Module S3

Plus loin avec SQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Plus loin avec SQL

Module S3

TITRE : Plus loin avec SQL
SOUS-TITRE : Module S3

REVISION: 22.09
DATE: 02 septembre 2022

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

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

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

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

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

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

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

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

Table des Matières

1 PLUS LOIN AVEC SQL

1.1 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.

1.1.1 MENU

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

1.1.2 MENU (SUITE)

- Fonctions de base
- Vues
- Requêtes préparées

1.1.3 OBJECTIFS

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

1.2 VALEUR NULL

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

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

1.2.1 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).

1.2.2 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)
```

1.2.3 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 :

Plus loin avec SQL

```
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.

1.2.4 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[2] .

Exemples

Comparaison directe avec `NULL`, qui est invalide :

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

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

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

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)
```

1.2.5 NULL ET LES AGRÉGATS

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

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

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

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

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

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

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

1.2.6 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
```

1.3 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.

1.3.1 REGROUPEMENT DE DONNÉES

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

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

1.3.2 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.

1.3.3 AGRÉGATS SIMPLES

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

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

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

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

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

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

Exemples :

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

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

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

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

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

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

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

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

1.3.4 CALCULS D'AGRÉGATS

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

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

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

La documentation de PostgreSQL permet d'obtenir la liste des fonctions d'agrégats disponibles[4] .

Exemples

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

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

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

Calcul de moyenne avec des valeurs NULL :

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

INSERT INTO test VALUES (10,10) ;
INSERT INTO test VALUES (20,20) ;
INSERT INTO test VALUES (30,30) ;
INSERT INTO test VALUES (null,0) ;

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

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

Plus loin avec SQL

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

1.3.5 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
```

1.3.6 CLAUSE HAVING

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

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

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

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

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

Exemples

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

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

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

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

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

1.4 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

1.4.1 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.

1.4.2 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
...
```

1.4.3 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') ;
```

1.4.4 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.

1.4.5 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) ;
```

1.4.6 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`.

1.4.7 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) ;
```

1.4.8 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) ;
```

1.4.9 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 ;
```

1.4.10 CLAUSE EXISTS

`EXISTS (sous-requete)`

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

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

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

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

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

1.5 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.

1.5.1 CONDITIONS DE JOINTURE DANS `JOIN` OU DANS `WHERE` ?

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

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

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

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

conditions de jointure et les tables auxquelles elles s'appliquent le cas échéant.

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

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

et :

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

1.5.2 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
```

1.5.3 JOINTURE INTERNE

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

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

La condition de jointure est généralement une égalité, ce qui permet d'associer entre elles

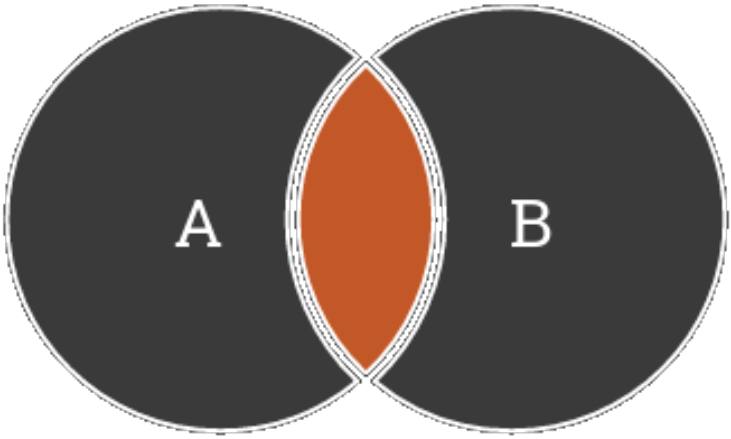

Figure 1: Schéma de jointure interne

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.

1.5.4 SYNTAXE D'UNE JOINTURE INTERNE

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

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

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

La clause `NATURAL` permet de réaliser la jointure entre deux tables en utilisant les colonnes qui portent le même nom sur les deux tables comme condition de jointure. `NATURAL JOIN` est fortement déconseillé car elle peut facilement entraîner des comportements inattendus.

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

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

1.5.5 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`

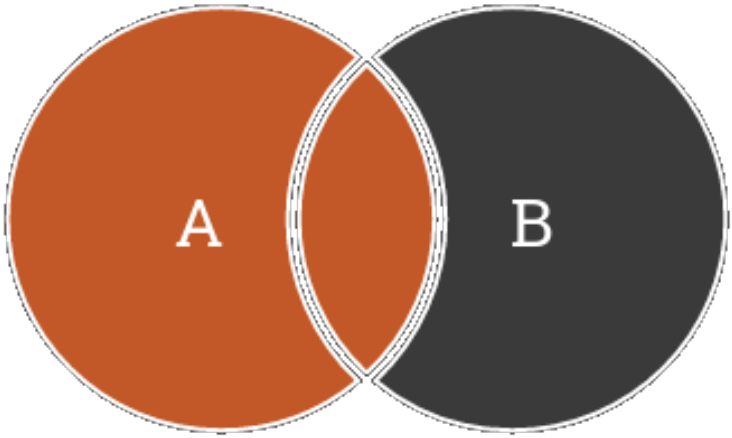

Figure 2: Schéma de jointure externe gauche

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

1.5.6 JOINTURE EXTERNE - 2

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

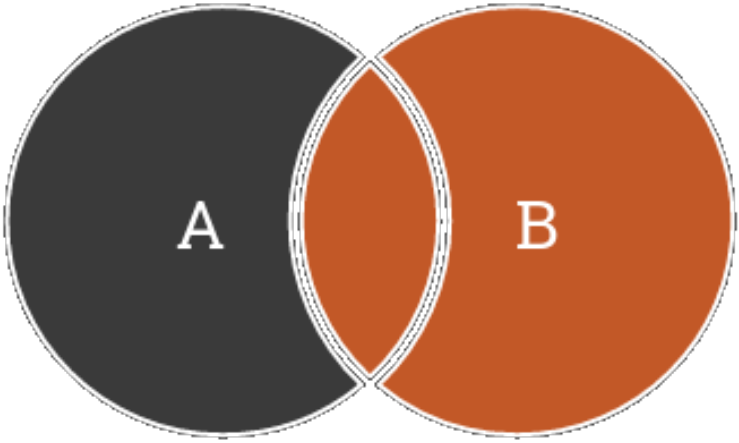

Figure 3: Schéma de jointure externe droite

1.5.7 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`

1.5.8 SYNTAXE D'UNE JOINTURE EXTERNE À GAUCHE

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

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

Les voici en exemple :

