Formation DEV1

Introduction à SQL

25.09

Table des matières

Sur ce document

Formation	Formation DEV1
Titre	Introduction à SQL
Révision	25.09
ISBN	978-2-38168-154-2
PDF	https://dali.bo/dev1_pdf
EPUB	https://dali.bo/dev1_epub
HTML	https://dali.bo/dev1_html
Slides	https://dali.bo/dev1_slides

Vous trouverez en ligne les différentes versions complètes de ce document. Les solutions de TP ne figurent pas forcément dans la version imprimée, mais sont dans les versions numériques (PDF ou HTML).

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

[1] mailto:formation@dalibo.com

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 :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, Nizar Hamadi, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Cédric Martin, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cette licence interdit la réutilisation pour l'apprentissage d'une IA. Elle couvre les diapositives, les manuels eux-mêmes et les travaux pratiques.

Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

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. Il se concentre sur la lecture de données déjà présentes en base. Ceci permet aussi d'aborder 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
- SGBD : **S**ystème de **G**estion de **B**ases de **D**onnées
  - *Database Management System* (DBMS) en anglais
  - programme assurant la gestion et l'accès à une ou plusieurs bases de données
- SGBDR (ou RDBMS)
  - modèle **R**elationnel
  - 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.

Un Système de Gestion de Bases de Données (SGBD[1]), ou *Database Management System* (DBMS) en anglais, assure la gestion d'une base de données informatisée. Il permet l'accès aux données, leur modification, et assure que ces opérations en conservent la cohérence.

On voit aussi parfois le sigle SGBDR pour les bases « relationnelles » comme MySQL/MariaDB, Oracle, PostgreSQL…

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] https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_gestion_de_base_de_donn%C3%A9es

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. Cependant, des recherches cherchaient déjà à rendre indépendantes la vue logique et 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
  - à partir des années 70
  - standardisé par la norme SQL (1986)
- Modèle objet
  - structure objet
  - pas de standard
- Modèle relationnel-objet
  - le standard SQL ajoute des concepts objets

Le modèle relationnel est issu des travaux d'Edgar F. Codd[2] 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. Mathématicien de formation, Codd 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, il a mis au point le langage SQL. Ce langage a été développé dans les années 70, et est à l'origine du standard SQL, qui s'est répandu dans les années 80 et qui a rendu le modèle relationnel très populaire. La puissance de calcul du matériel de cette époque a permis la concrétisation de l'implémentation de cette couche d'abstraction entre modèle logique et stockage physique des données.

[2] https://fr.wikipedia.org/wiki/Edgar_Frank_Codd

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.

La norme du modèle relationnel a néanmoins été étendue 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*
  - clé-valeur (Redis)
  - graphe (Neo4J)
  - document (MongoDB, CouchDB)
  - orienté colonne (HBase)
  - pas de norme de langage de requête
- 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, il n'y a ni standard ni norme.

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 pour paralléliser les traitements. En contrepartie, la durabilité des données n'est pas assurée, au contraire d'une base relationnelle qui assure la durabilité dès la réponse à un `COMMIT`. La cohérence des données entre nœuds est un autre sujet. Toute amélioration d'un outil sur ces points a évidemment un impact sur les performances.

Un des avantages des technologies NoSQL 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 MapReduce[3] en soi. Simplement, le langage SQL étant déclaratif, il est conceptuellement opposé à la description fine des traitements qu'on doit réaliser avec MapReduce. C'est le travail de l'optimiseur d'effectuer ce genre d'opérations, et non celui du développeur.

Un meilleur argument des bases NoSQL est le côté *schemaless*, permettant d'enregistrer des documents sans trop se soucier de leur cohérence, de leur contenu, sans devoir réfléchir par avance à un format de données aussi rigide que peut l'être une table SQL. La flexibilité est un atout, mais il faut être conscient que le manque de cohérence ou de rigueur se paie. forcément plus tard.

[3] https://en.wikipedia.org/wiki/MapReduce

La « mode » du NoSQL est un peu retombée ces dernières années. En effet, les bases de données classiques ont profité des progrès matériels (disques NVMe et nombreux cœurs sur un serveur, par exemple) ; la parallélisation est mieux gérée : par requête, sur des secondaires, voir avec du *sharding* ; et les algorithmes se sont améliorés. Tout cela permet de procéder à des tâches de plus en plus lourdes tout en conservant les avantages d'un SGBDR : SQL standardisé, verrous bien gérés, cohérence forte des données.

Enfin, les SGBDR ont intégré quelques fonctionnalités NoSQL, en premier lieu le stockage des documents JSON, ce qui permet de les utiliser là où ils sont pertinents (schémas flexibles).

Le choix d'un outil résulte donc d'un arbitrage entre différentes priorités.

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 (tables, vues, fonctions…) en actions à réaliser sur des objets physiques (fichiers, sockets, mémoire…).

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 à 3 états
  - Vrai, Faux, `NULL` (= inconnu)

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 ».

Cette dernière valeur est aussi connue comme `NULL`, et son utilisation dans les conditions est parfois délicate.

Pour un humain ou pour l'application, `NULL` peut signifier : « inconnu », « non pertinent » ou « non encore renseigné ».

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 étran-

gères), 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 performances; 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[4] (*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 débarrasser 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.

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

1.2.8 Langage SQL

- Norme ISO 9075
  - dernière version stable : SQL :2023
- Langage déclaratif
  - on décrit le résultat et pas la façon de l'obtenir
- Traitement ensembliste
  - par opposition au traitement procédural
  - « on effectue des opérations sur des relations pour obtenir des relations »

Le langage SQL a été normalisé par l'ANSI en 1986 et est devenu une norme ISO internationale en 1987. La norme a subi plusieurs évolutions[5] dans le but d'ajouter des fonctionnalités correspondantes aux attentes de l'industrie logicielle. Parmi ces améliorations, notons l'intégration de quelques fonctionnalités objets pour le modèle relationnel-objet (SQL-99), ou les puissantes « fonctions de fenêtrage » (SQL :2003, SQL :2008). Le dernier standard est SQL :2023.

Le standard n'est suivi strictement ni par PostgreSQL ni ses concurrents. De nombreuses fonctionnalités apparaissent chez l'un ou l'autre avant d'être standardisées, si elles le sont. Chaque produit doit aussi gérer la compatibilité avec un historique parfois très ancien. PostgreSQL est le moteur qui se rapproche le plus du standard, au point de devenir une référence, mais possède néanmoins quelques écarts et de nombreuses extensions.

1.2.9 SQL est un langage

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

Hormis la syntaxe, il n'y a pas de règles strictes concernant l'écriture de requêtes SQL (majuscules, minuscules, espaces…). 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.

[5] https://fr.wikipedia.org/wiki/Structured_Query_Language#Historique

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

Cet exemple est tiré du forum postgresql.fr[6]. 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 ;
```

- ou encore celle-ci, réécrite avec pgFormatter[7] ?

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

La base de données exécutera sans souci l'un comme l'autre code, avec bien sûr le même résultat.

Sans se tenir forcément à un format précis, il convient surtout d'adopter un style lisible, et cohérent avec le code SQL existant.

[6] https://forum.postgresql.fr/viewtopic.php?id=2610
[7] https://sqlformat.darold.net/

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 l'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 parfois *hints*) qui influencent le comportement de l'optimiseur, mais PostgreSQL ne possède pas nativement ce genre de mécanisme.

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 y 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 y 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 ;
- `MERGE` : ajoute ou met à jour une ligne selon une condition ;
- `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 encadrer manuellement vos commandes avec `BEGIN ;` et `COMMIT ;` pour créer une transaction de plus d'un ordre. Le mode d'autocommit dépend en fait 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 d'autres bases de données peuvent avoir un comportement par défaut inverse.

Le `ROLLBACK` est implicite en cas de sortie brutale (plantage…).

Noter que, contrairement à d'autres bases, 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[8].

1.3.2 Liste de sélection

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

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

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

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

Les exemples ci-dessous utilisent la base de données **cave** qu'il vous faudra peut-être installer. La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

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
```

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 apparaîtront dans leur ordre physique (attention si l'ordre change !).

L'utilisation de `SELECT *` est généralement une mauvaise pratique. Le code de production ne devrait pas en contenir. De nombreux soucis de performance sont liés à trop de colonnes récupérées sans être utilisées au final (volumétrie, inhibition d'optimisations…).

```
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 une déduplication systématique des données juste avant de retourner les résultats de la requête, ce qui va souvent 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)
  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(s.contenance * s.nombre) || ' litres'
  FROM stock AS s
  JOIN contenant 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 utilisées, 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 (noter la présence du fuseau horaire) :

```
SELECT now();
              now
-------------------------------
  2025-08-04 18:03:08.396493+02
```

Affichage de l'âge du 1er janvier 2000 :

```
SELECT age(date '2000-01-01');
          age
------------------------
 25 years 7 mons 3 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 s'agit souvent d'une table physique ou d'une vue, mais ce peut être aussi une vue matérialisée ou 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 précisées ici, généralement pour les associer avec 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.

Les alias sont très utiles pour la lisibilité, notamment s'il y a beaucoup de tables ou si leurs noms sont longs. Ils sont même vitaux quand on utilise la même table plusieurs fois dans une requête. N'hésitez pas à rajouter des alias de table dès qu'il y en a plus d'une.

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 définit 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);
```

1.3.10 Nommage des objets

- Noms d'objets par défaut en minuscules
  - `a-z`, `_` et chiffres, si possible
  - `Nom_Objet` devient `nom_objet`
- Au besoin, 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.

Même si c'est parfaitement possible, il est préférable de ne pas utiliser d'accents ou de caractères exotiques dans les noms des objets, car il y a parfois des surprises en fonction des outils et systèmes

d'exploitation utilisés. Les noms de tables et de champs sont généralement masqués par les outils de requêtage, ou peuvent être aliasés pour la présentation.

Il est donc conseillé d'utiliser une notation des objets ne comprenant que des caractères minuscules, au besoin des chiffres et `_` à la place d'un espace.

Dans les données par contre, rien ne vous interdit d'utiliser tout ce que permet l'encodage Unicode :

```
SELECT nom AS "Nom", prenom as "Prénom"
  FROM eleves ;
```

```
      Nom       |   Prénom
----------------+-------------
 L'Huillier     | Jean-Pierre
 Sigurðardóttir | Þór
 Einarsson      | Björn
```

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 de différence `!=` peut aussi s'écrire `<>`.

L'opérateur de négation `NOT` permet d'inverser la véracité des prédicats. Un prédicat faux retournera donc vrai et inversement. 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.

D'ailleurs, au moindre doute, n'hésitez pas à ajouter des 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.

Sans clause `ORDER BY`, aucun ordre des lignes n'est garanti ! PostgreSQL les renverra dans l'ordre où il les trouvera, ce qui dépend de la méthode choisie par l'optimiseur, des emplacements physiques sur le disques et des manipulations nécessaires, et cet ordre peut varier entre deux exécutions !

L'expression de tri 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 très complexe, par exemple avec une fonction d'un calcul sur plusieurs colonnes.

L'ordre de tri peut être inversé 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.

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. Rappelons que le standard SQL ne garantit 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 contrainte et cohérente. Les valeurs possibles d'une donnée vont dépendre de son type. 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 a une représentation textuelle, une représentation en mémoire et sur disque. Ainsi, un `integer` est représenté en interne 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 pos-

sible 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`, `float`
  - `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

La norme 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 excédera cette taille ne sera pas prise en compte et générera 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, qui peuvent compter en *octets*. En effet, de nombreux caractères peuvent nécessiter plusieurs octets.

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 -32 768 et +32 767. Le type `integer` ou `int`, codé sur 4 octets, permet de stocker des valeurs comprises entre -2 147 483 648 et +2 147 483 647. Enfin, le type `bigint`, codé sur 8 octets, permet de stocker des valeurs comprises entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807. 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 : `.`
- Hexadécimal, octal, binaire possibles
- Exemples :
  - `42`
  - `3.14159`, `0.5`, `-.005`
  - `1.0e6`, `-314e-2`
  - `1_000_000` (v16+)
  - `0xFF`, `0o377`, `0b11111111`
- Conversion de type :

```
SELECT REAL '1.23', cast ('1.23' AS real), '1.23'::float4 ;
```

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 `+` ou `-` en avant n'est pas forcément considéré comme faisant partie de la constante ; il est un opérateur appliqué à la constante. PostgreSQL accepte des `_` intercalaires pour la lisibilité depuis sa version 16.

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`.

Des notations hexadécimales, octales, binaires sont possibles :

```
SELECT 0xFF AS hexa, 0o377 AS octal, 0b1111_1111 AS binaire ;
```

```
 hexa | octal | binaire
------+-------+---------
  255 |   255 |     255
```

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`) de plusieurs manières différentes :

```
SELECT '1.23', 1.23 , REAL '1.23', cast ('1.23' AS real), '1.23'::float4 \gdesc
```

```
  Column  |  Type
----------+---------
```

```
?column? | text
?column? | numeric
float4   | real
float4   | real
float4   | real
```

La première syntaxe ne fonctionne que pour des constantes. La syntaxe `::` se rencontre fréquemment et est très pratique, mais est propre à PostgreSQL. La norme SQL conseille l'opérateur `cast`.

— Documentation officielle :
— Conversions de type[9]
— Constantes d'autres types[10]

1.4.9 Booléens

— `boolean`
— 3 valeurs possibles
  — `TRUE`
  — `FALSE`
  — `NULL` (valeur absente/non pertinente)

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;
```

[9] https://docs.postgresql.fr/current/sql-expressions.html#SQL-SYNTAX-TYPE-CASTS
[10] https://docs.postgresql.fr/current/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC

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 6 h 00, heure d'été, à Paris, il est 21 h 00 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 :

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

```
          timezone
----------------------------
 2017-08-29 10:20:10.123321
```

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 abrégé : `'CEST'`
  - décalage : `'+02'`

Le paramètre `timezone` du fichier de configuration `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 métropolitaine 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
```

ou encore (`AT LOCAL TIME` est disponible depuis PostgreSQL 17) :

```
SHOW timezone;

   TimeZone
--------------
 Europe/Paris

SELECT '2024-11-26 00:00:00 +11' AT LOCAL TIME ; -- minuit à Nouméas

        time
---------------------
 2024-11-26 01:00:00
```

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
```

ou encore :

```
SELECT '2023-10-23 6:30:00+02'::timestamptz AT TIME ZONE 'Europe/London';

      timezone
---------------------
 2023-10-23 05:30:00
```

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[11]).

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

1.4.16 JSON

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

Les types `json` et `jsonb` permettent de stocker des documents JSON. Ces deux types permettent de vérifier la structure du document JSON ainsi que des fonctions de manipulations spécifiques (voir la documentation officielle[12]).

Sous PostgreSQL, 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
- Incrémentés :
  - principe de l'« autoincrément »
  - `serial`
  - `smallserial`
  - `bigserial`
  - équivalent à un type entier associé à une séquence et avec une valeur par défaut
  - 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 cependant préférable de passer par un type `IDENTITY` que par ces types dérivés.

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

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

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

1.4.18 Types additionnels hors standard SQL

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

Les types standards ne sont pas toujours suffisants pour représenter certaines données. Comme tous ses concurrents, PostgreSQL propose des types de données supplémentaires 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 (de nombres, de dates, de chaînes…) 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 `json` a été complété 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 `hstore` est un type clé/valeur qui par rapport à JSON a l'intérêt de la simplicité.

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

— Modèle relationnel : extrêmement efficace
— 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 cœur des performances

— Auteur : Markus Winand

— Éditeur : auto-édité
— ISBN : 978-3950307832
— site Internet[13]

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 !

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

1.6 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s10_solutions.

Requêtes simples :

> 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 ?

Requêtes sur la base tpc :

Pour les questions suivantes, il faudra se connecter à la base de données **tpc**. La base **tpc** (dump de 31 Mo, pour 267 Mo sur le disque au final) et ses utilisateurs peuvent être installés comme suit :

```
curl -kL https://dali.bo/tp_tpc -o /tmp/tpc.dump
curl -kL https://dali.bo/tp_tpc_roles -o /tmp/tpc_roles.sql
# Exécuter le script de création des rôles
psql < /tmp/tpc_roles.sql
# Création de la base
createdb --owner tpc_owner tpc
# L'erreur sur un schéma 'public' existant est normale
pg_restore -d tpc /tmp/tpc.dump
```

Les mots de passe sont dans le script `/tmp/tpc_roles.sql`. Pour vous connecter :

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

Le schéma suivant montre les différentes tables de la base :

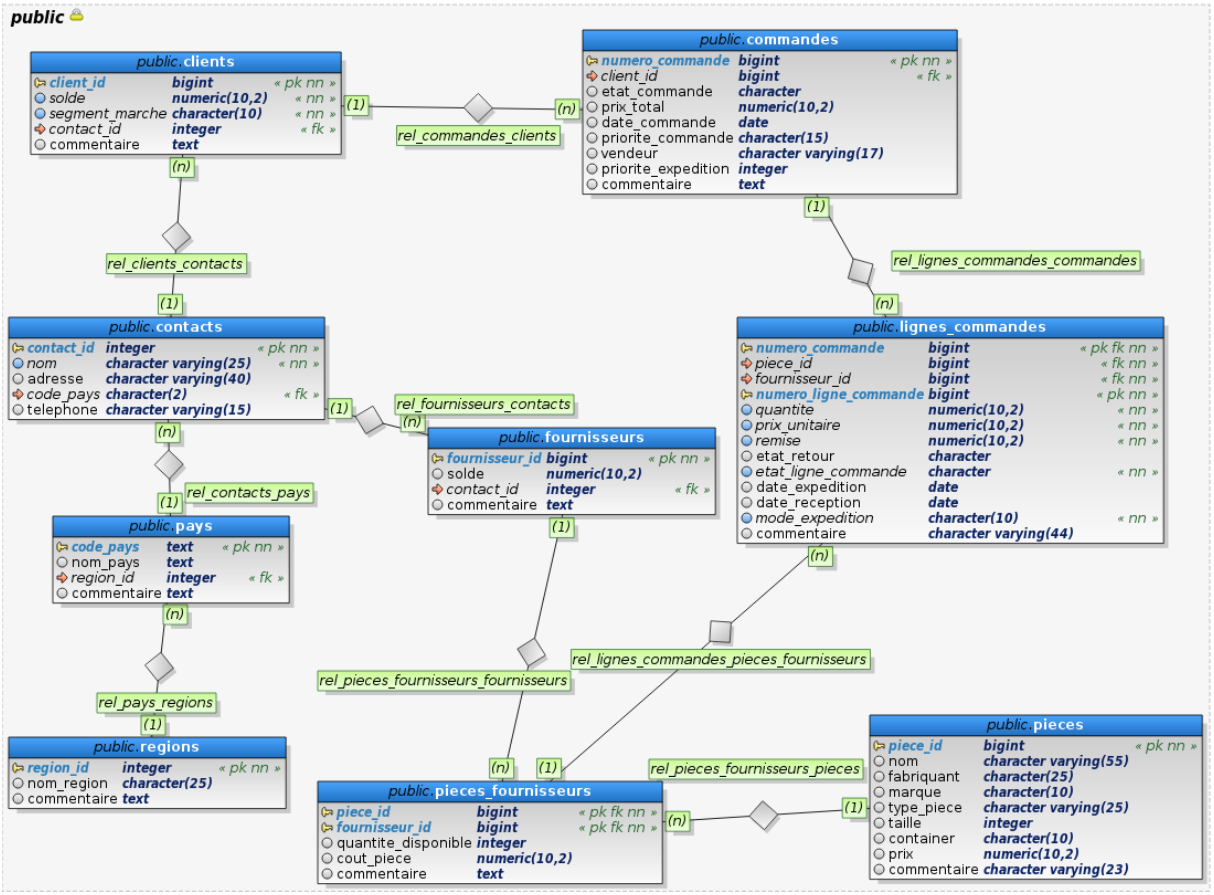


FIGURE 1/ .1 – Schéma base tpc

Avec `psql`, pour changer de base, vous pouvez lancer la méta-commande suivante :

```
\c tpc
```

> 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 (table `lignes_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
(…)
             132 | Client18152
```

> 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 son numéro, sa date ainsi que son âge.

Sortie attendue :

```
numero_commande | date_commande |            age
----------------+---------------+---------------------------
          54274 | 2011-01-01    | 5336 days 10:09:01.6222
(…)
```

2/ Types de base

- PostgreSQL offre un système de typage complet
  - types standards
  - types avancés propres à PostgreSQL

2.0.1 Préambule

- SQL possède un typage fort
  - le type utilisé décrit la donnée manipulée
  - garantit l'intégrité des données
  - primordial au niveau fonctionnel
  - garantit les performances

2.0.2 Menu

- Qu'est-ce qu'un type ?
- Les types SQL standards
  - numériques
  - temporels
  - textuels et binaires
- Les types avancés de PostgreSQL

2.0.3 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

2.1 LES TYPES DE DONNÉES

- Qu'est-ce qu'un type ?
- Représentation physique
- Impacts sur l'intégrité
- Impacts fonctionnels

2.1.1 Qu'est-ce qu'un type ?

- Un type définit :
  - les valeurs que peut prendre une donnée
  - les opérateurs applicables à cette donnée

2.1.2 Impact sur les performances

- Choisir le bon type pour :
  - optimiser les performances
  - optimiser le stockage

2.1.3 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 :

```
CREATE TABLE test_date (dt date);

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');

                                      ^
```

```
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');

