SQL moderne et requêtes analytiques

BDAV-2 – Bases de Données AVancées 2

THION Romuald

Licence Informatique UNC 2022

1 Introduction

On peut séparer les traitement exécutés sur un SGBD-R en deux grandes familles :

— **transactionnel** (*Online Transaction Processing* – OLTP) :
  — nombreuses transactions concurrentes;
  — opération `INSERT`, `UPDATE`, `DELETE` mélées;
  — `SELECT` généralement simples et précis,
    — e.g., qui ne retournent qu'un seul tuple;
  — un temps d'accès attendu en *millisecondes*;
  — intérêt à *normaliser* la base pour garantir une **cohérence** maximale.
— **analytique** (*Online Analytical Processing* – OLAP) :
  — gros volume de données, peu de concurrence;
  — essentiellement des modifications `INSERT`;
    — *append only*, on ne revient pas sur l'histoire;
  — des `SELECT` complexes avec de nombreux agrégats;
  — temps d'accès en *secondes* acceptables;
  — intérêt à *dénormaliser* pour avoir une **performance** maximale.

Il y a des bases de données adaptées aux deux types de travaux. PostgreSQL est plutôt OLTP mais il est assez performant et dispose d'outils pour faire des requêtes analytiques.

On va voir de nouvelles constructions SQL, dont certaines pour les requêtes (de sélection) analytiques. Ce sont des requêtes qui reviennent très souvent, que l'on peut écrire en SQL:1992 mais qui sont très laborieuses. On va voir ici des ajouts des standards SQL contemporatins (SQL:1999, SQL:2003, SQL:2011). Enfin, on verra un peu les CTE et leur pouvoir d'expression étendu.

1.1 Jeux de données

Pour la mise en place, exécuter le fichier dataset.sql qui produit :

- une table `sensor` contenant des données générées aléatoirement,
- un jeu d'essai *RH* avec une table `emp` pour les employés et une table `dep` pour la hiérarchie des services.

1.2 Références

Documentation officielle PostgreSQL

- *windows function*
  - https://www.postgresql.org/docs/current/tutorial-window.html
  - https://www.postgresql.org/docs/current/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS
  - https://www.postgresql.org/docs/current/functions-window.html
- `GROUPING SETS`, `CUBE`, et `ROLLUP` pour les opération *cube*
  - https://www.postgresql.org/docs/current/queries-table-expressions.html#QUERIES-GROUPING-SETS
- `FILTER` clause pour calculer des *pivots*
  - https://www.postgresql.org/docs/14/sql-expressions.html#SYNTAX-AGGREGATES
- `LATERAL JOIN` *subqueries*
  - https://www.postgresql.org/docs/current/queries-table-expressions.html#QUERIES-LATERAL
- `[MATERIALIZED] VIEW`
  - https://www.postgresql.org/docs/current/tutorial-views.html
  - https://www.postgresql.org/docs/current/sql-createview.html
  - https://www.postgresql.org/docs/current/sql-creatematerializedview.html
- `WITH [RECURSIVE]` clause, *Common Table Expression*
  - https://www.postgresql.org/docs/current/queries-with.html

En complément :

- un exposé *Postgres Window Magic* de Bruce MOMJIAN https://momjian.us/main/presentations/sql.html (vidéo et slides)
- https://modern-sql.com/ *A lot has changed since SQL-92* par Markus WINAND.

2 Fonctions de fenêtrage (*windows function*)

Une fonction de fenêtrage effectue un calcul sur un jeu d'enregistrements liés d'une certaine façon à l'enregistrement courant. On peut les rapprocher des calculs réalisables par une fonction d'agrégat. Cependant, ***les fonctions de fenêtrage n'entraînent pas le regroupement des enregistrements traités en un seul****, [...]. À la place, chaque enregistrement garde son identité propre.* Doc PostgreSQL (fr)

On a souvent besoin de combiner le résultat d'un agrégat avec des données *de la même table*. Comme on va le voir sur l'exemple suivant, on a une tension entre :

— le regroupement dont on a besoin pour le calcul
— le résultat final qu'on ne veut **pas** regroupés, où on veut toutes les lignes.

