Module S7

Analyse de données avec SQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Analyse de données avec SQL

Module S7

TITRE : Analyse de données avec SQL
SOUS-TITRE : Module S7

REVISION: 22.09
DATE: 02 septembre 2022
COPYRIGHT: © 2005-2022 DALIBO SARL SCOP
LICENCE: Creative Commons BY-NC-SA

Postgres®, PostgreSQL® and the Slonik Logo are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission.
(Les noms PostgreSQL® et Postgres®, et le logo Slonik sont des marques déposées par PostgreSQL Community Association of Canada.
Voir https://www.postgresql.org/about/policies/trademarks/)

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

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

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

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

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

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

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

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

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

Table des Matières

1 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

1.2 AGRÉGATS

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

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

La plupart des exemples utilisent une table `employes` définie telle que :

```
CREATE TABLE employes (
  matricule char(8) primary key,
  nom       text    not null,
  service   text,
```

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

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

SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

Ainsi, on peut déduire le salaire moyen avec la fonction `avg()`, les salaires maximum et minimum versés par la société avec les fonctions `max()` et `min()`, ainsi que la somme totale des salaires versés avec la fonction `sum()` :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
  FROM employes;
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 4900.0000000000000000 |        10000.00 |         3000.00 |       24500.00
```

La base de données réalise les calculs sur l'ensemble des données de la table et n'affiche que le résultat du calcul.

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

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
```

```
       sum(salaire) AS somme_salaires
  FROM employes
 WHERE service = 'Courrier';
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 3750.0000000000000000 |         4500.00 |         3000.00 |        7500.00
(1 ligne)
```

En revanche, il n'est pas possible de référencer d'autres colonnes pour les afficher à côté du résultat d'un calcul d'agrégation à moins de les utiliser comme critère de regroupement :

```
SELECT avg(salaire), nom FROM employes;
ERROR:  column "employes.nom" must appear in the GROUP BY clause or be used in
        an aggregate function
LIGNE 1 : SELECT avg(salaire), nom FROM employes;
                               ^
```

1.2.1 AGRÉGATS AVEC GROUP BY

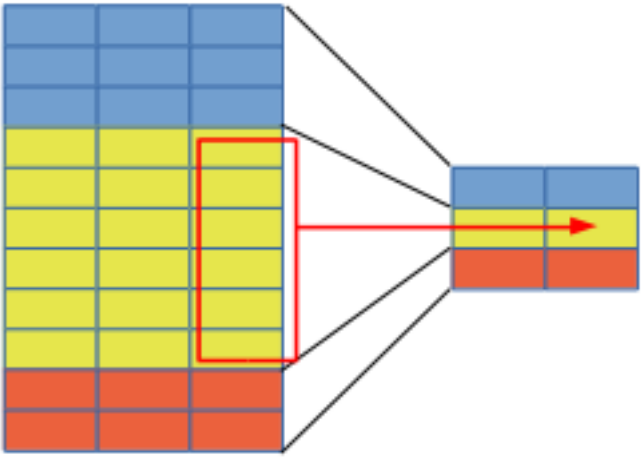
- agrégat + `GROUP BY`
- Utilité
  - effectue des calculs sur des regroupements : moyenne, somme, comptage, etc.
  - regroupement selon un critère défini par la clause `GROUP BY`
  - exemple : calcul du salaire moyen de chaque service

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.

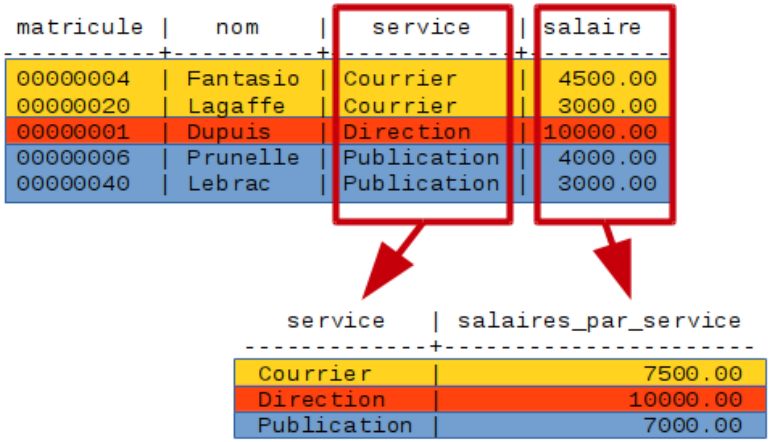
matricule	nom	service	salaire
00000004	Fantasio	Courrier	4500.00
00000020	Lagaffe	Courrier	3000.00
00000001	Dupuis	Direction	10000.00
00000006	Prunelle	Publication	4000.00
00000040	Lebrac	Publication	3000.00

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 regroupe-

ment puis les valeurs calculées par les fonctions d'agrégation :



1.2.2 GROUP BY : PRINCIPE



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

1.2.3 GROUP BY : EXEMPLES

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

   service   | salaires_par_service
-------------+----------------------
 Courrier    |              7500.00
 Direction   |             10000.00
 Publication |              7000.00
(3 lignes)
```

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

On peut bien entendu combiner plusieurs opérations d'agrégations :

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

   service   | salaires_par_service | salaire_moyen_service
-------------+----------------------+-----------------------
 Courrier    |              7500.00 |  3750.0000000000000000
 Direction   |             10000.00 | 10000.0000000000000000
 Publication |              7000.00 |  3500.0000000000000000
(3 lignes)
```

On peut combiner le résultat de deux requêtes d'agrégation avec `UNION ALL`, si les ensembles retournées sont de même type :

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

```
   service    | salaires_par_service
--------------+----------------------
 Courrier     |              7500.00
 Direction    |             10000.00
 Publication  |              7000.00
 Total        |             24500.00
(4 lignes)
```

On le verra plus loin, cette dernière requête peut être écrite plus simplement avec les `GROUPING SETS`, mais qui nécessitent au minimum PostgreSQL 9.5.