INSERT INTO test_date VALUES ('2015-07-17');
```

2.1.4 Impacts fonctionnels

- Un type de données offre des opérateurs spécifiques :
  - comparaison
  - manipulation
- Exemple : une date est-elle comprise entre deux dates données ?

2.2 TYPES NUMÉRIQUES

- Entiers
- Flottants
- Précision fixée

2.2.1 Types numériques : entiers

- 3 types entiers :
  - `smallint` : 2 octets
  - `integer` : 4 octets
  - `bigint` : 8 octets
- Valeur exacte
- Signé
- Utilisation :
  - véritable entier
  - clé technique

Les `smallint` couvrent des valeurs de -32 768 à +32 767. Attention à ne réserver leur utilisation qu'à ce qui ne dépassera pas cette plage.

Les `integer` vont de -2,1 à +2,1 milliards environ. Ce n'est pas toujours suffisant. Les `bigint` sur 8 octets vont jusque environ 9,2 10^{18}.

- Documentation officielle : Types entiers[1]

2.2.2 Types numériques : flottants

- 2 types flottants :
  - `real` (= `float4`)
  - `double precision` (= `float8`)
- Données numériques « floues »
  - valeurs non exactes
- Utilisation :
  - stockage des données issues de capteurs

[1] https://docs.postgresql.fr/current/datatype-numeric.html#DATATYPE-INT

Ces types n'ont pas de bornes mais une précision limitée.

`real` (4 octets) peut aller de 10^{-37} à 10^{37} avec une précision d'au moins six chiffres décimaux. Le type `double precision` a une étendue de 10^{-307} à 10^{308} avec une précision d'au moins quinze chiffres.

— Documentation officielle : Types flottants[2]

2.2.3 Types numériques : numeric

— 1 type
  — `numeric(…, …)`
— Type exact
  — mais calcul lent
— Précision choisie : totale, partie décimale
— Utilisation :
  — données financières
  — calculs exacts

Le type `numeric` est destiné aux calculs précis (financiers ou scientifiques par exemple) avec une précision arbitraire, avec une certaine lenteur et une consommation mémoire ou d'espace de stockage potentiellement plus grande que les types flottants.

— Documentation officielle : Nombres à précision arbitraire[3]

2.2.4 Opérations sur les numériques

— Indexable : `>`, `>=`, `=`, `<=`, `<`
— `+`, `-`, `/`, `*`, modulo (`%`), puissance (`^`)
— Pour les entiers :
  — `AND`, `OR`, `XOR` (`&`, `|`, `#`)
  — décalage de bits (*shifting*) : `>>`, `<<`
— Attention aux conversions (*casts*) / promotions !

Toutes les colonnes de types numériques sont indexables avec des index standards B-tree, permettant la recherche avec les opérateurs d'égalité, supérieur ou inférieur.

Pour les entiers, il est possible de réaliser des opérations bit-à-bit :

```
SELECT 2 | 4;
```

[2] https://docs.postgresql.fr/current/datatype-numeric.html#DATATYPE-FLOAT
[3] https://docs.postgresql.fr/current/datatype-numeric.html#DATATYPE-NUMERIC-DECIMAL

```
 ?column?
----------
        6

SELECT 7 & 3;

 ?column?
----------
        3
```

Il faut toutefois être vigilant face aux opérations de conversion de type implicites et celles de promotion de type numérique. En effet un index portant sur un champ numérique ne sera compatible qu'avec ce type.

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, comme le montrent les plans d'exécution :

```
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)

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)
```

Cela peut paraître contre-intuitif, mais la conversion est réalisée 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 qu'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.

2.2.5 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[4] 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 25 000 fois la valeur `0.4`, stockée soit en `float` soit en `numeric` :

```
CREATE TABLE t_float AS (
    SELECT 0.04::float AS cf,
           0.04::numeric AS cn
    FROM generate_series(1, 25000)
);

SELECT sum(cn), sum(cf) FROM t_float ;

   sum    |       sum
----------+-----------------
 1000.00 | 999.99999999967
```

Si l'on considère la performance de ces opérations, on remarque des temps d'exécution bien différents :

```
SELECT sum(cn) FROM t_float ;

   sum
---------
 1000.00

Temps : 10,611 ms

SELECT sum(cf) FROM t_float ;

       sum
-----------------
 999.99999999967

Temps : 6,434 ms
```

Pour aller (beaucoup) plus loin, le document suivant détaille le comportement des flottants selon le standard :

[4] https://fr.wikipedia.org/wiki/IEEE_754

— *What Every Computer Scientist Should Know About Floating-Point Arithmetic*[5]

[5] https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

2.3 TYPES TEMPORELS

- Date
- Date & heure
  - …avec ou sans fuseau

2.3.1 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
```

2.3.2 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
```

2.3.3 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`.

2.3.4 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
  - calculs tiennent compte des heures d'été aussi
  - 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

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). Même si un seul fuseau horaire est utilisé, il permet de s'épargner le calcul des heures d'été et d'hiver !

Les deux types occupent 8 octets, le fuseau horaire ne coûte donc pas plus cher à stocker.

2.3.5 Types temporels : interval

- `interval`
  - représente une durée
- Utilisation :
  - exprimer une durée
  - dans une requête, pour modifier une date/heure existante

2.3.6 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 un seul fuseau horaire est géré, les clients ne verront pas la différence. Si en revanche les besoins évoluent, il sera beaucoup plus simple de gérer les différents fuseaux 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 `range` dès lors que l'on souhaite stocker un couple « date de début/date de fin ». Nous aurons l'occasion de revenir sur ces types.

2.4 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 taille fixe), qu'on ne renconte plus guère que dans de très vieilles bases, et qui n'a même pas d'avantage de performance.)

2.4.1 Types chaînes : caractères

- `varchar(_n_)`, `text`
- Représentent une chaîne de caractères
- Valident l'encodage
- Valident la longueur maximale de la chaîne
  - la taille est une contrainte
- Utilisation :
  - stocker des chaînes de caractères non binaires

Un champ de type `varchar(10)` stocke une chaîne d'au plus 10 caractères. Une chaîne plus grande sera rejetée, et non tronquée (sauf si ce sont des espaces à la fin, c'est une exigence du standard). 10 est ici une limite, une plus petite chaîne consommera moins de mémoire et d'espace disque.

Il faut considérer la longueur d'une chaîne comme une contrainte fonctionnelle. Si la limite n'est pas vraiment définie (champ commentaire d'un blog, ou même un champ de nom de famille…), préférez un type `text` à une limite arbitraire.

`text` ne figure pas dans le standard SQL mais se rencontre fréquemment. C'est un équivalent de `varchar` sans limite de taille. La limite de taille théorique de 1 Go sera en pratique plus basse, mais un champ `text` de 200 Mo, par exemple, est possible. Ce n'est pas forcément une bonne idée.

2.4.2 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.

Voir le module de formation sur les types avancés[6].

2.4.3 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[7] (standard Unicode). PostgreSQL utilise par défaut UTF8 et il est chaudement conseillé de ne pas changer cela. De vieilles bases peuvent avoir conservé un encodage plus ancien.

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.

[6] https://dali.bo/s22_html#bytea
[7] https://fr.wikipedia.org/wiki/UTF-8

```
\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 renvoient 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
```

2.4.4 Collation & sources

Source des collations :

- le système : installations séparées nécessaires, différences entre OS
- 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[8] et la documentation officielle[9].

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         | …
 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         | …
```

[8] https://blog.2ndquadrant.com/icu-support-postgresql-10/
[9] https://docs.postgresql.fr/current/collation.html

2.5 TYPES AVANCÉS

- PostgreSQL propose des types plus avancés
  - faiblement structurés (JSON)
  - intervalle
  - géométriques
  - tableaux

2.5.1 Types faiblement structurés

- PostgreSQL propose plusieurs types faiblement structurés :
  - `hstore` : clé/valeur historique
  - XML : stockage, sans plus
  - JSON

Des types faiblement structurés peuvent apporter une souplesse que ne possède pas un schéma de base de données, par nature assez rigide.

Pour un type clé/valeur simple, `hstore` peut parfaitement faire l'affaire. Mais PostgreSQL sait manier du JSON depuis des années, qui est plus puissant et plus répandu.

Pour les détails, voir le module de formation sur les types avancés[10].

2.5.2 JSON

- `json`
  - stockage sous forme d'une chaîne de caractère
  - valide un document JSON sans modification
- `jsonb`
  - à préférer
  - stockage binaire optimisé
  - beaucoup plus de fonctions (dont JSONPath)

Pour manipuler du JSON dans PostgreSQL, préférer le type `jsonb`, compressé et offrant de nombreuses fonctionnalités et **possibilités d'indexation**. Le type `json` est historique et plus dédié à l'archivage à l'identique de documents JSON.

[10] https://dali.bo/s22_html

Pour les détails, voir le module de formation sur les types avancés[11].

[11] https://dali.bo/s22_html#json

2.6 TYPES INTERVALLE DE VALEURS

- Représentation d'intervalle
  - utilisable avec plusieurs types : entiers, dates, timestamps, etc.
  - contrainte d'exclusion

2.6.1 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 et 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`).

2.6.2 Range : 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 RANGETYPES-DEFINING[12].

```
-- 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;
```

[12] https://docs.postgresql.fr/current/rangetypes.html#RANGETYPES-DEFINING

```
      timerange
---------------------
 [11:10:00,23:00:00]
```

2.6.3 Range & contrainte 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`.

```
\x on
\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)).
```

2.7 TYPES GÉOMÉTRIQUES

- Plusieurs types natifs 2D :
  - `point`, `line`, `lseg` (segment),
  - `polygon`, `circle`, `box`, `path`
- Utilisation :
  - stockage de géométries simples, sans référentiel de projection
- Pour la géographie :
  - extension PostGIS

Certaines applications peuvent profiter des types géométriques. Voir la documentation officielle :

- Types géométriques[13]
- Fonctions et opérateurs de géométrie[14]

Pour de la cartographie, l'extension PostGIS[15] est la référence. Une fois installée, elle apporte de nombreux types et fonctions dédiés.

[13] https://docs.postgresql.fr/17/datatype-geometric.html
[14] https://docs.postgresql.fr/current/functions-geometry.html#FUNCTIONS-GEOMETRY-OP-TABLE
[15] https://postgis.net/

2.8 TYPES UTILISATEURS

- Plusieurs types définissables par l'utilisateur
  - types composites
  - domaines
  - enums

2.8.1 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.

2.8.2 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[16]

[16] https://docs.postgresql.fr/current/datatype-enum.html

2.8.3 Conclusion

— Des types de base assez simples
— Le choix du type est capital

3/ Création d’objet et mises à jour

3.1 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.

3.1.1 Menu

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

3.1.2 Objectifs

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

3.2 DDL

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

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

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

Si un objet dépendant de l'objet à supprimer a lui aussi une dépendance, sa dépendance sera également supprimée. Ainsi de suite jusqu'à la dernière dépendance.

3.2.5 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`
  - existe par défaut dans une base PostgreSQL
  - ouvert en écriture par défaut ! (< v15)
  - parfois supprimé pour la sécurité

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 purement logique, suivant un schéma interne à l'organisation. 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.

Jusque PostgreSQL 14, tout le monde a le droit d'y créer des objets. Cela posait des problèmes de sécurité, donc ce droit était souvent révoqué, ou le schéma `public` supprimé.
À partir de PostgreSQL 15, le droit d'écriture dans `public` doit être donné explicitement.

Il existe des schémas système masqués, par exemple pour des tables temporaires ou les tables système (schéma `pg_catalog`).

3.2.6 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 changer le propriétaire d'un schéma.

Enfin, l'ordre `DROP SCHEMA` permet de supprimer un schéma si il est vide. 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
```

3.2.7 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`.

Il est possible de vérifier la configuration de la variable `search_path` à l'aide de la commande `SHOW` :

```
SHOW search_path;

  search_path
----------------
 "$user",public
```

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;
```

3.2.8 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 séries 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.

3.2.9 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
```

3.2.10 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 modifiées 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
```

3.2.11 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).

3.2.12 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
- Ne pas utiliser pour une numérotation « sans trou » !

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. C'est cependant nécessaire, notamment pour des raisons de performance.

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

```
SELECT currval('testseq');

 currval
---------
      15

SELECT nextval('testseq');

 nextval
---------
      16

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
```

Une séquence n'est pas le bon outil s'il vous faut générer des suites de nombres « sans trou », par exemple des numéros de factures. Une séquence est conçue d'abord pour livrer des numéros uniques, et certaines valeurs peuvent être « perdues ».
Une suite unique réclame des techniques plus complexes nécessitant un verrouillage plus lourd.

3.2.13 Type SERIAL

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

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

PostgreSQL possède `identity` à partir de PostgreSQL 10. Il était déjà possible d'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 la 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` :

```
\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 :

```
\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.

3.2.14 Domaines

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

Un domaine est un type standard (numérique, texte…) auquel ont été ajoutées des contraintes particulières.

Les domaines sont utiles pour ne pas définir les mêmes contraintes sur plusieurs colonnes. La maintenance en est ainsi facilitée.

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` :

Commençons par le domaine et une table d'exemple :

```
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 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"
```

L'erreur ci-dessus est logique vu qu'on ne peut avoir qu'un entier strictement positif.

```
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 1000);
INSERT INTO employes (nom, paye) VALUES ('Bertrand', NULL);
```

Tous les employés doivent avoir un salaire. Il faut donc modifier la contrainte pour s'assurer qu'aucune valeur `NULL` (vide) ne soit saisie

```
ALTER DOMAIN salaire SET NOT NULL;

ERROR:  column "paye" of table "employes" contains null values
```

En effet, une ligne avec `NULL` est déjà présente, il faut la corriger pour pouvoir ajouter la contrainte.

```
UPDATE employes SET paye=1500 WHERE nom='Bertrand';

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 donc bien vérifiée, et la ligne avec `NULL` rejetée.

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` pour détruire aussi les objets qui dépendent du domaine.

Soyez très prudent en supprimant des objets avec `CASCADE` !

```
DROP DOMAIN salaire CASCADE;

NOTICE:  drop cascades to table employes column paye
DROP DOMAIN
```

Le domaine a été supprimée ainsi que toutes les colonnes de ce type :

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

Exemples : 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 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 »
```

3.2.15 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.

3.2.16 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.

3.2.17 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.

3.2.18 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. Quelle que soit la colonne, il est ensuite possible d'ajouter des contraintes.

3.2.19 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)
```

3.2.20 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 habituelle 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
);
```

3.2.21 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).

3.2.22 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. L'utilisation de la commande `ALTER TABLE` sur un serveur en production doit donc souvent passer par une opération de maintenance planifiée.

Certaines opérations nécessitent de vérifier que les données satisfassent les nouvelles contraintes. C'est évident lors de l'ajout d'une clé primaire ou d'une contrainte `NOT NULL`, par exemple. Or, relire

une grosse table peut donc être très coûteux ! Dans certains cas, PostgreSQL sait éviter un contrôle inutile, par exemple lors d'un passage de `varchar(10)` à `varchar(20)`.

Certaines opérations nécessitent une réécriture de la table. Par exemple, convertir une colonne de type `varchar(5)` vers le type `int` impose une réécriture de la table car il n'y a pas de compatibilité binaire entre les deux types.

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 la vérification des données, voire de la réécriture de la table.

3.2.23 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.

3.2.24 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 de 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.

3.2.25 Clé primaire d'une table

- Identifie une ligne de manière unique
- Une seule clé primaire par table
- Une ou plusieurs colonnes
- À choisir parmi les clés candidates
  - clé naturelle ou artificielle (technique, invisible)
  - les autres clés possibles peuvent être `UNIQUE`
- Index automatique

Une « clé primaire » permet d'identifier une ligne de façon unique.

Il n'existe qu'une seule clé primaire par table.

Une clé primaire exige que toutes les valeurs de la ou des colonnes qui composent cette clé soient uniques et non nulles. La clé 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. Cette clé peut être « naturelle » et visible de l'utilisateur, ou purement technique et non visible. Le débat entre les deux modélisations a longtemps fait rage.

Clé primaire naturelle :

Par exemple, une table des factures peut avoir comme clé primaire « naturelle » le numéro de la facture. C'est le plus intuitif.

De nombreuses applications utilisent des « clés naturelles », par exemple un numéro de commande ou de Sécurité Sociale. Ces clés peuvent même être composées de plusieurs champs.

Cela peut poser des soucis techniques. Par exemple, il y a des contraintes légales sur l'incrémentation des numéros de facture qui obligent à le générer tard dans le processus. Enregistrer une facture incomplète sans son numéro devient compliqué. Un numéro provisoire est possible, mais le changer implique de modifier aussi toutes les tables qui y font référence.

De manière générale, on évitera toujours de modifier une clé primaire, car elle sert d'identifiant vers la ligne dans d'autres tables, et une modification serait à répercuter dans toutes ces tables !

Liée à notre table des factures, une table des lignes de facture aurait alors comme clé primaire composée le numéro de la facture et le numéro de la ligne. Cela fonctionne, mais une clé composée, d'ailleurs

souvent de type texte, est moins performante pour les jointures qu'un champ monocolonne numérique.

Clé primaire technique :

> Les bonnes pratiques conseillent plutôt d'identifier chaque ligne de chaque table par une clé technique **monocolonne** et **numérique**, de valeur arbitraire et sans aucune signification métier.

Cette *surrogate keys* est donc une clé primaire destinée à résoudre les soucis techniques des clés primaires naturelles.

L'utilisateur final de l'application ne la verra pas.

Cette clé technique doit être générée avec une séquence, ou un UUID (quasi-aléatoire). Ce qui compte est l'unicité.

La clé technique n'a pas à être modifiée puisque sa valeur n'a pas de sens fonctionnel.

Les jointures se font alors sur cet unique champ numérique, de manière efficace.

Les clés « naturelles » restent présentes sous forme d'un champ d'une table, avec une contrainte d'unicité. Si cette clé naturelle doit être modifiée pour une raison ou une autre, il suffit de changer la valeur dans le champ de la table, et il n'y a aucune modification à faire dans les tables possédant une clé étrangère vers cette table.

Dans notre exemple, la table des factures porterait une clé primaire numérique `facture_id` arbitraire, et un champ `numero_facture`, unique, qui, lui, peut être modifié ou être temporairement vide. La table des lignes de facture porterait une clé primaire technique `facture_ligne_id` sans lien, et une clé étrangère (non composée) reprenant `facture_id`, et un champ unique `numero_ligne`.

Exemple 2 :

Pour une assurance ou un garagiste, une table des véhicules ne peut avoir pour clé primaire la plaque d'immatriculation : elle peut changer, elle peut être inconnue, provisoire, absente, voire fausse.

Un identifiant technique est largement préférable.

Exemple 3 :

Une table des clients ne peut pas porter de clé primaire naturelle liée aux nom, prénom, date de naissance… à cause des nombreuses homonymies, et des changements et corrections d'état-civil possibles.

Un code client est plus envisageable, mais il pourrait changer aussi (migration de logiciel, fusion de bases clients…). Le code client peut devenir une clé unique dans une table des clients portant une clé purement technique.

Exemple 4 :

Une table de personnes identifiées par le numéro de Sécurité Sociale ne peut utiliser ce dernier comme clé naturelle : valeur parfois absente ou inconnue, changement possible, voire doublons (!), sans parler de la confidentialité.

Un code de personne propre à l'application est plus pertinent.

Là encore, on créera plutôt une clé technique, et le code de personne et le code de Sécurité Sociale sont juste des champs uniques.

Exemple 5 :

Une table des commandes clients porte une référence vers une table des adresses. Cette table des adresses doit impérativement porter une clé technique : il n'y a guère de sens à créer un « code adresse », et surtout les clients changent régulièrement d'adresse.

Un changement d'adresse d'un client consiste à créer une nouvelle ligne dans la table des adresses et à modifier l'identifiant d'adresse dans la table des clients. Les anciennes commandes et factures doivent en effet conserver l'identifiant des anciennes adresses.

Exemple 6 :

La table des adresses possède un champ renvoyant à une table des pays.

Un code pays ISO est ce qui se rapproche le plus d'une clé naturelle acceptable comme clé technique : courte (`FR` , `DE`), normalisée et sans risque de changement (le code reste identique et n'est pas recyclé).

Index :

La création d'une clé primaire crée implicitement un index sur le champ, pour des raisons de performance.

3.2.26 Déclaration d'une clé primaire

Construction :

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

- Séquence
- `serial`
- `GENERATED ALWAYS BY IDENTITY`
- UUID

Exemple avec clé technique manuelle :

```
CREATE TABLE region
  (
    id       int   PRIMARY KEY,
    libelle  text  NOT NULL  UNIQUE
  );

INSERT INTO region VALUES (1, 'Alsace');
INSERT INTO region VALUES (2, 'Île-de-France');
```

La clé primaire est forcément `NOT NULL` : ceci va être rejeté :

```
INSERT INTO region VALUES (NULL, 'Corse');

ERROR:  null value in column "id" of relation "region" violates not-null constraint
DÉTAIL : Failing row contains (null, Corse).

INSERT INTO region VALUES (1, 'Corse');

ERROR:  duplicate key value violates unique constraint "region_pkey"
DÉTAIL : Key (id)=(1) already exists.

TABLE region ;

 id |    libelle
----+---------------
  1 | Alsace
  2 | Île de France
```

Exemple avec séquence :

La séquence génère des identifiants que l'on peut utiliser :

```
CREATE SEQUENCE departement_seq AS int ;

CREATE TABLE departement
  (
    id       int   PRIMARY KEY,
    libelle  text  NOT NULL UNIQUE,
    code     varchar(3) NOT NULL UNIQUE,
    region_id int NOT NULL REFERENCES region (id)
  );

INSERT INTO departement
SELECT nextval ('departement_seq'), 'Bas-Rhin', '67', 1 ;

INSERT INTO departement
VALUES ( nextval ('departement_seq'),  'Haut-Rhin', '68', 1 ),
       ( nextval ('departement_seq'),  'Paris',     '75', 2 ) ;
```

Noter que cette dernière erreur va « consommer » un numéro de séquence, ce qui en fait n'a pas d'importance pour une clé technique.

```
INSERT INTO departement
SELECT nextval ('departement_seq'), 'Yvelines', '78', null ;

ERROR:  null value in column "region_id" of relation "departement" violates not-null
 ↪  constraint
DETAIL : Failing row contains (4, Yvelines, 78, null).

WITH nouveaudept AS (
  INSERT INTO departement
  SELECT nextval ('departement_seq'), 'Yvelines', '78', 2
  RETURNING *)
SELECT * FROM nouveaudept ;

 id | libelle  | code | region_id
----+----------+------+-----------
   | Yvelines | 78   |         2
```

```
TABLE departement ;
```

```
 id |  libelle   | code | region_id
----+------------+------+-----------
  1 | Bas-Rhin   | 67   |         1
  2 | Haut-Rhin  | 68   |         1
  3 | Paris      | 75   |         2
  5 | Yvelines   | 78   |         2
```

Exemples avec serial :

Noter que `serial` est considéré comme un type (il existe aussi `bigserial`). Il y a création d'une séquence en arrière-plan, utilisée de manière transparente.

```
CREATE TABLE ville
  (
    id       serial   PRIMARY KEY,
    libelle  text     NOT NULL,
    departement_id int NOT NULL REFERENCES departement
  );

INSERT INTO ville (libelle, departement_id) VALUES ('Strasbourg',1);
INSERT INTO ville (libelle, departement_id) VALUES ('Paris',3);
```

Interférer avec l'autoincrémentation est une très mauvaise idée :

```
INSERT INTO ville VALUES (3, 'Mulhouse', 2);
INSERT INTO ville (libelle, departement_id) VALUES ('Sélestat',1);

ERROR:  duplicate key value violates unique constraint "ville_pkey"
DETAIL : Key (id)=(3) already exists.
```

Exemple avec IDENTITY :

```
CREATE TABLE ville2
  (
    id       int   GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    libelle  text  NOT NULL,
    departement_id int NOT NULL REFERENCES departement
  );
```

Le maniement est identique au `serial`. Il faut préférer l'identité pour des raisons de respect de standard SQL et à cause de quelques spécificités dans la gestion automatique des séquences sous-jacentes.