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

1.5.9 SYNTAXE D'UNE JOINTURE EXTERNE À DROITE

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

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

1.5.10 SYNTAXE D'UNE JOINTURE EXTERNE COMPLÈTE

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

1.5.11 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.

1.6 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;
```

1.6.1 CASE SIMPLE

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

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

```
CASE nom_region
  WHEN 'Afrique'  THEN 1
  WHEN 'Amérique' THEN 2
  WHEN 'Asie'     THEN 3
```

```
  WHEN 'Europe'   THEN 4
  ELSE 0
END
```

1.6.2 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
```

1.6.3 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
```

1.7 OPÉRATEURS ENSEMBLISTES

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

1.7.1 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 ;
```

1.7.2 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
```

1.7.3 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
```

1.8 FONCTIONS DE BASE

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

PostgreSQL propose un nombre conséquent de fonctions permettant de manipuler les différents types de données disponibles. Les étudier de façon exhaustive n'est pas l'objet de ce module. Néanmoins, le manuel de PostgreSQL établit une liste complète des fonctions disponibles dans le SGBD[5] .

---

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

1.8.1 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.

1.8.2 OPÉRATIONS SIMPLES SUR LES CHAÎNES

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

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

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

-----------------
 Bonjour, Monde!
```

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

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

---------
 Texte 1
```

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

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

-------------
           6
```

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

```
SELECT lower('Bonjour, Monde!') ;

-----------------
 bonjour, monde!

SELECT upper('Bonjour, Monde!') ;

-----------------
 BONJOUR, MONDE!
```

1.8.3 MANIPULATIONS DE CHAÎNES

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

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

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

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

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

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

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

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

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

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

La combinaison des deux est intéressante :

```
SELECT version() ;
```

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

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

1.8.4 MANIPULATION DE TYPES NUMÉRIQUES

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

1.8.5 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[6] .

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
```

---

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

1.8.6 FONCTIONS NUMÉRIQUES COURANTES

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

Ces fonctions sont décrites dans le manuel[7] .

1.8.7 GÉNÉRATION DE DONNÉES

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

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

```
SELECT generate_series(1, 4) ;

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

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

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

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

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

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

Plus loin avec SQL

```
SELECT random() ;

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

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

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

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

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

```
SET SEED = 0.123 ;
```

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

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

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

Ces fonctions sont décrites dans le manuel de PostgreSQL[8] .

1.8.8 MANIPULATION DE DATES

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

---

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

1.8.9 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!
```

1.8.10 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
```

1.8.11 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
```

1.8.12 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
```

1.8.13 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
```

1.8.14 CHAÎNE VERS DATE

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

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

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

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

Ces fonctions sont détaillées dans la section concernant les fonctions de formatage de données du manuel[9] .

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

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

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

SELECT current_timestamp ;

              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
```

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

1.8.15 GÉNÉRATION DE DONNÉES

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

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

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

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

1.9 VUES

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

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

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

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

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

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

1.9.1 CRÉATION D'UNE VUE

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

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

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

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

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

La clause `nom` permet de nommer la vue. La clause `nom_colonne, …`' permet lister explicitement les colonnes retournées par une vue, cette clause est optionnelle mais recom-

mandée pour mieux documenter la vue.

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

Exemples

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

CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data ;

GRANT SELECT ON phone_number TO secretary ;
```

1.9.2 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
```

1.9.3 SÉCURISATION D'UNE VUE

- Sécuriser une vue
  - droits avec `GRANT` et `REVOKE`
- Utiliser les vues comme moyen de filtrer les lignes est dangereux
  - option `security_barrier`

Il est possible d'accorder (ou de révoquer) à un utilisateur les mêmes droits sur une vue que sur une table :

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

Le droit `SELECT` autorise un utilisateur à lire une table. Les droits `INSERT`, `UPDATE` et `DELETE` permettent de contrôler les accès en mise à jour à une vue.

Les droits `TRUNCATE` et `REFERENCES` n'ont pas d'utilité avec une vue. Ils ne sont tout simplement pas supportés car `TRUNCATE` n'agit que sur une table et une clé étrangère ne peut être liée d'une table qu'à une autre table.

Les vues sont parfois utilisées pour filtrer les lignes pouvant être lues par l'utilisateur. Cette protection peut être contournée si l'utilisateur a la possibilité de créer une fonction. L'option `security_barrier` permet d'éviter ce problème.

Exemples

```
postgres=# CREATE TABLE elements (id serial, contenu text, prive boolean) ;
CREATE TABLE
postgres=# INSERT INTO elements (contenu, prive)
VALUES ('a', false), ('b', false), ('c super prive', true),
       ('d', false), ('e prive aussi', true) ;
INSERT 0 5

postgres=# SELECT * FROM elements ;

 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
```

La table `elements` contient cinq lignes, dont trois considérés comme privés. Nous allons donc créer une vue ne permettant de voir que les lignes publiques.

