Module M6

Statistiques pour le planificateur

25.03.1

Table des matières

Sur ce document

Formation	Module M6
Titre	Statistiques pour le planificateur
Révision	25.03.1
PDF	https://dali.bo/m6_pdf
EPUB	https://dali.bo/m6_epub
HTML	https://dali.bo/m6_html
Slides	https://dali.bo/m6_slides
TP	https://dali.bo/m6_tp
TP (solutions)	https://dali.bo/m6_solutions

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

Chers lectrices & lecteurs,

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

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

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

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

Versions de PostgreSQL couvertes

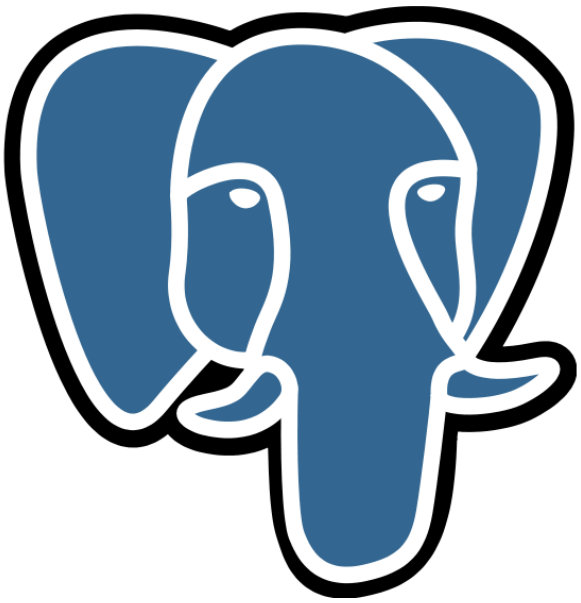
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ Statistiques pour le planificateur



1.1 INTRODUCTION

- Indispensables !
- Statistiques sur les données pour le planificateur
  - pour le choix du parcours
  - comme le choix des jointures
- Les statistiques indiquent :
  - la cardinalité d'un filtre → stratégie d'accès
  - la cardinalité d'une jointure → algorithme de jointure
  - la cardinalité d'un regroupement → algorithme de regroupement
- Sans bonnes statistiques, pas de bons plans !

Connaître le coût unitaire de traitement d'une ligne est utile, mais sans savoir combien de lignes sont à traiter, il est impossible de calculer le coût total d'une requête. PostgreSQL s'appuie donc principalement sur les statistiques, qui sont un échantillonnage des données d'une table, pour guider le planificateur. Ces statistiques permettent de déterminer la répartition et la qualité des valeurs dans une colonne.

Elles sont essentielles pour évaluer la sélectivité des filtres (clauses `WHERE`, jointures) et choisir la stratégie optimale : utiliser un index, organiser les jointures ou choisir un algorithme de regroupement. Sans statistiques précises et à jour, le planificateur pourrait générer des plans d'exécution inefficaces, impactant les performances des requêtes. Ces décisions s'appuient sur les informations contenues dans les statistiques ainsi que sur certains paramètres de configuration, qui permettent d'orienter le planificateur.

Ces statistiques sont collectées automatiquement, mais peuvent parfois être obsolètes si les tables évoluent rapidement. Il est donc essentiel de maintenir les statistiques à jour avec la commande `ANALYZE` pour garantir des plans de requête efficaces.

Les statistiques sur les données utilisées par le planificateur ne doivent pas être confondues avec les statistiques d'activité recensées dans les vues `pg_stat_*` et `pg_statio_*` (chargement système, connexions, etc.) qui sont distinctes et ne seront pas évoquées ici.

Voir au besoin les modules de formation *H1 - Supervision de PostgreSQL*[1] ou *H2 - Analyses et diagnostics*[2].

[1] https://dali.bo/h1_html
[2] https://dali.bo/h2_html

1.2 AU MENU

- Statistiques de volumétrie
- Statistiques sur les données
  - principe
  - statistiques étendues
- `ANALYZE` et `autovacuum`

Le présent module nous permettra de comprendre sur quelles informations l'optimiseur se base pour calculer ses coûts. En effet, les choix de parcours de table et d'index ne sont pas les mêmes selon la quantité et la qualité des données concernées. Les statistiques permettent d'obtenir ces informations.

1.3 STATISTIQUES DE VOLUMÉTRIE

1.3.1 pg_class

```
SELECT relpages, reltuples FROM pg_class WHERE relname = "employes";
```

- pour tables et index
- `relpages` : taille en blocs
  - dont `relallvisible`
- `reltuples` : lignes
- Mise à jour : `VACUUM` / `ANALYZE`

L'optimiseur a besoin de deux données statistiques pour une table ou un index : sa taille physique et le nombre de lignes portées par l'objet. Ces deux données statistiques sont stockées dans la table `pg_class`.

- La taille physique sur le disque de la table ou de l'index est exprimée en nombre de blocs de 8 ko, et stockée dans la colonne `relpages`.
- `relallvisible` est le nombre de blocs dont toutes les lignes sont visibles (« vivantes »).
- La cardinalité de la table ou de l'index, c'est-à-dire le nombre de lignes, est stockée dans la colonne `reltuples`. Il s'agit des lignes vivantes, sans les lignes mortes non nettoyées (lesquelles se trouvent dans `pg_stat_user_tables`).

Attention : ces valeurs sont des estimations, pas forcément à jour. La mise à jour a lieu essentiellement lors d'un `VACUMM` ou `ANALYZE` récent.

Ces chiffres permettent au planificateur de calculer une densité de lignes dans les blocs à récupérer. Quand il aura estimé un nombre de lignes à lire dans une table ou un index, il pourra ainsi connaître le nombre de blocs à lire.

1.4 STATISTIQUES SUR LES DONNÉES

1.4.1 Tables de statistiques

- Échantillonner pour mieux planifier
- Estimer la sélectivité d'une clause `WHERE` ou `JOIN`
- Tables `pg_statistic` (par colonne), `pg_statistic_ext`…
- Vues **`pg_stats`** (par colonne), `pg_stats_ext` et `pg_stats_ext_exprs`

Les statistiques sur les données permettent de déterminer la sélectivité d'un filtre (prédicat d'une clause `WHERE` ou une condition de jointure) et donc quelle sera la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué.

Prenons un exemple avec la base **employes** (le SQL pour l'alimenter est sur https://dali.bo/tp_employes_services). Avec un filtre peu sélectif, `date_embauche = '2006-09-01'`, la requête suivante ramène pratiquement l'intégralité de la table. PostgreSQL choisit donc une lecture séquentielle de la table, ou *Seq Scan* :

```
EXPLAIN (ANALYZE, TIMING OFF)
  SELECT *
  FROM employes_big
  WHERE date_embauche='2006-09-01';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..10901.69 rows=498998 width=40)
                                        (actual rows=499004 loops=1)
   Filter: (date_embauche = '2006-09-01'::date)
   Rows Removed by Filter: 11
 Planning time: 0.027 ms
 Execution time: 42.624 ms
```

La partie `cost` montre que l'optimiseur estime que la lecture ramène 498 998 lignes. Comme on peut le voir, ce n'est pas exact : elle en récupère 499 004. Ce n'est qu'une estimation basée sur des statistiques selon la répartition des données et ces estimations sont la plupart du temps un peu erronées. L'important est de savoir si l'erreur est négligeable ou si elle est importante. Dans notre cas, elle est négligeable. On lit aussi que 11 lignes ont été filtrées pendant le parcours (et 499 004 + 11 correspond bien aux 499 015 lignes de la table).

Avec un filtre sur une valeur beaucoup plus sélective, la requête ne ramène que 2 lignes. L'optimiseur préfère donc passer par l'index présent sur la colonne :

```
EXPLAIN (ANALYZE, TIMING OFF)
  SELECT *
  FROM employes_big
  WHERE date_embauche='2006-01-01';
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
        (cost=0.42..4.44 rows=1 width=41) (actual rows=2 loops=1)
   Index Cond: (date_embauche = '2006-01-01'::date)
 Planning Time: 0.213 ms
 Execution Time: 0.090 ms
```

Dans ce deuxième essai, l'optimiseur estime ramener 1 ligne. En réalité, il en ramène 2. L'estimation reste relativement précise étant donné le volume de données.

Dans le premier cas, l'optimiseur prévoit de sélectionner l'essentiel de la table et estime qu'il est moins coûteux de passer par une lecture séquentielle de la table plutôt qu'une lecture d'index. Dans le second cas, où le filtre est très sélectif, une lecture par index est plus appropriée.

Pour comprendre les choix de l'optimiseur, il est possible de consulter les statistiques dont il dispose. Pour les statistiques sur chaque colonne séparément existe la vue `pg_stats`. Cette vue a été créée pour faciliter la compréhension des statistiques stockées dans la table système `pg_statistic`.

Les vues `pg_stats_ext` et `pg_stats_ext_exprs` sont basées sur les tables systèmes `pg_statistic_ext` et `pg_statistic_ext_data`, qui concernent les statistiques étendues. Celles-ci peuvent porter sur plus d'une colonne, et par défaut elles ne sont pas calculées.

Ces tables sont alimentées par la commande `ANALYZE`.

1.5 STATISTIQUES MONOVARIÉES

1.5.1 Statistiques monovariées

- Pour chaque colonne de chaque table
- Adaptées à la majorité des cas
  - forte ou faible contrainte de type
  - forte ou faible variabilité des données
- Calculées automatiquement par `ANALYZE`

Par défaut, PostgreSQL échantillonne chaque colonne de chaque table avec `ANALYZE` pour évaluer la qualité des données. En fonction des contraintes de type (nombre, chaîne de caractères, binaires) et de la variabilité des valeurs observées, différentes statistiques adaptées vont être calculées.

1.5.2 Valeurs distincts (N-Distinct)

```
-[ RECORD 1 ]----------+-----------------------------------------------
schemaname             | public
tablename              | employes
attname                | date_embauche
n_distinct             | -0.5
```

- Estimation basée sur l'échantillon
- Si > 0 : nombre de valeurs distinctes dans la colonne
  - constant indépendamment de la volumétrie
- Si < 0 : nombre de valeurs distinctes / nombre de lignes × -1
  - proportionnel à la volumétrie
- -1 : toutes les valeurs diffèrent (ex : PK)
- La valeur peut être forcée

Les statistiques d'une colonne d'une table se récupèrent ainsi :

```
SELECT * FROM pg_stats
WHERE tablename = 'employes' AND attname = 'date_embauche' ;
```

On ne s'intéresse pour le moment qu'au champ `n_distinct`.

Si une colonne contient moins de 10 % de valeurs distinctes par rapport au nombre total de lignes (exemple : colonne `département` dans une table `adresses`), PostgreSQL considère que ce nombre reste fixe. La valeur de `n_distinct` dans `pg_stats` est alors positive et correspond exactement au compte de valeurs uniques.

Si le nombre de valeurs distinctes dépasse 10 % (exemple champ `auteur` dans une tables `commentaires`), l'optimiseur estime que ce nombre augmentera proportionnellement avec la taille de la table. Dans ce cas, `n_distinct` devient négatif pour indiquer un ratio d'accroissement. Par exemple, une valeur de -0.15 signifie que l'optimiseur prévoit que 15 % des lignes seront des valeurs distinctes (si la table passe à 20 000 lignes, il estimera environ 3 000 valeurs distinctes).

