GROUPING`, associée à ROLLUP, permet de déterminer si la ligne courante correspond à un regroupement donné. Elle est de la forme d'un masque de bit converti au format décimal :

```
SELECT row_number()
         OVER ( ORDER BY grouping(piece,region)) AS ligne,
       grouping(piece,region)::bit(2) AS g,
```

```
       piece,
       region,
       sum(quantite)
FROM stock
GROUP BY CUBE (piece,region)
ORDER BY g ;
```

```
 ligne | g  | piece  | region | sum
-------+----+--------+--------+-----
     1 | 00 | clous  | est    | 150
     2 | 00 | clous  | nord   |  10
     3 | 00 | ecrous | est    | 110
     4 | 00 | ecrous | ouest  |  10
     5 | 00 | ecrous | sud    |  90
     6 | 00 | vis    | nord   | 130
     7 | 00 | vis    | ouest  | 110
     8 | 00 | vis    | sud    | 110
     9 | 01 | vis    |        | 350
    10 | 01 | ecrous |        | 210
    11 | 01 | clous  |        | 160
    12 | 10 |        | ouest  | 120
    13 | 10 |        | sud    | 200
    14 | 10 |        | est    | 260
    15 | 10 |        | nord   | 140
    16 | 11 |        |        | 720
```

Voici un autre exemple :

```
SELECT COALESCE(service,
    CASE
        WHEN GROUPING(service) = 0 THEN 'Unknown' ELSE 'Total'
    END) AS service,
       sum(salaire) AS salaires_service, count(*) AS nb_employes
  FROM employes
 GROUP BY ROLLUP (service);
```

```
  service    | salaires_service | nb_employes
-------------+------------------+-------------
 Courrier    |          7500.00 |           2
 Direction   |         50000.00 |           1
 Publication |          7000.00 |           2
 Total       |         64500.00 |           5
(4 rows)
```

Ou appliqué à l'exemple un peu plus complexe :

```
SELECT COALESCE(type_client,
       CASE
         WHEN GROUPING(type_client) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS type_client,
       COALESCE(code_pays,
       CASE
         WHEN GROUPING(code_pays) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
```

```
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           | Total     |  111557177.00
(...)
 P           | US        |  296424154.49
 P           | Total     | 4692152751.08
 Total       | Total     | 5217862160.65
```

6.6.8 CUBE

- CUBE
  - PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête
  - sur toutes les clauses de regroupement

La clause CUBE est une autre fonctionnalité d'analyse type OLAP du langage SQL. Tout comme ROLLUP, elle s'utilise dans la clause GROUP BY.

6.6.9 CUBE : exemple visuel

sum (quantite) … CUBE (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total	120	50	90	60	320

6.6.10 CUBE : exemple ordre sql

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY CUBE (piece,region);
```

Cette requête est équivalente à la requête suivante utilisant `GROUPING SETS` :

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY GROUPING SETS (
  (),
  (piece),
  (region),
  (piece,region)
  );
```

Elle permet de réaliser des regroupements sur l'ensemble des combinaisons possibles des clauses de regroupement indiquées. Pour de plus amples détails, se référer à cet article Wikipédia[1].

En reprenant la requête de l'exemple précédent :

```
SELECT type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
```

[1] https://en.wikipedia.org/wiki/OLAP_cube

```
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle retournera le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           |           |  111557177.00
 E           | CA        |   28457655.81
 E           | CN        |   25537539.68
 E           | DE        |   25508815.68
 E           | DZ        |   24821750.17
 E           | FR        |  209402443.24
 E           | IN        |   26788642.27
 E           | PE        |   24541974.54
 E           | RU        |   25397116.39
 E           | US        |   23696294.79
 E           |           |  414152232.57
 P           | CA        |  292975985.52
 P           | CN        |  287795272.87
 P           | DE        |  287337725.21
 P           | DZ        |  302501132.54
 P           | FR        | 2341977444.49
 P           | IN        |  295256262.73
 P           | PE        |  300278960.24
 P           | RU        |  287605812.99
 P           | US        |  296424154.49
 P           |           | 4692152751.08
             |           | 5217862160.65
             | CA        |  327706809.65
             | CN        |  321261454.05
             | DE        |  319488602.46
             | DZ        |  333727307.87
             | FR        | 2606641183.25
             | IN        |  329268913.95
             | PE        |  332177174.71
             | RU        |  319769574.36
             | US        |  327821140.35
```