Requête : *donner toutes les informations de chaque employé ainsi que la différence entre son salaire et le salaire moyen de son équipe.*

2.1 Solution traditionnelle

Sans fenêtrage, on fait une sous-requête d'agrégation **et** une jointure sur la même table, avec une requête imbriquée `FROM` ou une CTE `WITH` :

```
SELECT emp.*, round(salary - sal.avg) AS delta
FROM emp JOIN (
    SELECT depname, AVG(salary) as avg
    FROM emp GROUP BY depname) AS sal
    ON emp.depname = sal.depname
ORDER BY depname, empno;
```

```
  depname  | empno | salary | delta
-----------+-------+--------+-------
 develop   |     7 |   4200 |  -820
 develop   |     8 |   6000 |   980
 develop   |     9 |   4500 |  -520
 develop   |    10 |   5200 |   180
 develop   |    11 |   5200 |   180
 personnel |     2 |   3900 |   200
 personnel |     5 |   3500 |  -200
 sales     |     1 |   5000 |   300
 sales     |     3 |   4800 |   100
 sales     |     4 |   4800 |   100
 sales     |    12 |   4200 |  -500
(11 rows)
```

La même avec `WITH` (on viendra sur cet opérateur), préférée personnellement car le rend la sous-requête plus lisible mais qui a **exactement le même plan d'exécution** (et donc le même résultat) :

```
-- l'aggrégat sur emp
WITH sal AS(
    SELECT depname, AVG(salary) as avg
    FROM emp GROUP BY depname
)

-- la jointure entre emp et l'agrégat
SELECT emp.*, round(salary - sal.avg) AS delta
FROM emp JOIN sal
    ON emp.depname = sal.depname
ORDER BY depname, empno;
```

2.2 Solution *windows*

Avec fenêtrage, on précise *la fenêtre* (ou *partition*, mais le terme est polysémique), c'est-à-dire *le groupement intermédiaire sur lequel faire le calcul*, ici de moyenne. *Notez les parenthèses un peu surprenantes de la fonction* `round`.

```
SELECT  emp.*,
        round(salary - avg(salary) OVER (PARTITION BY depname)) AS delta
FROM emp
ORDER BY depname, empno;
```

On peut grâce aux *windows function* faire des opérations **difficiles à exprimer** sur le `GROUP BY`, par exemple le calcul *du rang* (dense ou pas) du salarié dans son équipe.

Requête : *donner le rang dense de chaque employé (les ex-aequos ne créant pas de trous) par ordre de salaire décroissant au sein de son équipe avec l'écart à la moyenne entre son salaire et celle de son équipe.* **TODO**. *Indice* : utiliser la fonction `dense_rank`.

2.3 Performance des fonctions de fenêtrage

En plus de l'expression concise (mais quelque fois assez absconse) et de l'expressivité augmentée par rapport aux agrégats usuels SQL:19992, les fonctions de fenêtrage sont **performantes**. On reprend le jeu d'essai avec un peu plus de volume en générant 1 000 services et 100 000 employés, consulter et exécuter le fichier random_emp_dep.sql.