Exemple avec une clé composée :

Nous verrons plus loin un exemple avec une clé primaire composée, qui doit se déclarer alors après les colonnes :

```
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
  );
```

3.2.27 Contrainte d'unicité

- Garantit l'unicité des valeurs d'une ou plusieurs colonnes
- 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.

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 juste l'index `UNIQUE` peut malgré tout être référencée par une clé étrangère.

Les contraintes d'unicité créent implicitement et obligatoirement un index pour implémenter cette unicité.

3.2.28 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, sans erreur :

```
CREATE TABLE utilisateurs(id integer);
INSERT INTO utilisateurs VALUES (10);
INSERT INTO utilisateurs VALUES (10);
```

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

```
TRUNCATE utilisateurs;
ALTER TABLE utilisateurs ADD UNIQUE(id);
INSERT INTO utilisateurs (id) VALUES (10);
INSERT INTO utilisateurs (id) VALUES (11);
INSERT INTO utilisateurs (id) VALUES (11);
```

```
ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(11) already exists.
```

Le cas de `NULL` est un peu particulier. Par défaut, plusieurs valeurs `NULL` peuvent figurer dans un index car elles ne sont pas considérées comme égales, mais de valeur inconnue (*unknown*). Par exemple, une table de personnes physiques peut contenir un champ `numero_secu` qui ne doit pas contenir de doublon, mais n'est pas forcément rempli, donc contient de nombreuses valeurs nulles.

Ici, on peut insérer plusieurs valeurs NULL :

```
INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT INTO utilisateurs (id) VALUES (NULL);
```

Ce comportement est modifiable en version 15, et une seule valeur `NULL` sera tolérée. Lors de la création de la contrainte, il faut préciser ce nouveau comportement :

```
TRUNCATE utilisateurs;
ALTER TABLE utilisateurs DROP CONSTRAINT utilisateurs_id_key;
ALTER TABLE utilisateurs ADD UNIQUE NULLS NOT DISTINCT(id);
INSERT INTO utilisateurs (id) VALUES (10);
INSERT INTO utilisateurs (id) VALUES (10);

ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(10) already exists.

INSERT INTO utilisateurs (id) VALUES (11);
INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT INTO utilisateurs (id) VALUES (NULL);

ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(null) already exists.
```

3.2.29 Intégrité référentielle

- **Contrainte d'intégrité référentielle**
  - ou **Clé étrangère**
- Référence une **clé primaire** ou un groupe de colonnes `UNIQUE` et `NOT NULL`
- Garantit 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. Toute insertion ou modification qui viole cette règle est rejetée.

C'est ce mécanisme qui permet de garantir qu'une commande sera liée à un client existant dans la table des clients, ou que le pays dans une adresse existe bien dans la table des pays. Et à l'inverse, on

ne pourra supprimer un pays ou un client dans les tables des commandes ou adresses qui possèdent une contrainte d'intégrité dessus.

À titre d'exemple nous allons utiliser la base `cave`. La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

Le schéma suivant montre les différentes tables de la base :

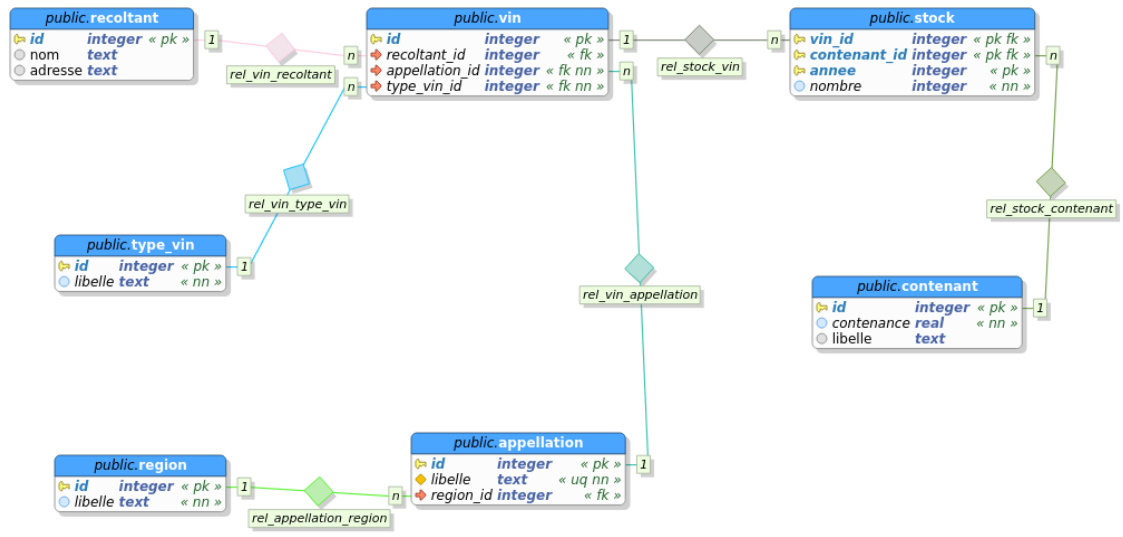


FIGURE 3/ .1 – Schéma base cave

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`.

3.2.30 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
  );
```

Cette table possède une clé primaire composée : il ne peut exister qu'une seule ligne pour un trio vin, contenant, année précis.

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;
```

Ces trois dernières options sont optionnelles, et les valeurs par défaut sont différentes (voir plus bas).

```
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 est accepté :

```
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 mère est refusé :

```
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".

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
```

Mettre à jour la référence dans la table mère ne fonctionnera pas car la contrainte a été définie pour refuser les mises à jour (`ON UPDATE NO ACTION`) :

```
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 mère référencée dans la table fille va propager la suppression jusqu'à la table fille (`ON DELETE CASCADE`) :

```
DELETE FROM mere WHERE id=2;

DELETE 1

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1

SELECT * FROM mere;

 id |  t
----+------
  1 | val1
```

3.2.31 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 multicolonnes, 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
```

3.2.32 Clé primaire et colonne identité

- Identité d'un enregistrement
- `GENERATED … AS IDENTITY`
  - `ALWAYS`
  - `BY DEFAULT`
- Préférer à `serial`
- Ne dispense pas d'ajouter la contrainte de clé primaire !

La syntaxe `GENERATED … AS IDENTITY` 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`). `GENERATED … AS IDENTITY` est apparue dans PostgreSQL après le pseudo-type `serial`, d'utilisation similaire, mais il en corrige certains défauts.

Pour les raisons qui suivent, préférez l'utilisation de `GENERATED … AS IDENTITY` à celle de `serial`, lequel peut se rencontrer encore dans de nombreuses applications.

Tout d'abord, `GENERATED … AS IDENTITY` fait partie du standard SQL, alors que le pseudo-type `serial` est propre à PostgreSQL.

`serial` n'est en effet qu'un raccourci pour créer à la fois une séquence, un champ entier et un `DEFAULT nextval()` pour alimenter ce champ, et cela peut poser quelques soucis. Notamment, le DDL saisi par l'utilisateur diffère de celui stocké en base ou sorti par `pg_dump`, ce qui n'est pas idéal. Ou encore, l'ordre de copie de structure de tables `CREATE TABLE … LIKE (INCLUDING ALL)` copie le `serial` sans en changer la séquence ni en créer une autre : on a alors une séquence partagée par deux tables ! Il n'est pas non plus 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);
INSERT INTO personnes (nom) VALUES ('Dupont') ;
INSERT INTO personnes (nom) VALUES ('Durand') ;

SELECT * FROM personnes ;

 id |  nom
----+--------
  1 | Dupont
  2 | Durand

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

SELECT * FROM personnes ;

 id |  nom
----+--------
  1 | Dupont
  2 | Durand
  3 | Martin
  3 | Dupond
```

3.2.33 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 entraîner 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
```

3.2.34 Vérifications

- Présence d'une valeur
  - `NOT NULL`
  - NB : obligatoire dans une clé primaire
- 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. Dit autrement, elle doit obligatoirement être renseignée. Par défaut, une colonne peut avoir une valeur `NULL`, donc n'est pas obligatoirement renseignée. Rappelons qu'une clé primaire ne peut pas être à `NULL`.

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 `NULL` 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. Une fonction définie par l'utilisateur peut être utilisée à condition de respecter ces mêmes règles.

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.

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0),
    prix_promotion numeric,
```

```
    CONSTRAINT promo_valide CHECK (prix_promotion > 0 AND prix > prix_promotion)
);
```

Cet exemple est inspiré de la documentation officielle, page Contraintes de vérification[1].

3.2.35 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` :

```
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 :

[1] https://docs.postgresql.fr/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS

```
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

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2

UPDATE fille SET mere_id=3 WHERE mere_id=1;

COMMIT;
```

3.2.36 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 fort impact sur les performances et sur la maintenabilité de la base de données. Il vaut mieux privilégier les contraintes déclaratives et les colonnes générées, et n'utiliser les triggers qu'en dernier recours.

3.2.37 Colonnes par défaut et générées

```
CREATE TABLE paquet (
  code       text           PRIMARY KEY,
  reception  timestamptz    DEFAULT now(),
  livraison  timestamptz    DEFAULT now() + interval '3d',
  largeur    int,    longueur int,   profondeur int,
  volume     int
      GENERATED ALWAYS AS ( largeur * longueur * profondeur )
      STORED    CHECK (volume > 0.0)
  ) ;
```

- `DEFAULT` : expressions très simples, modifiables
- `GENERATED`
  - fonctions « immutables », ne dépendant **que** de la ligne
  - (avant v17) difficilement modifiables
  - peuvent porter des contraintes

Les champs `DEFAULT` sont très utilisés, mais PostgreSQL supporte les colonnes générées (ou calculées).

3.2.37.1 DEFAULT …

Les champs avec une clause `DEFAULT` sont remplis automatiquement à la création de la ligne quand une valeur n'est pas fournie dans l'ordre `INSERT` (celui-ci peut ne renseigner que certains champs).

Seules sont autorisées avec `DEFAULT` des expressions simples, sans variable, ni utilisation de champs de la ligne, ni sous-requête. Sont acceptées des constantes, certains calculs ou fonctions simples, comme `now()`, ou un appel à `nextval ('nom_séquence')`.

Ajouter une clause `DEFAULT` sur un champ existant calcule les valeurs pour toutes les lignes pré-existantes de la table, ce qui peut entraîner la réécriture de la table ! Une optimisation évite de tout réécrire si les anciennes lignes se retrouvent toutes avec la même valeur constante.

La valeur par défaut peut être écrasée, par déclaration explicite du champ lors de l'`INSERT`, ou plus tard avec `UPDATE`.

Changer l'expression d'une clause `DEFAULT` est possible :

```
ALTER TABLE paquet
ALTER COLUMN livraison SET DEFAULT now()+interval '4d';
```

mais cela n'a pas d'impact sur les lignes existantes, juste les nouvelles.

3.2.37.2 GENERATED ALWAYS AS (...) STORED

La syntaxe est :

```
nomchamp  <type> GENERATED ALWAYS AS ( <expression> ) STORED ;
```

Les colonnes générées sont recalculées à chaque fois que les champs sur lesquels elles sont basées changent, donc aussi lors d'un `UPDATE`. Ces champs calculés sont impérativement marqués `ALWAYS`, c'est-à-dire obligatoires et non modifiables, et `STORED`, c'est-à-dire stockés sur le disque (et non re-calculés à la volée comme dans une vue). Ils ne doivent pas se baser sur d'autres champs calculés.

Un intérêt est que les champs calculés peuvent porter des contraintes, par exemple la clause `CHECK` ci-dessous, mais encore des clés étrangères ou unique.

Exemple :

```
CREATE TABLE paquet (
  code       text          PRIMARY KEY,
  reception  timestamptz   DEFAULT now(),
  livraison  timestamptz   DEFAULT now() + interval '3d',
  largeur    int,    longueur int,   profondeur int,
  volume     int
      GENERATED ALWAYS AS ( largeur * longueur * profondeur )
      STORED    CHECK (volume > 0.0)
  ) ;

INSERT INTO paquet (code, largeur, longueur, profondeur)
VALUES ('ZZ1', 3, 5, 10) ;

\x on

TABLE paquet ;

-[ RECORD 1 ]----------------------------
code       | ZZ1
reception  | 2024-04-19 18:02:41.021444+02
livraison  | 2024-04-22 18:02:41.021444+02
largeur    | 3
longueur   | 5
profondeur | 10
volume     | 150

-- Les champs DEFAULT sont modifiables
-- Changer la largeur va modifier le volume
```

```
UPDATE paquet
SET largeur=4,
    livraison = '2024-07-14'::timestamptz,
    reception = '2024-04-20'::timestamptz
WHERE code='ZZ1' ;

TABLE paquet ;

-[ RECORD 1 ]----------------------
code       | ZZ1
reception  | 2024-04-20 00:00:00+02
livraison  | 2024-07-14 00:00:00+02
largeur    | 4
longueur   | 5
profondeur | 10
volume     | 200

-- Le volume ne peut être modifié
UPDATE paquet
SET volume = 250
WHERE code = 'ZZ1' ;

ERROR:  column "volume" can only be updated to DEFAULT
DETAIL : Column "volume" is a generated column.
```

Expression immutable :

Avec `GENERATED`, l'expression du calcul doit être « immutable », c'est-à-dire ne dépendre **que** des autres champs de la même ligne, n'utiliser que des fonctions elles-mêmes immutables, et rien d'autre. Il n'est donc pas possible d'utiliser des fonctions comme `now()`, ni des fonctions de conversion de date dépendant du fuseau horaire, ou du paramètre de formatage de la session en cours (toutes choses autorisées avec `DEFAULT`), ni des appels à d'autres lignes ou tables…

La colonne calculée peut être convertie en colonne « normale » :

```
ALTER TABLE paquet ALTER COLUMN volume DROP EXPRESSION ;
```

Modifier l'expression n'est pas possible avant PostgreSQL 17, sauf à supprimer la colonne générée et en créer une nouvelle. Il faut alors recalculer toutes les lignes et réécrire toute la table, ce qui peut être très lourd.

À partir de PostgreSQL 17, l'expression est modifiable avec cette syntaxe :

```
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS  ( largeur * longueur * profondeur + 1 ) ;
```

Attention, la table est totalement bloquée le temps de la réécriture (verrou `AccessExclusiveLock`).

Utilisation d'une fonction :

Il est possible de créer sa propre fonction pour l'expression, qui doit aussi être immutable :

```
CREATE OR REPLACE FUNCTION volume (l int, h int, p int)
RETURNS int
AS $$
 SELECT l * h * p ;
$$
```

```
LANGUAGE sql
-- cette fonction dépend uniquement des données de la ligne donc :
PARALLEL SAFE
IMMUTABLE ;

-- Changement à partir de PostgreSQL v17
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS  ( volume (largeur, longueur, profondeur) );

-- Changement avant PostgreSQL 16
ALTER TABLE paquet DROP COLUMN volume ;
ALTER TABLE paquet ADD COLUMN volume int
  GENERATED ALWAYS AS ( volume (largeur, longueur, profondeur) )
  STORED;

TABLE paquet ;

-[ RECORD 1 ]----------------------
code       | ZZ1
reception  | 2024-04-20 00:00:00+02
livraison  | 2024-07-14 00:00:00+02
largeur    | 4
longueur   | 5
profondeur | 10
volume     | 200
```

Attention : modifier la fonction ne réécrit pas spontanément la table, il faut forcer la réécriture avec par exemple :

```
UPDATE paquet SET longueur = longueur ;
```

et ceci dans la même transaction que la modification de fonction. On pourrait imaginer de négliger cet `UPDATE` pour garder les valeurs déjà présentes qui suivraient d'anciennes règles… mais ce serait une erreur. En effet, les valeurs calculées ne figurent pas dans une sauvegarde logique, et en cas de restauration, tous les champs sont recalculés avec la dernière formule !

On préférera donc gérer l'expression dans la définition de la table dans les cas simples.

Un autre piège : il faut résister à la tentation de déclarer une fonction comme immutable sans la certitude qu'elle l'est bien (penser aux paramètres de session, aux fuseaux horaires…), sous peine d'incohérences dans les données.

Cas d'usage :

Les colonnes générées économisent la création de triggers, ou de vues de « présentation ». Elles facilitent la dénormalisation de données calculées dans une même table tout en garantissant l'intégrité.

Un cas d'usage courant est la dénormalisation d'attributs JSON pour les manipuler comme des champs de table classiques :

```
ALTER TABLE personnes
ADD COLUMN  lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED ;
```

L'accès au champ est notablement plus rapide que l'analyse systématique du champ JSON.

Par contre, les colonnes `GENERATED` ne sont **pas** un bon moyen pour créer des champs portant la dernière mise à jour. Certes, PostgreSQL ne vous empêchera pas de déclarer une fonction (abusivement) immutable utilisant `now()` ou une variante. Mais ces informations seront perdues en cas de restauration logique. Dans ce cas, les triggers restent une option plus complexe mais plus propre.

3.3 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 (sauf fonction)
- Ajout de données dans une table
  - `INSERT`
- Modification des données d'une table
  - `UPDATE`
- L'un ou l'autre
  - `MERGE`
- Suppression des données d'une table
  - `DELETE`

L'ordre `SELECT` permet de lire une ou plusieurs tables, sans modification. Il faut toutefois savoir que `SELECT` peut aussi servir à appeler des fonctions, et que celles-ci sont susceptibles de faire à peu près n'importe quoi dans la base.

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.

Depuis PostgreSQL 15, il existe un ordre `MERGE` (conforme au standard SQL) permettant par exemple de mettre à jour une ligne dont la clé existe, ou de l'insérer au besoin. L'utilisation peut être complexe[2].

Enfin, l'ordre `DELETE` permet de supprimer des lignes.

Ces derniers ordres ne peuvent modifier qu'une seule table à la fois. Si on souhaite par exemple insérer des données dans deux tables, il est nécessaire de réaliser deux ordres `INSERT` distincts.

Noter qu'il est possible de regrouper plusieurs `SELECT` / `DELETE` / `INSERT` /etc. dans une même requête, grâce aux *Common Table Expressions*[3], que nous n'aborderons pas ici.

[2] https://docs.postgresql.fr/current/sql-merge.html
[3] https://docs.postgresql.fr/current/queries-with.html

3.3.1 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 }
```

— Précisez toujours les champs à renseigner

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 des colonnes est optionnelle. L'ordre des noms des colonnes dans la liste n'a pas d'importance particulière, il suffit de nommer les colonnes mises à jour.

Si la liste 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` premiers noms de colonnes si seules `N` valeurs de colonnes sont fournies dans la clause `VALUES` ou dans la requête.

Cette dernière pratique est à éviter car elle rend votre code sensible aux modifications de structures : ajout de champs, modification de l'ordre des champs… Cela arrive plus facilement qu'on ne le croit.
Prenez l'habitude de nommer les champs à renseigner.

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. Si vous avez défini des triggers avant insertion, ils se déclencheront aussi.

Dans l'exemple suivant, seules deux colonnes sur cinq sont renseignées à l'insertion :

```
CREATE TABLE demoins (
    id    int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    zero  int DEFAULT 0,
    x     float,
    y     float GENERATED ALWAYS AS (x/2.0) STORED,
    z     timestamp with time zone) ;

INSERT INTO demoins (x, z)
SELECT 3.14159, '2025-07-01' ;

TABLE demoins ;
```

```
 id | zero |    x    |    y     |           z
----+------+---------+----------+------------------------
  1 |    0 | 3.14159 | 1.570795 | 2025-07-01 00:00:00+02
```

3.3.2 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);
```

3.3.3 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 exactement à l'ordre précisé dans l'ordre `INSERT`. Leur type de données doit également correspondre.

Exemple :

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;
```

3.3.4 Mise à jour de données : 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`.

3.3.5 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 colonnes 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 = expression`. 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, dont 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 aussi de réaliser des mises à jour en utilisant une sous-requête, sans 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)
   );
```

3.3.6 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';
```

Suppression de tous les livres :

```
DELETE FROM livres ;
```

N'oubliez jamais de préciser la clause `WHERE` !

Pour des raisons techniques, il faut savoir qu'il est beaucoup plus rapide de vider une table avec l'ordre `TRUNCATE TABLE livres ;` qu'avec un `DELETE` sans clause `WHERE`.

3.3.7 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 syntaxe propre à PostgreSQL et très pratique. 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.

Exemple :

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
```

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

```
 nombre
--------
     16
```

3.4 TRANSACTIONS

- ACID
  - **A**tomicité
  - un traitement se fait en entier…
  - …ou pas du tout
- TCL pour *Transaction Control Language*
  - ouvrir une transaction : `BEGIN`
  - valide une transaction : `COMMIT`
  - annule une transaction : `ROLLBACK`
  - points de sauvegarde : `SAVEPOINT`

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

Un client qui ouvre une transaction et déroule des ordres, est isolé des autres sessions, dans le sens où les autres sessions *ne voient pas* ses modifications avant la validation par `COMMIT`.

Pour les autres sessions, même appartenant au même utilisateur, les données sont soit telles qu'elles étaient *avant* la transaction, soit telles qu'elles sont devenues *après* validation.

L'utilisateur qui fait les modifications peut voir les éventuels états intermédiaires d'une transaction entre deux ordres (il peut faire des `SELECT` sur ce qu'il vient de modifier), mais uniquement depuis la session de la transaction.

Il est impossible de voir les étapes intermédiaires d'un ordre SQL, car une session exécute séquentiellement les ordres, et attend la fin de chaque ordre avant de poursuivre.

Par défaut, une transaction en cours voit les modifications validées dans d'autres sessions parallèles. Ce « niveau d'isolation » peut se changer pour que la transaction ne voit pas les modifications d'autres transaction, comme si la base était figée pendant sa durée.

Le langage SQL définit qu'une transaction peut être ouverte avec `BEGIN`, et à la fin 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`.

3.4.1 Auto-commit et transactions

- Par défaut, PostgreSQL fonctionne en autocommit
  - à moins d'ouvrir explicitement une transaction
- Ouvrir une transaction
  - `BEGIN ;`

Une transaction débute toujours par un `BEGIN ;`. (`BEGIN TRANSACTION` et `START TRANSACTION` sont des synonymes, mais se rencontrent très peu.)