Dans ce genre de contexte, lorsque le regroupement est réalisé sur l'ensemble des valeurs d'un critère de regroupement, alors la valeur qui apparaît est NULL pour la colonne correspondante. Si la colonne

possède des valeurs NULL légitimes, il est alors difficile de les distinguer. On utilise alors la fonction GROUPING() qui permet de déterminer si le regroupement porte sur l'ensemble des valeurs de la colonne. L'exemple suivant montre une requête qui exploite cette fonction :

```
SELECT GROUPING(type_client,code_pays)::bit(2),
       GROUPING(type_client)::boolean g_type_cli,
       GROUPING(code_pays)::boolean g_code_pays,
       type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 grouping | g_type_cli | g_code_pays | type_client | code_pays |    montant
----------+------------+-------------+-------------+-----------+---------------
 00       | f          | f           | A           | CA        |    6273168.32
 00       | f          | f           | A           | CN        |    7928641.50
 00       | f          | f           | A           | DE        |    6642061.57
 00       | f          | f           | A           | DZ        |    6404425.16
 00       | f          | f           | A           | FR        |   55261295.52
 00       | f          | f           | A           | IN        |    7224008.95
 00       | f          | f           | A           | PE        |    7356239.93
 00       | f          | f           | A           | RU        |    6766644.98
 00       | f          | f           | A           | US        |    7700691.07
 01       | f          | t           | A           |           |  111557177.00
(...)
 01       | f          | t           | P           |           | 4692152751.08
 11       | t          | t           |             |           | 5217862160.65
 10       | t          | f           |             | CA        |  327706809.65
 10       | t          | f           |             | CN        |  321261454.05
 10       | t          | f           |             | DE        |  319488602.46
 10       | t          | f           |             | DZ        |  333727307.87
 10       | t          | f           |             | FR        | 2606641183.25
 10       | t          | f           |             | IN        |  329268913.95
 10       | t          | f           |             | PE        |  332177174.71
 10       | t          | f           |             | RU        |  319769574.36
 10       | t          | f           |             | US        |  327821140.35
(40 rows)
```

L'application sera alors à même de gérer la présentation des résultats en fonction des valeurs de `grouping` ou `g_type_client` et `g_code_pays`.

6.7 TRAVAUX PRATIQUES

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

```
     Table "public.brno_2015"
  Column   |   Type   | Modifiers
-----------+----------+-----------
 no_tour   | integer  |
 no_pilote | integer  |
 lap_time  | interval |
```

Une table `pilotes` permet de connaître les détails d'un pilote :

```
      Table "public.pilotes"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 no          | integer |
 nom         | text    |
 nationalite | text    |
 ecurie      | text    |
 moto        | text    |
```

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

Agrégation

1. Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?
2. Déterminer quel est le pilote le plus régulier (écart-type).

Window Functions

3. Afficher la place sur le podium pour chaque coureur.
4. À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.
5. Pour chaque tour, afficher :

- le nom du pilote ;
- son rang dans le tour ;
- son temps depuis le début de la course ;
- dans le tour, la différence de temps par rapport au premier.

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?
7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.
8. En quelle position a terminé le coureur qui a doublé le plus de personnes ? Combien de personnes a-t-il doublées ?

Grouping Sets

Ce TP nécessite PostgreSQL 9.5 ou supérieur. Il s'appuie sur les tables présentes dans le schéma `magasin`.

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.
10. Ajouter également le montant total des commandes depuis le début de l'activité.
11. Ajouter également le montant total des commandes par pays.

