Module J2

Comprendre EXPLAIN

24.04

Table des matières

Sur ce document

Formation	Module J2
Titre	Comprendre EXPLAIN
Révision	24.04
PDF	https://dali.bo/j2_pdf
EPUB	https://dali.bo/j2_epub
HTML	https://dali.bo/j2_html
Slides	https://dali.bo/j2_slides
TP	https://dali.bo/j2_tp
TP (solutions)	https://dali.bo/j2_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 :

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

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

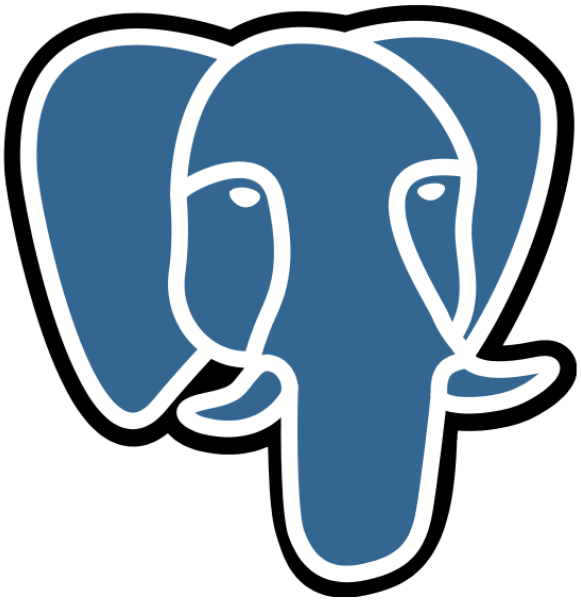
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Comprendre EXPLAIN



1.1 INTRODUCTION

- Le matériel, le système et la configuration sont importants pour les performances
- Mais il est aussi essentiel de se préoccuper des requêtes et de leurs performances

Face à un problème de performances, l'administrateur se retrouve assez rapidement face à une (ou plusieurs) requête(s). Une requête en soi représente très peu d'informations. Suivant la requête, des dizaines de plans peuvent être sélectionnés pour l'exécuter. Il est donc nécessaire de pouvoir trouver le plan d'exécution et de comprendre ce plan. Cela permet de mieux appréhender la requête et de mieux comprendre les pistes envisageables pour la corriger.

Ce qui suit se concentrera sur les plans d'exécution.

1.1.1 Au menu

- Exécution globale d'une requête
- Planificateur : utilité, statistiques et configuration
- `EXPLAIN`
- Nœuds d'un plan
- Outils

Nous ferons quelques rappels et approfondissements sur la façon dont une requête s'exécute globalement, et sur le planificateur : en quoi est-il utile, comment fonctionne-t-il, et comment le configurer.

Nous ferons un tour sur le fonctionnement de la commande `EXPLAIN` et les informations qu'elle fournit. Nous verrons aussi plus en détail l'ensemble des opérations utilisables par le planificateur, et comment celui-ci choisit un plan.

1.2 EXÉCUTION GLOBALE D'UNE REQUÊTE

- L'exécution peut se voir sur deux niveaux
  - niveau système
  - niveau SGBD
- De toute façon, composée de plusieurs étapes

L'exécution d'une requête peut se voir sur deux niveaux :

- ce que le système perçoit ;
- ce que le SGBD fait.

Une lenteur dans une requête peut se trouver dans l'un ou l'autre de ces niveaux.

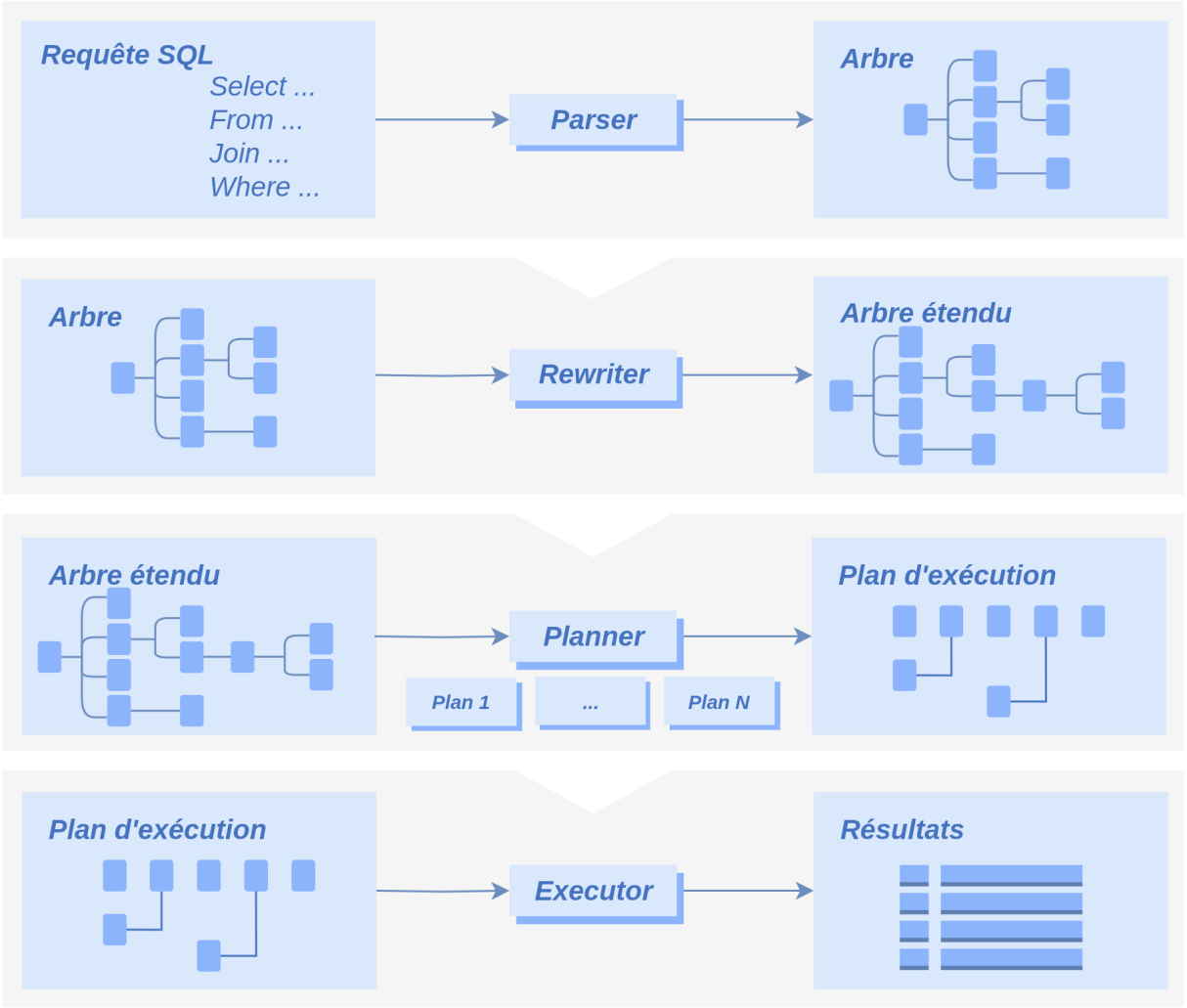
1.2.1 Niveau système

- Le client envoie une requête au serveur de bases de données
- Le serveur l'exécute
- Puis il renvoie le résultat au client

PostgreSQL est un système client-serveur. L'utilisateur se connecte via un outil (le client) à une base d'une instance PostgreSQL (le serveur). L'outil peut envoyer une requête au serveur, celui-ci l'exécute et finit par renvoyer les données résultant de la requête ou le statut de la requête.

Généralement, l'envoi de la requête est rapide. Par contre, la récupération des données peut poser problème si une grosse volumétrie est demandée sur un réseau à faible débit. L'affichage peut aussi être un problème (afficher une ligne sera plus rapide qu'afficher un million de lignes, afficher un entier est plus rapide qu'afficher un document texte de 1 Mo, etc.).

1.2.2 Traitement d'une requête



Lorsque le serveur récupère la requête, un ensemble de traitements est réalisé.

Tout d'abord, le *parser* va réaliser une analyse syntaxique de la requête.

Puis le *rewriter* va réécrire, si nécessaire, la requête. Pour cela, il prend en compte les règles, les vues non matérialisées et les fonctions SQL.

Si une règle demande de changer la requête, la requête envoyée est remplacée par la nouvelle.

Si une vue non matérialisée est utilisée, la requête qu'elle contient est intégrée dans la requête envoyée. Il en est de même pour une fonction SQL intégrable.

Ensuite, le *planner* va générer l'ensemble des plans d'exécutions. Il calcule le coût de chaque plan, puis il choisit le plan le moins coûteux, donc le plus intéressant.

Enfin, l'*executer* exécute la requête.

Pour cela, il doit commencer par récupérer les verrous nécessaires sur les objets ciblés. Une fois les verrous récupérés, il exécute la requête.

Une fois la requête exécutée, il envoie les résultats à l'utilisateur.

Plusieurs goulets d'étranglement sont visibles ici. Les plus importants sont :

- la planification (à tel point qu'il est parfois préférable de ne générer qu'un sous-ensemble de plans, pour passer plus rapidement à la phase d'exécution) ;
- la récupération des verrous (une requête peut attendre plusieurs secondes, minutes, voire heures, avant de récupérer les verrous et exécuter réellement la requête) ;
- l'exécution de la requête ;
- l'envoi des résultats à l'utilisateur.

Il est possible de tracer l'exécution des différentes étapes grâce aux options `log_parser_stats`, `log_planner_stats` et `log_executor_stats`. Voici un exemple complet :

- Mise en place de la configuration sur la session :

```
SET log_parser_stats TO on;
SET log_planner_stats TO on;
SET log_executor_stats TO on;
SET client_min_messages TO log;
```

- Exécution de la requête :

```
SELECT fonction, COUNT(*) FROM employes_big GROUP BY fonction ORDER BY fonction;
```

- Trace du *parser* :

```
LOG:  PARSER STATISTICS
DÉTAIL : ! system usage stats:
!       0.000026 s user, 0.000017 s system, 0.000042 s elapsed
!       [0.013275 s user, 0.008850 s system total]
!       17152 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/3 [0/575] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
LOG:  PARSE ANALYSIS STATISTICS
DÉTAIL : ! system usage stats:
!       0.000396 s user, 0.000263 s system, 0.000660 s elapsed
!       [0.013714 s user, 0.009142 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/32 [0/607] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace du *rewriter* :

```
LOG:  REWRITER STATISTICS
DÉTAIL : ! system usage stats:
!       0.000010 s user, 0.000007 s system, 0.000016 s elapsed
!       [0.013747 s user, 0.009165 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/1 [0/608] page faults/reclaims, 0 [0] swaps
```

```
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace du *planner* :

```
DÉTAIL : ! system usage stats:
!       0.000255 s user, 0.000170 s system, 0.000426 s elapsed
!       [0.014021 s user, 0.009347 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/25 [0/633] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace de l'*executer* :

```
LOG:  EXECUTOR STATISTICS
DÉTAIL : ! system usage stats:
!       0.044788 s user, 0.004177 s system, 0.131354 s elapsed
!       [0.058917 s user, 0.013596 s system total]
!       46268 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/468 [0/1124] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       4/16 [9/16] voluntary/involuntary context switches
```

- Résultat de la requête :

```
     fonction      | count
-------------------+--------
 Commercial        |      2
 Comptable         |      1
 Consultant        | 499005
 Développeur       |      2
 Directeur Général |      1
 Responsable       |      4
```

1.2.3 Exceptions

- Procédures stockées (appelées avec `CALL`)
- Requêtes DDL
- Instructions `TRUNCATE` et `COPY`
- Pas de réécriture, pas de plans d'exécution…
  - une exécution directe

Il existe quelques requêtes qui échappent à la séquence d'opérations présentées précédemment. Toutes les opérations DDL (modification de la structure de la base), les instructions `TRUNCATE` et

`COPY` (en partie) sont vérifiées syntaxiquement, puis directement exécutées. Les étapes de réécriture et de planification ne sont pas réalisées.

Le principal souci pour les performances sur ce type d'instructions est donc l'obtention des verrous et l'exécution réelle.

1.3 QUELQUES DÉFINITIONS

- Prédicat
  - filtre de la clause `WHERE`
  - conditions de jointure
- Sélectivité
  - % de lignes retournées après application d'un prédicat
- Cardinalité
  - nombre de lignes d'une table
  - nombre de lignes retournées après filtrages

Un prédicat est une condition de filtrage présente dans la clause `WHERE` d'une requête. Par exemple `colonne = valeur`. On parle aussi de prédicats de jointure pour les conditions de jointures présentes dans la clause `WHERE` ou suivant la clause `ON` d'une jointure.

La sélectivité est liée à l'application d'un prédicat sur une table. Elle détermine le nombre de lignes remontées par la lecture d'une relation suite à l'application d'une clause de filtrage, ou prédicat. Elle peut être vue comme un coefficient de filtrage d'un prédicat. La sélectivité est exprimée sous la forme d'un pourcentage. Pour une table de 1000 lignes, si la sélectivité d'un prédicat est de 10 %, la lecture de la table en appliquant le prédicat devrait retourner 10 % des lignes, soit 100 lignes.

La cardinalité représente le nombre de lignes d'une relation. En d'autres termes, la cardinalité représente le nombre de lignes d'une table ou de la sortie d'un nœud. Elle représente aussi le nombre de lignes retournées par la lecture d'une table après application d'un ou plusieurs prédicats.

1.3.1 Jeu de tests

- Tables
  - `services` : 4 lignes
  - `services_big` : 40 000 lignes
  - `employes` : 14 lignes
  - `employes_big` : ~500 000 lignes
- Index
  - `service*`.`num_service` (clés primaires)
  - `employes*`.`matricule` (clés primaires)
  - `employes*`.`date_embauche`
  - `employes_big`.`num_service` (clé étrangère)

Les deux volumétries différentes vont permettre de mettre en évidence certains effets.

1.3.2 Jeu de tests (schéma)

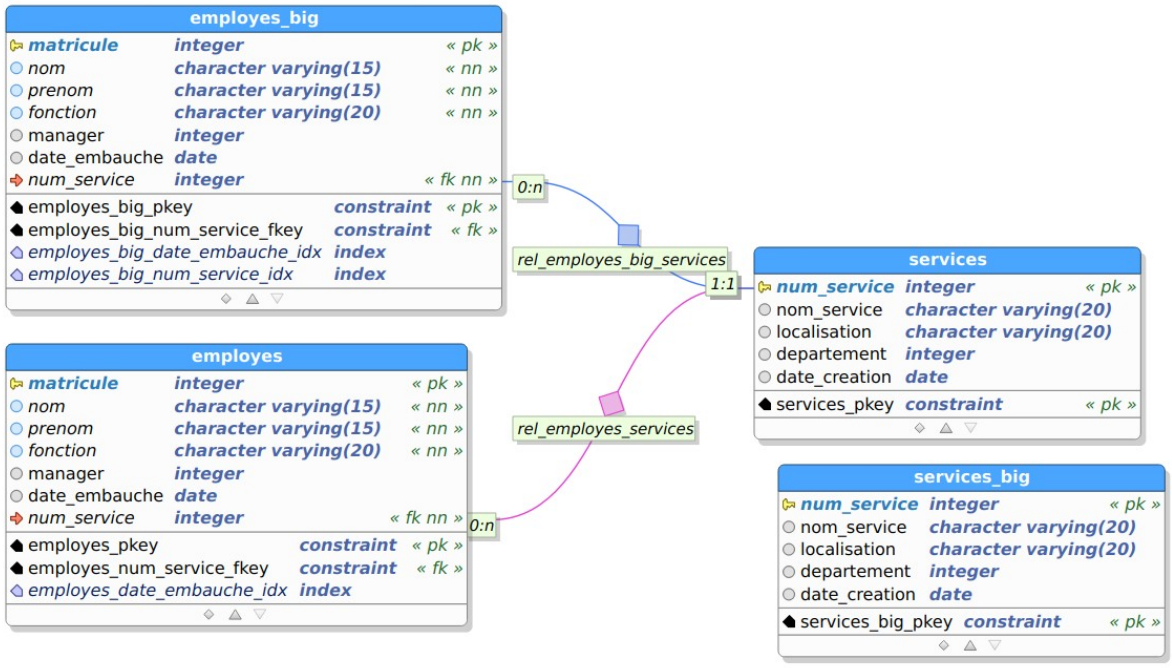


Figure 1/ .1: Tables employés & services

Les tables suivantes nous serviront d'exemple par la suite. Le script de création se télécharge et s'installe ainsi dans une nouvelle base **employes** :

```
curl -kL https://dali.bo/tp_employes_services -o employes_services.sql
createdb employes
psql employes < employes_services.sql
```

Les quelques tables occupent environ 80 Mo sur le disque.

```
-- suppression des tables si elles existent

DROP TABLE IF EXISTS services CASCADE;
DROP TABLE IF EXISTS services_big CASCADE;
DROP TABLE IF EXISTS employes CASCADE;
DROP TABLE IF EXISTS employes_big CASCADE;

-- définition des tables

CREATE TABLE services (
    num_service serial PRIMARY KEY,
    nom_service character varying(20),
    localisation character varying(20),
    departement integer,
    date_creation date
);
```

```
CREATE TABLE services_big (
    num_service serial PRIMARY KEY,
    nom_service character varying(20),
    localisation character varying(20),
    departement integer,
    date_creation date
);

CREATE TABLE employes (
    matricule     serial primary key,
    nom           varchar(15) not null,
    prenom        varchar(15) not null,
    fonction      varchar(20) not null,
    manager       integer,
    date_embauche date,
    num_service   integer not null references services (num_service)
);

CREATE TABLE employes_big (
    matricule     serial primary key,
    nom           varchar(15) not null,
    prenom        varchar(15) not null,
    fonction      varchar(20) not null,
    manager       integer,
    date_embauche date,
    num_service   integer not null references services (num_service)
);

-- ajout des données

INSERT INTO services
VALUES
    (1, 'Comptabilité', 'Paris', 75, '2006-09-03'),
    (2, 'R&D', 'Rennes', 40, '2009-08-03'),
    (3, 'Commerciaux', 'Limoges', 52, '2006-09-03'),
    (4, 'Consultants', 'Nantes', 44, '2009-08-03');

INSERT INTO services_big (nom_service, localisation, departement, date_creation)
VALUES
    ('Comptabilité', 'Paris', 75, '2006-09-03'),
    ('R&D', 'Rennes', 40, '2009-08-03'),
    ('Commerciaux', 'Limoges', 52, '2006-09-03'),
    ('Consultants', 'Nantes', 44, '2009-08-03');

INSERT INTO services_big (nom_service, localisation, departement, date_creation)
  SELECT s.nom_service, s.localisation, s.departement, s.date_creation
  FROM services s, generate_series(1, 10000);

INSERT INTO employes VALUES
    (33, 'Roy', 'Arthur', 'Consultant', 105, '2000-06-01', 4),
    (81, 'Prunelle', 'Léon', 'Commercial', 97, '2000-06-01', 3),
    (97, 'Lebowski', 'Dude', 'Responsable', 104, '2003-01-01', 3),
    (104, 'Cruchot', 'Ludovic', 'Directeur Général', NULL, '2005-03-06', 3),
    (105, 'Vacuum', 'Anne-Lise', 'Responsable', 104, '2005-03-06', 4),
    (119, 'Thierrie', 'Armand', 'Consultant', 105, '2006-01-01', 4),
```

```
    (120, 'Tricard', 'Gaston', 'Développeur', 125, '2006-01-01', 2),
    (125, 'Berlicot', 'Jules', 'Responsable', 104, '2006-03-01', 2),
    (126, 'Fougasse', 'Lucien', 'Comptable', 128, '2006-03-01', 1),
    (128, 'Cruchot', 'Joséphа', 'Responsable', 105, '2006-03-01', 1),
    (131, 'Lareine-Leroy', 'Émilie', 'Développeur', 125, '2006-06-01', 2),
    (135, 'Brisebard', 'Sylvie', 'Commercial', 97, '2006-09-01', 3),
    (136, 'Barnier', 'Germaine', 'Consultant', 105, '2006-09-01', 4),
    (137, 'Pivert', 'Victor', 'Consultant', 105, '2006-09-01', 4);

-- on copie la table employes
INSERT INTO employes_big SELECT * FROM employes;

-- duplication volontaire des lignes d'un des employés
INSERT INTO employes_big
  SELECT i, nom,prenom,fonction,manager,date_embauche,num_service
  FROM employes_big,
  LATERAL generate_series(1000, 500000) i
  WHERE matricule=137;

-- création des index
CREATE INDEX ON employes(date_embauche);
CREATE INDEX ON employes_big(date_embauche);
CREATE INDEX ON employes_big(num_service);

-- calcul des statistiques sur les nouvelles données
VACUUM ANALYZE;
```

1.3.3 Requête étudiée

```
SELECT matricule, nom, prenom, nom_service, fonction, localisation
FROM   employes emp
JOIN   services ser ON (emp.num_service = ser.num_service)
WHERE  ser.localisation = 'Nantes';
```

Cette requête nous servira d'exemple. Elle permet de déterminer les employés basés à Nantes et pour résultat :

```
 matricule |   nom    |  prenom   | nom_service |  fonction   | localisation
-----------+----------+-----------+-------------+-------------+--------------
        33 | Roy      | Arthur    | Consultants | Consultant  | Nantes
       105 | Vacuum   | Anne-Lise | Consultants | Responsable | Nantes
       119 | Thierrie | Armand    | Consultants | Consultant  | Nantes
       136 | Barnier  | Germaine  | Consultants | Consultant  | Nantes
       137 | Pivert   | Victor    | Consultants | Consultant  | Nantes
```

En fonction du cache, elle dure de 1 à quelques millisecondes.

1.3.4 Plan de la requête étudiée

L'objet de ce module est de comprendre son plan d'exécution :

```
Hash Join  (cost=1.06..2.28 rows=4 width=48)
  Hash Cond: (emp.num_service = ser.num_service)
  ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
  ->  Hash  (cost=1.05..1.05 rows=1 width=21)
        ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
              Filter: ((localisation)::text = 'Nantes'::text)
```

La directive `EXPLAIN` permet de connaître le plan d'exécution d'une requête. Elle permet de savoir par quelles étapes va passer le SGBD pour répondre à la requête.

Ce plan montre une jointure par hachage. La table `services` est parcourue intégralement (*Seq Scan*), mais elle est filtrée sur le critère sur « Nantes ».

Un *hash* de la colonne `num_service` des lignes résultantes de ce filtrage est effectué, et comparé aux valeurs rencontrées lors d'un parcours complet de `employes`.

S'affichent également les coûts estimés des opérations et le nombre de lignes que PostgreSQL s'attend à trouver à chaque étape.

1.4 PLANIFICATEUR

Rappels :

- SQL est un langage déclaratif
- Planificateur : trouver le meilleur plan
- Énumère tous les plans d'exécution possible
  - tous ou presque…
- Statistiques + configuration + règles → coût
- Coût le plus bas = meilleur plan

Le but du planificateur est assez simple. Pour une requête, il existe de nombreux plans d'exécution possibles. Il va donc tenter d'énumérer tous les plans d'exécution possibles ; même si leur nombre devient vite colossal dans une requête complexe : chaque table peut être accédée selon différents plans, selon l'un ou l'autre critère ou une combinaison, les algorithmes de jointure possibles sont multiples, etc.

Lors de cette énumération des différents plans, il calcule leur coût. Cela lui permet d'en ignorer certains alors qu'ils sont incomplets si leur plan d'exécution est déjà plus coûteux que les autres. Pour calculer le coût, il dispose d'informations sur les données (des statistiques), d'une configuration (réalisée par l'administrateur de bases de données) et d'un ensemble de règles inscrites en dur.

À la fin de l'énumération et du calcul de coût, il ne lui reste plus qu'à sélectionner le plan qui a le plus petit coût, à priori celui qui sera le plus rapide pour la requête demandée.

Le coût d'un plan est une valeur calculée sans unité ni signification physique.

1.4.1 Règles

- Règle 1 : récupérer le bon résultat
- Règle 2 : le plus rapidement possible
  - en minimisant les opérations disques
  - en préférant les lectures séquentielles
  - en minimisant la charge CPU
  - en minimisant l'utilisation de la mémoire

Le planificateur suit deux règles :

- il doit récupérer le bon résultat : un résultat rapide mais faux n'a aucun intérêt ;
- il doit le récupérer le plus rapidement possible.

Cette deuxième règle lui impose de minimiser l'utilisation des ressources : en tout premier lieu les opérations disques vu qu'elles sont les plus coûteuses, mais aussi la charge CPU (charge des CPU utilisés et nombre de CPU utilisés) et l'utilisation de la mémoire.

Dans le cas des opérations disques, s'il doit en faire, il doit souvent privilégier les opérations séquentielles aux dépens des opérations aléatoires (qui demandent un déplacement de la tête de disque, opération la plus coûteuse sur les disques magnétiques).

1.4.2 Outils de l'optimiseur

- L'optimiseur s'appuie sur :
  - un mécanisme de calcul de coûts
  - des statistiques sur les données
  - le schéma de la base de données

Pour déterminer le chemin d'exécution le moins coûteux, l'optimiseur devrait connaître précisément les données mises en œuvre dans la requête, les particularités du matériel et la charge en cours sur ce matériel. Cela est impossible. Ce problème est contourné en utilisant deux mécanismes liés l'un à l'autre :

- un mécanisme de calcul de coût de chaque opération ;
- des statistiques sur les données.

Pour quantifier la charge nécessaire pour répondre à une requête, PostgreSQL utilise un mécanisme de coût. Il part du principe que chaque opération a un coût plus ou moins important. Les statistiques sur les données permettent à l'optimiseur de requêtes de déterminer assez précisément la répartition des valeurs d'une colonne d'une table, sous la forme d'histogramme. Il dispose encore d'autres informations comme la répartition des valeurs les plus fréquentes, le pourcentage de `NULL`, le nombre de valeurs distinctes, etc.

Toutes ces informations aideront l'optimiseur à déterminer la sélectivité d'un filtre (prédicat de la clause `WHERE`, condition de jointure) et donc la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué. Enfin, l'optimiseur s'appuie sur le schéma de la base de données afin de déterminer différents paramètres qui entrent dans le calcul du plan d'exécution : contrainte d'unicité sur une colonne, présence d'une contrainte `NOT NULL`, etc.

1.4.3 Optimisations

- À partir du modèle de données
  - suppression de jointures externes inutiles
- Transformation des sous-requêtes
  - certaines sous-requêtes transformées en jointures
  - ex : `critere IN (SELECT ...)`
- Appliquer les prédicats le plus tôt possible
  - réduit le jeu de données manipulé
  - CTE : barrière avant la v12 !
- Intègre le code des fonctions SQL simples (*inline*)
  - évite un appel de fonction coûteux

Suppression des jointures externes inutiles

À partir du modèle de données et de la requête soumise, l'optimiseur de PostgreSQL va pouvoir déterminer si une jointure externe n'est pas utile à la production du résultat.

Sous certaines conditions, PostgreSQL peut supprimer des jointures externes, à condition que le résultat ne soit pas modifié. Dans l'exemple suivant, il ne sert à rien d'aller consulter la table `services` (ni données à récupérer, ni filtrage à faire, et même si la table est vide, le `LEFT JOIN` ne provoquera la disparition d'aucune ligne) :

```
EXPLAIN
  SELECT e.matricule, e.nom, e.prenom
  FROM employes e
  LEFT JOIN services s
    ON (e.num_service = s.num_service)
  WHERE e.num_service = 4 ;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=19)
   Filter: (num_service = 4)
```

Toutefois, si le prédicat de la requête est modifié pour s'appliquer sur la table `services`, la jointure est tout de même réalisée, puisqu'on réalise un test d'existence sur cette table `services` :

