

Module S70

Analyse de données avec SQL



25.09

Table des matières

Sur ce document	1
Chers lectrices & lecteurs,	1
À propos de DALIBO	1
Remerciements	2
Forme de ce manuel	2
Licence Creative Commons CC-BY-NC-SA	2
Marques déposées	3
Versions de PostgreSQL couvertes	3
1/ SQL pour l'analyse de données	5
1.1 Préambule	6
1.1.1 Menu	6
1.1.2 Objectifs	6
1.1.3 Tables d'exemple	6
1.2 Agrégats	10
1.2.1 Agrégats avec GROUP BY	11
1.2.2 GROUP BY : principe	12
1.2.3 GROUP BY : syntaxe et exemple	13
1.2.4 Agrégats et ORDER BY	14
1.2.5 Utiliser ORDER BY avec un agrégat	14
1.2.6 Créer un tableau avec un agrégat : array_agg	15
1.3 Clause FILTER	16
1.3.1 Filtrer avec CASE	16
1.3.2 Filtrer avec FILTER (WHERE...)	17
1.4 Fonctions de fenêtrage	18
1.4.1 Fenêtrage et regroupement : premier exemple	18
1.4.2 Fonctions de fenêtrage : utilisation	20
1.4.3 Fenêtrage et regroupement : OVER (PARTITION BY ...)	21
1.4.4 Fenêtrage et tri : OVER (ORDER BY ...)	22
1.4.5 Fenêtrage et tri : row_number()	22
1.4.6 Fenêtrage et tri : numéroté des lignes sans critère	23
1.4.7 Fenêtrage et tri : rang	24
1.4.8 Fenêtrage et tri : somme glissante (exemple)	24
1.4.9 Fenêtrage et tri : somme glissante (principe)	25
1.4.10 Fenêtrage : regroupement et tri	26
1.4.11 Regroupement et tri : exemple	26
1.4.12 Regroupement et tri : principe	28
1.4.13 Fonctions analytiques	28
1.4.14 lead() et lag() : exemple	29
1.4.15 lead() et lag() : principe	30
1.4.16 first/last/nth_value	30
1.4.17 first/last/nth_value : exemple	31

1.4.18	Clause WINDOW	32
1.4.19	Fenêtre de travail	33
1.4.20	Fenêtre de travail avec RANGE	33
1.4.21	Fenêtre de travail avec RANGE : exemple	34
1.4.22	Fenêtre de travail avec ROWS	34
1.4.23	Fenêtre de travail avec GROUPS	36
1.4.24	Définition de la fenêtre : EXCLUDE	36
1.5	WITHIN GROUP	37
1.5.1	WITHIN GROUP : exemple	37
1.6	Regroupement avancés	38
1.6.1	GROUPING SETS : données d'exemple	38
1.6.2	GROUPING SETS : exemple visuel	39
1.6.3	Émuler les GROUPING SETS avec GROUP BY	39
1.6.4	GROUPING SETS : exemple	40
1.6.5	ROLLUP	41
1.6.6	ROLLUP : exemple visuel	41
1.6.7	ROLLUP : exemple et résultat	42
1.6.8	CUBE	44
1.6.9	CUBE : exemple visuel	44
1.6.10	CUBE : Syntaxe	45
1.6.11	GROUPING SETS, ROLLUP ou CUBE?	46
1.6.12	Filtrer les lignes d'un certain regroupement	47
1.6.13	Affichage des tableaux croisés	48
1.7	Conclusion	51
1.8	Travaux pratiques	52
1.9	Travaux pratiques (solutions)	54
Les formations Dalibo		65
	Cursus des formations	65
	Les livres blancs	66
	Téléchargement gratuit	66

Sur ce document

Formation	Module S70
Titre	Analyse de données avec SQL
Révision	25.09
PDF	https://dali.bo/s70_pdf
EPUB	https://dali.bo/s70_epub
HTML	https://dali.bo/s70_html
Slides	https://dali.bo/s70_slides
TP	https://dali.bo/s70_tp
TP (solutions)	https://dali.bo/s70_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

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.com¹ !

À propos de DALIBO

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

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

¹<mailto:formation@dalibo.com>

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**². Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

²<http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

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

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

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées³ par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

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

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

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

³<https://www.postgresql.org/about/policies/trademarks/>

1/ SQL pour l'analyse de données

1.1 PRÉAMBULE



- Analyser des données est facile avec PostgreSQL
 - opérations d'agrégation disponibles
 - fonctions OLAP avancées

1.1.1 Menu



- Agrégation de données
- Clause `FILTER`
- Fonctions `WINDOW`
- `GROUPING SETS`, `ROLLUP`, `CUBE`
- `WITHIN GROUPS`

1.1.2 Objectifs



- Écrire des requêtes encore plus complexes
- Analyser les données en amont
 - pour ne récupérer que le résultat

1.1.3 Tables d'exemple



- Table `employees`
- Tables `population` et `continents`

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

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

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

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

```
-- Données
```

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'Dupuis', 'Direction', 10000.00),
('00000004', 'Fantasio', 'Courrier', 4500.00),
('00000006', 'Prunelle', 'Publication', 4000.00),
('00000020', 'Lagaffe', 'Courrier', 3000.00),
('00000040', 'Lebrac', 'Publication', 3000.00);
```

```
SELECT * FROM employes ;
```

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

(5 lignes)

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

```
-- Nettoyage des tables si elles existent
```

```
DROP TABLE IF EXISTS population CASCADE ;
DROP TABLE IF EXISTS continents CASCADE ;
```

```
-- Tables
```

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

```
-- Données des continents
```

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

```
-- Données des pays
```

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

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

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

-- Échantillon

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

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

1.2 AGRÉGATS



- SQL dispose de fonctions de calcul d'agrégats
- Utilité :
 - calcul de sommes, moyennes, valeur minimale et maximale
 - nombreuses fonctions statistiques disponibles

À l'aide des fonctions de calcul d'agrégats, on peut réaliser un certain nombre de calculs permettant d'analyser les données d'une table.

Ainsi, on peut calculer :

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

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
FROM   employes;
```

salaire_moyen	salaire_maximum	salaire_minimum	somme_salaires
4900.00000000000000000000	10000.00	3000.00	24500.00

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

Si l'on applique un filtre sur les données, par exemple pour ne prendre en compte que le service *Courrier*, alors PostgreSQL réalise le calcul uniquement sur les données issues de la lecture :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
FROM   employes
WHERE  service = 'Courrier';
```

salaire_moyen	salaire_maximum	salaire_minimum	somme_salaires
3750.00000000000000000000	4500.00	3000.00	7500.00

Une limitation : il n'est pas possible de référencer d'autres colonnes pour les afficher à côté du résultat d'un calcul d'agrégation à moins de les utiliser comme critère de regroupement avec `GROUP BY` :

```
SELECT avg(salaire), nom FROM employes;
```

```
ERROR: column "employes.nom" must appear in the GROUP BY clause or be used in
       an aggregate function
```

```
LIGNE 1 : SELECT avg(salaire), nom FROM employes;
                        ^
```

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

1.2.1 Agrégats avec GROUP BY



- agrégat + `GROUP BY`
- Utilité
 - effectue des calculs sur des regroupements : moyenne, somme, comptage, etc.
 - regroupement selon un critère défini par la clause `GROUP BY`
 - exemple : calcul du salaire moyen de chaque service

L'opérateur d'agrégat `GROUP BY` indique à la base de données que l'on souhaite regrouper les données selon les mêmes valeurs d'une colonne.

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

FIGURE 1/ .1 – Opérateur GROUP BY

Des calculs pourront être réalisés sur les données agrégées selon le critère de regroupement donné. Le résultat sera alors représenté en n'affichant que les colonnes de regroupement puis les valeurs calculées par les fonctions d'agrégation :

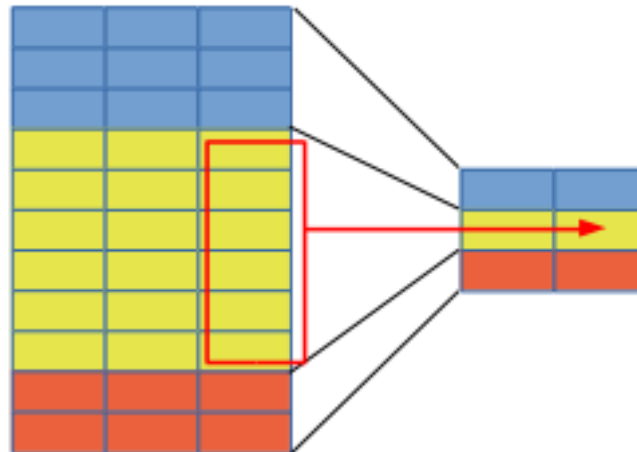


FIGURE 1/.2 – Agrégats

1.2.2 GROUP BY : principe

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

service	salaires_par_service
Courrier	7500.00
Direction	10000.00
Publication	7000.00

L'agrégation est ici réalisée sur la colonne `service`. En guise de calcul d'agrégation, une somme est réalisée sur les salaires payés dans chaque service.

1.2.3 GROUP BY : syntaxe et exemple



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

service	salaires_par_service
Courrier	7500.00
Direction	10000.00
Publication	7000.00

(3 lignes)

SQL permet depuis le début de réaliser des calculs d'agrégation. Pour cela, la base de données observe les critères de regroupement définis dans la clause `GROUP BY` de la requête et effectue l'opération sur l'ensemble des lignes qui correspondent au critère de regroupement.

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