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)
```

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

```
SELECT service,
       array_agg(nom ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |   liste_employes
--------------+--------------------
 Courrier     | {Fantasio,Lagaffe}
 Direction    | {Dupuis}
 Publication | {Lebrac,Prunelle}
```

1.3 CLAUSE FILTER

- Clause `FILTER`
- Utilité :
  - filtrer les données sur les agrégats
  - évite les expressions `CASE` complexes
- SQL:2003
- Intégré dans la version 9.4

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

1.3.1 FILTRER AVEC CASE

- La syntaxe suivante était utilisée :

```
SELECT count(*) AS compte_pays,
       count(CASE WHEN r.nom_region='Europe' THEN 1
                  ELSE NULL
             END) AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

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

1.3.2 FILTRER AVEC FILTER

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

```
SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe')
                AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

L'exemple suivant montre l'utilisation de la clause `FILTER` et son équivalent écrit avec une expression `CASE` :

```
sql=# SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe') AS compte_pays_europeens,
       count(CASE WHEN r.nom_region='Europe' THEN 1 END)
           AS oldschool_compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
 compte_pays | compte_pays_europeens | oldschool_compte_pays_europeens
-------------+-----------------------+---------------------------------
          25 |                     5 |                               5
(1 ligne)
```

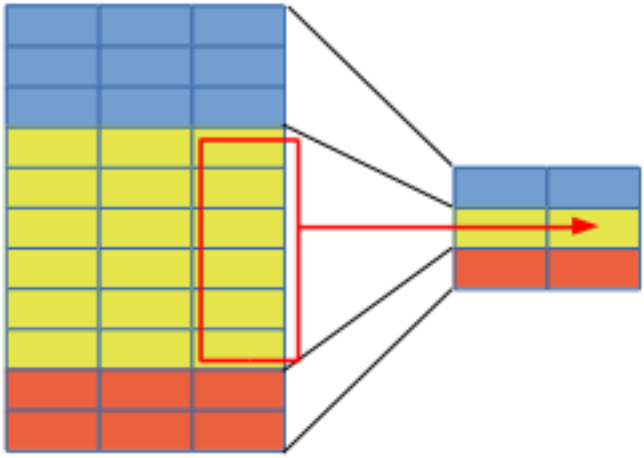
1.4 FONCTIONS DE FENÊTRAGE

- Fonctions *window*
  - travaille sur des ensembles de données regroupés et triés indépendamment de la requête principale
- Utilisation :
  - utiliser plusieurs critères d'agrégation dans la même requête
  - utiliser des fonctions de classement
  - faire référence à d'autres lignes de l'ensemble de données

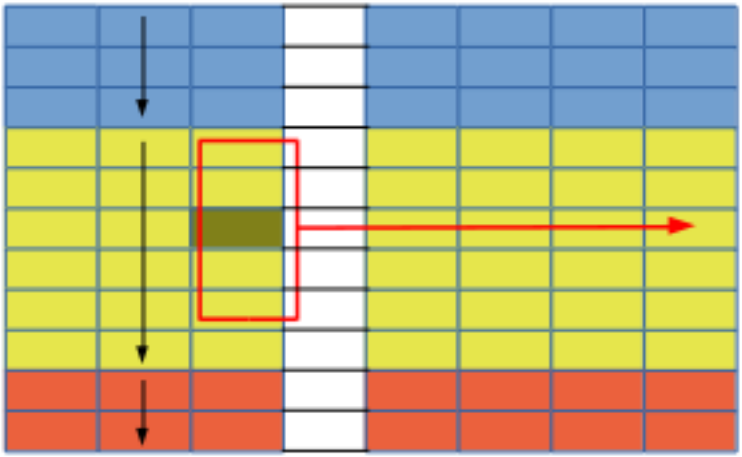
PostgreSQL supporte les fonctions de fenêtrage depuis la version 8.4. Elles apportent des fonctionnalités analytiques à PostgreSQL, et permettent d'écrire beaucoup plus simplement certaines requêtes.

Prenons un exemple.