```
EXPLAIN
  SELECT e.matricule, e.nom, e.prenom
  FROM employes e
```

```
  LEFT JOIN services s
    ON (e.num_service = s.num_service)
  WHERE s.num_service = 4;

                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.00..2.27 rows=5 width=19)
   ->  Seq Scan on services s  (cost=0.00..1.05 rows=1 width=4)
         Filter: (num_service = 4)
   ->  Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=23)
         Filter: (num_service = 4)
```

Transformation des sous-requêtes

Certaines sous-requêtes sont transformées en jointure :

```
EXPLAIN
  SELECT *
  FROM employes emp
  JOIN (SELECT * FROM services WHERE num_service = 1) ser
    ON (emp.num_service = ser.num_service) ;

                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.00..2.25 rows=2 width=64)
   ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=21)
         Filter: (num_service = 1)
   ->  Seq Scan on employes emp  (cost=0.00..1.18 rows=2 width=43)
         Filter: (num_service = 1)
```

La sous-requête `ser` a été remontée dans l'arbre de requête pour être intégrée en jointure.

Application des prédicats au plus tôt

Lorsque cela est possible, PostgreSQL essaye d'appliquer les prédicats au plus tôt :

```
EXPLAIN
  SELECT MAX(date_embauche)
  FROM (SELECT * FROM employes WHERE num_service = 4) e
  WHERE e.date_embauche < '2006-01-01' ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=1.21..1.22 rows=1 width=4)
   ->  Seq Scan on employes  (cost=0.00..1.21 rows=2 width=4)
         Filter: ((date_embauche < '2006-01-01'::date) AND (num_service = 4))
```

Les deux prédicats `num_service = 4` et `date_embauche < '2006-01-01'` ont été appliqués en même temps, réduisant ainsi le jeu de données à considérer dès le départ. C'est généralement une bonne chose.

Mais en cas de problème, il est possible d'utiliser une CTE matérialisée (*Common Table Expression*, clause `WITH … AS MATERIALIZED (…)`) pour bloquer cette optimisation et forcer PostgreSQL à exécuter le contenu de la requête en premier[1]. En versions 12 et ultérieures, une CTE est par défaut non

[1] https://docs.postgresql.fr/current/queries-with.html#QUERIES-WITH-CTE-MATERIALIZATION

matérialisée et donc intégrée avec le reste de la requête (du moins dans les cas simples comme ci-dessus), comme une sous-requête. On retombe exactement sur le plan précédent :

```
-- v12 : CTE sans MATERIALIZED (comportement par défaut)
EXPLAIN
  WITH e AS ( SELECT * FROM employes WHERE num_service = 4 )
  SELECT MAX(date_embauche)
  FROM e
  WHERE e.date_embauche < '2006-01-01';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=1.21..1.22 rows=1 width=4)
   ->  Seq Scan on employes  (cost=0.00..1.21 rows=2 width=4)
         Filter: ((date_embauche < '2006-01-01'::date) AND (num_service = 4))
```

Pour recréer la « barrière d'optimisation », il est nécessaire d'ajouter le mot-clé `MATERIALIZED` :

```
-- v12 : CTE avec MATERIALIZED
EXPLAIN
  WITH e AS MATERIALIZED ( SELECT * FROM employes WHERE num_service = 4 )
  SELECT MAX(date_embauche)
  FROM e
  WHERE e.date_embauche < '2006-01-01';

                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=1.29..1.30 rows=1 width=4)
   CTE e
     ->  Seq Scan on employes  (cost=0.00..1.18 rows=5 width=43)
           Filter: (num_service = 4)
   ->  CTE Scan on e  (cost=0.00..0.11 rows=2 width=4)
         Filter: (date_embauche < '2006-01-01'::date)
```

La CTE est alors intégralement exécutée avec son filtre propre, avant que le deuxième filtre soit appliqué dans un autre nœud. Jusqu'en version 11 incluse, ce dernier comportement était celui par défaut, et les CTE étaient une source fréquente de problèmes de performances.

Function inlining

Voici deux fonctions, la première écrite en SQL, la seconde en PL/pgSQL :

```
CREATE OR REPLACE FUNCTION add_months_sql(mydate date, nbrmonth integer)
  RETURNS date AS
$BODY$
SELECT ( mydate + interval '1 month' * nbrmonth )::date;
$BODY$
  LANGUAGE SQL;

CREATE OR REPLACE FUNCTION add_months_plpgsql(mydate date, nbrmonth integer)
  RETURNS date AS
$BODY$
 BEGIN RETURN ( mydate + interval '1 month' * nbrmonth ); END;
$BODY$
  LANGUAGE plpgsql;
```

Si l'on utilise la fonction écrite en PL/pgSQL, on retrouve l'appel de la fonction dans la clause `Filter` du plan d'exécution de la requête :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes
  WHERE date_embauche = add_months_plpgsql(now()::date, -1);

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes (actual time=0.354..0.354 rows=0 loops=1)
   Filter: (date_embauche = add_months_plpgsql((now())::date, '-1'::integer))
   Rows Removed by Filter: 14
   Buffers: shared hit=1
 Planning Time: 0.199 ms
 Execution Time: 0.509 ms
```

Effectivement, PostgreSQL ne sait pas intégrer le code des fonctions PL/pgSQL dans ses plans d'exécution.

En revanche, en utilisant la fonction écrite en langage SQL, la définition de la fonction est directement intégrée dans la clause de filtrage de la requête :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes
  WHERE date_embauche = add_months_sql(now()::date, -1);

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes (actual time=0.014..0.014 rows=0 loops=1)
   Filter: (date_embauche = (((now())::date + '-1 mons'::interval))::date)
   Rows Removed by Filter: 14
   Buffers: shared hit=1
 Planning Time: 0.111 ms
 Execution Time: 0.027 ms
```

Le temps d'exécution a été divisé presque par 20 sur ce jeu de données très réduit, montrant l'impact de l'appel d'une fonction dans une clause de filtrage.

Dans les deux cas ci-dessus, PostgreSQL a négligé l'index sur `date_embauche` : la table ne faisait de toute façon qu'un bloc ! Mais pour de plus grosses tables, l'index sera nécessaire, et la différence entre fonctions PL/pgSQL et SQL devient alors encore plus flagrante. Avec la même requête sur la table `employes_big`, beaucoup plus grosse, on obtient ceci :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes_big
  WHERE date_embauche = add_months_plpgsql(now()::date, -1);

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes_big (actual time=464.531..464.531 rows=0 loops=1)
   Filter: (date_embauche = add_months_plpgsql((now())::date, '-1'::integer))
   Rows Removed by Filter: 499015
```

```
   Buffers: shared hit=4664
 Planning:
   Buffers: shared hit=61
 Planning Time: 0.176 ms
 Execution Time: 465.848 ms
```

La fonction portant sur une « boîte noire », l'optimiseur n'a comme possibilité que le parcours complet de la table.

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes_big
  WHERE date_embauche = add_months_sql(now()::date, -1);

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
                                    (actual time=0.016..0.016 rows=0 loops=1)
   Index Cond: (date_embauche = (((now())::date + '-1 mons'::interval))::date)
   Buffers: shared hit=3
 Planning Time: 0.143 ms
 Execution Time: 0.032 ms
```

La fonction SQL est intégrée, l'optimiseur voit le critère dans `date_embauche` et peut donc se poser la question de l'utiliser (et ici, la réponse est oui : 3 blocs contre 4664, tous présents dans le cache dans cet exemple).

D'où une exécution beaucoup plus rapide.

1.4.4 Décisions

L'optimiseur doit choisir :

- Stratégie d'accès aux lignes
  - parcours de table, d'index, fonction, etc.
- Stratégie d'utilisation des jointures
  - ordre
  - ordre des tables jointes
  - type (*Nested Loop*, *Merge/Sort Join*, *Hash Join*…)
- Stratégie d'agrégation
  - brut, trié, haché
- En version parallélisée ?
- Tenir compte de la consommation mémoire

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut utiliser un parcours de table, ou parcourir un index et revenir à la table, ou se contenter de l'index (*Index Only Scan*). Il existe encore d'autres types de parcours. Les accès aux tables et index sont généralement les premières opérations utilisées.

Pour joindre les tables, l'ordre est très important pour essayer de réduire la masse des données manipulées. Les jointures se font toujours entre deux des tables impliquées, pas plus ; ou entre une table et le résultat d'un nœud, ou entre les résultats de deux nœuds.

Pour la jointure elle-même, il existe plusieurs méthodes différentes : boucles imbriquées, hachage, tri-fusion…

Il existe également plusieurs algorithmes d'agrégation des lignes. Un tri peut être nécessaire pour une jointure, une agrégation, ou pour un `ORDER BY`, et là encore il y a plusieurs algorithmes possibles. L'optimiseur peut aussi décider d'utiliser un index (déjà trié) pour éviter ce tri.

Certaines des opérations ci-dessus sont parallélisables. Certaines sont aussi susceptibles de consommer beaucoup de mémoire, l'optimiseur doit en tenir compte.

1.4.5 Parallélisation

- Processus supplémentaires pour certains nœuds
  - parer à la limitation par le CPU
- En lecture (sauf exceptions)
- Parcours séquentiel
- Jointures : *Nested Loop* / *Hash Join* / *Merge Join*
- Agrégats
- Parcours d'index (B-Tree uniquement)
- Création d'index B-Tree
- Certaines créations de table et vues matérialisées
- `DISTINCT` (v15)

Principe :

À partir d'une certaine quantité de données à traiter par un nœud, un ou plusieurs processus auxiliaires (*parallel workers*) apparaissent pour répartir la charge sur d'autres processeurs. Sans cela, une requête n'est traitée que par un seul processus sur un seul processeur.

> Il ne s'agit pas de lire une table avec plusieurs processus mais de répartir le traitement des lignes. La parallélisation n'est donc utile que si le CPU est le facteur limitant. Par exemple, un simple `SELECT` sur une grosse table sans `WHERE` ne mènera pas à un parcours parallélisé.

La parallélisation concerne en premier lieu les parcours de tables (*Seq Scan*), les jointures (*Nested Loop*, *Hash Join*, *Merge Join*), ainsi que certaines fonctions d'agrégat (comme `min`, `max`, `avg`, `sum`, etc.) ; mais encore les parcours d'index B-Tree (*Index Scan*, *Index Only Scan* et *Bitmap Scan*) La parallélisation est en principe disponible pour les autres types d'index, mais ils n'en font pas usage pour l'instant.

La parallélisation ne concerne encore que les opérations en lecture. Il y a des exceptions, comme la création des index B-Tree de façon parallélisée. Certaines créations de table avec `CREATE TABLE … AS`, `SELECT … INTO` sont aussi parallélisables, ainsi que `CREATE MATERIALIZED VIEW`.

En version 15, il devient possible de paralléliser des clauses `DISTINCT`.

Paramétrage :

Le paramétrage s'est affiné au fil des versions. Les paramètres suivants sont valables à partir de la version 13.

Le paramètre `max_parallel_workers_per_gather` (2 par défaut) désigne le nombre de processus auxiliaires maximum d'un nœud d'une requête. `max_parallel_maintenance_workers` (2 par défaut) est l'équivalent dans les opérations de maintenance (réindexation notamment). Trop de processus parallèles peuvent mener à une saturation de CPU ; l'exécuteur de PostgreSQL ne lancera donc pas plus de `max_parallel_workers` processus auxiliaires simultanés (8 par défaut), lui-même limité par `max_worker_processes` (8 par défaut). En pratique, on ajustera le nombre de *parallel workers* en fonction des CPU de la machine et de la charge attendue.

La mise en place de l'infrastructure de parallélisation a un coût, défini par `parallel_setup_cost` (1000 par défaut), et des tailles de table ou index minimales, en-dessous desquels la parallélisation n'est pas envisagée.

La plupart de ces paramètres peuvent être modifiés dans une sessions par `SET`.

1.4.6 Limites actuelles de la parallélisation

- Lourd à déclencher
- Pas sur les écritures de données
- Très peu d'opérations DDL gérées
- Pas en cas de verrous
- Pas sur les curseurs
- En évolution à chaque version

Même si cette fonctionnalité évolue au fil des versions majeures, des limitations assez fortes restent présentes[2], notamment :

- elle est assez lourde à mettre en place, elle a donc un coût d'entrée qui la rend inutile quand il y a peu de lignes ;
- pas de parallélisation pour les écritures de données (`INSERT`, `UPDATE`, `DELETE`, etc.),
- peu de parallélisation sur les opérations DDL (par exemple un `ALTER TABLE` ne peut pas être parallélisé)

Il y a des cas particuliers, notamment `CREATE TABLE AS` ou `CREATE MATERIALIZED VIEW`, parallélisable à partir de la v11 ; ou le niveau d'isolation *serializable*: avant la v12, il ne permet aucune parallélisation.

[2] https://docs.postgresql.fr/current/when-can-parallel-query-be-used.html

1.5 MÉCANISME DE COÛTS & STATISTIQUES

- Modèle basé sur les coûts
  - quantifier la charge pour répondre à une requête
- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne issue d'une lecture de table ou d'index
  - appliquer un opérateur

L'optimiseur statistique de PostgreSQL utilise un modèle de calcul de coût. Les coûts calculés sont des indications arbitraires sur la charge nécessaire pour répondre à une requête. Chaque facteur de coût représente une unité de travail : lecture d'un bloc, manipulation des lignes en mémoire, application d'un opérateur sur des données.

1.5.1 Coûts unitaires

- Coûts à connaître :
  - accès au disque séquentiel / non séquentiel
  - traitement d'un enregistrement issu d'une table
  - traitement d'un enregistrement issu d'un index
  - application d'un opérateur
  - traitement d'un enregistrement dans un parcours parallélisé
  - mise en place d'un parcours parallélisé
  - mise en place du JIT, du parallélisme…
- Chaque coût = un paramètre
  - modifiable dynamiquement avec `SET`

Pour quantifier la charge nécessaire pour répondre à une requête, PostgreSQL utilise un mécanisme de coût. Il part du principe que chaque opération a un coût plus ou moins important.

Divers paramètres permettent d'ajuster les coûts relatifs. Ces coûts sont arbitraires, à ne comparer qu'entre eux, et ne sont pas liés directement à des caractéristiques physiques du serveur.

- `seq_page_cost` (1 par défaut) représente le coût relatif d'un accès séquentiel à un bloc sur le disque, c'est-à-dire à un bloc lu en même temps que ses voisins dans la table ;
- `random_page_cost` (4 par défaut) représente le coût relatif d'un accès aléatoire (isolé) à un bloc : 4 signifie que le temps d'accès de déplacement de la tête de lecture de façon aléatoire est estimé 4 fois plus important que le temps d'accès en séquentiel — ce sera moins avec un bon disque, voire 1 pour un SSD ;
- `cpu_tuple_cost` (0,01 par défaut) représente le coût relatif de la manipulation d'une ligne en mémoire ;
- `cpu_index_tuple_cost` (0,005 par défaut) répercute le coût de traitement d'une donnée issue d'un index ;
- `cpu_operator_cost` (défaut 0,0025) indique le coût d'application d'un opérateur sur une donnée ;
- `parallel_tuple_cost` (0,1 par défaut) indique le coût estimé du transfert d'une ligne d'un processus à un autre ;
- `parallel_setup_cost` (1000 par défaut) indique le coût de mise en place d'un parcours parallélisé, une procédure assez lourde qui ne se rentabilise pas pour les petites requêtes ;
- `jit_above_cost` (100 000 par défaut), `jit_inline_above_cost` (500 000 par défaut), `jit_optimize_above_cost` (500 000 par défaut) représentent les seuils d'activation de divers niveaux du JIT (*Just In Time* ou compilation à la volée des requêtes), outil qui ne se rentabilise que sur les gros volumes.

En général, on ne modifie pas ces paramètres sans justification sérieuse. Le plus fréquemment, on peut être amené à diminuer `random_page_cost` si le serveur dispose de disques rapides, d'une carte RAID équipée d'un cache important ou de SSD. Mais en faisant cela, il faut veiller à ne pas déstabiliser des plans optimaux qui obtiennent des temps de réponse constants. À trop diminuer `random_page_cost`, on peut obtenir de meilleurs temps de réponse si les données sont en cache, mais aussi des temps de réponse dégradés si les données ne sont pas en cache.

Pour des besoins particuliers, ces paramètres sont modifiables dans une session. Ils peuvent être modifiés dynamiquement par l'application avec l'ordre `SET` pour des requêtes bien particulières, pour éviter de toucher au paramétrage général.

1.6 STATISTIQUES

- Combien de lignes va-t-on traiter ?
- Toutes les décisions du planificateur se basent sur les statistiques
  - le choix du parcours
  - comme le choix des jointures
- Mettre à jour les statistiques sur les données :
  - `ANALYZE`
- Sans bonnes statistiques, pas de bons plans !

Connaître le coût unitaire de traitement d'une ligne est une bonne chose, mais si on ne sait pas le nombre de lignes à traiter, on ne peut pas calculer le coût total. Le planificateur se base alors principalement sur les statistiques pour ses décisions. Avec ces informations et le paramétrage, l'optimiseur saura par exemple calculer le ratio d'un filtre et décider s'il faut passer par un index, ou calculer le ratio d'une jointure pour choisir la stratégie de jointure. Le choix du parcours, le choix des jointures, le choix de l'ordre des jointures, tout cela dépend des statistiques (et un peu de la configuration).

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 fréquemment.

La mise à jour se fait avec l'instruction `ANALYZE` qui peut être exécutée manuellement ou automatiquement (le démon autovacuum s'en occupe généralement, mais compléter avec une tâche planifiée avec cron ou les tâches planifiées sous Windows est possible). Nous allons voir comment les consulter.

1.6.1 Utilisation des statistiques

- 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

Les statistiques sur les données permettent à l'optimiseur de requêtes de déterminer assez précisément la répartition des valeurs d'une colonne d'une table, sous la forme d'un histogramme de répartition des valeurs. Il dispose encore d'autres informations comme la répartition des valeurs les plus fréquentes, le pourcentage de `NULL`, le nombre de valeurs distinctes, le niveau de corrélation entre valeurs et place sur le disque, etc.

L'optimiseur peut donc 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.

Ainsi, avec un filtre peu sélectif, `date_embauche = '2006-09-01'`, la requête va ramener pratiquement l'intégralité de la table. PostgreSQL choisira 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 va ramener 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 seront 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 ramènera que 2 lignes. L'optimiseur préférera donc passer par l'index que l'on a créé :

```
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.

1.6.2 Statistiques des tables et index

- Dans `pg_class`
  - `relpages` : taille
  - `reltuples` : lignes

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 de la table ou de l'index est exprimée en nombre de blocs de 8 ko et stockée dans la colonne `relpages`. La cardinalité de la table ou de l'index, c'est-à-dire le nombre de lignes, est stockée dans la colonne `reltuples`.

L'optimiseur utilisera ces deux informations pour apprécier la cardinalité de la table en fonction de sa volumétrie courante en calculant sa densité estimée puis en utilisant cette densité multipliée par le nombre de blocs actuel de la table pour estimer le nombre de lignes réel de la table :

```
density = reltuples / relpages;
tuples = density * curpages;
```

1.6.3 Statistiques : mono-colonne

- Nombre de valeurs distinctes
- Nombre d'éléments qui n'ont pas de valeur (`NULL`)
- Largeur d'une colonne
- Distribution des données
  - tableau des valeurs les plus fréquentes
  - histogramme de répartition des valeurs

Au niveau d'une colonne, plusieurs données statistiques sont stockées :

- le nombre de valeurs distinctes ;
- le nombre d'éléments qui n'ont pas de valeur (`NULL`) ;
- la largeur moyenne des données portées par la colonne ;
- le facteur de corrélation entre l'ordre des données triées et la répartition physique des valeurs dans la table ;
- la distribution des données.

La distribution des données est représentée sous deux formes qui peuvent être complémentaires. Tout d'abord, un tableau de répartition permet de connaître les valeurs les plus fréquemment rencontrées et la fréquence d'apparition de ces valeurs. Un histogramme de distribution des valeurs rencontrées permet également de connaître la répartition des valeurs pour la colonne considérée.

1.6.4 Stockage des statistiques mono-colonne

- Stockage dans `pg_statistic`
  - préférer la vue `pg_stats`
- Une table nouvellement créée n'a pas de statistiques
- Utilisation :

```
SELECT * FROM pg_stats
  WHERE  schemaname = 'public'
  AND tablename     = 'employes'
  AND attname       = 'date_embauche' \gx
```

La vue `pg_stats` a été créée pour faciliter la compréhension des statistiques récupérées par la commande `ANALYZE` et stockées dans `pg_statistic`.

1.6.5 Vue pg_stats

```
-[ RECORD 1 ]----------+-------------------------------------------------
 ↪  ---------
schemaname             | public
tablename              | employes
attname                | date_embauche
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -0.5
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}
histogram_bounds       | {2003-01-01,2006-06-01}
correlation            | 1
most_common_elems      | ¤
most_common_elem_freqs | ¤
elem_count_histogram   | ¤
```

Ce qui prècède est le contenu de `pg_stats` pour la colonne `date_embauche` de la table `employes`.

Trois champs identifient cette colonne :

- `schemaname` : nom du schéma (jointure possible avec `pg_namespace`)
- `tablename` : nom de la table (jointure possible avec `pg_class`, intéressant pour récupérer `reltuples` et `relpages`)
- `attname` : nom de la colonne (jointure possible avec `pg_attribute`, intéressant pour récupérer `attstatstarget`, valeur d'échantillon)

Suivent ensuite les colonnes de statistiques.

inherited :

Si `true`, les statistiques incluent les valeurs de cette colonne dans les tables filles. Ce n'est pas le cas ici.

null_frac

Cette statistique correspond au pourcentage de valeurs `NULL` dans l'échantillon considéré. Elle est toujours calculée. Il n'y a pas de valeurs nulles dans l'exemple ci-dessus.

avg_width

Il s'agit de la largeur moyenne en octets des éléments de cette colonne. 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`).

n_distinct

Si cette colonne contient un nombre positif, il s'agit du nombre de valeurs distinctes dans l'échantillon. Cela arrive uniquement quand le nombre de valeurs distinctes possibles semble fixe.

Si cette colonne contient un nombre négatif, il s'agit du nombre de valeurs distinctes dans l'échantillon divisé par le nombre de lignes. Cela survient uniquement quand le nombre de valeurs distinctes possibles semble variable. -1 indique donc que toutes les valeurs sont distinctes, -0,5 que chaque valeur apparaît deux fois (c'est en moyenne le cas ici).

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur `=`.

Il est possible de forcer cette colonne a une valeur constante en utilisant l'ordre `ALTER TABLE nom_table ALTER COLU` où `parametre` vaut soit :

- `n_distinct` pour une table standard,
- 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 l'utilisation de parcours d'index à la place de parcours de 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.

most_common_vals

Cette colonne 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 `=`.

most_common_freqs

Cette colonne 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`.

histogram_bounds

PostgreSQL prend l'échantillon récupéré par `ANALYZE`. Il trie ces valeurs. Ces données triées sont partagées en x tranches égales (aussi appelées classes), où x dépend de la valeur du paramètre `default_statistics_target` ou de la configuration spécifique de la colonne. Il construit ensuite un tableau dont chaque valeur correspond à la valeur de début d'une tranche.

most_common_elems, most_common_elem_freqs, elem_count_histogram

Ces trois colonnes sont équivalentes aux trois précédentes, mais uniquement pour les données de type tableau.

correlation

Cette colonne est la corrélation statistique entre l'ordre physique et l'ordre logique des valeurs de la colonne. Si sa valeur est proche de -1 ou 1, un parcours d'index est privilégié. Si elle est proche de 0, un parcours séquentiel est mieux considéré.

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur `<`.

1.6.6 Statistiques : multi-colonnes

- Pas par défaut
- `CREATE STATISTICS`
- Trois types de statistique
  - nombre de valeurs distinctes
  - dépendances fonctionnelles
  - liste MCV

Par défaut, la commande `ANALYZE` de PostgreSQL calcule des statistiques mono-colonnes uniquement. Elle peut aussi calculer certaines statistiques multi-colonnes. En effet, les valeurs des colonnes ne sont pas indépendantes et peuvent varier ensemble.

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 (depuis la version 12).

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

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, *i.e.* l'estimation du nombre de lignes (10013) est très proche de la réalité (10001) 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 mono-colonnes par défaut. Pour avoir des statistiques sur les deux colonnes, 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;
```

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 (10038 lignes) est beaucoup plus proche de la réalité (10001). Cela ne change rien au plan choisi dans ce cas précis, mais dans certains cas la différence peut être énorme.

Maintenant, prenons le cas d'un regroupement :

```
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 (dependencies,ndistinct)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;

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 aux statistiques spécifiques aux deux colonnes.

PostgreSQL 12 ajoute la méthode MCV (*most common values*) qui permet d'aller plus loin sur l'estimation du nombre de lignes. Notamment, elle permet de mieux estimer le nombre de lignes à partir d'un prédicat utilisant les opérations `<` et `>`. En voici un exemple :

```
DROP STATISTICS 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 ;

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
```

Une limitation existait avant PostgreSQL 13 : un seul objet statistique pouvait être utilisé par table et une requête ne pouvait utiliser qu'un seul objet statistique pour chaque table.

1.6.7 Statistiques sur les expressions

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

- Résout le problème des statistiques difficiles à estimer
- Pas créées par défaut
- Ne pas oublier `ANALYZE`
- (Avant v14 : index fonctionnel nécessaire)

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)
```

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 ques 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 provoque le calcul de statistiques dédiées. Or l'index fonctionnel n'a pas toujours d'intérêt : dans l'exemple précédent, presque toutes les dates d'embauche sont en 2006, et l'index ne serait donc pas utilisé.

1.6.8 Catalogues pour les statistiques étendues

Vues disponibles :

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

Les statistiques étendues sont visibles dans des tables comme `pg_statistic_ext` et `pg_statistic_ext_data` (à partir de la version 12), mais il est plus aisé de passer par la vue `pg_stats_ext` :

```
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      |
 ↪  {0.2512,0.2511666666666665,0.24886666666666668,0.24876666666666666}
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 \gx
```

```
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             | 1
most_common_vals       | {2006}
most_common_freqs      | {1}
histogram_bounds       |
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

1.6.9 ANALYZE

- `ANALYZE [ VERBOSE ] [ table [ ( colonne [, ...] ) ] [, ...] ]`
  - sans argument : base entière
  - avec argument : table complète ou certaines colonnes
- Un échantillon de table → statistiques
- Table vide : conserve les anciennes statistiques
- Nouvelle table : valeur par défaut

`ANALYZE` est l'ordre SQL permettant de mettre à jour les statistiques sur les données. Sans argument, l'analyse se fait sur la base complète. Si un ou plusieurs arguments sont donnés, ils doivent correspondre au nom des tables à analyser (en les séparant par des virgules). Il est même possible d'indiquer les colonnes à traiter.

En fait, cette instruction va exécuter un calcul d'un certain nombre de statistiques. Elle ne va pas lire la table entière, mais seulement un échantillon. Sur cet échantillon, chaque colonne sera traitée pour récupérer quelques informations comme le pourcentage de valeurs `NULL`, les valeurs les plus fréquentes et leur fréquence, sans parler d'un histogramme des valeurs. Toutes ces informations sont stockées dans le catalogue système nommé `pg_statistics`, accessible par la vue `pg_stats`, 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.

1.6.10 Fréquence d'analyse

- Dépend principalement de la fréquence des requêtes DML
- Autovacuum fait l'`ANALYZE` mais…
  - pas sur les tables temporaires
  - pas assez rapidement parfois
- Cron
  - `psql`
  - 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. Néanmoins, un `ANALYZE` tous les jours semble un minimum, sauf cas spécifique.