6.8 TRAVAUX PRATIQUES (SOLUTIONS)

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

```
     Table "public.brno_2015"
  Column   |   Type   | Modifiers
-----------+----------+-----------
 no_tour   | integer  |
 no_pilote | integer  |
 lap_time  | interval |
```

Une table `pilotes` permet de connaître les détails d'un pilote :

```
      Table "public.pilotes"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 no          | integer |
 nom         | text    |
 nationalite | text    |
 ecurie      | text    |
 moto        | text    |
```

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

Agrégation

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

1. Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?

Le coureur :

```
SELECT nom, max(lap_time) - min(lap_time) as ecart
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 GROUP BY 1
 ORDER BY 2
 LIMIT 1;
```

La requête donne le résultat suivant :

```
       nom       |    ecart
-----------------+-------------
 Jorge LORENZO | 00:00:04.661
```

2. Déterminer quel est le pilote le plus régulier (écart-type).

Nous excluons le premier tour car il s'agit d'une course avec départ arrêté, donc ce tour est plus lent que les autres, ici d'au moins 8 secondes :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 WHERE no_tour > 1
 GROUP BY 1
 ORDER BY 2
 LIMIT 1;
```

Le résultat montre le coureur qui a abandonné en premier :

```
       nom       |      stddev
-----------------+-------------------
 Alex DE ANGELIS | 0.130107647741847
```

On s'aperçoit qu'Alex De Angelis n'a pas terminé la course. Il semble donc plus intéressant de ne prendre en compte que les pilotes qui ont terminé la course et toujours en excluant le premier tour (il y a 22 tours sur cette course, on peut le positionner soit en dur dans la requête, soit avec un sous-select permettant de déterminer le nombre maximum de tours) :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 WHERE no_tour > 1
   AND no_pilote in (SELECT no_pilote FROM brno_2015 WHERE no_tour=22)
GROUP BY 1
ORDER BY 2
LIMIT 1;
```

Le pilote 19 a donc été le plus régulier :

```
       nom       |      stddev
-----------------+-------------------
 Alvaro BAUTISTA | 0.222825823492654
```

Window Functions

Si ce n'est pas déjà fait, nous positionnons le search_path pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

3. Afficher la place sur le podium pour chaque coureur.

Les coureurs qui ne franchissent pas la ligne d'arrivée sont dans le classement malgré tout. Il faut donc tenir compte de cela dans l'affichage des résultats.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
       nom, ecurie, total_time
  FROM (SELECT no_pilote,
               sum(lap_time) over (PARTITION BY no_pilote) as total_time,
               max(no_tour) over (PARTITION BY no_pilote) as max_lap
          FROM brno_2015
       ) AS race_data
  JOIN pilotes
```

```
    ON (race_data.no_pilote = pilotes.no)
 GROUP BY nom, ecurie, max_lap, total_time
 ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

```
 rang |       nom        |           ecurie             |  total_time
------+------------------+------------------------------+--------------
    1 | Jorge LORENZO    | Movistar Yamaha MotoGP       | 00:42:53.042
    2 | Marc MARQUEZ     | Repsol Honda Team            | 00:42:57.504
    3 | Valentino ROSSI  | Movistar Yamaha MotoGP       | 00:43:03.439
    4 | Andrea IANNONE   | Ducati Team                  | 00:43:06.113
    5 | Dani PEDROSA     | Repsol Honda Team            | 00:43:08.692
    6 | Andrea DOVIZIOSO | Ducati Team                  | 00:43:08.767
    7 | Bradley SMITH    | Monster Yamaha Tech 3        | 00:43:14.863
    8 | Pol ESPARGARO    | Monster Yamaha Tech 3        | 00:43:16.282
    9 | Aleix ESPARGARO  | Team SUZUKI ECSTAR           | 00:43:36.826
   10 | Danilo PETRUCCI  | Octo Pramac Racing           | 00:43:38.303
   11 | Yonny HERNANDEZ  | Octo Pramac Racing           | 00:43:43.015
   12 | Scott REDDING    | EG 0,0 Marc VDS              | 00:43:43.216
   13 | Alvaro BAUTISTA  | Aprilia Racing Team Gresini  | 00:43:47.479
   14 | Stefan BRADL     | Aprilia Racing Team Gresini  | 00:43:47.666
   15 | Loris BAZ        | Forward Racing               | 00:43:53.358
   16 | Hector BARBERA   | Avintia Racing               | 00:43:54.637
   17 | Nicky HAYDEN     | Aspar MotoGP Team            | 00:43:55.43
   18 | Mike DI MEGLIO   | Avintia Racing               | 00:43:58.986
   19 | Jack MILLER      | CWM LCR Honda                | 00:44:04.449
   20 | Claudio CORTI    | Forward Racing               | 00:44:43.075
   21 | Karel ABRAHAM    | AB Motoracing                | 00:44:55.697
   22 | Maverick VIÑALES | Team SUZUKI ECSTAR           | 00:29:31.557
   23 | Cal CRUTCHLOW    | CWM LCR Honda                | 00:27:38.315
   24 | Eugene LAVERTY   | Aspar MotoGP Team            | 00:08:04.096
   25 | Alex DE ANGELIS  | E-Motion IodaRacing Team     | 00:06:05.782
(25 rows)
```