```
SELECT service,
       sum(salaire) AS salaires_par_service
FROM   employes GROUP BY service
UNION ALL
SELECT 'Total' AS service,
       sum(salaire) AS salaires_par_service
FROM   employes;
```

service	salaires_par_service
Courrier	7500.00
Direction	10000.00
Publication	7000.00
Total	24500.00

(4 lignes)

On le verra plus loin, cette dernière requête peut être écrite plus simplement avec les `GROUPING SETS`.

1.2.4 Agrégats et ORDER BY



- Extension propriétaire de PostgreSQL
- `ORDER BY` dans la fonction d'agrégat
- Utilité :
 - ordonner les données agrégées
 - surtout utile avec `array_agg`, `string_agg` et `xmlagg`

Les fonctions `array_agg`, `string_agg` et `xmlagg` permettent d'agréger des éléments dans un tableau, dans une chaîne ou dans une arborescence XML. Autant l'ordre dans lequel les données sont utilisées n'a pas d'importance lorsque l'on réalise un calcul d'agrégat classique, autant cet ordre va influencer la façon dont les données seront produites par les trois fonctions citées plus haut. En effet, le tableau généré par `array_agg` est composé d'éléments ordonnés, de même que la chaîne de caractères ou l'arborescence XML.

1.2.5 Utiliser ORDER BY avec un agrégat



```
SELECT service,
       string_agg(nom, ' ' ORDER BY nom) AS liste_employes
FROM   employes
GROUP BY service;
```

service	liste_employes
Courrier	Fantasio, Lagaffe
Direction	Dupuis
Publication	Lebrac, Prunelle

(3 lignes)

La requête suivante permet d'obtenir, pour chaque service, la liste des employés dans un tableau, trié par ordre alphabétique :

```
SELECT service,
       string_agg(nom, ' ' ORDER BY nom) AS liste_employes
FROM   employes
GROUP BY service;
```

service	liste_employes
Courrier	Fantasio, Lagaffe
Direction	Dupuis
Publication	Lebrac, Prunelle

(3 lignes)

1.2.6 Créer un tableau avec un agrégat : array_agg



```
SELECT service,  
       array_agg(nom ORDER BY nom) AS liste_employes  
FROM   employes  
GROUP BY service;
```

service	liste_employes
Courrier	{Fantasio,Lagaffe}
Direction	{Dupuis}
Publication	{Lebrac,Prunelle}

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

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

1.3 CLAUSE FILTER



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

La clause `FILTER` permet de remplacer des expressions complexes écrites avec `CASE` et donc de simplifier l'écriture de requêtes réalisant un filtrage dans une fonction d'agrégat.

1.3.1 Filtrer avec CASE



- Syntaxe fastidieuse :

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

compte_pays	compte_pays_europeens
88	34

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

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

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

Le `CASE` peut devenir arbitrairement complexe.

Cela fonctionne, mais dès que l'on a besoin d'avoir de multiples filtres, ou des filtres plus complexes, la requête devient très rapidement peu lisible et difficile à maintenir. Le risque d'erreur est également élevé.

1.3.2 Filtrer avec FILTER (WHERE...)



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

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

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

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

1.4 FONCTIONS DE FENÊTRAGE



Des fonctionnalités trop peu connues

1.4.1 Fenêtrage et regroupement : premier exemple



```
SELECT matricule, salaire, service,
       SUM(salaire) OVER (PARTITION BY service)
       AS total_salaire_service
FROM employes;
```

matricule	salaire	service	total_salaire_service
00000004	4500.00	Courrier	7500.00
00000020	3000.00	Courrier	7500.00
00000001	10000.00	Direction	10000.00
00000006	4000.00	Publication	7000.00
00000040	3000.00	Publication	7000.00

Les calculs réalisés par cette requête sont identiques à ceux réalisés avec une agrégation utilisant `GROUP BY`. La principale différence est que l'on évite ici de perdre le détail des données tout en disposant des données agrégées dans le résultat de la requête.

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

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

Les données sont regroupées par service :

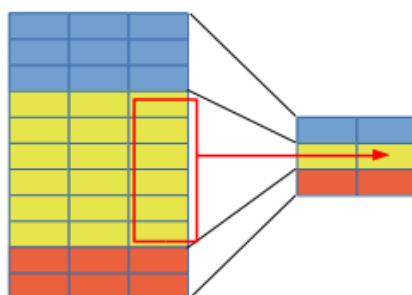


FIGURE 1/ .3 – Agrégation de lignes

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

```
SELECT matricule, salaire, service,
       SUM(salaire) OVER (PARTITION BY service)
       AS total_salaire_service
FROM   employes ;
```

matricule	salaire	service	total_salaire_service
00000004	4500.00	Courrier	7500.00
00000020	3000.00	Courrier	7500.00
00000001	10000.00	Direction	10000.00
00000006	4000.00	Publication	7000.00
00000040	3000.00	Publication	7000.00

Ce schéma résume le principe :

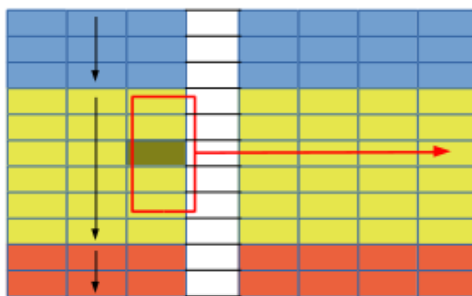


FIGURE 1/ .4 – Fonction de fenêtrage

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

- des sous-requêtes effectuées sur chaque ligne :

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

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

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

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

```

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

```

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



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

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

1.4.2 Fonctions de fenêtrage : utilisation



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

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

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

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

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

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

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

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



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

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

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



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

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



OVER (ORDER BY matricule)

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

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

1.4.5 Fenêtrage et tri : row_number()



- Pour numéroter des lignes :

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

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

(5 lignes)

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

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

1.4.6 Fenêtrage et tri : numéroté des lignes sans critère



— Pour numéroté des lignes :

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

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

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

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

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

row_number	matricule	nom
3	00000006	Prunelle
5	00000040	Lebrac
4	00000020	Lagaffe

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

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

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

row_number	matricule	nom
1	00000006	Prunelle
2	00000040	Lebrac
3	00000020	Lagaffe

1.4.7 Fenêtrage et tri : rang



— Rang selon le salaire :

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

matricule	nom	salaire	service	rank	dense_rank
00000020	Lagaffe	3000.00	Courrier	1	1
00000040	Lebrac	3000.00	Publication	1	1
00000006	Prunelle	4000.00	Publication	3	2
00000004	Fantasio	4500.00	Courrier	4	3
00000001	Dupuis	10000.00	Direction	5	4

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

Ces fonctions seraient beaucoup plus compliquées à écrire en SQL.

1.4.8 Fenêtrage et tri : somme glissante (exemple)



— Calcul d'une somme glissante :

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

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

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

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

```

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

```

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

1.4.9 Fenêtrage et tri : somme glissante (principe)



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

matricule	salaire	
00000001	10000.00	
00000004	4500.00	
00000006	4000.00	
00000020	3000.00	
00000040	3000.00	

Fenêtre de calcul pour la ligne courante

SUM(salaire)

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

Lorsque l'on utilise une clause de tri, la portion de données visible par l'opérateur d'agrégat correspond aux données comprises entre la première ligne examinée et la ligne courante.

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

```

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

```

```

)
FROM employes
ORDER BY matricule ;

```

Nous verrons que cette fenêtre peut se changer.

1.4.10 Fenêtrage : regroupement et tri



- On peut combiner :
 - `OVER (PARTITION BY ... ORDER BY ...)`
- Utilité :
 - travailler sur des jeux de données ordonnés et isolés les uns des autres

Il est possible de combiner les clauses de fenêtrage `PARTITION BY` et `ORDER BY`. Cela permet d'isoler des jeux de données entre eux avec la clause `PARTITION BY`, tout en appliquant un critère de tri avec la clause `ORDER BY`. Beaucoup d'applications sont possibles si l'on associe à cela les nombreuses fonctions analytiques disponibles.

1.4.11 Regroupement et tri : exemple