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

Pour l'utilisateur, cela est en fait géré par l'outil client, qui ajoute des `BEGIN` invisibles si on le configure avec autocommit à `off`. La transaction ne finira qu'avec un `COMMIT`, un `ROLLBACK` ou la déconnexion (le `ROLLBACK` est alors implicite).

3.4.2 Validation ou annulation d'une transaction

- Valider une transaction
  - `COMMIT ;`
- Annuler une transaction
  - `ROLLBACK ;`
- Sans validation, une transaction est forcément annulée
- Quasi-instantané sous PostgreSQL

Une transaction est toujours terminée par un `COMMIT ;` quand on veut que les modifications soient définitivement enregistrées. `END ;` existe aussi mais est rarissime et moins clair.

`ROLLBACK` quitte la transaction en annulant *toutes* les modifications qui y ont été faites, donc en revenant à l'état précédant la transaction. D'autres transactions ont pu faire d'autres actions pendant ce temps, elles ne seront pas annulées.

Ce retour à l'état initial doit se comprendre d'un point de vue logique et pas physique. Les modifications que vous avez pu effectuer dans une transaction terminée par un `ROLLBACK` ont quand même une incidence sur le stockage de vos données. Il peut y avoir eu de nombreux changements qui sont éventuellement à nettoyer plus tard par PostgreSQL, et peuvent avoir un impact sur les performances.
N'abusez donc pas des `ROLLBACK` si vous pouvez l'éviter.

Contrairement à certains produits concurrents, PostgreSQL exécute presque instantanément les ordres `COMMIT` et `ROLLBACK`.
En simplifiant, PostgreSQL se limite à noter que la transaction est valide ou pas, et que dans le futur les modifications concernées dans les fichiers de données doivent être prises en compte ou ignorées.

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

Exemple :

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

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

 nombre
--------
     16

COMMIT;
```

3.4.3 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'autocommit :

```
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();
```

3.4.4 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 ;`
  - et on continue dans la même transaction

Au sein d'une transaction, les points de sauvegarde (ordre `SAVEPOINT nom ;`) permettent d'encadrer un traitement sur lequel on peut vouloir revenir sans quitter la transaction (par exemple, selon le retour d'un test), ou au cas où un ordre tombe en erreur.

Revenir à l'état du point de sauvegarde se fait avec `ROLLBACK TO SAVEPOINT nom ;` pour annuler uniquement la partie du traitement voulue, ou pour revenir à l'état d'avant l'erreur, sans annuler le début de la transaction. On peut ensuite continuer.

Les *savepoints* n'existent qu'au sein de la transaction ; on ne peut bien sûr revenir sur une validation par `COMMIT` qu'une fois celui-ci terminé.

L'ordre `RELEASE SAVEPOINT nom ;` permet de « libérer » (oublier) un *savepoint* précédent ainsi que ceux posés *après*. C'est surtout utile pour libérer quelques ressources dans des transactions complexes.

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.

N'abusez pas des `SAVEPOINT` : ils complexifient le code, et trop de `ROLLBACK` partiels peuvent entraîner des soucis de performance. Les `SAVEPOINT` sont rarement nécessaires quand on crée des transactions vraiment atomiques en « tout ou rien ». Réservez-les aux transactions vraiment complexes.

Exemple :

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
```

3.5 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`), modifiées (`ALTER`) et/ou supprimées (`DROP`).

3.5.1 Questions

```
BEGIN ;

WITH q AS (
  SELECT * FROM questions
  )
INSERT INTO reponses
SELECT * FROM q
WHERE texte_reponse IS NOT NULL ;

COMMIT ;
```

3.6 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s30_solutions.

Cet exercice utilise la base **tpc**. La base **tpc** (dump de 31 Mo, pour 267 Mo sur le disque au final) et ses utilisateurs peuvent être installés comme suit :

```
curl -kL https://dali.bo/tp_tpc -o /tmp/tpc.dump
curl -kL https://dali.bo/tp_tpc_roles -o /tmp/tpc_roles.sql
# Exécuter le script de création des rôles
psql < /tmp/tpc_roles.sql
# Création de la base
createdb --owner tpc_owner tpc
# L'erreur sur un schéma 'public' existant est normale
pg_restore -d tpc /tmp/tpc.dump
```

Les mots de passe sont dans le script `/tmp/tpc_roles.sql`. 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.

> 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`).

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

> 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 `@`).

> Valider la contrainte dans une transaction de test.

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

> 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.

> 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`.

> 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.

4/ Plus loin avec SQL

4.1 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.

4.1.1 Menu

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

4.1.2 Menu (suite)

- Fonctions de base
- Vues
- Requêtes préparées

4.1.3 Objectifs

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

Les exemples suivants utilisent souvent la base **cave**. La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

4.2 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 à 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.

4.2.1 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).

4.2.2 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)
```

4.2.3 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.

4.2.4 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 ;

 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)
```

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

4.2.5 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 quant à elle 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
```

4.2.6 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
```

4.3 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.

4.3.1 Regroupement de données

— Regroupement de données :
  `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.

4.3.2 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.

4.3.3 Agrégats simples

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

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

```
 (null) |            2 |       0
      1 |            1 |       1
      2 |            2 |       2
```

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

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

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

4.3.4 Calculs d'agrégats

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

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

```
  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) ;
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
```

4.3.5 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
```

4.3.6 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`.

La clause `HAVING` est toujours accompagnée de la clause `GROUP BY`. La clause `GROUP BY` quant à elle n'est pas toujours accompagnée de 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 connaître 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
```

4.4 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

4.4.1 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.

4.4.2 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
…
```

4.4.3 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') ;
```

4.4.4 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.

4.4.5 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) ;
```

4.4.6 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`.

4.4.7 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) ;
```

4.4.8 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) ;
```

4.4.9 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 ;
```

4.4.10 Clause EXISTS

```
EXISTS (sous-requête)
```

- 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) ;
```

4.5 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.

4.5.1 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`
  - historique

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 ;
```

4.5.2 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
```

4.5.3 Jointure interne

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

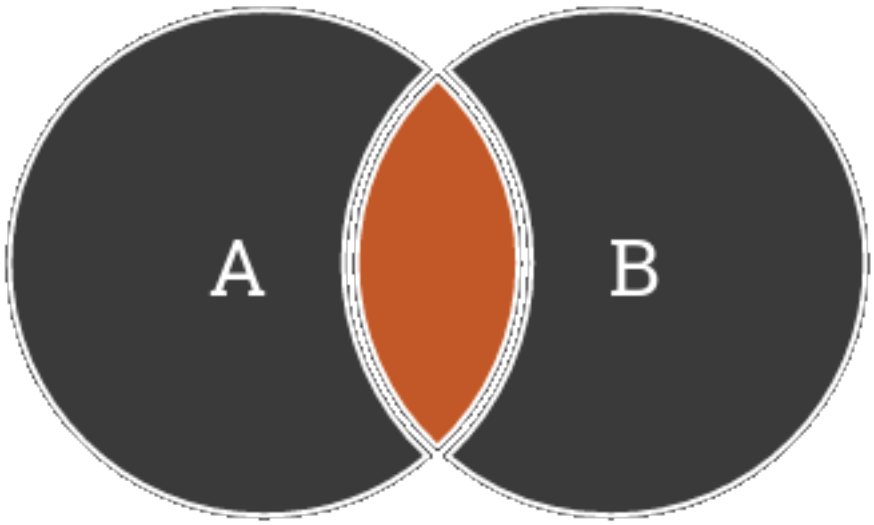


Figure 4/ .1 – Schéma de jointure interne

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.

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.

4.5.4 Syntaxe d'une jointure interne

— Condition de jointure par prédicats :

```
table1 [INNER] JOIN table2 ON prédicat […]
```

— Condition de jointure implicite par liste des colonnes impliquées :

```
table1 [INNER] JOIN table2 USING (colonne [, …])
```

— Liste des colonnes de même nom (dangereux) :

```
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 déconseillée 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) ;
```

4.5.5 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`

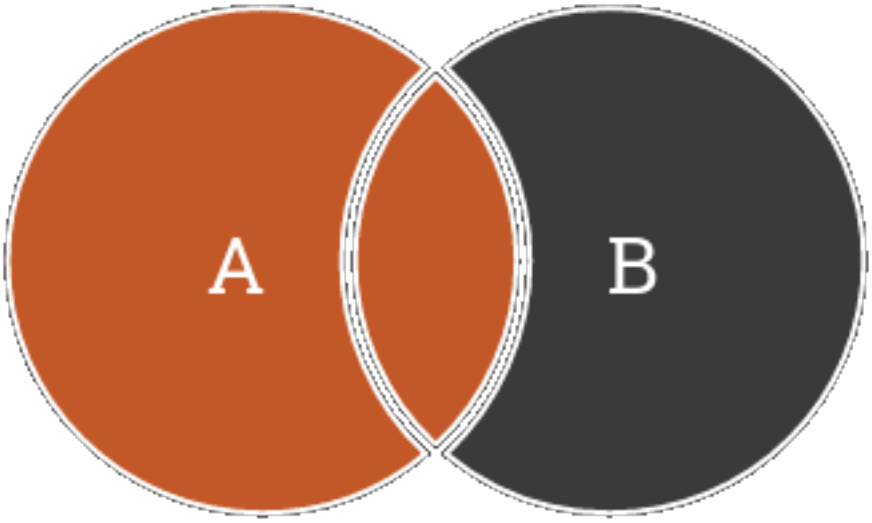


Figure 4/ .2 – Schéma de jointure externe gauche

Il existe deux types de jointures externes : 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.

4.5.6 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`

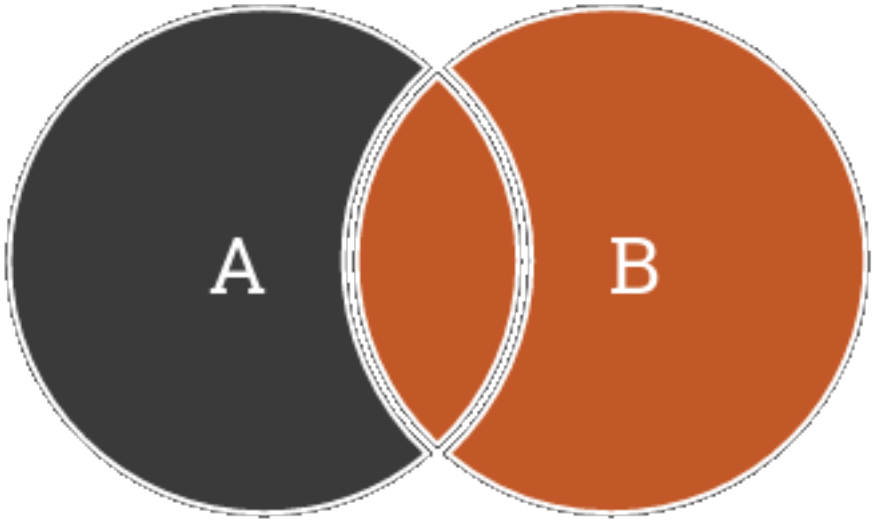


FIGURE 4/ .3 – Schéma de jointure externe droite

4.5.7 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`

4.5.8 Syntaxe d'une jointure externe à gauche

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

```
SELECT article.art_titre, auteur.aut_nom
  FROM article
  LEFT JOIN auteur
    ON (article.aut_id=auteur.aut_id) ;
```

— par liste de colonnes :

```
SELECT article.art_titre, auteur.aut_nom
  FROM article
  LEFT JOIN auteur
    USING (aut_id) ;
```

4.5.9 Syntaxe d'une jointure externe à droite

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

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

4.5.10 Syntaxe d'une jointure externe complète

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

4.5.11 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.

4.6 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;
```

4.6.1 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 associer 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
```

4.6.2 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
```

4.6.3 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
```

4.7 OPÉRATEURS ENSEMBLISTES

- `UNION`
- `INTERSECT`
- `EXCEPT`

4.7.1 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 ;
```

4.7.2 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
```

4.7.3 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
```

4.8 FONCTIONS DE BASE

— Transtypage
— Manipulation de chaînes
— 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].

4.8.1 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

4.8.2 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 types 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!
```

4.8.3 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 caractères à partir d'une chaîne en entrée. Il faut lui indiquer, en plus de la chaîne source, la position de départ, et la longueur 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 caractères 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
```

4.8.4 Manipulation de types numériques

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

4.8.5 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
```

4.8.6 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 la documentation[6].

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

4.8.7 Génération de données

```
-- Suite d'entiers, 1 par ligne
SELECT i   FROM   generate_series(0, 100, 1) i ;

-- Nombre entre 0 et 1.0
SELECT random() ;
-- Entre 1 et 100
SELECT (random()*100)::int ;

-- Entre 2 bornes (v17+)
SELECT random (20.0, 50.0) ;  -- numeric
SELECT random (-100, 100) ;   -- entier
```

La fonction `generate_series` est spécifique à PostgreSQL et permet de générer une suite d'entiers compris entre une borne de départ et une borne de fin :

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

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

Avec un incrément pour chaque itération :

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

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

Cette fonction est aussi utilisée pour générer des lignes à volonté :

```
SELECT i FROM generate_series(1,10,3) i ;
```

```
 i
----
  1
  4
  7
 10
(4 lignes)
```

La fonction `random()` génère un nombre aléatoire, de type `double precision` (synonyme de `float` et `float8`), 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
```

Depuis PostgreSQL 17, il existe une version plus pratique avec deux bornes, et qui peut renvoyer, selon les paramètres, un `integer`, un `bigint`, ou un `numeric` :

```
SELECT random(  1.5, 1.9 ) ,      -- numeric
       random( -100, 100),        -- int
       random( 1e10, 1e10+100) ; -- bigint

 random | random |   random
--------+--------+-------------
    1.8 |    -22 | 10000000084
```

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].

4.8.8 Manipulation de dates

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

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

4.8.9 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
```

```
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!
```

4.8.10 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
```

4.8.11 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
```

4.8.12 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
```

4.8.13 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
```

4.8.14 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:M`
— 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 le paramètre `DateStyle` est égal à `'SQL, DMY'`.

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

SELECT current_timestamp ;

              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
```

4.8.15 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
```

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

```
 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)
```

4.9 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 sont des vraies tables créées à partir d'une requête (et rafraîchies uniquement sur demande explicitement).

4.9.1 Création d'une vue

- Une vue porte un nom au même titre qu'une table
  - elle sera nommée avec les mêmes règles
- 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 d'ordre par rapport à l'ordre `CREATE` original.

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.

Exemple :

```
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 ;
```

4.9.2 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
```

4.9.3 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.

Exemple :

```
CREATE TABLE elements (id serial, contenu text, prive boolean) ;
INSERT INTO elements (contenu, prive)
VALUES ('a', false), ('b', false), ('c super prive', true),
       ('d', false), ('e prive aussi', true) ;

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.

```
CREATE OR REPLACE VIEW elements_public AS
SELECT * FROM elements
WHERE CASE WHEN current_user='postgres' THEN TRUE
           ELSE NOT prive
      END ;

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

-- Création d'un utilisateur nommé u1
CREATE ROLE u1 LOGIN ;
GRANT SELECT ON elements_public TO u1;
```

La connexion avec l'utilisateur `u1` suppose que la configuration nécessaire a été faite par le DBA.

```
\c - u1
You are now connected to database "postgres" as user "u1".
```

Tentative de lecture de la table puis de la vue :

```
SELECT * FROM elements ;

ERROR:  permission denied for relation elements

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`.

Cependant, `u1` peut révéler le contenu de `elements` à travers une fonction spécifique. Cette fonction peut divulguer les informations des lignes de données qu'elle traite avant même qu'elles ne soient filtrées par la vue :

```
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
           -- définir un coût d'exécution très bas pour exécuter
           -- cette fonction avant le filtre dans la vue
           COST 0.00000000000000000001 ;

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 le plan d'exécution de cette requête avec la commande `EXPLAIN` :

```
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.

Pour interdire l'optimisation malvenue du planificateur, ajouter à la vue l'option `security_barrier` :

```
postgres=> \c - postgres
You are now connected to database "postgres" as user "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 ;

\c - u1
You are now connected to database "postgres" as user "u1".

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
```

```
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, et peut donc forcer l'optimiseur à s'écarter du chemin optimal.

Sur `security_barrier`, Robert Haas a écrit un très bon article de blog[9].

4.9.4 Mise à jour des vues

— *Updatable view*
— `WITH CHECK OPTION`
— Mises à jour non triviales : trigger `INSTEAD OF`

Le moteur permet de mettre à jour les données dans 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 au plus une clause `WHERE` pour restreindre les résultats.

La clause `WITH CHECK OPTION` empêche l'utilisateur d'insérer des données qui ne satisfont pas les critères de filtrage de la vue. En effet, par défaut, il est par exemple possible d'insérer un numéro de téléphone privé alors que la vue ne permet pas d'afficher les numéros privés. `WITH CHECK OPTION` doit être demandée explicitement.

Pour gérer les cas plus complexes, PostgreSQL permet de créer des triggers `INSTEAD OF` sur des vues. 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.

[9] https://rhaas.blogspot.com/2012/03/security-barrier-views.html

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
```

Une alternative aux triggers serait encore d'utiliser le système de règles (*rules*), mais son utilisation est fortement déconseillée en raison de la difficulté de débogage et de maintenance.

4.9.5 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.

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.

4.10 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.

4.10.1 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 prendra 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.

4.11 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.

4.11.1 Questions

N'hésitez pas, c'est le moment !

4.12 TRAVAUX PRATIQUES

Ces TP utilisent la base **tpc**. La base **tpc** (dump de 31 Mo, pour 267 Mo sur le disque au final) et ses utilisateurs peuvent être installés comme suit :

```
curl -kL https://dali.bo/tp_tpc -o /tmp/tpc.dump
curl -kL https://dali.bo/tp_tpc_roles -o /tmp/tpc_roles.sql
# Exécuter le script de création des rôles
psql < /tmp/tpc_roles.sql
# Création de la base
createdb --owner tpc_owner tpc
# L'erreur sur un schéma 'public' existant est normale
pg_restore -d tpc /tmp/tpc.dump
```

Les mots de passe sont dans le script `/tmp/tpc_roles.sql`. Pour vous connecter :

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

Le schéma suivant montre les différentes tables de la base :

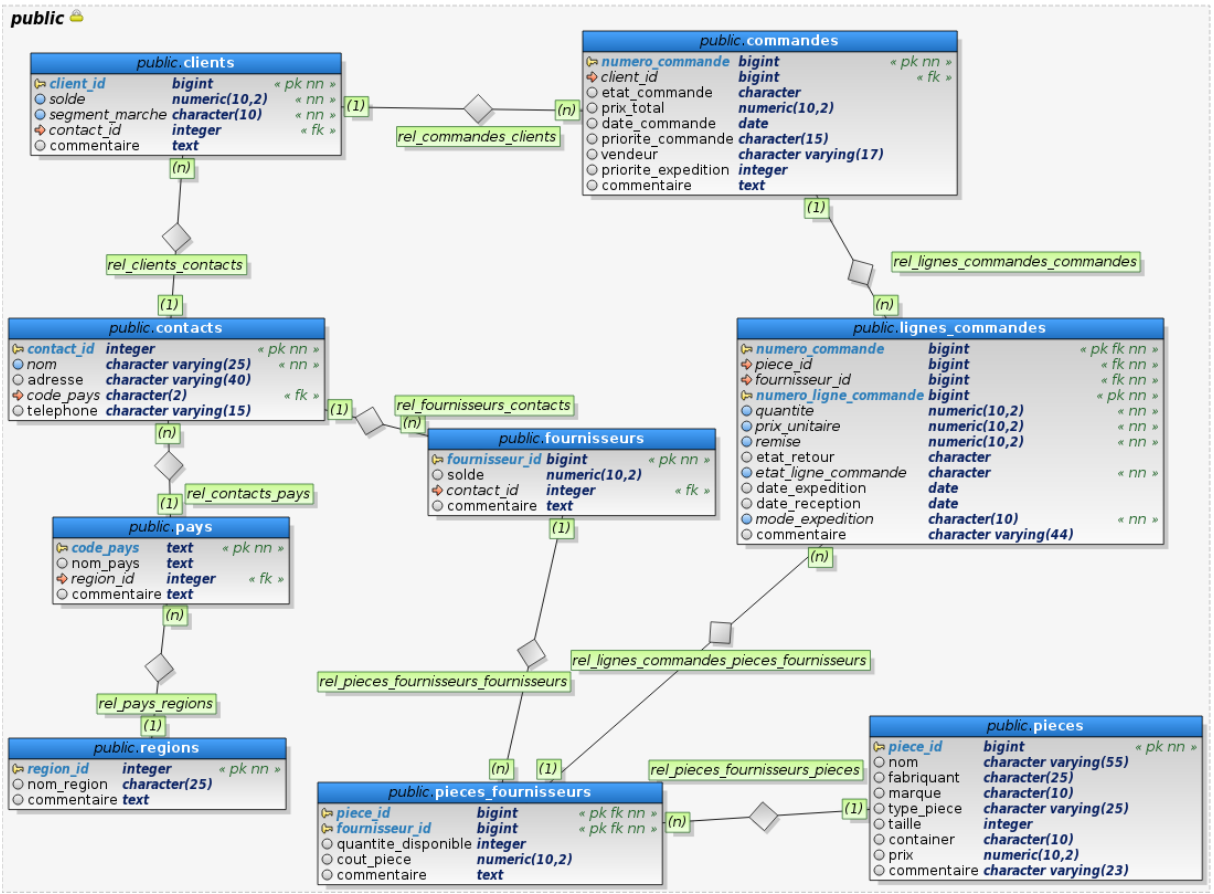


FIGURE 4/ .4 – Schéma base tpc

4.12.1 TP n°1

> Affichez, par pays, le nombre de fournisseurs.

Sortie attendue :

```
           nom_pays             | nombre
--------------------------------+--------
 ARABIE SAOUDITE                |    425
 ARGENTINE                      |    416
 (…)
```

> Affichez, par continent (`regions`), le nombre de fournisseurs.

Sortie attendue :

```
        nom_region          | nombre
----------------------------+--------
 Afrique                    |   1906
 Moyen-Orient               |   2113
 Europe                     |   2094
 Asie                       |   2002
 Amérique                   |   1885
```

> Affichez le nombre de commandes triées 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
```

> 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
 (…)
```

> 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
(…)
```

> 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)
```

> 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)
```

> Déterminez le taux de retour des marchandises (l'état à `R` indiquant qu'une marchandise est retournée).

Sortie attendue :

```
taux_retour
-------------
       24.29
```

> Déterminez le mode d'expédition qui est le plus rapide, en moyenne.

Sortie attendue :

```
 mode_expedition |       delai
-----------------+--------------------
 AIR             | 7.4711070230494535
```

> 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
```

> 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.