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

CREATE VIEW

postgres=# SELECT * FROM elements_public ;

 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t

postgres=# CREATE USER u1;
CREATE ROLE

postgres=# GRANT SELECT ON elements_public TO u1;
GRANT

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

postgres=> SELECT * FROM elements ;
ERROR:  permission denied for relation elements

postgres=> SELECT * FROM elements_public ;

 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

L'utilisateur `u1` n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Mais le contenu peut être trahi par une simple fonction qui trahit le contenu des lignes que l'exécuteur rencontre avant de les filtrer :

```
postgres=> CREATE OR REPLACE FUNCTION abracadabra(integer, text, boolean)
           RETURNS bool AS $$
```

```
          BEGIN
            -- afficher chaque ligne rencontrée
            RAISE NOTICE '% - % - %', $ 1, $ 2, $ 3 ;
            RETURN true ;
          END$$
          LANGUAGE plpgsql
          -- désigner un coût bas pour exécuter cette fonction
          -- avant le filtre dans la vue
          COST 0.000000000000000000001 ;
CREATE FUNCTION

postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive) ;

NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  3 - c super prive - t
NOTICE:  4 - d - f
NOTICE:  5 - e prive aussi - t
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

Que s'est-il passé ? pour comprendre, il suffit de regarder l'`EXPLAIN` de cette requête :

```
postgres=> EXPLAIN SELECT * FROM elements_public
WHERE abracadabra(id, contenu, prive) ;

                            QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on elements  (cost=0.00..28.15 rows=202 width=37)
   Filter: (abracadabra(id, contenu, prive) AND
           CASE WHEN ("current_user"() = 'u1'::name)
                THEN (NOT prive) ELSE true END)
```

La requête contient deux filtres : celui dans la vue, celui dans la fonction `abracadra`. On a déclaré un coût si faible pour cette dernière que PostgreSQL, pour optimiser,l'exécute avant le filtre de la vue. Du coup, la fonction voit toutes les lignes de la table et peut trahir leur contenu.

Seul moyen d'échapper à cette optimisation du planificateur, utiliser l'option `security_barrier` :

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

postgres=# CREATE OR REPLACE VIEW elements_public
           WITH (security_barrier)
```

```
          AS
          SELECT * FROM elements
          WHERE CASE WHEN current_user='postgres' THEN true ELSE NOT prive END ;

CREATE VIEW

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

postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);

NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  4 - d - f
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f

postgres=> EXPLAIN SELECT * FROM elements_public WHERE
abracadabra(id, contenu, prive) ;

                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..34.20 rows=202 width=37)
   Filter: abracadabra(elements_public.id, elements_public.contenu,
           elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.15 rows=605 width=37)
         Filter: CASE WHEN ("current_user"() = 'u1'::name)
                 THEN (NOT prive) ELSE true END
```

Il peut y avoir un impact en performance : le filtre de la vue s'applique d'abord, donc force peut forcer l'optimiseur à s'écarter du chemin optimal.

Voir aussi cet article de blog[10] .

---

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

1.9.4 MISE À JOUR DES VUES

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

Depuis PostgreSQL 9.3, le moteur permet de mettre à jour des vues simples, ou plus exactement de mettre à jour les tables sous-jacentes au travers de la vue. Les critères déterminant si une vue peut être mise à jour ou non sont assez simples à résumer : la vue doit reprendre la définition de la table avec éventuellement une clause `WHERE` pour restreindre les résultats.

La clause `WITH CHECK OPTION` (à partie de PostgreSQL 9.4) empêche l'utilisateur d'insérer des données qui ne satisfont pas les critères de filtrage de la vue. Sans elle, il est possible d'insérer un numéro de téléphone privé alors que la vue ne permet pas d'afficher les numéros privés. Cette option doit être demandée explicitement.

Pour gérer les cas plus complexes, PostgreSQL permet de créer des triggers `INSTEAD OF` sur des vues. Une alternative est d'utiliser le système de règles (`RULES`) mais cette pratique est peu recommandée en raison de la difficulté de débogage et de maintenance.

Un trigger `INSTEAD OF` permet de déclencher une fonction utilisateur lorsqu'une opération de mise à jour est déclenchée sur une vue. Le code de la fonction sera exécuté en lieu et place de la mise à jour.

Exemples

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

CREATE VIEW maj_phone_number (person, phone, private) AS
    SELECT person, phone, private
      FROM phone_data
     WHERE private = false ;

-- On peut insérer des données car les colonnes de la vue correspondent aux
-- colonnes de la table
INSERT INTO maj_phone_number VALUES ('Titi', '0123456789', false) ;

-- On parvient même à insérer des données qui ne pourront pas être affichées
-- par la vue. Ça peut être gênant.
INSERT INTO maj_phone_number VALUES ('Loulou', '0123456789', true) ;

SELECT * FROM maj_phone_number ;

 person |   phone    | private
--------+------------+---------
```

```
 Titi   | 0123456789 | f

-- L'option WITH CHECK OPTION rajoute une sécurité
CREATE OR REPLACE VIEW maj_phone_number (person, phone, private) AS
    SELECT person, phone, private
    FROM phone_data
    WHERE private = false
    WITH CHECK OPTION ;

INSERT INTO maj_phone_number VALUES ('Lili', '9993456789', true);
ERROR:  new row violates check option for view "maj_phone_number"
DETAIL : Failing row contains (Lili, 9993456789, t).

-- Cas d'une vue avec un champ calculé
CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data ;

-- On ne peut pas insérer de données car les colonnes de la vue ne
-- correspondent pas à celles de la table
INSERT INTO phone_number VALUES ('Fifi', '0123456789');
ERROR:  cannot insert into column "phone" of view "phone_number"
DETAIL:  View columns that are not columns of their base relation are not updatable.