```
SELECT service, AVG(salaire)
FROM employe
GROUP BY service
```



```
SELECT service, id_employe, salaire,
  AVG(salaire) OVER (
    PARTITION BY service
    ORDER BY age
    ROWS BETWEEN 2 PRECEEDING AND 2 FOLLOWING
  )
FROM employes
```



1.4.1 REGROUPEMENT

- Regroupement
  - clause `OVER (PARTITION BY ...)`
- Utilité :
  - plusieurs critères de regroupement différents
  - avec des fonctions de calcul d'agrégats

La clause `OVER` permet de définir la façon dont les données sont regroupées - uniquement pour la colonne définie - avec la clause `PARTITION BY`.

Les exemples vont utiliser cette table `employes` :

```
exemple=# SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

1.4.2 REGROUPEMENT : EXEMPLE

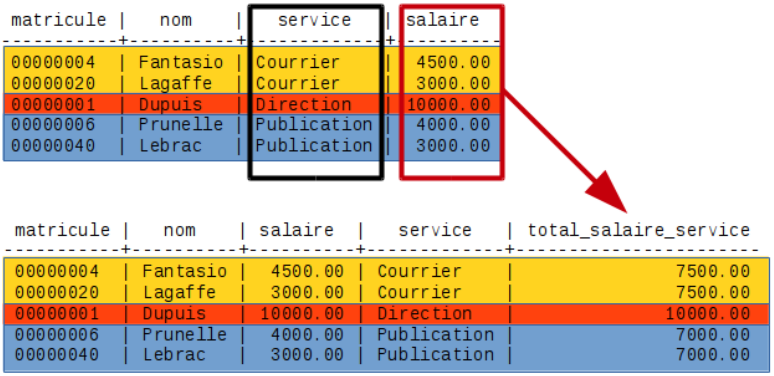
```
SELECT matricule, salaire, service,
       SUM(salaire) OVER (PARTITION BY service)
                AS total_salaire_service
  FROM employes;
```

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

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

1.4.3 REGROUPEMENT : PRINCIPE

```
SUM(salaire) OVER (PARTITION BY service)
```



Entouré de noir, le critère de regroupement et entouré de rouge, les données sur lesquelles sont appliqués le calcul d'agrégat.

1.4.4 REGROUPEMENT : SYNTAXE

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

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

1.4.5 TRI

- Tri
  - `OVER (ORDER BY …)`
- Utilité :
  - numéroter les lignes : `row_number()`
  - classer des résultats : `rank()`, `dense_rank()`
  - faire appel à d'autres lignes du résultat : `lead()`, `lag()`

1.4.6 TRI : EXEMPLE

- Pour numéroter des lignes :

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

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

La fonction `row_number()` permet de numéroter les lignes selon un critère de tri défini dans la clause `OVER`.

L'ordre de tri de la clause `OVER` n'influence pas l'ordre de tri explicite d'une requête :

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

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

On dispose aussi de fonctions de classement, pour déterminer par exemple les employés les moins bien payés :

```
SELECT matricule, nom, salaire, service,
       rank() OVER (ORDER BY salaire),
       dense_rank() OVER (ORDER BY salaire)
  FROM employes ;
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

La fonction de fenêtrage `rank()` renvoie le classement en autorisant des trous dans la numérotation, et `dense_rank()` le classement sans trous.

1.4.7 TRI : EXEMPLE AVEC UNE SOMME

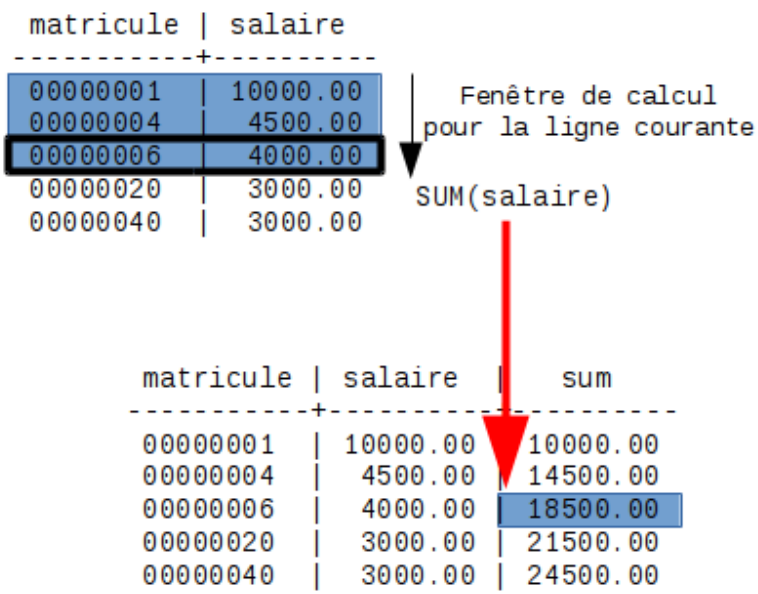
- Calcul d'une somme glissante :

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

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

1.4.8 TRI : PRINCIPE

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



Lorsque l'on utilise une clause de tri, la portion de données visible par l'opérateur d'agrégat correspond aux données comprises entre la première ligne examinée et la ligne courante. La fenêtre est définie selon le critère `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Nous verrons plus loin que nous pouvons modifier ce comportement.

1.4.9 TRI : SYNTAXE

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

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

1.4.10 REGROUPEMENT ET TRI

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

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

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.

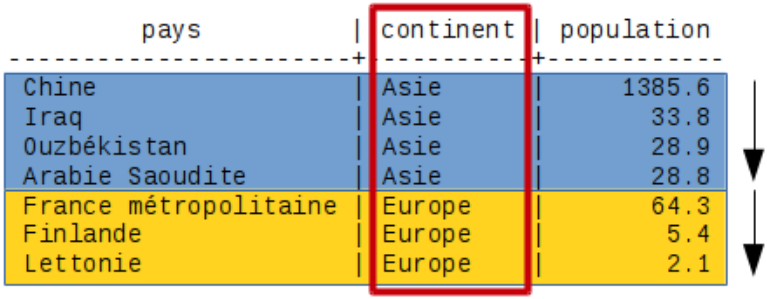
Voici un extrait plus complet du résultat de la requête présentée ci-dessus :