```

SELECT continent, pays, population,
       rank() OVER (PARTITION BY continent
                    ORDER BY population DESC)
       AS rang
FROM population;

```

continent	pays	population	rang
Afrique	Nigéria	173.6	1
Afrique	Éthiopie	94.1	2
Afrique	Égypte	82.1	3
Afrique	Rép. dém. du Congo	67.5	4
(...)			
Amérique du Nord	États-Unis	320.1	1
Amérique du Nord	Canada	35.2	2
(...)			

Si l'on applique les deux clauses `PARTITION BY` et `ORDER BY` à une fonction de fenêtrage, alors le critère de tri est appliqué dans la partition et chaque partition est indépendante l'une de l'autre.

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

Voici un extrait plus complet du résultat de la requête présentée ci-dessus :

DALIBO Formations

continent	pays	population	rang_pop
Afrique	Nigéria	173.6	1
Afrique	Éthiopie	94.1	2
Afrique	Égypte	82.1	3
Afrique	Rép. dém. du Congo	67.5	4
Afrique	Afrique du Sud	52.8	5
Afrique	Tanzanie	49.3	6
Afrique	Kenya	44.4	7
Afrique	Algérie	39.2	8
Afrique	Ouganda	37.6	9
Afrique	Maroc	33.0	10
Afrique	Ghana	25.9	11
Afrique	Mozambique	25.8	12
Afrique	Madagascar	22.9	13
Afrique	Côte-d'Ivoire	20.3	14
Afrique	Niger	17.8	15
Afrique	Burkina Faso	16.9	16
Afrique	Zimbabwe	14.1	17
Afrique	Soudan	14.1	17
Afrique	Tunisie	11.0	19
Amérique du Nord	États-Unis	320.1	1
Amérique du Nord	Canada	35.2	2
Amérique latine. Caraïbes	Brésil	200.4	1
Amérique latine. Caraïbes	Mexique	122.3	2
Amérique latine. Caraïbes	Colombie	48.3	3
Amérique latine. Caraïbes	Argentine	41.4	4
Amérique latine. Caraïbes	Pérou	30.4	5
Amérique latine. Caraïbes	Venezuela	30.4	5
Amérique latine. Caraïbes	Chili	17.6	7
Amérique latine. Caraïbes	Équateur	15.7	8
Amérique latine. Caraïbes	Guatemala	15.5	9
Amérique latine. Caraïbes	Cuba	11.3	10

(...)

1.4.12 Regroupement et tri : principe



OVER (**PARTITION BY** continent
ORDER BY population **DESC**)

pays	continent	population
Chine	Asie	1385.6
Iraq	Asie	33.8
Ouzbékistan	Asie	28.9
Arabie Saoudite	Asie	28.8
France métropolitaine	Europe	64.3
Finlande	Europe	5.4
Lettonie	Europe	2.1

Cette construction ne pose aucune difficulté syntaxique. La norme impose de placer la clause `PARTITION BY` avant la clause `ORDER BY`, c'est la seule chose à retenir au niveau de la syntaxe.

1.4.13 Fonctions analytiques



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

Sans les fonctions analytiques, il serait difficile en SQL d'écrire des requêtes nécessitant de faire appel à des données provenant d'autres lignes que la ligne courante.

Par exemple, pour renvoyer la liste détaillée de tous les employés ET le salaire le plus élevé du service auquel il appartient, on peut utiliser la fonction `first_value()` :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire)
         OVER (PARTITION BY service
              ORDER BY salaire DESC)
         AS salaire_maximum_service
FROM   employes ;
```

matricule		nom		salaire		service		salaire_maximum_service
-----------	--	-----	--	---------	--	---------	--	-------------------------

00000004	Fantasio	4500.00	Courrier	4500.00
00000020	Lagaffe	3000.00	Courrier	4500.00
00000001	Dupuis	10000.00	Direction	10000.00
00000006	Prunelle	4000.00	Publication	4000.00
00000040	Lebrac	3000.00	Publication	4000.00

1.4.14 lead() et lag() : exemple



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

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

(53 lignes)

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

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

1.4.15 lead() et lag() : principe



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

pays	continent	population	lag
Chine	Asie	1385.6	
Iraq	Asie	33.8	1385.6
Ouzbékistan	Asie	28.9	33.8
Arabie Saoudite	Asie	28.8	28.9
France métropolitaine	Europe	64.3	
Finlande	Europe	5.4	64.3
Lettonie	Europe	2.1	5.4

lag(population, 1)

1.4.16 first/last/nth_value



- `first_value(colonne)`
— retourne la première valeur pour la colonne
- `last_value(colonne)`
— retourne la dernière valeur pour la colonne
- `nth_value(colonne, n)`
— retourne la n-ème valeur (en comptant à partir de 1) pour la colonne

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

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

La liste complète est dans la documentation¹.

1.4.17 first/last/nth_value : exemple



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

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



Lorsque la clause `ORDER BY` est utilisée pour définir une fenêtre, la fenêtre visible depuis la ligne courante commence par défaut à la première ligne de résultat et s'arrête à la ligne courante incluse (clause implicite `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

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

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

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

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

Afrique du Sud | Afrique | 52.8 | 52.8
...

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

1.4.18 Clause WINDOW



— Pour factoriser la définition d'une fenêtre :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
FROM   employes
WINDOW w AS (ORDER BY salaire) ;
```

— Plusieurs fenêtres possibles

Il arrive que l'on ait besoin d'utiliser plusieurs fonctions de fenêtrage au sein d'une même requête qui utilisent la même définition de fenêtre (même clause `PARTITION BY` et/ou `ORDER BY`). Afin d'éviter de dupliquer cette clause, il est possible de définir une fenêtre nommée et de l'utiliser à plusieurs endroits de la requête. Par exemple, l'exemple précédant des fonctions de classement pourrait s'écrire :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
FROM   employes
WINDOW w AS (ORDER BY salaire);
```

matricule	nom	salaire	service	rank	dense_rank
00000020	Lagaffe	3000.00	Courrier	1	1
00000040	Lebrac	3000.00	Publication	1	1
00000006	Prunelle	4000.00	Publication	3	2
00000004	Fantasio	4500.00	Courrier	4	3
00000001	Dupuis	10000.00	Direction	5	4

(5 lignes)

À noter qu'il est possible de définir de multiples définitions de fenêtres au sein d'une même requête, et qu'une définition de fenêtre peut surcharger la clause `ORDER BY` si la définition parente ne l'a pas définie. Par exemple, la requête SQL suivante est correcte :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w_asc,
       dense_rank() OVER w_desc
FROM   employes
WINDOW w AS (PARTITION BY service),
       w_asc AS (w ORDER BY salaire),
       w_desc AS (w ORDER BY salaire DESC);
```

1.4.19 Fenêtre de travail



- La fenêtre d'`OVER (ORDER BY ...)` par défaut est :

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

- Toutes les lignes :

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

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

Lorsque la clause `ORDER BY` est utilisée pour définir une fenêtre, la fenêtre visible depuis la ligne courante commence par défaut à la première ligne de résultat et s'arrête à la ligne courante incluse (clause implicite `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

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

1.4.20 Fenêtre de travail avec RANGE



- Intervalle à bornes *flou*
- Borne de départ :
 - `UNBOUNDED PRECEDING` : depuis le début de la partition
 - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
 - `UNBOUNDED FOLLOWING` : jusqu'à la fin de la partition
 - `CURRENT ROW` : jusqu'à la ligne courante

**OVER (PARTITION BY ...
ORDER BY ...
RANGE BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING)**

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

1.4.21 Fenêtre de travail avec RANGE : exemple



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

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

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

1.4.22 Fenêtre de travail avec ROWS



- Intervalle borné par un nombre de ligne défini avant et après la ligne courante
 - Borne de départ :
 - `xxx PRECEDING` : depuis les xxx valeurs devant la ligne courante
 - `CURRENT ROW` : depuis la ligne courante
 - Borne de fin :
 - `xxx FOLLOWING` : depuis les xxx valeurs derrière la ligne courante
 - `CURRENT ROW` : jusqu'à la ligne courante
- ```
OVER (PARTITION BY ...
 ORDER BY ...
 ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING)
```
- Exemple : moyenne glissante

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

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

### Exemple :

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

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

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

(12 lignes)

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

### 1.4.23 Fenêtre de travail avec GROUPS



- Intervalle borné par un groupe de lignes de valeurs identiques défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx groupes de valeurs identiques devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante ou le premier élément identique selon `ORDER BY`
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx groupes de valeurs identiques derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante ou le dernier élément identique selon `ORDER BY`

**OVER (PARTITION BY ...  
ORDER BY ...  
GROUPS BETWEEN 2 PRECEDING AND 1 FOLLOWING**

### 1.4.24 Définition de la fenêtre : EXCLUDE



- Lignes à exclure de la fenêtre de données
- `EXCLUDE CURRENT ROW` : exclut la ligne courante
- `EXCLUDE GROUP` : exclut la ligne courante et le groupe de valeurs identiques dans l'ordre
- `EXCLUDE TIES` : exclut les valeurs identiques à la ligne courante dans l'ordre mais pas la ligne courante
- `EXCLUDE NO OTHERS` : pas d'exclusion (par défaut)

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

## 1.5 WITHIN GROUP



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

La clause `WITHIN GROUP` est une clause pour les agrégats utilisant des fonctions dont les données doivent être triées. Quelques fonctions ont été ajoutées pour profiter au mieux de cette nouvelle clause.

### 1.5.1 WITHIN GROUP : exemple