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur `=`. Par exemple les types `point` ou `json` ne supportent pas `=`, alors que `int`, `text`, `jsonb`… le supportent.

Il est possible de forcer cette colonne à une valeur constante en utilisant l'ordre :

```
ALTER TABLE nom_table ALTER COLUMN nom_colonne SET (n_distinct = <valeur>);
```

(ou `n_distinct_inherited` pour une table comprenant des partitions).

Pour les grosses tables contenant des valeurs distinctes, indiquer une grosse valeur ou la valeur -1 permet de favoriser les parcours d'index à la place des parcours bitmap. C'est aussi utile pour des tables où les données ne sont pas réparties de façon homogène, et où la collecte de cette statistique est alors faussée.

1.5.3 Most Common Values (MCV)

```
-[ RECORD 1 ]-----+-----------------------------------------------------
schemaname        | public
tablename         | employes
attname           | date_embauche
most_common_vals  |
 ↪  {2006-03-01,2006-09-01,2000-06-01,2005-03-06,2006-01-01}
most_common_freqs | {0.214286,0.214286,0.142857,0.142857,0.142857}
```

- Liste des valeurs les plus fréquentes
- Liste des fréquences de ces valeurs
- Variable en fonction de l'échantillonnage

Les informations relatives aux valeurs les plus fréquentes sont visibles dans les colonnes `most_common_vals` et `most_common_freqs` de la vue `pg_stats`.

La colonne `most_common_vals` contient une liste triée des valeurs les plus communes. Elle peut être `NULL` si les valeurs semblent toujours aussi communes ou si le type de données n'a pas d'opérateur `=`.

La colonne `most_common_freqs` contient une liste triée des fréquences pour les valeurs les plus communes. Cette fréquence est en fait le nombre d'occurrences de la valeur divisé par le nombre de lignes. Elle est `NULL` si `most_common_vals` est `NULL`.

La vue `pg_stats` comporte aussi des colonnes `most_common_elems`, `most_common_elem_freqs`, qui sont les équivalents pour les données de type tableau. Dans l'exemple suivant, on considère un tableau de mots-clés associés à chaque article :

```
SELECT * FROM articles;
 id |            tags
----+----------------------------
 33 | {stylo,bleu,bille}
 34 | {cahier,spirale,A4}
 35 | {agrafeuse,métal,compacte}
 36 | {stylo,noir,gel}
 37 | {cahier,A4,quadrillé}
 38 | {cahier,spirale,A4}
 39 | {stylo,bleu,bille}
 40 | {cahier,A4,spirale}
 41 | {stylo,bleu,roller}
 42 | {stylo,noir,gel}
 (10 lignes)
```

Dans ce genre de cas, les champs `most_common_elems` et `most_common_elem_freqs` de `pg_stats` sont renseignés, on peut y retrouver les tags les plus présents et selon quelle fréquence :

```
SELECT
    unnest(most_common_elems::text::text[]) AS tag,
    unnest(most_common_elem_freqs) AS frequency
FROM pg_stats
WHERE tablename = 'articles' AND attname = 'tags';

    tag    | frequency
-----------+-----------
 A4        |       0.4
 agrafeuse |       0.1
 bille     |       0.2
 bleu      |       0.3
 cahier    |       0.4
 compacte  |       0.1
 gel       |       0.2
 métal     |       0.1
 noir      |       0.2
 quadrillé |       0.1
 roller    |       0.1
 spirale   |       0.3
 stylo     |       0.5
           |       0.1 -- fréquence minimale observée
           |       0.5 -- fréquence maximale observée
```

```
           |         0 -- fréquence des éléments NULL
(16 lignes)
```

Par exemple, le tag « cahier » est présent dans les tags de 4 des 10 lignes de la table, sa fréquence est donc de 40 % (0,4). Les trois dernières valeurs contenues dans `most_common_elem_freqs` permettent à l'optimiseur de directement connaître les fréquences minimales, maximales et associée aux valeurs NULL de l'ensemble.

1.5.4 Histogramme

```
-[ RECORD 1
 ↪  ]-------+-----------------------------------------------------
schemaname          | public
tablename           | produits
attname             | prix
histogram_bounds    | {4.00,6.12,9.14,23.55,99.57}
```

- Décrit la répartition des données (hors MCV et `NULL`)
- Nombre de lignes sensiblement égal entre chaque borne de l'histogramme
- N'est pas calculé par `ANALYZE` si toutes les valeurs sont dans les MCV

Pour chaque colonne, les valeurs récupérées par l'échantillonnage sont triées. Ces données triées sont partagées en N intervalles de valeurs, d'amplitudes différentes mais de même nombre d'éléments. N dépend de la valeur du paramètre `default_statistics_target`, par défaut 100, et de la configuration spécifique de la colonne. PostgreSQL construit ensuite un tableau dont chaque valeur correspond à la valeur de début d'une tranche.

L'histogramme illustré ci-dessus a été construit ainsi :

```
-- Création de la table
CREATE TABLE produits (
    id SERIAL PRIMARY KEY,
    nom TEXT,
    prix NUMERIC(10,2)
);

-- Ajustement de l'échantillonnage pour l'exemple
ALTER TABLE produits ALTER COLUMN prix SET STATISTICS 4;

-- Insertion de données
INSERT INTO produits (nom, prix)
SELECT
    'Produit ' || g,
    CASE floor(random() * 4)
        WHEN 0 THEN ROUND((4 + random() * 2)::numeric, 2)    -- 25% [4-6)
        WHEN 1 THEN ROUND((6 + random() * 2)::numeric, 2)    -- 25% [6-8)
        WHEN 2 THEN ROUND((8 + random() * 9)::numeric, 2)    -- 25% [8-17)
```

```
        ELSE ROUND((17 + random() * 83)::numeric, 2)        -- 25% [17-100)
    END
FROM generate_series(1, 45000) g;

-- Calcul des statistiques
ANALYZE produits;

-- Affichage de l'histogramme
SELECT schemaname, tablename, attname, histogram_bounds
FROM pg_stats
WHERE tablename = 'produits'
AND attname = 'prix' \gx

-[ RECORD 1 ]-------+-----------------------------------------------------
schemaname          | public
tablename           | produits
attname             | prix
histogram_bounds    | {4.00,6.12,9.14,23.55,99.57}
```

Ces valeurs peuvent légèrement varier entre deux appels différents d'`ANALYZE`, car l'échantillonnage change à chaque fois.

La requête suivante permet de vérifier que les 5 bornes de l'histogramme permettent bien une répartition équilibrée des lignes :

```
WITH bounds AS (
    -- Extraction des bornes de l'histogramme
    SELECT unnest(histogram_bounds::text::numeric[]) AS bound
    FROM pg_stats
    WHERE tablename = 'produits' AND attname = 'prix'
),
intervals AS (
    -- Génération des paires de bornes (borne_inf, borne_sup)
    SELECT
        bound AS borne_inf,
        lead(bound) OVER (ORDER BY bound) AS borne_sup
    FROM bounds
),
counts AS (
    -- Comptage des lignes de produits dans chaque intervalle
    SELECT
        borne_inf,
        borne_sup,
        COUNT(*) AS nombre_de_produits
    FROM intervals
    JOIN produits
        ON produits.prix >= borne_inf
        AND (borne_sup IS NULL OR produits.prix < borne_sup)
    GROUP BY borne_inf, borne_sup
)
SELECT
    borne_inf,
    borne_sup,
```

```
    nombre_de_produits,
    -- Calcul du pourcentage arrondi à 2 décimales
    ROUND(nombre_de_produits * 100.0 / SUM(nombre_de_produits) OVER (), 2) AS
     ↪  pourcentage
FROM counts
ORDER BY borne_inf;

 borne_inf | borne_sup | nombre_de_produits | pourcentage
-----------+-----------+--------------------+-------------
      4.01 |      5.97 |              11218 |       24.94
      5.97 |      8.06 |              11534 |       25.64
      8.06 |     20.48 |              11481 |       25.52
     20.48 |     99.07 |              10636 |       23.65
     99.07 |           |                111 |        0.25
(5 lignes)
```

L'échantillonnage a permis d'avoir quatre lots répartis de manière équilibrée, mais pas parfaite. L'information est de suffisamment bonne qualité pour l'optimiseur, surtout lorsque la valeur de `default_statistics_target` est suffisamment élevée.

Comme pour `most_common_elems` et `most_common_elems_freqs`, la colonne `elem_count_histogram` de la vue `pg_stats` permet de calculer des histogrammes pour les colonnes de type tableau.

1.5.5 Valeurs nulles

```
-[ RECORD 1 ]----------+-------------------------------------------------
schemaname             | public
tablename              | clients
attname                | telephone_fixe
null_frac              | 0.98
```

- Estimation du nombre de valeur nulles dans la colonne

Cette statistique, colonne `null_frac` de la vue `pg_stats`, correspond au pourcentage de valeurs `NULL` dans l'échantillon considéré par `ANALYZE`. Elle est toujours calculée. Il n'y a pas de valeurs nulles dans l'exemple ci-dessus.

1.5.6 Corrélation physique

```
-[ RECORD 1 ]----------+-----------------------------------------------
schemaname             | public
tablename              | clients
attname                | date_inscription
correlation            | 1
```

- Varie de -1 à 1
- Indique si les données sont ordonnées dans les fichiers
- Arbitre la pertinence de l'usage d'un index
  - forte corrélation = peu de blocs à lire
  - faible corrélation = données dispersées

La colonne `correlation` de la vue `pg_stats` indique la corrélation entre l'ordre physique et l'ordre logique des valeurs de la colonne. Dans notre exemple, les données sont parfaitement triées sur le disque selon le champ `date_inscription`, la corrélation est donc de 1. Si les données sont physiquement triées, il y aura moins de blocs à récupérer sur le disque pour des valeurs proches ou identiques que si elles étaient totalement dispersées dans la table.