> 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
```

> 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
(…)
```

> 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
```

> Écrivez une requête pour supprimer ces commandes. Vérifiez le travail avant de valider.

> 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
(…)
```

> 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
(…)
```

> Affichez le pays qui a connu, en nombre, le plus de commandes sur l'année 2011.

Sortie attendue :

```
    nom_pays     | count
-----------------+-------
 ARABIE SAOUDITE |  1074
```

> 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
(…)
```

> 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'
(…)
```

> 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
(…)
```

4.12.2 TP n°2

> Ajouter une adresse mail à chaque contact avec la concaténation du nom avec le texte « @dalibo.com ».

> Concaténer nom et adresse mail des contacts français sous la forme « nom <mail> ».

> Même demande mais avec le nom en majuscule et l'adresse mail en minuscule.

> Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`.

> Écrivez une requête qui calcul la **somme** d'une commande en fonction de la quantité, le prix unitaire ainsi que la remise d'un produit. Vous trouverez ces informations dans la table `lignes_commandes`.

> Mettre à jour la colonne `prix_total` de la table `commandes` avec la **somme** récupérée de la table `lignes_commandes`.

> Récupérer le montant total des commandes par mois pour l'année 2010. Les montants seront arrondis à deux décimales.

> Supprimer les commandes de mai 2010.

> Ré-exécuter la requête trouvée à la septième question.

Qu'observez-vous ?

Corriger le problème rencontré.

Créer une vue calculant le prix total de chaque commande.

Réécrire la requête de la question 7 pour utiliser la vue créée au point 10.

5/ SQL avancé pour le transactionnel

5.0.1 Préambule

— SQL et PostgreSQL proposent de nombreuses fonctionnalités 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.

5.0.2 Menu

— `LIMIT` / `OFFSET`
— Jointures `LATERAL`
— `UPSERT` : `INSERT` ou `UPDATE`
— *Common Table Expressions*
— Serializable Snapshot Isolation

5.0.3 Objectifs

— Aller au-delà de SQL :92
— Concevoir des requêtes simples pour résoudre des problèmes complexes

Beaucoup de personnes écrivant du SQL ne connaissent que les bases du SQL :92. Le langage a cependant de nombreuses fonctionnalités très puissantes à connaître.

5.1 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 `LIMIT` est cependant plus connue et est plus concise.

5.1.1 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 un jeu d'essai créé ainsi :

```
-- Si la table existe déjà, la détruire
DROP TABLE IF EXISTS employes CASCADE ;

-- Création de la table
CREATE TABLE employes (
matricule char(8) PRIMARY KEY,
nom       text NOT NULL,
service   text,
salaire   numeric(7,2)
);

-- Données
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('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);

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
```

5.1.2 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)
```

5.1.3 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
```

5.1.4 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
```

5.1.5 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
— https://use-the-index-luke.com/fr/no-offset

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, 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[1] suivant :

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=0.43..18.26 rows=10 width=115)
        (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[2] 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[3].

[1] https://explain.dalibo.com/plan/xEs
[2] https://explain.dalibo.com/plan/V05
[3] https://use-the-index-luke.com/fr/no-offset

Dans notre cas, la solution revient à 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
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
```

5.2 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.

On a ainsi une modification et la récupération des valeurs générées, modifiées ou supprimées en un seul ordre.

5.2.1 RETURNING : exemple

```
CREATE TABLE test_returning (id serial primary key, val integer);
-- Insertion de la valeur val seulement
INSERT INTO test_returning (val)
  VALUES (10)
RETURNING id, val;

 id | val
----+-----
  1 |  10
(1 ligne)
```

La clause `RETURNING` permet, par exemple, de récupérer la valeur de colonnes portant une valeur par défaut, comme la valeur créée par une séquence, comme sur l'exemple ci-dessus.

`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
```

Associée à l'ordre `DELETE`, `RETURNING` renvoie les lignes supprimées :

```
DELETE FROM test_returning
 WHERE val < 30
 RETURNING id, val;

 id | val
----+-----
  1 |  20
```

5.3 UPSERT (INSERT … ON CONFLICT)

- `INSERT` ou `UPDATE` ?
  - `INSERT … ON CONFLICT DO { NOTHING | UPDATE }`
- Utilité :
  - mettre à jour en cas de conflit sur un `INSERT`
  - ne rien faire en cas de conflit sur un `INSERT`
- Plutôt préférer `MERGE`

Par *upsert*, on entend la syntaxe `INSERT … ON CONFLICT DO { NOTHING | UPDATE }` qui permet d'insérer une ligne, ou de la mettre à jour si elle existe déjà (c'est-à-dire si la clé qui sert à l'identifier est mise à jour).

L'implémentation de PostgreSQL de `ON CONFLICT DO UPDATE` est une opération atomique, c'est-à-dire que PostgreSQL garantit que l'une ou l'autre seulement des conditions sera effectuée, sans souci en cas de traitements en parallèle et de suppression (sauf erreur pour une autre raison). En comparaison, plusieurs approches naïves présentent des problèmes de concurrences d'accès. (Voir les différentes approches décrites dans cet article de Depesz[4]).

Il faut évidemment qu'il y ait une contrainte d'unicité pour l'identification des lignes.

Une bonne conception évitera autant que possible de modifier les mêmes lignes par des traitements simultanés, ne serait-ce que pour éviter des ralentissements à cause de verrous. Mais ce n'est pas toujours possible.

La clause `INSERT … ON CONFLICT` est très proche de la commande `MERGE` (voir plus bas), plus proche du standard, mais qui n'est apparue dans PostgreSQL que plus tard.

5.3.1 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] https://www.depesz.com/2012/06/10/why-is-upsert-so-complicated/

5.3.2 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.

5.3.3 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).
```

5.3.4 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)
```

5.3.5 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 table `employes`, 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.

5.3.6 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 là 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[5].

5.3.7 MERGE

```
MERGE INTO mesures_capteurs c
USING import_mesures_capteurs i
ON c.id = i .id
WHEN NOT MATCHED THEN
  INSERT (id, top_mesure, derniere_mesure, derniere_maj)
  VALUES (i.id, i.mesure, i.mesure, current_timestamp)
WHEN MATCHED AND ( c.derniere_maj + INTERVAL '10 days' <=
 ↪  current_timestamp ) THEN
  DELETE
WHEN MATCHED AND ( c.top_mesure > i.mesure ) THEN
  UPDATE
  SET derniere_mesure = i.mesure, derniere_maj = current_timestamp
WHEN MATCHED THEN
  UPDATE SET top_mesure = i.mesure, derniere_mesure = i.mesure,
   ↪  derniere_maj = current_timestamp
;
```

`MERGE` est très voisin de `INSERT … ON CONFLICT` mais n'est apparu qu'avec PostgreSQL 15. Leurs mécanismes diffèrent complètement et les deux syntaxes ont leur utilité.

`MERGE` est conforme au standard SQL (avec quelques extensions de syntaxe propres à PostgreSQL), a une syntaxe plus complexe et permet aussi des `DELETE` dans la table cible.

`MERGE` ne s'appuie pas sur les contraintes d'unicité et est plus susceptible de mener à des erreurs en cas de concurrence d'accès forte, au contraire de `INSERT … ON CONFLICT`.

Lors de son exécution, la commande commence par réaliser une jointure entre la source de donnée et la table cible. La source de donnée peut être une table ou une requête quelconque. La condition de

[5] https://docs.postgresql.fr/current/sql-insert.html

jointure ne doit contenir que des colonnes des tables source et cible qui participent à la jointure, et la jointure ne doit produire qu'une ligne pour chaque ligne candidate.

Chaque ligne candidate se voit assigner le statut `[NOT] MATCHED`, suivant que la jointure a été un succès ou non. Ensuite, les clauses `WHEN` sont évaluées dans l'ordre où elles sont spécifiées. Seule l'action associée à la première clause `WHEN` qui renvoie « vrai » est exécutée.

Lorsqu'elles sont exécutées, les actions ont les mêmes effets que des ordres `INSERT`, `UPDATE` ou `DELETE` classiques. La syntaxe est similaire, à la différence prêt qu'il n'y a ni clause `FROM` ni clause `WHERE`. Les actions agissent sur la cible, utilisent les lignes courantes de la jointure et agissent sur la cible.

Il est possible de spécifier `DO NOTHING` si on souhaite ignorer la ligne en cours. Ce résultat peut également être obtenu si aucune clause n'est évaluée à vrai.

`INSERT … ON CONFLICT UPDATE` garantit l'exécution atomique d'un `INSERT` ou d'un `UPDATE` même en cas de forte concurrence d'accès. La commande `MERGE` n'a pas ce genre de garantie. Si un `INSERT` est exécutée en même temps que le `MERGE`, il est possible que le `MERGE` ne la voit pas et choisisse d'utiliser son action `INSERT`, ce qui aboutira à une erreur de violation de contrainte d'unicité. C'est la raison pour laquelle la commande `MERGE` avait été initialement refusée et remplacée par `INSERT … ON CONFLICT`.

Pour les détails et les différences avec `INSERT … ON CONFLICT`, voir :

— Notre workshop de la version 15[6], avec l'exemple ci-dessus complet ;
— la documentation de PostgreSQL[7].

[6] https://dali.bo/workshop15_html#développement-changement-syntaxe-sql
[7] https://www.postgresql.fr/docs/current/sql-merge.html

5.4 LATERAL

— Jointures `LATERAL`
  — SQL :99
  — équivalent d'une boucle `foreach`

`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. L'implémentation dans la plupart des SGBD reste cependant relativement récente.

C'est une manière d'introduire une forme de fonctionnement procédural, alors que le SQL est de nature ensembliste. Ce peut être très utile dans certains cas.

5.4.1 LATERAL : utilité

— Utilité :
  — Top-N à partir de plusieurs tables
  — jointure avec une fonction retournant un ensemble
  — exemple : *afficher les 5 derniers messages des 5 derniers sujets actifs d'un forum*

`LATERAL` 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.

5.4.2 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;
```

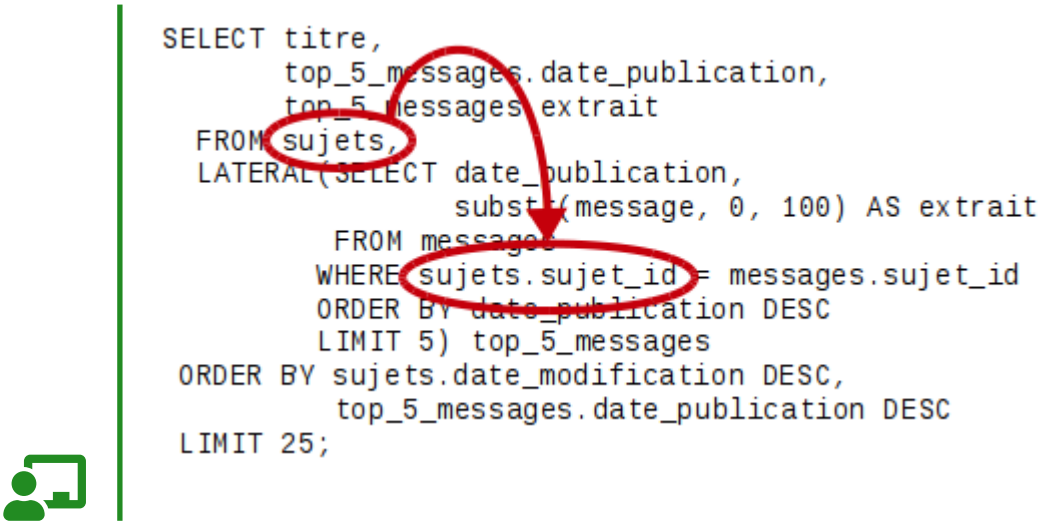
Il aurait été possible de réaliser cette requête par d'autres moyens, mais `LATERAL` permet d'obtenir la requête la plus performante. Une autre approche quasiment aussi performante aurait été de faire appel à une fonction retournant les 5 enregistrements souhaités.

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);
```

5.4.3 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.

5.4.4 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 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*).

5.4.5 LATERAL : exemple avec une fonction

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets
       -- cette fonction peut retourner plusieurs lignes
       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;
```

5.5 COMMON TABLE EXPRESSIONS (CTE)

— *Common Table Expressions*
  — clauses `WITH` et `WITH RECURSIVE`
— Utilité :
  — factoriser des sous-requêtes

5.5.1 CTE et SELECT

— Utilité
  — factoriser des sous-requêtes
  — améliorer la lisibilité d'une requête

Les *Common Table Expressions*, ou 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, éventuellement en cascade, et au final utilisées dans la requête principale.

5.5.2 CTE et SELECT : exemple

```
WITH resultat AS (
   /* requête complexe */
)
SELECT *
  FROM resultat
  WHERE nb < 5;
```

La première utilité d'une CTE est de factoriser la définition d'une sous-requête commune, comme `resultat` dans l'exemple ci-dessus.

Cela est logiquement équivalent à écrire une vue `resultat` de même définition et à l'utiliser dans la requête. La CTE évite la création, la maintenance et la suppression de ces vues. La souplesse et, nous le verrons, même les performances peuvent y gagner. Évidemment, si la définition de `resultat` est utilisée dans plusieurs requêtes, une vue à part entière sera plus pertinente.

L'exemple suivant illustre une réutilisation multiple d'une même CTE. Il utilise la base d'exemple **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin  /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

La CTE `commandes_2014` définit un ensemble de commandes, et ce jeu de données est utilisé deux fois, dans chacun des membres du `UNION ALL`.

```
WITH commandes_2014 AS (
-- vue des données sur 2014
SELECT c.numero_commande, c.client_id, quantite*prix_unitaire AS montant
  FROM magasin.commandes c
  JOIN magasin.lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
)
-- Regroupement par type_client
SELECT type_client, NULL AS pays, SUM(montant) AS montant_total_commande
  FROM commandes_2014
  JOIN magasin.clients
    ON (commandes_2014.client_id = clients.client_id)
 GROUP BY type_client
--
UNION ALL
--
-- Regroupement par type_pays
SELECT NULL, code_pays AS pays, SUM(montant)
  FROM commandes_2014 r
  JOIN magasin.clients cl
    ON (r.client_id = cl.client_id)
  JOIN magasin.contacts co
    ON (cl.contact_id = co.contact_id)
 GROUP BY code_pays;
```

Le plan d'exécution de la requête ci-dessous peut être obtenu en précédant la requête de `EXPLAIN (ANALYZE, COSTS OFF, SUMMARY OFF)`. Une vision graphique plus complète est visible sur https://explain.dalibo.com/plan/92hce934hg875a80.

Ce plan montre que la vue `commandes_2014` est exécutée une seule fois, au début. Son résultat est ensuite relu par les deux opérations de regroupements définies dans la requête principale (voir les mentions `CTE Scan on commandes_2014`) :

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Append (actual time=1295.413..1835.434 rows=12 loops=1)
   CTE commandes_2014
     ->  Hash Join (actual time=97.041..781.457 rows=1226456 loops=1)
           Hash Cond: (l.numero_commande = c.numero_commande)
           ->  Seq Scan on lignes_commandes l (actual time=0.041..175.880
↪  rows=3141967 loops=1)
           ->  Hash (actual time=96.483..96.484 rows=390331 loops=1)
                 Buckets: 524288  Batches: 1  Memory Usage: 22393kB
                 ->  Seq Scan on commandes c (actual time=1.403..53.373 rows=390331
↪  loops=1)
                       Filter: ((date_commande >= '2014-01-01'::date) AND
↪  (date_commande <= '2014-12-31'::date))
                       Rows Removed by Filter: 609669
   ->  HashAggregate (actual time=1295.412..1295.415 rows=3 loops=1)
         Group Key: clients.type_client
         Batches: 1  Memory Usage: 24kB
         ->  Hash Join (actual time=109.251..1130.601 rows=1226456 loops=1)
               Hash Cond: (commandes_2014.client_id = clients.client_id)
               ->  CTE Scan on commandes_2014 (actual time=97.045..936.931
↪  rows=1226456 loops=1)
               ->  Hash (actual time=12.139..12.140 rows=100000 loops=1)
                     Buckets: 131072  Batches: 1  Memory Usage: 5712kB
                     ->  Seq Scan on clients (actual time=0.009..4.834 rows=100000
↪  loops=1)
   ->  HashAggregate (actual time=540.009..540.013 rows=9 loops=1)
         Group Key: co.code_pays
         Batches: 1  Memory Usage: 24kB
         ->  Hash Join (actual time=30.661..386.341 rows=1226456 loops=1)
               Hash Cond: (cl.contact_id = co.contact_id)
               ->  Hash Join (actual time=12.125..224.655 rows=1226456 loops=1)
                     Hash Cond: (r.client_id = cl.client_id)
                     ->  CTE Scan on commandes_2014 r (actual time=0.001..46.190
↪  rows=1226456 loops=1)
                     ->  Hash (actual time=12.059..12.060 rows=100000 loops=1)
                           Buckets: 131072  Batches: 1  Memory Usage: 5712kB
                           ->  Seq Scan on clients cl (actual time=0.010..4.815
↪  rows=100000 loops=1)
               ->  Hash (actual time=18.476..18.477 rows=110005 loops=1)
                     Buckets: 131072  Batches: 1  Memory Usage: 6181kB
                    ->  Seq Scan on contacts co (actual time=0.007..9.229 rows=110005
↪  loops=1)
```

Sans CTE, il aurait fallu décrire deux fois la même sous-requête, ou créer une vue `commandes_2014` qui aurait été appelée deux fois. Le coût d'exécution aurait été multiplié par deux, car il aurait fallu exécuter la sous-requête deux fois au lieu d'une.

Le temps de la requête, le résultat de la CTE peut être stocké en mémoire ou sur disque : PostgreSQL s'en occupe. Dans l'exemple ci-dessus, le résultat de la CTE est matérialisé (en mémoire ou sur disque) pour être relu. Cette méthode évite souvent le recours à des alternatives beaucoup plus lourdes comme les vues matérialisées ou les tables de travail temporaires. De nombreuses CTE ne nécessitent pas de matérialisation.

5.5.3 CTE et SELECT : syntaxe

```
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> ;
```

On utilise aussi les CTE pour améliorer la lisibilité des requêtes complexes.

On peut enchaîner plusieurs vues les unes à la suite des autres comme dans l'exemple ci-dessous.

Il est donc possible de définir un jeu de données en plusieurs étapes dans la même requête, dans plusieurs clauses séparées, avec par exemple une première CTE pour une sélection des données, puis de l'utiliser dans une deuxième CTE pour une jointure, puis faire l'agrégat dans une troisième CTE, avec un `SELECT` final ne procédant qu'à un formatage. Une requête plus complexe peut devenir plus lisible. PostgreSQL, en interne, sait rassembler toutes ces vues en une requête unique.

5.5.4 CTE, MATERIALIZED, et barrière d'optimisation

— Clause `MATERIALIZED` pour forcer une barrière d'optimisation
— Normalement automatique, et pour le meilleur
— Parfois un souci

Normalement, PostgreSQL « fusionne » le contenu d'un CTE avec les autres de la même requête pour choisir la manière dont il va récupérer les données

```
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=1.21..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))
```

Il pourrait y avoir des cas où le planificateur de PostgreSQL échoue à comprendre le bon plan. Un CTE peut permettre de le forcer à exécuter les CTE dans l'ordre qui a été précisé. Cela s'appelle une « barrière d'optimisation », et cela peut se faire grâce au mot-clé `MATERIALIZED` :

```
-- CTE avec MATERIALIZED
EXPLAIN
  WITH e AS MATERIALIZED ( 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)
```

La CTE est alors intégralement exécutée avec son filtre propre, matérialisée, puis son contenu est relu dans un autre nœud et le deuxième filtre appliqué .

En général, ce genre de manipulation n'est pas nécessaire. (Jusqu'en version 11 incluse, le `MATERIALIZED` était celui par défaut, et les CTE étaient une source fréquente de problèmes de performances.)

À l'inverse, PostgreSQL peut choisir de « matérialiser » une CTE, notamment quand elle est utilisée plusieurs fois dans une requête (voir l'exemple plus haut).

5.5.5 CTE en écriture

- CTE avec des requêtes en modification
  - avec `INSERT` / `UPDATE` / `DELETE`
  - et éventuellement `RETURNING`
- Exemples d'utilisation :
  - requête complexe
  - écriture dans plusieurs tables
  - déplacement de données (archivage…)

Les CTE peuvent contenir des ordres d'écriture. Grâce à `RETURNING`, les données modifiées/insérées/effacées peuvent être réutilisées dans une autre partie de la requête.

Cette technique permet des requêtes très complexes capables d'écrire dans plusieurs lignes, ou déplaçant des données d'une table à une autre, dans un seul ordre. Sans les CTE, il faudrait des transactions plus complexes, avec des tables de travail, du code procédural, voire des curseurs…

5.5.6 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.

En plus de ce cas d'usage simple, il est possible d'utiliser cette fonctionnalité pour déboguer 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.

5.5.7 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.

5.5.8 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
```

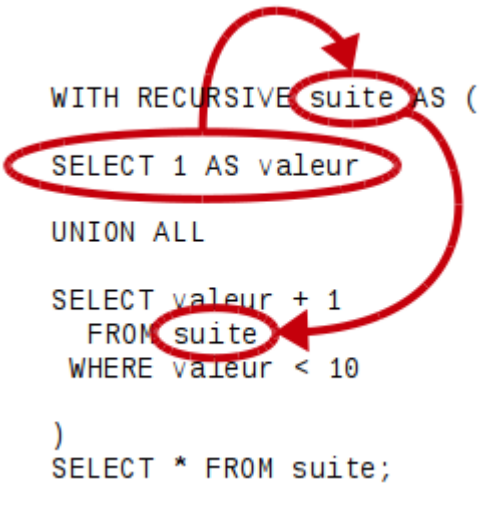
```
 10
```

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. À 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.

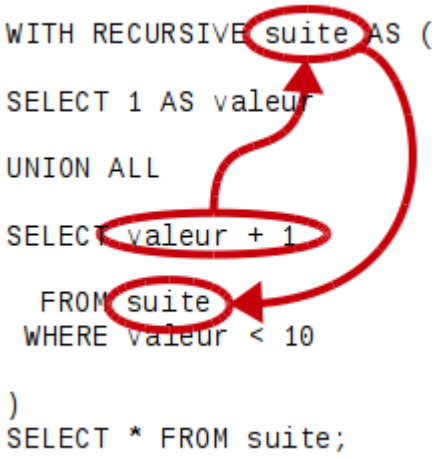
5.5.9 CTE récursive : principe

— 1ère étape : initialisation de la récursion



5.5.10 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;
```

5.5.11 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)
```

5.6 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 %, or 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` (`… SET salaire = salaire*1.05 WHERE …`) au lieu ;
- en verrouillant les données lues avec `SELECT FOR UPDATE` : ces verrous peuvent cependant ralentir l'application ;

— 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`.