```
SELECT continent,
 percentile_disc(0.5)
 WITHIN GROUP (ORDER BY population) AS "mediane",
 percentile_disc(0.95)
 WITHIN GROUP (ORDER BY population) AS "95pct",
 ROUND(AVG(population), 1) AS moyenne
FROM population
GROUP BY continent ;
```

| continent                 | mediane | 95pct  | moyenne |
|---------------------------|---------|--------|---------|
| Afrique                   | 33.0    | 173.6  | 44.3    |
| Amérique du Nord          | 35.2    | 320.1  | 177.7   |
| Amérique latine. Caraïbes | 30.4    | 200.4  | 53.3    |
| Asie                      | 53.3    | 1252.1 | 179.9   |
| Europe                    | 9.4     | 82.7   | 21.8    |

Cet exemple permet d'afficher le continent, la médiane de la population par continent et la population du pays le moins peuplé parmi les 5 % de pays les plus peuplés de chaque continent.

## 1.6 REGROUPEMENT AVANCÉS



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

Les `GROUPING SETS` permettent de définir plusieurs clauses d'agrégation `GROUP BY`. Les résultats seront présentés comme si plusieurs requêtes d'agrégation avec les clauses `GROUP BY` mentionnées étaient assemblées avec `UNION ALL`.

### 1.6.1 GROUPING SETS : données d'exemple



```
DROP TABLE IF EXISTS stock ;

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

INSERT INTO stock
VALUES ('ecrous', 'est', 50),
 ('ecrous', 'ouest', 0),
 ('ecrous', 'sud', 40),
 ('clous', 'est', 70),
 ('clous', 'nord', 0),
 ('vis', 'ouest', 50),
 ('vis', 'sud', 50),
 ('vis', 'nord', 60) ;
```

### 1.6.2 GROUPING SETS : exemple visuel

sum (quantite) ... grouping sets (piece,region)

| piece/region | est | ouest | sud | nord | Total |
|--------------|-----|-------|-----|------|-------|
| ecrous       | 50  | 0     | 40  |      | 90    |
| clous        | 70  |       |     | 0    | 70    |
| vis          |     | 50    | 50  | 60   | 160   |
| Total        | 120 | 50    | 90  | 60   |       |



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

### 1.6.3 Émuler les GROUPING SETS avec GROUP BY



Sans `GROUPING SET` : deux `GROUP BY`

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

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

Le comportement de la clause `GROUPING SETS` peut être émulé avec deux requêtes utilisant chacune une clause `GROUP BY` sur les colonnes de regroupement souhaitées et en prévoyant des colonnes vides à `NULL` pour les champs non calculés.

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

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

#### QUERY PLAN

Append

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

### 1.6.4 GROUPING SETS : exemple



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

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

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

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

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

## QUERY PLAN

```

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

```

## 1.6.5 ROLLUP



- ROLLUP
- Utilité :
  - calcul de totaux dans la même requête

La clause `ROLLUP` est une fonctionnalité d'analyse type OLAP du langage SQL. Elle s'utilise dans la clause `GROUP BY`, tout comme `GROUPING SETS`

## 1.6.6 ROLLUP : exemple visuel



sum (quantite) ... ROLLUP (piece,region)

| piece/region | est | ouest | sud | nord | Total |
|--------------|-----|-------|-----|------|-------|
| ecrous       | 50  | 0     | 40  |      | 90    |
| clous        | 70  |       |     | 0    | 70    |
| vis          |     | 50    | 50  | 60   | 160   |
| Total        |     |       |     |      | 320   |

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

### 1.6.7 ROLLUP : exemple et résultat



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

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

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

Cette requête est équivalente à la requête suivante utilisant `GROUPING SETS` :

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

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

#### Exemple :

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

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

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

Pour ce TP, figer les paramètres suivants :

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

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

```
SELECT type_client, code_pays,
 SUM(quantite*prix_unitaire) AS montant
FROM magasin.commandes c
JOIN magasin.lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN magasin.clients cl
 ON (c.client_id = cl.client_id)
JOIN magasin.contacts co
 ON (cl.contact_id = co.contact_id)
WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays)
ORDER BY type_client NULLS FIRST,
 code_pays NULLS FIRST;
```

Elle produit le résultat suivant :

| type_client | code_pays | montant       |
|-------------|-----------|---------------|
| A           |           | 5217862160.65 |
|             |           | 111557177.00  |
|             | CA        | 6273168.32    |
|             | CN        | 7928641.50    |
|             | DE        | 6642061.57    |
|             | DZ        | 6404425.16    |
|             | FR        | 55261295.52   |
|             | IN        | 7224008.95    |
|             | PE        | 7356239.93    |
|             | RU        | 6766644.98    |
| E           | US        | 7700691.07    |
|             |           | 414152232.57  |
|             | CA        | 28457655.81   |
|             | CN        | 25537539.68   |
|             | DE        | 25508815.68   |
|             | DZ        | 24821750.17   |
|             | FR        | 209402443.24  |
|             | IN        | 26788642.27   |
|             | PE        | 24541974.54   |
|             | RU        | 25397116.39   |
| P           | US        | 23696294.79   |
|             |           | 4692152751.08 |
|             | CA        | 292975985.52  |
|             | CN        | 287795272.87  |
|             | DE        | 287337725.21  |
|             | DZ        | 302501132.54  |
|             | FR        | 2341977444.49 |
|             | IN        | 295256262.73  |
|             | PE        | 300278960.24  |
|             | RU        | 287605812.99  |
|             | US        | 296424154.49  |

### 1.6.8 CUBE



- CUBE
- Utilité :
  - calcul de totaux dans la même requête
  - sur toutes les clauses de regroupement

La clause `CUBE` est une autre fonctionnalité d'analyse type OLAP du langage SQL. Tout comme `ROLLUP`, elle s'utilise dans la clause `GROUP BY`.

### 1.6.9 CUBE : exemple visuel



| sum (quantite) ... CUBE (piece,region) |     |       |     |      |       |
|----------------------------------------|-----|-------|-----|------|-------|
| piece/region                           | est | ouest | sud | nord | Total |
| ecrous                                 | 50  | 0     | 40  |      | 90    |
| clous                                  | 70  |       |     | 0    | 70    |
| vis                                    |     | 50    | 50  | 60   | 160   |
| Total                                  | 120 | 50    | 90  | 60   | 320   |

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

<sup>2</sup>[https://en.wikipedia.org/wiki/OLAP\\_cube](https://en.wikipedia.org/wiki/OLAP_cube)

### 1.6.10 CUBE : Syntaxe



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

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

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

Cette requête est équivalente à la requête suivante utilisant `GROUPING SETS` :

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

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

```
SELECT type_client, code_pays,
 SUM(quantite*prix_unitaire) AS montant
FROM magasin.commandes c
JOIN magasin.lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN magasin.clients cl
 ON (c.client_id = cl.client_id)
JOIN magasin.contacts co
 ON (cl.contact_id = co.contact_id)
WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays)
```

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

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

| type_client | code_pays | montant       |
|-------------|-----------|---------------|
|             |           | 5217862160.65 |
|             | CA        | 327706809.65  |
|             | CN        | 321261454.05  |
|             | DE        | 319488602.46  |
|             | DZ        | 333727307.87  |
|             | FR        | 2606641183.25 |
|             | IN        | 329268913.95  |
|             | PE        | 332177174.71  |
|             | RU        | 319769574.36  |
|             | US        | 327821140.35  |
| A           |           | 111557177.00  |
| A           | CA        | 6273168.32    |
| A           | CN        | 7928641.50    |
| A           | DE        | 6642061.57    |
| A           | DZ        | 6404425.16    |
| A           | FR        | 55261295.52   |
| A           | IN        | 7224008.95    |
| A           | PE        | 7356239.93    |
| A           | RU        | 6766644.98    |
| A           | US        | 7700691.07    |
| E           |           | 414152232.57  |
| E           | CA        | 28457655.81   |
| E           | CN        | 25537539.68   |
| E           | DE        | 25508815.68   |
| E           | DZ        | 24821750.17   |
| E           | FR        | 209402443.24  |
| E           | IN        | 26788642.27   |
| E           | PE        | 24541974.54   |
| E           | RU        | 25397116.39   |
| E           | US        | 23696294.79   |
| P           |           | 4692152751.08 |
| P           | CA        | 292975985.52  |
| P           | CN        | 287795272.87  |
| P           | DE        | 287337725.21  |
| P           | DZ        | 302501132.54  |
| P           | FR        | 2341977444.49 |
| P           | IN        | 295256262.73  |
| P           | PE        | 300278960.24  |
| P           | RU        | 287605812.99  |
| P           | US        | 296424154.49  |

### 1.6.11 GROUPING SETS, ROLLUP ou CUBE ?



- CUBE peut le plus
- mais forcément plus coûteux

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

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

### 1.6.12 Filtrer les lignes d'un certain regroupement



Pour que l'application repère les regroupements :

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

Pour que l'application distingue rigoureusement les lignes appartenant à un certain niveau de regroupement, on peut utiliser la fonction `GROUPING`. En effet la colonne de regroupement peut posséder des valeurs `NULL` légitimes, il est alors difficile de les distinguer. L'exemple suivant montre une requête qui exploite cette fonction :