Dans l'exemple ci-dessous, on considère la table `produits` avec un champ `stock`. Les produits les plus anciens ont tendance à avoir davantage de stock que ceux récemment entrés en catalogue, la corrélation est donc proche de -1 :

```
SELECT schemaname, tablename, attname, correlation
FROM pg_stats
WHERE tablename = 'produits'
AND attname = 'stock' \gx

-[ RECORD 1 ]-----------
schemaname  | public
tablename   | produits
attname     | stock
correlation | -0.9380958
```

Si la corrélation physique est proche de 0, les lignes sont dispersées dans la table sans lien avec la valeur du champ Passer par un index est alors moins intéressant en terme de nombre de blocs à lire par ligne ramenée. Un parcours séquentiel a alors plus de chances d'être choisi. Par exemple :

```
SELECT schemaname, tablename, attname, correlation
FROM pg_stats
WHERE tablename = 'commandes'
AND attname = 'date_achat' \gx

-[ RECORD 1 ]-------------
schemaname  | public
tablename   | commandes
```

```
attname     | date_achat
correlation | -0.001096768
```

Si cette colonne `date_achat` est critique pour le métier, et que beaucoup de requêtes y font référence pour un tri, on peut trier explicitement la table selon ce champ avec la commande `CLUSTER` :

```
-- Création d'un index sur `date_achat``
CREATE INDEX idx_commandes_date ON commandes(date_achat);

-- CLUSTER
CLUSTER commandes USING idx_commandes_date;

-- ANALYZE
ANALYZE commandes;

-- Résultat
SELECT schemaname, tablename, attname, correlation
FROM pg_stats
WHERE tablename = 'commandes'
AND attname = 'date_achat' \gx

-[ RECORD 1 ]-----------
schemaname  | public
tablename   | commandes
attname     | date_achat
correlation | 1
```

Comparons les plans d'exécution avant et après `CLUSTER` :

```
-- Requête
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_achat BETWEEN '2023-06-01' AND '2023-06-30';

-- Plan avant CLUSTER (simplifié)
    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on commandes  (actual time=1.621..10.927 rows=8192)
   Heap Blocks: exact=541
   Buffers: shared hit=541 read=9
   -> Bitmap Index Scan on idx_commandes_date (actual time=1.475 rows=8192)
      Buffers: shared read=9
 Planning Time: 1.003 ms
 Execution Time: 11.662 ms -- TOTAL 12.665 ms

-- Plan après CLUSTER (simplifié)

    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_commandes_date on commandes (actual time=0.084..2.150
 ↪  rows=8192)
   Index Cond: (date_achat BETWEEN '2023-06-01' AND '2023-06-30')
   Buffers: shared hit=55
 Planning Buffers: shared hit=9
 Planning Time: 0.454 ms
 Execution Time: 2.812 ms  -- TOTAL 3.266 ms
```

Dans le premier cas, l'index est bien utilisé mais via un *Bitmap Index Scan*, qui référence les blocs et non directement chacune des 8136 lignes. Dans ce cas, 550 blocs ont dû être lus. Après la commande `CLUSTER`, les données sont physiquement triées sur disque selon le champ `date_achat`. L'*Index Scan* classique est alors choisi, et le temps d'exécution est réduit de 74 %. Dix fois moins de blocs ont été lus.

`CLUSTER` réécrit, défragmente et réindexe la table, et l'opération est totalement bloquante. Cette commande est donc très lourde. Elle est surtout utile en fin d'un batch ou d'un chargement de données.

La corrélation physique imposée par `CLUSTER` n'est pas forcément conservée par PostgreSQL. Au fur et à mesure que les lignes sont modifiées, elles peuvent se retrouver dans n'importe quelle partie vide de la table.

La corrélation peut être à `NULL` si le type de données de la colonne ne supporte pas l'opérateur de comparaison `<`.

1.5.7 Largeur moyenne des données

```
-[ RECORD 1 ]----------+-----------------------------------------------
schemaname             | public
tablename              | employes
attname                | date_embauche
avg_width              | 4
```

- Calculé pour les champs non `NULL` de taille variable (`text`, `json`, `jsonb`, binaires, etc.)
- Constant pour les autres types (`integer`, `boolean`, `char`, etc.)

La largeur moyenne en octets des éléments d'une colonne est indiquée dans le champ `avg_width` de la vue `pg_stats`. Elle est constante pour les colonnes dont le type est à taille fixe (`integer`, `boolean`, `char`, etc.). Dans le cas du type `char(n)`, il s'agit du nombre de caractères saisissables +1. Il est variable pour les autres (principalement `text`, `varchar`, `bytea`, `jsonb`…).

Dans le cas d'un champ avec des valeurs assez grandes (plusieurs kilooctets), `avg_width` est faussé, car le mécanisme TOAST[3] se déclenche. Les champs sont alors compressés, voire déportés dans une table annexe. `avg_width` calcule alors la longueur physique moyenne du champ dans la table, compressé ou pas, parfois réduite à un pointeur vers la table TOAST associé.

1.6 ÉCHANTILLONNAGE

1.6.1 Taille de l'échantillon

- Échantillon de 30 000 lignes par défaut
  - `default_statistics_target` = 100
  - × 300 lignes

Par défaut, un `ANALYZE` récupère 30 000 lignes d'une table. Cette valeur se calcule à partir du paramètre `default_statistics_target` (100 par défaut). La taille de l'échantillon est de 300 fois `default_statistics_target`.

Les statistiques générées à partir de cet échantillon sont bonnes si la table ne contient pas des millions de lignes. Au-delà, il arrive très souvent que la répartition des données dans une colonne soit régulière, et l'échantillon reste assez représentatif pour donner de bonnes estimations.

1.6.2 Modifier l'échantillon

```
-- Configurable pour chaque colonne
ALTER TABLE t ALTER COLUMN c SET STATISTICS 300 ;
-- Configurable pour chaque statistique étendue
ALTER STATISTICS nom SET STATISTICS valeur ;
```

- Échantillon plus important
  - plus précis dans certains cas
  - temps de planification plus long
- `ANALYZE` pour rafraîchir les statistiques

Si l'échantillon n'est plus assez représentatif, les estimations qui en découlent peuvent mener à de mauvais plans d'exécution. Un échantillon plus gros peut corriger ce souci. Pour cela, il faut augmenter la valeur du paramètre `default_statistics_target`.

Il est possible de configurer ce paramétrage colonne par colonne :

```
ALTER TABLE nom_table ALTER nom_colonne SET STATISTICS <valeur> ;
ANALYZE nom_table ;
```

Les statistiques seront alors plus précises, mais elles seront plus longues à calculer par `ANALYZE`, prendront plus de place sur le disque et en RAM, et demanderont plus de travail au planificateur pour

générer le plan optimal. Augmenter cette valeur n'a donc pas que des avantages : on évitera de dépasser 1000. Généralement, l'échantillonnage ne sera augmenté que pour certaines colonnes posant souci.

1.7 STATISTIQUES MULTIVARIÉES

1.7.1 Statistiques multivariées

- Pas par défaut
- `CREATE STATISTICS`
- Trois types de statistiques
  - nombre de valeurs distinctes
  - dépendances fonctionnelles
  - liste MCV
- `ANALYZE` après la création
- vue `pg_stat_ext`

Par défaut, la commande `ANALYZE` de PostgreSQL calcule des statistiques monocolonnes uniquement. Elle peut aussi calculer certaines statistiques multicolonnes. En effet, les valeurs des colonnes ne sont pas indépendantes et peuvent varier ensemble. Estimer correctement le nombre de lignes retournées par une combinaison de critères devient donc impossible sans connaître cette relation.

Pour cela, il est nécessaire de créer un objet statistique avec l'ordre SQL `CREATE STATISTICS`. Cet objet indique les colonnes concernées ainsi que le type de statistique souhaité.

PostgreSQL supporte trois types de statistiques pour ces objets :

- `ndistinct` pour le nombre de valeurs distinctes sur ces colonnes ;
- `dependencies` pour les dépendances fonctionnelles ;
- `mcv` pour une liste des valeurs les plus fréquentes.

Pour créer une statistique d'un certain type, la syntaxe est :

```
CREATE STATISTICS stat_services (<type>) ON champ1,champ2  FROM nom_table ;
```

mais on peut calculer les trois types à la fois avec cette syntaxe allégée (ce qui est pratique mais peut consommer inutilement du CPU) :

```
CREATE STATISTICS stat_services ON champ1,champ2  FROM nom_table ;
```

Dans tous les cas, cela peut permettre d'améliorer fortement les estimations de nombre de lignes, ce qui peut amener de meilleurs plans d'exécution.

1.7.2 Dépendances fonctionnelles

```
CREATE STATISTICS stat_services (dependencies)
  ON code_postal, ville, departement
  FROM services;
```

- Indique une dépendance fonctionnelle entre deux colonnes
- La valeur d'une colonne varie en fonction d'une autre

Prenons un exemple. On peut voir sur ces deux requêtes que les statistiques sont à jour :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris';
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..786.05 rows=10013 width=28)
                     (actual time=0.019..4.773 rows=10001 loops=1)
   Filter: ((localisation)::text = 'Paris'::text)
   Rows Removed by Filter: 30003
 Planning time: 0.863 ms
 Execution time: 5.289 ms
```

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE departement=75;
```

```
                              QUERY PLAN
------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..786.05 rows=10013 width=28)
                     (actual time=0.020..7.013 rows=10001 loops=1)
   Filter: (departement = 75)
   Rows Removed by Filter: 30003
 Planning time: 0.219 ms
 Execution time: 7.785 ms
```

Cela fonctionne bien, *ie* l'estimation du nombre de lignes (10 013) est très proche de la réalité (10 001) dans le cas spécifique où le filtre se fait sur une seule colonne. Par contre, si le filtre se fait sur le lieu Paris *et* le département 75, l'estimation diffère d'un facteur 4, à 2506 lignes :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris'
  AND departement=75;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=2506 width=28)
                     (actual time=0.032..7.081 rows=10001 loops=1)
   Filter: (((localisation)::text = 'Paris'::text) AND (departement = 75))
```

```
   Rows Removed by Filter: 30003
 Planning time: 0.257 ms
 Execution time: 7.767 ms
```

En fait, il y a une dépendance fonctionnelle entre ces deux colonnes (être dans le département 75 implique d'être à Paris), mais PostgreSQL ne le sait pas car ses statistiques sont monocolonnes par défaut. Pour avoir des statistiques sur les deux colonnes à la fois, il faut créer un objet statistique dédié :

```
CREATE STATISTICS stat_services_big (dependencies)
  ON localisation, departement
  FROM services_big;
```

Après création de l'objet, il ne faut pas oublier de calculer les statistiques :

```
ANALYZE services_big;
```

On peut consulter les statistiques générées, qui indiquent bien la corrélation entre les champs 3 et 4 de la table :

```
SELECT * FROM pg_stats_ext WHERE statistics_name='stat_services_big' \gx
```

```
-[ RECORD 1 ]----------+----------------------------------------
schemaname             | public
tablename              | services_big
statistics_schemaname  | public
statistics_name        | stat_services_big
statistics_owner       | postgres
attnames               | {localisation,departement}
exprs                  | ø
kinds                  | {f}
inherited              | f
n_distinct             | ø
dependencies           | {"3 => 4": 1.000000, "4 => 3": 1.000000}
most_common_vals       | ø
most_common_val_nulls  | ø
most_common_freqs      | ø
most_common_base_freqs | ø
```

Ceci fait, on peut de nouveau regarder les estimations :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris'
  AND departement=75;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=10038 width=28)
                     (actual time=0.008..6.249 rows=10001 loops=1)
   Filter: (((localisation)::text = 'Paris'::text) AND (departement = 75))
   Rows Removed by Filter: 30003
 Planning time: 0.121 ms
 Execution time: 6.849 ms
```

Cette fois, l'estimation (10 038 lignes) est beaucoup plus proche de la réalité (10 001). Cela ne change rien au plan choisi dans ce cas précis, mais dans certains cas la différence peut être énorme.