```
         continent         |         pays        | population | rang_pop
---------------------------+---------------------+------------+-----------
 Afrique                   | Nigéria             |      173.6 |        1
 Afrique                   | Éthiopie            |       94.1 |        2
 Afrique                   | Égypte              |       82.1 |        3
 Afrique                   | Rép. dém. du Congo  |       67.5 |        4
 Afrique                   | Afrique du Sud      |       52.8 |        5
```

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

1.4.12 REGROUPEMENT ET TRI : PRINCIPE

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



1.4.13 REGROUPEMENT ET TRI : SYNTAXE

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

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

1.4.14 FONCTIONS ANALYTIQUES

- PostgreSQL dispose d'un certain nombre de fonctions analytiques
- Utilité :
  - faire référence à d'autres lignes du même ensemble
  - évite les auto-jointures complexes et lentes

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

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

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
 matricule |   nom    | salaire  |   service   | salaire_maximum_service
-----------+----------+----------+-------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier    |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier    |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction   |                10000.00
 00000006  | Prunelle |  4000.00 | Publication |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication |                 4000.00
(5 lignes)
```

Il existe également les fonctions suivantes :

- `last_value(colonne)` : renvoie la dernière valeur pour la colonne ;
- `nth(colonne, n)` : renvoie la n-ème valeur (en comptant à partir de **1**) pour la colonne ;

- `lag(colonne, n)` : renvoie la valeur située en n-ème position **avant** la ligne en cours pour la colonne ;
- `lead(colonne, n)` : renvoie la valeur située en n-ème position **après** la ligne en cours pour la colonne ;
  - pour ces deux fonctions, le n est facultatif et vaut **1** par défaut ;
  - ces deux fonctions acceptent un 3ème argument facultatif spécifiant la valeur à renvoyer si aucune valeur n'est trouvée en n-ème position avant ou après. Par défaut, `NULL` sera renvoyé.

1.4.15 LEAD() ET LAG()

- `lead(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **après** la ligne courante
- `lag(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **avant** la ligne courante

La construction `lead(colonne)` est équivalente à `lead(colonne, 1)`. De même, la construction `lag(colonne)` est équivalente à `lag(colonne, 1)`. Il s'agit d'un raccourci pour utiliser la valeur précédente ou la valeur suivante d'une colonne dans la fenêtre définie.

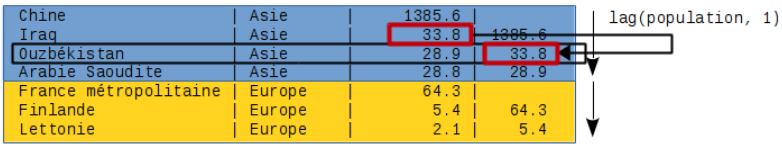
1.4.16 LEAD() ET LAG() : EXEMPLE

```
SELECT pays, continent, population,
       lag(population) OVER (PARTITION BY continent
                             ORDER BY population DESC)
  FROM population;
        pays          | continent | population |  lag
----------------------+-----------+------------+--------
 Chine                | Asie      |     1385.6 |
 Iraq                 | Asie      |       33.8 | 1385.6
 Ouzbékistan          | Asie      |       28.9 |   33.8
 Arabie Saoudite      | Asie      |       28.8 |   28.9
 France métropolitaine | Europe    |       64.3 |
 Finlande             | Europe    |        5.4 |   64.3
 Lettonie             | Europe    |        2.1 |    5.4
```

La requête présentée en exemple ne s'appuie que sur un jeu réduit de données afin de montrer un résultat compréhensible.

1.4.17 LEAD() ET LAG() : PRINCIPE

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



`NULL` est renvoyé lorsque la valeur n'est pas accessible dans la fenêtre de données, comme par exemple si l'on souhaite utiliser la valeur d'une colonne appartenant à la ligne précédant la première ligne de la partition.

1.4.18 FIRST/LAST/NTH_VALUE

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

Utilisé avec `ORDER BY` et `PARTITION BY`, la fonction `first_value()` permet par exemple d'obtenir le salaire le plus élevé d'un service :

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

```
 matricule |   nom    | salaire  |   service   | salaire_maximum_service
-----------+----------+----------+-------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier    |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier    |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction   |                10000.00
 00000006  | Prunelle |  4000.00 | Publication |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication |                 4000.00
(5 lignes)
```

1.4.19 FIRST/LAST/NTH_VALUE : EXEMPLE

```
SELECT pays, continent, population,
       first_value(population)
           OVER (PARTITION BY continent
                 ORDER BY population DESC)
  FROM population;

       pays       | continent | population | first_value
------------------+-----------+------------+-------------
 Chine            | Asie      |     1385.6 |      1385.6
 Iraq             | Asie      |       33.8 |      1385.6
 Ouzbékistan      | Asie      |       28.9 |      1385.6
 Arabie Saoudite  | Asie      |       28.8 |      1385.6
 France           | Europe    |       64.3 |        64.3
 Finlande         | Europe    |        5.4 |        64.3
 Lettonie         | Europe    |        2.1 |        64.3
```

Lorsque que la clause `ORDER BY` est utilisée pour définir une fenêtre, la fenêtre visible depuis la ligne courante commence par défaut à la première ligne de résultat et s'arrête à la ligne courante.