```
SELECT GROUPING(type_client,code_pays)::bit(2),
 GROUPING(type_client)::boolean g_type_cli,
 GROUPING(code_pays)::boolean g_code_pays,
 type_client,
 code_pays,
 SUM(quantite*prix_unitaire) AS montant
FROM magasin.commandes c
JOIN magasin.lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN magasin.clients cl
 ON (c.client_id = cl.client_id)
JOIN magasin.contacts co
 ON (cl.contact_id = co.contact_id)
WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

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

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

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

(40 rows)

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

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

### 1.6.13 Affichage des tableaux croisés



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

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

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

Ponctuellement, deux outils intégrés à PostgreSQL peuvent dépanner.

### psql et crosstabview :

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

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

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

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

Ou pour inverser :

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

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

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

### tablefunc :

Il existe une extension fournie avec PostgreSQL nommée `tablefunc` qui fournit des fonctions dédiées aux tableaux croisés. Leur maniement n'est pas évident. Il est conseillé d'utiliser la fonction `crosstab (text,text)` qui tient bien compte des valeurs absentes. Cette fonction demande que la requête à formater lui soit fournie comme chaîne de caractère.

```
CREATE EXTENSION IF NOT EXISTS tablefunc ;
```

```
SELECT *
FROM crosstab(
```

```
$$SELECT coalesce (piece, 'Total'),
 coalesce (region, 'Total'),
 sum(quantite)
FROM stock
GROUP BY CUBE (piece,region)
ORDER BY GROUPING (piece,region)
$$,
'SELECT DISTINCT region FROM stock ORDER BY region')
AS ct(piece text, "Est" text, "Nord" text, "Ouest" text, "Sud" text);
```

| piece  | Est | Nord | Ouest | Sud |
|--------|-----|------|-------|-----|
| vis    |     |      | 50    |     |
| ecrous |     |      | 0     |     |
| clous  |     | 0    |       |     |
| vis    |     | 60   |       |     |
| clous  | 70  |      |       |     |
| vis    |     |      |       | 50  |
| ecrous | 50  |      |       | 40  |
| vis    |     |      |       |     |
| clous  |     |      |       |     |
| Total  | 120 | 60   | 50    | 90  |

Attention à la cohérence entre la deuxième requête et les titres sur la dernière ligne, ce n'est pas dynamique!

Cet exemple utilise aussi `$$` à la place des guillemets droits pour délimiter une chaîne, car une requête contient souvent elle-même des chaînes.

— Documentation : `tablefunc`<sup>3</sup>

### Pivot dynamique :

Ce billet de Daniel Vérité de 2018<sup>4</sup> décrit comment obtenir un tableau croisé réellement dynamique. Cela réclame toutefois d'écrire un peu de code.

---

<sup>3</sup><https://docs.postgresql.fr/current/tablefunc.html>

<sup>4</sup><https://blog-postgresql.verite.pro/2018/06/02/crosstab-pivot.html>

## 1.7 CONCLUSION



- Les fonctions fenêtrées existent depuis SQL :2003
- Ne vous limitez pas au SQL du XX<sup>e</sup> siècle

## 1.8 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur [https://dali.bo/s70\\_solutions](https://dali.bo/s70_solutions).

La table `brno2015` peut être téléchargée et restaurée ainsi :

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

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

| Table "public.brno_2015" |          |           |
|--------------------------|----------|-----------|
| Column                   | Type     | Modifiers |
| no_tour                  | integer  |           |
| no_pilote                | integer  |           |
| lap_time                 | interval |           |

Une table `pilotes` permet de connaître les détails d'un pilote :

| Table "public.pilotes" |         |           |
|------------------------|---------|-----------|
| Column                 | Type    | Modifiers |
| no                     | integer |           |
| nom                    | text    |           |
| nationalite            | text    |           |
| ecurie                 | text    |           |
| moto                   | text    |           |

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

### Agrégation

Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?

Déterminer quel est le pilote le plus régulier (écart-type).

### Window Functions

Afficher la place sur le podium pour chaque coureur.

À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.

Pour chaque tour, afficher :

- le nom du pilote;
- son rang dans le tour;
- son temps depuis le début de la course;
- dans le tour, la différence de temps par rapport au premier.

Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?

Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.

En quelle position a terminé le coureur qui a doublé le plus de personnes ? Combien de personnes a-t-il doublées ?

### Grouping Sets

Ce TP s'appuie sur les tables présentes dans le schéma `magasin`.

En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.

Ajouter également le montant total des commandes depuis le début de l'activité.

Ajouter également le montant total des commandes par pays.

À partir de la requête précédente, ajouter une colonne par critère de regroupement, de type booléen, qui est positionnée à `true` lorsque le regroupement est réalisé sur l'ensemble des valeurs de la colonne.

## 1.9 TRAVAUX PRATIQUES (SOLUTIONS)

La table `brno2015` peut être téléchargée et restaurée ainsi :

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

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

| Table "public.brno_2015" |          |           |
|--------------------------|----------|-----------|
| Column                   | Type     | Modifiers |
| no_tour                  | integer  |           |
| no_pilote                | integer  |           |
| lap_time                 | interval |           |

Une table `pilotes` permet de connaître les détails d'un pilote :

| Table "public.pilotes" |         |           |
|------------------------|---------|-----------|
| Column                 | Type    | Modifiers |
| no                     | integer |           |
| nom                    | text    |           |
| nationalite            | text    |           |
| ecurie                 | text    |           |
| moto                   | text    |           |

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

### Agrégation

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour?

Le coureur :

```
SELECT nom, max(lap_time) - min(lap_time) as ecart
FROM brno_2015
JOIN pilotes
ON (no_pilote = no)
GROUP BY 1
ORDER BY 2
LIMIT 1;
```

La requête donne le résultat suivant :

| nom           | ecart        |
|---------------|--------------|
| Jorge LORENZO | 00:00:04.661 |

Déterminer quel est le pilote le plus régulier (écart-type).

Nous excluons le premier tour car il s'agit d'une course avec départ arrêté, donc ce tour est plus lent que les autres, ici d'au moins 8 secondes :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
FROM brno_2015
JOIN pilotes
ON (no_pilote = no)
WHERE no_tour > 1
GROUP BY 1
ORDER BY 2
LIMIT 1;
```

Le résultat montre le coureur qui a abandonné en premier :

| nom             | stddev            |
|-----------------|-------------------|
| Alex DE ANGELIS | 0.130107647741847 |

On s'aperçoit qu'Alex De Angelis n'a pas terminé la course. Il semble donc plus intéressant de ne prendre en compte que les pilotes qui ont terminé la course et toujours en excluant le premier tour (il y a 22 tours sur cette course, on peut le positionner soit en dur dans la requête, soit avec un sous-select permettant de déterminer le nombre maximum de tours) :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
FROM brno_2015
JOIN pilotes
ON (no_pilote = no)
WHERE no_tour > 1
AND no_pilote in (SELECT no_pilote FROM brno_2015 WHERE no_tour=22)
GROUP BY 1
ORDER BY 2
LIMIT 1;
```

Le pilote 19 a donc été le plus régulier :

| nom             | stddev            |
|-----------------|-------------------|
| Alvaro BAUTISTA | 0.222825823492654 |

## Window Functions

Si ce n'est pas déjà fait, nous positionnons le search\_path pour chercher les objets du schéma brno2015 :

```
SET search_path = brno2015;
```

Afficher la place sur le podium pour chaque coureur.