5.6.1 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 n'interdira pas la lecture par une autre session, 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 contrepartie, 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.

Une application bien conçue interceptera l'erreur et décidera d'abandonner ou recommencer plus tard la mise à jour, au lieu d'attendre, peut-être très longtemps, que le verrou soit levé.

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
```

5.6.2 SKIP LOCKED

- `SELECT FOR UPDATE SKIP LOCKED`
- Utilité :
  - files d'attentes parallélisables

Une dernière fonctionnalité intéressante de `SELECT FOR UPDATE` est idéal dans le cas de différents *workers* qui consomment des tâches d'une table contenant une file d'attente. Il s'agit de la clause `SKIP LOCKED`.

En prenant une table représentant la file d'attente suivante, peuplée avec des données générées :

```
CREATE TABLE liste_taches_a_faire (id serial primary key, val text);
-- Données
INSERT INTO liste_taches_a_faire (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 liste_taches_a_faire
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)
```

La session va s'occuper ensuite de ces lignes avant de rendre la main, ce qui peut prendre plus ou moins de temps.

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. Le deuxième `SELECT` ne rendra la main lorsque la transaction n°1 se terminera, et s'apercevra peut-être ensuite que les lignes ont déjà été traitées par la première session.

Avec la clause `SKIP LOCKED`, les 10 premières verrouillées par la transaction n°1 seront ignorées puisque déjà verrouillées, et ce sont les 10 lignes suivantes qui seront selectionnées, verrouillées et retournées par l'ordre `SELECT` :

```
BEGIN TRANSACTION;

SELECT *
FROM liste_taches_a_faire
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 liste_taches_a_faire
 WHERE id IN (…);
COMMIT;
```

De même pour la seconde transaction, qui aura traité d'autres lignes en parallèle de la transaction n°1.

5.7 SERIALIZABLE SNAPSHOT ISOLATION

SSI : Serializable Snapshot Isolation
- 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 plusieurs modes d'isolation, pour un bon compromis entre fiabilité et performances. Par défaut, en mode `READ COMMITTED`, une transaction en cours d'exécution peut voir les modifications committés par d'autres transactions parallèles. En mode `REPEATABLE READ`, une transaction ne voit que les données telles qu'au début de la transaction, et ses propres modifications.

PostgreSQL fournit un autre 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.

L'intérêt est par exemple de pouvoir garantir que certaines validations touchant d'autres tables pourront s'effectuer sans *race conditions*, et ce sans poser des verrous trop gênants. L'application doit bien sûr être prête à ce que ses transactions échouent souvent, et à retenter ensuite.

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 sur un domaine assez large.

Dans l'exemple suivant, des enregistrements avec une colonne couleur contiennent 'blanc' ou 'rouge'. Deux utilisateurs essayent simultanément de convertir tous les enregistrements vers une couleur unique, mais chacun dans une direction opposée. Un utilisateur veut passer tous les blancs en rouge, et l'autre tous les rouges 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 'rouge'
    else 'blanc' end from x;
```

Session 1 :

```
set default_transaction_isolation = 'serializable';
```

```
begin;
update points set couleur = 'rouge'
where couleur = 'blanc';
```

Session 2 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'blanc'
where couleur = 'rouge';
```

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 = 'rouge'
   where couleur = 'blanc';
 commit;
```

Il n'y a pas de transaction concurrente pour gêner.

```
select * from points order by id;
```

```
 id | couleur
----+-------
  1 | rouge
  2 | rouge
  3 | rouge
  4 | rouge
  5 | rouge
  6 | rouge
  7 | rouge
  8 | rouge
  9 | rouge
 10 | rouge
(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

5.8 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

5.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s60_solutions.

Jointure latérale

Cette série de question utilise la base de TP **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin  /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

> 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` :

```
curl -kL https://dali.bo/tp_genealogie  -o genealogie.dump
createdb genealogie
pg_restore -O -d genealogie genealogie.dump
# le message d'erreur sur le schéma public est normal
```

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 ancètres de Adèle TAILLANDIER

Réseau social

La table `socialnet` peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_socialnet -o /tmp/socialnet.dump
createdb socialnet
pg_restore -O -d socialnet /tmp/socialnet.dump
# le message d'erreur sur le schéma public est normal
```

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 et restaurée ainsi :

```
curl -kL https://dali.bo/tp_brno2015 -o /tmp/brno2015.dump
createdb  brno2015
pg_restore -O -d brno2015 /tmp/brno2015.dump
# une erreur sur l'existence du schéma public est normale
```

> 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

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

6.1.3 Tables d'exemple

— Table `employes`
— Tables `population` et `continents`

La plupart des exemples utilisent une petite table `employes` à créer ainsi, dans la base de votre choix :

```
-- Si la table existe déjà, la détruire
DROP TABLE IF EXISTS employes CASCADE ;

-- Création de la table
CREATE TABLE employes (
matricule char(8) PRIMARY KEY,
nom       text NOT NULL,
service   text,
```

```
salaire   numeric(7,2)
);

-- Données
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('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);

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)
```

D'autres exemples utilisent une table des pays avec leur population, que voici :

```
-- Nettoyage des tables si elles existent
DROP TABLE IF EXISTS population CASCADE ;
DROP TABLE IF EXISTS continents CASCADE ;

-- Tables
CREATE TABLE continents (
    continent text PRIMARY KEY
) ;
CREATE TABLE population (
    pays text PRIMARY KEY,
    population numeric,
    superficie numeric,
    densite numeric,
    continent text REFERENCES continents NOT NULL
) ;

-- Données des continents
INSERT INTO continents
VALUES ('Amérique du Nord'), ('Europe'), ('Asie'),
('Amérique latine. Caraïbes'), ('Afrique'), ('Antarctique'),
('Océanie');

-- Données des pays
INSERT INTO population (pays, population, superficie, densite, continent)
VALUES ('Allemagne', 82.7, 357, 232, 'Europe'),
('Autriche', 8.5, 84, 101, 'Europe'),
('Belgique', 11.1, 31, 364, 'Europe'),
('Biélorussie', 9.4, 208, 45, 'Europe'),
('Bulgarie          ', 7.2, 111, 65, 'Europe'),
('Croatie', 4.3, 57, 76, 'Europe'),
('Danemark', 5.6, 43, 130, 'Europe'),
('Espagne', 46.9, 506, 93, 'Europe'),
('Estonie', 1.3, 45, 29, 'Europe'),
```

```
('Finlande', 5.4, 337, 16, 'Europe'),
('France métropolitaine', 64.3, 552, 117, 'Europe'),
('Grèce', 11.1, 132, 84, 'Europe'),
('Hongrie', 10.0, 93, 107, 'Europe'),
('Irlande', 4.6, 70, 66, 'Europe'),
('Italie', 61.0, 301, 202, 'Europe'),
('Lettonie', 2.1, 65, 32, 'Europe'),
('Lituanie', 3.0, 65, 46, 'Europe'),
('Luxembourg', 0.5, 3, 205, 'Europe'),
('Malte', 0.4, 0, 1358, 'Europe'),
('Moldavie', 3.5, 34, 103, 'Europe'),
('Norvège', 5.0, 324, 13, 'Europe'),
('Pays-Bas', 16.8, 42, 404, 'Europe'),
('Pologne', 38.2, 312, 118, 'Europe'),
('Portugal', 10.6, 92, 115, 'Europe'),
('République tchèque', 10.7, 79, 136, 'Europe'),
('Roumanie', 21.7, 238, 91, 'Europe'),
('Royaume-Uni', 63.1, 242, 260, 'Europe'),
('Féd. de Russie', 142.8, 17098, 8, 'Europe'),
('Serbie', 9.5, 88, 108, 'Europe'),
('Slovaquie', 5.5, 49, 111, 'Europe'),
('Slovénie', 2.1, 20, 102, 'Europe'),
('Suède', 9.6, 450, 21, 'Europe'),
('Suisse', 8.1, 41, 196, 'Europe'),
('Ukraine', 45.2, 604, 75, 'Europe'),
('Afrique du Sud', 52.8, 1221, 43, 'Afrique'),
('Algérie', 39.2, 2382, 16, 'Afrique'),
('Burkina Faso ', 16.9, 274, 62, 'Afrique'),
('Côte-d''Ivoire', 20.3, 322, 63, 'Afrique'),
('Égypte', 82.1, 1002, 82, 'Afrique'),
('Éthiopie', 94.1, 1104, 85, 'Afrique'),
('Ghana', 25.9, 239, 109, 'Afrique'),
('Kenya', 44.4, 581, 76, 'Afrique'),
('Madagascar', 22.9, 587, 39, 'Afrique'),
('Maroc', 33.0, 447, 74, 'Afrique'),
('Mozambique', 25.8, 802, 32, 'Afrique'),
('Niger', 17.8, 1267, 14, 'Afrique'),
('Nigéria', 173.6, 924, 188, 'Afrique'),
('Ouganda', 37.6, 242, 156, 'Afrique'),
('Rép. dém. du Congo ', 67.5, 2345, 29, 'Afrique'),
('Soudan', 14.1, 197, 72, 'Afrique'),
('Tanzanie', 49.3, 945, 52, 'Afrique'),
('Tunisie', 11.0, 164, 67, 'Afrique'),
('Zimbabwe', 14.1, 391, 36, 'Afrique'),
('Canada', 35.2, 9985, 4, 'Amérique du Nord'),
('États-Unis', 320.1, 9629, 33, 'Amérique du Nord'),
('Argentine', 41.4, 2780, 15, 'Amérique latine. Caraïbes'),
('Brésil', 200.4, 8515, 24, 'Amérique latine. Caraïbes'),
('Chili', 17.6, 756, 23, 'Amérique latine. Caraïbes'),
('Colombie', 48.3, 1142, 42, 'Amérique latine. Caraïbes'),
('Cuba', 11.3, 110, 102, 'Amérique latine. Caraïbes'),
('Équateur', 15.7, 256, 56, 'Amérique latine. Caraïbes'),
('Guatemala ', 15.5, 109, 142, 'Amérique latine. Caraïbes'),
('Mexique ', 122.3, 1964, 62, 'Amérique latine. Caraïbes'),
('Pérou', 30.4, 1285, 24, 'Amérique latine. Caraïbes'),
('Venezuela', 30.4, 912, 33, 'Amérique latine. Caraïbes'),
('Afghanistan ', 30.6, 652, 47, 'Asie'),
```

```
('Arabie Saoudite', 28.8, 2005, 13, 'Asie'),
('Bangladesh ', 156.6, 144, 1087, 'Asie'),
('Chine', 1385.6, 9597, 144, 'Asie'),
('Corée du Nord ', 24.9, 121, 207, 'Asie'),
('Corée du Sud', 49.3, 100, 495, 'Asie'),
('Inde', 1252.1, 3287, 381, 'Asie'),
('Indonésie', 249.9, 1911, 131, 'Asie'),
('Iraq', 33.8, 435, 77, 'Asie'),
('Iran', 77.4, 1629, 47, 'Asie'),
('Japon', 127.1, 378, 336, 'Asie'),
('Malaisie', 29.7, 331, 90, 'Asie'),
('Myanmar ( Birmanie)', 53.3, 677, 79, 'Asie'),
('Népal', 27.8, 147, 189, 'Asie'),
('Ouzbékistan', 28.9, 447, 65, 'Asie'),
('Pakistan', 182.1, 796, 229, 'Asie'),
('Philippines ', 98.4, 300, 328, 'Asie'),
('Sri Lanka ', 21.3, 66, 324, 'Asie'),
('Syrie', 21.9, 185, 118, 'Asie'),
('Thaïlande', 67.0, 513, 131, 'Asie'),
('Turquie', 74.9, 784, 96, 'Asie'),
('Viêt Nam', 91.7, 331, 276, 'Asie'),
('Yémen', 24.4, 528, 46, 'Asie') ;
```

```
-- Échantillon
SELECT * FROM population LIMIT 5;
```

```
       pays        | population | superficie | densite | continent
-------------------+------------+------------+---------+-----------
 Allemagne         |       82.7 |        357 |     232 | Europe
 Autriche          |        8.5 |         84 |     101 | Europe
 Belgique          |       11.1 |         31 |     364 | Europe
 Biélorussie       |        9.4 |        208 |      45 | Europe
 Bulgarie          |        7.2 |        111 |      65 | Europe
```

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.

Ainsi, on peut calculer :

- le salaire moyen des employés de la table avec la fonction `avg()` ;
- les salaires maximum et minimum avec les fonctions `max()` et `min()` ;
- 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
```

Ici, la base de données réalise les calculs sur l'ensemble des données de la table, car il n'y a ni clause `WHERE` ni jointure. Ne s'affiche que le résultat du calcul, pas les données qui ont été utilisées.

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
```

Une limitation : 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 avec `GROUP BY` :

```
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;
                               ^
```

En effet, cela reviendrait à afficher une donnée agrégée (`avg()`) sur une ligne qui fait partie de l'agrégat. Ce n'est pas prévu par le SQL dans sa version originale. Nous verrons plus loin les fonctions de fenêtrage pour faire rigoureusement ce genre de chose.

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
```

Figure 6/ .1 – Opérateur GROUP BY

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 :

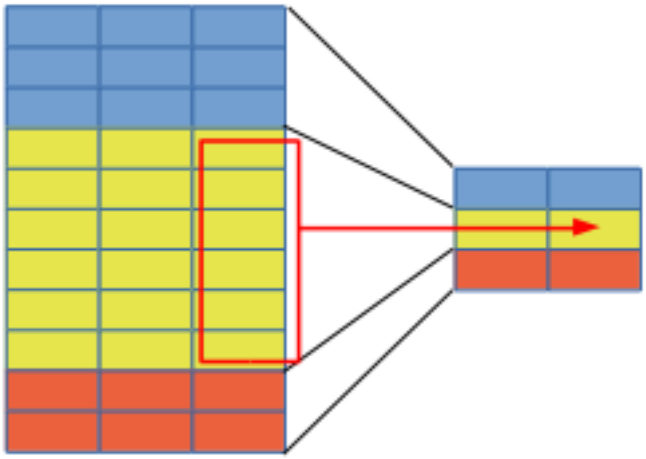
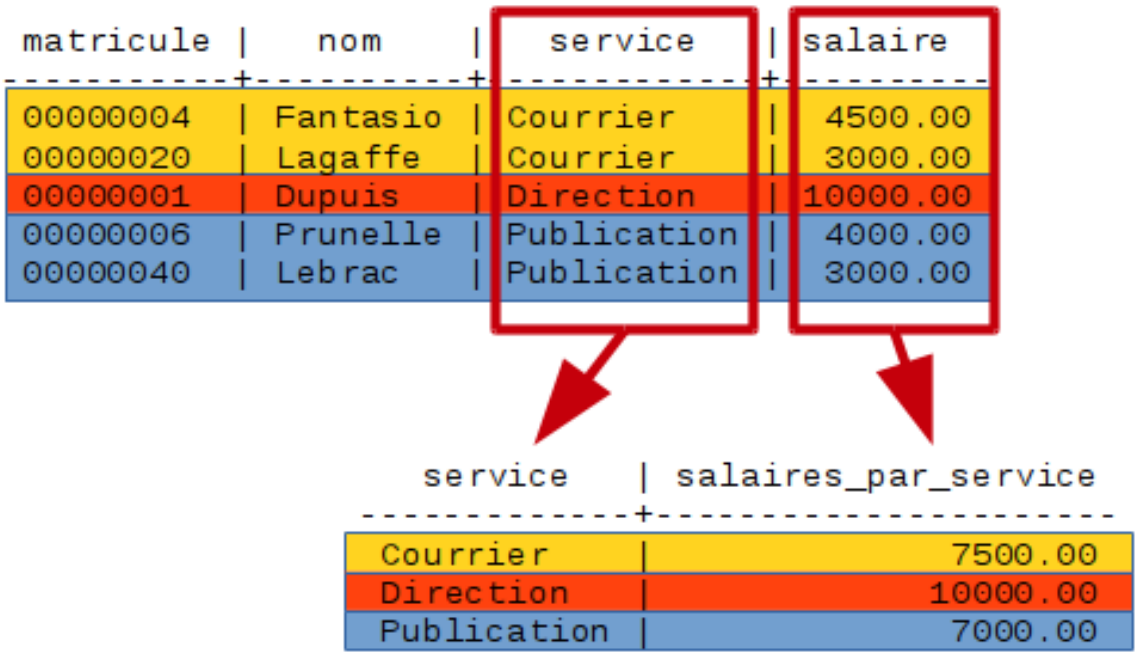


Figure 6/ .2 – Agrégats

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 : syntaxe et exemple

```
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.

Pour avoir un total sur tous les services, une première technique est de le calculer séparément, et de combiner le résultat des deux requêtes d'agrégation avec `UNION ALL`, si les lignes 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`.

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)
```

6.2.6 Créer un tableau avec un agrégat : array_agg

```
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}
```

Il est possible de réaliser la même chose mais pour obtenir un tableau plutôt qu'une chaîne de caractère.

Un tableau est un type composé dans PostgreSQL. Plusieurs valeurs peuvent être stockées dans un unique champ d'une ligne. Le maniement ensuite est bien sûr un peu plus compliqué.

6.3 CLAUSE FILTER

- Clause `FILTER`
- Utilité :
  - filtrer les données sur les agrégats
  - évite les expressions `CASE` complexes
- SQL :2003

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

- Syntaxe fastidieuse :

```
SELECT count(*) AS compte_pays,
       count(CASE WHEN continent='Europe' THEN 'Oui'
                  ELSE NULL
              END) AS compte_pays_europeens
FROM   population p ;

 compte_pays | compte_pays_europeens
-------------+-----------------------
          88 |                    34
```

Le premier calcul `count(*)` compte simplement le nombre de lignes. Le second, `count(CASE … ELSE NULL END)`, compte les pays européens. Pour cela, il utilise un opérateur `CASE` qui renvoie `NULL` si le pays n'est pas en Europe, et `Oui` sinon (n'importe quelle valeur non nulle aurait fait l'affaire), et au final le `count` renvoie le nombre de valeurs non nulles calculées sur la ligne.

Une variante de cette requête peut s'écrire avec `sum()`, qui somme des 1 et des 0 selon que le pays est en Europe ou pas :

```
SELECT count(*) AS compte_pays,
       sum(CASE WHEN continent='Europe' THEN 1 ELSE 0 END)
       AS compte_pays_europeens
FROM   population p ;
```

Le `CASE` peut devenir arbitrairement complexe.

Cela fonctionne, mais dès que l'on a besoin d'avoir de multiples filtres, ou des 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 (WHERE…)

— La même requête écrite avec la clause `FILTER` :

```
SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE continent='Europe')
                AS compte_pays_europeens
FROM population ;
```

La clause `FILTER (WHERE…)` simplifie le calcul. Il suffit d'y préciser le critère de filtrage. Les lignes ne le respectant pas seront ignorées.

La clause `WHERE` peut elle-même devenir aussi complexe que l'on veut.

6.4 FONCTIONS DE FENÊTRAGE

Des fonctionnalités trop peu connues

6.4.1 Fenêtrage et regroupement : premier 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 ici de perdre le détail des données tout en disposant des données agrégées dans le résultat de la requête.

Les calculs du champ `total_salaire_service` sont ceux que l'on a déjà pu effectuer ainsi :

```
SELECT service, sum(salaire)
FROM   employes
GROUP BY service ;
```

Les données sont regroupées par service :

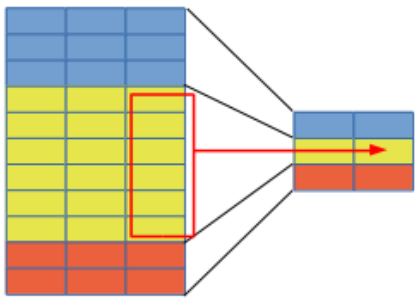


FIGURE 6/ .3 – Agrégation de lignes

Avec le fenêtrage, le salaire est sommé par `service` (comme indiqué par la clause `OVER`), et le résultat est reporté sur chaque ligne correspondant à ce service. La valeur peut se répéter d'une ligne à l'autre.