Par exemple, si l'on exécute la même requête en utilisant `last_value()` plutôt que `first_value()`, on récupère à chaque fois la valeur de la colonne sur la ligne courante :

```
SELECT pays, continent, population,
       last_value(population) OVER (PARTITION BY continent
                                    ORDER BY population DESC)
  FROM population;

         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Chine                 | Asie      |     1385.6 |     1385.6
 Iraq                  | Asie      |       33.8 |       33.8
 Ouzbékistan           | Asie      |       28.9 |       28.9
 Arabie Saoudite       | Asie      |       28.8 |       28.8
 France métropolitaine | Europe    |       64.3 |       64.3
 Finlande              | Europe    |        5.4 |        5.4
 Lettonie              | Europe    |        2.1 |        2.1
(7 rows)
```

Il est alors nécessaire de redéfinir le comportement de la fenêtre visible pour que la fonction se comporte comme attendu, en utilisant `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` - cet aspect sera décrit dans la section sur les possibilités de modification de la définition de la fenêtre.

1.4.20 CLAUSE WINDOW

- Pour factoriser la définition d'une fenêtre :

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

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

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
  WINDOW w AS (ORDER BY salaire);
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

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

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

1.4.21 CLAUSE WINDOW : SYNTAXE

```
SELECT fonction_agregat OVER nom,
       fonction_agregat_2 OVER nom ...
       ...
  FROM <liste_tables>
 WHERE <predicats>
 WINDOW nom AS (PARTITION BY ... ORDER BY ...)
```

1.4.22 DÉFINITION DE LA FENÊTRE

- La fenêtre de travail par défaut est :

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

- Trois modes possibles :
  - `RANGE`
  - `ROWS`
  - `GROUPS` (v11+)
- Nécessite une clause `ORDER BY`

1.4.23 DÉFINITION DE LA FENÊTRE : RANGE

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

```
OVER (PARTITION BY ...
      ORDER BY ...
      RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

1.4.24 DÉFINITION DE LA FENÊTRE : ROWS

- Indique un intervalle borné par un nombre de ligne défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx valeurs devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx valeurs derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
      ORDER BY ...
      ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

1.4.25 DÉFINITION DE LA FENÊTRE : GROUPS

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

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

Ceci n'est disponible que depuis la version 11.

1.4.26 DÉFINITION DE LA FENÊTRE : EXCLUDE

- Indique des lignes à exclure de la fenêtre de données (v11+)
- `EXCLUDE CURRENT ROW` : exclut la ligne courante
- `EXCLUDE GROUP` : exclut la ligne courante et le groupe de valeurs identiques dans l'ordre
- `EXCLUDE TIES` exclut et le groupe de valeurs identiques à la ligne courante dans l'ordre mais pas la ligne courante
- `EXCLUDE NO OTHERS` : pas d'exclusion (valeur par défaut)

Ceci n'est disponible que depuis la version 11.

1.4.27 DÉFINITION DE LA FENÊTRE : EXEMPLE

```
SELECT pays, continent, population,
       last_value(population)
        OVER (PARTITION BY continent ORDER BY population
              RANGE BETWEEN UNBOUNDED PRECEDING
                        AND UNBOUNDED FOLLOWING)
  FROM population;

         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Arabie Saoudite       | Asie      |       28.8 |     1385.6
 Ouzbékistan           | Asie      |       28.9 |     1385.6
 Iraq                  | Asie      |       33.8 |     1385.6
 Chine (4)             | Asie      |     1385.6 |     1385.6
 Lettonie              | Europe    |        2.1 |       64.3
 Finlande              | Europe    |        5.4 |       64.3
 France métropolitaine | Europe    |       64.3 |       64.3
```

1.5 WITHIN GROUP

- `WITHIN GROUP`
  - PostgreSQL 9.4
- Utilité :
  - calcul de médianes, centiles

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

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.

Pour rappel, la table contient les données suivantes :

```
postgres=# SELECT * FROM population ORDER BY continent, population;
         pays          | population | superficie | densite | continent
-----------------------+------------+------------+---------+-----------
 Tunisie               |       11.0 |        164 |      67 | Afrique
 Zimbabwe              |       14.1 |        391 |      36 | Afrique
 Soudan                |       14.1 |        197 |      72 | Afrique
 Burkina Faso          |       16.9 |        274 |      62 | Afrique
(...)
```

En ajoutant le support de cette clause, PostgreSQL améliore son support de la norme SQL 2008 et permet le développement d'analyses statistiques plus élaborées.

1.6 GROUPING SETS

- `GROUPING SETS/ROLLUP/CUBE`
- Extension de `GROUP BY`
- PostgreSQL 9.5
- Utilité :
  - présente le résultat de plusieurs agrégations différentes
  - réaliser plusieurs agrégations différentes dans la même requête

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

1.6.1 GROUPING SETS : JEU DE DONNÉES

stock		
piece	region	quantite
ecrous	est	50
ecrous	ouest	0
ecrous	sud	40
clous	est	70
clous	nord	40
vis	ouest	50
vis	sud	50
vis	nord	60

stock				
piece/region	est	ouest	sud	nord
ecrous	50	0	40	
clous	70			0
vis		50	50	60