Les coureurs qui ne franchissent pas la ligne d'arrivée sont dans le classement malgré tout. Il faut donc tenir compte de cela dans l'affichage des résultats.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
 nom, ecurie, total_time
FROM (SELECT no_pilote,
 sum(lap_time) over (PARTITION BY no_pilote) as total_time,
 max(no_tour) over (PARTITION BY no_pilote) as max_lap
 FROM brno_2015
) AS race_data
JOIN pilotes
 ON (race_data.no_pilote = pilotes.no)
GROUP BY nom, ecurie, max_lap, total_time
ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

| rang | nom              | ecurie                      | total_time   |
|------|------------------|-----------------------------|--------------|
| 1    | Jorge LORENZO    | Movistar Yamaha MotoGP      | 00:42:53.042 |
| 2    | Marc MARQUEZ     | Repsol Honda Team           | 00:42:57.504 |
| 3    | Valentino ROSSI  | Movistar Yamaha MotoGP      | 00:43:03.439 |
| 4    | Andrea IANNONE   | Ducati Team                 | 00:43:06.113 |
| 5    | Dani PEDROSA     | Repsol Honda Team           | 00:43:08.692 |
| 6    | Andrea DOVIZIOSO | Ducati Team                 | 00:43:08.767 |
| 7    | Bradley SMITH    | Monster Yamaha Tech 3       | 00:43:14.863 |
| 8    | Pol ESPARGARO    | Monster Yamaha Tech 3       | 00:43:16.282 |
| 9    | Aleix ESPARGARO  | Team SUZUKI ECSTAR          | 00:43:36.826 |
| 10   | Danilo PETRUCCI  | Octo Pramac Racing          | 00:43:38.303 |
| 11   | Yonny HERNANDEZ  | Octo Pramac Racing          | 00:43:43.015 |
| 12   | Scott REDDING    | EG 0,0 Marc VDS             | 00:43:43.216 |
| 13   | Alvaro BAUTISTA  | Aprilia Racing Team Gresini | 00:43:47.479 |
| 14   | Stefan BRADL     | Aprilia Racing Team Gresini | 00:43:47.666 |
| 15   | Loris BAZ        | Forward Racing              | 00:43:53.358 |
| 16   | Hector BARBERA   | Avintia Racing              | 00:43:54.637 |
| 17   | Nicky HAYDEN     | Aspar MotoGP Team           | 00:43:55.43  |
| 18   | Mike DI MEGLIO   | Avintia Racing              | 00:43:58.986 |
| 19   | Jack MILLER      | CWM LCR Honda               | 00:44:04.449 |
| 20   | Claudio CORTI    | Forward Racing              | 00:44:43.075 |
| 21   | Karel ABRAHAM    | AB Motoracing               | 00:44:55.697 |
| 22   | Maverick VIÑALES | Team SUZUKI ECSTAR          | 00:29:31.557 |
| 23   | Cal CRUTCHLOW    | CWM LCR Honda               | 00:27:38.315 |
| 24   | Eugene LAVERTY   | Aspar MotoGP Team           | 00:08:04.096 |
| 25   | Alex DE ANGELIS  | E-Motion IodaRacing Team    | 00:06:05.782 |

(25 rows)

À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.

La requête n'est pas beaucoup modifiée, seule la fonction `first_value()` est utilisée pour déterminer le temps du vainqueur, temps qui sera ensuite retranché au temps du coureur courant.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
 nom, ecurie, total_time,
 total_time - first_value(total_time)
```

```

OVER (ORDER BY max_lap desc, total_time asc) AS difference
FROM (SELECT no_pilote,
 sum(lap_time) over (PARTITION BY no_pilote) as total_time,
 max(no_tour) over (PARTITION BY no_pilote) as max_lap
 FROM brno_2015
) AS race_data
JOIN pilotes
ON (race_data.no_pilote = pilotes.no)
GROUP BY nom, ecurie, max_lap, total_time
ORDER BY max_lap desc, total_time asc;

```

La requête affiche le résultat suivant :

| r  | nom              | ecurie                | total_time   | difference    |
|----|------------------|-----------------------|--------------|---------------|
| 1  | Jorge LORENZO    | Movistar Yamaha [...] | 00:42:53.042 | 00:00:00      |
| 2  | Marc MARQUEZ     | Repsol Honda Team     | 00:42:57.504 | 00:00:04.462  |
| 3  | Valentino ROSSI  | Movistar Yamaha [...] | 00:43:03.439 | 00:00:10.397  |
| 4  | Andrea IANNONE   | Ducati Team           | 00:43:06.113 | 00:00:13.071  |
| 5  | Dani PEDROSA     | Repsol Honda Team     | 00:43:08.692 | 00:00:15.65   |
| 6  | Andrea DOVIZIOSO | Ducati Team           | 00:43:08.767 | 00:00:15.725  |
| 7  | Bradley SMITH    | Monster Yamaha Tech 3 | 00:43:14.863 | 00:00:21.821  |
| 8  | Pol ESPARGARO    | Monster Yamaha Tech 3 | 00:43:16.282 | 00:00:23.24   |
| 9  | Aleix ESPARGARO  | Team SUZUKI ECSTAR    | 00:43:36.826 | 00:00:43.784  |
| 10 | Danilo PETRUCCI  | Octo Pramac Racing    | 00:43:38.303 | 00:00:45.261  |
| 11 | Yonny HERNANDEZ  | Octo Pramac Racing    | 00:43:43.015 | 00:00:49.973  |
| 12 | Scott REDDING    | EG 0,0 Marc VDS       | 00:43:43.216 | 00:00:50.174  |
| 13 | Alvaro BAUTISTA  | Aprilia Racing [...]  | 00:43:47.479 | 00:00:54.437  |
| 14 | Stefan BRADL     | Aprilia Racing [...]  | 00:43:47.666 | 00:00:54.624  |
| 15 | Loris BAZ        | Forward Racing        | 00:43:53.358 | 00:01:00.316  |
| 16 | Hector BARBERA   | Avintia Racing        | 00:43:54.637 | 00:01:01.595  |
| 17 | Nicky HAYDEN     | Aspar MotoGP Team     | 00:43:55.43  | 00:01:02.388  |
| 18 | Mike DI MEGLIO   | Avintia Racing        | 00:43:58.986 | 00:01:05.944  |
| 19 | Jack MILLER      | CWM LCR Honda         | 00:44:04.449 | 00:01:11.407  |
| 20 | Claudio CORTI    | Forward Racing        | 00:44:43.075 | 00:01:50.033  |
| 21 | Karel ABRAHAM    | AB Motoracing         | 00:44:55.697 | 00:02:02.655  |
| 22 | Maverick VIÑALES | Team SUZUKI ECSTAR    | 00:29:31.557 | -00:13:21.485 |
| 23 | Cal CRUTCHLOW    | CWM LCR Honda         | 00:27:38.315 | -00:15:14.727 |
| 24 | Eugene LAVERTY   | Aspar MotoGP Team     | 00:08:04.096 | -00:34:48.946 |
| 25 | Alex DE ANGELIS  | E-Motion Ioda[...]    | 00:06:05.782 | -00:36:47.26  |

(25 rows)

Pour chaque tour, afficher :

- le nom du pilote;
- son rang dans le tour;
- son temps depuis le début de la course;
- dans le tour, la différence de temps par rapport au premier.

Pour construire cette requête, nous avons besoin d'obtenir le temps cumulé tour après tour pour chaque coureur. Nous commençons donc par écrire une première requête :

```

SELECT *,
 SUM(lap_time)
 OVER (PARTITION BY no_pilote ORDER BY no_tour) AS temps_tour_glissant
FROM brno_2015

```

Elle retourne le résultat suivant :

| no_tour | no_pilote | lap_time     | temps_tour_glissant |
|---------|-----------|--------------|---------------------|
| 1       | 4         | 00:02:02.209 | 00:02:02.209        |
| 2       | 4         | 00:01:57.57  | 00:03:59.779        |
| 3       | 4         | 00:01:57.021 | 00:05:56.8          |
| 4       | 4         | 00:01:56.943 | 00:07:53.743        |
| 5       | 4         | 00:01:57.012 | 00:09:50.755        |
| 6       | 4         | 00:01:57.011 | 00:11:47.766        |
| 7       | 4         | 00:01:57.313 | 00:13:45.079        |
| 8       | 4         | 00:01:57.95  | 00:15:43.029        |
| 9       | 4         | 00:01:57.296 | 00:17:40.325        |
| 10      | 4         | 00:01:57.295 | 00:19:37.62         |
| 11      | 4         | 00:01:57.185 | 00:21:34.805        |
| 12      | 4         | 00:01:57.45  | 00:23:32.255        |
| 13      | 4         | 00:01:57.457 | 00:25:29.712        |
| 14      | 4         | 00:01:57.362 | 00:27:27.074        |
| 15      | 4         | 00:01:57.482 | 00:29:24.556        |
| 16      | 4         | 00:01:57.358 | 00:31:21.914        |
| 17      | 4         | 00:01:57.617 | 00:33:19.531        |
| 18      | 4         | 00:01:57.594 | 00:35:17.125        |
| 19      | 4         | 00:01:57.412 | 00:37:14.537        |
| 20      | 4         | 00:01:57.786 | 00:39:12.323        |
| 21      | 4         | 00:01:58.087 | 00:41:10.41         |
| 22      | 4         | 00:01:58.357 | 00:43:08.767        |

(...)

Cette requête de base est ensuite utilisée dans une CTE qui sera utilisée par la requête répondant à la question de départ. La colonne `temps_tour_glissant` est utilisée pour calculer le rang du pilote dans la course, est affiché et le temps cumulé du meilleur pilote est récupéré avec la fonction

`first_value` :