L'exécution périodique peut se faire avec cron (ou les tâches planifiées sous Windows). Il n'existe pas d'outil PostgreSQL pour lancer un seul `ANALYZE`, mais l'outil `vacuumdb` a une option `--analyze-only`. Ces deux ordres sont équivalents :

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

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

Le démon `autovacuum` fait aussi des `ANALYZE`. La fréquence dépend de sa configuration. Cependant, il faut connaître deux particularités de cet outil :

- Ce démon a sa propre connexion à la base. Il ne peut donc pas voir les tables temporaires appartenant aux autres sessions. Il ne sera donc pas capable de mettre à jour leurs statistiques.
- Après une insertion ou une mise à jour massive, autovacuum ne va pas forcément lancer un `ANALYZE` immédiat. 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. Pour des mises à jour plus régulières dans une grande table, il est assez fréquent qu'on doive réduire la valeur d'`autovacuum_analyze_scale_factor` (par défaut 10 % de la table doit être modifié pour déclencher automatiquement un `ANALYZE`).

1.6.11 Échantillon statistique

- `default_statistics_target` = 100
  - × 300 → 30 000 lignes au hasard
- Configurable par colonne

```
ALTER TABLE matable ALTER COLUMN nomchamp SET STATISTICS 300 ;
```

- Configurable par statistique étendue (v13+)

```
ALTER STATISTICS nom SET STATISTICS valeur ;
```

- `ANALYZE` ensuite
- Coût : temps de planification

Par défaut, un `ANALYZE` récupère 30 000 lignes d'une table. Les statistiques générées à partir de cet échantillon sont bonnes si la table ne contient pas des millions de lignes. Si c'est le cas, il faudra augmenter la taille de l'échantillon. Pour cela, il faut augmenter la valeur du paramètre `default_statistics_target` (100 par défaut). La taille de l'échantillon est de 300 fois `default_statistics_target`.

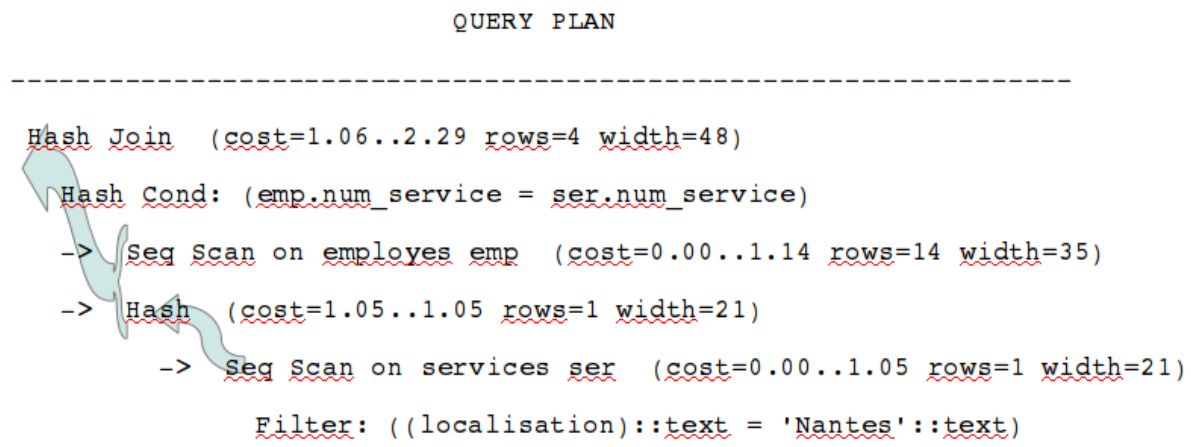
Si on l'augmente, les statistiques seront plus précises grâce à un échantillon plus important. Mais de ce fait, elles seront plus longues à calculer, prendront plus de place sur le disque et en RAM, et demanderont plus de travail au planificateur pour générer le plan optimal. Augmenter cette valeur n'a donc pas que des avantages : on évitera de dépasser 1000.

Il est possible de configurer ce paramètrage table par table et colonne par colonne :

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

Ne pas oublier de relancer un `ANALYZE nom_table ;` juste après.

1.7 LECTURE D'UN PLAN



Un plan d'exécution se lit en partant du nœud se trouvant le plus à droite et en remontant jusqu'au nœud final. Quand le plan contient plusieurs nœuds, le premier nœud exécuté est celui qui se trouve le plus à droite. Celui qui est le plus à gauche (la première ligne) est le dernier nœud exécuté. Tous les nœuds sont exécutés simultanément, et traitent les données dès qu'elles sont transmises par le nœud parent (le ou les nœuds justes en dessous, à droite).

Chaque nœud montre les coûts estimés dans le premier groupe de parenthèses. `cost` est un couple de deux coûts : la première valeur correspond au coût pour récupérer la première ligne (souvent nul dans le cas d'un parcours séquentiel) ; la deuxième valeur correspond au coût pour récupérer toutes les lignes (elle dépend essentiellement de la taille de la table lue, mais aussi d'opération de filtrage). `rows` correspond au nombre de lignes que le planificateur pense récupérer à la sortie de ce nœud. `width` est la largeur en octets de la ligne.

Cet exemple simple permet de voir le travail de l'optimiseur :

```
SET enable_nestloop TO off;
EXPLAIN
  SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM employes emp
  JOIN services ser ON (emp.num_service = ser.num_service)
  WHERE ser.localisation = 'Nantes';

                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=1.06..2.34 rows=4 width=48)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
   ->  Hash  (cost=1.05..1.05 rows=1 width=21)
         ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
               Filter: ((localisation)::text = 'Nantes'::text)

RESET enable_nestloop;
```

Ce plan débute en bas par la lecture de la table `services`. L'optimiseur estime que cette lecture ramènera une seule ligne (`rows=1`), que cette ligne occupera 21 octets en mémoire (`width=21`). Il s'agit

de la sélectivité du filtre `WHERE localisation = 'Nantes'`. Le coût de départ de cette lecture est de 0 (`cost=0.00`). Le coût total de cette lecture est de 1,05, qui correspond à la lecture séquentielle d'un seul bloc (paramètre `seq_page_cost`) et à la manipulation des 4 lignes de la table `services` (donc 4 * `cpu_tuple_cost` + 4 * `cpu_operator_cost`). Le résultat de cette lecture est ensuite haché par le nœud *Hash*, qui précède la jointure de type *Hash Join*.

La jointure peut maintenant commencer, avec le nœud *Hash Join*. Il est particulier, car il prend 2 entrées : la donnée hachée initialement, et les données issues de la lecture d'une seconde table (peu importe le type d'accès). Le nœud a un coût de démarrage de 1,06, soit le coût du hachage additionné au coût de manipulation du tuple de départ. Il s'agit du coût de production du premier tuple de résultat. Le coût total de production du résultat est de 2,34. La jointure par hachage démarre réellement lorsque la lecture de la table `employes` commence. Cette lecture remontera 14 lignes, sans application de filtre. La totalité de la table est donc remontée et elle est très petite donc tient sur un seul bloc de 8 ko. Le coût d'accès total est donc facilement déduit à partir de cette information. À partir des sélectivités précédentes, l'optimiseur estime que la jointure ramènera 4 lignes au total.

1.7.1 Rappel des options d'EXPLAIN

- `ANALYZE` : exécution (danger !)
- `BUFFERS` : blocs *read*/*hit*/*written*/*dirtied*, *shared*/*local*/*temp*
- `GENERIC_PLAN` : plan générique (requête préparée, v16)
- `SETTINGS` : paramètres configurés pour l'optimisation
- `WAL` : nombre d'enregistrements et nombre d'octets écrits dans les journaux
- `COSTS` : par défaut
- `TIMING` : par défaut
- `VERBOSE` : colonnes considérées
- `SUMMARY` : temps de planification
- `FORMAT` : sortie en text, XML, JSON, YAML

Au fil des versions, `EXPLAIN` a gagné en options. L'une d'entre elles permet de sélectionner le format en sortie. Toutes les autres permettent d'obtenir des informations supplémentaires, ou au contraire de masquer des informations affichées par défaut.

Option ANALYZE

Le but de cette option est d'obtenir les informations sur l'exécution réelle de la requête.

Avec `ANALYZE`, la requête est réellement exécutée ! Attention donc aux `INSERT` / `UPDATE` / `DELETE`. N'oubliez pas non plus qu'un `SELECT` peut appeler des fonctions qui écrivent dans la base. Dans le doute, pensez à englober l'appel dans une transaction que vous annulerez après coup.

Voici un exemple utilisant cette option :

```
BEGIN;
EXPLAIN (ANALYZE) SELECT * FROM employes WHERE matricule < 100 ;
ROLLBACK;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                (actual time=0.004..0.005 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.027 ms
 Execution time: 0.013 ms
```

Quatre nouvelles informations apparaissent, toutes liées à l'exécution réelle de la requête :

- `actual time` :
  - la première valeur correspond à la durée en milliseconde pour récupérer la première ligne ;
  - la deuxième valeur est la durée en milliseconde pour récupérer toutes les lignes ;
- `rows` est le nombre de lignes *réellement* récupérées : comparer au nombre de la première parenthèse permet d'avoir une idée de la justesse des statistiques et de l'estimation ;
- `loops` est le nombre d'exécutions de ce nœud, car certains peuvent être répétés de nombreuses fois.

Multiplier la durée par le nombre de boucles pour obtenir la durée réelle d'exécution du nœud !

L'intérêt de cette option est donc de trouver l'opération qui prend du temps dans l'exécution de la requête, mais aussi de voir les différences entre les estimations et la réalité (notamment au niveau du nombre de lignes).

Option BUFFERS

Cette option n'est en pratique utilisable qu'avec l'option `ANALYZE`. Elle est désactivée par défaut.

Elle indique le nombre de blocs impactés par chaque nœud du plan d'exécution, en lecture comme en écriture.

Voici un exemple de son utilisation :

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM employes WHERE matricule < 100;

                      QUERY PLAN
-------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
              (actual time=0.002..0.004 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
   Buffers: shared hit=1
 Planning time: 0.024 ms
 Execution time: 0.011 ms
```

La nouvelle ligne est la ligne `Buffers`. `shared hit` indique un accès à une table ou index dans les *shared buffers* de PostgreSQL. Ces autres indications peuvent se rencontrer :

Informations	Type d'objet concerné	Explications
Shared hit	Table ou index permanent	Lecture d'un bloc dans le cache
Shared read	Table ou index permanent	Lecture d'un bloc hors du cache
Shared written	Table ou index permanent	Écriture d'un bloc
Local hit	Table ou index temporaire	Lecture d'un bloc dans le cache

Informations	Type d'objet concerné	Explications
Local read	Table ou index temporaire	Lecture d'un bloc hors du cache
Local written	Table ou index temporaire	Écriture d'un bloc
Temp read	Tris et hachages	Lecture d'un bloc
Temp written	Tris et hachages	Écriture d'un bloc

`EXPLAIN (BUFFERS)`, sans `ANALYZE`, fonctionne certes à partir de PostgreSQL 13, mais n'affiche alors que les quelques blocs consommés par la planification.

Option GENERIC_PLAN

Cette option (à partir de PostgreSQL 16) sert quand on cherche le plan générique planifié pour une requête préparée (c'est-à-dire dont les paramètres seront fournis séparément).

```
EXPLAIN (GENERIC_PLAN, SUMMARY ON)
SELECT * FROM t1 WHERE c1 < $1 ;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.15..14.98 rows=333 width=8)
   Index Cond: (c1 < $1)
 Planning Time: 0.195 ms
```

Option SETTINGS

Cette option permet d'obtenir les valeurs des paramètres spécifiques à l'optimisation de requêtes qui ne sont pas à leur valeur par défaut. Elle est désactivée par défaut.

```
EXPLAIN (SETTINGS) SELECT * FROM employes_big WHERE matricule=33;
```

```
                               QUERY PLAN
-------------------------------------------------------------------------
Index Scan using employes_big_pkey on employes_big (cost=0.42..8.44 rows=1 width=41)
  Index Cond: (matricule = 33)
```

```
SET enable_indexscan TO off;
EXPLAIN (SETTINGS) SELECT * FROM employes_big WHERE matricule=33;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big  (cost=4.43..8.44 rows=1 width=41)
   Recheck Cond: (matricule = 33)
   ->  Bitmap Index Scan on employes_big_pkey  (cost=0.00..4.43 rows=1 width=0)
         Index Cond: (matricule = 33)
 Settings: enable_indexscan = 'off'
```

Option WAL

Cette option permet d'obtenir le nombre d'enregistrements et le nombre d'octets écrits dans les journaux de transactions. Elle apparaît avec PostgreSQL 13 et est désactivée par défaut.

```
CREATE TABLE t1 (id integer);
EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT generate_series(1, 1000) ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Insert on t1  (cost=0.00..15.02 rows=1000 width=12)
           (actual time=1.457..1.458 rows=0 loops=1)
   WAL: records=2009 bytes=123824
   ->  Subquery Scan on "*SELECT*"
       (cost=0.00..15.02 rows=1000 width=12)
       (actual time=0.003..0.146 rows=1000 loops=1)
         ->  ProjectSet  (cost=0.00..5.02 rows=1000 width=4)
                (actual time=0.002..0.068 rows=1000 loops=1)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
                  (actual time=0.001..0.001 rows=1 loops=1)
 Planning Time: 0.033 ms
 Execution Time: 1.479 ms
```

Option COSTS

Activée par défaut, l'option `COSTS` indique les estimations du planificateur. La désactiver permet de gagner un peu en lisibilité.

```
EXPLAIN (COSTS OFF) SELECT * FROM employes WHERE matricule < 100;
```

```
         QUERY PLAN
----------------------------
 Seq Scan on employes
   Filter: (matricule < 100)
```

```
EXPLAIN (COSTS ON) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
   Filter: (matricule < 100)
```

Option TIMING

Cette option n'est utilisable qu'avec l'option `ANALYZE` et est activée par défaut. Elle ajoute les informations sur les durées en milliseconde. Sa désactivation peut être utile sur certains systèmes où le chronométrage prend beaucoup de temps et allonge inutilement la durée d'exécution de la requête.

Voici un exemple de son utilisation :

```
EXPLAIN (ANALYZE, TIMING ON) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
              (actual time=0.003..0.004 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.022 ms
 Execution time: 0.010 ms
```

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
```

```
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                                (actual rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.025 ms
 Execution time: 0.010 ms
```

Option VERBOSE

L'option `VERBOSE` permet d'afficher des informations supplémentaires comme la liste des colonnes en sortie, le nom de la table qualifié du nom du schéma, le nom de la fonction qualifié du nom du schéma, le nom du déclencheur (trigger), etc. Elle est désactivée par défaut.

```
EXPLAIN (VERBOSE) SELECT * FROM employes WHERE matricule < 100;

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on public.employes  (cost=0.00..1.18 rows=3 width=43)
   Output: matricule, nom, prenom, fonction, manager, date_embauche,
           num_service
   Filter: (employes.matricule < 100)
```

On voit dans cet exemple que le nom du schéma est ajouté au nom de la table. La nouvelle section `Output` indique la liste des colonnes de l'ensemble de données en sortie du nœud.

Option SUMMARY

Cette option apparaît en version 10, et permet d'afficher ou non le résumé final indiquant la durée de la planification et de l'exécution. Par défaut, un `EXPLAIN` simple n'affiche pas le résumé, mais un `EXPLAIN ANALYZE` le fait.

```
EXPLAIN SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)

EXPLAIN (SUMMARY ON) SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
 Planning time: 0.014 ms

EXPLAIN (ANALYZE) SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
              (actual time=0.002..0.003 rows=14 loops=1)
 Planning time: 0.013 ms
 Execution time: 0.009 ms

EXPLAIN (ANALYZE, SUMMARY OFF) SELECT * FROM employes;
```

```
                      QUERY PLAN
------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
               (actual time=0.002..0.003 rows=14 loops=1)
```

Option FORMAT

Par défaut, la sortie est sous forme d'un texte destiné à être lu par un humain, mais il est possible de choisir un format balisé parmi XML, JSON et YAML. Voici ce que donne la commande `EXPLAIN` avec le format XML :

```
EXPLAIN (FORMAT XML) SELECT * FROM employes WHERE matricule < 100;
```

```
                         QUERY PLAN
------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">+
   <Query>                                               +
     <Plan>                                              +
       <Node-Type>Seq Scan</Node-Type>                   +
       <Parallel-Aware>false</Parallel-Aware>            +
       <Relation-Name>employes</Relation-Name>           +
       <Alias>employes</Alias>                           +
       <Startup-Cost>0.00</Startup-Cost>                 +
       <Total-Cost>1.18</Total-Cost>                     +
       <Plan-Rows>3</Plan-Rows>                          +
       <Plan-Width>43</Plan-Width>                       +
       <Filter>(matricule &lt; 100)</Filter>             +
     </Plan>                                             +
   </Query>                                              +
 </explain>
(1 row)
```

Les signes `+` en fin de ligne indiquent un retour à la ligne lors de l'utilisation de l'outil `psql`. Il est possible de ne pas les afficher en configurant l'option `format` de `psql` à `unaligned`. Cela se fait ainsi :

```
\pset format unaligned
```

Ces formats semi-structurés sont utilisés principalement par des outils, car le contenu est plus facile à analyser, voire un peu plus complet.

1.7.2 Statistiques, cardinalités & coûts

- Détermine à partir des statistiques
  - cardinalité des prédicats
  - cardinalité des jointures
- Coût d'accès déterminé selon
  - des cardinalités
  - volumétrie des tables

Afin de comparer les différents plans d'exécution possibles pour une requête et choisir le meilleur, l'optimiseur a besoin d'estimer un coût pour chaque nœud du plan.

L'estimation la plus cruciale est celle liée aux nœuds de parcours de données, car c'est d'eux que découlera la suite du plan. Pour estimer le coût de ces nœuds, l'optimiseur s'appuie sur les informations statistiques collectées, ainsi que sur la valeur de paramètres de configuration.

Les deux notions principales de ce calcul sont la cardinalité (nombre de lignes estimées en sortie d'un nœud) et la sélectivité (fraction des lignes conservées après l'application d'un filtre).

Voici ci-dessous un exemple de calcul de cardinalité et de détermination du coût associé.

Calcul de cardinalité

Pour chaque prédicat et chaque jointure, PostgreSQL va calculer sa sélectivité et sa cardinalité. Pour un prédicat, cela permet de déterminer le nombre de lignes retournées par le prédicat par rapport au nombre total de lignes de la table. Pour une jointure, cela permet de déterminer le nombre de lignes retournées par la jointure entre deux tables.

L'optimiseur dispose de plusieurs façons de calculer la cardinalité d'un filtre ou d'une jointure selon que la valeur recherchée est une valeur unique, que la valeur se trouve dans le tableau des valeurs les plus fréquentes ou dans l'histogramme. Cherchons comment calculer la cardinalité d'un filtre simple sur une table `employes` de 14 lignes, par exemple `WHERE num_service = 1`.

Ici, la valeur recherchée se trouve directement dans le tableau des valeurs les plus fréquentes (dans les champs `most_common_vals` et `most_common_freqs`) la cardinalité peut être calculée directement.

```
SELECT * FROM pg_stats
WHERE tablename = 'employes'
  AND attname   = 'num_service' \gx

-[ RECORD 1 ]----------+----------------------------------------------
schemaname             | public
tablename              | employes
attname                | num_service
```

```
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -0.2857143
most_common_vals       | {4,3,2,1}
most_common_freqs      | {0.35714287,0.2857143,0.21428572,0.14285715}
histogram_bounds       | ¤
correlation            | 0.10769231
…
```

La requête suivante permet de récupérer la fréquence d'apparition de la valeur recherchée :

```
SELECT tablename, attname, value, freq
  FROM (SELECT tablename, attname, mcv.value, mcv.freq FROM pg_stats,
              LATERAL ROWS FROM (unnest(most_common_vals::text::int[]),
                                 unnest(most_common_freqs)) AS mcv(value, freq)
         WHERE tablename = 'employes'
           AND attname = 'num_service') get_mcv
 WHERE value = 1;

 tablename |   attname   | value |   freq
-----------+-------------+-------+----------
 employes  | num_service |     1 | 0.142857
```

Si l'on n'avait pas eu affaire à une des valeurs les plus fréquentes, il aurait fallu passer par l'histogramme des valeurs (`histogram_bounds`, ici vide car il y a trop peu de valeurs), pour calculer d'abord la sélectivité du filtre pour en déduire ensuite la cardinalité.

Une fois cette fréquence obtenue, l'optimiseur calcule la cardinalité du prédicat `WHERE num_service = 1` en la multipliant avec le nombre total de lignes de la table :

```
SELECT 0.142857 * reltuples AS cardinalite_predicat
  FROM pg_class
 WHERE relname = 'employes';

 cardinalite_predicat
----------------------
             1.999998
```

Le calcul est cohérent avec le plan d'exécution de la requête impliquant la lecture de `employes` sur laquelle on applique le prédicat évoqué plus haut :

```
EXPLAIN SELECT * FROM employes WHERE num_service = 1;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=2 width=43)
   Filter: (num_service = 1)
```

Calcul de coût

Notre table `employes` peuplée de 14 lignes va permettre de montrer le calcul des coûts réalisés par l'optimiseur. L'exemple présenté ci-dessous est simplifié. En réalité, les calculs sont plus complexes, car ils tiennent également compte de la volumétrie réelle de la table.

Le coût de la lecture séquentielle de la table `employes` est calculé à partir de deux composantes. Tout d'abord, le nombre de pages (ou blocs) de la table permet de déduire le nombre de blocs à accéder pour lire la table intégralement. Le paramètre `seq_page_cost` (coût d'accès à un bloc dans un parcours complet) sera appliqué ensuite pour obtenir le coût de l'opération :

```
SELECT relname, relpages * current_setting('seq_page_cost')::float AS cout_acces
  FROM pg_class
 WHERE relname = 'employes';

 relname  | cout_acces
----------+------------
 employes |          1
```

Cependant, le coût d'accès seul ne représente pas le coût de la lecture des données. Une fois que le bloc est monté en mémoire, PostgreSQL doit décoder chaque ligne individuellement. L'optimiseur multiplie donc par `cpu_tuple_cost` (0,01 par défaut) pour estimer le coût de manipulation des lignes :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float AS cout
  FROM pg_class
 WHERE relname = 'employes';

 relname  | cout
----------+------
 employes | 1.14
```

Le calcul est bon :

```
EXPLAIN SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
```

Avec un filtre dans la requête, les traitements seront plus lourds. Par exemple, en ajoutant le prédicat `WHERE date_embauche='2006-01-01'`, il faut non seulement extraire les lignes les unes après les autres, mais également appliquer l'opérateur de comparaison utilisé. L'optimiseur utilise le paramètre `cpu_operator_cost` pour déterminer le coût d'application d'un filtre :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float
       + reltuples * current_setting('cpu_operator_cost')::float AS cost
  FROM pg_class
 WHERE relname = 'employes';

 relname  | cost
----------+-------
 employes | 1.175
```

Ce nombre se retrouve dans le plan, à l'arrondi près :

```
EXPLAIN SELECT * FROM employes WHERE date_embauche='2006-01-01';
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=2 width=43)
   Filter: (date_embauche = '2006-01-01'::date)
```

Pour aller plus loin dans le calcul de sélectivité, de cardinalité et de coût, la documentation de PostgreSQL contient un exemple complet de calcul de sélectivité et indique les références des fichiers sources dans lesquels fouiller pour en savoir plus[3].

[3] https://docs.postgresql.fr/current/planner-stats-details.html

1.8 NŒUDS D'EXÉCUTION LES PLUS COURANTS

- Un plan est composé de nœuds
- qui produisent des données
- ou en consomment et en retournent
- Chaque nœud consomme les données produites par le(s) nœud(s) parent(s)
- Le nœud final retourne les données à l'utilisateur

Les plans sont extrêmement sensibles aux données elles-mêmes bien sûr, aux paramètres, aux tailles réelles des objets, à la version de PostgreSQL, à l'ordre des données dans la table, voire au moment du passage d'un `VACUUM`. Il n'est donc pas étonnant de trouver parfois des plans différents de ceux reproduits ici pour une même requête.

1.8.1 Nœuds de type parcours

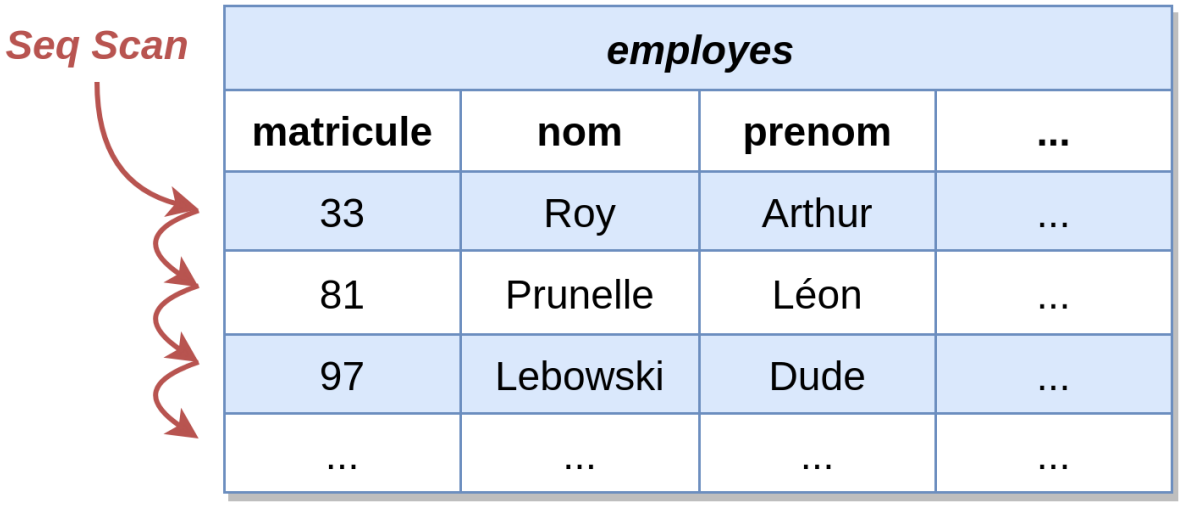
- Parcours de table
- Parcours d'index
- Autres parcours

Par parcours, on entend le renvoi d'un ensemble de lignes provenant soit d'un fichier, soit d'un traitement. Le fichier peut correspondre à une table ou à une vue matérialisée, et on parle dans ces deux cas d'un parcours de table. Le fichier peut aussi correspondre à un index, auquel cas on parle de parcours d'index. Un parcours peut être un traitement dans différents cas, principalement celui d'une procédure stockée.

1.8.2 Parcours de table

- *Seq Scan*
- *Parallel Seq Scan*

1.8.3 Parcours de table : Seq Scan



Seq Scan :

Les parcours de tables sont les principales opérations qui lisent les données des tables (normales, temporaires ou non journalisées) et des vues matérialisées. Elles ne prennent donc pas d'autre nœud en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

L'opération *Seq Scan* correspond à une lecture séquentielle d'une table, aussi appelée *Full table scan* sur d'autres SGBD. Il consiste à lire l'intégralité de la table, du premier bloc au dernier bloc. Une clause de filtrage peut être appliquée.

Ce nœud apparaît lorsque la requête nécessite de lire l'intégralité ou la majorité de la table :

```
EXPLAIN SELECT * FROM employes;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
```

Ce nœud peut également filtrer directement les données, la présence de la clause *Filter* montre le filtre appliqué par le nœud à la lecture des données :

```
EXPLAIN SELECT * FROM employes WHERE matricule=135;
```

```
                       QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=1 width=43)
   Filter: (matricule = 135)
```

1.8.4 Parcours de table : paramètres

- *Seq Scan*
  - `seq_page_cost` (défaut : 1)
  - `cpu_tuple_cost` & `cpu_operator_cost`
  - `enable_seqscan`
- *Parallel Seq Scan*
  - `parallel_tuple_cost`, `min_parallel_table_scan_size`
  - et les autres paramètres de la parallélisation

Seq Scan :

Le coût d'un *Seq Scan* sera fonction du nombre de blocs à parcourir, et donc du paramètre `seq_page_cost`, ainsi que du nombre de lignes à décoder et, optionnellement, à filtrer. La valeur de `seq_page_cost`, par défaut à 1, est rarement modifiée ; elle est surtout importante comparée à `random_page_cost`.