```
CREATE TABLE stock AS SELECT * FROM (
    VALUES ('ecrous', 'est',   50),
           ('ecrous', 'ouest',  0),
           ('ecrous', 'sud',   40),
           ('clous',  'est',   70),
           ('clous',  'nord',   0),
           ('vis',    'ouest', 50),
           ('vis',    'sud',   50),
           ('vis',    'nord',  60)
) AS VALUES(piece, region, quantite);
```

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	

1.6.3 GROUPING SETS : EXEMPLE ORDRE SQL

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
 piece  | region | sum
--------+--------+-----
 clous  |        |  70
 ecrous |        |  90
 vis    |        | 160
        | est    | 120
        | nord   |  60
        | ouest  |  50
        | sud    |  90
```

1.6.4 GROUPING SETS : ÉQUIVALENT

- On peut se passer de la clause `GROUPING SETS`
  - mais la requête sera plus lente

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

```
    GROUP BY region;
```

Le comportement de la clause `GROUPING SETS` peut être émulée avec deux requêtes utilisant chacune une clause `GROUP BY` sur les colonnes de regroupement souhaitées.

Cependant, le plan d'exécution de la requête équivalente conduit à deux lectures et peut être particulièrement coûteux si le jeu de données sur lequel on souhaite réaliser les agrégations est important :

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

```
                            QUERY PLAN
------------------------------------------------------------------------
 Append  (cost=1.12..2.38 rows=7 width=44)
   ->  HashAggregate  (cost=1.12..1.15 rows=3 width=45)
         Group Key: stock.piece
         ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=9)
   ->  HashAggregate  (cost=1.12..1.16 rows=4 width=44)
         Group Key: stock_1.region
         ->  Seq Scan on stock stock_1  (cost=0.00..1.08 rows=8 width=8)
```

La requête utilisant la clause `GROUPING SETS` propose un plan bien plus efficace :

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

```
                          QUERY PLAN
---------------------------------------------------------------
 GroupAggregate  (cost=1.20..1.58 rows=14 width=17)
   Group Key: piece
   Sort Key: region
     Group Key: region
   ->  Sort  (cost=1.20..1.22 rows=8 width=13)
         Sort Key: piece
         ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=13)
```

1.6.5 ROLLUP

- `ROLLUP`
- PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête

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

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

1.6.7 ROLLUP : EXEMPLE ORDRE SQL

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

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

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

Sur une requête un peu plus intéressante, effectuant des statistiques sur des ventes :

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

```
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           |           |  111557177.00
(...)
 P           | RU        |  287605812.99
 P           | US        |  296424154.49
 P           |           | 4692152751.08
             |           | 5217862160.65
```

Une fonction `GROUPING`, associée à ROLLUP, permet de déterminer si la ligne courante correspond à un regroupement donné. Elle est de la forme d'un masque de bit converti au format décimal :

```
SELECT row_number()
         OVER ( ORDER BY grouping(piece,region)) AS ligne,
       grouping(piece,region)::bit(2) AS g,
       piece,
       region,
       sum(quantite)
FROM stock
GROUP BY CUBE (piece,region)
ORDER BY g ;
```

```
 ligne | g  | piece  | region | sum
-------+----+--------+--------+-----
     1 | 00 | clous  | est    | 150
     2 | 00 | clous  | nord   |  10
     3 | 00 | ecrous | est    | 110
     4 | 00 | ecrous | ouest  |  10
     5 | 00 | ecrous | sud    |  90
```

```
     6 | 00 | vis    | nord   | 130
     7 | 00 | vis    | ouest  | 110
     8 | 00 | vis    | sud    | 110
     9 | 01 | vis    |        | 350
    10 | 01 | ecrous |        | 210
    11 | 01 | clous  |        | 160
    12 | 10 |        | ouest  | 120
    13 | 10 |        | sud    | 200
    14 | 10 |        | est    | 260
    15 | 10 |        | nord   | 140
    16 | 11 |        |        | 720
```

Voici un autre exemple :

```
SELECT COALESCE(service,
    CASE
        WHEN GROUPING(service) = 0 THEN 'Unknown' ELSE 'Total'
    END) AS service,
       sum(salaire) AS salaires_service, count(*) AS nb_employes
  FROM employes
 GROUP BY ROLLUP (service);
  service    | salaires_service | nb_employes
-------------+------------------+-------------
 Courrier    |          7500.00 |           2
 Direction   |         50000.00 |           1
 Publication |          7000.00 |           2
 Total       |         64500.00 |           5
(4 rows)
```

Ou appliqué à l'exemple un peu plus complexe :

```
SELECT COALESCE(type_client,
       CASE
         WHEN GROUPING(type_client) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS type_client,
       COALESCE(code_pays,
       CASE
         WHEN GROUPING(code_pays) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
```

```
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           | Total     |  111557177.00
(...)
 P           | US        |  296424154.49
 P           | Total     | 4692152751.08
 Total       | Total     | 5217862160.65
```

1.6.8 CUBE

- `CUBE`
  - PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête
  - sur toutes les clauses de regroupement

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

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

1.6.10 CUBE : EXEMPLE ORDRE SQL

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

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

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

Elle permet de réaliser des regroupements sur l'ensemble des combinaisons possibles des clauses de regroupement indiquées. Pour de plus amples détails, se référer à cet article Wikipédia[2] .