1.7.3 Valeurs distinctes (N-Distinct)

```
CREATE STATISTICS stat_services (ndistinct)
  ON couleur, cépage
  FROM vins;
```

- Indique que plusieurs colonnes sont liées
- Utile pour les clauses `GROUP BY`

Poursuivons l'exemple précédent et prenons le cas d'un regroupement, ici sur une seule colonne :

```
EXPLAIN (ANALYZE)
  SELECT localisation, COUNT(*)
  FROM   services_big
  GROUP BY localisation ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=886.06..886.10 rows=4 width=14)
          (actual time=12.925..12.926 rows=4 loops=1)
   Group Key: localisation
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=6)
                           (actual time=0.010..2.779 rows=40004 loops=1)
 Planning time: 0.162 ms
 Execution time: 13.033 ms
```

L'estimation du nombre de lignes pour un regroupement sur une colonne est très bonne.

À présent, testons avec un regroupement sur deux colonnes :

```
EXPLAIN (ANALYZE)
  SELECT localisation, departement, COUNT(*)
  FROM services_big
  GROUP BY localisation, departement;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=986.07..986.23 rows=16 width=18)
           (actual time=15.830..15.831 rows=4 loops=1)
   Group Key: localisation, departement
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=10)
                            (actual time=0.005..3.094 rows=40004 loops=1)
 Planning time: 0.102 ms
 Execution time: 15.860 ms
```

Là aussi, on constate un facteur d'échelle de 4 entre l'estimation (16 lignes) et la réalité (4). Et là aussi, un objet statistique peut fortement aider :

```
DROP STATISTICS IF EXISTS stat_services_big;
```

```
CREATE STATISTICS stat_services_big (ndistinct)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;
```

Les nouvelles statistiques indiquent 4 valeurs distinctes dans le regroupement des deux colonnes, du moins selon l'échantillon :

```
SELECT * FROM  pg_stats_ext WHERE statistics_name='stat_services_big' \gx
```

```
-[ RECORD 1 ]----------+--------------------------
schemaname             | public
tablename              | services_big
statistics_schemaname  | public
statistics_name        | stat_services_big
statistics_owner       | postgres
attnames               | {localisation,departement}
exprs                  | ø
kinds                  | {d}
inherited              | f
n_distinct             | {"3, 4": 4}
dependencies           | ø
most_common_vals       | ø
most_common_val_nulls  | ø
most_common_freqs      | ø
most_common_base_freqs | ø
```

```
EXPLAIN (ANALYZE)
  SELECT localisation, departement, COUNT(*)
  FROM services_big
  GROUP BY localisation, departement;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=986.07..986.11 rows=4 width=18)
          (actual time=14.351..14.352 rows=4 loops=1)
   Group Key: localisation, departement
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=10)
                           (actual time=0.013..2.786 rows=40004 loops=1)
 Planning time: 0.305 ms
 Execution time: 14.413 ms
```

L'estimation est bien meilleure grâce à l'ajout de statistiques N-Distinct spécifiques aux deux colonnes.

1.7.4 Most Common Values (MCV)

```
CREATE STATISTICS stat_services (mcv)
  ON couleur, cépage
  FROM vins;
```

- Même principe que pour une seule colonne
- Supporte les opérateurs d'inégalité `>`, `>`, `<=` et `>=`

Les statistiques multicolonnes de type MCV (*most common values*) fonctionnent de la même manière que pour une seule colonne. Les valeurs les plus fréquentes et leur fréquence d'occurrence sont calculées en fonction du nombre total de lignes et du contenu de l'échantillon.

Un des intérêts est de mieux estimer le nombre de lignes à partir d'un prédicat utilisant les opérations `<` et `>`. En voici un exemple :

```
DROP STATISTICS IF EXISTS stat_services_big;

EXPLAIN (ANALYZE)
  SELECT *
  FROM services_big
  WHERE localisation='Paris'
  AND  departement > 74 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
  Seq Scan on services_big  (cost=0.00..886.06 rows=2546 width=28)
                            (actual time=0.031..19.569 rows=10001 loops=1)
   Filter: ((departement > 74) AND ((localisation)::text = 'Paris'::text))
   Rows Removed by Filter: 30003
 Planning Time: 0.186 ms
 Execution Time: 21.403 ms
```

Il y a donc une erreur d'un facteur 4 (2 546 lignes estimées contre 10 001 réelles) que l'on peut corriger :

```
CREATE STATISTICS stat_services_big (mcv)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;
```

Les statistiques calculées sur l'échantillon sont celles-ci :

```
SELECT * FROM pg_stats_ext WHERE statistics_name='stat_services_big' \gx
```

```
-[ RECORD 1
 ↪  ]----------+-------------------------------------------------------------
schemaname             | public
tablename              | services_big
```

```
statistics_schemaname  | public
statistics_name        | stat_services_big
statistics_owner       | postgres
attnames               | {localisation,departement}
exprs                  | ø
kinds                  | {m}
inherited              | f
n_distinct             | ø
dependencies           | ø
most_common_vals       | { {Nantes,44},{Rennes,40},{Limoges,52},{Paris,75} }
most_common_val_nulls  | { {f,f},{f,f},{f,f},{f,f} }
most_common_freqs      | {0.2523,0.2497,0.2495,0.2485}
most_common_base_freqs |
 ↪  {0.06365529000000002,0.062350090000000004,0.06225025,0.06175225}
```

```
EXPLAIN (ANALYZE)
  SELECT *
  FROM services_big
  WHERE localisation='Paris'
  AND  departement > 74;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=10030 width=28)
                           (actual time=0.017..18.092 rows=10001 loops=1)
   Filter: ((departement > 74) AND ((localisation)::text = 'Paris'::text))
   Rows Removed by Filter: 30003
 Planning Time: 0.337 ms
 Execution Time: 18.907 ms
```

On peut créer les trois types de statistiques en un ordre :

```
CREATE STATISTICS stat_services_big
  ON localisation, departement
  FROM services_big;
```

1.8 STATISTIQUES ÉTENDUES

1.8.1 Statistiques sur les expressions

```
CREATE STATISTICS employe_big_extract
ON extract('year' FROM date_embauche) FROM employes_big;
```

- Pas créées par défaut
- Résout le problème des statistiques difficiles à estimer
- À partir de la v14 (index fonctionnel nécessaire avant)
- `ANALYZE` après la création
- Vue `pg_stat_ext_exprs`

À partir de la version 14, il est possible de créer un objet statistique sur des expressions.

Les statistiques sur des expressions permettent de résoudre le problème des estimations sur les résultats de fonctions ou d'expressions. C'est un problème récurrent et impossible à résoudre sans statistiques dédiées.

On voit dans cet exemple que les statistiques pour l'expression `extract('year' from data_embauche)` sont erronées :

```
EXPLAIN SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;

                        QUERY PLAN
-------------------------------------------------------------
 Gather  (cost=1000.00..9552.15 rows=2495 width=40)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
       (cost=0.00..8302.65 rows=1040 width=40)
         Filter: (date_part('year'::text,
                 (date_embauche)::timestamp without time zone)
                 = '2006'::double precision)
```

En effet, l'optimiseur n'a aucune idée des valeurs qui vont sortir de la fonction `extract`, même en ayant des statistiques sur `date_embauche` (c'est pourtant évident pour un humain ici, mais ce n'est pas le cas général). Le planificateur applique donc généralement un « forfait », ici 0,5 % de lignes satisfaisant le critère. (Quelques fonctions possèdent une « fonction support » annexe qui renseigne l'optimiseur sur la cardinalité[4]).

[4] https://docs.postgresql.fr/current/xfunc-optimization.html

L'ordre suivant crée des statistiques supplémentaires sur les résultats de l'expression. Les résultats sont calculés pour un échantilon des lignes et collectés en même temps que les statistiques monocolonnes habituelles. Il ne faut pas oublier de lancer manuellement la collecte la première fois :

```
CREATE STATISTICS employe_big_extract
    ON extract('year' FROM date_embauche) FROM employes_big;
ANALYZE employes_big;
```

Les estimations du plan sont désormais correctes :

```
EXPLAIN SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..12149.22 rows=498998 width=40)
   Filter: (EXTRACT(year FROM date_embauche) = '2006'::numeric)
```

Cet objet statistique apparaît dans `psql` dans un `\d employes_big` :

```
…
Objets statistiques :
    "public.employe_big_extract" ON EXTRACT(year FROM date_embauche) FROM
 ↪  employes_big
```

Avant la version 14, calculer ces statistiques est possible indirectement, en créant un index fonctionnel sur l'expression, ce qui entraîne aussi le calcul de statistiques dédiées lors du prochain `ANALYZE`. Or l'index fonctionnel n'a pas toujours d'intérêt : dans l'exemple précédent, presque toutes les dates d'embauche sont en 2006. L'index ne serait donc pas utilisé alors qu'il est peut-être lourd à créer et maintenir.

Ne pas confondre ces statistiques sur les résultats d'une fonction avec les données de la table avec les paramètres que l'on peut poser sur une fonction, comme `ROWS` (qui précise combien de lignes une fonction va retourner), ou `COSTS` (qui modifie le coût d'utilisation d'une fonction). Tous ces moyens sont complémentaires.

1.8.2 Catalogues pour les statistiques étendues

Vues disponibles :

- `pg_stats_ext`
- `pg_stats_ext_exprs` (pour les expressions, v14)

Les statistiques étendues sont stockées dans les tables `pg_statistic_ext` et `pg_statistic_ext_data` ; la vue `pg_stats_ext` permet de consulter ces informations de manière plus lisible.

```
SELECT * FROM pg_stats_ext \gx
```

```
-[ RECORD 1 ]----------+----------------------------------------
schemaname             | public
tablename              | employes_big
statistics_schemaname  | public
statistics_name        | employe_big_extract
statistics_owner       | postgres
attnames               |
exprs                  | {"EXTRACT(year FROM date_embauche)"}
kinds                  | {e}
inherited              | f
n_distinct             |
dependencies           |
most_common_vals       |
most_common_val_nulls  |
most_common_freqs      |
most_common_base_freqs |
-[ RECORD 2 ]----------+----------------------------------------
schemaname             | public
tablename              | services_big
statistics_schemaname  | public
statistics_name        | stat_services_big
statistics_owner       | postgres
attnames               | {localisation,departement}
exprs                  |
kinds                  | {m}
inherited              | f
n_distinct             |
dependencies           |
most_common_vals       | { {Paris,75},{Limoges,52},{Rennes,40},{Nantes,44} }
most_common_val_nulls  | { {f,f},{f,f},{f,f},{f,f} }
most_common_freqs      |
 ↪  {0.2512,0.25116666666666665,0.24886666666666668,0.24876666666666666}
most_common_base_freqs |
 ↪  {0.06310144,0.06308469444444444,0.061934617777777784,0.06188485444444444}
```

On voit qu'il n'y a pas d'informations détaillées sur les statistiques sur expression (première ligne). Elles sont visibles dans `pg_stats_ext_exprs` (à partir de la version 14) :

```
SELECT * FROM pg_stats_ext_exprs  \gx
```

```
-[ RECORD 1 ]----------+---------------------------------
schemaname             | public
tablename              | employes_big
statistics_schemaname  | public
statistics_name        | employe_big_extract
statistics_owner       | postgres
expr                   | EXTRACT(year FROM date_embauche)
inherited              | f
null_frac              | 0
avg_width              | 8
n_distinct             | 2
most_common_vals       | {2006}
most_common_freqs      | {0.9999667}
histogram_bounds       |
correlation            | 1
most_common_elems      |
```

```
most_common_elem_freqs |
elem_count_histogram   |
```

1.9 COLLECTE DES STATISTIQUES

1.9.1 Quand collecter les statistiques ?

- À la demande
  - `ANALYZE`
  - `VACUUM ANALYZE`
  - `vacuumdb --analyze / --analyze-only`
- Automatiquement
  - processus `autovacuum`
  - paramètres `autovacuum_analyze_*`
- Suivi :
  - `pg_stat_user_tables`

`ANALYZE` est l'ordre SQL permettant de mettre à jour les statistiques sur les données. Il peut être lancé de trois manières différentes :

1.9.1.1 Ordre ANALYZE

La commande `ANALYZE` lance l'échantillonnage :

```
ANALYZE nom_table1, nom_table2 ;  -- table par table
ANALYZE (VERBOSE) ;               -- toute la base, avec les détails
```

Un lancement manuel est conseillé après toute modification lourde, entre les étapes d'un batch, voire périodiquement.