CREATE OR REPLACE FUNCTION phone_number_insert_row()
  RETURNS TRIGGER
  LANGUAGE plpgsql
AS $function$
BEGIN
  INSERT INTO phone_data (person, phone, private)
    VALUES (NEW.person, NEW.phone, false);
  RETURN NEW ;
END ;
$function$;

CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON phone_number
    FOR EACH ROW
    EXECUTE PROCEDURE phone_number_insert_row();

-- Avec le trigger, c'est maintenant possible.
INSERT INTO phone_number VALUES ('Rominet', '0123456788');

SELECT * FROM phone_number ;

 person  |   phone
---------+------------
 Titi    | 0123456789
 Loulou  |
```

```
Rominet | 0123456788
```

1.9.5 MAUVAISES UTILISATIONS DES VUES

- Prolifération des vues
  - créer une vue doit se justifier
  - ne pas créer une vue par table
- Vues trop complexes utilisées comme interface
- Vues empilées

La création d'une vue doit être pensée préalablement et doit se justifier du point de vue de l'application ou d'une règle métier. Toute vue créée doit être documentée, au moins en plaçant un commentaire sur la vue pour la documenter.

Bien qu'une vue n'ait pas de représentation physique, elle occupe malgré tout un peu d'espace disque. En effet, le catalogue système comporte une entrée pour chaque vue créée, autant d'entrées qu'il y a de colonnes à la vue, etc. Trop de vues entraîne donc malgré tout l'augmentation de la taille du catalogue système, donc une empreinte mémoire plus importante car ce catalogue reste en général systématiquement présent en cache.

Un problème fréquent est celui de vues complexes calculant beaucoup de choses pour le confort de l'usager... au prix des performances quand l'utilisateur n'a pas besoin de ces informations. L'optimiseur ne peut pas forcément tout élaguer.

Pour cette raison, et pour des raisons de facilité de maintenance, il faut aussi éviter d'empiler les vues.

1.10 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.

1.10.1 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.

1.11 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.

1.11.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.12 TRAVAUX PRATIQUES 1 (ÉNONCÉS)

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

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

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

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

Le schéma suivant montre les différentes tables de la base :

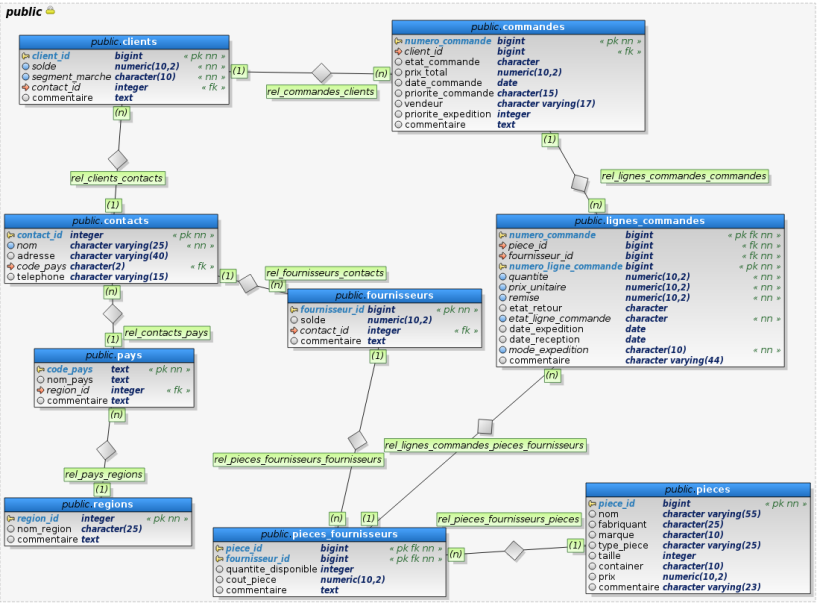

Figure 4: Schéma base tpc

1. Affichez, par pays, le nombre de fournisseurs.

Sortie attendue :

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

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

Sortie attendue :

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

3. Affichez le nombre de commandes trié selon le nombre de lignes de commandes au sein de chaque commande.

Sortie attendue :

```
 num | count
-----+-------
   1 | 13733
   2 | 27816
   3 | 27750
   4 | 27967
   5 | 27687
   6 | 27876
   7 | 13895
```

4. Pour les 30 premières commandes (selon la date de commande), affichez le prix total de la commande, en appliquant la remise accordée sur chaque article commandé. La sortie sera triée de la commande la plus chère à la commande la moins chère.

Sortie attendue :

```
 numero_commande | prix_total
-----------------+------------
               3 |  259600.00
              40 |  258959.00
               6 |  249072.00
              69 |  211330.00
              70 |  202101.00
               4 |  196132.00
 (...)
```

5. Affichez, par année, le total des ventes. La date de commande fait foi. La sortie sera triée par année.

Sortie attendue :

```
 annee |  total_vente
-------+---------------
  2005 | 3627568010.00
  2006 | 3630975501.00
  2007 | 3627112891.00
(...)
```

6. Pour toutes les commandes, calculez le temps moyen de livraison, depuis la date d'expédition. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

Sortie attendue :

```
temps_moyen_livraison
----------------------
 8 jour(s)
```

7. Pour les 30 commandes les plus récentes (selon la date de commande), calculez le temps moyen de livraison de chaque commande, depuis la date de commande. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

Sortie attendue :

```
temps_moyen_livraison
----------------------
 38 jour(s)
```

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

Sortie attendue :

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

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

Sortie attendue :

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

10. Un bug applicatif est soupçonné, déterminez s'il existe des commandes dont la date de commande est postérieure à la date d'expédition des articles.

Sortie attendue :