4. À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.

La requête n'est pas beaucoup modifiée, seule la fonction `first_value()` est utilisée pour déterminer le temps du vainqueur, temps qui sera ensuite retranché au temps du coureur courant.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
       nom, ecurie, total_time,
       total_time - first_value(total_time)
         OVER (ORDER BY max_lap desc, total_time asc) AS difference
  FROM (SELECT no_pilote,
               sum(lap_time) over (PARTITION BY no_pilote) as total_time,
               max(no_tour) over (PARTITION BY no_pilote) as max_lap
          FROM brno_2015
       ) AS race_data
  JOIN pilotes
    ON (race_data.no_pilote = pilotes.no)
 GROUP BY nom, ecurie, max_lap, total_time
 ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

```
 r|       nom       |          ecurie      | total_time  |  difference
--+-----------------+----------------------+-------------+---------------
 1| Jorge LORENZO   | Movistar Yamaha [...]|00:42:53.042 | 00:00:00
 2| Marc MARQUEZ    | Repsol Honda Team    |00:42:57.504 | 00:00:04.462
 3| Valentino ROSSI | Movistar Yamaha [...]|00:43:03.439 | 00:00:10.397
 4| Andrea IANNONE  | Ducati Team          |00:43:06.113 | 00:00:13.071
 5| Dani PEDROSA    | Repsol Honda Team    |00:43:08.692 | 00:00:15.65
 6| Andrea DOVIZIOSO| Ducati Team          |00:43:08.767 | 00:00:15.725
 7| Bradley SMITH   | Monster Yamaha Tech 3|00:43:14.863 | 00:00:21.821
 8| Pol ESPARGARO   | Monster Yamaha Tech 3|00:43:16.282 | 00:00:23.24
 9| Aleix ESPARGARO | Team SUZUKI ECSTAR   |00:43:36.826 | 00:00:43.784
10| Danilo PETRUCCI | Octo Pramac Racing   |00:43:38.303 | 00:00:45.261
11| Yonny HERNANDEZ | Octo Pramac Racing   |00:43:43.015 | 00:00:49.973
12| Scott REDDING   | EG 0,0 Marc VDS      |00:43:43.216 | 00:00:50.174
13| Alvaro BAUTISTA | Aprilia Racing [...] |00:43:47.479 | 00:00:54.437
14| Stefan BRADL    | Aprilia Racing [...] |00:43:47.666 | 00:00:54.624
15| Loris BAZ       | Forward Racing       |00:43:53.358 | 00:01:00.316
16| Hector BARBERA  | Avintia Racing       |00:43:54.637 | 00:01:01.595
17| Nicky HAYDEN    | Aspar MotoGP Team    |00:43:55.43  | 00:01:02.388
18| Mike DI MEGLIO  | Avintia Racing       |00:43:58.986 | 00:01:05.944
19| Jack MILLER     | CWM LCR Honda        |00:44:04.449 | 00:01:11.407
20| Claudio CORTI   | Forward Racing       |00:44:43.075 | 00:01:50.033
21| Karel ABRAHAM   | AB Motoracing        |00:44:55.697 | 00:02:02.655
22| Maverick VIÑALES| Team SUZUKI ECSTAR   |00:29:31.557 | -00:13:21.485
23| Cal CRUTCHLOW   | CWM LCR Honda        |00:27:38.315 | -00:15:14.727
24| Eugene LAVERTY  | Aspar MotoGP Team    |00:08:04.096 | -00:34:48.946
25| Alex DE ANGELIS | E-Motion Ioda[...]    |00:06:05.782 | -00:36:47.26
(25 rows)
```