Commande vacuumdb

La commande système `vacuumdb` s'exécute depuis le shell, et génère :

```
# Nettoyage (VACUUM) et analyze des tables
vacuumdb --analyze  --echo
# ANALYZE uniquement
vacuumdb --analyze-only --echo
```

autovacuum

Le processus `autovacuum` assure certaines opérations de maintenance des tables en tâche de fond, notamment le nettoyage des lignes mortes des tables et le rafraîchissement des statistiques sur les données. Nous en parlerons plus loin.

Suivi

La vue `pg_stat_user_tables` contient, entre autres et pour chaque table :

- `last_analyze` : date et heure de la dernière commande `ANALYZE` manuelle ou scriptée ;
- `last_autoanalyze` : date et heure du dernier `ANALYZE` lancé par le démon `autovacuum`.

```
SELECT relname, last_analyze, last_autoanalyze
FROM pg_stat_user_tables
WHERE relname = 'service_counters_94' \gx
```

```
-[ RECORD 1 ]-------+------------------------------
relname             | service_counters_94
last_analyze        | 2025-03-12 10:23:50.401407+01
last_autoanalyze    | 2025-03-11 23:49:18.162277+01
```

1.9.2 ANALYZE

```
ANALYZE; -- base entière
ANALYZE pgbench_tellers; -- une table
ANALYZE pgbench_accounts (abalance) ; -- une colonne
```

- Un échantillon de table → statistiques
- Table vide : conserve les anciennes statistiques
- Nouvelle table : valeur par défaut

Sans argument, l'ordre `ANALYZE` se fait sur la base complète. Si un ou plusieurs arguments sont donnés, ils doivent correspondre aux noms des tables à analyser (en les séparant par des virgules). Il est même possible d'indiquer les colonnes à traiter.

Cette instruction va exécuter les calculs de statistiques que nous avons évoqués plus haut. Elle ne va pas lire la table entière, mais seulement un échantillon, sur lequel chaque colonne sera traitée pour récupérer le pourcentage de valeurs `NULL`, les valeurs les plus fréquentes, leur fréquence, l'histogramme des valeurs… Toutes ces informations sont stockées dans le catalogue système nommé `pg_statistics`, accessible par la vue `pg_stats`, ou les tables `pg_stats_ext` et `pg_stats_ext_exprs`, comme vu précédemment.

Dans le cas d'une table vide, les anciennes statistiques sont conservées. S'il s'agit d'une nouvelle table, les statistiques sont initialement vides.
À partir de la version 14, lors de la planification, une table vide est bien considérée comme telle au niveau de son nombre de lignes, mais avec 10 blocs au minimum.
Pour les versions antérieures, une nouvelle table (nouvelle dans le sens `CREATE TABLE` mais aussi `VACUUM FULL` et `TRUNCATE`) n'est jamais considérée vide par l'optimiseur, qui utilise alors des valeurs par défaut dépendant de la largeur moyenne d'une ligne et d'un nombre arbitraire de blocs.

Jusque PostgreSQL 16 inclus, seuls le propriétaire de la table et le superutilisateur peuvent lancer `ANALYZE` sur une table. À partir de PostgreSQL 17, un utilisateur de maintenance dédié, sans droit de lecture sur les tables, peut lancer la commande, soit sur toutes les tables s'il a le rôle `pg_maintain`, soit sur les tables où on lui aura octroyé un `GRANT MAINTAIN`. Ce droit de maintenance couvre aussi `VACUUM`, `CLUSTER` et `REINDEX`, entre autres.

1.9.3 Fréquence d'analyse

- Dépend principalement de la fréquence des requêtes DML
- Le processus `autovacuum` fait l'`ANALYZE` mais…
  - pas sur les tables temporaires
  - pas assez rapidement parfois
- Planification au niveau système avec `cron`
  - `psql -c "ANALYZE"`
  - ou `vacuumdb --analyze-only`

Les statistiques doivent être mises à jour fréquemment. La fréquence exacte dépend surtout de la fréquence des requêtes d'insertion, de modification ou de suppression des lignes des tables. Planifier un `ANALYZE` tous les jours par prudence est envisageable.

Sans statistiques à jour, le choix du planificateur a un fort risque d'être mauvais. Il est donc important que les statistiques soient mises à jour régulièrement.

L'exécution périodique peut se faire avec cron (ou les tâches planifiées sous Windows) en utilisant l'option `--analyze-only` de `vacuumdb`. Ces deux ordres sont équivalents :

```
vacuumdb --analyze-only -t matable -d mabase
```

```
psql -c "ANALYZE matable" -d mabase
```

(En toute rigueur, `vacuumdb` ajoute une commande `VACUUM (ONLY_DATABASE_STATS)` depuis PostgreSQL 16 pour limiter la fragmentation des tables système, ce qui est généralement à conseiller.)

Le démon `autovacuum` effectue aussi des `ANALYZE`. Dans beaucoup de cas, il suffit de se reposer sur lui. L'`autovacuum` se déclenche quand un certain pourcentage de lignes ont été modifiées depuis le dernier rafraîchissement, soit 10 % par défaut. Ce seuil est très souvent réduit pour les grosses tables.

Pour les détails et le paramétrage de l'`autovacuum`, voir le module *M5 - VACUUM & autovacuum*[5].

Il faut connaître ses particularités liées aux statistiques :

- L'autovacuum a sa propre connexion à la base. Il ne peut donc pas voir les tables temporaires appartenant aux autres sessions, et ne sera donc pas capable de mettre à jour leurs statistiques. Il faut donc lancer manuellement `ANALYZE` sur les tables temporaires dans la session qui les a créées.
- Après une insertion ou une mise à jour massive, `autovacuum` ne va pas lancer un `ANALYZE` immédiatement. En effet, il ne cherche les tables à traiter que toutes les minutes (par défaut). Si, après la mise à jour massive, une requête est immédiatement exécutée, il y a de fortes chances qu'elle s'exécute avec des statistiques obsolètes. Il est préférable dans ce cas de lancer un `ANALYZE` manuel sur la ou les tables concernées juste après l'insertion ou la mise à jour massive. Ce cas est typique des batchs.
- Les grandes tables souffrent souvent de mises à jour trop peu fréquentes (par défaut, 10 % des lignes de la table doivent être modifié pour déclencher automatiquement un `ANALYZE`). Pour des mises à jour plus régulières, il est assez fréquent de réduire la valeur du paramètre `autovacuum_analyze_scale_factor` de la table.

[5] https://dali.bo/m5_html

1.10 PROBLÈMES LES PLUS COURANTS

- Statistiques obsolètes
- Opérations ponctuelles (batch, import de données)
- Dépendances fonctionnelles

Par défaut, PostgreSQL calcule les statistiques régulièrement grâce au processus `autovacuum`, une colonne à la fois. Ceci peut avoir ses limites lors de chargement en masse de données ou quand des colonnes sont fortement corrélées entre elles.

1.10.1 Statistiques obsolètes

Les statistiques sont-elles à jour ?

- Traitement lourd
  - faire tout de suite `ANALYZE`
- Table trop grosse
  - régler l'échantillonnage
  - régler l'autovacuum sur cette table
- Retard de mise à jour suite à crash ou restauration

Il est fréquent que les statistiques ne soient pas à jour. Il peut y avoir plusieurs causes.

Gros chargement :

Cela arrive souvent après le chargement massif d'une table ou une mise à jour massive sans avoir fait une nouvelle collecte des statistiques à l'issue de ces changements. En effet, bien qu'autovacuum soit présent, il peut se passer un certain temps entre le moment où le traitement est fait et le moment où autovacuum déclenche une collecte de statistiques. À fortiori si le traitement complet est imbriqué dans une seule transaction.

On utilisera l'ordre `ANALYZE table` pour déclencher explicitement la collecte des statistiques après un tel traitement. Notamment, un traitement batch devra comporter des ordres `ANALYZE` juste après les ordres SQL qui modifient fortement les données :

```
COPY table_travail FROM '/tmp/fichier.csv';
ANALYZE table_travail;
SELECT * FROM table_travail;
```

Volumétrie importante :

Un autre problème qui peut se poser avec les statistiques concerne les tables de très forte volumétrie. Dans certains cas, l'échantillon de données ramené par `ANALYZE` n'est pas assez précis pour donner à l'optimiseur de PostgreSQL une vision suffisamment juste des données. Il choisira alors de mauvais plans d'exécution.

Il est possible d'augmenter la taille de l'échantillon de données analysées, ainsi que la précision des statistiques, pour les colonnes où cela est important, à l'aide de cet ordre vu plus haut :

```
ALTER TABLE nom_table ALTER nom_colonne SET STATISTICS valeur;
```

Autre problème courant avec les grosses tables : l'`autovacuum` se base par défaut sur la proportion de lignes modifiées par rapport à celles existantes pour savoir s'il doit déclencher un `ANALYZE` (10 % par défaut). Au fil du temps, beaucoup de tables grossissent en accumulant des lignes statiques. À volume d'activité constante, les lignes actives représentent alors une proportion de plus en plus faible et l'`autovacuum` se déclenche de moins en moins souvent. Il est courant de descendre la valeur de `autovacuum_analyze_scale_factor` pour compenser. On peut aussi chercher à isoler les données statiques dans leur partition.

Perte des statistiques d'activité après un arrêt brutal :

Le fonctionnement du collecteur des statistiques d'activité implique qu'un arrêt brutal de PostgreSQL les réinitialisent. (Il s'agit des statistiques sur les lignes insérées ou modifiées, que l'on trouve notamment dans `pg_stat_user_tables`, pas des statistiques sur les données, qui sont bien préservées.) Elles ne sont pas directement utilisées par le planificateur, mais cette réinitialisation peut mener à un retard dans l'activité de l'autovacuum et la mise à jour des statistiques des données. Après un plantage, un arrêt en mode immédiat ou une restauration physique, il est donc conseillé de relancer un `ANALYZE` général (et même un `VACUUM` ensuite si possible).