On va comparer la solution traditionnelle SQL:1992 avec celle avec le fenêtrage grâce à la commande `EXPLAIN ANALYZE`. Pour la solution traditionnelle, on obtient le plan suivant où la jointure est très efficace (un seul tuple par `depname` dans la sous-requête `q`)

```
 Sort  (cost=12877.47..13127.47 rows=100000 width=82) (actual time=226.366..236.422 rows=100000 loops=1)
   Sort Key: emp.depname
   Sort Method: quicksort  Memory: 10885kB
   ->  Hash Join  (cost=2172.04..4572.65 rows=100000 width=82) (actual time=57.991..104.160 rows=100000
         Hash Cond: (emp.depname = q.depname)
         ->  Seq Scan on emp  (cost=0.00..1637.00 rows=100000 width=18) (actual time=0.019..6.254 rows=
         ->  Hash  (cost=2159.52..2159.52 rows=1001 width=38) (actual time=57.960..57.962 rows=1001 loop
               Buckets: 1024  Batches: 1  Memory Usage: 59kB
               ->  Subquery Scan on q  (cost=2137.00..2159.52 rows=1001 width=38) (actual time=57.340..5
                     ->  HashAggregate  (cost=2137.00..2149.51 rows=1001 width=38) (actual time=57.336.
                           Group Key: emp_1.depname
                           Batches: 1  Memory Usage: 321kB
                           ->  Seq Scan on emp emp_1  (cost=0.00..1637.00 rows=100000 width=10) (actual
 Planning Time: 0.332 ms
 Execution Time: 239.777 ms
```

Avec la fonction de fenêtrage, le plan est débarassé de la jointure, le plan est le suivant.

```
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 WindowAgg  (cost=9941.82..12191.82 rows=100000 width=50) (actual time=155.243..213.663 rows=100000 loop
   ->  Sort  (cost=9941.82..10191.82 rows=100000 width=18) (actual time=155.212..166.585 rows=100000 loo
         Sort Key: depname
         Sort Method: quicksort  Memory: 10885kB
         ->  Seq Scan on emp  (cost=0.00..1637.00 rows=100000 width=18) (actual time=0.017..16.426 rows=
 Planning Time: 0.126 ms
 Execution Time: 216.939 ms
```

Sur 100 exécutions, on obtient les statistiques suivantes, légèrement en faveur des fonctions de fenêtrage sur ce cas.

```
agrégat: mean = 432.10 ms, stdev = 4.80 ms, median = 431.09 ms
windows: mean = 387.73 ms, stdev = 3.60 ms, median = 387.43 ms
```

2.3.1 Notes

Un programme Python de comparaison de performance de requêtes est fourni, fichier bench.py.

2.4 Clauses `ORDER BY` et `RANGE/ROWS/GROUP` des fenêtres

La syntaxe complète des `WINDOWS` est riche. On peut définir la partition et l'ordre de tri au sein de la partition, ici la syntaxe générale

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

Au sein de la fenêtre, c'est-à-dire le sous-ensemble des tuples de la partition pris en compte pour le calcul, on peut utiliser des fonctions qui permettent de référencer les tuples précédents comme `lag()`.

Requête : sur la table `sensor` *pour chaque capteur, le temps en seconde entre deux relevés consécutifs. Indice* fonction `lag()` et soustraction de dates.

On aura besoin ici de manipulation de dates, voir la création de la date `sensor` dans dataset.sql Quand la définition de la fenêtre est longue ou employée sur plusieurs attributs, on peut la définir avec la clause `WINDOWS` et la réemployer comme suit.

```
SELECT sensor.*,
       time_stamp - (lag(time_stamp) OVER w) AS delta
FROM sensor
WINDOW w AS (PARTITION BY sensorid ORDER BY time_stamp ASC)
ORDER BY sensorid, time_stamp;
```

On peut aussi avoir besoin de définir la fênetre elle-même, pour typiquement fixer un intervalle sur lequel on souhaite regrouper. ici la syntaxe des `frame_clause`, `frame_start` et `frame_end`

```
# The optional frame_clause can be one of

{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]

# where frame_start and frame_end can be one of

UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING

# and frame_exclusion can be one of

EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

C'est assez complet et parfois subtil.

Requête : *donner, sans différencier les capteurs, la somme cumulée de la valeur.* **TODO**.

Requête : *donner, sans différencier les capteurs, la moyenne glissante de la valeur sur 1 minute. Indice* utiliser `RANGE BETWEEN '1 MINUTE' PRECEDING` dans la fenêtre. **TODO**.

3 Clause `FILTER`

La clause `FILTER` permet de mettre une condition (dans le `SELECT`) sur les tuples considérés par un agrégat. Cela permet de faire des *pivots*, typiques OLAP, appellés aussi *tableaux croisés*, qui consistent à créer un tableau 2D avec *une colonne* pour chaque valeur d'un attribut. Comme pour le fenêtrage simple, on évite la solution traditionnelle SQL:1992 avec autant de jointures que de colonnes.

Requête : *pour chaque service, compter le nombre d'employés avec un salaire dans l'intervalle [1, 1000[, ceux dans [1000, 2000[etc.*

On peut commencer par la requête préliminaire suivante.