Parallel Seq Scan :

Il est possible d'avoir un parcours parallélisé d'une table (*Parallel Seq Scan*). La parallélisation doit être activée comme décrit plus haut, notamment :

- les paramètres `max_parallel_workers_per_gather` et `max_parallel_workers` doivent être tous deux supérieurs à 0, ce qui est le cas par défaut ;
- la table à traiter doit avoir une taille minimale (`min_parallel_table_scan_size` est à 8 Mo) ; et plus elle sera grosse plus il y aura de *workers* ;

Pour que ce type de parcours soit valable, il faut que l'optimiseur soit persuadé que le problème sera le temps CPU et non la bande passante disque. Il y a cependant un coût pour chaque ligne (paramètre `parallel_tuple_cost`). Autrement dit, dans la majorité des cas, il faut un filtre sur une table importante pour que la parallélisation se déclenche.

Dans les exemples suivants, la parallélisation est activée :

```
SET max_parallel_workers_per_gather TO 5 ;   /*  défaut : 2 */

-- Plan d'exécution parallélisé
EXPLAIN SELECT * FROM employes_big WHERE num_service <> 4;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
```

```
->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
      Filter: (num_service <> 4)
```

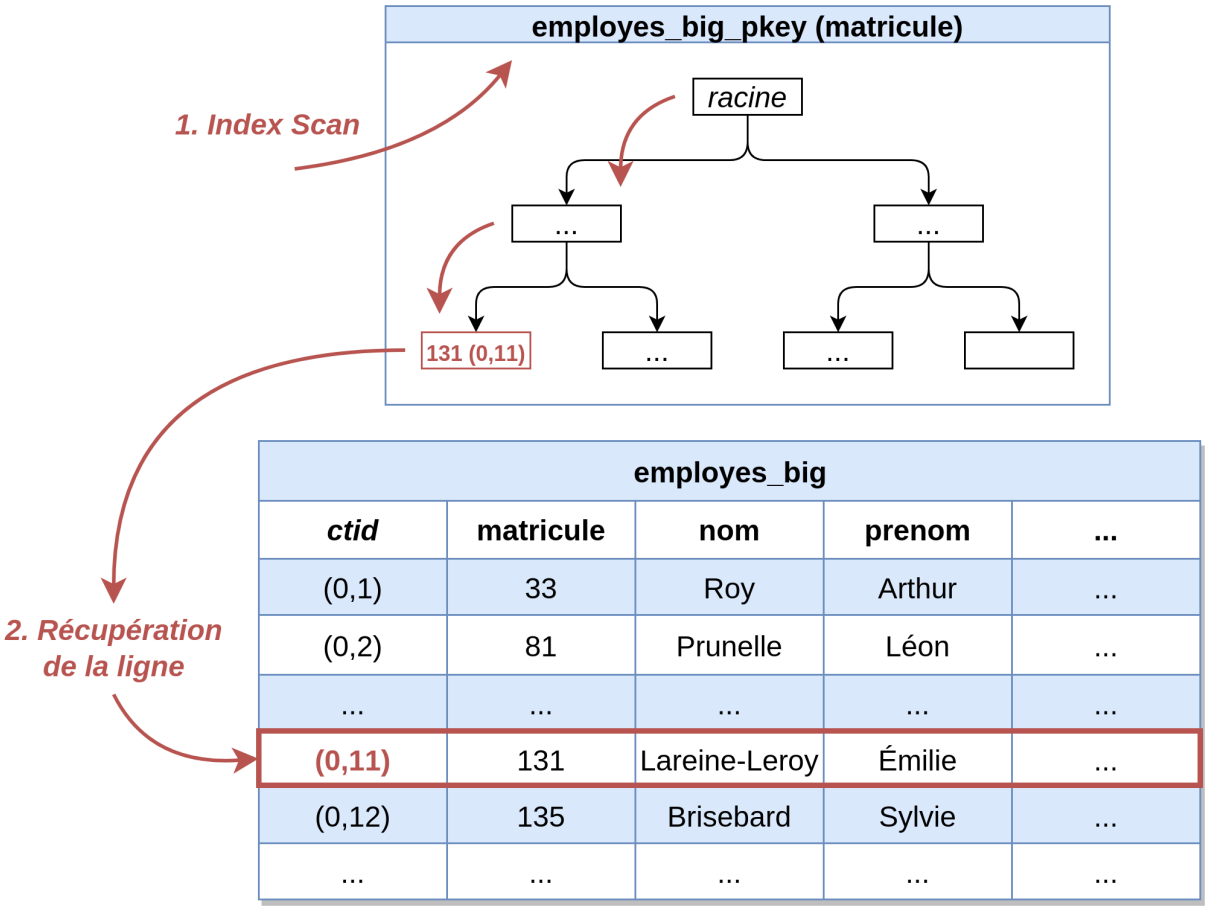
Ici, deux processus supplémentaires seront exécutés pour réaliser la requête. Dans le cas de ce type de parcours, chaque processus prend un bloc et traite toutes les lignes de ce bloc. Quand un processus a terminé de traiter son bloc, il regarde quel est le prochain bloc à traiter et le traite. (À partir de la version 14, il prend même un groupe de blocs pour profiter de la fonctionnalité de *read ahead* du noyau.)

1.8.5 Parcours d'index

- *Index Scan*
- *Index Only Scan*
  - index couvrants
- *Bitmap Index Scan*
- et leurs versions parallélisées (B-Tree)

PostgreSQL dispose de trois moyens d'accéder aux données à travers les index. Le plus connu est l'*Index Scan*, qui possède plusieurs variantes.

1.8.6 Index Scan



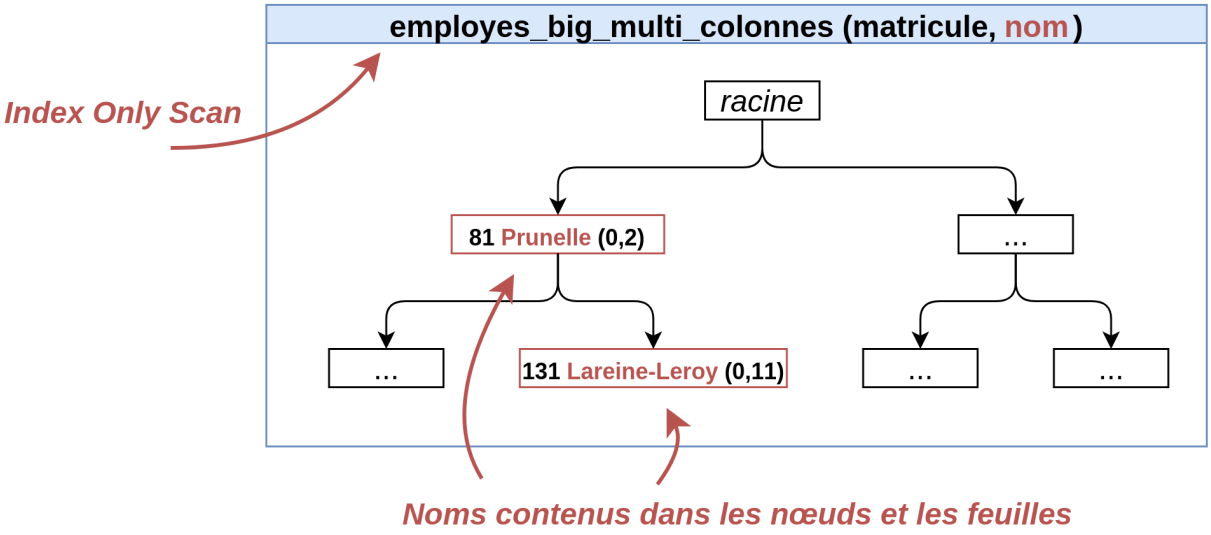
Le nœud *Index Scan* consiste à parcourir les blocs d'index jusqu'à trouver les pointeurs vers les blocs contenant les données. À chaque pointeur trouvé, PostgreSQL lit le bloc de la table pointée pour retrouver l'enregistrement et s'assurer notamment de sa visibilité pour la transaction en cours. De ce fait, il y a beaucoup d'accès non séquentiels pour lire l'index et la table.

```
EXPLAIN SELECT * FROM employes_big WHERE matricule = 132;
```

```
                    QUERY PLAN
---------------------------------------------------
 Index Scan using employes_big_pkey on employes_big
   (cost=0.42..8.44 rows=1 width=41)
   Index Cond: (matricule = 132)
```

Ce type de nœud ne permet pas d'extraire directement les données à retourner depuis l'index, sans passer par la lecture des blocs correspondants de la table.

1.8.7 Index Only Scan



- Besoin d'un `VACUUM` récent

Le nœud *Index Only Scan* est une version plus performante de l'*Index Scan*, à condition que les colonnes retournées par la requête fassent partie de l'index :

```
EXPLAIN SELECT matricule FROM employes_big WHERE matricule < 132;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Only Scan using employes_big_pkey on employes_big
   (cost=0.42..5.82 rows=80 width=4)
   Index Cond: (matricule < 132)
```

Il n'y a donc plus besoin d'accéder à la table, ce qui est encore plus appréciable avec de nombreuses lignes.

Mais pour que ce type de nœud soit réellement efficace, il faut bien s'assurer que la table en relation soit fréquemment traitée par des opérations `VACUUM`. En effet, les informations de visibilité des lignes ne sont pas stockées dans l'index. Pour savoir si la ligne trouvée dans l'index est visible ou pas par la session, soit il faut aller voir dans la table (et on revient à un *Index Scan*), soit il faut la garantie que le bloc ait été nettoyé de lignes potentiellement invisibles (le `VACUUM` stocke cela dans la *visibility map* de table).

En ajoutant le champ `nom` dans la requête, l'optimiseur se rabat sur un *Index Scan*, car ce champ n'est pas dans l'index, et sa valeur doit être trouvée en lisant la ligne dans la table.

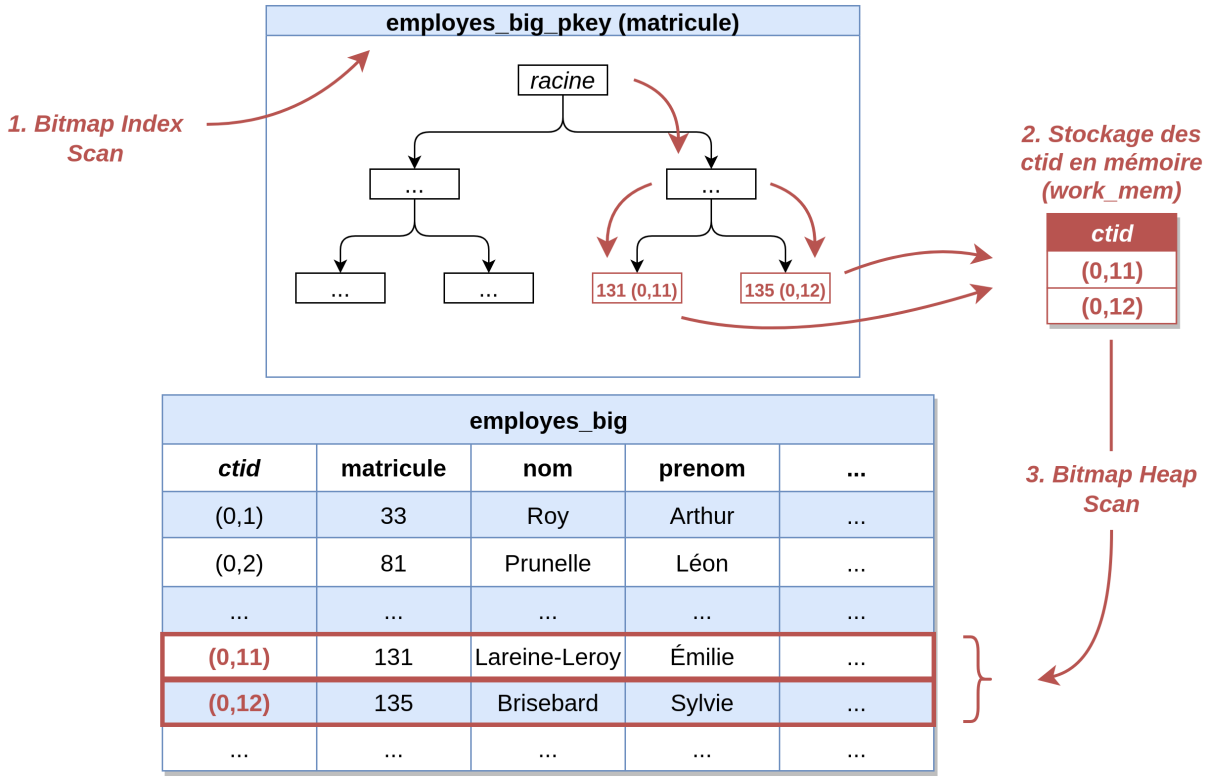
Il existe des index dits « couvrants », destinés à obtenir des *Index Only Scans*. Ils peuvent contenir des données en plus des champs indexés :

```
CREATE UNIQUE INDEX ON employes_big (matricule) INCLUDE (nom) ;

EXPLAIN SELECT matricule, nom FROM employes_big WHERE matricule < 132;

                          QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using employes_big_matricule_nom_idx on employes_big
   (cost=0.42..9.82 rows=80 width=11)
   Index Cond: (matricule < 132)
```

1.8.8 Bitmap Scan



Ce dernier parcours est particulièrement efficace pour des opérations de type *Range Scan*, c'est-à-dire où PostgreSQL doit retourner une plage de valeurs, ou pour combiner le résultat de la lecture de plusieurs index.

Contrairement à d'autres SGBD, un index *bitmap* de PostgreSQL n'a aucune existence sur disque : il est créé en mémoire lorsque son utilisation a un intérêt. Le but est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table :

- lecture en un bloc de l'index ;
- lecture en un bloc de la partie intéressante de la table (dans l'ordre physique de la table, pas dans l'ordre logique de l'index).

```
SET enable_indexscan TO off ;
```

```
EXPLAIN
  SELECT * FROM employes_big WHERE matricule BETWEEN 200000 AND 300000;

                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big
   (cost=2108.46..8259.35 rows=99126 width=41)
   Recheck Cond: ((matricule >= 200000) AND (matricule <= 300000))
   ->  Bitmap Index Scan on employes_big_pkey*
       (cost=0.00..2083.68 rows=99126 width=0)
         Index Cond: ((matricule >= 200000) AND (matricule <= 300000))

RESET enable_indexscan;
```

Exemple de combinaison du résultat de la lecture de plusieurs index :

```
EXPLAIN
  SELECT * FROM employes_big
  WHERE matricule BETWEEN   1000 AND 100000
     OR matricule BETWEEN 200000 AND 300000;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big
   (cost=4265.09..12902.67 rows=178904 width=41)
   Recheck Cond: (((matricule >= 1000) AND (matricule <= 100000))
               OR ((matricule >= 200000) AND (matricule <= 300000)))
   ->  BitmapOr  (cost=4265.09..4265.09 rows=198679 width=0)
         ->  Bitmap Index Scan on employes_big_pkey
               (cost=0.00..2091.95 rows=99553 width=0)
               Index Cond: ((matricule >= 1000) AND (matricule <= 100000))
         ->  Bitmap Index Scan on employes_big_pkey
               (cost=0.00..2083.68 rows=99126 width=0)
               Index Cond: ((matricule >= 200000) AND (matricule <= 300000))
```

1.8.9 Parcours d'index : paramètres importants

- `random_page_cost` (4 ou moins ?)
- `cpu_index_tuple_cost`
- `effective_cache_size` (⅔ de la RAM ?)
- Selon le disque :
  - `effective_io_concurrency`
  - `maintenance_io_concurrency`
- `min_parallel_index_scan_size`
- `enable_indexscan`, `enable_indexonlyscan`, `enable_bitmapscan`

Index Scan :

L'*Index Scan* n'a d'intérêt que s'il y a très peu de lignes à récupérer, surtout si les disques sont mécaniques. Il faut donc que le filtre soit très sélectif.

Le rapport entre les paramètres `seq_page_cost` et `random_page_cost` est d'importance majeure dans le choix face à un *Seq Scan*. Plus il est proche de 1, plus les parcours d'index seront favorisés par rapport aux parcours de table.

Index Only Scan :

Il n'y a pas de paramètre dédié à ce parcours. S'il est possible, l'optimiseur le préfère à un *Index Scan*.

Bitmap Index Scan :

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[4], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. À l'inverse, il faut tenir compte du nombre de requêtes simultanées qui utiliseront ce nœud. Le défaut est seulement de 1, et la valeur maximale est 1000. Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[5].

Toujours à partir de la version 13, un nouveau paramètre apparaît : `maintenance_io_concurrency`. Il a le même but que `effective_io_concurrency`, mais pour les opérations de maintenance, non les requêtes. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Sa valeur par défaut est de 10, et il faut penser à le monter aussi si on adapte `effective_io_concurrency`.

Enfin, le paramètre `effective_cache_size` indique à PostgreSQL une estimation de la taille du cache disque du système (total du *shared buffers* et du cache du système). Une bonne pratique est de positionner ce paramètre à ⅔ de la quantité totale de RAM du serveur. Sur un système Linux, il est possible de donner une estimation plus précise en s'appuyant sur la valeur de colonne `cached` de la commande `free`. Mais si le cache n'est que peu utilisé, la valeur trouvée peut être trop basse pour pleinement favoriser l'utilisation des *Bitmap Index Scan*.

Parallélisation

Il est possible de paralléliser les parcours d'index. Cela donne donc les nœuds *Parallel Index Scan*, *Parallel Index Only Scan* et *Parallel Bitmap Heap Scan*. Cette infrastructure est actuellement uniquement utilisée pour les index B-Tree. Par contre, pour le *Bitmap Scan*, seul le parcours de la table est parallélisé. Un parcours parallélisé d'un index n'est considéré qu'à partir du moment où l'index a une taille supérieure à la valeur du paramètre `min_parallel_index_scan_size` (512 ko par défaut).

[4] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[5] https://docs.postgresql.fr/13/release.html

1.8.10 Autres parcours

- *Function Scan*
- *Values Scan*
- …et d'autres

On retrouve le nœud *Function Scan* lorsqu'une requête utilise directement le résultat d'une fonction, comme par exemple, dans des fonctions d'informations système de PostgreSQL :

```
EXPLAIN SELECT * from pg_get_keywords();
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Function Scan on pg_get_keywords  (cost=0.03..4.03 rows=400 width=65)
```

Il existe d'autres types de parcours, rarement rencontrés. Ils sont néanmoins détaillés en annexe[6].

1.8.11 Nœuds de jointure

- PostgreSQL implémente les 3 algorithmes de jointures habituels
  - *Nested Loop* : boucle imbriquée
  - *Hash Join* : hachage de la table interne
  - *Merge Join* : tri-fusion
- Parallélisation
- Pour `EXISTS`, `IN` et certaines jointures externes
  - *Hash Semi Join* & *Hash Anti Join*
- Paramètres :
  - `work_mem` (et `hash_mem_multiplier`)
  - `seq_page_cost` & `random_page_cost` .
  - `enable_nestloop`, `enable_hashjoin`, `enable_mergejoin`

Le choix du type de jointure dépend non seulement des données mises en œuvre, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment des paramètres `work_mem`,

[6] https://dali.bo/j6_html

`hash_mem_multiplier`, `seq_page_cost` et `random_page_cost`.

Nested Loop :

La *Nested Loop* se retrouve principalement dans les jointures de petits ensembles de données. Dans l'exemple suivant, le critère sur `services` ramène très peu de lignes, il ne coûte pas grand-chose d'aller piocher à chaque fois dans l'index de `employes_big`.

```
EXPLAIN SELECT matricule, nom, prenom, nom_service, fonction, localisation
FROM   employes_big emp
JOIN   services ser ON (emp.num_service = ser.num_service)
WHERE  ser.localisation = 'Nantes';
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.42..10053.94 rows=124754 width=46)
   ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
         Filter: ((localisation)::text = 'Nantes'::text)
   ->  Index Scan using employes_big_num_service_idx on employes_big emp
         (cost=0.42..7557.81 rows=249508 width=33)
         Index Cond: (num_service = ser.num_service)
```

Hash Join :

Le *Hash Join* se retrouve lorsque l'ensemble de la table interne est petit. L'optimiseur construit alors une table de hachage avec les valeurs de la ou les colonne(s) de jointure de la table interne. Il réalise ensuite un parcours de la table externe, et, pour chaque ligne de celle-ci, recherche des lignes correspondantes dans la table de hachage, toujours en utilisant la ou les colonne(s) de jointure comme clé

```
EXPLAIN SELECT matricule, nom, prenom, nom_service, fonction, localisation
FROM   employes_big emp
JOIN   services ser ON (emp.num_service = ser.num_service);
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Hash Join  (cost=0.19..8154.54 rows=499015 width=45)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes_big emp  (cost=0.00..5456.55 rows=499015 width=32)
   ->  Hash  (cost=0.14..0.14 rows=4 width=21)
         ->  Seq Scan on services ser  (cost=0.00..0.14 rows=4 width=21)
```

Cette opération réclame de la mémoire de tri, visible avec `EXPLAIN (ANALYZE)` (dans le pire des cas, ce sera un fichier temporaire).

Merge Join :

La jointure par tri-fusion, ou *Merge Join*, prend deux ensembles de données triés en entrée et restitue l'ensemble de données après jointure. Cette jointure est assez lourde à initialiser si PostgreSQL ne peut pas utiliser d'index, mais elle a l'avantage de retourner les données triées directement :

```
EXPLAIN
  SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM   employes_big emp
  JOIN   services_big ser ON (emp.num_service = ser.num_service)
  ORDER BY ser.num_service;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Merge Join  (cost=0.82..20094.77 rows=499015 width=49)
   Merge Cond: (emp.num_service = ser.num_service)
   ->  Index Scan using employes_big_num_service_idx on employes_big emp
       (cost=0.42..13856.65 rows=499015 width=33)
   ->  Index Scan using services_big_pkey on services_big ser
       (cost=0.29..1337.35 rows=40004 width=20)
```

Il s'agit d'un algorithme de jointure particulièrement efficace pour traiter les volumes de données importants, surtout si les données sont pré-triées grâce à l'existence d'un index.

Hash Anti/Semi Join :

Les clauses `EXISTS` et `NOT EXISTS` mettent également en œuvre des algorithmes dérivés de semi et anti-jointures. En voici un exemple avec la clause `EXISTS` :

```
EXPLAIN
  SELECT *
  FROM services s
  WHERE EXISTS (SELECT 1
                FROM employes_big e
                WHERE e.date_embauche>s.date_creation
                  AND s.num_service = e.num_service) ;

                           QUERY PLAN
-----------------------------------------------------------------
 Hash Semi Join  (cost=17841.84..19794.91 rows=1 width=25)
   Hash Cond: (s.num_service = e.num_service)
   Join Filter: (e.date_embauche > s.date_creation)
   ->  Seq Scan on services s  (cost=0.00..1.04 rows=4 width=25)
   ->  Hash  (cost=9654.15..9654.15 rows=499015 width=8)
         ->  Seq Scan on employes_big e
             (cost=0.00..9654.15 rows=499015 width=8)
```

Un plan sensiblement identique s'obtient avec `NOT EXISTS`. Le nœud *Hash Semi Join* est remplacé par *Hash Anti Join* :

```
EXPLAIN
  SELECT *
  FROM services s
  WHERE NOT EXISTS (SELECT 1
                    FROM employes_big e
                    WHERE e.date_embauche>s.date_creation
                      AND s.num_service = e.num_service);

                           QUERY PLAN
-----------------------------------------------------------------
 Hash Anti Join  (cost=17841.84..19794.93 rows=3 width=25)
   Hash Cond: (s.num_service = e.num_service)
   Join Filter: (e.date_embauche > s.date_creation)
   ->  Seq Scan on services s  (cost=0.00..1.04 rows=4 width=25)
   ->  Hash  (cost=9654.15..9654.15 rows=499015 width=8)
         ->  Seq Scan on employes_big e
             (cost=0.00..9654.15 rows=499015 width=8)
```

Hash Join parallélisé :

Ces nœuds sont parallélisables. Pour les *Parallel Hash Join*, la table hachée est même commune pour les différents *workers* [7], et le calcul de celle-ci est réparti sur ceux-ci. Par contraste, pour les nœuds *Merge Join*, *Nested Loop* et *Hash Join* (non complètement parallélisé), seul le parcours de la table externe peut être parallélisé, tandis que la table interne est parcourue (voire hachée) entièrement par chaque worker.

Exemple (testé en version 16.1) :

```
CREATE TABLE foo(id serial, a int);
CREATE TABLE bar(id serial, foo_a int, b int);
INSERT INTO foo(a) SELECT i*2 FROM generate_series(1,1000000) i;
INSERT INTO bar(foo_a, b) SELECT i*2, i%7 FROM generate_series(1,100) i;
VACUUM ANALYZE foo, bar;

EXPLAIN (ANALYZE, VERBOSE, COSTS OFF)
SELECT foo.a, bar.b FROM foo JOIN bar ON (foo.a = bar.foo_a) WHERE a % 3 = 0;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------------
 Gather (actual time=0.192..21.305 rows=33 loops=1)
   Output: foo.a, bar.b
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join (actual time=10.757..16.903 rows=11 loops=3)
         Output: foo.a, bar.b
         Hash Cond: (foo.a = bar.foo_a)
         Worker 0:  actual time=16.118..16.120 rows=0 loops=1
         Worker 1:  actual time=16.132..16.134 rows=0 loops=1
         ->  Parallel Seq Scan on public.foo (actual time=0.009..12.961 rows=111111
↪  loops=3)
               Output: foo.id, foo.a
               Filter: ((foo.a % 3) = 0)
               Rows Removed by Filter: 222222
               Worker 0:  actual time=0.011..12.373 rows=102953 loops=1
               Worker 1:  actual time=0.011..12.440 rows=102152 loops=1
         ->  Hash (actual time=0.022..0.023 rows=100 loops=3)
               Output: bar.b, bar.foo_a
               Buckets: 1024  Batches: 1  Memory Usage: 12kB
               Worker 0:  actual time=0.027..0.027 rows=100 loops=1
               Worker 1:  actual time=0.026..0.026 rows=100 loops=1
               ->  Seq Scan on public.bar (actual time=0.008..0.013 rows=100 loops=3)
                     Output: bar.b, bar.foo_a
                     Worker 0:  actual time=0.012..0.017 rows=100 loops=1
                     Worker 1:  actual time=0.011..0.016 rows=100 loops=1
 Planning Time: 0.116 ms
 Execution Time: 21.321 ms
```

Dans ce plan, la table externe `foo` est lue de manière parallélisée, les trois processus se partageant son contenu. Chacun a sa version de la toute petite table interne `bar`, qui est hachée trois fois (les deux *workers* et le processus principal lisent les 100 lignes).