```
count
-------
     2
```

11. Écrivez une requête qui corrige les données erronés en positionnant la date de commande à la date d'expédition la plus ancienne des marchandises. Vérifiez qu'elle soit correcte. Cette requête permet de corriger des calculs de statistiques sur les délais de livraison.

12. Écrivez une requête qui calcule le délai total maximal de livraison de la totalité d'une commande donnée, depuis la date de la commande.

Sortie attendue pour la commande n°1 :

```
delai_max
-----------
       102
```

13. Écrivez une requête pour déterminer les 10 commandes dont le délai de livraison, entre la date de commande et la date de réception, est le plus important, pour l'année 2011 uniquement.

Sortie attendue :

```
 numero_commande | delai
-----------------+-------
          413510 |   146
          123587 |   143
          224453 |   143
(...)
```

14. Un autre bug applicatif est détecté. Certaines commandes n'ont pas de lignes de commandes. Écrivez une requête pour les retrouver.

```
-[ RECORD 1 ]------------------------
numero_commande     | 91495
client_id           | 93528
etat_commande       | P
prix_total          |
date_commande       | 2007-07-07
priorite_commande   | 5-NOT SPECIFIED
vendeur             | Vendeur 000006761
priorite_expedition | 0
commentaire         | xxxxxxxxxxxxx
```

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

16. Écrivez une requête pour déterminer les 20 pièces qui ont eu le plus gros volume de commande.

Sortie attendue :

```
                    nom                    |  sum
-------------------------------------------+--------
 lemon black goldenrod seashell plum       | 461.00
 brown lavender dim white indian           | 408.00
 burlywood white chiffon blanched lemon    | 398.00
(...)
```

17. Affichez les fournisseurs des 20 pièces qui ont été le plus commandées sur l'année 2011.

Sortie attendue :

```
     nom      | piece_id
--------------+----------
 Supplier4395 |   191875
 Supplier4397 |   191875
 Supplier6916 |   191875
 Supplier9434 |   191875
 Supplier4164 |    11662
 Supplier6665 |    11662
(...)
```

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

Sortie attendue :

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

19. Affichez pour les commandes passées en 2011, la liste des continents (`régions`) et la marge brute d'exploitation réalisée par continents, triés dans l'ordre décroissant.

Sortie attendue :

```
        nom_region         |   benefice
---------------------------+---------------
 Moyen-Orient              | 2008595508.00
(...)
```

20. Affichez le nom, le numéro de téléphone et le pays des fournisseurs qui ont un commentaire contenant le mot clé `Complaints` :

Sortie attendue :

```
 nom_fournisseur |    telephone    |           nom_pays
-----------------+-----------------+-------------------------------
 Supplier3873    | 10-741-199-8614 | IRAN, RÉPUBLIQUE ISLAMIQUE D'
(...)
```

21. Déterminez le top 10 des fournisseurs ayant eu le plus long délai de livraison, entre la date de commande et la date de réception, pour l'année 2011 uniquement.

Sortie attendue :

```
 fournisseur_id | nom_fournisseur | delai
----------------+-----------------+-------
           9414 | Supplier9414    |   146
(...)
```

1.13 TRAVAUX PRATIQUES 2 (ÉNONCÉS)

Ce TP s'exécute sur la base **tpc** :

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

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

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

4. Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`. Mettre à jour la colonne `prix_total` des

commandes avec les informations des lignes de la table `lignes_commandes`.

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

6. Supprimer les commandes de mai 2010.

7. Ré-exécuter la requête trouvée au point 5.

8. Qu'observez-vous ?

9. Corriger le problème rencontré.

10. Créer une vue calculant le prix total de chaque commande.

11. Réécrire la requête du point 5 pour utiliser la vue créée au point 10.

1.14 TRAVAUX PRATIQUES 1 (SOLUTIONS)

> 1. Affichez, par pays, le nombre de fournisseurs.

```
SELECT p.nom_pays, count(*)
FROM fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
GROUP BY p.nom_pays ;
```

> 2. Affichez, par continent (`regions`), le nombre de fournisseurs.

```
SELECT r.nom_region, count(*)
FROM fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
   JOIN regions r ON p.region_id = r.region_id
GROUP BY r.nom_region ;
```

> 3. Affichez le nombre de commandes trié selon le nombre de lignes de commandes au sein de chaque commande.

```
SELECT
   nombre_lignes_commandes,
   count(*) AS nombre_total_commandes
FROM (
   /* cette sous-requête permet de compter le nombre de lignes de commande de
      chaque commande, et remonte cette information à la requête principale */
   SELECT count(numero_ligne_commande) AS nombre_lignes_commandes
   FROM lignes_commandes
   GROUP BY numero_commande
   ) comm_agg
/* la requête principale aggrège et trie les données sur ce nombre de lignes
   de commandes pour compter le nombre de commandes distinctes ayant le même
   nombre de lignes de commandes */
GROUP BY nombre_lignes_commandes
ORDER BY nombre_lignes_commandes DESC ;
```

> 4. Pour les 30 premières commandes (selon la date de commande), affichez le prix total de la commande, en appliquant la remise accordée sur chaque article commandé. La

> sortie sera triée de la commande la plus chère à la commande la moins chère.

```
SELECT c.numero_commande, sum(quantite * prix_unitaire - remise) prix_total
FROM (
   SELECT numero_commande, date_commande
   FROM commandes
   ORDER BY date_commande
   LIMIT 30
   ) c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
GROUP BY c.numero_commande
ORDER BY sum(quantite * prix_unitaire - remise) DESC ;
```

> 5. Affichez, par année, le total des ventes. La date de commande fait foi. La sortie sera triée par année.