5. Pour chaque tour, afficher :

   - le nom du pilote ;
   - son rang dans le tour ;
   - son temps depuis le début de la course ;
   - dans le tour, la différence de temps par rapport au premier.

Pour construire cette requête, nous avons besoin d'obtenir le temps cumulé tour après tour pour chaque coureur. Nous commençons donc par écrire une première requête :

```
SELECT *,
       SUM(lap_time)
         OVER (PARTITION BY no_pilote ORDER BY no_tour) AS temps_tour_glissant
 FROM brno_2015
```

Elle retourne le résultat suivant :

```
 no_tour | no_pilote |   lap_time   | temps_tour_glissant
---------+-----------+--------------+---------------------
       1 |         4 | 00:02:02.209 | 00:02:02.209
       2 |         4 | 00:01:57.57  | 00:03:59.779
       3 |         4 | 00:01:57.021 | 00:05:56.8
       4 |         4 | 00:01:56.943 | 00:07:53.743
       5 |         4 | 00:01:57.012 | 00:09:50.755
       6 |         4 | 00:01:57.011 | 00:11:47.766
       7 |         4 | 00:01:57.313 | 00:13:45.079
```

```
       8 |          4 | 00:01:57.95   | 00:15:43.029
       9 |          4 | 00:01:57.296  | 00:17:40.325
      10 |          4 | 00:01:57.295  | 00:19:37.62
      11 |          4 | 00:01:57.185  | 00:21:34.805
      12 |          4 | 00:01:57.45   | 00:23:32.255
      13 |          4 | 00:01:57.457  | 00:25:29.712
      14 |          4 | 00:01:57.362  | 00:27:27.074
      15 |          4 | 00:01:57.482  | 00:29:24.556
      16 |          4 | 00:01:57.358  | 00:31:21.914
      17 |          4 | 00:01:57.617  | 00:33:19.531
      18 |          4 | 00:01:57.594  | 00:35:17.125
      19 |          4 | 00:01:57.412  | 00:37:14.537
      20 |          4 | 00:01:57.786  | 00:39:12.323
      21 |          4 | 00:01:58.087  | 00:41:10.41
      22 |          4 | 00:01:58.357  | 00:43:08.767
(…)
```

Cette requête de base est ensuite utilisée dans une CTE qui sera utilisée par la requête répondant à la question de départ. La colonne `temps_tour_glissant` est utilisée pour calculer le rang du pilote dans la course, est affiché et le temps cumulé du meilleur pilote est récupéré avec la fonction `first_value` :

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time)
      OVER (PARTITION BY no_pilote
            ORDER BY no_tour
            )  as temps_tour_glissant
    FROM brno_2015
    ORDER BY no_pilote, no_tour
)

SELECT no_tour, nom,
rank() OVER (PARTITION BY no_tour
             ORDER BY temps_tour_glissant ASC
            ) as place_course,
temps_tour_glissant,
temps_tour_glissant - first_value(temps_tour_glissant)
OVER (PARTITION BY no_tour
      ORDER BY temps_tour_glissant asc
    ) AS difference
FROM temps_glissant t
JOIN pilotes p ON p.no = t.no_pilote;
```

On pouvait également utiliser une simple sous-requête pour obtenir le même résultat :