```
SELECT depname, (1000*(salary / 1000))::text || '-' || (1000*(salary / 1000)+1000)::text  AS tranche, co
FROM emp
GROUP BY depname, salary / 1000;
```

```
  depname  |  tranche  | nb
-----------+-----------+----
 sales     | 5000-6000 |  1
 sales     | 4000-5000 |  3
 personnel | 3000-4000 |  2
 develop   | 5000-6000 |  2
 develop   | 4000-5000 |  2
 develop   | 6000-7000 |  1
(6 rows)
```

Elle a toutefois au moins deux problèmes :

- on a pas d'effectif de 0 quand il n'y a aucun employé du service dans la catégorie,
- on souhaiterait avoir autant de colonnes qu'il y a de tranches salariales, sous forme de *tableau de contingence* (très utilisé en statistique).

Il faut *pivoter* ce résulat de requête, comme dans l'illustration ci-dessous, reprise de modern SQL.

```
SELECT  depname,
        count(empno) FILTER (WHERE salary BETWEEN 3000 and 3999) AS "[3000, 4000[",
        count(empno) FILTER (WHERE salary BETWEEN 4000 and 4999) AS "[4000, 5000[",
        count(empno) FILTER (WHERE salary BETWEEN 5000 and 5999) AS "[5000, 6000[",
        count(empno) FILTER (WHERE salary BETWEEN 6000 and 6999) AS "[6000, 7000[",
        count(empno) FILTER (WHERE salary >= 7000) AS "[7000, Inf["
FROM emp
GROUP BY depname;
```