```
WITH temps_glissant AS (
 SELECT no_tour, no_pilote, lap_time,
 sum(lap_time)
 OVER (PARTITION BY no_pilote
 ORDER BY no_tour
) as temps_tour_glissant
 FROM brno_2015
 ORDER BY no_pilote, no_tour
)

SELECT no_tour, nom,
 rank() OVER (PARTITION BY no_tour
 ORDER BY temps_tour_glissant ASC
) as place_course,
 temps_tour_glissant,
 temps_tour_glissant - first_value(temps_tour_glissant)
 OVER (PARTITION BY no_tour
 ORDER BY temps_tour_glissant asc
) AS difference
FROM temps_glissant t
JOIN pilotes p ON p.no = t.no_pilote;
```

On pouvait également utiliser une simple sous-requête pour obtenir le même résultat :

```

SELECT no_tour,
 nom,
 rank()
 OVER (PARTITION BY no_tour
 ORDER BY temps_tour_glissant ASC
) AS place_course,
 temps_tour_glissant,
 temps_tour_glissant - first_value(temps_tour_glissant)
 OVER (PARTITION BY no_tour
 ORDER BY temps_tour_glissant asc
) AS difference
FROM (
 SELECT *, SUM(lap_time)
 OVER (PARTITION BY no_pilote
 ORDER BY no_tour)
 AS temps_tour_glissant
 FROM brno_2015) course
JOIN pilotes
 ON (pilotes.no = course.no_pilote)
ORDER BY no_tour;

```

La requête fournit le résultat suivant :

| no.        | nom              | place_c. | temps_tour_glissant | difference   |
|------------|------------------|----------|---------------------|--------------|
| 1          | Jorge LORENZO    | 1        | 00:02:00.83         | 00:00:00     |
| 1          | Marc MARQUEZ     | 2        | 00:02:01.058        | 00:00:00.228 |
| 1          | Andrea DOVIZIOSO | 3        | 00:02:02.209        | 00:00:01.379 |
| 1          | Valentino ROSSI  | 4        | 00:02:02.329        | 00:00:01.499 |
| 1          | Andrea IANNONE   | 5        | 00:02:02.597        | 00:00:01.767 |
| 1          | Bradley SMITH    | 6        | 00:02:02.861        | 00:00:02.031 |
| 1          | Pol ESPARGARO    | 7        | 00:02:03.239        | 00:00:02.409 |
| ( ... )    |                  |          |                     |              |
| 2          | Jorge LORENZO    | 1        | 00:03:57.073        | 00:00:00     |
| 2          | Marc MARQUEZ     | 2        | 00:03:57.509        | 00:00:00.436 |
| 2          | Valentino ROSSI  | 3        | 00:03:59.696        | 00:00:02.623 |
| 2          | Andrea DOVIZIOSO | 4        | 00:03:59.779        | 00:00:02.706 |
| 2          | Andrea IANNONE   | 5        | 00:03:59.9          | 00:00:02.827 |
| 2          | Bradley SMITH    | 6        | 00:04:00.355        | 00:00:03.282 |
| 2          | Pol ESPARGARO    | 7        | 00:04:00.87         | 00:00:03.797 |
| 2          | Maverick VIÑALES | 8        | 00:04:01.187        | 00:00:04.114 |
| (...)      |                  |          |                     |              |
| (498 rows) |                  |          |                     |              |

Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour?

Il est ici nécessaire de sélectionner pour chaque tour le temps du meilleur tour. On peut alors sélectionner les tours pour lesquels le temps du tour est égal au meilleur temps :

```

WITH temps_glissant AS (
 SELECT no_tour, no_pilote, lap_time,
 sum(lap_time)
 OVER (PARTITION BY no_pilote
 ORDER BY no_tour
) as temps_tour_glissant
 FROM brno_2015

```

```

ORDER BY no_pilote, no_tour
),
classement_tour AS (
 SELECT no_tour, no_pilote, lap_time,
 rank() OVER (
 PARTITION BY no_tour
 ORDER BY temps_tour_glissant
) as place_course,
 temps_tour_glissant,
 min(lap_time) OVER (PARTITION BY no_pilote) as meilleur_temps
 FROM temps_glissant
)

SELECT no_tour, nom, place_course, lap_time
FROM classement_tour t
JOIN pilotes p ON p.no = t.no_pilote
WHERE lap_time = meilleur_temps;

```

Ce qui donne le résultat suivant :

| no_tour | nom              | place_course | lap_time     |
|---------|------------------|--------------|--------------|
| 4       | Jorge LORENZO    | 1            | 00:01:56.169 |
| 4       | Marc MARQUEZ     | 2            | 00:01:56.048 |
| 4       | Valentino ROSSI  | 3            | 00:01:56.747 |
| 6       | Andrea IANNONE   | 5            | 00:01:56.86  |
| 6       | Dani PEDROSA     | 7            | 00:01:56.975 |
| 4       | Andrea DOVIZIOSO | 4            | 00:01:56.943 |
| 3       | Bradley SMITH    | 6            | 00:01:57.25  |
| 17      | Pol ESPARGARO    | 8            | 00:01:57.454 |
| 4       | Aleix ESPARGARO  | 12           | 00:01:57.844 |
| 4       | Daniilo PETRUCCI | 11           | 00:01:58.121 |
| 9       | Yonny HERNANDEZ  | 14           | 00:01:58.53  |
| 2       | Scott REDDING    | 14           | 00:01:57.976 |
| 3       | Alvaro BAUTISTA  | 21           | 00:01:58.71  |
| 3       | Stefan BRADL     | 16           | 00:01:58.38  |
| 3       | Loris BAZ        | 19           | 00:01:58.679 |
| 2       | Hector BARBERA   | 15           | 00:01:58.405 |
| 2       | Nicky HAYDEN     | 16           | 00:01:58.338 |
| 3       | Mike DI MEGLIO   | 18           | 00:01:58.943 |
| 4       | Jack MILLER      | 22           | 00:01:59.007 |
| 2       | Claudio CORTI    | 24           | 00:02:00.377 |
| 14      | Karel ABRAHAM    | 23           | 00:02:01.716 |
| 3       | Maverick VIÑALES | 8            | 00:01:57.436 |
| 3       | Cal CRUTCHLOW    | 11           | 00:01:57.652 |
| 3       | Eugene LAVERTY   | 20           | 00:01:58.977 |
| 3       | Alex DE ANGELIS  | 23           | 00:01:59.257 |

(25 rows)

Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.

```

WITH nb_tour AS (
 SELECT max(no_tour) FROM brno_2015
),

```

```

temps_glissant AS (
 SELECT no_tour, no_pilote, lap_time,
 sum(lap_time) OVER (
 PARTITION BY no_pilote
 ORDER BY no_tour
) as temps_tour_glissant,
 max(no_tour) OVER (PARTITION BY no_pilote) as total_tour
 FROM brno_2015
),
classement_tour AS (
 SELECT no_tour, no_pilote, lap_time, total_tour,
 rank() OVER (
 PARTITION BY no_tour
 ORDER BY temps_tour_glissant
) as place_course
 FROM temps_glissant
)
SELECT no_pilote
FROM classement_tour t
JOIN nb_tour n ON n.max = t.total_tour
GROUP BY no_pilote
HAVING count(DISTINCT place_course) = 1;

```

Elle retourne le résultat suivant :

```

no_pilote

 93
 99

```

En quelle position a terminé le coureur qui a doublé le plus de personnes? Combien de personnes a-t-il doublées?