```
SELECT no_tour,
  nom,
  rank()
    OVER (PARTITION BY no_tour
          ORDER BY temps_tour_glissant ASC
          ) AS place_course,
  temps_tour_glissant,
  temps_tour_glissant - first_value(temps_tour_glissant)
    OVER (PARTITION BY no_tour
          ORDER BY temps_tour_glissant asc
```

```
        ) AS difference
FROM (
  SELECT *, SUM(lap_time)
    OVER (PARTITION BY no_pilote
          ORDER BY no_tour)
          AS temps_tour_glissant
  FROM brno_2015) course
  JOIN pilotes
    ON (pilotes.no = course.no_pilote)
ORDER BY no_tour;
```

La requête fournit le résultat suivant :

```
no.|       nom        | place_c. | temps_tour_glissant |  difference
---+------------------+----------+---------------------+--------------
 1 | Jorge LORENZO    |        1 | 00:02:00.83         | 00:00:00
 1 | Marc MARQUEZ     |        2 | 00:02:01.058        | 00:00:00.228
 1 | Andrea DOVIZIOSO |        3 | 00:02:02.209        | 00:00:01.379
 1 | Valentino ROSSI  |        4 | 00:02:02.329        | 00:00:01.499
 1 | Andrea IANNONE   |        5 | 00:02:02.597        | 00:00:01.767
 1 | Bradley SMITH    |        6 | 00:02:02.861        | 00:00:02.031
 1 | Pol ESPARGARO    |        7 | 00:02:03.239        | 00:00:02.409
( ..)
 2 | Jorge LORENZO    |        1 | 00:03:57.073        | 00:00:00
 2 | Marc MARQUEZ     |        2 | 00:03:57.509        | 00:00:00.436
 2 | Valentino ROSSI  |        3 | 00:03:59.696        | 00:00:02.623
 2 | Andrea DOVIZIOSO |        4 | 00:03:59.779        | 00:00:02.706
 2 | Andrea IANNONE   |        5 | 00:03:59.9          | 00:00:02.827
 2 | Bradley SMITH    |        6 | 00:04:00.355        | 00:00:03.282
 2 | Pol ESPARGARO    |        7 | 00:04:00.87         | 00:00:03.797
 2 | Maverick VIÑALES |        8 | 00:04:01.187        | 00:00:04.114
(…)
(498 rows)
```

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?

Il est ici nécessaire de sélectionner pour chaque tour le temps du meilleur tour. On peut alors sélectionner les tours pour lequels le temps du tour est égal au meilleur temps :

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time)
        OVER (PARTITION BY no_pilote
              ORDER BY no_tour
             ) as temps_tour_glissant
    FROM brno_2015
    ORDER BY no_pilote, no_tour
),

classement_tour AS (
    SELECT no_tour, no_pilote, lap_time,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant,
```

```
    min(lap_time) OVER (PARTITION BY no_pilote) as meilleur_temps
    FROM temps_glissant
)

SELECT no_tour, nom, place_course, lap_time
FROM classement_tour t
JOIN pilotes p ON p.no = t.no_pilote
WHERE lap_time = meilleur_temps;
```

Ce qui donne le résultat suivant :

```
 no_tour |       nom        | place_course |   lap_time
---------+------------------+--------------+--------------
       4 | Jorge LORENZO    |            1 | 00:01:56.169
       4 | Marc MARQUEZ     |            2 | 00:01:56.048
       4 | Valentino ROSSI  |            3 | 00:01:56.747
       6 | Andrea IANNONE   |            5 | 00:01:56.86
       6 | Dani PEDROSA     |            7 | 00:01:56.975
       4 | Andrea DOVIZIOSO |            4 | 00:01:56.943
       3 | Bradley SMITH    |            6 | 00:01:57.25
      17 | Pol ESPARGARO    |            8 | 00:01:57.454
       4 | Aleix ESPARGARO  |           12 | 00:01:57.844
       4 | Danilo PETRUCCI  |           11 | 00:01:58.121
       9 | Yonny HERNANDEZ  |           14 | 00:01:58.53
       2 | Scott REDDING    |           14 | 00:01:57.976
       3 | Alvaro BAUTISTA  |           21 | 00:01:58.71
       3 | Stefan BRADL     |           16 | 00:01:58.38
       3 | Loris BAZ        |           19 | 00:01:58.679
       2 | Hector BARBERA   |           15 | 00:01:58.405
       2 | Nicky HAYDEN     |           16 | 00:01:58.338
       3 | Mike DI MEGLIO   |           18 | 00:01:58.943
       4 | Jack MILLER      |           22 | 00:01:59.007
       2 | Claudio CORTI    |           24 | 00:02:00.377
      14 | Karel ABRAHAM    |           23 | 00:02:01.716
       3 | Maverick VIÑALES |            8 | 00:01:57.436
       3 | Cal CRUTCHLOW    |           11 | 00:01:57.652
       3 | Eugene LAVERTY   |           20 | 00:01:58.977
       3 | Alex DE ANGELIS  |           23 | 00:01:59.257
(25 rows)
```