```
  depname  | [3000, 4000[ | [4000, 5000[ | [5000, 6000[ | [6000, 7000[ | [7000, Inf[
-----------+--------------+--------------+--------------+--------------+------------
 personnel |            2 |            0 |            0 |            0 |           0
 sales     |            0 |            3 |            1 |            0 |           0
 develop   |            0 |            2 |            2 |            1 |           0
(3 rows)
```

On peut aussi utiliser une extension comme `tablefunc` qui fait quelque chose de similaire. L'extension s'installe par la commande `CREATE EXTENSION tablefunc;`.

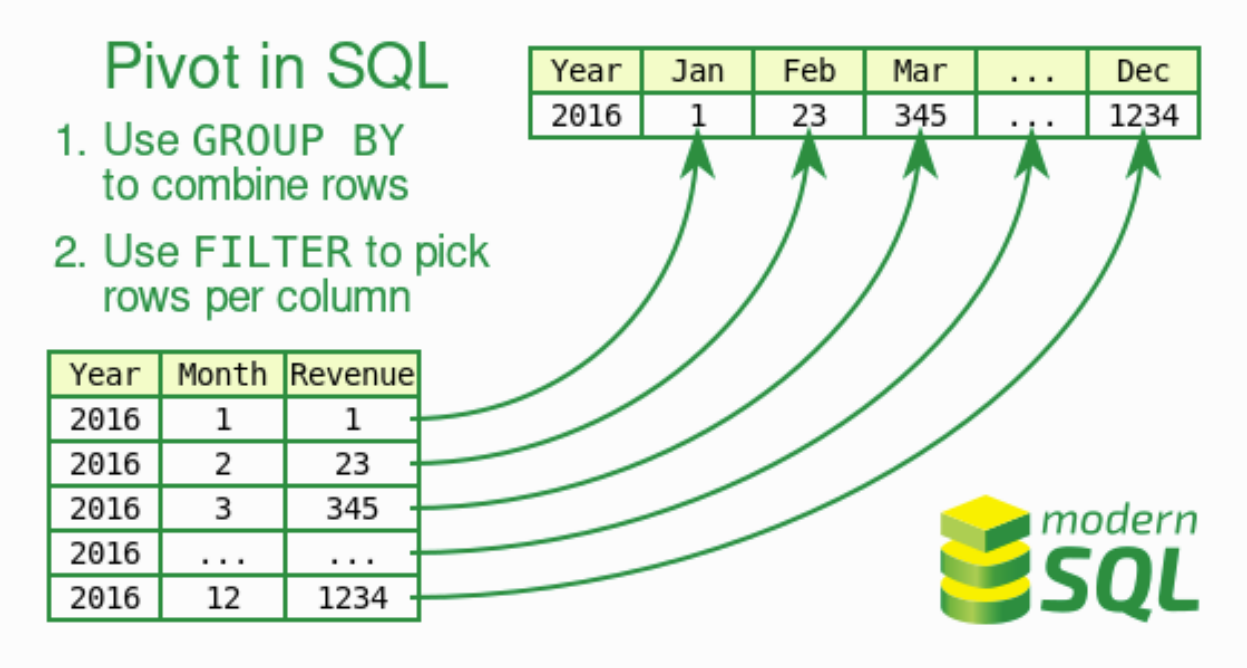

FIGURE 1 – Illustration du pivot qui transforme les valeurs d'une colonne en autant de colonnes

```
SELECT *
  FROM crosstab(
      'SELECT depname, salary / 1000 AS tranche, count(empno) AS nb FROM emp GROUP BY depname, salary
      'SELECT DISTINCT salary/1000 from emp ORDER by 1'
  ) AS (depname text, "[3000, 4000[" int, "[4000, 5000[" int, "[5000, 6000[" int, "[6000, 7000[" int)
```

Notez les `NULL` au lieu des valeurs `0` de la version avec les `FILTER`.

```
  depname  | [3000, 4000[ | [4000, 5000[ | [5000, 6000[ | [6000, 7000[
-----------+--------------+--------------+--------------+--------------
 sales     |            ∅ |            3 |            1 |            ∅
 personnel |            2 |            ∅ |            ∅ |            ∅
 develop   |            ∅ |            2 |            2 |            1
(3 rows)
```

Une des limites de l'approche ici est que la liste des colonnes est *statique*. Les SGBD (R)OLAP ont des opérateurs spécifiques pour éviter ceci, mais ici on a une limite du modèle relationnel où les colonnes doivent être connues *avant* l'exécution de la requête. On peut, pour éviter ceci :

- faire le pivot dans l'application hôte, par exemple avec DataFrame.pivot_table() si on est en Python;
- générer la requête programmatiquement, côté serveur ou côté client, avec par exemple un *template*;
- utiliser une commande spéciale de `psql` comme `\crosstabview` qui pivote le dernier résulat de requête, mais ceci ne fonctionnera **que** pour `psql`.

```
SELECT  depname, salary / 1000 AS tranche, count(empno) AS nb
FROM emp
GROUP BY depname, salary / 1000;

\crosstabview
```

```
  depname  | 5 | 4 | 3 | 6
-----------+---+---+---+---
 sales     | 1 | 3 |   |
 personnel |   |   | 2 |
 develop   | 2 | 2 |   | 1
(3 rows)
```

4 Les opérateurs de regroupement `GROUPING SETS`

Ces opérateurs permettent de spécifier plusieurs dimensions sur lesquelles agréger les données selon **toutes ou parties des dimensions**. Là aussi, une opération typique (R)OLAP. Par exemple, pour les capteurs, on voudrait avoir le nombre de relevés :

— pour chaque capteur et pour tous les capteurs;
— pour chaque minute et pour toutes les dates.

Si on a c capteur et d dates, on obtient un tableau de de taille $(c+1) \times (d+1)$ avec les *sommes marginales*.

La solution classique consiste à faire l'`UNION` des **quatre agrégats calculés séparéments** :