```

WITH temps_glissant AS (
 SELECT no_tour, no_pilote, lap_time,
 sum(lap_time) OVER (
 PARTITION BY no_pilote
 ORDER BY no_tour
) as temps_tour_glissant
 FROM brno_2015
),
classement_tour AS (
 SELECT no_tour, no_pilote, lap_time,
 rank() OVER (
 PARTITION BY no_tour
 ORDER BY temps_tour_glissant
) as place_course,
 temps_tour_glissant
 FROM temps_glissant
),
depassement AS (
 SELECT no_pilote,
 last_value(place_course) OVER (PARTITION BY no_pilote) as rang,
 CASE
 WHEN lag(place_course) OVER (
 PARTITION BY no_pilote
 ORDER BY no_tour

```

```

) - place_course < 0
 THEN 0
 ELSE lag(place_course) OVER (
 PARTITION BY no_pilote
 ORDER BY no_tour
) - place_course
 END AS depasse
FROM classement_tour t
)

SELECT no_pilote, rang, sum(depasse)
FROM depassement
GROUP BY no_pilote, rang
ORDER BY sum(depasse) DESC
LIMIT 1;

```

### Grouping Sets

La suite de ce TP est maintenant réalisé avec la base de formation habituelle. Attention, ce TP nécessite l'emploi d'une version 9.5 ou supérieure de PostgreSQL.

Tout d'abord, nous positionnons le search\_path pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.

```

SELECT extract('year' from date_commande) AS annee, code_pays,
 SUM(quantite*prix_unitaire) AS montant_total_commande
FROM commandes c
JOIN lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN clients
 ON (c.client_id = clients.client_id)
JOIN contacts co
 ON (clients.contact_id = co.contact_id)
GROUP BY GROUPING SETS (
 (extract('year' from date_commande), code_pays),
 (extract('year' from date_commande))
);

```

Le résultat attendu est :

| annee | code_pays | montant_total_commande |
|-------|-----------|------------------------|
| 2003  | DE        | 49634.24               |
| 2003  | FR        | 10003.98               |
| 2003  |           | 59638.22               |
| 2008  | CA        | 1016082.18             |
| 2008  | CN        | 801662.75              |
| 2008  | DE        | 694787.87              |
| 2008  | DZ        | 663045.33              |
| 2008  | FR        | 5860607.27             |
| 2008  | IN        | 741850.87              |
| 2008  | PE        | 1167825.32             |

|       |  |    |  |             |
|-------|--|----|--|-------------|
| 2008  |  | RU |  | 577164.50   |
| 2008  |  | US |  | 928661.06   |
| 2008  |  |    |  | 12451687.15 |
| (...) |  |    |  |             |

Ajouter également le montant total des commandes depuis le début de l'activité.

L'opérateur de regroupement `ROLL UP` amène le niveau d'agrégation sans regroupement :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
 SUM(quantite*prix_unitaire) AS montant_total_commande
FROM commandes c
JOIN lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN clients
 ON (c.client_id = clients.client_id)
JOIN contacts co
 ON (clients.contact_id = co.contact_id)
GROUP BY ROLLUP (extract('year' from date_commande), code_pays);
```

Ajouter également le montant total des commandes par pays.

Cette fois, l'opérateur `CUBE` permet d'obtenir l'ensemble de ces informations :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
 SUM(quantite*prix_unitaire) AS montant_total_commande
FROM commandes c
JOIN lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN clients
 ON (c.client_id = clients.client_id)
JOIN contacts co
 ON (clients.contact_id = co.contact_id)
GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

À partir de la requête précédente, ajouter une colonne par critère de regroupement, de type booléen, qui est positionnée à `true` lorsque le regroupement est réalisé sur l'ensemble des valeurs de la colonne.

Ces colonnes booléennes permettent d'indiquer à l'application comment gérer la présentation des résultats.

```
SELECT grouping(extract('year' from date_commande))::boolean AS g_annee,
 grouping(code_pays)::boolean AS g_pays,
 extract('year' from date_commande) AS annee,
 code_pays,
 SUM(quantite*prix_unitaire) AS montant_total_commande
FROM commandes c
JOIN lignes_commandes l
 ON (c.numero_commande = l.numero_commande)
JOIN clients
 ON (c.client_id = clients.client_id)
```

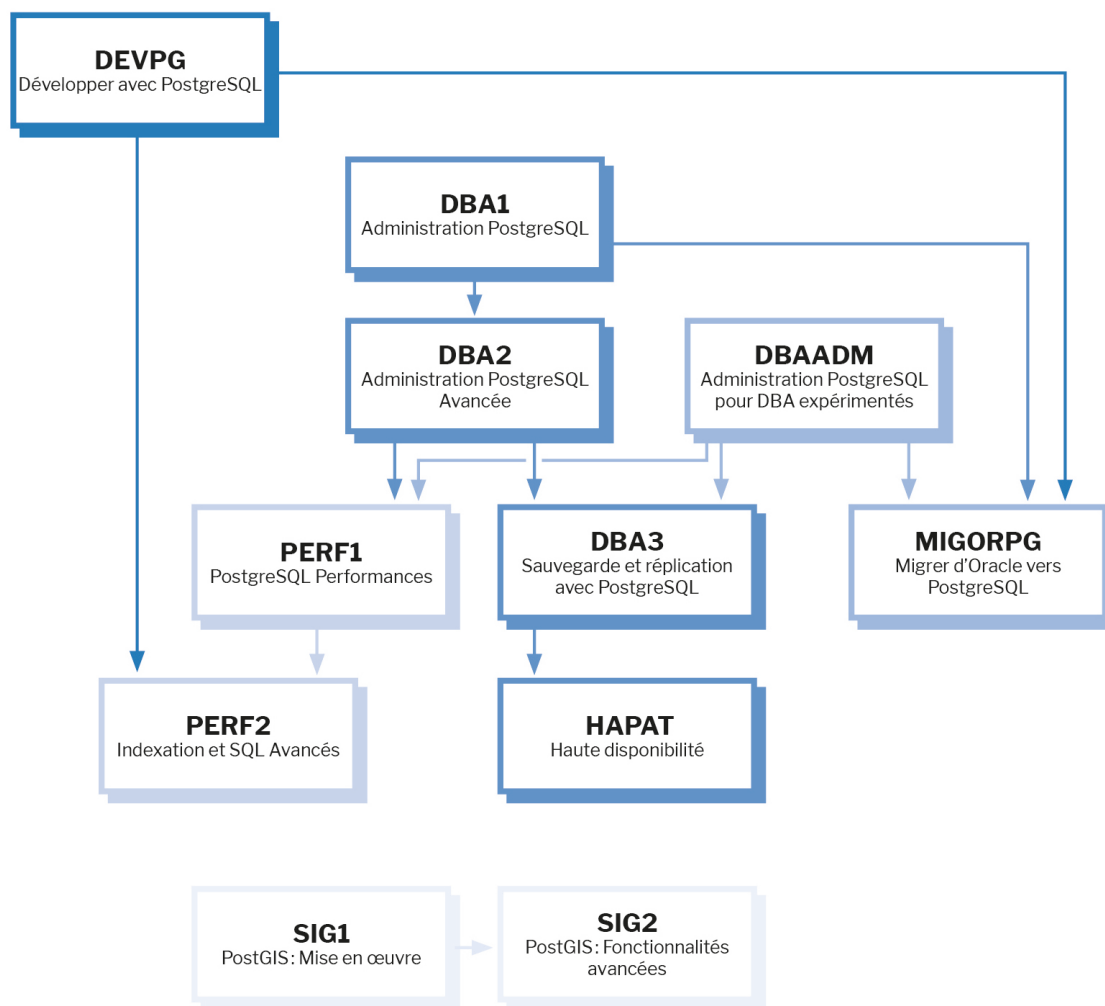
```
JOIN contacts co
 ON (clients.contact_id = co.contact_id)
GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

# Les formations Dalibo

Retrouvez nos formations et le calendrier sur <https://dali.bo/formation>

Pour toute information ou question, n'hésitez pas à nous écrire sur [contact@dalibo.com](mailto:contact@dalibo.com).

## Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL  
<https://dali.bo/dba1>
- DBA2 : Administration PostgreSQL avancé  
<https://dali.bo/dba2>
- DBA3 : Sauvegarde et réplication avec PostgreSQL  
<https://dali.bo/dba3>
- DEV1 : Introduction à SQL  
<https://dali.bo/dev1>
- DEVPG : Développer avec PostgreSQL  
<https://dali.bo/devpg>
- PERF1 : PostgreSQL Performances  
<https://dali.bo/perf1>
- PERF2 : Indexation et SQL avancés  
<https://dali.bo/perf2>
- MIGORPG : Migrer d'Oracle à PostgreSQL  
<https://dali.bo/migorpg>
- HAPAT : Haute disponibilité avec PostgreSQL  
<https://dali.bo/hapat>

### Les livres blancs

- Migrer d'Oracle à PostgreSQL  
<https://dali.bo/dlb01>
- Industrialiser PostgreSQL  
<https://dali.bo/dlb02>
- Bonnes pratiques de modélisation avec PostgreSQL  
<https://dali.bo/dlb04>
- Bonnes pratiques de développement avec PostgreSQL  
<https://dali.bo/dlb05>

### Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.