```
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
```

Ce schéma résume le principe :

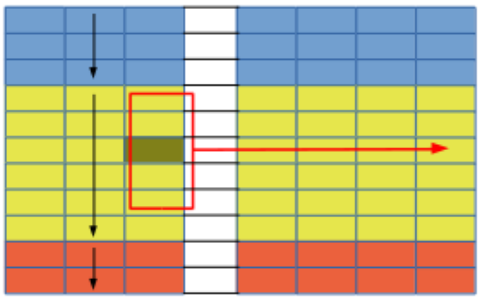


Figure 6/ .4 – Fonction de fenêtrage

Sans les fonctions de fenêtrage, le code SQL est beaucoup plus compliqué. Les possibilités menant au même résultat sont :

— des sous-requêtes effectuées sur chaque ligne :

```
SELECT e.matricule, e.salaire, e.service,
       (SELECT SUM(salaire) FROM employes e2 WHERE e2.service=e.service)
       AS total_salaire_service
FROM employes e;
```

— des sous-requêtes par ligne effectuées via une clause `LATERAL` :

```
SELECT e.matricule, e.salaire, e.service,
       el.total_salaire_service
FROM employes e,
LATERAL (SELECT SUM(salaire) AS total_salaire_service
         FROM employes e2
         WHERE e2.service=e.service
         ) el
;
```

— un pré-calcul par une sous-requête dans le `FROM`, ou un CTE comme ici, joint ensuite à la table originale :

```
WITH el AS (
         SELECT service, SUM(salaire) AS total_salaire_service
         FROM employes e2
         GROUP BY service
         )
SELECT e.matricule, e.salaire, e.service,
       el.total_salaire_service
FROM employes e
INNER JOIN el USING (service)
;
```

Tout cela est plus compliqué, voire illisible et source d'erreurs quand les critères deviennent complexes.

Les fonctions de fenêtrage sont optimisées et offrent de bien meilleures performances que les requêtes ci-dessus !

Notamment, la syntaxe avec le fenêtrage ne parcourt la table qu'un fois, au contraire de toutes les requêtes en SQL plus simple ci-dessus, qui la parcourent deux fois ou plus.

6.4.2 Fonctions de fenêtrage : utilisation

- Fonctions de fenêtrage avec `OVER ( … )`
  - travail sur des ensembles de données, regroupés et triés, indépendamment de la requête principale
- Utilisation :
  - référence à d'autres lignes de l'ensemble de données
  - différents critères d'agrégation dans la même requête
  - fonctions de classement
- Exemples :
  - ratios entre ligne et ensemble
  - sommes courantes, moyennes glissantes
  - évolution entre deux lignes
- Performances

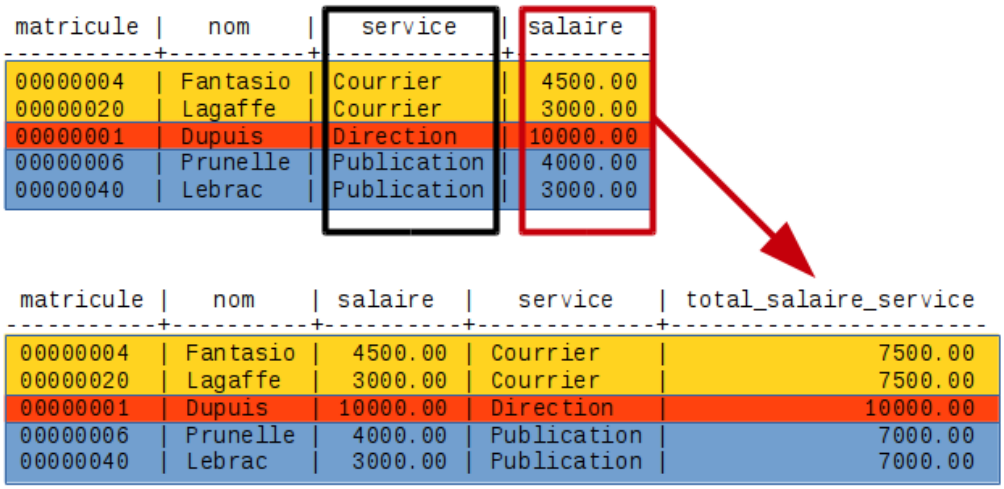
L'exemple précédent est très simple mais il montre déjà que les fonctions de fenêtrage permettent de travailler sur un ensemble de lignes, filtrées mais que l'on pourra aussi ordonner, tout en affichant le résultat d'une ligne. Plusieurs fonctions de fenêtrage sont utilisables (par exemple `sum() OVER ()` et `avg() SUM OVER()`).

La clause `OVER` permet de définir la façon dont les données sont regroupées uniquement pour la colonne définie. La clause `PARTITION BY` définit un regroupement mais il y a déjà d'autres possibilités.

Il est déjà possible de calculer un ratio entre une valeur de la ligne et un agrégat par fenêtrage par une simple division. Avec une syntaxe un peu plus complexe, les fonctions de fenêtrage permettent de calculer des sommes courantes, ou l'évolution d'une valeur d'une ligne à l'autre.

La syntaxe est plus simple que des requêtes SQL complexes, et les performances sont meilleures.

6.4.3 Fenêtrage et regroupement : OVER (PARTITION BY ...)



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

matricule	nom	salaire	service	total_salaire_service
00000004	Fantasio	4500.00	Courrier	7500.00
00000020	Lagaffe	3000.00	Courrier	7500.00
00000001	Dupuis	10000.00	Direction	10000.00
00000006	Prunelle	4000.00	Publication	7000.00
00000040	Lebrac	3000.00	Publication	7000.00

```
SELECT …
 <agregation> OVER (PARTITION BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

— NB : rien à voir avec le partitionnement d'une table

Le terme `PARTITION BY` permet d'indiquer les critères de regroupement de la fenêtre sur laquelle on souhaite travailler.

Ne pas confondre cette clause avec le « partitionnement » d'une table, qui consiste, en simplifiant, à découper une grande table en sous-tables physiques plus petites. Les deux concepts n'ont rien à voir. `PARTITION BY` consiste uniquement à regrouper logiquement des lignes récupérées par la requête.

6.4.4 Fenêtrage et tri : OVER (ORDER BY ...)

```
OVER (ORDER BY matricule)
```

- Utilité :
  - numéroter les lignes : `row_number()`
  - classer des résultats : `rank()`, `dense_rank()`
  - faire appel à d'autres lignes du résultat : `lead()`, `lag()`
- Le tri du fenêtrage est indépendant du `ORDER BY` des lignes

La clause `OVER (ORDER BY…)` permet de calculer une fonction qui dépend d'un tri comme des fonctions de numérotation de ligne. Cet ordre est indépendant de celui qu'une clause `ORDER BY` peut imposer aux lignes une fois qu'elles ont toutes été calculées.

6.4.5 Fenêtrage et tri : row_number()

- Pour numéroter des lignes :

```
SELECT row_number() OVER (ORDER BY nom),
       matricule, nom
FROM   employes
ORDER BY matricule;

 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          5 | 00000006  | Prunelle
          3 | 00000020  | Lagaffe
          4 | 00000040  | Lebrac
(5 lignes)
```

La fonction `row_number()` permet de numéroter les lignes selon un critère de tri défini dans la clause `OVER`. Dans l'exemple ci-dessus, le `row_number` suit l'ordre alphabétique (Dupuis puis Fantasio puis Lagaffe, etc.), puis les lignes calculées sont ordonnées par matricule.

La clause `(ORDER BY …)` peut contenir plusieurs colonnes, ou des précisions comme `DESC` ou `NULLS FIRST`.

6.4.6 Fenêtrage et tri : numéroter des lignes sans critère

— Pour numéroter des lignes :

```
SELECT row_number() OVER (),
       matricule, nom
FROM   employes
ORDER BY nom DESC ;
```

```
 row_number | matricule |   nom
------------+-----------+----------
          3 | 00000006  | Prunelle
          5 | 00000040  | Lebrac
          4 | 00000020  | Lagaffe
          2 | 00000004  | Fantasio
          1 | 00000001  | Dupuis
```

Si l'on cherche simplement à numéroter les lignes d'un résultat, il faut connaître `row_number() OVER ()`.

Attention, la numérotation a lieu ici *avant* le `ORDER BY` qui trie par `nom`. En conséquence, il peut aussi y avoir un piège avec `LIMIT` :

```
SELECT row_number() OVER (),
       matricule, nom
FROM   employes
ORDER BY nom DESC
LIMIT 3;
```

```
 row_number | matricule |   nom
------------+-----------+----------
          3 | 00000006  | Prunelle
          5 | 00000040  | Lebrac
          4 | 00000020  | Lagaffe
```

Si l'on tient à un affichage allant strictement de 1 à N, l'idéal est d'avoir un tri du `row_number()` cohérent avec l'`ORDER BY` final.

Si, pour une raison ou une autre, ce n'est pas possible ou facile, un CTE permet d'ajouter `row_number()` tout à la fin :

```
WITH vraierequete AS (
  SELECT matricule, nom
  FROM   employes
  ORDER BY nom DESC
  LIMIT  3 )
SELECT row_number() OVER(), vraierequete.*
FROM vraierequete ;
```

```
 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000006  | Prunelle
          2 | 00000040  | Lebrac
          3 | 00000020  | Lagaffe
```

6.4.7 Fenêtrage et tri : rang

— Rang selon le salaire :

```
SELECT matricule, nom, salaire, service,
       rank() OVER (ORDER BY salaire),
       dense_rank() OVER (ORDER BY salaire) -- pas de trous
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
```

La fonction de fenêtrage `rank()` renvoie un classement en autorisant des trous dans la numérotation quand il y a ex-aequos, et `dense_rank()` le classement sans trous.

Ces fonctions seraient beaucoup plus compliquées à écrire en SQL.

6.4.8 Fenêtrage et tri : somme glissante (exemple)

— 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
```

`OVER (ORDER BY …)` peut aussi être utilisé avec `sum()`. Là où `SUM () OVER (PARTITION BY….)` calculait une somme d'un regroupement, `SUM() OVER (ORDER BY…)` calcule la somme courante selon le tri en cours.

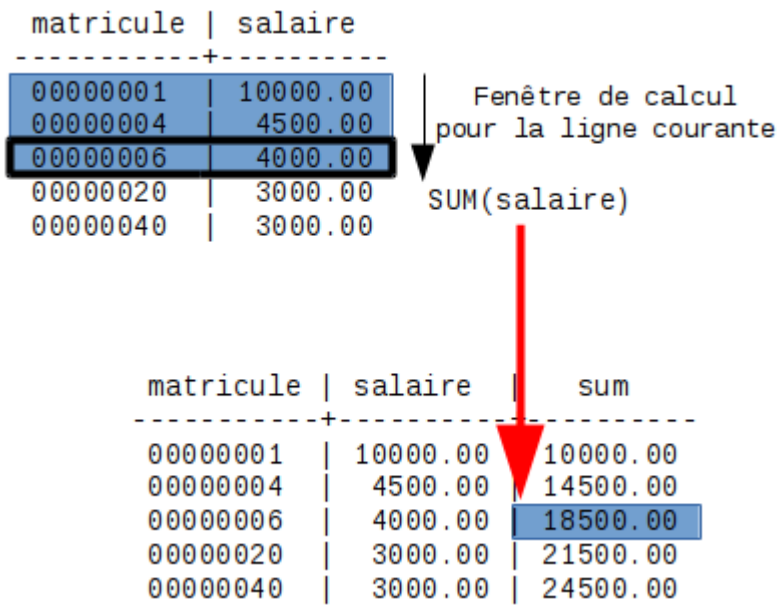
Là encore, attention à la cohérence entre le tri du calcul et celui de l'affichage si cela a une importance :

```
SELECT matricule, salaire,
       SUM(salaire) OVER (ORDER BY matricule)
FROM   employes
ORDER BY matricule DESC ;
```

```
 matricule | salaire  |   sum
-----------+----------+----------
 00000040  |  3000.00 | 24500.00
 00000020  |  3000.00 | 21500.00
 00000006  |  4000.00 | 18500.00
 00000004  |  4500.00 | 14500.00
 00000001  | 10000.00 | 10000.00
```

6.4.9 Fenêtrage et tri : somme glissante (principe)

```
SUM(salaire) OVER (ORDER BY matricule)
```



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.

Par défaut, la fenêtre de calcul intègre les lignes précédant celle en cours, incluse, ce qui correspond à ceci avec la syntaxe complète :

```
SELECT matricule, salaire,
       SUM(salaire) OVER (
        ORDER BY matricule
        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

```
       )
FROM   employes
ORDER BY matricule  ;
```

Nous verrons que cette fenêtre peut se changer.

6.4.10 Fenêtrage : regroupement et tri

- On peut combiner :
  - `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.

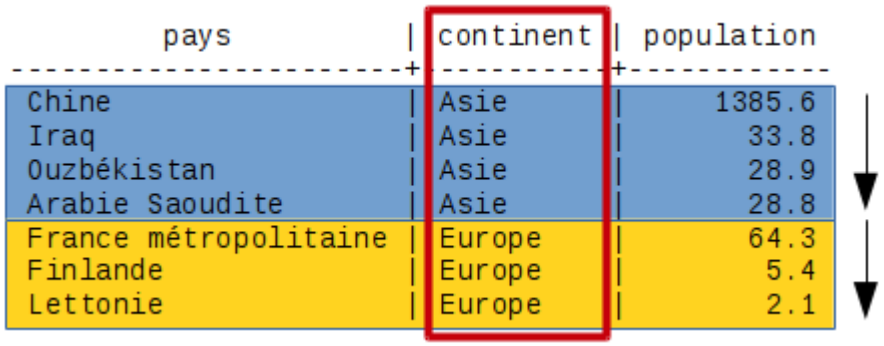
Ci-dessus, le Nigéria et les États-Unis sont bien les pays les plus peuplés de leur continent respectif.

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

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.

6.4.13 Fonctions analytiques

- `first_value ()`, `last_value`, `nth ()`, `lag()`, `lead()`...
- À utiliser avec les fonctions de fenêtrage
- Utilité :
  - faire référence à d'autres lignes du même ensemble
  - évite les auto-jointures complexes et lentes

Sans les fonctions analytiques, il serait 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
```

6.4.14 lead() et lag() : exemple

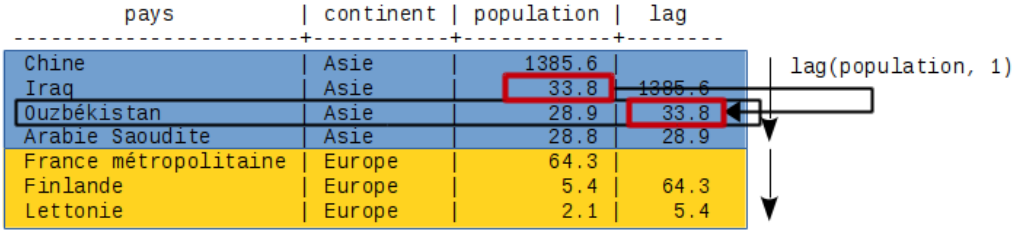
```
SELECT pays, continent, population,
       lag(population) OVER (PARTITION BY continent
                             ORDER BY population DESC)
FROM   population
WHERE  continent in ('Europe','Afrique');

         pays          | continent | population |  lag
-----------------------+-----------+------------+--------
 Nigéria               | Afrique   |      173.6 |
 Éthiopie              | Afrique   |       94.1 | 173.6
 Égypte                | Afrique   |       82.1 |  94.1
 …
 Tunisie               | Afrique   |       11.0 |  14.1
 Féd. de Russie        | Europe    |      142.8 |
 Allemagne             | Europe    |       82.7 | 142.8
 France métropolitaine | Europe    |       64.3 |  82.7
 Royaume-Uni           | Europe    |       63.1 |  64.3
 Italie                | Europe    |       61.0 |  63.1
 …
 Malte                 | Europe    |        0.4 |   0.5
(53 lignes)
```

L'exemple ci-dessus utilise une fonction de fenêtrage par continent.

Avec `lag()`, il est possible de ramener la valeur de la ligne précédente selon le tri sur `population` sur la ligne en cours. `NULL` est renvoyé lorsque la valeur n'est pas accessible dans la fenêtre de données, comme pour le pays le plus peuplé de chaque continent.

6.4.15 lead() et lag() : principe

```
lag(population) OVER (PARTITION BY continent
                      ORDER BY population DESC)
```



6.4.16 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

Figurent plus haut des exemples avec `dense_rank` ou `row_number`. Il existe également les fonctions suivantes :

- `last_value(colonne)` : renvoie la dernière valeur pour la colonne ;
- `nth_value(colonne, n)` : renvoie la `n`e valeur (en comptant à partir de 1) pour la colonne ;
- `lag(colonne, n)` : renvoie la valeur située en `n`e position **avant** la ligne en cours pour la colonne ;
- `lead(colonne, n)` : renvoie la valeur située en `n`e position **après** la ligne en cours pour la colonne ;
  - pour ces deux fonctions, le `n` est facultatif et vaut 1 par défaut : `lead(colonne)` est équivalente à `lead(colonne, 1)` et `lag(colonne)` est équivalente à `lag(colonne, 1)`, pour récupérer les valeurs suivante ou précédente de la colonne ;
  - ces deux fonctions acceptent un troisième argument facultatif spécifiant la valeur à renvoyer si aucune valeur n'est trouvée en `n`e position avant ou après. Par défaut, `NULL` est renvoyé.

La liste complète est dans la documentation[1].

6.4.17 first/last/nth_value : exemple

```
SELECT pays, continent, population,
       first_value(population)
           OVER (PARTITION BY continent
                 ORDER BY population DESC)
FROM   population
WHERE  continent in ('Europe','Afrique');
```

```
         pays          | continent | population | first_value
-----------------------+-----------+------------+-------------
 Nigéria               | Afrique   |      173.6 |       173.6
 Éthiopie              | Afrique   |       94.1 |       173.6
 Égypte                | Afrique   |       82.1 |       173.6
 …
 Féd. de Russie        | Europe    |      142.8 |       142.8
 Allemagne             | Europe    |       82.7 |       142.8
 France métropolitaine | Europe    |       64.3 |       142.8
 …
```

Lorsque 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 incluse (clause implicite `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

Cela pose un souci avec `last_value()` qui récupère la dernière valeur d'une fenêtre où la dernière valeur triée est toujours celle de la ligne :

```
-- Faux !
SELECT pays, continent, population,
       last_value(population) OVER (
              PARTITION BY continent
              ORDER BY population DESC)
FROM   population
WHERE  continent in ('Europe','Afrique');
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Nigéria               | Afrique   |      173.6 |      173.6
 Éthiopie              | Afrique   |       94.1 |       94.1
 Égypte                | Afrique   |       82.1 |       82.1
 Rép. dém. du Congo    | Afrique   |       67.5 |       67.5
```

[1] https://docs.postgresql.fr/current/functions-window.html#FUNCTIONS-WINDOW-TABLE

```
 Afrique du Sud        | Afrique    |       52.8 |       52.8
…
```

Nous allons voir qu'il est alors nécessaire de redéfinir le comportement de la fenêtre visible pour que la fonction se comporte comme attendu.

6.4.18 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) ;
```

— Plusieurs fenêtres possibles

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 précédant 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);
```

6.4.19 Fenêtre de travail

— La fenêtre d'`OVER (ORDER BY …)` par défaut est :

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

— Toutes les lignes :

```
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

— Trois modes possibles :
  — par `RANGE` (par plage ; ci-dessus)
  — par `ROWS` (par nombre de lignes)
  — par `GROUPS` (selon valeurs des lignes)

Lorsque 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 incluse (clause implicite `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

La clause `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` inclut toutes les lignes avant et après ligne courante (voir plus haut l'exemple avec `last_value`).

6.4.20 Fenêtre de travail avec RANGE

— 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 UNBOUNDED PRECEDING
            AND     UNBOUNDED FOLLOWING )
```

La première option est de définir des bornes floues avec `UNBOUNDED PRECEDING/FOLLOWING` comme vu précédemment.

6.4.21 Fenêtre de travail avec RANGE : exemple

```
SELECT pays, continent, population,
       last_value(population) OVER (
            PARTITION BY continent
            ORDER BY population DESC
            RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
            )
FROM   population
WHERE  continent in ('Europe','Afrique');
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Nigéria               | Afrique   |      173.6 |       11.0
 Éthiopie              | Afrique   |       94.1 |       11.0
 Égypte                | Afrique   |       82.1 |       11.0
 Rép. dém. du Congo    | Afrique   |       67.5 |       11.0
 …
 Tunisie               | Afrique   |       11.0 |       11.0
 Féd. de Russie        | Europe    |      142.8 |        0.4
 Allemagne             | Europe    |       82.7 |        0.4
 France métropolitaine | Europe    |       64.3 |        0.4
 …
 Malte                 | Europe    |        0.4 |        0.4
```

L'exemple ci-dessus avec `last_value`, pour ramener la valeur la plus faible de l'ensemble, se corrige donc ainsi :

6.4.22 Fenêtre de travail avec ROWS

- 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 )
```

- Exemple : moyenne glissante

La syntaxe `ROWS BETWEEN xxx PRECEDING  AND xxx FOLLOWING` permet de restreindre la fenêtre à

un certain nombre de lignes avant et après la ligne en cours. Un cas d'utilisation est constitué par les moyennes glissantes.

Exemple :

Le jeu de valeur suivant donne un salaire brut sur douze mois qui varie chaque mois. La valeur moyenne du salaire mensuel sur toute l'année se calcule avec `avg () OVER ()`, et est une moyenne sur toutes les données. Avec `avg() OVER(ORDER BY mois)` se calcule une moyenne sur la fenêtre par défaut, c'est-à-dire depuis le premier mois jusque la ligne courante. Enfin, le dernier champ utilise la syntaxe `ROWS` pour définir une moyenne mobile arithmétique, c'est-à-dire que la valeur sur la ligne est la moyenne des trois valeurs des lignes précédente, suivante, et en cours (sauf pour les première et dernière lignes qui ne moyennent que deux valeurs).

```
WITH salaires (mois, brut) AS
(VALUES (6,2000),(7,2000),(8,1500),(2,2000),
 (9,2500), (10,2000),(4,2000),(11,1000),(12,0),
 (3,2000), (5,2000),(1,0)
)
SELECT mois, brut,
round(avg(brut) OVER() ,1) AS moy_annee,
round(avg(brut) OVER(ORDER BY mois) ,1) AS moy_depuis_janvier,
round(avg(brut) OVER(ORDER BY mois ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) ,1)
  AS moy_glissante
FROM salaires ;
```

```
 mois | brut | moy_annee | moy_depuis_janvier | moy_glissante
------+------+-----------+--------------------+---------------
    1 |    0 |    1583.3 |                0.0 |        1000.0
    2 | 2000 |    1583.3 |             1000.0 |        1333.3
    3 | 2000 |    1583.3 |             1333.3 |        2000.0
    4 | 2000 |    1583.3 |             1500.0 |        2000.0
    5 | 2000 |    1583.3 |             1600.0 |        2000.0
    6 | 2000 |    1583.3 |             1666.7 |        2000.0
    7 | 2000 |    1583.3 |             1714.3 |        1833.3
    8 | 1500 |    1583.3 |             1687.5 |        2000.0
    9 | 2500 |    1583.3 |             1777.8 |        2000.0
   10 | 2000 |    1583.3 |             1800.0 |        1833.3
   11 | 1000 |    1583.3 |             1727.3 |        1000.0
   12 |    0 |    1583.3 |             1583.3 |         500.0
(12 lignes)
```

Noter que l'ordre d'affichage selon le mois n'est qu'un artefact dû à l'algorithme de calcul. Cet ordre n'est pas garanti sans ajout d'une clause `ORDER BY` finale, qui ne changerait pas les valeurs des données.