```
SELECT
   extract ('year' FROM date_commande),
   sum(quantite * prix - remise) AS prix_total
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
   JOIN pieces p ON lc.piece_id = p.piece_id
GROUP BY extract ('year' FROM date_commande)
ORDER BY extract ('year' FROM date_commande) ;
```

> 6. Pour toutes les commandes, calculez le temps moyen de livraison, depuis la date d'expédition. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

```
SELECT ceil(avg(date_reception - date_expedition))::text || ' jour(s)'
FROM lignes_commandes lc ;
```

> 7. Pour les 30 commandes les plus récentes (selon la date de commande), calculez le temps moyen de livraison de chaque commande, depuis la date de commande. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

```
SELECT count(*), ceil(avg(date_reception - date_commande))::text || ' jour(s)'
FROM (
   SELECT numero_commande, date_commande
```

```
   FROM commandes
   ORDER BY date_commande DESC
   LIMIT 30
   ) c
JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande ;
```

Note : la colonne `date_commande` de la table `commandes` n'a pas de contrainte `NOT NULL`, il est donc possible d'avoir des commandes sans date de commande renseignée. Dans ce cas, ces commandes vont remonter par défaut en haut de la liste, puisque la clause `ORDER BY` renvoie les `NULL` après les valeurs les plus grandes, et que l'on inverse le tri. Pour éviter que ces commandes ne faussent les résultats, il faut donc les exclure de la sous-requête, de la façon suivante :

```
   SELECT numero_commande, date_commande
   FROM commandes
   WHERE date_commande IS NOT NULL
   ORDER BY date_commande DESC
   LIMIT 30
```

> 8. Déterminez le taux de retour des marchandises (l'état à `R` indiquant qu'une marchandise est retournée).

```
SELECT
   round(
      sum(
         CASE etat_retour
            WHEN 'R' THEN 1.0
            ELSE 0.0
         END
      ) / count(*)::numeric * 100,
      2
   )::text || ' %' AS taux_retour
FROM lignes_commandes ;
```

La clause `FILTER` des fonctions d'agrégation permet d'écrire une telle requête plus facilement :

```
SELECT
   round(
      count(*) FILTER (WHERE etat_retour = 'R') / count(*)::numeric * 100,
      2
   )::text || ' %' AS taux_retour
FROM lignes_commandes ;
```

> 9. Déterminez le mode d'expédition qui est le plus rapide, en moyenne.

```
SELECT mode_expedition, avg(date_reception - date_expedition)
FROM lignes_commandes lc
GROUP BY mode_expedition
ORDER BY avg(date_reception - date_expedition) ASC
LIMIT 1 ;
```

> 10. Un bug applicatif est soupçonné, déterminez s'il existe des commandes dont la date de commande est postérieure à la date d'expédition des articles.

```
SELECT count(*)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
      AND c.date_commande > lc.date_expedition ;
```

> 11. Écrivez une requête qui corrige les données erronés en positionnant la date de commande à la date d'expédition la plus ancienne des marchandises. Vérifiez qu'elle soit correcte. Cette requête permet de corriger des calculs de statistiques sur les délais de livraison.

Afin de se protéger d'une erreur de manipulation, on ouvre une transaction :

```
BEGIN;

UPDATE commandes c_up
SET date_commande = (
   SELECT min(date_expedition)
   FROM commandes c
      JOIN lignes_commandes lc ON lc.numero_commande = c.numero_commande
         AND c.date_commande > lc.date_expedition
   WHERE c.numero_commande = c_up.numero_commande
   )
WHERE EXISTS (
   SELECT 1
   FROM commandes c2
      JOIN lignes_commandes lc ON lc.numero_commande = c2.numero_commande
         AND c2.date_commande > lc.date_expedition
      WHERE c_up.numero_commande  = c2.numero_commande
   GROUP BY 1
   ) ;
```

La requête réalisée précédemment doit à présent retourner 0 :

```
SELECT count(*)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
      AND c.date_commande > lc.date_expedition ;
```

Si c'est le cas, on valide la transaction :

```
COMMIT ;
```

Si ce n'est pas le cas, il doit y avoir une erreur dans la transaction, on l'annule :

```
ROLLBACK ;
```

> 12. Écrivez une requête qui calcule le délai total maximal de livraison de la totalité d'une commande donnée, depuis la date de la commande.

Par exemple pour la commande dont le numéro de commande est le `1` :

```
SELECT max(date_reception - date_commande)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
WHERE c.numero_commande = 1 ;
```

> 13. Écrivez une requête pour déterminer les 10 commandes dont le délai de livraison, entre la date de commande et la date de réception, est le plus important, pour l'année 2011 uniquement.

```
SELECT
   c.numero_commande,
   max(date_reception - date_commande)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
    AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY c.numero_commande
ORDER BY max(date_reception - date_commande) DESC
LIMIT 10 ;
```

> 14. Un autre bug applicatif est détecté. Certaines commandes

> n'ont pas de lignes de commandes. Écrivez une requête pour les retrouver.

Pour réaliser cette requête, il faut effectuer une jointure spéciale, nommée « Anti-jointure ». Il y a plusieurs façons d'écrire ce type de jointure. Les différentes méthodes sont données de la moins efficace à la plus efficace.

La version la moins performante est la suivante, avec `NOT IN` :

```
SELECT c.numero_commande
FROM commandes
WHERE numero_commande NOT IN (
   SELECT numero_commande
   FROM lignes_commandes
   ) ;
```

Il n'y a aucune corrélation entre la requête principale et la sous-requête. PostgreSQL doit donc vérifier pour chaque ligne de `commandes` que `numero_commande` n'est pas présent dans l'ensemble retourné par la sous-requête. Il est préférable d'éviter cette syntaxe.

Autre écriture possible, avec `LEFT JOIN` :