[7] https://write-skew.blogspot.com/2018/01/parallel-hash-for-postgresql.html

Si `bar` est beaucoup plus grosse que `foo`, le plan bascule sur un *Parallel Hash Join* dont `bar` est cette fois la table externe :

```
INSERT INTO bar(foo_a, b) SELECT i*2, i%7 FROM generate_series(1,300000) i;
VACUUM ANALYZE bar;

EXPLAIN (ANALYZE, VERBOSE, COSTS OFF)
SELECT foo.a, bar.b FROM foo JOIN bar ON (foo.a = bar.foo_a) WHERE a % 3 = 0;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Gather (actual time=69.490..95.741 rows=100033 loops=1)
   Output: foo.a, bar.b
   Workers Planned: 1
   Workers Launched: 1
   ->  Parallel Hash Join (actual time=66.408..84.866 rows=50016 loops=2)
         Output: foo.a, bar.b
         Hash Cond: (bar.foo_a = foo.a)
         Worker 0:  actual time=63.450..83.008 rows=52081 loops=1
         ->  Parallel Seq Scan on public.bar (actual time=0.002..5.332 rows=150050
↪  loops=2)
               Output: bar.id, bar.foo_a, bar.b
               Worker 0:  actual time=0.002..5.448 rows=148400 loops=1
         ->  Parallel Hash (actual time=49.467..49.468 rows=166666 loops=2)
               Output: foo.a
               Buckets: 262144 (originally 8192)  Batches: 4 (originally 1)  Memory
↪  Usage: 5344kB
               Worker 0:  actual time=48.381..48.382 rows=158431 loops=1
               ->  Parallel Seq Scan on public.foo (actual time=0.007..21.265
↪  rows=166666 loops=2)
                     Output: foo.a
                     Filter: ((foo.a % 3) = 0)
                     Rows Removed by Filter: 333334
                     Worker 0:  actual time=0.009..20.909 rows=158431 loops=1
 Planning Time: 0.083 ms
 Execution Time: 97.567 ms
```

Le *Hash* de `foo` se fait par deux processus qui ne traitent cette fois que la moitié du million de lignes, en en filtrant les ⅔ (la dernière ligne indique quasiment le tiers de 500 000). Le coût de démarrage de ce *hash* parallélisé est cependant assez lourd (la jointure ne commence qu'après 66 ms). Pour la même requête, PostgreSQL 10, qui ne connaît pas le *Parallel Hash Join*, procédait à un *Hash Join* classique et prenait 50 % plus longtemps. Précisons encore qu'augmenter `work_mem` ne change pas le plan, mais permet cas d'accélérer un peu les hachages (réduction du nombre de *batches*).

1.8.12 Nœuds de tris et de regroupements

- Deux nœuds de tri :
  - *Sort*
  - *Incremental Sort*
- Regroupement/agrégation :
  - *Aggregate*
  - *Hash Aggregate*
  - *Group Aggregate*
  - *Mixed Aggregate*
  - *Partial/Finalize Aggregate*
- Paramètres :
  - `enable_hashagg`
  - `work_mem` & `hash_mem_multiplier` (v13)

Pour réaliser un tri, PostgreSQL dispose de deux nœuds : *Sort* et *Incremental Sort*. Leur efficacité va dépendre du paramètre `work_mem` qui va définir la quantité de mémoire que PostgreSQL pourra utiliser pour un tri.

Sort :

```
EXPLAIN (ANALYZE)
  SELECT * FROM employes ORDER BY fonction;

                            QUERY PLAN
------------------------------------------------------------------
 Sort  (cost=1.41..1.44 rows=14 width=43)
       (actual time=0.013..0.014 rows=14 loops=1)
   Sort Key: fonction
   Sort Method: quicksort  Memory: 26kB
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
                     (actual time=0.003..0.004 rows=14 loops=1)
 Planning time: 0.021 ms
 Execution time: 0.021 ms
```

Si le tri ne tient pas en mémoire, l'algorithme de tri gère automatiquement le débordement sur disque (26 Mo ici) :

```
EXPLAIN (ANALYZE)
  SELECT * FROM employes_big ORDER BY fonction;

                                QUERY PLAN
-------------------------------------------------------------------------
```

```
 Sort  (cost=70529.24..71776.77 rows=499015 width=40)
   (actual time=252.827..298.948 rows=499015 loops=1)
   Sort Key: fonction
   Sort Method: external sort  Disk: 26368kB
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=40)
                           (actual time=0.003..29.012 rows=499015 loops=1)
 Planning time: 0.021 ms
 Execution time: 319.283 ms
```

Cependant, un tri est coûteux. Donc si un index existe sur le champ de tri, PostgreSQL aura tendance à l'utiliser. Les données sont déjà triées dans l'index, il suffit de le parcourir pour lire les lignes dans l'ordre :

```
EXPLAIN SELECT * FROM employes_big  ORDER BY matricule;
```

```
                  QUERY PLAN
-----------------------------------------------
 Index Scan using employes_pkey on employes
   (cost=0.42..17636.65 rows=499015 width=41)
```

Et ce, dans n'importe quel ordre de tri :

```
EXPLAIN SELECT * FROM employes_big ORDER BY matricule DESC;
```

```
                     QUERY PLAN
-----------------------------------------------------
 Index Scan Backward using employes_pkey on employes
   (cost=0.42..17636.65 rows=499015 width=41)
```

Le choix du type d'opération de regroupement dépend non seulement des données mises en œuvres, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment du paramètre `work_mem`.

Comme vu précédemment, PostgreSQL sait utiliser un index pour trier les données. Cependant, dans certains cas, il ne sait pas utiliser l'index alors qu'il pourrait le faire. Prenons un exemple.

Voici un jeu de données contenant une table à trois colonnes, et un index sur une colonne :

```
DROP TABLE IF exists t1;
CREATE TABLE t1 (c1 integer, c2 integer, c3 integer);
INSERT INTO t1 SELECT i, i+1, i+2 FROM generate_series(1, 10000000) AS i;
CREATE INDEX t1_c2_idx ON t1(c2);
VACUUM ANALYZE t1;
```

PostgreSQL sait utiliser l'index pour trier les données. Par exemple, voici le plan d'exécution pour un tri sur la colonne `c2` (colonne indexée au niveau de l'index `t1_c2_idx`) :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2;
```

```
                               QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t1_c2_idx on t1  (cost=0.43..313749.06 rows=10000175 width=12)
                            (actual time=0.016..1271.115 rows=10000000 loops=1)
   Buffers: shared hit=81380
 Planning Time: 0.173 ms
 Execution Time: 1611.868 ms
```

Par contre, si on essaie de trier par rapport aux colonnes `c2` et `c3`, les versions 12 et antérieures ne savent pas utiliser l'index, comme le montre ce plan d'exécution :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather Merge  (cost=697287.64..1669594.86 rows=8333480 width=12)
           (actual time=1331.307..3262.511 rows=10000000 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=54149, temp read=55068 written=55246
   ->  Sort  (cost=696287.62..706704.47 rows=4166740 width=12)
         (actual time=1326.112..1766.809 rows=3333333 loops=3)
         Sort Key: c2, c3
         Sort Method: external merge  Disk: 61888kB
         Worker 0:  Sort Method: external merge  Disk: 61392kB
         Worker 1:  Sort Method: external merge  Disk: 92168kB
         Buffers: shared hit=54149, temp read=55068 written=55246
         ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                        (actual time=0.015..337.901 rows=3333333 loops=3)
               Buffers: shared hit=54055
 Planning Time: 0.068 ms
 Execution Time: 3716.541 ms
```

Comme PostgreSQL ne sait pas utiliser un index pour réaliser ce tri, il passe par un parcours de table (parallélisé dans le cas présent), puis effectue le tri, ce qui prend beaucoup de temps, encore plus s'il faut déborder sur disque. La durée d'exécution a plus que doublé.

Incremental Sort :

La version 13 est beaucoup plus maline à cet égard. Elle est capable d'utiliser l'index pour faire un premier tri des données (sur la colonne `c2` d'après notre exemple), puis elle trie les données du résultat par rapport à la colonne `c3` :

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
             (actual time=0.082..2427.099 rows=10000000 loops=1)
   Sort Key: c2, c3
   Presorted Key: c2
   Full-sort Groups: 312500  Sort Method: quicksort  Average Memory: 26kB  Peak
↪  Memory: 26kB
   Buffers: shared hit=81387
   ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000
↪  width=12)
                                  (actual time=0.007..1263.517 rows=10000000 loops=1)
         Buffers: shared hit=81380
 Planning Time: 0.059 ms
 Execution Time: 2766.530 ms
```

La requête en version 12 prenait 3,7 secondes. La version 13 n'en prend que 2,7 secondes. On remarque un nouveau type de nœud, le *Incremental Sort*, qui s'occupe de re-trier les données après un renvoi de données triées, grâce au parcours d'index. Le plan distingue bien la clé déjà triée (`c2`) et celles à trier (`c2, c3`).

L'apport en performance est déjà très intéressant, mais il devient remarquable si on utilise une clause `LIMIT`. Voici le résultat en version 12 :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3 LIMIT 10;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=186764.17..186765.34 rows=10 width=12)
      (actual time=718.576..724.791 rows=10 loops=1)
   Buffers: shared hit=54149
   ->  Gather Merge  (cost=186764.17..1159071.39 rows=8333480 width=12)
                        (actual time=718.575..724.788 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=54149
         ->  Sort  (cost=185764.15..196181.00 rows=4166740 width=12)
                  (actual time=716.606..716.608 rows=10 loops=3)
               Sort Key: c2, c3
               Sort Method: top-N heapsort  Memory: 25kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 25kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
               Buffers: shared hit=54149
              ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                                (actual time=0.010..347.085 rows=3333333 loops=3)
                     Buffers: shared hit=54055
 Planning Time: 0.044 ms
 Execution Time: 724.818 ms
```

Et celui en version 13 :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.48..1.24 rows=10 width=12) (actual time=0.027..0.029 rows=10 loops=1)
   Buffers: shared hit=4
   ->  Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
                           (actual time=0.027..0.027 rows=10 loops=1)
         Sort Key: c2, c3
         Presorted Key: c2
         Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 25kB  Peak
↪  Memory: 25kB
         Buffers: shared hit=4
         ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000
↪  width=12)
                                              (actual time=0.012..0.014 rows=11 loops=1)
               Buffers: shared hit=4
 Planning Time: 0.052 ms
 Execution Time: 0.038 ms
```

La requête passe donc de 724 ms à 0,029 ms.

En version 16, l'*Incremental Sort* peut aussi servir à accélérer les `DISTINCT` :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT DISTINCT c2,c1,c3 FROM t1;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
```

```
 Unique (actual time=39.208..2479.229 rows=10000000 loops=1)
   Buffers: shared read=81380
   ->  Incremental Sort (actual time=39.206..1841.165 rows=10000000 loops=1)
         Sort Key: c2, c1, c3
         Presorted Key: c2
         Full-sort Groups: 312500  Sort Method: quicksort  Average Memory: 26kB  Peak
↪  Memory: 26kB
         Buffers: shared read=81380
         ->  Index Scan using t1_c2_idx on t1 (actual time=39.182..949.921
↪  rows=10000000 loops=1)
               Buffers: shared read=81380
 Planning:
   Buffers: shared read=3
 Planning Time: 0.274 ms
 Execution Time: 2679.447 ms
```

Cela devrait accélérer de nombreuses requêtes qui possèdent abusivement une clause `DISTINCT` ajoutée par un ETL, si le premier champ du `DISTINCT` est par chance indexé, comme ici :

```
EXPLAIN (COSTS OFF) SELECT DISTINCT date_commande, *  FROM commandes ;

                          QUERY PLAN
---------------------------------------------------------------
 Unique
   ->  Incremental Sort
         Sort Key: date_commande, numero_commande, client_id, mode_expedition,
↪  commentaire
         Presorted Key: date_commande
         ->  Index Scan using commandes_date_commande_idx on commandes
```

Aggregate :

Concernant les opérations d'agrégations, on retrouve un nœud de type *Aggregate* lorsque la requête réalise une opération d'agrégation simple, sans regroupement :

```
EXPLAIN SELECT count(*) FROM employes;

                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=1.18..1.19 rows=1 width=8)
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=0)
```

Hash Aggregate :

Si l'optimiseur estime que l'opération d'agrégation tient en mémoire (paramètre `work_mem`), il va utiliser un nœud de type *HashAggregate* :

```
EXPLAIN SELECT fonction, count(*) FROM employes GROUP BY fonction;

                           QUERY PLAN
---------------------------------------------------------------
 HashAggregate  (cost=1.21..1.27 rows=6 width=20)
   Group Key: fonction
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=12)
```

Avant la version 13, l'inconvénient de ce nœud est que sa consommation mémoire n'est pas limitée par `work_mem`, il continuera malgré tout à allouer de la mémoire. Dans certains cas, heureusement très rares, l'optimiseur peut se tromper suffisamment pour qu'un nœud *HashAggregate* consomme plusieurs gigaoctets de mémoire et sature ainsi la mémoire du serveur.

La version 13 améliore cela en utilisant le disque à partir du moment où la mémoire nécessaire dépasse la multiplication de la valeur du paramètre `work_mem` et celle du paramètre `hash_mem_multiplier` (2 par défaut à partir de la version 15, 1 auparavant). La requête sera plus lente, mais la mémoire ne sera pas saturée.

Group Aggregate :

Lorsque l'optimiseur estime que le volume de données à traiter ne tient pas dans `work_mem` ou quand il peut accéder aux données pré-triées, il utilise plutôt l'algorithme *GroupAggregate* :

```
EXPLAIN SELECT matricule, count(*) FROM employes_big GROUP BY matricule;

                          QUERY PLAN
---------------------------------------------------------------
 GroupAggregate  (cost=0.42..20454.87 rows=499015 width=12)
   Group Key: matricule
   Planned Partitions: 16
   ->  Index Only Scan using employes_big_pkey on employes_big
       (cost=0.42..12969.65 rows=499015 width=4)
```

Mixed Aggregate :

Le *Mixed Aggregate* est très efficace pour les clauses `GROUP BY GROUPING SETS` ou `GROUP BY CUBE` grâce à l'utilisation de *hashs* :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT manager, fonction, num_service, COUNT(*)
  FROM employes_big
  GROUP BY CUBE(manager,fonction,num_service) ;

                          QUERY PLAN
---------------------------------------------------------------
 MixedAggregate  (cost=0.00..34605.17 rows=27 width=27)
        (actual time=581.562..581.573 rows=51 loops=1)
   Hash Key: manager, fonction, num_service
   Hash Key: manager, fonction
   Hash Key: manager
   Hash Key: fonction, num_service
   Hash Key: fonction
   Hash Key: num_service, manager
   Hash Key: num_service
   Group Key: ()
   Batches: 1  Memory Usage: 96kB
   Buffers: shared hit=4664
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=19)
                        (actual time=0.015..35.840 rows=499015 loops=1)
```

```
        Buffers: shared hit=4664
 Planning time: 0.223 ms
 Execution time: 581.671 ms
```

(Comparer avec le plan et le temps obtenus auparavant, que l'on peut retrouver avec `SET enable_hashagg TO off;`)

Le calcul d'un agrégat peut être parallélisé. Dans ce cas, deux nœuds sont utilisés : un pour le calcul partiel de chaque processus (*Partial Aggregate*), et un pour le calcul final (*Finalize Aggregate*). Voici un exemple de plan :

```
EXPLAIN (ANALYZE,COSTS OFF)
  SELECT date_embauche, count(*), min(date_embauche), max(date_embauche)
  FROM employes_big
  GROUP BY date_embauche;

                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize GroupAggregate (actual time=92.736..92.740 rows=7 loops=1)
   Group Key: date_embauche
   ->  Sort (actual time=92.732..92.732 rows=9 loops=1)
         Sort Key: date_embauche
         Sort Method: quicksort  Memory: 25kB
         ->  Gather (actual time=92.664..92.673 rows=9 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial HashAggregate
                   (actual time=89.531..89.532 rows=3 loops=3)
                     Group Key: date_embauche
                     ->  Parallel Seq Scan on employes_big
                         (actual time=0.011..35.801 rows=166338 loops=3)
 Planning time: 0.127 ms
 Execution time: 95.601 ms
```

1.8.13 Les autres nœuds

- *Limit*
- *Unique* (`DISTINCT`)
- *Append* (`UNION ALL`), *Except*, *Intersect*
- *Gather* (parallélisme)
- *InitPlan*, *Subplan*, etc.
- *Memoize* (14+)

Limit :

On rencontre le nœud `Limit` lorsqu'on limite le résultat avec l'ordre `LIMIT` :

```
EXPLAIN SELECT * FROM employes_big LIMIT 1;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.00..0.02 rows=1 width=40)
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=40)
```

Le nœud *Sort* utilisera dans ce cas une méthode de tri appelée *top-N heapsort* qui permet d'optimiser le tri pour retourner les n premières lignes :

```
EXPLAIN ANALYZE
  SELECT * FROM employes_big ORDER BY fonction LIMIT 5;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=17942.61..17942.62 rows=5 width=40)
      (actual time=80.359..80.360 rows=5 loops=1)
   ->  Sort  (cost=17942.61..19190.15 rows=499015 width=40)
                (actual time=80.358..80.359 rows=5 loops=1)
         Sort Key: fonction
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on employes_big
             (cost=0.00..9654.15 rows=499015 width=40)
             (actual time=0.005..27.506 rows=499015 loops=1)
 Planning time: 0.035 ms
 Execution time: 80.375 ms
```

Unique :

On retrouve le nœud *Unique* lorsque l'on utilise `DISTINCT` pour dédoublonner le résultat d'une requête :

```
EXPLAIN SELECT DISTINCT matricule FROM employes_big;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Unique  (cost=0.42..14217.19 rows=499015 width=4)
   ->  Index Only Scan using employes_big_pkey on employes_big
       (cost=0.42..12969.65 rows=499015 width=4)
```

On le verra plus loin, il est souvent plus efficace d'utiliser `GROUP BY` pour dédoublonner les résultats d'une requête.

Append, Except, Intersect :

Les nœuds *Append*, *Except* et *Intersect* se rencontrent avec les opérateurs ensemblistes `UNION`, `EXCEPT` et `INTERSECT`. Par exemple, avec `UNION ALL` :

```
EXPLAIN
  SELECT * FROM employes
  WHERE num_service = 2
  UNION ALL
  SELECT * FROM employes
  WHERE num_service = 4;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=0.00..2.43 rows=8 width=43)
   ->  Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
         Filter: (num_service = 2)
   ->  Seq Scan on employes employes_1  (cost=0.00..1.18 rows=5 width=43)
         Filter: (num_service = 4)
```

InitPlan :

Le nœud *InitPlan* apparaît lorsque PostgreSQL a besoin d'exécuter une première sous-requête pour ensuite exécuter le reste de la requête. Il est assez rare :

```
EXPLAIN
  SELECT *,
    (SELECT nom_service FROM services WHERE num_service=1)
  FROM employes WHERE num_service = 1;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on employes  (cost=1.05..2.23 rows=2 width=101)
   Filter: (num_service = 1)
   InitPlan 1 (returns $0)
     ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=10)
           Filter: (num_service = 1)
```

SubPlan :

Le nœud *SubPlan* est utilisé lorsque PostgreSQL a besoin d'exécuter une sous-requête pour filtrer les données :

```
EXPLAIN
  SELECT * FROM employes
    WHERE num_service NOT IN (SELECT num_service FROM services
                              WHERE nom_service = 'Consultants');
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on employes  (cost=1.05..2.23 rows=7 width=43)
   Filter: (NOT (hashed SubPlan 1))
   SubPlan 1
     ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=4)
           Filter: ((nom_service)::text = 'Consultants'::text)
```

Gather :

Le nœud *Gather* n'apparaît que s'il y a du parallélisme. Il est utilisé comme nœud de rassemblement des données.

Memoize :

Apparu avec PostgreSQL 14, le nœud *Memoize* est un cache de résultat qui permet d'optimiser les performances d'autres nœuds en mémorisant des données qui risquent d'être accédées plusieurs fois de suite. Pour le moment, ce nœud n'est utilisable que pour les données de l'ensemble interne d'un *Nested Loop*.

D'autres types de nœuds peuvent également être trouvés dans les plans d'exécution. L'annexe décrit tous ces nœuds en détail.

1.9 PROBLÈMES LES PLUS COURANTS

- L'optimiseur se trompe parfois
  - mauvaises statistiques
  - écriture particulière de la requête
  - problèmes connus de l'optimiseur

L'optimiseur de PostgreSQL est sans doute la partie la plus complexe de PostgreSQL. Il se trompe rarement, mais certains facteurs peuvent entraîner des temps d'exécution très lents, voire catastrophiques de certaines requêtes.

1.9.1 Statistiques pas à jour

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 : 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 réinitialise. (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.)

1.9.2 Colonnes corrélées

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

- Si `c1 = 1` pour 20 % des lignes
- et `c2 = 1` 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 ;
```

PostgreSQL conserve des statistiques par colonne simple. Mais dans la vie, les valeurs ne sont pas indépendantes. Dans cet exemple, les calculs d'estimation du résultat seront mauvais :

```
CREATE TABLE corr1 AS
SELECT mod(i,5) AS c1 ,mod(i,10) AS c2, i FROM generate_series (1,100000) i;
CREATE INDEX ON corr1 (c1,c2) ;
SELECT c1,c2, count(*) FROM corr1 GROUP BY 1,2 ORDER BY 1,2;
```

```
 c1 | c2 | count
----+----+-------
  0 |  0 | 10000
  0 |  5 | 10000
  1 |  1 | 10000
  1 |  6 | 10000
  2 |  2 | 10000
…
(10 lignes)
```

Dans l'exemple ci-dessus, le planificateur sait que l'estimation pour `c1=1` est de 20 % et que l'estimation pour `c2=1` est de 10 %. Par contre, il n'a aucune idée de l'estimation pour `c1=1 AND c2=1`. Faute de mieux, il multiplie les deux estimations et obtient 2 % (20 % × 10 %), soit environ 2000 lignes, ce qui est faux :

```
ANALYZE corr1 ;
EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 1 ;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=29.40..636.28 rows=2059 width=12)
                     (actual time=0.653..3.034 rows=10000 loops=1)
   Recheck Cond: ((c1 = 1) AND (c2 = 1))
   Heap Blocks: exact=541
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..28.88 rows=2059 width=0)
```

```
                                    (actual time=0.480..0.481 rows=10000 loops=1)
         Index Cond: ((c1 = 1) AND (c2 = 1))
```

Pour corriger cela, il faut générer des statistiques sur plusieurs colonnes spécifiques. Ce 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 */

EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 1 ;

                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=139.85..867.39 rows=10103 width=12)
                       (actual time=0.748..3.505 rows=10000 loops=1)
   Recheck Cond: ((c1 = 1) AND (c2 = 1))
   Heap Blocks: exact=541
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..137.32 rows=10103 width=0)
                                     (actual time=0.563..0.564 rows=10000 loops=1)
         Index Cond: ((c1 = 1) AND (c2 = 1))
```

Dans ce cas précis, de meilleures statistiques ne changent pas le plan. Par contre, avec le critère `c1 = 1 AND c2 = 2` (qui ne renvoie rien), les meilleures statistique permettent de basculer du même *Bitmap Scan* que ci-dessus à un *Index Scan* plus léger :

```
EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 2 ;

                            QUERY PLAN
------------------------------------------------------------------
 Index Scan using corr1_c1_c2_idx on corr1  (cost=0.29..8.31 rows=1 width=12)
                                   (actual time=0.010..0.011 rows=0 loops=1)
   Index Cond: ((c1 = 1) AND (c2 = 2))
```

1.9.3 La jointure de trop

- PostgreSQL choisit l'ordre des jointures
  - uniquement pour les X premières tables
  - où X = `join_collapse_limit` (défaut : 8)
- Les jointures supplémentaires sont ajoutées *après*
- … d'où plans non optimaux
- → augmenter `join_collapse_limit` si nécessaire (12-15)
  - ainsi que `from_collapse_limit`

Voici un exemple complet de ce problème. Disons que `join_collapse_limit` est configuré à 2 (le défaut est en réalité 8).

```
SET join_collapse_limit TO 2 ;
```

Nous allons déjà créer deux tables et les peupler avec 1 million de lignes chacune :

```
CREATE TABLE t1 (id integer);
INSERT INTO t1 SELECT generate_series(1, 1000000);
CREATE TABLE t2 (id integer);
INSERT INTO t2 SELECT generate_series(1, 1000000);
ANALYZE;
```

Maintenant, nous allons demander le plan d'exécution pour une jointure entre les deux tables :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=30832.00..70728.00 rows=1000000 width=8)
            (actual time=2355.012..6141.672 rows=1000000 loops=1)
   Hash Cond: (t1.id = t2.id)
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=4)
                       (actual time=0.012..1137.629 rows=1000000 loops=1)
   ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
             (actual time=2354.750..2354.753 rows=1000000 loops=1)
         Buckets: 131072  Batches: 16  Memory Usage: 3227kB
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                             (actual time=0.008..1144.492 rows=1000000 loops=1)
 Planning Time: 0.095 ms
 Execution Time: 7246.491 ms
```

PostgreSQL choisit de lire la table `t2`, de remplir une table de hachage avec le résultat de cette lecture, puis de parcourir la table `t1`, et enfin de tester la condition de jointure grâce à la table de hachage.

Ajoutons maintenant une troisième table, sans données cette fois :

```
CREATE TABLE t3 (id integer);
```

Et ajoutons une jointure à la requête précédente. Cela nous donne cette requête :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id
  JOIN t3 ON t2.id=t3.id;
```

Son plan d'exécution, avec la configuration par défaut de PostgreSQL, sauf le `join_collapse_limit` à 2, est :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=77972.88..80334.59 rows=2550 width=12)
         (actual time=2902.385..2913.956 rows=0 loops=1)
```

```
   Workers Planned: 2
   Workers Launched: 2
   ->  Merge Join  (cost=76972.88..79079.59 rows=1062 width=12)
                (actual time=2894.440..2894.615 rows=0 loops=3)
         Merge Cond: (t1.id = t3.id)
         ->  Sort  (cost=76793.10..77834.76 rows=416667 width=8)
                  (actual time=2894.405..2894.572 rows=1 loops=3)
               Sort Key: t1.id
               Sort Method: external merge  Disk: 5912kB
               Worker 0:  Sort Method: external merge  Disk: 5960kB
               Worker 1:  Sort Method: external merge  Disk: 5848kB
               ->  Parallel Hash Join  (cost=15428.00..32202.28 rows=416667 width=8)
                                  (actual time=1892.071..2400.515 rows=333333 loops=3)
                     Hash Cond: (t1.id = t2.id)
                     ->  Parallel Seq Scan on t1  (cost=0.00..8591.67 rows=416667
↪  width=4)
                                               (actual time=0.007..465.746 rows=333333
↪  loops=3)
                     ->  Parallel Hash  (cost=8591.67..8591.67 rows=416667 width=4)
                              (actual time=950.509..950.514 rows=333333 loops=3)
                           Buckets: 131072  Batches: 16  Memory Usage: 3520kB
                           ->  Parallel Seq Scan on t2  (cost=0.00..8591.67
↪  rows=416667 width=4)
                                                    (actual time=0.017..471.653 rows=333333
↪  loops=3)
         ->  Sort  (cost=179.78..186.16 rows=2550 width=4)
                   (actual time=0.028..0.032 rows=0 loops=3)
               Sort Key: t3.id
               Sort Method: quicksort  Memory: 25kB
               Worker 0:  Sort Method: quicksort  Memory: 25kB
               Worker 1:  Sort Method: quicksort  Memory: 25kB
               ->  Seq Scan on t3  (cost=0.00..35.50 rows=2550 width=4)
                                   (actual time=0.019..0.020 rows=0 loops=3)
 Planning Time: 0.120 ms
 Execution Time: 2914.661 ms
```

En effet, dans ce cas, PostgreSQL va trier les jointures sur les 2 premières tables (soit `t1` et `t2`), et il ajoutera ensuite les autres jointures dans l'ordre indiqué par la requête. Donc, ici, il joint `t1` et `t2`, puis le résultat avec `t3`, ce qui nous donne une requête exécutée en un peu moins de 3 secondes. C'est beaucoup quand on considère que la table `t3` est vide et que le résultat sera forcément vide lui aussi (l'optimiseur a certes estimé trouver 2550 lignes dans `t3`, mais cela reste très faible par rapport aux autres tables).