En reprenant la requête de l'exemple précédent :

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

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

```
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle retournera le résultat suivant :

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

```
            | US          |  327821140.35
```

Dans ce genre de contexte, lorsque le regroupement est réalisé sur l'ensemble des valeurs d'un critère de regroupement, alors la valeur qui apparaît est `NULL` pour la colonne correspondante. Si la colonne possède des valeurs `NULL` légitimes, il est alors difficile de les distinguer. On utilise alors la fonction `GROUPING()` qui permet de déterminer si le regroupement porte sur l'ensemble des valeurs de la colonne. L'exemple suivant montre une requête qui exploite cette fonction :

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

Elle produit le résultat suivant :

```
 grouping | g_type_cli | g_code_pays | type_client | code_pays |    montant
----------+------------+-------------+-------------+-----------+---------------
 00       | f          | f           | A           | CA        |    6273168.32
 00       | f          | f           | A           | CN        |    7928641.50
 00       | f          | f           | A           | DE        |    6642061.57
 00       | f          | f           | A           | DZ        |    6404425.16
 00       | f          | f           | A           | FR        |   55261295.52
 00       | f          | f           | A           | IN        |    7224008.95
 00       | f          | f           | A           | PE        |    7356239.93
 00       | f          | f           | A           | RU        |    6766644.98
 00       | f          | f           | A           | US        |    7700691.07
 01       | f          | t           | A           |           |  111557177.00
(...)
 01       | f          | t           | P           |           | 4692152751.08
 11       | t          | t           |             |           | 5217862160.65
 10       | t          | f           |             | CA        |  327706809.65
 10       | t          | f           |             | CN        |  321261454.05
 10       | t          | f           |             | DE        |  319488602.46
 10       | t          | f           |             | DZ        |  333727307.87
 10       | t          | f           |             | FR        | 2606641183.25
```

```
 10       | t          | f          |            | IN        |  329268913.95
 10       | t          | f          |            | PE        |  332177174.71
 10       | t          | f          |            | RU        |  319769574.36
 10       | t          | f          |            | US        |  327821140.35
(40 rows)
```

L'application sera alors à même de gérer la présentation des résultats en fonction des valeurs de `grouping` ou `g_type_client` et `g_code_pays`.

1.7 TRAVAUX PRATIQUES

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

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

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

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

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

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

Agrégation

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

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

Window Functions

3. Afficher la place sur le podium pour chaque coureur.

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

5. Pour chaque tour, afficher :

- le nom du pilote ;
- son rang dans le tour ;
- son temps depuis le début de la course ;

- dans le tour, la différence de temps par rapport au premier.

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?
7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.
8. En quelle position a terminé le coureur qui a doublé le plus de personnes ? Combien de personnes a-t-il doublées ?

Grouping Sets

Ce TP nécessite PostgreSQL 9.5 ou supérieur. Il s'appuie sur les tables présentes dans le schéma `magasin`.

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.
10. Ajouter également le montant total des commandes depuis le début de l'activité.
11. Ajouter également le montant total des commandes par pays.

1.8 TRAVAUX PRATIQUES (SOLUTIONS)

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

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

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

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

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

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

Agrégation

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

```
SET search_path = brno2015;
```

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

Le coureur :

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

La requête donne le résultat suivant :

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

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

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

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

Le résultat montre le coureur qui a abandonné en premier :

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

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

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

Le pilote 19 a donc été le plus régulier :

```
       nom       |      stddev
-----------------+------------------
 Alvaro BAUTISTA | 0.222825823492654
```

Window Functions

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

```
SET search_path = brno2015;
```

3. Afficher la place sur le podium pour chaque coureur.

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

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

La requête affiche le résultat suivant :

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

```
   23 | Cal CRUTCHLOW    | CWM LCR Honda               | 00:27:38.315
   24 | Eugene LAVERTY   | Aspar MotoGP Team           | 00:08:04.096
   25 | Alex DE ANGELIS  | E-Motion IodaRacing Team    | 00:06:05.782
(25 rows)
```

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

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

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

La requête affiche le résultat suivant :