```
SELECT sensorid AS sensorid, date_trunc('minute', time_stamp) AS time_stamp, count(value) AS nb
FROM sensor
GROUP BY sensorid, date_trunc('minute', time_stamp)

UNION

SELECT NULL, date_trunc('minute', time_stamp), count(value)
FROM sensor
GROUP BY date_trunc('minute', time_stamp)

UNION

SELECT sensorid AS sensorid, NULL AS time_stamp, count(value)
FROM sensor
GROUP BY sensorid

UNION

SELECT NULL AS sensorid, NULL AS time_stamp, count(value) AS nb
FROM sensor
ORDER BY sensorid NULLS FIRST, time_stamp NULLS FIRST;
```

TODO le faire en utilisant les `GROUPING SET`, ici l'opérateur `CUBE` en particulier

TODO utiliser enfin `\crosstabview` pour avoir une représentation en 2D comme suit

```
 sensorid |  ∅   | 14:26:00 | 14:27:00 | 14:28:00 | 14:29:00 | 14:30:00 | 14:31:00 | 14:32:00 | 14:33:0
----------+------+----------+----------+----------+----------+----------+----------+----------+--------
        ∅ | 1000 |       58 |       60 |       60 |       60 |       60 |       60 |       60 |       6
        0 |   58 |        6 |        2 |        5 |        4 |        4 |        3 |        4 |
        1 |  115 |        7 |        6 |        9 |       13 |        7 |        8 |        6 |
        2 |   87 |        4 |       10 |        5 |        6 |       10 |        4 |        4 |
        3 |   91 |        6 |        6 |        5 |          |        6 |        6 |        8 |
        4 |   90 |        2 |        9 |        6 |        1 |        7 |        8 |        6 |
```

```
       5 |   99 |       9 |       3 |       4 |      10 |       4 |       4 |       4 |
       6 |   98 |       5 |       8 |       2 |       5 |       3 |       6 |       5 |
       7 |  101 |       4 |       3 |       6 |       5 |      10 |       5 |       6 |
       8 |  101 |       4 |       5 |       5 |       6 |       5 |       7 |       5 |
       9 |  117 |       8 |       6 |      10 |       8 |       3 |       7 |      11 |
      10 |   43 |       3 |       2 |       3 |       2 |       1 |       2 |       1 |
(12 rows)
```

5 JOIN LATERAL

Par défaut, on ne peut pas faire de sous-requêtes coréllées dans le `FROM`, l'opérateur `JOIN LATERAL` lève cette restriction qui permet de faire des *sous-requêtes latérale*. Le principe est, pour chaque tuple de la table de gauche, calculer la requête qui produit la table de droite en utilisant les valeur du tuple de gauche.

Requête : *donner la variation de valeur entre un relevé et celui qui le précède immédiatement* (peut importe le capteur concerné). Ici, on va utiliser avec un `CROSS JOIN LATERAL` après avoir donné un rang aux tuples.

```
WITH ordered_sensor AS(
  SELECT sensor.*, dense_rank() OVER (ORDER BY time_stamp ASC) AS rank
  FROM sensor
  ORDER BY time_stamp ASC
)

SELECT s1.*, s1.value - s2.value as delta
FROM ordered_sensor AS s1
     CROSS JOIN LATERAL
     (SELECT s2.* FROM ordered_sensor AS s2 WHERE s1.rank = s2.rank + 1 ) AS s2
;
```

Avec le plan suivant :