1.10.2 Colonnes corrélées

```
SELECT * FROM corr1 WHERE c1=1 AND c2=1
```

- Si `c1 = 1` pour 20 % des lignes
- et `c2 = 2` pour 10 % des lignes
- Alors le planificateur calcule : 2 % des lignes (20 % × 10 %)
    - Mais en réalité ?
- Pour corriger :

```
CREATE STATISTICS corr1_c1_c2 ON c1,c2 FROM corr1 ;
```

Avec les statistiques par défaut, le planificateur sait que l'estimation pour `c1=1` est de 20 %

et que l'estimation pour `c2=2` est de 10 %. Par contre, il n'a aucune idée de l'estimation pour `c1=1 AND c2=2`. Faute de mieux, il multiplie les deux estimations et obtient 2 % (20 % × 10 %), soit environ 2000 lignes, au lieu de 0.

```
ANALYZE corr1;
EXPLAIN SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 2 ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=28.70..634.57 rows=1991 width=12)
   Recheck Cond: ((c1 = 1) AND (c2 = 2))
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..28.20 rows=1991 width=0)
         Index Cond: ((c1 = 1) AND (c2 = 2))
```

Nous avons vu que la création de statistiques multicolonnes n'est pas automatique, il faut créer un objet statistique avec l'ordre `CREATE STATISTICS` :

```
CREATE STATISTICS corr1_c1_c2 ON c1,c2 FROM corr1 ;
ANALYZE corr1 ;    -- ne pas oublier !
```

Dans ce cas précis, les meilleures statistiques permettent une meilleure estimation (une ligne), et donc de basculer du *Bitmap Scan* à un *Index Scan* plus léger :

```
 EXPLAIN SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 2 ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using corr1_c1_c2_idx on corr1  (cost=0.29..8.22 rows=1 width=12)
   Index Cond: ((c1 = 1) AND (c2 = 2))
```

1.11 CONCLUSION

- Un échantillonnage des données très efficace…
- …mais pas parfait !
- Il faut bien comprendre son fonctionnement
- Possibilité d'ajuster la collecte des statistiques si besoin

1.12 RÉFÉRENCES

- **Plongée profonde dans les plans d'exécution**
  - Frédéric Yhuel, PGSessions 17, février 2024
  - https://youtu.be/xneDEIzBw3I
- **A Deep Dive into Postgres Statistics**
  - Louise Grandjonc Leinweber, PGConf.EU, octobre 2024
  - https://youtu.be/ApAClPFJ_rU

1.12.1 Questions

N'hésitez pas, c'est le moment !

1.13 QUIZ

https://dali.bo/m6_quiz

1.14 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m6_solutions.

1.14.1 Préambule

Ce TP se base sur la configuration par défaut de PostgreSQL, sauf précision contraire.

- Préciser `\timing` dans `psql` pour afficher les temps d'exécution de la recherche.

- Pour rendre les plans plus lisibles, désactiver le JIT et le parallélisme :

```
SET jit TO off ;
SET max_parallel_workers_per_gather TO 0 ;
```

- Pour mettre en évidence les effets de cache, lancer plusieurs fois les requêtes. Dans `psql`, il est possible de les rappeler avec `\g`, ou la touche **flèche haut** du clavier.

Les valeurs (taille, temps d'exécution) varieront à cause de plusieurs critères :

- les machines sont différentes ;
- le jeu de données peut avoir partiellement changé depuis la rédaction du TP ;

1.14.2 Vue *pg_stats*

But : Comprendre les informations lisibles dans pg_stats

Pour cet exercice, nous allons étudier deux tables dans la base **hr** modélisant un service du personnel.

Le script (de 31 ko) de la base **hr** peut être récupéré et installé ainsi :

```
curl -kL https://dali.bo/tp_hr -o hr.sql
createdb hr          # Création de la base
psql -X1 hr < hr.sql
```

- Les statistiques ont-elles été calculées ? (utiliser la vue `pg_stat_user_tables`)

- Rafraîchir les statistiques sur les tables `employees` et `locations`.

Retrouvons les informations calculées par `ANALYZE` pour la table `employees`.

- Comparer le nombre réel de lignes dans la table `employees` avec l'estimation faite par `ANALYZE`. (utiliser la vue `pg_class`)

- Quel est le taux de valeurs nulles pour la colonne `commission_pct` ?

- Quelles sont les valeurs les plus fréquentes pour la colonne `salary` ?

- Afficher l'histogramme pour la colonne `salary`.

- Quelle est l'estimation du nombre de valeurs distinctes de `job_id` ? Cette estimation est-elle juste ? Comment l'améliorer ?

- Selon quelle colonne les données d'`employees` sont-elles le mieux ordonnées sur disque ?

La base **hr** dispose de nombreux commentaires sur les objets qui la composent. On peut les voir avec les commandes `\d+` et `\dt+` de psql. Ces commentaires sont de tailles variables et stockés dans la table `pg_description` du catalogue système.

- Quelle est la largeur moyenne de la colonne `description` de `pg_description` ?

1.14.3 Corrélation entre colonnes

But : Optimiser une requête avec corrélations

Nous allons utiliser deux tables listant des colis qui doivent être distribués dans des villes.

La base **correlations** (dump de 51 Mo pour 865 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_correlations -o correlations.dump
createdb correlations
pg_restore -d correlations correlations.dump
```

- Charger le dump. Ne pas oublier les opérations habituelles après un chargement.

Dans la table `villes`, on trouve les villes et leur code postal. Ces colonnes sont très fortement corrélées, mais pas identiques :

- plusieurs villes peuvent partager le même code postal ;
- une ville peut avoir plusieurs codes postaux ;
- des villes de départements différents ont le même nom, mais pas le même code postal.

- Activer la mesure des durées des I/O dans la session, désactiver le JIT et le parallélisme.

- Dans la requête suivante, quelle est la stratégie principale ?
- Est-elle efficace ?

```
-- Cette requête liste les colis d'une liste de villes précisées
EXPLAIN (ANALYZE,BUFFERS)
SELECT *
FROM   colis
WHERE  id_ville IN (
       SELECT  id_ville
       FROM    villes
       WHERE   localite = 'PARIS'
         AND   codepostal LIKE '75%'
);
```

- Quelles sont les volumétries attendues et obtenues ?
- Comparer avec un filtre uniquement sur la ville ou le département.
- Quel est le problème fondamental ?

- Tenter d'améliorer l'estimation avec `CREATE STATISTICS`.

- Créer une fonction SQL comportant les deux critères : les statistiques associées sont-elles justes ?

- Les statistiques améliorées mènent-elles à un résultat plus rapide ?

NB : Cet exercice sur les corrélations entre colonnes est malheureusement peu représentatif.

1.15 TRAVAUX PRATIQUES (SOLUTIONS)

1.15.1 Préambule

Ce TP se base sur la configuration par défaut de PostgreSQL, sauf précision contraire.

- Préciser `\timing` dans `psql` pour afficher les temps d'exécution de la recherche.

- Pour rendre les plans plus lisibles, désactiver le JIT et le parallélisme :

```
SET jit TO off ;
SET max_parallel_workers_per_gather TO 0 ;
```

- Pour mettre en évidence les effets de cache, lancer plusieurs fois les requêtes. Dans `psql`, il est possible de les rappeler avec `\g`, ou la touche **flèche haut** du clavier.

Les valeurs (taille, temps d'exécution) varieront à cause de plusieurs critères :

- les machines sont différentes ;
- le jeu de données peut avoir partiellement changé depuis la rédaction du TP ;

1.15.2 Vue pg_stats

Pour cet exercice, nous allons étudier deux tables dans la base **hr** modélisant un service du personnel.

Le script (de 31 ko) de la base **hr** peut être récupéré et installé ainsi :

```
curl -kL https://dali.bo/tp_hr -o hr.sql
createdb hr          # Création de la base
psql -X1 hr < hr.sql
```

- Les statistiques ont-elles été calculées ? (utiliser la vue `pg_stat_user_tables`)

La commande `\d+ pg_stat_user_tables` dans psql nous permet d'identifier les colonnes que nous devons chercher :

```
                                View "pg_catalog.pg_stat_user_tables"
      Column        |           Type           | Collation | Nullable | Default |
↪  Storage | ...
--------------------+--------------------------+-----------+----------+---------+--
↪  -------+----
 relid              | oid                      |           |          |         | plain   |
 schemaname         | name                     |           |          |         |
↪  plain   |
 relname            | name                     |           |          |         | plain
↪  |
```

```
 seq_scan            | bigint                   |           |          |         |
↪  plain   |
 seq_tup_read        | bigint                   |           |          |         |
↪  plain   |
 idx_scan            | bigint                   |           |          |         |
↪  plain   |
 idx_tup_fetch       | bigint                   |           |          |         |
↪  plain   |
 n_tup_ins           | bigint                   |           |          |         |
↪  plain   |
 n_tup_upd           | bigint                   |           |          |         |
↪  plain   |
 n_tup_del           | bigint                   |           |          |         |
↪  plain   |
 n_tup_hot_upd       | bigint                   |           |          |         |
↪  plain   |
 n_live_tup          | bigint                   |           |          |         |
↪  plain   |
 n_dead_tup          | bigint                   |           |          |         |
↪  plain   |
 n_mod_since_analyze | bigint                   |           |          |         |
↪  plain   |
 n_ins_since_vacuum  | bigint                   |           |          |         |
↪  plain   |
 last_vacuum         | timestamp with time zone |           |          |         |
↪  plain   |
 last_autovacuum     | timestamp with time zone |           |          |         |
↪  plain   |
 last_analyze        | timestamp with time zone |           |          |         |
↪  plain   |
 last_autoanalyze    | timestamp with time zone |           |          |         |
↪  plain   |
 vacuum_count        | bigint                   |           |          |         |
↪  plain   |
 autovacuum_count    | bigint                   |           |          |         |
↪  plain   |
 analyze_count       | bigint                   |           |          |         |
↪  plain   |
 autoanalyze_count   | bigint                   |           |          |         |
↪  plain   |
(...)
```

L'information se trouve dans les colonnes `last_analyze` et `last_autoanalyze`. La requête suivante nous permet de vérifier le calcul des statistiques pour chaque table :

```
SELECT relname, last_analyze, last_autoanalyze
FROM pg_stat_user_tables;
```

```
   relname   | last_analyze | last_autoanalyze
-------------+--------------+------------------
 job_history |              |
 countries   |              |
 departments |              |
 locations   |              |
 employees   |              |
 regions     |              |
```

```
 jobs        |              |
(7 rows)
```

Dans notre cas, les statistiques n'ont pas encore été calculées.

- Rafraîchir les statistiques sur les tables `employees` et `locations`.

Lançons la commande `ANALYZE` sur les tables `employees` et `locations` :

```
ANALYZE employees, locations;
ANALYZE
```

La vue `pg_stat_user_tables` nous permet de vérifier l'opération :

```
SELECT relname, last_analyze, last_autoanalyze
FROM pg_stat_user_tables;
```

```
   relname   |         last_analyze          |       last_autoanalyze
-------------+-------------------------------+-------------------------------
 job_history |                               |
 countries   |                               |
 departments |                               |
 locations   | 2024-12-17 08:42:00.108691+01 |
 employees   | 2024-12-17 08:42:00.107698+01 | 2024-12-17 08:39:04.767175+01
 regions     |                               |
 jobs        |                               |
(7 rows)
```

Dans notre cas, nous remarquons qu'*autoanalyze* est passé sur `employees` peu avant la collecte manuelle.

Losrqu'on consulte la vue `pg_stats` on remarque que les statistiques sont présentes pour employees, mais pas encore pour departments :

```
SELECT attname FROM pg_stats WHERE tablename='employees';
```

```
    attname
----------------
 employee_id
 first_name
 last_name
 email
 phone_number
 hire_date
 job_id
 salary
 commission_pct
 manager_id
 department_id
(11 rows)
```

```
SELECT attname FROM pg_stats WHERE tablename='departments';
```

```
 attname
```

```
---------
(0 row)
```

Retrouvons les informations calculées par `ANALYZE` pour la table `employees`.

> - Quel est le taux de valeurs nulles pour la colonne `commission_pct` ?