```
 r|       nom       |          ecurie      | total_time  |  difference
--+-----------------+----------------------+-------------+---------------
 1| Jorge LORENZO   | Movistar Yamaha [...]|00:42:53.042 | 00:00:00
 2| Marc MARQUEZ    | Repsol Honda Team    |00:42:57.504 | 00:00:04.462
 3| Valentino ROSSI | Movistar Yamaha [...]|00:43:03.439 | 00:00:10.397
 4| Andrea IANNONE  | Ducati Team          |00:43:06.113 | 00:00:13.071
 5| Dani PEDROSA    | Repsol Honda Team    |00:43:08.692 | 00:00:15.65
 6| Andrea DOVIZIOSO| Ducati Team          |00:43:08.767 | 00:00:15.725
 7| Bradley SMITH   | Monster Yamaha Tech 3|00:43:14.863 | 00:00:21.821
 8| Pol ESPARGARO   | Monster Yamaha Tech 3|00:43:16.282 | 00:00:23.24
 9| Aleix ESPARGARO | Team SUZUKI ECSTAR   |00:43:36.826 | 00:00:43.784
10| Danilo PETRUCCI | Octo Pramac Racing   |00:43:38.303 | 00:00:45.261
11| Yonny HERNANDEZ | Octo Pramac Racing   |00:43:43.015 | 00:00:49.973
12| Scott REDDING   | EG 0,0 Marc VDS      |00:43:43.216 | 00:00:50.174
13| Alvaro BAUTISTA | Aprilia Racing [...] |00:43:47.479 | 00:00:54.437
14| Stefan BRADL    | Aprilia Racing [...] |00:43:47.666 | 00:00:54.624
15| Loris BAZ       | Forward Racing       |00:43:53.358 | 00:01:00.316
16| Hector BARBERA  | Avintia Racing       |00:43:54.637 | 00:01:01.595
17| Nicky HAYDEN    | Aspar MotoGP Team    |00:43:55.43  | 00:01:02.388
```

```
18| Mike DI MEGLIO  | Avintia Racing       |00:43:58.986 | 00:01:05.944
19| Jack MILLER     | CWM LCR Honda        |00:44:04.449 | 00:01:11.407
20| Claudio CORTI   | Forward Racing       |00:44:43.075 | 00:01:50.033
21| Karel ABRAHAM   | AB Motoracing        |00:44:55.697 | 00:02:02.655
22| Maverick VIÑALES| Team SUZUKI ECSTAR   |00:29:31.557 | -00:13:21.485
23| Cal CRUTCHLOW   | CWM LCR Honda        |00:27:38.315 | -00:15:14.727
24| Eugene LAVERTY  | Aspar MotoGP Team    |00:08:04.096 | -00:34:48.946
25| Alex DE ANGELIS | E-Motion Ioda[...]    |00:06:05.782 | -00:36:47.26
(25 rows)
```

5. Pour chaque tour, afficher :

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

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

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

Elle retourne le résultat suivant :

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

```
      19 |         4 | 00:01:57.412 | 00:37:14.537
      20 |         4 | 00:01:57.786 | 00:39:12.323
      21 |         4 | 00:01:58.087 | 00:41:10.41
      22 |         4 | 00:01:58.357 | 00:43:08.767
(…)
```

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

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

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

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

```
SELECT no_tour,
  nom,
  rank()
    OVER (PARTITION BY no_tour
          ORDER BY temps_tour_glissant ASC
          ) AS place_course,
  temps_tour_glissant,
  temps_tour_glissant - first_value(temps_tour_glissant)
    OVER (PARTITION BY no_tour
           ORDER BY temps_tour_glissant asc
          ) AS difference
FROM (
```

```
  SELECT *, SUM(lap_time)
    OVER (PARTITION BY no_pilote
          ORDER BY no_tour)
          AS temps_tour_glissant
  FROM brno_2015) course
  JOIN pilotes
    ON (pilotes.no = course.no_pilote)
ORDER BY no_tour;
```

La requête fournit le résultat suivant :

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

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

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

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

classement_tour AS (
```

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

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

Ce qui donne le résultat suivant :

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

7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même

position tout au long de la course.

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

Elle retourne le résultat suivant :

```
 no_pilote
-----------
        93
        99
```

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

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

```
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant
    FROM temps_glissant
),
depassement  AS (
    SELECT no_pilote,
      last_value(place_course) OVER (PARTITION BY no_pilote) as rang,
    CASE
        WHEN lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course < 0
        THEN 0
        ELSE lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course
        END AS depasse
    FROM classement_tour t
)

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

Grouping Sets

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

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

```
SET search_path = magasin;
```

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

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
```

```
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY GROUPING SETS (
   (extract('year' from date_commande), code_pays),
   (extract('year' from date_commande))
 );
```

Le résultat attendu est :

```
 annee | code_pays | montant_total_commande
-------+-----------+------------------------
  2003 | DE        |               49634.24
  2003 | FR        |               10003.98
  2003 |           |               59638.22
  2008 | CA        |             1016082.18
  2008 | CN        |              801662.75
  2008 | DE        |              694787.87
  2008 | DZ        |              663045.33
  2008 | FR        |             5860607.27
  2008 | IN        |              741850.87
  2008 | PE        |             1167825.32
  2008 | RU        |              577164.50
  2008 | US        |              928661.06
  2008 |           |            12451687.15
(...)
```

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

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

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

11. Ajouter également le montant total des commandes par pays.

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

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

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

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

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

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

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

LIVRES BLANCS

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

TÉLÉCHARGEMENT GRATUIT

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

DALIBO, L'EXPERTISE POSTGRESQL

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

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

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