```
 Merge Join  (cost=223.99..331.49 rows=5000 width=84)
   Merge Cond: (s1.rank = ((s2.rank + 1)))
   CTE ordered_sensor
     ->  WindowAgg  (cost=66.83..84.33 rows=1000 width=24)
           ->  Sort  (cost=66.83..69.33 rows=1000 width=16)
                 Sort Key: sensor.time_stamp
                 ->  Seq Scan on sensor  (cost=0.00..17.00 rows=1000 width=16)
   ->  Sort  (cost=69.83..72.33 rows=1000 width=52)
         Sort Key: s1.rank
         ->  CTE Scan on ordered_sensor s1  (cost=0.00..20.00 rows=1000 width=52)
   ->  Sort  (cost=69.83..72.33 rows=1000 width=40)
         Sort Key: ((s2.rank + 1))
         ->  CTE Scan on ordered_sensor s2  (cost=0.00..20.00 rows=1000 width=40)
```

TODO faire la même chose avec la *window function* `lag` et comparer les plans d'exécution.

TODO évaluer la performance empirique avec bench.py. Pour cel, générez un jeu de données plus gros. L'écart doit être substantiel.

6 Vues, vues récursives et vues matérialisées

6.1 Principe

Les vue ne sont pas particulièrement *modernes* mais *très utilisées* :

- une vue est *une requête à laquelle on a donné un nom* et qui s'utilise *comme une table*, en refaisant le calcul (s'il n'est ni dans le cache ni matérialisé).
- on peut **matérialiser la vue**, auquel cas la vue devient *vraiment* une table avec des tuples **persistants** :
  - dans ce cas là il faut *rafraîchir* la vue quand les données sources change et mettre à jour la table de stockage.

Les cas d'usages des vues sont :

- fournir une **interface** au client, s'abstraire des tables concrètes,
- **contrôler les accès** en limitant les colonnes ou lignes accessibles,
- **factoriser** les requêtes fréquentes,
- assurer la **performance** des requêtes analytiques via la matérialisation.

6.2 Exemple

Une vue sur les services :

```
DROP VIEW IF EXISTS dep_summary;
CREATE VIEW dep_summary AS(
  SELECT depname, count(empno) AS nb_emp
  FROM dep LEFT OUTER JOIN emp USING (depname)
  GROUP BY depname
  ORDER BY dep
);

TABLE dep_summary;
INSERT INTO dep VALUES ('test', NULL);
TABLE dep_summary;

-- la vue est à jour : la requête a été recalculée
```

6.3 Les *Common Table Expression* (CTE)

- le `WITH` *non résursif* est utilisable comme une vue *à portée locale*
  - très pratique pour *organiser les grosses requêtes*
- avec le mot-clef `[RECURSIVE]` on peut construire des vues qui font référence **à elle-même dansleur définition**
  - étend **considérablement** l'expressivité de SQL
    - permet de faire des parcours d'arbres/graphes
  - peut conduire à des requêtes *qui ne terminent jamais* !

6.4 CTE non récursives

Dans l'exemple suivant, `small_dep` est une *vue* locale, dont la définition sera utilisée soit :

- `NOT MATERIALIZED`, c'est-à-dire *dépliée* lors de l'évaluation de la requête, son contenu est injecté dans la requête englobante et l'optimisation est faite globalement;

— `MATERIALIZED`, les évaluations sont séquentielles : la sous-requête, puis la requête englobante utilise le résutat qui a été enregistré.