6.4.23 Fenêtre de travail avec GROUPS

- 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 selon `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 selon `ORDER BY`

```
OVER (PARTITION BY …
      ORDER BY …
      GROUPS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

6.4.24 Définition de la fenêtre : EXCLUDE

- Lignes à exclure de la fenêtre de données
- `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 les valeurs identiques à la ligne courante dans l'ordre mais pas la ligne courante
- `EXCLUDE NO OTHERS` : pas d'exclusion (par défaut)

La clause `EXCLUDE` permet d'ignorer des lignes dans la fenêtre, par exemple la ligne courante seule, ou les lignes de même valeur que la valeur courante, avec ou sans la ligne courante.

6.5 WITHIN GROUP

— `WITHIN GROUP`
— Utilité :
  — calcul de médianes, centiles

La clause `WITHIN GROUP` est une 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.

6.6 REGROUPEMENT AVANCÉS

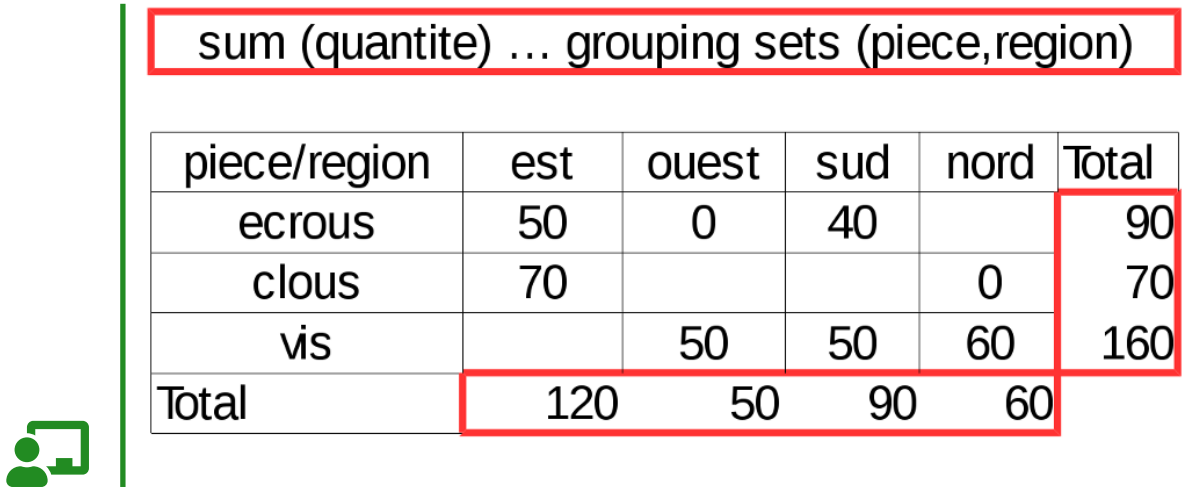
- `GROUPING SETS / ROLLUP / CUBE`
- Extension de `GROUP BY`
- Utilité :
  - plusieurs agrégations différentes dans la même requête
  - moins de requêtes, meilleures performances

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 : données d'exemple

```
DROP TABLE IF EXISTS stock  ;

CREATE TABLE stock (
  piece     text,
  region    text,
  quantite  integer) ;

INSERT INTO stock
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) ;
```

6.6.2 GROUPING SETS : exemple visuel



Le but est ici d'obtenir les totaux indiqués en rouge : des totaux par type de pièces, et par région, indépendamment. La clause `GROUPING SETS` permet cela en précisant juste les champs.

6.6.3 Émuler les GROUPING SETS avec GROUP BY

Sans `GROUPING SET` : deux `GROUP BY`

```
SELECT piece, NULL AS region, sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region, sum(quantite)
  FROM STOCK
  GROUP BY region;
```

```
 piece  | region | sum
--------+--------+-----
 vis    |        | 160
 ecrous |        |  90
 clous  |        |  70
        | ouest  |  50
        | nord   |  60
        | est    | 120
        | sud    |  90
```

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 et en prévoyant des colonnes vides à `NULL` pour les champs non calculés.

Surtout, cette requête duplique beaucoup de logique, ce qui va poser souci si elle est plus complexe et est souvent modifiée. De plus, son plan d'exécution ci-dessous indique que PostgreSQL procède à deux lectures séparées. Ce peut être particulièrement coûteux sur une grande table :

```
EXPLAIN (COSTS OFF)
SELECT  piece, NULL AS region, sum(quantite)
  FROM  stock
  GROUP BY piece
UNION ALL
SELECT NULL AS piece, region, sum(quantite)
  FROM STOCK
  GROUP BY region;
```

```
              QUERY PLAN
---------------------------------------
 Append
   ->  HashAggregate
         Group Key: stock.piece
         ->  Seq Scan on stock
   ->  HashAggregate
         Group Key: stock_1.region
         ->  Seq Scan on stock stock_1
```

6.6.4 GROUPING SETS : exemple

```
SELECT piece,region,sum(quantite)
FROM   stock
GROUP BY GROUPING SETS (piece,region);
```

```
 piece  | region | sum
--------+--------+-----
 vis    |        | 160
 ecrous |        |  90
 clous  |        |  70
        | ouest  |  50
        | nord   |  60
        | est    | 120
        | sud    |  90
```

Les `GROUPING SETS` servent à définir différents champs indépendants de regrouper les données. Dans ce cas précis, l'intérêt est d'abord d'éviter de dupliquer le contenu de la requête.

Le plan d'exécution montre aussi que la table n'est parcourue qu'une seule fois pour calculer les deux agrégats en même temps, il y a donc aussi un intérêt en performances.

```
EXPLAIN (COSTS OFF)
SELECT piece,region,sum(quantite)
FROM   stock
GROUP BY GROUPING SETS (piece,region);
```

```
       QUERY PLAN
-------------------------
 HashAggregate
   Hash Key: piece
   Hash Key: region
   ->  Seq Scan on stock
```

6.6.5 ROLLUP

- `ROLLUP`
- 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

Il s'agit à présent d'obtenir des agrégats par type de pièce, puis des détails pour chaque pièce et chaque région, et un total final. `ROLLUP` va permettre cela.

6.6.7 ROLLUP : exemple et résultat

```
SELECT piece, region, sum(quantite)
FROM   stock
GROUP BY ROLLUP (piece,region) ;

 piece  | region | sum
--------+--------+-----
        |        | 320
 ecrous | ouest  |   0
 clous  | nord   |   0
 vis    | nord   |  60
 clous  | est    |  70
 vis    | sud    |  50
 ecrous | est    |  50
 ecrous | sud    |  40
 vis    | ouest  |  50
 vis    |        | 160
 ecrous |        |  90
 clous  |        |  70
```

L'ordre des lignes est comme d'habitude non défini. On pourra rajouter un `ORDER BY`. Proposer une présentation lisible pour un humain est laissé en général à l'outil de restitution.

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));
```

Le plan d'exécution est le même, `ROLLUP` est donc d'abord une facilité syntaxique intéressante.

Exemple :

Cet exemple utilise la base **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin  /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

Cet exemple calcule des agrégats par type de client ou de montant. Dans la clause `ORDER BY` finale , on prévoit que les totaux précéderont les détails.

```
SELECT type_client, code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM magasin.commandes c
  JOIN magasin.lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN magasin.clients cl
    ON (c.client_id = cl.client_id)
  JOIN magasin.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)
ORDER BY type_client NULLS FIRST,
         code_pays NULLS FIRST;
```

Elle produit le résultat suivant :

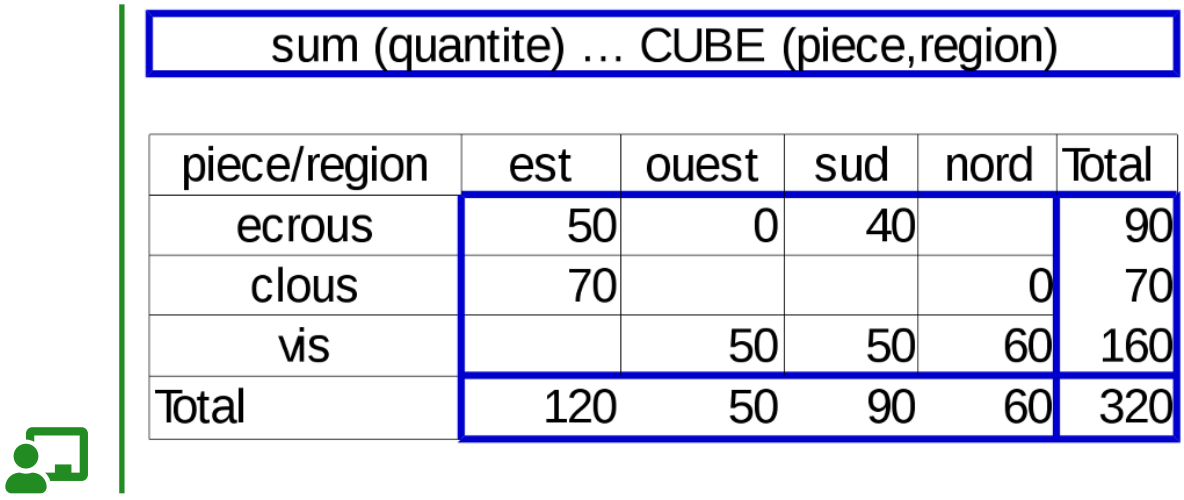
```
 type_client | code_pays |    montant
-------------+-----------+---------------
             |           | 5217862160.65
 A           |           |  111557177.00
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 E           |           |  414152232.57
 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
 P           |           | 4692152751.08
 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
```

6.6.8 CUBE

- `CUBE`
- 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

`CUBE` permet de réaliser des regroupements sur l'ensemble des combinaisons possibles des clauses de regroupement indiquées, avec les totaux intermédiaires et le total final. Pour de plus amples détails, se référer à l'article Wikipédia sur le cube OLAP[2].

[2] https://en.wikipedia.org/wiki/OLAP_cube

6.6.10 CUBE : Syntaxe

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY CUBE (piece,region);

 piece  | region | sum
--------+--------+-----
        |        | 320
 ecrous | ouest  |   0
 clous  | nord   |   0
 vis    | nord   |  60
 clous  | est    |  70
 vis    | sud    |  50
 ecrous | est    |  50
 ecrous | sud    |  40
 vis    | ouest  |  50
 vis    |        | 160
 ecrous |        |  90
 clous  |        |  70
        | ouest  |  50
        | nord   |  60
        | est    | 120
        | sud    |  90
```

On a donc bien les regroupements par type de pièce et par région, ceux par type de pièces, pas région, et un total final.

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)
  );
```

En reprenant la requête de l'exemple précédent dans la base **magasin** :

```
SELECT type_client, code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM magasin.commandes c
  JOIN magasin.lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN magasin.clients cl
    ON (c.client_id = cl.client_id)
  JOIN magasin.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)
```

```
ORDER BY type_client NULLS FIRST,
         code_pays NULLS FIRST;
```

Par rapport au résultat précédent, on obtient en plus les totaux par pays :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
             |           | 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
 A           |           |  111557177.00
 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
 E           |           |  414152232.57
 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
 P           |           | 4692152751.08
 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
```

6.6.11 GROUPING SETS, ROLLUP ou CUBE ?

— `CUBE` peut le plus
— mais forcément plus coûteux

On pourrait imaginer qu'il suffit de toujours faire un `CUBE` en filtrant certaines lignes sur des valeurs `NULL`, et que les syntaxes `ROLLUP` ou `GROUPING SETS` n'ont pas vraiment d'intérêt.

Cependant, `CUBE` calcule plus de données, et il peut être notablement plus lent, surtout quand les volumétries sont non triviales. `ROLLUP` et `GROUPING SETS` permettent de préciser exactement les calculs dont on a besoin.

6.6.12 Filtrer les lignes d'un certain regroupement

Pour que l'application repère les regroupements :

```
SELECT GROUPING(type_client,code_pays)::bit(2),
       GROUPING(type_client)::boolean AS g_type_cli,
       GROUPING(code_pays)::boolean   AS g_code_pays,
       type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
       …
```

Pour que l'application distingue rigoureusement les lignes appartenant à un certain niveau de regroupement, on peut utiliser la fonction `GROUPING`. En effet la colonne de regroupement peut posséder des valeurs `NULL` légitimes, il est alors difficile de les distinguer. 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 magasin.commandes c
  JOIN magasin.lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN magasin.clients cl
    ON (c.client_id = cl.client_id)
  JOIN magasin.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);
```

Dans son résultat, la première ligne indique le total. C'est la seule où les booléens `g_type_cli` et `g_code_pays` sont à `true`.

```
 grouping | g_type_cli | g_code_pays | type_client | code_pays |    montant
----------+------------+-------------+-------------+-----------+---------------
 11       | t          | t           |             |           | 5217862160.65
 00       | f          | f           | P           | DZ        |  302501132.54
 00       | f          | f           | P           | FR        | 2341977444.49
 00       | f          | f           | P           | RU        |  287605812.99
```

```
 00       | f           | f           | E           | RU       |   25397116.39
 00       | f           | f           | A           | DE       |    6642061.57
 00       | f           | f           | A           | FR       |   55261295.52
 00       | f           | f           | P           | CN       |  287795272.87
 00       | f           | f           | A           | IN       |    7224008.95
 00       | f           | f           | E           | IN       |   26788642.27
 00       | f           | f           | A           | DZ       |    6404425.16
 00       | f           | f           | P           | IN       |  295256262.73
 00       | f           | f           | E           | CN       |   25537539.68
 00       | f           | f           | A           | CA       |    6273168.32
 00       | f           | f           | E           | PE       |   24541974.54
 00       | f           | f           | E           | FR       |  209402443.24
 00       | f           | f           | P           | DE       |  287337725.21
 00       | f           | f           | A           | US       |    7700691.07
 00       | f           | f           | A           | PE       |    7356239.93
 00       | f           | f           | A           | CN       |    7928641.50
 00       | f           | f           | E           | US       |   23696294.79
 00       | f           | f           | P           | PE       |  300278960.24
 00       | f           | f           | E           | DE       |   25508815.68
 00       | f           | f           | E           | CA       |   28457655.81
 00       | f           | f           | P           | CA       |  292975985.52
 00       | f           | f           | E           | DZ       |   24821750.17
 00       | f           | f           | A           | RU       |    6766644.98
 00       | f           | f           | P           | US       |  296424154.49
 01       | f           | t           | P           |          | 4692152751.08
 01       | f           | t           | E           |          |  414152232.57
 01       | f           | t           | A           |          |  111557177.00
 10       | t           | f           |             | RU       |  319769574.36
 10       | t           | f           |             | CA       |  327706809.65
 10       | t           | f           |             | IN       |  329268913.95
 10       | t           | f           |             | CN       |  321261454.05
 10       | t           | f           |             | PE       |  332177174.71
 10       | t           | f           |             | US       |  327821140.35
 10       | t           | f           |             | DZ       |  333727307.87
 10       | t           | f           |             | FR       | 2606641183.25
 10       | t           | f           |             | DE       |  319488602.46
(40 rows)
```

La présentation finale est laissée à la charge de l'application. Elle sera à même de gérer la présentation des résultats en fonction des valeurs des champs `grouping`, `g_type_client` ou `g_code_pays`.

Le résultat de `GROUPING` peut servir aussi de tri (voir plus bas).

6.6.13 Affichage des tableaux croisés

— PostgreSQL ne renvoie que des lignes
— Dans `psql` : `\crosstabview`
— Extension : `tablefunc`, fonction `crosstab (text,text)`

PostgreSQL ne renvoie que des lignes au nom de colonne figé. Un tableau croisé, dont les noms de

colonnes dépendent des données, va à l'encontre de cette logique. Convertir un jeu de lignes obtenu avec `CUBE`, par exemple, en tableau croisé est à la charge de l'application (tableur, outil de Business Intelligence, etc…)

Ponctuellement, deux outils intégrés à PostgreSQL peuvent dépanner.

psql et crosstabview :

Cette fonctionnalité existe uniquement dans le client `psql`. Le maniement est simple :

```
-- Paramétrage pour l'affichage dans psql
\pset null TOTAL
\pset linestyle unicode

SELECT piece AS "Pièces", region AS "Région", sum(quantite)
FROM stock
GROUP BY CUBE (piece,region)
ORDER BY GROUPING (piece,region)
 \crosstabview
```

Pièces	ouest	nord	est	sud	TOTAL
vis	50	60		50	160
ecrous	0		50	40	90
clous		0	70		70
TOTAL	50	60	120	90	320

Ou pour inverser :

```
\crosstabview "Région" "Pièces"
```

Région	vis	ecrous	clous	TOTAL
ouest	50	0		50
nord	60		0	60
est		50	70	120
sud	50	40		90
TOTAL	160	90	70	320

Remarquer que modifier l'affichage du `NULL` en lui substituant `TOTAL` ne concerne pas les colonnes vides en milieu de tableau. L'`ORDER BY GROUPING(…)` garantit que les totaux seront à la fin du tableau.

tablefunc :

Il existe une extension fournie avec PostgreSQL nommée `tablefunc` qui fournit des fonctions dédiées aux tableaux croisés. Leur maniement n'est pas évident. Il est conseillé d'utiliser la fonction `crosstab (text,text)` qui tient bien compte des valeurs absentes. Cette fonction demande que la requête à formater lui soit fournie comme chaîne de caractère.

```
CREATE EXTENSION IF NOT EXISTS tablefunc ;

SELECT *
FROM crosstab(
```

```
  $$SELECT coalesce (piece, 'Total'),
           coalesce (region, 'Total'),
           sum(quantite)
    FROM stock
    GROUP BY CUBE (piece,region)
    ORDER BY GROUPING (piece,region)
    $$,
  'SELECT DISTINCT region FROM stock ORDER BY region')
AS ct(piece text, "Est" text, "Nord" text, "Ouest" text, "Sud" text);
```

```
 piece  | Est | Nord | Ouest | Sud
--------+-----+------+-------+-----
 vis    |     |      | 50    |
 ecrous |     |      | 0     |
 clous  |     | 0    |       |
 vis    |     | 60   |       |
 clous  | 70  |      |       |
 vis    |     |      |       | 50
 ecrous | 50  |      |       | 40
 vis    |     |      |       |
 clous  |     |      |       |
 Total  | 120 | 60   | 50    | 90
```

Attention à la cohérence entre la deuxième requête et les titres sur la dernière ligne, ce n'est pas dynamique !

Cet exemple utilise aussi `$$` à la place des guillemets droits pour délimiter une chaîne, car une requête contient souvent elle-même des chaînes.

— Documentation : tablefunc[3]

Pivot dynamique :

Ce billet de Daniel Vérité de 2018[4] décrit comment obtenir un tableau croisé réellement dynamique. Cela réclame toutefois d'écrire un peu de code.

[3] https://docs.postgresql.fr/current/tablefunc.html
[4] https://blog-postgresql.verite.pro/2018/06/02/crosstab-pivot.html

6.7 CONCLUSION

— Les fonctions fenêtrées existent depuis SQL :2003
— Ne vous limitez pas au SQL du XX^{e} siècle

6.8 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s70_solutions.

La table `brno2015` peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_brno2015 -o /tmp/brno2015.dump
createdb  brno2015
pg_restore -O -d brno2015 /tmp/brno2015.dump
# une erreur sur l'existence du schéma public est normale
```

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

> Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?

> Déterminer quel est le pilote le plus régulier (écart-type).

Window Functions

> Afficher la place sur le podium pour chaque coureur.

> À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.

> 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 chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?

> Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.

> 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 s'appuie sur les tables présentes dans le schéma `magasin`.

> En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.

> Ajouter également le montant total des commandes depuis le début de l'activité.

> Ajouter également le montant total des commandes par pays.

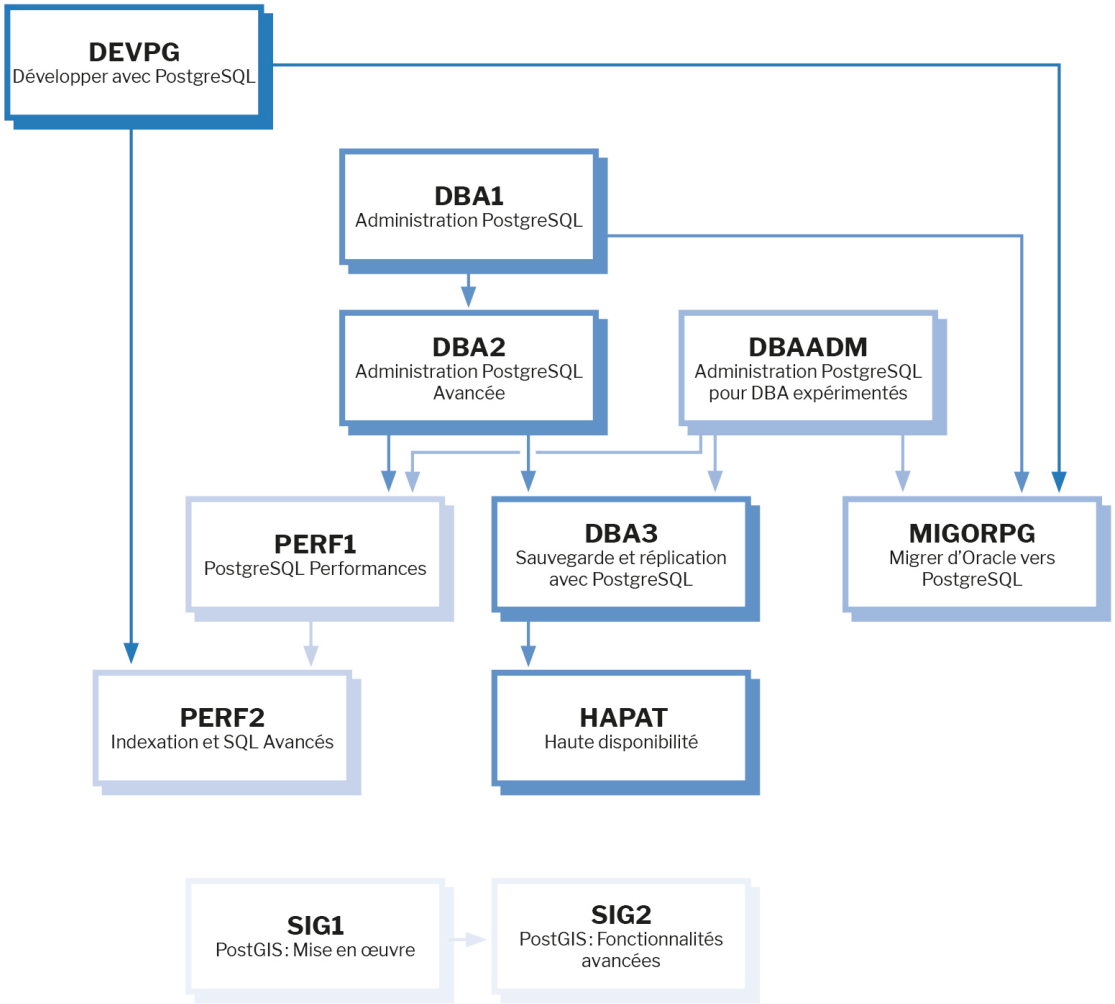
> À partir de la requête précédente, ajouter une colonne par critère de regroupement, de type booléen, qui est positionnée à `true` lorsque le regroupement est réalisé sur l'ensemble des valeurs de la colonne.

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
- DEV1 : Introduction à SQL
  https://dali.bo/dev1
- 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.