Maintenant, voici le plan d'exécution pour la même requête avec un `join_collapse_limit` à 3 :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id
  JOIN t3 ON t2.id=t3.id ;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=35861.44..46281.24 rows=2550 width=12)
         (actual time=14.943..15.617 rows=0 loops=1)
```

```
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join  (cost=34861.44..45026.24 rows=1062 width=12)
                  (actual time=0.119..0.134 rows=0 loops=3)
         Hash Cond: (t2.id = t1.id)
         ->  Parallel Seq Scan on t2  (cost=0.00..8591.67 rows=416667 width=4)
                                      (actual time=0.010..0.011 rows=1 loops=3)
         ->  Hash  (cost=34829.56..34829.56 rows=2550 width=8)
                   (actual time=0.011..0.018 rows=0 loops=3)
               Buckets: 4096  Batches: 1  Memory Usage: 32kB
               ->  Hash Join  (cost=30832.00..34829.56 rows=2550 width=8)
                              (actual time=0.008..0.013 rows=0 loops=3)
                     Hash Cond: (t3.id = t1.id)
                     ->  Seq Scan on t3  (cost=0.00..35.50 rows=2550 width=4)
                                         (actual time=0.006..0.007 rows=0 loops=3)
                     ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
                               (never executed)
                           ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000
↪  width=4)
                                               (never executed)
 Planning Time: 0.331 ms
 Execution Time: 15.662 ms
```

Déjà, on voit que la planification a pris plus de temps. La durée reste très basse (0,3 milliseconde) ceci dit.

Cette fois, PostgreSQL commence par joindre `t3` à `t1`. Comme `t3` ne contient aucune ligne, `t1` n'est même pas parcourue (texte `never executed`) et le résultat de cette première jointure renvoie 0 lignes. De ce fait, la création de la table de hachage est très rapide. La table de hachage étant vide, le parcours de `t2` est abandonné après la première ligne lue. Cela nous donne une requête exécutée en 15 millisecondes.

Une configuration adéquate de `join_collapse_limit` est donc essentielle pour de bonnes performances, notamment sur les requêtes réalisant un grand nombre de jointures.

Il est courant de monter `join_collapse_limit` à 12 si l'on a des requêtes avec autant de tables (y compris celles des vues).

Il existe un paramètre très voisin, `from_collapse_limit`, qui définit à quelle profondeur « aplatir » les sous-requêtes. On le monte à la même valeur que `join_collapse_limit`.

Comme le temps de planification augmente très vite avec le nombre de tables, il vaut mieux ne pas monter `join_collapse_limit` beaucoup plus haut sans tester que ce n'est pas contre-productif. Dans la session concernée, il reste possible de définir :

```
SET join_collapse_limit = … ;
SET from_collapse_limit = … ;
```

À l'inverse, la valeur 1 permet de forcer les jointures dans l'ordre de la clause `FROM`, ce qui est à réserver aux cas désespérés.

Au-delà de 12 tables intervient encore un autre mécanisme, l'optimiseur génétique (GEQO[8]). Pour limiter le nombre de plans étudiés, seul un échantillonnage aléatoire est testé puis recombiné.

1.9.4 Prédicats et statistiques

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

- L'optimiseur n'a pas de statistiques sur le résultat de la fonction `extract`
- Il estime la sélectivité du prédicat à 0,5 % …
- `CREATE STATISTIC` (v14)

Lorsque les valeurs des colonnes sont transformées par un calcul ou par une fonction, l'optimiseur n'a aucun moyen de connaître la sélectivité d'un prédicat. Il utilise donc une estimation codée en dur dans le code de l'optimiseur : 0,5 % du nombre de lignes de la table. Dans la requête suivante, l'optimiseur estime alors que la requête va ramener 2495 lignes :

```
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)
```

Ces 2495 lignes correspondent à 0,5 % de la table `employes_big`.

Nous avons vu qu'il est préférable de réécrire la requête de manière à pouvoir utiliser les statistiques existantes sur la colonne, mais ce n'est pas toujours aisé ou même possible.

Dans ce cas, on peut se rabattre sur l'ordre `CREATE STATISTICS`. Nous avons vu plus haut qu'il permet de calculer des statistiques sur des résultats d'expressions (ne pas oublier `ANALYZE`).

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

Les estimations du plan sont désormais correctes :

[8] https://docs.postgresql.fr/current/geqo-pg-intro.html

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

Avant PostgreSQL 14, il est nécessaire de créer un index fonctionnel sur l'expression pour que des statistiques soient calculées.

1.9.5 Problème avec LIKE

```
SELECT * FROM t1 WHERE c2 LIKE 'x%';
```

- PostgreSQL peut utiliser un index dans ce cas
- **MAIS** si l'encodage n'est pas C
  - déclarer l'index avec une classe d'opérateur
  - `varchar_pattern_ops` / `text_pattern_ops`, etc.

```
CREATE INDEX ON matable (champ_texte varchar_pattern_ops);
```

- Outils pour `LIKE '%mot%'` :
  - `pg_trgm`,
  - *Full Text Search*

Il existe cependant une spécificité à PostgreSQL : dans le cas d'une recherche avec préfixe, il peut utiliser directement un index sur la colonne si l'encodage est « C ». Or le collationnement par défaut d'une base est presque toujours `en_US.UTF-8` ou `fr_FR.UTF-8`, selon les choix à l'installation de l'OS ou de PostgreSQL :

```
\l
                      Liste des bases de données
    Nom    | Propriétaire | Encodage | Collationnement | Type caract. | …
-----------+--------------+----------+-----------------+--------------+---
 pgbench   | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
 postgres  | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
 template0 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  | …
 template1 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  | …
 textes_10 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
```

Il faut alors utiliser une classe d'opérateur lors de la création de l'index. Cela donnera par exemple :

```
CREATE INDEX i1 ON t1 (c2 varchar_pattern_ops);
```

Ce n'est qu'à cette condition qu'un `LIKE 'mot%'` pourra utiliser l'index. Par contre, l'opérateur `varchar_pattern_ops` ne permet pas de trier (`ORDER BY` notamment), faute de collation, il faudra

donc peut-être indexer deux fois la colonne.

Un encodage C (purement anglophone) ne nécessite pas l'ajout d'une classe d'opérateurs `varchar_pattern_ops`.

Pour les recherches à l'intérieur d'un texte (`LIKE '%mot%'`), il existe deux autres options :

- `pg_trgm` est une extension permettant de faire des recherches de type par trigramme et un index GIN ou GiST ;
- la *Full Text Search* est une fonctionnalité extrêmement puissante, mais avec une syntaxe différente.

1.9.6 DELETE lent

- `DELETE` lent
- Généralement un problème de clé étrangère

```
Delete  (actual time=111.251..111.251 rows=0 loops=1)
  ->  Hash Join  (actual time=1.094..21.402 rows=9347 loops=1)
        ->  Seq Scan on lot_a30_descr_lot
            (actual time=0.007..11.248 rows=34934 loops=1)
        ->  Hash  (actual time=0.501..0.501 rows=561 loops=1)
              ->  Bitmap Heap Scan on lot_a10_pdl
                  (actual time=0.121..0.326 rows=561 loops=1)
                    Recheck Cond: (id_fantoir_commune = 320013)
                    ->  Bitmap Index Scan on...
                        (actual time=0.101..0.101 rows=561 loops=1)
                          Index Cond: (id_fantoir_commune = 320013)
Trigger for constraint fk_lotlocal_lota30descrlot:
  time=1010.358 calls=9347
Trigger for constraint fk_nonbatia21descrsuf_lota30descrlot:
  time=2311695.025 calls=9347
Total runtime: 2312835.032 ms
```

Parfois, un `DELETE` peut prendre beaucoup de temps à s'exécuter. Cela peut être dû à un grand nombre de lignes à supprimer. Cela peut aussi être dû à la vérification des contraintes étrangères.

Dans l'exemple ci-dessus, le `DELETE` met 38 minutes à s'exécuter (2 312 835 ms), pour ne supprimer aucune ligne. En fait, c'est la vérification de la contrainte `fk_nonbatia21descrsuf_lota30descrlot` qui prend pratiquement tout le temps. C'est d'ailleurs pour cette raison qu'il est recommandé de positionner des index sur les clés étrangères, car cet index permet d'accélérer la recherche liée à la contrainte.

Attention donc aux contraintes de clés étrangères pour les instructions DML !

1.9.7 Dédoublonnage

```
SELECT   DISTINCT t1.*   FROM t1 JOIN t2 ON (t1.id=t2.t1_id);
```

- `DISTINCT` est souvent utilisé pour dédoublonner les lignes
  - souvent utilisé de manière abusive
  - tri !!
  - barrière à l'optimisation
- Penser à :
  - `DISTINCT ON`
  - `GROUP BY`
- Une clé primaire permet de dédoublonner efficacement

Un `DISTINCT` est une opération coûteuse à cause du tri nécessaire. Il est fréquent de le voir ajouté abusivement, « par prudence » ou pour compenser un bug de jointure. De plus il constitue une « barrière à l'optimisation » s'il s'agit d'une partie de requête.

Si le résultat contient telles quelles les clés primaires de toutes les tables jointes, le `DISTINCT` est mathématiquement inutile ! PostgreSQL ne sait malheureusement pas repérer tout seul ce genre de cas.

Quand le dédoublonnage est justifié, il faut savoir qu'il y a deux alternatives principales au `DISTINCT`. Leurs efficacités relatives sont très dépendantes du paramétrage mémoire (`work_mem`) ou des volumétries, ou encore de la présence d'index permettant d'éviter le tri.

- Un `GROUP BY` des colonnes retournées est fastidieux à coder, mais donne parfois un plan efficace. Cette astuce est plus fréquemment utile avant PostgreSQL 13.
- Une autre possibilité est d'utiliser la syntaxe `DISTINCT ON (champs)`, qui renvoie la première ligne rencontrée sur une clé fournie (documentation[9]).

Exemples (sous PostgreSQL 15.2, configuration par défaut sur une petite configuration, cache chaud) :

Il s'agit ici d'afficher la liste des membres des différents services.

- Plan avec `DISTINCT` : notez le tri sur disque.

[9] https://docs.postgresql.fr/current/sql-select.html#SQL-DISTINCT

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT DISTINCT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Unique (actual time=2930.441..4765.048 rows=499015 loops=1)
   ->  Sort (actual time=2930.435..3351.819 rows=499015 loops=1)
         Sort Key: employes_big.matricule, employes_big.nom, employes_big.prenom,
↪  employes_big.fonction, employes_big.manager, employes_big.date_embauche,
↪  employes_big.num_service, services.nom_service, services.localisation,
↪  services.departement
         Sort Method: external merge  Disk: 38112kB
         ->  Hash Join (actual time=0.085..1263.867 rows=499015 loops=1)
               Hash Cond: (employes_big.num_service = services.num_service)
               ->  Seq Scan on employes_big (actual time=0.030..273.710 rows=499015
↪  loops=1)
               ->  Hash (actual time=0.032..0.035 rows=4 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on services (actual time=0.014..0.020 rows=4 loops=1)
 Planning Time: 0.973 ms
 Execution Time: 4938.634 ms
```

- Réécriture avec `GROUP BY` : il n'y a pas de gain en temps dans ce cas précis, mais il n'y a plus de tri sur disque, car l'index sur la clé primaire est utilisé. Noter que PostgreSQL est assez malin pour repérer les clés primaire (ici `matricule` et `num_service`). Il évite alors d'inclure dans les données à regrouper ces clés, et tous les champs de la première table.

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service)
GROUP BY
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Group (actual time=0.409..5067.075 rows=499015 loops=1)
   Group Key: employes_big.matricule, services.nom_service, services.localisation,
↪  services.departement
   ->  Incremental Sort (actual time=0.405..3925.924 rows=499015 loops=1)
         Sort Key: employes_big.matricule, services.nom_service,
↪  services.localisation, services.departement
         Presorted Key: employes_big.matricule
         Full-sort Groups: 15595  Sort Method: quicksort  Average Memory: 28kB  Peak
↪  Memory: 28kB
         ->  Nested Loop (actual time=0.092..2762.395 rows=499015 loops=1)
```

```
               ->  Index Scan using employes_big_pkey on employes_big (actual
↪  time=0.050..861.828 rows=499015 loops=1)
               ->  Memoize (actual time=0.001..0.001 rows=1 loops=499015)
                     Cache Key: employes_big.num_service
                     Cache Mode: logical
                     Hits: 499011  Misses: 4  Evictions: 0  Overflows: 0  Memory
↪  Usage: 1kB
                     ->  Index Scan using services_pkey on services (actual
↪  time=0.012..0.012 rows=1 loops=4)
                           Index Cond: (num_service = employes_big.num_service)
 Planning Time: 0.900 ms
 Execution Time: 5190.287 ms
```

- Si l'on monte `work_mem` de 4 à 100 Mo, les deux versions basculent sur ce plan, ici plus efficace, qui n'utilise plus l'index, mais ne trie qu'en mémoire, avec la même astuce que ci-dessus.

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate (actual time=3122.612..3849.449 rows=499015 loops=1)
   Group Key: employes_big.matricule, services.nom_service, services.localisation,
↪  services.departement
   Batches: 1  Memory Usage: 98321kB
   ->  Hash Join (actual time=0.136..1354.195 rows=499015 loops=1)
         Hash Cond: (employes_big.num_service = services.num_service)
        ->  Seq Scan on employes_big (actual time=0.050..322.423 rows=499015 loops=1)
         ->  Hash (actual time=0.042..0.046 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on services (actual time=0.020..0.026 rows=4 loops=1)
 Planning Time: 0.967 ms
 Execution Time: 3970.353 ms
```

- La technique la plus propre consiste à indiquer quel est le critère fonctionnel pour dédupliquer. Ici, la structure des tables impose qu'il n'y ait qu'un service par matricule, ce n'est pas évident en regardant la requête. Les index suffisent à ramener une ligne de `service` pour chacune d'`employes_big`.

```
RESET work_mem ;

EXPLAIN (COSTS OFF, ANALYZE)
SELECT DISTINCT ON (matricule)
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Unique (actual time=0.093..3812.414 rows=499015 loops=1)
   ->  Nested Loop (actual time=0.090..2741.919 rows=499015 loops=1)
         ->  Index Scan using employes_big_pkey on employes_big (actual
↪  time=0.049..847.356 rows=499015 loops=1)
         ->  Memoize (actual time=0.001..0.001 rows=1 loops=499015)
               Cache Key: employes_big.num_service
               Cache Mode: logical
```

```
              Hits: 499011  Misses: 4  Evictions: 0  Overflows: 0  Memory Usage: 1kB
              ->  Index Scan using services_pkey on services (actual
↪  time=0.012..0.012 rows=1 loops=4)
                    Index Cond: (num_service = employes_big.num_service)
Planning Time: 0.711 ms
Execution Time: 3982.201 ms
```

- Le plus propre et performant reste tout de même de remarquer que les deux clés primaires sont dans le résultat, et que le `DISTINCT` est inutile. La jointure peut se faire de manière plus classique.

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join (actual time=0.083..1014.796 rows=499015 loops=1)
   Hash Cond: (employes_big.num_service = services.num_service)
   ->  Seq Scan on employes_big (actual time=0.027..214.360 rows=499015 loops=1)
   ->  Hash (actual time=0.032..0.036 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on services (actual time=0.013..0.019 rows=4 loops=1)
 Planning Time: 0.719 ms
 Execution Time: 1117.126 ms
```

Si les `DISTINCT` sont courants et critiques dans votre application, notez que le nœud est parallélisable depuis PostgreSQL 15.

1.9.8 Index inutilisés

- Statistiques pas à jour/peu précises/oubliées
- Trop de lignes retournées
- Ordre des colonnes de l'index (B-tree)
- Index trop gros
- Prédicat avec transformation

```
WHERE col1 + 2 > 5  →  WHERE col1 > 5 - 2
```

- Opérateur non supporté par l'index

```
WHERE col1 <> 'valeur';
```

- Paramètres
  - `random_page_cost`
  - `effective_cache_size`

Il est relativement fréquent de créer soigneusement un index, et que PostgreSQL ne daigne pas l'utiliser. Il peut y avoir plusieurs raisons à cela.

Problème de statistiques :

Les statistiques de la table peuvent être périmées ou imprécises, pour les causes vues plus haut.

Un index fonctionnel possède ses propres statistiques : il faut donc penser à lancer `ANALYZE` après sa création. De même après un `CREATE STATISTICS`.

Nombre de lignes trouvées dans l'index :

Il faut se rappeler que PostgreSQL aura tendance à ne pas utiliser un index s'il doit chercher trop de lignes (ou plutôt de blocs), dans l'index comme dans la table ensuite. Il sera par contre tenté si cet index permet d'éviter des tris ou s'il est couvrant, et pas trop gros. La dispersion des lignes rencontrées dans la table est un facteur également pris en compte.

Colonnes d'un index B-tree :

Un index B-tree multicolonnes est inutilisable, en tout cas beaucoup moins performant, si les premiers champs ne sont pas fournis. L'ordre des colonnes a son importance.

Taille d'un index :

PostgreSQL tient compte de la taille des index. Un petit index peut être préféré à un index multicolonne auquel on a ajouté trop de champs pour qu'il soit couvrant.

Problèmes de prédicats :

Dans d'autres cas, les prédicats d'une requête ne permettent pas à l'optimiseur de choisir un index pour répondre à une requête. C'est le cas lorsque le prédicat inclut une transformation de la valeur d'une colonne. L'exemple suivant est assez naïf, mais assez représentatif et démontre bien le problème :

```
SELECT * FROM employes WHERE date_embauche + interval '1 month' = '2006-01-01';
```

Avec une telle construction, l'optimiseur ne saura pas tirer partie d'un quelconque index, à moins d'avoir créé un index fonctionnel sur `date_embauche + interval '1 month'`, mais cet index est largement contre-productif par rapport à une réécriture de la requête.

Ce genre de problème se rencontre plus souvent avec des prédicats sur des dates :

```
SELECT * FROM employes WHERE date_trunc('month', date_embauche) = 12;
```

ou encore plus fréquemment rencontré :

```
SELECT * FROM employes WHERE extract('year' from date_embauche) = 2006;
SELECT * FROM employes WHERE upper(prenom) = 'GASTON';
```

Opérateurs non-supportés :

Les index B-tree supportent la plupart des opérateurs généraux sur les variables scalaires (entiers, chaînes, dates, mais pas les types composés comme les géométries, hstore…), mais pas la différence (`<>` ou `!=`). Par nature, il n'est pas possible d'utiliser un index pour déterminer *toutes les valeurs sauf une*. Mais ce type de construction est parfois utilisé pour exclure les valeurs les plus fréquentes d'une colonne. Dans ce cas, il est possible d'utiliser un index partiel qui, en plus, sera très petit car il n'indexera qu'une faible quantité de données par rapport à la totalité de la table :

```
EXPLAIN SELECT * FROM employes_big WHERE num_service<>4;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..8264.74 rows=17 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=7 width=41)
         Filter: (num_service <> 4)
```

La création d'un index partiel permet d'en tirer partie :

```
CREATE INDEX ON employes_big(num_service) WHERE num_service<>4;

EXPLAIN SELECT * FROM employes_big WHERE num_service<>4;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using employes_big_num_service_idx1 on employes_big
   (cost=0.14..12.35 rows=17 width=40)
```

Paramétrage de PostgreSQL

Plusieurs paramètres de PostgreSQL influencent l'optimiseur sur l'utilisation ou non d'un index :

- `random_page_cost` : indique à PostgreSQL la vitesse d'un accès aléatoire par rapport à un accès séquentiel (`seq_page_cost`) ;

- `effective_cache_size` : indique à PostgreSQL une estimation de la taille du cache disque du système.

Le paramètre `random_page_cost` a une grande influence sur l'utilisation des index en général. Il indique à PostgreSQL le coût d'un accès disque aléatoire. Il est à comparer au paramètre `seq_page_cost` qui indique à PostgreSQL le coût d'un accès disque séquentiel. Ces coûts d'accès sont purement arbitraires et n'ont aucune réalité physique.

Dans sa configuration par défaut, PostgreSQL estime qu'un accès aléatoire est 4 fois plus coûteux qu'un accès séquentiel. Les accès aux index étant par nature aléatoires alors que les parcours de table sont par nature séquentiels, modifier ce paramètre revient à favoriser l'un par rapport à l'autre. Cette valeur est bonne dans la plupart des cas. Mais si le serveur de bases de données dispose d'un système disque rapide, c'est-à-dire une bonne carte RAID et plusieurs disques SAS rapides en RAID10, ou du SSD, il est possible de baisser ce paramètre à 3 voire 2 ou 1.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[10]

1.9.9 Écriture du SQL

- `NOT IN` avec une sous-requête
  - remplacer par `NOT EXISTS`
- `UNION` entraîne un tri systématique
  - préférer `UNION ALL`
- Sous-requête dans le `SELECT`
  - utiliser `LATERAL`

La façon dont une requête SQL est écrite peut aussi avoir un effet négatif sur les performances. Il n'est pas possible d'écrire tous les cas possibles, mais certaines formes d'écritures reviennent souvent.

La clause `NOT IN` n'est pas performante lorsqu'elle est utilisée avec une sous-requête. L'optimiseur ne parvient pas à exécuter ce type de requête efficacement.

```
SELECT *
  FROM services
 WHERE num_service NOT IN (SELECT num_service FROM employes_big);
```

Il est nécessaire de la réécrire avec la clause `NOT EXISTS`, par exemple :

[10] https://www.depesz.com/index.php/2010/09/09/why-is-my-index-not-being-used/

```
SELECT *
  FROM services s
 WHERE NOT EXISTS (SELECT 1
                   FROM employes_big e
                   WHERE s.num_service = e.num_service);
```

1.9.10 Absence de hints

- Certains regrettent l'absence de *hints*
- C'est la politique du projet :
  - vouloir ne signifie pas avoir besoin
  - PostgreSQL est un projet libre qui a le luxe de se défaire de la pression du marché
  - cela permet d'être plus facilement et rapidement mis au courant des problèmes de l'optimiseur
- Ne pensez pas être plus intelligent que le planificateur
- Mais il ne peut faire qu'avec ce qu'il a

L'absence de la notion de *hints*, qui permettent au DBA de forcer l'optimiseur à choisir des plans d'exécution jugés pourtant trop coûteux, est voulue. Elle a même été intégrée dans la liste des fonctionnalités dont la communauté ne voulait pas (« *Features We Do Not Want*[11] »).

L'absence des *hints* est très bien expliquée dans un billet de Josh Berkus, ancien membre de la Core Team de PostgreSQL[12] :

Le fait que certains DBA demandent cette fonctionnalité ne veut pas dire qu'ils ont réellement besoin de cette fonctionnalité. Parfois ce sont de mauvaises habitudes d'une époque révolue, où les optimiseurs étaient parfaitement stupides. Ajoutons à cela que les SGBD courants étant des projets commerciaux, ils sont forcément plus poussés à accéder aux demandes des clients, même si ces demandes ne se justifient pas, ou sont le résultat de pressions de pur court terme. Le fait que PostgreSQL soit un projet libre permet justement aux développeurs du projet de choisir les fonctionnalités implémentées suivant leurs idées, et non pas la pression du marché.

Selon le wiki sur le sujet[13], l'avis de la communauté PostgreSQL est que les *hints*, du moins tels qu'ils sont implémentés ailleurs, mènent à une plus grande complexité du code applicatif, donc à des problèmes de maintenabilité, interfèrent avec les mises à jour, risquent d'être contre-productifs au fur et à mesure que vos tables grossissent, et sont généralement inutiles. Sur le long terme, il vaut mieux rapporter un problème rencontré avec l'optimiseur pour qu'il soit définitivement corrigé. L'absence de *hints* permet d'être plus facilement et rapidement mis au courant des problèmes de l'optimiseur.

[11] https://wiki.postgresql.org/wiki/Todo#Features_We_Do_Not_Want
[12] https://it.toolbox.com/blogs/josh-berkus/why-postgresql-doesnt-have-query-hints-020411
[13] https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion

Sur le long terme, cela est meilleur pour le projet comme pour les utilisateurs. Cela a notamment mené à améliorer l'optimiseur et le recueil des statistiques.

L'accumulation de *hints* dans un système a tendance à poser problème lors de l'évolution des besoins, de la volumétrie ou après des mises à jour. Si le plan d'exécution généré n'est pas optimal, il est préférable de chercher à comprendre d'où vient l'erreur. Il est rare que l'optimiseur se trompe : en général c'est lui qui a raison. Mais il ne peut faire qu'avec les statistiques à sa disposition, le modèle qu'il voit, les index que vous avez créés. Nous avons vu dans ce module quelles pouvaient être les causes entraînant des erreurs de plan :

- mauvaise écriture de requête ;
- modèle de données pas optimal ;
- manque d'index adéquats (et PostgreSQL en possède une grande variété) ;
- statistiques pas à jour ;
- statistiques pas assez fines ;
- colonnes corrélées ;
- paramétrage de la mémoire ;
- paramétrage de la profondeur de recherche de l'optimiseur ;
- …

Ajoutons qu'il existe des outils comme PoWA[14] pour vous aider à optimiser des requêtes.

[14] https://powa.readthedocs.io/en/latest/

1.10 OUTILS D'OPTIMISATION

- auto_explain
- plantuner
- HypoPG

1.10.1 auto_explain

- Tracer les plans des requêtes lentes automatiquement
- Contrib officielle
- Mise en place globale (traces) :
  - globale :

```
shared_preload_libraries='auto_explain'   -- redémarrage !
ALTER DATABASE erp SET auto_explain.log_min_duration = '3s' ;
```

  - session :

```
LOAD 'auto_explain' ;
SET auto_explain.log_analyze TO true;
```

L'outil `auto_explain` est habituellement activé quand on a le sentiment qu'une requête devient subitement lente à certains moments, et qu'on suspecte que son plan diffère entre deux exécutions. Elle permet de tracer dans les journaux applicatifs, voire dans la console, le plan de la requête dès qu'elle dépasse une durée configurée.

C'est une « contrib » officielle de PostgreSQL (et non une extension). Tracer systématiquement le plan d'exécution d'une requête souvent répétée prend de la place, et est assez coûteux. C'est donc un outil à utiliser parcimonieusement. En général on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre `auto_explain.log_min_duration`. Par défaut, ce paramètre vaut -1 pour ne tracer aucun plan.

Comme dans un `EXPLAIN` classique, on peut activer les options (par exemple `ANALYZE` ou `TIMING` avec, respectivement, un `SET auto_explain.log_analyze TO true;` ou un `SET auto_explain.log_timing TO true;`) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'`EXPLAIN`, notamment : `auto_explain.log_buffers`, `auto_explain.log_settings`.

Quant à `auto_explain.sample_rate`, il permet de ne tracer qu'un échantillon des requêtes (voir la documentation[15]).

Pour utiliser `auto_explain` globalement, il faut charger la bibliothèque au démarrage dans le fichier `postgresql.conf` via le paramètre `shared_preload_libraries`.

```
shared_preload_libraries='auto_explain'
```

Après un redémarrage de l'instance, il est possible de configurer les paramètres de capture des plans d'exécution par base de données. Dans l'exemple ci-dessous, l'ensemble des requêtes sont tracées sur la base de données `bench`, qui est utilisée par `pgbench`.