```
SELECT c.numero_commande
FROM commandes c
   LEFT JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
/* c'est le filtre suivant qui permet de ne conserver que les lignes de la
   table commandes qui n'ont PAS de correspondance avec la table
   numero_commandes */
WHERE lc.numero_commande IS NULL ;
```

Enfin, l'écriture généralement préférée, tant pour la lisibilité que pour les performances, avec `NOT EXISTS` :

```
SELECT c.numero_commande
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   ) ;
```

> 15. Écrivez une requête pour supprimer ces commandes. Vérifiez le travail avant de valider.

Afin de se protéger d'une erreur de manipulation, on ouvre une transaction :

```
BEGIN ;
```

La requête permettant de supprimer ces commandes est dérivée de la version `NOT EXISTS` de la requête ayant permis de trouver le problème :

```
DELETE
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   )
-- on peut renvoyer directement les numeros de commande qui ont été supprimés :
-- RETURNING numero_commande
;
```

Pour vérifier que le problème est corrigé :

```
SELECT count(*)
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   ) ;
```

Si la requête ci-dessus remonte `0`, alors la transaction peut être validée :

```
COMMIT ;
```

> 16. Écrivez une requête pour déterminer les 20 pièces qui ont eu le plus gros volume de commande.

```
SELECT p.nom,
   sum(quantite)
FROM pieces p
   JOIN lignes_commandes lc ON p.piece_id = lc.piece_id
GROUP BY p.nom
ORDER BY sum(quantite) DESC
LIMIT 20 ;
```

> 17. Affichez les fournisseurs des 20 pièces qui ont été le plus commandées sur l'année 2011.

```
SELECT co.nom, max_p.piece_id, total_pieces
FROM (
   /* cette sous-requête est sensiblement la même que celle de l'exercice
```

```
    précédent, sauf que l'on remonte cette fois l'id de la piece plutôt
    que son nom pour pouvoir faire la jointure avec pieces_fournisseurs, et
    que l'on ajoute une jointure avec commandes pour pouvoir filtrer sur
    l'année 2011 */
SELECT
    p.piece_id,
    sum(quantite) AS total_pieces
FROM pieces p
    JOIN lignes_commandes lc ON p.piece_id = lc.piece_id
    JOIN commandes c ON c.numero_commande = lc.numero_commande
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
    AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY p.piece_id
ORDER BY sum(quantite) DESC
LIMIT 20
) max_p
/* il faut passer par la table de liens pieces_fournisseurs pour récupérer
    la liste des fournisseurs d'une piece */
JOIN pieces_fournisseurs pf ON max_p.piece_id = pf.piece_id
JOIN fournisseurs f ON f.fournisseur_id = pf.fournisseur_id
-- la jointure avec la table contact permet d'afficher le nom du fournisseur
JOIN contacts co ON f.contact_id = co.contact_id ;
```

> 18. Affichez le pays qui a connu, en nombre, le plus de commandes sur l'année 2011.

```
SELECT nom_pays,
    count(c.numero_commande)
FROM commandes c
    JOIN clients cl ON (c.client_id = cl.client_id)
    JOIN contacts co ON (cl.contact_id = co.contact_id)
    JOIN pays p ON (co.code_pays = p.code_pays)
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
    AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY p.nom_pays
ORDER BY count(c.numero_commande) DESC
LIMIT 1 ;
```

> 19. Affichez pour les commandes passées en 2011, la liste des continents (`régions`) et la marge brute d'exploitation réalisée par continents, triés dans l'ordre décroissant.

```
SELECT
    nom_region,
    round(sum(quantite * prix - remise) - sum(quantite * cout_piece), 2)
```

```
    AS marge_brute
FROM
   commandes c
   JOIN lignes_commandes lc ON lc.numero_commande = c.numero_commande
   /* il faut passer par la table de liens pieces_fournisseurs pour récupérer
      la liste des fournisseurs d'une piece - attention, la condition de
      jointure entre lignes_commandes et pieces_fournisseurs porte sur deux
      colonnes ! */
   JOIN pieces_fournisseurs pf ON lc.piece_id = pf.piece_id
      AND lc.fournisseur_id = pf.fournisseur_id
   JOIN pieces p ON p.piece_id = pf.piece_id
   JOIN fournisseurs f ON f.fournisseur_id = pf.fournisseur_id
   JOIN clients cl ON c.client_id = cl.client_id
   JOIN contacts co ON cl.contact_id = co.contact_id
   JOIN pays pa ON co.code_pays = pa.code_pays
   JOIN regions r ON r.region_id = pa.region_id
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY nom_region
ORDER BY sum(quantite * prix - remise) - sum(quantite * cout_piece) DESC ;
```

> 20. Affichez le nom, le numéro de téléphone et le pays des fournisseurs qui ont un commentaire contenant le mot clé `Complaints` :

```
SELECT
   nom,
   telephone,
   nom_pays
FROM
   fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
WHERE f.commentaire LIKE '%Complaints%' ;
```

> 21. Déterminez le top 10 des fournisseurs ayant eu le plus long délai de livraison, entre la date de commande et la date de réception, pour l'année 2011 uniquement.

```
SELECT
   f.fournisseur_id,
   co.nom,
   max(date_reception - date_commande)
FROM
   lignes_commandes lc
```

```
   JOIN commandes c ON c.numero_commande = lc.numero_commande
   JOIN pieces_fournisseurs pf ON lc.piece_id = pf.piece_id
      AND lc.fournisseur_id = pf.fournisseur_id
   JOIN fournisseurs f ON pf.fournisseur_id = f.fournisseur_id
   JOIN contacts co ON f.contact_id = co.contact_id
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY f.fournisseur_id, co.nom
ORDER BY max(date_reception - date_commande) DESC
LIMIT 10 ;
```

1.15 TRAVAUX PRATIQUES 2 (SOLUTIONS)

> 1. Ajouter une adresse mail à chaque contact avec la concaténation du nom avec le texte « @dalibo.com ».