```
SELECT null_frac
FROM pg_stats
WHERE tablename='employees'
AND attname='commission_pct';

 null_frac
-----------
 0.6728972
(1 ligne)
```

> - Comparer le nombre réel de lignes dans la table `employees` avec l'estimation faite par `ANALYZE`. (utiliser la vue `pg_class`)

```
 SELECT reltuples FROM pg_class WHERE relname='employees';

 reltuples
-----------
       107
(1 row)
```

Le calcul exact se fait avec la fonction count() de `SELECT` :

```
SELECT count(*) FROM employees;
 count
-------
   107
(1 row)
```

La table étant petite, l'échantillon permet d'avoir une estimation exact de la quantité de lignes.

> - Quel est le taux **réel** de valeurs nulles pour la colonne `commission_pct` ?

```
SELECT
COUNT(*) AS count_total,
COUNT(*) - COUNT(commission_pct) AS count_null,
(COUNT(*) - COUNT(commission_pct))::float/COUNT(*)::float AS null_frac
FROM employees;
 count_total | count_null |     null_frac
-------------+------------+--------------------
         107 |         72 | 0.6728971962616822
(1 ligne)
```

Le taux réel de valeurs nulles pour la colonne est égal à l'estimation. Ce qui est normal car la table ne comporte que 107 lignes, soit moins que l'échantillon minimal par défaut de 30000.

> – Quelles sont les valeurs les plus fréquentes pour la colonne `salary` ?

```
SELECT
    unnest(most_common_vals::text::int[]) AS most_common_val,
    unnest(most_common_freqs) AS most_common_freq
FROM pg_stats
WHERE tablename = 'employees'
  AND attname = 'salary';
```

```
  most_common_val | most_common_freq
-----------------+------------------
             2500 |      0.056074765
             2600 |      0.037383176
             2800 |      0.037383176
             3100 |      0.037383176
             3200 |      0.037383176
             9000 |      0.037383176
            10000 |      0.037383176
             2900 |      0.028037382
             7000 |      0.028037382
             8000 |      0.028037382
             9500 |      0.028037382
            11000 |      0.028037382
            12000 |      0.028037382
             2200 |      0.018691588
             2400 |      0.018691588
             2700 |      0.018691588
             3000 |      0.018691588
             3300 |      0.018691588
             3600 |      0.018691588
             4200 |      0.018691588
             4800 |      0.018691588
             6000 |      0.018691588
             6200 |      0.018691588
             6500 |      0.018691588
             7500 |      0.018691588
             8200 |      0.018691588
            10500 |      0.018691588
            17000 |      0.018691588
(28 rows)
```

Le nombre de valeurs recensées est variable en fonction de la taille de l'échantillon et de la répartition des valeurs, la colonne des noms `last_name` recense moins de valeurs fréquentes dans `pg_stats`.

```
SELECT
    unnest(most_common_vals::text::text[]) AS most_common_val,
    unnest(most_common_freqs) AS most_common_freq
FROM pg_stats
WHERE tablename = 'employees'
  AND attname = 'last_name';
```

```
 most_common_val | most_common_freq
-----------------+------------------
```

```
 Cambrault       |      0.018691588
 Grant           |      0.018691588
 King            |      0.018691588
 Smith           |      0.018691588
 Taylor          |      0.018691588
(5 rows)
```

- Afficher l'histogramme pour la colonne `salary`.

```
WITH bounds AS (
    SELECT
        generate_series(1, array_length(histogram_bounds, 1) - 1) AS bucket_idx,
        histogram_bounds::text::numeric[] AS bounds
    FROM pg_stats
    WHERE tablename = 'employees' AND attname = 'salary'
)
SELECT
    bounds[bucket_idx] AS lower_bound,
    bounds[bucket_idx + 1] AS upper_bound,
    (SELECT COUNT(*)
     FROM employees
     WHERE salary >=  bounds[bucket_idx]
       AND salary < bounds[bucket_idx + 1]) AS count
FROM bounds;
```

- Quelle est l'estimation du nombre de valeurs distinctes de `commission_pct` ?

```
SELECT attname, n_distinct
FROM pg_stats
WHERE tablename='employees'
AND attname='commission_pct' ;

    attname     | n_distinct
----------------+------------
 commission_pct |          7
(1 row)
```

La valeur est supérieure à 0, ceci signifie que PostgreSQL estime que le nombre de valeurs distinctes pour `commission_pct` est décorrélé du nombre total de lignes dans la table.

- Quelle est l'estimation du nombre de valeurs distinctes de `last_name` ?

```
SELECT attname, n_distinct
FROM pg_stats
WHERE tablename='employees'
AND attname='last_name' ;

attname   | n_distinct
----------+-------------
 last_name | -0.95327103
(1 row)
```

Ici, la valeur est négative, ceci signifie que PostgreSQL estime que le nombre de valeurs distinctes de `last_name` est proportionnel au nombre total de lignes de la table. Ici le coefficient de proportionnalité est proche de 1.

- Quelle est l'estimation du nombre de valeurs distinctes de `job_id` ? Cette estimation est-elle juste ? Comment l'améliorer ?

```
SELECT attname, n_distinct
FROM pg_stats
WHERE tablename='employees'
AND attname='job_id' ;

attname  | n_distinct
----------+-------------
 job_id  | -0.17757009
(1 row)
```

Ici, PostgreSQL estime aussi que le nombre de valeurs distinctes de `job_id` est proportionnel au nombre total de lignes. Il faudrait augmenter l'échantillon pour qu'il comprenne que ce n'est pas le cas.

- Selon quelle colonne les données d'`employees` sont-elles le mieux ordonnées sur disque ?

```
SELECT attname, correlation FROM pg_stats WHERE tablename='employees';

    attname     | correlation
----------------+--------------
 department_id  |   0.09441016
 commission_pct |   -0.4140056
 employee_id    |            1
 first_name     |  0.061756697
 last_name      |  -0.10506672
 email          |  0.052459884
 phone_number   |  0.094104506
 hire_date      |   0.15537138
 job_id         |   0.12764749
 salary         | -0.030016262
 manager_id     |    0.5118422
(11 rows)
```

L'ordre des données sur disque correspond au tri par clé primaire `employee_id`.

La base **hr** dispose de nombreux commentaires sur les objets qui la composent. On peut les voir avec les commandes `\d+` et `\dt+` de psql. Ces commentaires sont de tailles variables et stockés dans la table `pg_description` du catalogue système.

- Quelle est la largeur moyenne de la colonne `description` de `pg_description` ?

```
SELECT attname, avg_width FROM pg_stats WHERE tablename='pg_description' AND
↪  attname='description';
```

```
   attname    | avg_width
--------------+-----------
 description |        25
(1 row)
```

Les descriptions (commentaires) présents dans l'instance ont une largeur moyenne de 25 octets.

Conclusion

Cet exercice nous a fait explorer les vues `pg_stats` et `pg_class`, qui révèlent les données essentielles collectées par PostgreSQL, comme la distribution des valeurs, le nombre de lignes ou la taille des tables. Ces informations sont au cœur des décisions de l'optimiseur de requêtes et indispensables pour diagnostiquer les performances et optimiser les requêtes.

1.15.3 Corrélation entre colonnes

- Charger le dump. Ne pas oublier les opérations habituelles après un chargement.

Si la base cible s'appelle par exemple **correlations** :

```
$ pg_restore -d correlations correlations.dump
$ vacuumdb --analyze correlations
```

- Activer la mesure des durées des I/O dans la session, désactiver le JIT et le parallélisme.

```
SET track_io_timing TO on;
SET jit TO off ;
SET max_parallel_workers_per_gather TO 0;
```

- Dans la requête suivante, quelle est la stratégie principale ?
- Est-elle efficace ?

```
-- Cette requête liste les colis d'une liste de villes précisées
EXPLAIN (ANALYZE,BUFFERS)
SELECT *
FROM   colis
WHERE  id_ville IN (
       SELECT  id_ville
       FROM    villes
       WHERE   localite = 'PARIS'
         AND   codepostal LIKE '75%'
);
```

Le plan est :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop          (cost=5.85..12897.76 rows=3093 width=16)
              (actual time=27.220..820.321 rows=170802 loops=1)
   Buffers: shared hit=52994 read=121189
   I/O Timings: read=303.505
   ->  Seq Scan on villes  (cost=0.00..1209.32 rows=17 width=8)
```

```
                  (actual time=27.078..29.278 rows=940 loops=1)
        Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
        Rows Removed by Filter: 54015
        Buffers: shared read=385
        I/O Timings: read=2.686
  ->  Bitmap Heap Scan on colis  (cost=5.85..685.73 rows=182 width=16)
                        (actual time=0.040..0.816 rows=182 loops=940)
        Recheck Cond: (id_ville = villes.id_ville)
        Heap Blocks: exact=170515
        Buffers: shared hit=52994 read=120804
        I/O Timings: read=300.819
        ->  Bitmap Index Scan on idx_colis_ville
                                    (cost=0.00..5.80 rows=182 width=0)
                          (actual time=0.018..0.018 rows=182 loops=940)
              Index Cond: (id_ville = villes.id_ville)
              Buffers: shared hit=2805 read=478
              I/O Timings: read=1.903
Planning Time: 1.389 ms
Execution Time: 828.882 ms
```

Le plan est un *Nested Loop*. Pour chacune des lignes dans `villes` (obtenues par un *Seq Scan*), une lecture de `colis` a lieu (par *Bitmap Heap Scan*). C'est une boucle extrêmement coûteuse : 940 parcours de `colis` (1 par `id_ville`).

De plus les tables et index sont volumineux par rapport au cache, il y a des appels au disque (ou plutôt au cache de l'OS) (indicateurs `read`). Ce problème peut se mitiger avec le temps, mais même de longs accès en mémoire cache sont à éviter.

- Quelles sont les volumétries attendues et obtenues ?
- Comparer avec un filtre uniquement sur la ville ou le département.
- Quel est le problème fondamental ?

Le nombre de lignes obtenues (170 802) est plus de 55 fois supérieur à celui attendu (3093). Le problème se propage depuis l'estimation fausse sur `villes`. PostgreSQL fait ce choix parce qu'il estime que la condition

```
localite ='PARIS' AND codepostal LIKE '75%'
```

va ramener 17 enregistrements. En réalité, elle en ramène 940, soit 50 fois plus. Pourquoi PostgreSQL fait-il cette erreur ?