```
ALTER DATABASE bench SET auto_explain.log_min_duration = '0';
ALTER DATABASE bench SET auto_explain.log_analyze = true;
```

Attention, l'activation des traces complètes sur une base de données avec un fort volume de requêtes peut être très coûteux.

Un benchmark `pgbench` est lancé sur la base de données `bench` avec 1 client qui exécute 1 transaction par seconde pendant 20 secondes :

```
pgbench -c1 -R1 -T20 bench
```

Les plans d'exécution de l'ensemble les requêtes exécutées par `pgbench` sont alors tracés dans les traces de l'instance.

```
2021-07-01 13:12:55.790 CEST [1705] LOG:  duration: 0.041 ms  plan:
    Query Text: SELECT abalance FROM pgbench_accounts WHERE aid = 416925;
    Index Scan using pgbench_accounts_pkey on pgbench_accounts
        (cost=0.42..8.44 rows=1 width=4) (actual time=0.030..0.032 rows=1 loops=1)
        Index Cond: (aid = 416925)
2021-07-01 13:12:55.791 CEST [1705] LOG:  duration: 0.123 ms  plan:
    Query Text: UPDATE pgbench_tellers SET tbalance = tbalance + -3201 WHERE tid = 19;
    Update on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                               (actual time=0.120..0.121 rows=0 loops=1)
        ->  Seq Scan on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                                         (actual time=0.040..0.058 rows=1 loops=1)
            Filter: (tid = 19)
            Rows Removed by Filter: 99
2021-07-01 13:12:55.797 CEST [1705] LOG:  duration: 0.116 ms  plan:
    Query Text: UPDATE pgbench_branches SET bbalance = bbalance + -3201 WHERE bid = 5;
    Update on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.112..0.114 rows=0 loops=1)
        ->  Seq Scan on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.036..0.038 rows=1 loops=1)
```

[15] https://docs.postgresql.fr/current/auto-explain.html

```
            Filter: (bid = 5)
            Rows Removed by Filter: 9
[...]
```

Pour utiliser `auto_explain` uniquement dans la session en cours, il faut penser à descendre au niveau de message `LOG` (défaut de `auto_explain`). On procède ainsi :

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0;
SET auto_explain.log_analyze = true;
SET client_min_messages to log;
SELECT count(*)
   FROM pg_class, pg_index
   WHERE oid = indrelid AND indisunique;

LOG:  duration: 1.273 ms  plan:
Query Text: SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
Aggregate  (cost=38.50..38.51 rows=1 width=8)
    (actual time=1.247..1.248 rows=1 loops=1)
  ->  Hash Join  (cost=29.05..38.00 rows=201 width=0)
          (actual time=0.847..1.188 rows=198 loops=1)
        Hash Cond: (pg_index.indrelid = pg_class.oid)
        ->  Seq Scan on pg_index  (cost=0.00..8.42 rows=201 width=4)
                      (actual time=0.028..0.188 rows=198 loops=1)
              Filter: indisunique
              Rows Removed by Filter: 44
        ->  Hash  (cost=21.80..21.80 rows=580 width=4)
            (actual time=0.726..0.727 rows=579 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 29kB
              ->  Seq Scan on pg_class  (cost=0.00..21.80 rows=580 width=4)
                            (actual time=0.016..0.373 rows=579 loops=1)
 count
-------
   198
```

`auto_explain` est aussi un moyen de suivre les plans au sein de fonctions. Par défaut, un plan n'indique les compteurs de blocs *hit*, *read*, *temp*… que de l'appel global à la fonction.

Une fonction simple en PL/pgSQL est définie pour récupérer le solde le plus élevé dans la table `pgbench_accounts` :

```
CREATE OR REPLACE function f_max_balance() RETURNS int AS $$
    DECLARE
        acct_balance int;
    BEGIN
        SELECT max(abalance)
        INTO acct_balance
        FROM pgbench_accounts;
        RETURN acct_balance;
    END;
$$ LANGUAGE plpgsql ;
```

Un simple `EXPLAIN ANALYZE` de l'appel de la fonction ne permet pas d'obtenir le plan de la requête `SELECT max(abalance) FROM pgbench_accounts` contenue dans la fonction :

```
EXPLAIN (ANALYZE,VERBOSE) SELECT f_max_balance();

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=4) (actual time=49.214..49.216 rows=1 loops=1)
   Output: f_max_balance()
 Planning Time: 0.149 ms
 Execution Time: 49.326 ms
```

Par défaut, `auto_explain` ne va pas capturer plus d'information que la commande `EXPLAIN ANALYZE`. Le fichier log de l'instance capture le même plan lorsque la fonction est exécutée.

```
2021-07-01 15:39:05.967 CEST [2768] LOG:  duration: 42.937 ms  plan:
    Query Text: select f_max_balance();
    Result  (cost=0.00..0.26 rows=1 width=4)
            (actual time=42.927..42.928 rows=1 loops=1)
```

Il est cependant possible d'activer le paramètre `log_nested_statements` avant l'appel de la fonction, de préférence uniquement dans la ou les sessions concernées :

```
\c bench
SET auto_explain.log_nested_statements = true;
SELECT f_max_balance();
```

Le plan d'exécution de la requête SQL est alors visible dans les traces de l'instance :

```
2021-07-01 14:58:40.189 CEST [2202] LOG:  duration: 58.938 ms  plan:
Query Text: select max(abalance)
            from pgbench_accounts
Finalize Aggregate
(cost=22632.85..22632.86 rows=1 width=4)
(actual time=58.252..58.935 rows=1 loops=1)
    ->  Gather
        (cost=22632.64..22632.85 rows=2 width=4)
        (actual time=57.856..58.928 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate
                (cost=21632.64..21632.65 rows=1 width=4)
                (actual time=51.846..51.847 rows=1 loops=3)
            ->  Parallel Seq Scan on pgbench_accounts
                    (cost=0.00..20589.51 rows=417251 width=4)
                    (actual time=0.014..29.379 rows=333333 loops=3)
```

pgBadger est capable de lire les plans tracés par `auto_explain`, de les intégrer à son rapport et d'inclure un lien vers depesz.com[16] pour une version plus lisible.

[16] https://explain.depesz.com/

1.10.2 Extension plantuner

- Pour :
  - interdire certains index
  - forcer à zéro les statistiques d'une table vide
- Intéressant en développement pour tester les plans
  - pas en production !

Cette extension est disponible à cette adresse[17] (le miroir GitHub ne semble pas maintenu). Oleg Bartunov, l'un de ses auteurs, a publié en 2018 un article intéressant[18] sur son utilisation.

Il faudra récupérer le source et le compiler. La configuration est basée sur trois paramètres :

- `plantuner.enable_index` pour préciser les index à activer ;
- `plantuner.disable_index` pour préciser les index à désactiver ;
- `plantuner.fix_empty_table` pour forcer à zéro les statistiques des tables de 0 bloc.

Ils sont configurables à chaud, comme le montre l'exemple suivant :

```
LOAD 'plantuner';
EXPLAIN (COSTS OFF)
  SELECT * FROM employes_big WHERE date_embauche='1000-01-01';

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
   Index Cond: (date_embauche = '1000-01-01'::date)

SET plantuner.disable_index='employes_big_date_embauche_idx';

EXPLAIN (COSTS OFF)
  SELECT * FROM employes_big WHERE date_embauche='1000-01-01';

                       QUERY PLAN
-------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
         Filter: (date_embauche = '1000-01-01'::date)
```

Un des intérêts de cette extension est de pouvoir interdire l'utilisation d'un index, afin de pouvoir ensuite le supprimer de manière transparente, c'est-à-dire sans bloquer aucune requête applicative.

[17] http://www.sai.msu.su/~megera/wiki/plantuner
[18] https://obartunov.livejournal.com/197604.html

Cependant, généralement, cette extension a sa place sur un serveur de développement pour bien comprendre les choix de planification, pas sur un serveur de production. En tout cas, pas dans le but de tromper le planificateur.

Comme avec toute extension en C, un bug est susceptible de provoquer un plantage complet du serveur.

1.10.3 Extension pg_plan_hint

- Pour :
  - forcer l'utilisation d'un nœud entre deux tables
  - imposer une valeur de paramètre
  - appliquer automatiquement ces *hints* à des requêtes

Cette extension existe depuis longtemps. Elle doit être compilée et installée depuis le dépôt Github[19].

La documentation[20] en anglais peut être complétée par la version japonaise[21] plus à jour, ou cet article[22].

Comme avec toute extension en C, un bug est susceptible de provoquer un plantage complet du serveur !

[19] https://github.com/ossc-db/pg_hint_plan
[20] http://pghintplan.osdn.jp/pg_hint_plan.html
[21] http://pghintplan.osdn.jp/pg_hint_plan-ja.html
[22] https://docs.yugabyte.com/latest/explore/query-1-performance/pg-hint-plan/

1.10.4 Extension HypoPG

- Extension PostgreSQL
- Création d'index hypothétiques pour tester leur intérêt
  - avant de les créer pour de vrai
- Limitations : surtout B-Tree, statistiques

Cette extension est disponible sur GitHub[23] et dans les paquets du PGDG. Il existe trois fonctions principales et une vue :

- `hypopg_create_index()` pour créer un index hypothétique ;
- `hypopg_drop_index()` pour supprimer un index hypothétique particulier ou `hypopg_reset()` pour tous les supprimer ;
- `hypopg_list_indexes` pour les lister.

Un index hypothétique n'existe que dans la session, ni en mémoire ni sur le disque, mais le planificateur le prendra en compte dans un `EXPLAIN` simple (évidemment pas un `EXPLAIN ANALYZE`). En quittant la session, tous les index hypothétiques restants et créés sur cette session sont supprimés.

L'exemple suivant est basé sur la base dont le script peut être téléchargé sur https://dali.bo/tp_employes_services.

```
CREATE EXTENSION hypopg;

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
         Filter: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_create_index('CREATE INDEX ON employes_big(prenom)');

 indexrelid |            indexname
------------+---------------------------------
      24591 | <24591>btree_employes_big_prenom

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using <24591>btree_employes_big_prenom on employes_big
                                (cost=0.05..4.07 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
```

[23] https://github.com/HypoPG/hypopg

```
SELECT * FROM hypopg_list_indexes;

 indexrelid |            indexname             | nspname |   relname    | amname
------------+----------------------------------+---------+--------------+--------
      24591 | <24591>btree_employes_big_prenom | public  | employes_big | btree

SELECT * FROM hypopg_reset();

 hypopg_reset
--------------

(1 row)

CREATE INDEX ON employes_big(prenom);

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                       QUERY PLAN
---------------------------------------------------------
 Index Scan using employes_big_prenom_idx on employes_big
                      (cost=0.42..4.44 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
```

Le cas idéal d'utilisation est l'index B-Tree sur une colonne. Un index fonctionnel est possible, mais, faute de statistiques disponibles avant la création réelle de l'index, les estimations peuvent être fausses. Les autres types d'index sont moins bien ou non supportées.

1.11 CONCLUSION

- Planificateur très avancé
- Ne pensez pas être plus intelligent que lui
- Il faut bien comprendre son fonctionnement

1.11.1 Questions

N'hésitez pas, c'est le moment !

1.12 QUIZ

https://dali.bo/j2_quiz

1.13 TRAVAUX PRATIQUES

1.13.1 Préambule

Tous les TP se basent 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 ;
```

- Afin d'éviter tout effet dû au cache, autant du plan que des pages de données, nous utilisons parfois une sous-requête avec un résultat non déterministe (fonction `random()`).

- 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.

Ce TP utilise notamment la base **cave**. Son schéma est le suivant :

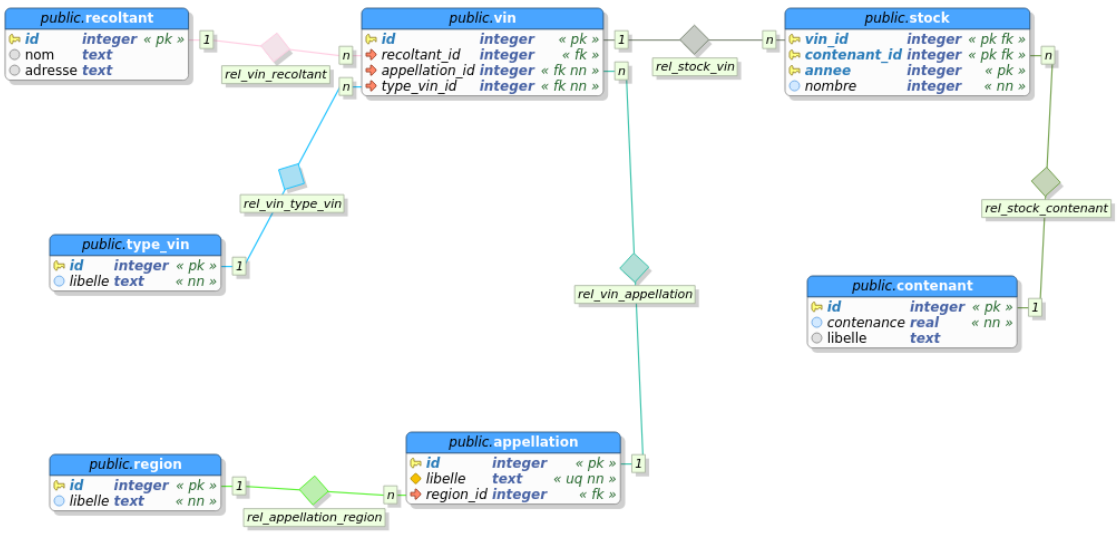


Figure 1/ .2: Schéma de la base cave

La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

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 ;

1.13.2 Optimisation d'une requête (partie 1)

But : Optimisation de requête

La requête suivante vise à récupérer un état des stocks pour une année prise au hasard :

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

EXPLAIN (ANALYZE, COSTS OFF)
SELECT
      m.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
FROM
     contenant c
     JOIN stock s
       ON s.contenant_id = c.id
     JOIN (SELECT round(random()*50)+1950 AS annee) m
       ON s.annee =  m.annee
     JOIN vin v
       ON s.vin_id = v.id
     LEFT JOIN appellation a
     ON v.appellation_id = a.id
GROUP BY m.annee||' - '||a.libelle;
```

- Exécuter la requête telle quelle et noter le plan et le temps d'exécution.

- Créer un index sur la colonne `stock.annee`.
- Exécuter la requête juste après la création de l'index.

- Rafraîchir les statistiques sur `stock`.
- Exécuter à nouveau la requête.

- Interdire à PostgreSQL les parcours de table avec `enable_seqscan` dans la session.
- Exécuter à nouveau la requête.

- Réautoriser les *Seq Scan*.

- Relancer la première requête ; chercher s'il y a un écart entre les nombres de lignes attendues et réellement ramenées.
- Quel est l'étape problématique ?

- Tenter de réécrire la requête pour l'optimiser en déplaçant la sélection de l'année dans la clause `WHERE`.
- Quel est le nouveau plan ?
- Les estimations sont-elles meilleures ?
- Le temps d'exécution est-il meilleur ?

1.13.3 Optimisation d'une requête (partie 2)

But : Optimisation de requête

L'exercice précédent nous a amené à cette requête :

```
EXPLAIN ANALYZE
SELECT
      s.annee||' - '||a.libelle AS millesime_region,
      sum(s.nombre) AS contenants,
      sum(s.nombre*c.contenance) AS litres
FROM
     contenant c
     JOIN stock s
       ON s.contenant_id = c.id
     JOIN vin v
       ON s.vin_id = v.id
     LEFT join appellation a
       ON v.appellation_id = a.id
WHERE s.annee = (SELECT round(random()*50)+1950 AS annee)
GROUP BY s.annee||' - '||a.libelle;
```

Cette écriture n'est pas optimale.

- Vérifier la pertinence de la dernière jointure sur `appellation`.
- Modifier la requête. Y a-t-il un impact sur le plan ?

- Tester avec une année précise (par exemple 1990).
- L'index sur `stock.annee` est-il utilisé ?
- Quelle est la différence avec le filtrage sur le résultat de la sous-requête ?
- Comment adapter la requête pour utiliser l'index ?

1.13.4 Requête avec beaucoup de tables

But : Optimiser une requête avec beaucoup de tables

- Importer la base **magasin** si elle n'est pas déjà chargée.

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

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

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

- Pour calculer le chiffre d'affaires gagné grâce au contact nommé Brahem Beatty via le transporteur « Royal Air Drone », tester cette requête et afficher son plan :

```
SET search_path TO magasin,facturation ;

SET max_parallel_workers_per_gather TO 0;      -- paramétrage pour simplifier les
 ↪ plans
SET jit TO off ;                               --

SELECT SUM (reglements.montant) AS somme_reglements
FROM       factures
INNER JOIN reglements USING (numero_facture)
INNER JOIN commandes USING (numero_commande)
INNER JOIN clients cl USING (client_id)
INNER JOIN types_clients USING (type_client)
INNER JOIN lignes_commandes lc USING (numero_commande)
INNER JOIN lots l ON (l.numero_lot = lc.numero_lot_expedition)
INNER JOIN transporteurs USING (transporteur_id)
INNER JOIN contacts ct ON (ct.contact_id = cl.contact_id)
WHERE      transporteurs.nom = 'Royal Air Drone'
AND        login = 'Beatty_Brahem' ;
```

- Comment améliorer le temps d'exécution SANS modifier la requête ni ajouter d'index ? (Il est évident et connu que le modèle de données est insuffisamment indexé, mais ce n'est pas le problème.)

- À l'inverse, sans modifier de paramètre, comment modifier la requête pour qu'elle s'exécute plus rapidement ?

1.13.5 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.

1.14 TRAVAUX PRATIQUES (SOLUTIONS)

1.14.1 Préambule

Tous les TP se basent 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 ;
```

- Afin d'éviter tout effet dû au cache, autant du plan que des pages de données, nous utilisons parfois une sous-requête avec un résultat non déterministe (fonction `random()`).

- 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.

Ce TP utilise notamment la base **cave**. Son schéma est le suivant :

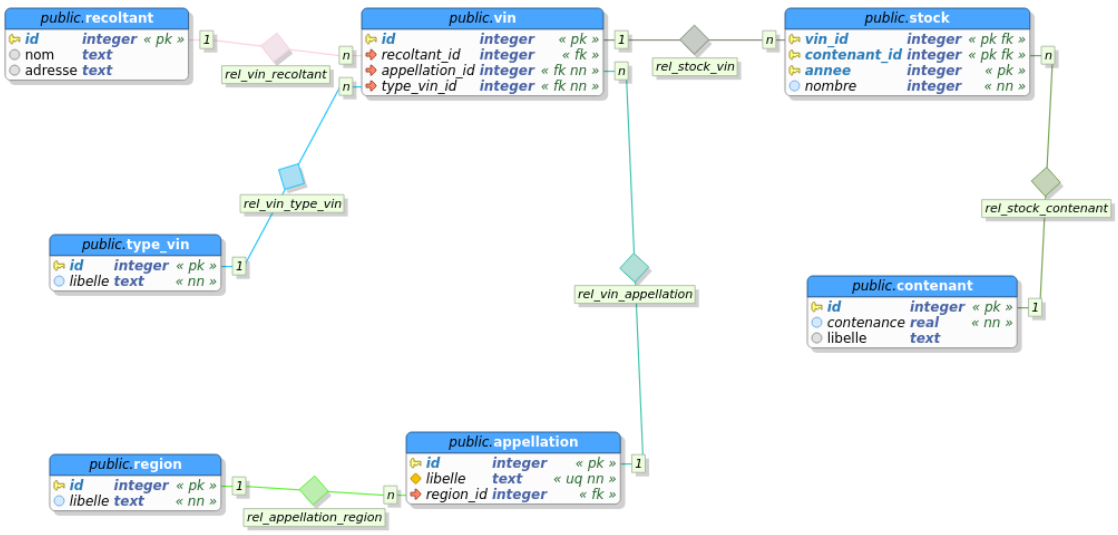


Figure 1/ .3: Schéma de la base cave

La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

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 Optimisation d'une requête (partie 1)

- Exécuter la requête telle quelle et noter le plan et le temps d'exécution.

L'exécution de la requête donne le plan suivant. Le temps comme le plan peuvent varier en fonction de la version exacte de PostgreSQL, de la machine utilisée, de son activité :

```
                                QUERY PLAN
--------------------------------------------------------------------------------
HashAggregate (actual time=199.630..199.684 rows=319 loops=1)
 Group Key: (((((round((random() * '50'::double precision))
                 + '1950'::double precision)))::text || ' - '::text)
                 || a.libelle)
 ->  Hash Left Join (actual time=61.631..195.614 rows=16892 loops=1)
       Hash Cond: (v.appellation_id = a.id)
       ->  Hash Join (actual time=61.531..190.045 rows=16892 loops=1)
             Hash Cond: (s.contenant_id = c.id)
             ->  Hash Join (actual time=61.482..186.976 rows=16892 loops=1)
                   Hash Cond: (s.vin_id = v.id)
                   ->  Hash Join (actual time=60.049..182.135 rows=16892 loops=1)
                         Hash Cond: ((s.annee)::double precision
                                    = ((round((random() * '50'::double precision))
                                    + '1950'::double precision)))
                         ->  Seq Scan on stock s (… rows=860588 loops=1)
                         ->  Hash (actual time=0.010..0.011 rows=1 loops=1)
                               Buckets: 1024  Batches: 1  Memory Usage: 9kB
                               ->  Result (… rows=1 loops=1)
                   ->  Hash (actual time=1.420..1.421 rows=6062 loops=1)
                         Buckets: 8192  Batches: 1  Memory Usage: 301kB
                         ->  Seq Scan on vin v (… rows=6062 loops=1)
             ->  Hash (actual time=0.036..0.036 rows=3 loops=1)
                   Buckets: 1024  Batches: 1  Memory Usage: 9kB
                   ->  Seq Scan on contenant c (… rows=3 loops=1)
       ->  Hash (actual time=0.090..0.090 rows=319 loops=1)
             Buckets: 1024  Batches: 1  Memory Usage: 25kB
             ->  Seq Scan on appellation a (… rows=319 loops=1)
Planning Time: 2.673 ms
Execution Time: 199.871 ms
```

- Créer un index sur la colonne `stock.annee`.
- Exécuter la requête juste après la création de l'index.

Instinctivement on s'attend à ce qu'un index sur `stock.annee` soit utile, puisque l'on sélectionne uniquement là-dessus. Mais il n'y en a pas.

```
CREATE INDEX stock_annee on stock (annee) ;
```

Cependant, le plan ne change pas si l'on relance la requête ci-dessus.

La raison est simple : au moment de la construction du plan, la valeur de l'année est inconnue. L'index est donc inutilisable.

- Rafraîchir les statistiques sur `stock`.
- Exécuter à nouveau la requête.

Peut-être `ANALYZE` a-t-il été oublié ? Dans l'idéal, un `VACUUM ANALYZE` est même préférable pour favoriser les `Index Only Scan`.

```
VACUUM ANALYZE STOCK ;
```

Mais cela n'a pas d'influence sur le plan. En fait, le premier plan ci-dessus montre que les statistiques sont déjà correctement estimées.

- Interdire à PostgreSQL les parcours de table avec `enable_seqscan` dans la session.
- Exécuter à nouveau la requête.

```
SET enable_seqscan TO off;
```

Nous remarquons que le temps d'exécution explose :

```
EXPLAIN (ANALYZE, COSTS OFF) SELECT …
```

```
GroupAggregate (actual time=1279.990..1283.367 rows=319 loops=1)
  Group Key: ((((((round((random() * '50'::double precision))
                  + '1950'::double precision)))::text || ' - '::text)
                  || a.libelle))
  ->  Sort (actual time=1279.965..1280.895 rows=16854 loops=1)
        Sort Key: ((((((round((random() * '50'::double precision))
                       + '1950'::double precision)))::text || ' - '::text)
                       || a.libelle))
        Sort Method: quicksort  Memory: 2109kB
        ->  Hash Left Join (actual time=11.163..1258.628 rows=16854 loops=1)
              Hash Cond: (v.appellation_id = a.id)
              ->  Hash Join (actual time=10.911..1247.542 rows=16854 loops=1)
                    Hash Cond: (s.vin_id = v.id)
                    ->  Nested Loop (actual time=0.070..1230.297
                                     rows=16854 loops=1)
                          Join Filter: (s.contenant_id = c.id)
                          Rows Removed by Join Filter: 17139
                          ->  Hash Join (actual time=0.056..1220.730
```

```
                                    rows=16854 loops=1)
                                Hash Cond: ((s.annee)::double precision =
                                           ((round((random() *
                                           '50'::double precision))
                                           + '1950'::double precision)))
                                ->  Index Scan using stock_pkey on stock s
                                    (actual time=0.011..1098.671 rows=860588 loops=1)
                                ->  Hash (actual time=0.007..0.007 rows=1 loops=1)
                                      Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                      ->  Result (…rows=1 loops=1)
                           ->  Materialize (… rows=2 loops=16854)
                                 ->  Index Scan using contenant_pkey on contenant c
                                     (actual time=0.007..0.009 rows=3 loops=1)
                     ->  Hash (actual time=10.826..10.826 rows=6062 loops=1)
                           Buckets: 8192  Batches: 1  Memory Usage: 301kB
                           ->  Index Scan using vin_pkey on vin v
                               (actual time=0.010..8.436 rows=6062 loops=1)
               ->  Hash (actual time=0.233..0.233 rows=319 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 25kB
                     ->  Index Scan using appellation_pkey on appellation a
                         (actual time=0.015..0.128 rows=319 loops=1)
Planning Time: 1.337 ms
Execution Time: 1283.467 ms
```

Le plan renvoyé peut être analysé avec https://explain.dalibo.com.

> – Réautoriser les *Seq Scan*.

```
RESET enable_seqscan;
```

> – Relancer la première requête ; chercher s'il y a un écart entre les nombres de lignes attendues et réellement ramenées.
> – Quel est l'étape problématique ?

Avec `COSTS ON` (qui est activé par défaut), les estimations attendues sont affichées, où l'on peut comparer ligne par ligne aux nombres réellement ramenés.

```
EXPLAIN (ANALYZE, COSTS ON) SELECT …

 HashAggregate  (cost=17931.35..17937.73 rows=319 width=48)
                (actual time=195.338..195.388 rows=319 loops=1)
   Group Key: (…)
…
```

L'estimation du nombre de lignes renvoyé par la requête est parfaite. Est-ce le cas pour tous les nœuds en-dessous ?

D'abord on note que les lignes à regrouper étaient 4 fois plus nombreuses que prévues :

```
   ->  Hash Left Join         (cost=180.68..17877.56 rows=4303 width=40)
                         (actual time=136.012..191.468 rows=16834 loops=1)
         Hash Cond: (v.appellation_id = a.id)
```

Cela ne veut pas dire que les statistiques sur les tables `v` ou `a` sont fausses, car les nœuds précédents ont déjà opéré des jointures et filtrages. Si on tente de descendre au nœud le plus profond qui montre un problème d'estimation, on trouve ceci :

```
->  Hash Join
(cost=0.04..17603.89          rows=4303 width=20)
(actual time=134.406..177.861 rows=16834 loops=1)
   Hash Cond: ((s.annee)::double precision =
        ((round((random() * '50'::double precision))
        + '1950'::double precision)))
```

Il s'agit de la jointure *hash* entre `stock` et `annee` sur une sélection aléatoire de l'année. PostgreSQL s'attend à 4303 lignes, et en retrouve 16 834, 4 fois plus.

Il ne s'agit pas d'un problème dans l'estimation de `stock` même, car il s'attend correctement à y balayer 860 588 lignes (il s'agit bien du nombre de lignes vivantes total de la table qui vont alimenter la jointure avec `annee`) :

```
->  Seq Scan on stock s
    (cost=0.00..13257.88 rows=860588 width=16)
    (actual time=0.012..66.563 rows=860588 loops=1)
```

La seconde partie du *hash* (le `SELECT` sur `annee`) est constitué d'un *bucket* correctement estimé à 1 ligne depuis le résultat de la sous-requête :

```
->  Hash          (cost=0.03..0.03 rows=1 width=8)
        (actual time=0.053..0.053 rows=1 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 9kB
      ->  Result  (cost=0.00..0.02 rows=1 width=8)
          (actual time=0.049..0.050 rows=1 loops=1)
```

Il y a donc un problème dans l'estimation du **nombre de lignes** ramenées par la jointure sur l'année choisie au hasard.

- Tenter de réécrire la requête pour l'optimiser en déplaçant la sélection de l'année dans la clause `WHERE`.
- Quel est le nouveau plan ?
- Les estimations sont-elles meilleures ?
- Le temps d'exécution est-il meilleur ?