```
BEGIN ;

UPDATE contacts
SET email = nom||'@dalibo.com' ;

COMMIT ;
```

Note : pour éviter de mettre à jour les contacts ayant déjà une adresse mail, il suffit d'ajouter une clause `WHERE` :

```
UPDATE contacts
SET email = nom||'@dalibo.com'
WHERE email IS NULL ;
```

> 2. Concaténer nom et adresse mail des contacts français sous la forme « nom <mail> ».

```
SELECT nom||' <'||email||'>'
FROM contacts
;
```

> 3. Même demande mais avec le nom en majuscule et l'adresse mail en minuscule.

```
SELECT upper(nom)||' <'||lower(email)||'>'
FROM contacts
;
```

> 4. Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`. Mettre à jour la colonne `prix_total` des commandes avec les informations des lignes de la table `lignes_commandes`.

```
ALTER TABLE commandes ADD COLUMN prix_total numeric(10,2) ;

BEGIN ;

UPDATE commandes c
SET prix_total= (
```

```
   /* cette sous-requête fait une jointure entre lignes_commandes et la table
      commandes à mettre à jour pour calculer le prix par commande */
   SELECT SUM(quantite * prix_unitaire - remise)
   FROM lignes_commandes lc
   WHERE lc.numero_commande=c.numero_commande
)
-- on peut récupérer le détail de la mise à jour directement dans la requête :
-- RETURNING numero_commande, prix_total
;
COMMIT ;
```

Une autre variante de cette requête serait :

```
UPDATE commandes c SET prix_total=prix_calc
FROM (
    SELECT numero_commande, SUM(quantite * prix_unitaire - remise) AS prix_calc
        FROM lignes_commandes
       GROUP BY numero_commande
) as prix_detail
WHERE prix_detail.numero_commande = c.numero_commande ;
```

Bien que cette dernière variante soit moins lisible, elle est bien plus rapide sur un gros volume de données.

> 5. Récupérer le montant total des commandes par mois pour l'année 2010. Les montants seront arrondis à deux décimales.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(prix_total),2) AS montant_total
FROM commandes
WHERE date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1 ;
```

Attention : il n'y a pas de contrainte `NOT NULL` sur le champ `date_commande`, donc s'il existe des commandes sans date de commande, celles-ci seront agrégées à part des autres, puisque `extract()` renverra `NULL` pour ces lignes.

> 6. Supprimer les commandes de mai 2010.

```
BEGIN;

/* en raison de la présence de clés étrangères, il faut en premier leur
   supprimer les lignes de la table lignes_commandes correspondant aux
```

```
   commandes à supprimer */
DELETE
FROM lignes_commandes
WHERE numero_commande IN (
   SELECT numero_commande
   FROM commandes
   WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
      AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
   );

-- ensuite seulement on peut supprimer les commandes
DELETE
FROM commandes
WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY') ;

COMMIT ;
```

Le problème de l'approche précédente est d'effectuer l'opération en deux temps. Il est possible de réaliser la totalité des suppressions dans les deux tables `lignes_commandes` et `commandes` en une seule requête en utilisant une *CTE* :

```
WITH del_lc AS (
   /* ici on déclare la CTE qui va se charger de supprimer les lignes dans la
      table lignes_commandes et retourner les numeros de commande supprimés */
   DELETE
   FROM lignes_commandes
   WHERE numero_commande IN (
      SELECT numero_commande
      FROM commandes
      WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
         AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
      )
   RETURNING numero_commande
)
/* requête principale, qui supprime les commandes dont les numéros
   correspondent aux numéros de commandes remontés par la CTE */
DELETE
FROM commandes c
WHERE EXISTS (
   SELECT 1
   FROM del_lc
   WHERE del_lc.numero_commande = c.numero_commande
   ) ;
```

> 7. Ré-exécuter la requête trouvée au point 5.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(prix_total),2) AS montant_total
FROM commandes
GROUP BY 1
ORDER BY 1 ;
```

> 8. Qu'observez-vous ?

La ligne correspondant au mois de mai a disparu.

> 9. Corriger le problème rencontré.

```
SELECT numero_mois, round(coalesce(sum(prix_total), 0.0),2) AS montant_total
  /* la fonction generate_series permet de générer une pseudo-table d'une
     colonne contenant les chiffres de 1 à 12 */
FROM generate_series(1, 12) AS numero_mois
  /* le LEFT JOIN entre cette pseudo-table et la table commandes permet de
     s'assurer que même si aucune commande n'a eu lieu sur un mois, la ligne
     correspondante sera tout de même présente */
LEFT JOIN commandes ON extract('month' from date_commande) = numero_mois
      AND date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
      AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1 ;
```

Notez l'utilisation de la fonction `coalesce()` dans le `SELECT`, afin d'affecter la valeur `0.0` aux lignes « ajoutées » par le `LEFT JOIN` qui n'ont par défaut aucune valeur (`NULL`).

> 10. Créer une vue calculant le prix total de chaque commande.

```
CREATE VIEW commande_montant AS
SELECT
   numero_commande,
   sum(quantite * prix_unitaire - remise) AS total_commande
FROM lignes_commandes
GROUP BY numero_commande ;
```

> 11. Réécrire la requête du point 5 pour utiliser la vue créée au point 10.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(total_commande),2) AS montant_total
FROM commandes c
JOIN commande_montant cm ON cm.numero_commande = c.numero_commande
WHERE date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1 ;
```

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

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

LIVRES BLANCS

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

TÉLÉCHARGEMENT GRATUIT

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

DALIBO, L'EXPERTISE POSTGRESQL

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

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

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