7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.

```
WITH nb_tour AS (
    SELECT max(no_tour) FROM brno_2015
),
temps_glissant AS (
    SELECT  no_tour, no_pilote, lap_time,
    sum(lap_time) OVER (
        PARTITION BY no_pilote
        ORDER BY no_tour
    ) as temps_tour_glissant,
    max(no_tour) OVER (PARTITION BY no_pilote) as total_tour
    FROM brno_2015
),
classement_tour AS (
```

```
    SELECT no_tour, no_pilote, lap_time, total_tour,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course
    FROM temps_glissant
)
SELECT no_pilote
FROM classement_tour t
JOIN nb_tour n ON n.max = t.total_tour
GROUP BY no_pilote
HAVING count(DISTINCT place_course) = 1;
```

Elle retourne le résultat suivant :

```
 no_pilote
-----------
        93
        99
```

8. En quelle position a terminé le coureur qui a doublé le plus de personnes. Combien de personnes a-t-il doublées ?

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time) OVER (
        PARTITION BY no_pilote
        ORDER BY no_tour
    ) as temps_tour_glissant
    FROM brno_2015
),
classement_tour AS (
    SELECT no_tour, no_pilote, lap_time,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant
    FROM temps_glissant
),
depassement  AS (
    SELECT no_pilote,
      last_value(place_course) OVER (PARTITION BY no_pilote) as rang,
    CASE
        WHEN lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course < 0
        THEN 0
        ELSE lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course
        END AS depasse
    FROM classement_tour t
)
```

```
SELECT no_pilote, rang, sum(depasse)
FROM depassement
GROUP BY no_pilote, rang
ORDER BY sum(depasse) DESC
LIMIT 1;
```

Grouping Sets

La suite de ce TP est maintenant réalisé avec la base de formation habituelle. Attention, ce TP nécessite l'emploi d'une version 9.5 ou supérieure de PostgreSQL.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY GROUPING SETS (
   (extract('year' from date_commande), code_pays),
   (extract('year' from date_commande))
 );
```

Le résultat attendu est :

```
 annee | code_pays | montant_total_commande
-------+-----------+------------------------
  2003 | DE        |                49634.24
  2003 | FR        |                10003.98
  2003 |           |                59638.22
  2008 | CA        |              1016082.18
  2008 | CN        |               801662.75
  2008 | DE        |               694787.87
  2008 | DZ        |               663045.33
  2008 | FR        |              5860607.27
  2008 | IN        |               741850.87
  2008 | PE        |              1167825.32
  2008 | RU        |               577164.50
  2008 | US        |               928661.06
  2008 |           |             12451687.15
(...)
```