```
WITH small_dep AS NOT MATERIALIZED (
  SELECT depname
  FROM dep LEFT OUTER JOIN emp USING (depname)
  GROUP BY depname
  HAVING count(empno) BETWEEN 1 AND 2
  ORDER BY depname
)

SELECT emp.*
FROM emp JOIN small_dep USING (depname);
```

TODO comparer les plans d'exécution des deux requêtes, avec/sans le mot-clef `NOT`.

TODO vérifier la différence de performance, qui n'est pas très importante sur ce cas. Quand peut-elle changer ?

6.5 Les CTE recursives

On peut définir des vues qui **font référence à elles-mêmes** et permet de calculer des résultats *impossible* en SQL sans récursion, comme *la fermeture transitive* d'une relation.

Par exemple, ici le calcul de *fermeture transitive* des services : pour chaque service, calculer **tous les sous-services qui en dépendent directement ou pas**. Autrement dit, dans l'arbre `dep` des services seuls les relations *enfant - parent* sont stockées, ici on veut *tous les descendants*, qu'elle que soit la profondeur. De base, la relation `dep` est comme suit :

```
   depname   |   parent
-------------+------------
 direction   | ∅
 production  | direction
 personnel   | direction
 sales       | direction
 develop     | production
 maintenance | production
 team 1      | develop
 team 2      | develop
(8 rows)
```

On construit la vue récursive, avec deux cas :

— **le cas de base**, si `e` est un *enfant* de `p`, alors `e` est un *descendant* de `p` : c'est le contenu de la table `dep`, les enfants immédiats;
— **le cas récursif**, si `e` est un *enfant* de `p` et que `p` est un **descendant** de `gp`, alors `e` est un *descendant* de `gp`.

```
WITH RECURSIVE dep_rec(depname, parent) AS (
  -- chemin de longueur 1
    SELECT depname, parent
    FROM dep
  UNION
  -- extension des chemins avec une nouvelle étape
    SELECT dep.depname, dep_rec.parent
    FROM dep JOIN dep_rec ON dep.parent = dep_rec.depname
```

```
)


SELECT * FROM dep_rec
ORDER BY depname, parent;
```

Si on utilise souvent la table `dep_rec`, on aura très envie d'en faire une vue (récursive), possiblement matérialisée.

```
DROP VIEW IF EXISTS dep_trans;

-- syntaxe abrégée pour les vues recursives, qui évite un sélect
-- https://www.postgresql.org/docs/current/sql-createview.html
CREATE RECURSIVE VIEW dep_trans(depname, parent) AS (
    SELECT depname, parent
    FROM dep
  UNION
    SELECT dep.depname, dep_trans.parent
    FROM dep JOIN dep_trans ON dep.parent = dep_trans.depname
);

DROP MATERIALIZED VIEW IF EXISTS dep_trans_m;

-- pas de syntaxe abrégée ici
CREATE MATERIALIZED VIEW dep_trans_m AS(
  WITH RECURSIVE dep_trans(depname, parent) AS (
      SELECT depname, parent
      FROM dep
    UNION
      SELECT dep.depname, dep_trans.parent
      FROM dep JOIN dep_trans ON dep.parent = dep_trans.depname
  )
  SELECT * FROM dep_trans
);
```

6.6 Exercices

Ecrire les requêtes suivantes :

- pour chaque service, calculer les sous-services qui en dépendent transitivement en comptant aussi **la profondeur** depuis `direction`.
- donner pour chaque service, le salaire min et le salaire max de tous les subordonnés (transitivement).
  - *Indice* pour le service `direction` on doit avoir le min et le max de l'ensemble de la société.
- donner pour chaque service, le nombre total d'employés qui en dépendent transitivement.
  - *Indice* calculer d'abord la table suivante puis utiliser la requête récursive.
  - Vérifier le comportement en ajoutant un tuple.
  - Penser à la reflexivité de la relation `dep_hierarchy`

```
INSERT INTO emp VALUES
    ('team 1'   , 13, 5200);
```

On doit avoir pour le nombre d'employés directs

```
   depname   | nb
-------------+----
```

```
 develop     |  5
 maintenance |  0
 personnel   |  2
 production  |  0
 sales       |  4
 team 1      |  1
 team 2      |  0
```

Et avec la transitivité

```
   depname   | sum
-------------+-----
 develop     |   6
 direction   |  12
 maintenance |   0
 personnel   |   2
 production  |   6
 sales       |   4
 team 1      |   1
 team 2      |   0
```