Les volumétries impliquées sont :

```
SELECT
  COUNT(*) AS nb_villes,
  COUNT(*) FILTER (WHERE localite='PARIS') AS nb_paris,
  COUNT(*) FILTER (WHERE codepostal LIKE '75%') AS nb_75,
  COUNT(*) FILTER (WHERE localite='PARIS'
                   AND codepostal LIKE '75%') AS nb_paris_75
FROM villes;
```

```
 nb_villes | nb_paris | nb_75 | nb_paris_75
-----------+----------+-------+-------------
     54955 |      940 |   998 |         940
```

Les statistiques reproduisent à peu près cela (les chiffres peuvent varier légèrement entre des installations à cause du choix de l'échantillon statistique) :

```
EXPLAIN SELECT * FROM villes ;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..934.55 rows=54955 width=27)

EXPLAIN SELECT * FROM villes WHERE localite='PARIS';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1071.94 rows=995 width=27)
   Filter: (localite = 'PARIS'::text)

EXPLAIN SELECT * FROM villes WHERE codepostal LIKE '75%';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1071.94 rows=1042 width=27)
   Filter: (codepostal ~~ '75%'::text)
```

L'estimation de la combinaison des deux critères est bien fausse :

```
EXPLAIN SELECT * FROM villes WHERE localite='PARIS'
                               AND codepostal LIKE '75%';

                               QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1209.32 rows=18 width=27)
   Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
```

D'après les statistiques, `villes` contient 54 955 enregistrements, 995 contenant PARIS (presque 2 %), 1042 commençant par 75 (presque 2 %).

Il y a donc 2 % d'enregistrements vérifiant chaque critère (c'est normal, ils sont presque équivalents). PostgreSQL, ignorant qu'il n'y a que Paris dans le département 75, part de l'hypothèse que les colonnes ne sont pas liées, et qu'il y a donc 2 % de 2 % (soit environ 0,04 %) des enregistrements qui vérifient les deux.

Si on fait le calcul exact, PostgreSQL croit donc avoir `(995/54955)×(1042/54955)×54955` = 18,8 enregistrements qui vérifient le critère complet, ce qui est évidemment faux.

Et un plan portant uniquement sur Paris (ou le département 75) a une estimation de volumétrie exacte :

```
EXPLAIN
  SELECT *
  FROM   colis
  WHERE  id_ville IN (
         SELECT  id_ville
         FROM    villes
         WHERE   localite = 'PARIS'
);
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=1083.94..181388.84 rows=174687 width=16)
   Hash Cond: (colis.id_ville = villes.id_ville)
   ->  Seq Scan on colis  (cost=0.00..154053.11 rows=9999911 width=16)
   ->  Hash  (cost=1071.94..1071.94 rows=960 width=8)
         ->  Seq Scan on villes  (cost=0.00..1071.94 rows=960 width=8)
               Filter: (localite = 'PARIS'::text)
```

- Tenter d'améliorer l'estimation avec `CREATE STATISTICS`.

Cette fonctionnalité est apparue dans la version 10. Pour calculer les corrélations entre les deux colonnes en question, la syntaxe est :

```
CREATE STATISTICS villes_localite_codepostal ON localite,codepostal FROM villes ;
```

Le rafraîchissement n'est pas automatique :

```
ANALYZE villes ;
```

Le résultat est-il concluant ?

```
EXPLAIN
SELECT *
FROM   colis
WHERE  id_ville IN (
        SELECT  id_ville
        FROM    villes
        WHERE   localite = 'PARIS'
          AND   codepostal LIKE '75%'
);
```

La réponse est non :

```
 Nested Loop  (cost=5.85..13653.22 rows=3275 width=16)
   ->  Seq Scan on villes  (cost=0.00..1209.32 rows=18 width=8)
         Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
   ->  Bitmap Heap Scan on colis  (cost=5.85..689.50 rows=183 width=16)
         Recheck Cond: (id_ville = villes.id_ville)
         ->  Bitmap Index Scan on idx_colis_ville  (cost=0.00..5.81 rows=183 width=0)
               Index Cond: (id_ville = villes.id_ville)
```

Dans notre cas les statistiques étendues n'aident pas. Par contre, cela aurait fonctionné avec des départements au lieu des codes postaux, ce qui est un contournement possible.

Cette colonne supplémentaire peut être alimentée par trigger ou avec `GENERATED ALWAYS AS (left(codepostal,2))` à partir de la v12.

- Créer une fonction SQL comportant les deux critères : les statistiques associées sont-elles justes ?

On peut indexer sur une fonction des deux critères. C'est un pis-aller mais la seule solution sûre. PostgreSQL calculera des statistiques sur le résultat de cette fonction à partir de l'échantillon au lieu de les calculer indirectement.

```
CREATE FUNCTION test_ville (ville text,codepostal text) RETURNS text
IMMUTABLE LANGUAGE SQL as $$
SELECT ville || '-' || codepostal
$$ ;

CREATE INDEX idx_test_ville ON villes (test_ville(localite , codepostal));

ANALYZE villes;

EXPLAIN
    SELECT * FROM colis WHERE id_ville IN (
    SELECT id_ville
    FROM villes
    WHERE test_ville(localite,codepostal) LIKE 'PARIS-75%'
);
                                QUERY PLAN
--------------------------------------------------------------------------------
 Hash Join  (cost=1360.59..181664.68 rows=201980 width=16)
   Hash Cond: (colis.id_ville = villes.id_ville)
   ->  Seq Scan on colis  (cost=0.00..154052.48 rows=9999848 width=16)
   ->  Hash  (cost=1346.71..1346.71 rows=1110 width=8)
         ->  Seq Scan on villes  (cost=0.00..1346.71 rows=1110 width=8)
               Filter: (((localite || '-'::text) || codepostal)
                                      ~~ 'PARIS-75%'::text)
```

On constate qu'avec cette méthode il n'y a plus d'erreur d'estimation (1110 est proche du réel 960). Cette méthode est bien sûr pénible à utiliser, et ne doit donc être réservée qu'aux quelques rares requêtes au comportement pathologique. Quitte à modifier le code, la colonne `departement` évoquée plus haut est peut-être plus simple et claire.

> – Les statistiques améliorées mènent-elles à un résultat plus rapide ?

De manière générale, des statistiques à jour aident à avoir un meilleur plan. Mais cela va aussi dépendre de la machine et de son paramétrage ! Tout ce TP a été effectué avec les paramètres par défaut, destinés à une machine très modeste :

```
shared_buffers = 128MB
work_mem = 4MB
random_page_cost = 4
seq_page_cost = 1
effective_cache_size = 4GB
```

Avec cette configuration, un *Hash Join*, assez consommateur, sera choisi. Sur une machine avec un SSD (voire juste de bons disques, ou si l'OS joue le rôle de cache), ceci peut être moins rapide que le *Nested Loop* de la requête d'origine, car l'accès à un bloc de table isolé n'est guère plus coûteux qu'au sein d'un parcours de table. Pour un SSD, `random_page_cost` peut être passé à 1, et le *Nested Loop* a plus de chance de se produire.

Conclusion

Que peut-on conclure de cet exercice ?

- que la ré-écriture est souvent la meilleure des solutions : interrogez-vous toujours sur la façon dont vous écrivez vos requêtes, plutôt que de mettre en doute PostgreSQL **a priori** ;

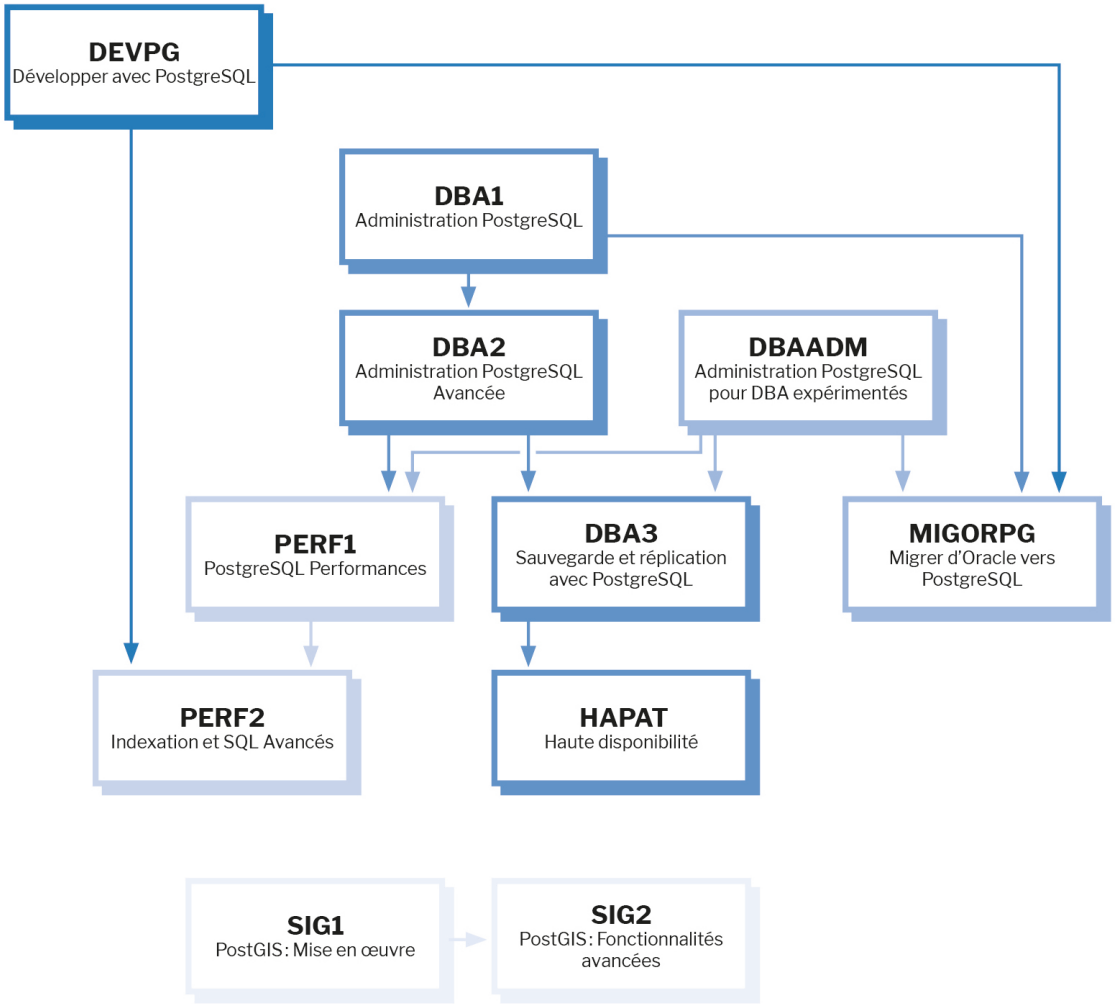
- que la ré-écriture de requête est souvent complexe
- néanmoins, surveillez un certain nombre de choses :
  - transtypages implicites suspects ;
  - jointures externes inutiles ;
  - sous-requêtes imbriquées ;
  - jointures inutiles (données constantes).

Les formations Dalibo

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

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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