10. Ajouter également le montant total des commandes depuis le début de l'activité.

L'opérateur de regroupement `ROLL UP` amène le niveau d'agrégation sans regroupement :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY ROLLUP (extract('year' from date_commande), code_pays);
```

11. Ajouter également le montant total des commandes par pays.

Cette fois, l'opérateur CUBE permet d'obtenir l'ensemble de ces informations :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

12. À partir de la requête précédente, ajouter une colonne par critère de regroupement, de type booléen, qui est positionnée à `true` lorsque le regroupement est réalisé sur l'ensemble des valeurs de la colonne.

Ces colonnes booléennes permettent d'indiquer à l'application comment gérer la présentation des résultats.

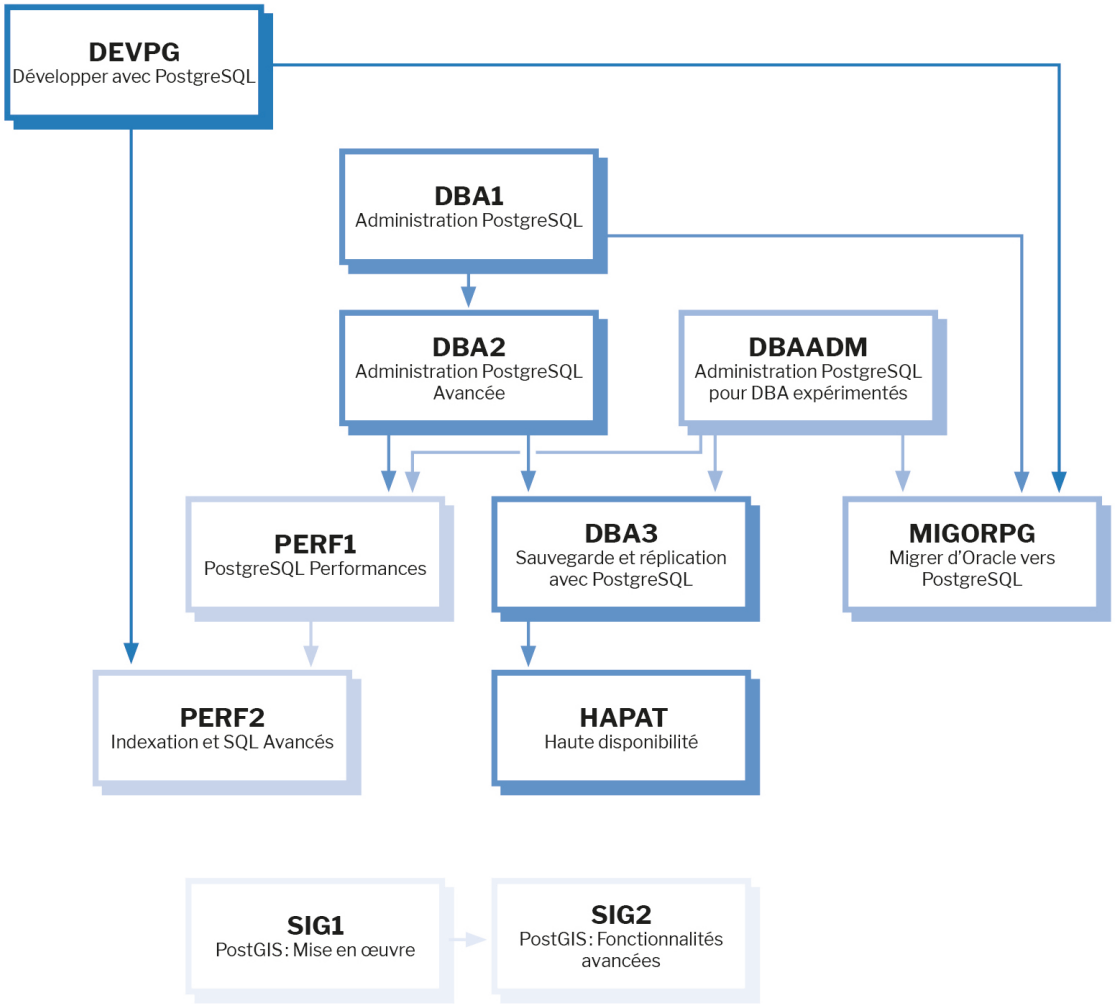
```
SELECT grouping(extract('year' from date_commande))::boolean AS g_annee,
       grouping(code_pays)::boolean AS g_pays,
       extract('year' from date_commande) AS annee,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- 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

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.