```
EXPLAIN ANALYZE
SELECT
      s.annee||' - '||a.libelle AS millesime_region,
      sum(s.nombre) AS contenants,
      sum(s.nombre*c.contenance) AS litres
FROM
     contenant c
     JOIN stock s
       ON s.contenant_id = c.id
     JOIN vin v
       ON s.vin_id = v.id
     LEFT join appellation a
       ON v.appellation_id = a.id
```

```
WHERE s.annee = (SELECT round(random()*50)+1950 AS annee)
GROUP BY s.annee||' - '||a.libelle;
```

Il y a une jointure en moins, ce qui est toujours appréciable. Nous pouvons faire cette réécriture parce que la requête `SELECT round(random()*50)+1950 AS annee` ne ramène qu'un seul enregistrement.

Le nouveau plan est :

```
 HashAggregate      (cost=17888.29..17974.35 rows=4303 width=48)
              (actual time=123.606..123.685 rows=319 loops=1)
   Group Key: (((s.annee)::text || ' - '::text) || a.libelle)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=8)
                 (… rows=1 loops=1)
   ->  Hash Left Join  (cost=180.64..17834.49 rows=4303 width=40)
                  (actual time=8.329..114.481 rows=17527 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=170.46..17769.84 rows=4303 width=16)
                    (actual time=7.847..101.390 rows=17527 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=169.40..17741.52 rows=4303 width=16)
                          (actual time=7.789..94.117 rows=17527 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Seq Scan on stock s
                                 (cost=0.00..17560.82 rows=4303 width=16)
                           (actual time=0.031..77.158 rows=17527 loops=1)
                       Filter: ((annee)::double precision = $0)
                       Rows Removed by Filter: 843061
                     ->  Hash         (cost=93.62..93.62 rows=6062 width=8)
                           (actual time=7.726..7.726 rows=6062 loops=1)
                       Buckets: 8192  Batches: 1  Memory Usage: 301kB
                       ->  Seq Scan on vin v
                                       (cost=0.00..93.62 rows=6062 width=8)
                             (actual time=0.016..3.563 rows=6062 loops=1)
               ->  Hash           (cost=1.03..1.03 rows=3 width=8)
                     (actual time=0.040..0.040 rows=3 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on contenant c
                                       (cost=0.00..1.03 rows=3 width=8)
                             (actual time=0.026..0.030 rows=3 loops=1)
         ->  Hash           (cost=6.19..6.19 rows=319 width=20)
                    (actual time=0.453..0.453 rows=319 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 25kB
               ->  Seq Scan on appellation a
                             (cost=0.00..6.19 rows=319 width=20)
                      (actual time=0.019..0.200 rows=319 loops=1)
 Planning Time: 2.227 ms
 Execution Time: 123.909 ms
```

Sur la machine testée, le temps d'exécution est réduit d'un tiers. Pourtant, le plan n'est que très peu différent, et les estimations ne sont pas meilleures (ce qui semble logique, PostgreSQL n'ayant pas plus d'informations sur la valeur exacte de l'année qui sera calculée).

La différence avec l'ancien plan est cette partie :

```
->  Seq Scan on stock s
                  (cost=0.00..17560.82 rows=4303 width=16)
              (actual time=0.031..77.158 rows=17527 loops=1)
      Filter: ((annee)::double precision = $0)
      Rows Removed by Filter: 843061
```

Le nouveau plan comprend le calcul de la varable `$0` (tout en haut) puis un parcours complet de `stock` et un filtrage au fur et à mesure des lignes sur `annee=$0`.

Il ne s'agit plus là d'une jointure par *hash*. Toute la construction d'une table de hachage sur la table `stock` est supprimée. PostgreSQL sait de manière absolue qu'il n'y aura qu'une seule valeur ramenée par sous-requête, grâce à `=`. Ce n'était pas évident pour lui car le résultat des fonctions forme un peu une « boîte noire ». Si on remplace le `=` par `IN`, on retombe sur le plan original.

Noter toutefois que la différence totale de coût au niveau de la requête est faible.

Que peut-on conclure de cet exercice ?

- l'optimiseur n'est pas tenu d'utiliser un index ;
- se croire plus malin que l'optimiseur est souvent contre-productif (`SET enable_seqscan TO off` n'a pas mené au résultat espéré) ;
- il vaut toujours mieux être explicite dans ce qu'on demande dans une requête ;
- il vaut mieux séparer jointure et filtrage.

Il reste un mystère qui sera couvert par un exercice suivant : pourquoi l'index sur `stock.annee` n'est-il pas utilisé ?

1.14.3 Optimisation d'une requête (partie 2)

- Vérifier la pertinence de la dernière jointure sur `appellation`.
- Modifier la requête. Y a-t-il un impact sur le plan ?

On peut se demander si la jointure externe (`LEFT JOIN`) est fondée :

```
LEFT JOIN appellation a ON v.appellation_id = a.id
```

Cela se traduit par « récupérer tous les tuples de la table `vin`, et pour chaque correspondance dans `appellation`, la récupérer, si elle existe ».

La description de la table `vin` est :

```
 \d vin
                                  Table « public.vin »
    Colonne     |  Type   | ... | NULL-able |           Par défaut
----------------+---------+-----+-----------+--------------------------------
 id             | integer |     | not null  | nextval('vin_id_seq'::regclass)
 recoltant_id   | integer |     |           |
 appellation_id | integer |     | not null  |
```

```
 type_vin_id    | integer |     | not null  |
Index :
    "vin_pkey" PRIMARY KEY, btree (id)
Contraintes de clés étrangères :
    "vin_appellation_id_fkey" FOREIGN KEY (appellation_id)
                              REFERENCES appellation(id) ON DELETE CASCADE
    "vin_recoltant_id_fkey" FOREIGN KEY (recoltant_id)
                            REFERENCES recoltant(id) ON DELETE CASCADE
    "vin_type_vin_id_fkey" FOREIGN KEY (type_vin_id)
                           REFERENCES type_vin(id) ON DELETE CASCADE
Référencé par :
    TABLE "stock" CONSTRAINT "stock_vin_id_fkey"
    FOREIGN KEY (vin_id) REFERENCES vin(id) ON DELETE CASCADE
```

`appellation_id` est `NOT NULL` : il y a forcément une valeur, qui est forcément dans `appelation`. De plus, la contrainte `vin_appellation_id_fkey` signifie qu'on a la certitude que pour chaque `vin.appellation.id`, il existe une ligne correspondante dans `appellation`.

À titre de vérification, deux `COUNT(*)` du résultat, une fois en `INNER JOIN` et une fois en `LEFT JOIN` montrent un résultat identique :

```
SELECT COUNT(*)
FROM vin v
  INNER JOIN appellation a ON (v.appellation_id = a.id);

 count
-------
  6057

SELECT COUNT(*)
FROM vin v
  LEFT JOIN appellation a ON (v.appellation_id = a.id);

 count
-------
  6057
```

On peut donc réécrire la requête sans la jointure externe, qui n'est pas fausse mais est généralement bien moins efficace qu'une jointure interne :

```
EXPLAIN ANALYZE
SELECT
      s.annee||' - '||a.libelle AS millesime_region,
      sum(s.nombre) AS contenants,
      sum(s.nombre*c.contenance) AS litres
FROM
     contenant c
     JOIN stock s
       ON s.contenant_id = c.id
     JOIN vin v
       ON s.vin_id = v.id
     JOIN appellation a
       ON v.appellation_id = a.id
WHERE s.annee =  (SELECT round(random()*50)+1950 AS annee)
GROUP BY s.annee||' - '||a.libelle;
```

Quant au plan, il est identique au plan précédent. Cela n'est pas étonnant : il n'y a aucun filtrage sur `appellation` et c'est une petite table, donc intuitivement on peut se dire que PostgreSQL fera la jointure une fois les autres opérations effectuées, sur le minimum de lignes. D'autre part, PostgreSQL est depuis longtemps capable de transformer un `LEFT JOIN` inutile en `INNER JOIN` quand la contrainte est là.

Si on observe attentivement le plan, on constate qu'on a toujours le parcours séquentiel de la table `stock`, qui est notre plus grosse table. Pourquoi a-t-il lieu ?

- Tester avec une année précise (par exemple 1990).
- L'index sur `stock.annee` est-il utilisé ?
- Quelle est la différence avec le filtrage sur le résultat de la sous-requête ?
- Comment adapter la requête pour utiliser l'index ?

Si on fige l'année, on constate que l'index sur `stock.annee` est bien utilisé, avec un temps d'exécution bien plus réduit :

```
EXPLAIN (ANALYSE, COSTS OFF)
SELECT
      s.annee||' - '||a.libelle AS millesime_region,
      sum(s.nombre) AS contenants,
      sum(s.nombre*c.contenance) AS litres
FROM
     contenant c
     JOIN stock s
       ON s.contenant_id = c.id
     JOIN vin v
       ON s.vin_id = v.id
     JOIN appellation a
       ON v.appellation_id = a.id
WHERE s.annee = 1950
GROUP BY s.annee||' - '||a.libelle;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate (actual time=48.827..48.971 rows=319 loops=1)
   Group Key: (((s.annee)::text || ' - '::text) || a.libelle)
   ->  Hash Join (actual time=8.889..40.737 rows=17527 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join (actual time=8.398..29.828 rows=17527 loops=1)
               Hash Cond: (s.vin_id = v.id)
               ->  Hash Join (actual time=0.138..14.374 rows=17527 loops=1)
                     Hash Cond: (s.contenant_id = c.id)
                     ->  Index Scan using stock_annee_idx on stock s
                           (actual time=0.066..6.587 rows=17527 loops=1)
                           Index Cond: (annee = 1950)
                     ->  Hash (actual time=0.017..0.018 rows=3 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                           ->  Seq Scan on contenant c (… rows=3 loops=1)
               ->  Hash (actual time=8.228..8.228 rows=6062 loops=1)
                     Buckets: 8192  Batches: 1  Memory Usage: 301kB
                     ->  Seq Scan on vin v (…)
         ->  Hash (actual time=0.465..0.465 rows=319 loops=1)
```

```
               Buckets: 1024  Batches: 1  Memory Usage: 25kB
               ->  Seq Scan on appellation a (…)
 Planning Time: 2.144 ms
 Execution Time: 49.317 ms
```

La partie qui diffère de l'ancien plan est celle-ci :

```
                     ->  Index Scan using stock_annee_idx on stock s
                           (actual time=0.066..6.587 rows=17527 loops=1)
                           Index Cond: (annee = 1950)
```

Quand précédemment on avait un parcours et un filtrage :

```
                     ->  Seq Scan on stock s
                                 (cost=0.00..17560.82 rows=4303 width=16)
                              (actual time=0.031..77.158 rows=17527 loops=1)
                           Filter: ((annee)::double precision = $0)
                           Rows Removed by Filter: 843061
```

Le nombre de lignes estimées et obtenues sont pourtant les mêmes. Pourquoi PostgreSQL utilise-t-il l'index pour filtrer sur `1950` et par pour `$0` ? Le filtre en fait diffère, le premier est `(annee = 1950)` (compatible avec un index), l'autre est `((annee)::double precision = $0)`, qui contient une conversion de `int` en `double precision` ! Et dans ce cas, l'index est inutilisable (comme à chaque fois qu'il y a une opération sur la colonne indexée).

La conversion a lieu parce que la fonction `round()` retourne un nombre à virgule flottante. La somme d'un nombre à virgule flottante et d'un entier est évidemment un nombre à virgule flottante. Si on veut que la fonction `round()` retourne un entier, il faut forcer explicitement sa conversion, via `CAST(xxx as int)` ou `::int`.

Le phénomène peut s'observer sur la requête avec 1950 en comparant `annee = 1950 + 1.0` : l'index ne sera plus utilisé.

Réécrivons encore une fois cette requête en homogénéisant les types :

```
EXPLAIN ANALYZE
SELECT
      s.annee||' - '||a.libelle AS millesime_region,
      sum(s.nombre) AS contenants,
      sum(s.nombre*c.contenance) AS litres
FROM
     contenant c
     JOIN stock s
       ON s.contenant_id = c.id
     JOIN vin v
       ON s.vin_id = v.id
     JOIN appellation a
       ON v.appellation_id = a.id
WHERE s.annee =  (SELECT (round(random()*50))::int + 1950 AS annee)
GROUP BY s.annee||' - '||a.libelle;
```

Voici son plan :

```
HashAggregate (actual time=28.208..28.365 rows=319 loops=1)
  Group Key: (((s.annee)::text || ' - '::text) || a.libelle)
  InitPlan 1 (returns $0)
    ->  Result (actual time=0.003..0.003 rows=1 loops=1)
  ->  Hash Join (actual time=2.085..23.194 rows=16891 loops=1)
        Hash Cond: (v.appellation_id = a.id)
        ->  Hash Join (actual time=1.894..16.358 rows=16891 loops=1)
              Hash Cond: (s.vin_id = v.id)
              ->  Hash Join (actual time=0.091..9.865 rows=16891 loops=1)
                    Hash Cond: (s.contenant_id = c.id)
                    ->  Index Scan using stock_annee_idx on stock s
                        (actual time=0.068..4.774 rows=16891 loops=1)
                          Index Cond: (annee = $0)
                    ->  Hash (actual time=0.013..0.013 rows=3 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Seq Scan on contenant c (…)
              ->  Hash (actual time=1.792..1.792 rows=6062 loops=1)
                    Buckets: 8192  Batches: 1  Memory Usage: 301kB
                    ->  Seq Scan on vin v (…)
        ->  Hash (actual time=0.183..0.183 rows=319 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 25kB
              ->  Seq Scan on appellation a (…)
Planning Time: 0.574 ms
Execution Time: 28.516 ms
```

On constate qu'on utilise enfin l'index de `stock`. Le temps d'exécution est bien meilleur. Ce problème d'incohérence de type était la cause fondamentale du ralentissement de la requête.

Noter au passage que le critère suivant ne fonctionnera pas, non à cause du type, mais parce qu'il est faux :

```
WHERE s.annee =  (round(random()*50))::int + 1950)
```

En effet, la comparaison entre `annee` et la valeur aléatoire se ferait à *chaque* ligne séparément, avec un résultat complètement faux. Pour choisir *une* année au hasard, il faut donc encapsuler le calcul dans une sous-requête, dont le résultat ramènera une seule ligne de manière garantie.

1.14.4 Requête avec beaucoup de tables

> – Importer la base **magasin** si elle n'est pas déjà chargée.

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

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

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

> – Pour calculer le chiffre d'affaires gagné grâce au contact nommé Brahem Beatty via le transporteur « Royal Air Drone », tester cette requête et afficher son plan :

```
SET search_path TO magasin,facturation ;

SET max_parallel_workers_per_gather TO 0;      -- paramétrage pour simplifier les
↪  plans
SET jit TO off ;                               --

SELECT SUM (reglements.montant) AS somme_reglements
FROM       factures
INNER JOIN reglements USING (numero_facture)
INNER JOIN commandes USING (numero_commande)
INNER JOIN clients cl USING (client_id)
INNER JOIN types_clients USING (type_client)
INNER JOIN lignes_commandes lc USING (numero_commande)
INNER JOIN lots l ON (l.numero_lot = lc.numero_lot_expedition)
INNER JOIN transporteurs USING (transporteur_id)
INNER JOIN contacts ct ON (ct.contact_id = cl.contact_id)
WHERE      transporteurs.nom = 'Royal Air Drone'
AND        login = 'Beatty_Brahem' ;
```

Cette requête s'exécute très lentement. Son plan simplifié est le suivant (la version complète est sur https://explain.dalibo.com/plan/D0U) :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate (actual time=3050.969..3050.978 rows=1 loops=1)
   ->  Hash Join (actual time=2742.616..3050.966 rows=4 loops=1)
         Hash Cond: (cl.contact_id = ct.contact_id)
         ->  Hash Join (actual time=2192.741..2992.578 rows=422709 loops=1)
               Hash Cond: (factures.numero_commande = commandes.numero_commande)
               ->  Hash Join (actual time=375.112..914.517 rows=1055812 loops=1)
                     Hash Cond: ((reglements.numero_facture)::text =
↪  (factures.numero_facture)::text)
                     ->  Seq Scan on reglements (actual time=0.007..96.963
↪  rows=1055812 loops=1)
                     ->  Hash (actual time=371.347..371.348 rows=1000000 loops=1)
                           Buckets: 1048576  Batches: 1  Memory Usage: 62880kB
                           ->  Seq Scan on factures (actual time=0.018..113.699
↪  rows=1000000 loops=1)
               ->  Hash (actual time=1813.741..1813.746 rows=393841 loops=1)
                     Buckets: 1048576  Batches: 1  Memory Usage: 29731kB
                    ->  Hash Join (actual time=558.943..1731.833 rows=393841 loops=1)
                           Hash Cond: (cl.type_client = types_clients.type_client)
                           ->  Hash Join (actual time=558.912..1654.443 rows=393841
↪  loops=1)
                                 Hash Cond: (commandes.client_id = cl.client_id)
                                 ->  Hash Join (actual time=533.279..1522.611
↪  rows=393841 loops=1)
                                       Hash Cond: (lc.numero_commande =
↪  commandes.numero_commande)
                                       ->  Hash Join (actual time=190.050..1073.358
↪  rows=393841 loops=1)
                                             Hash Cond: (lc.numero_lot_expedition =
↪  l.numero_lot)
```

```
                                                ->  Seq Scan on lignes_commandes lc
↪  (actual time=0.024..330.462 rows=3141967 loops=1)
                                                ->  Hash (actual time=189.059..189.061
↪  rows=125889 loops=1)
                                                      Buckets: 262144  Batches: 1  Memory
↪  Usage: 6966kB
                                                        ->  Hash Join (actual
↪  time=0.032..163.622 rows=125889 loops=1)
                                                             Hash Cond: (l.transporteur_id
↪  = transporteurs.transporteur_id)
                                                            ->  Seq Scan on lots l (actual
↪  time=0.016..68.766 rows=1006704 loops=1)
                                                               ->  Hash (actual
↪  time=0.010..0.011 rows=1 loops=1)
                                                                  Buckets: 1024  Batches: 1
↪  Memory Usage: 9kB
                                                                     ->  Seq Scan on
↪  transporteurs (actual time=0.006..0.007 rows=1 loops=1)
                                                                           Filter:
↪  ((nom)::text = 'Royal Air Drone'::text)
                                                                           Rows Removed by
↪  Filter: 4
                                          ->  Hash (actual time=339.432..339.432
↪  rows=1000000 loops=1)
                                                Buckets: 1048576  Batches: 1  Memory
↪  Usage: 55067kB
                                                ->  Seq Scan on commandes (actual
↪  time=0.028..118.268 rows=1000000 loops=1)
                                    ->  Hash (actual time=25.156..25.156 rows=100000
↪  loops=1)
                                       Buckets: 131072  Batches: 1  Memory Usage: 6493kB
                                          ->  Seq Scan on clients cl (actual
↪  time=0.006..9.926 rows=100000 loops=1)
                              ->  Hash (actual time=0.018..0.018 rows=3 loops=1)
                                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                    ->  Seq Scan on types_clients (actual
↪  time=0.010..0.011 rows=3 loops=1)
        ->  Hash (actual time=29.722..29.723 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
             ->  Seq Scan on contacts ct (actual time=17.172..29.716 rows=1 loops=1)
                   Filter: ((login)::text = 'Beatty_Brahem'::text)
                   Rows Removed by Filter: 110004
Planning Time: 1.390 ms
Execution Time: 3059.442 m
```

Le plan se résume ainsi : un premier filtre se fait sur le transporteur demandé (1 ligne sur 4). Puis toutes les jointures s'enchaînent, de manière certes peu efficace : toutes les tables sont parcourues intégralement. Enfin, les 422 709 lignes obtenues sont jointes à la table `contacts`, laquelle a été filtrée sur la personne demandée (1 ligne sur 110 005).

Le critère sur `contact` est de loin le plus discriminant : on s'attend à ce qu'il soit le premier pris en compte. Le plan complet montre que les estimations de volumétrie sont pourtant correctes.

> – Comment améliorer le temps d'exécution SANS modifier la requête ni ajouter d'index ? (Il est évident et connu que le modèle de données est insuffisamment indexé, mais ce n'est pas le problème.)

Il y a 9 tables. Avec autant de tables, il faut se rappeler de l'existence du paramètre `join_collapse_limit`. Vérifions que la valeur est celle par défaut, et testons une autre valeur :

```
SHOW join_collapse_limit ;
```

```
 join_collapse_limit
---------------------
 8
```

```
SET join_collapse_limit TO 9 ;
```

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT SUM (reglements.montant) AS somme_reglements
FROM       factures
INNER JOIN reglements USING (numero_facture)
INNER JOIN commandes USING (numero_commande)
INNER JOIN clients cl USING (client_id)
INNER JOIN types_clients USING (type_client)
INNER JOIN lignes_commandes lc USING (numero_commande)
INNER JOIN lots l ON (l.numero_lot = lc.numero_lot_expedition)
INNER JOIN transporteurs USING (transporteur_id)
INNER JOIN contacts ct ON (ct.contact_id = cl.contact_id)
WHERE      transporteurs.nom = 'Royal Air Drone'
AND        login = 'Beatty_Brahem' ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Aggregate (actual time=533.593..533.601 rows=1 loops=1)
   ->  Hash Join (actual time=464.437..533.589 rows=4 loops=1)
         Hash Cond: ((reglements.numero_facture)::text =
↪  (factures.numero_facture)::text)
         ->  Seq Scan on reglements (actual time=0.011..83.493 rows=1055812 loops=1)
         ->  Hash (actual time=354.413..354.420 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Hash Join (actual time=326.786..354.414 rows=4 loops=1)
                     Hash Cond: (factures.numero_commande = commandes.numero_commande)
                     ->  Seq Scan on factures (actual time=0.012..78.213 rows=1000000
↪  loops=1)
                     ->  Hash (actual time=197.837..197.843 rows=4 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                           ->  Hash Join (actual time=118.525..197.838 rows=4 loops=1)
                                 Hash Cond: (l.transporteur_id =
↪  transporteurs.transporteur_id)
                                 ->  Nested Loop (actual time=49.407..197.816 rows=35
↪  loops=1)
                                       ->  Nested Loop (actual time=49.400..197.701
↪  rows=35 loops=1)
                                             ->  Hash Join (actual time=49.377..197.463
↪  rows=10 loops=1)
                                                   Hash Cond: (commandes.client_id =
↪  cl.client_id)
```

```
                                                    ->  Seq Scan on commandes (actual
↪  time=0.003..88.021 rows=1000000 loops=1)
                                                  ->  Hash (actual time=30.975..30.978
↪  rows=1 loops=1)
                                                        Buckets: 1024  Batches: 1
↪  Memory Usage: 9kB
                                                        ->  Nested Loop (actual
↪  time=20.840..30.976 rows=1 loops=1)
                                                              ->  Hash Join (actual
↪  time=20.823..30.957 rows=1 loops=1)
                                                                    Hash Cond:
↪  (cl.contact_id = ct.contact_id)
                                                                    ->  Seq Scan on
↪  clients cl (actual time=0.003..6.206 rows=100000 loops=1)
                                                                  ->  Hash (actual
↪  time=16.168..16.169 rows=1 loops=1)
                                                                        Buckets: 1024
↪  Batches: 1  Memory Usage: 9kB
                                                                        ->  Seq Scan
↪  on contacts ct (actual time=6.660..16.143 rows=1 loops=1)
                                                                              Filter:
↪  ((login)::text = 'Beatty_Brahem'::text)
                                                                              Rows
↪  Removed by Filter: 110004
                                                          ->  Index Only Scan using
↪  types_clients_pkey on types_clients (actual time=0.013..0.013 rows=1 loops=1)
                                                                    Index Cond:
↪  (type_client = cl.type_client)
                                                                    Heap Fetches: 1
                                              ->  Index Scan using lignes_commandes_pkey
↪  on lignes_commandes lc (actual time=0.019..0.020 rows=4 loops=10)
                                                  Index Cond: (numero_commande =
↪  commandes.numero_commande)
                                        ->  Index Scan using lots_pkey on lots l (actual
↪  time=0.003..0.003 rows=1 loops=35)
                                              Index Cond: (numero_lot =
↪  lc.numero_lot_expedition)
                                  ->  Hash (actual time=0.009..0.009 rows=1 loops=1)
                                        Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                        ->  Seq Scan on transporteurs (actual
↪  time=0.006..0.007 rows=1 loops=1)
                                              Filter: ((nom)::text = 'Royal Air
↪  Drone'::text)
                                              Rows Removed by Filter: 4
Planning Time: 3.168 ms
Execution Time: 533.689 ms
```

(Le plan complet est sur https://explain.dalibo.com/plan/EQN).

Ce plan est 6 fois plus rapide. La différence essentielle tient dans le filtre effectué en premier : cette fois, c'est sur `contacts`. Puis toute la chaîne des jointures est à nouveau remontée, avec beaucoup moins de lignes qu'auparavant. C'est donc plus rapide, et les *Nested Loops* et *Index Scans* deviennent rentables. L'agrégat ne se fait plus que sur 4 lignes.

Avec le `join_collapse_limit` par défaut à 8, PostgreSQL joignait les 8 premières tables, sans

critère de filtrage vraiment discriminant, puis joignait le résultat à `contacts`. En augmentant `join_collapse_limit`, PostgreSQL s'est permis d'étudier les plans incluants `contacts`, sur lesquels porte le filtre le plus intéressant.

Noter que le temps de planification a plus que doublé, mais il est intéressant de perdre 1 ou 2 ms de planification pour gagner plusieurs secondes à l'exécution.

> – À l'inverse, sans modifier de paramètre, comment modifier la requête pour qu'elle s'exécute plus rapidement ?

Si l'on a accès au code de la requête, il est possible de la modifier afin que la table la plus discriminante figure dans les 8 premières tables.

```
RESET join_collapse_limit ;

SHOW join_collapse_limit ;

 join_collapse_limit
---------------------
 8

EXPLAIN (ANALYZE, COSTS OFF)
SELECT SUM (reglements.montant) AS somme_reglements
FROM       factures
INNER JOIN reglements USING (numero_facture)
INNER JOIN commandes USING (numero_commande)
INNER JOIN clients cl USING (client_id)
INNER JOIN contacts ct ON (ct.contact_id = cl.contact_id)  --- jointure déplacée
INNER JOIN types_clients USING (type_client)
INNER JOIN lignes_commandes lc USING (numero_commande)
INNER JOIN lots l ON (l.numero_lot = lc.numero_lot_expedition)
INNER JOIN transporteurs USING (transporteur_id)
WHERE magasin.transporteurs.nom = 'Royal Air Drone'
AND login = 'Beatty_Brahem' ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Aggregate (actual time=573.108..573.115 rows=1 loops=1)
   ->  Hash Join (actual time=498.176..573.103 rows=4 loops=1)
         Hash Cond: (l.transporteur_id = transporteurs.transporteur_id)
         ->  Hash Join (actual time=415.225..573.077 rows=35 loops=1)
               Hash Cond: ((reglements.numero_facture)::text =
↪  (factures.numero_facture)::text)
               ->  Seq Scan on reglements (actual time=0.003..92.461 rows=1055812
↪  loops=1)
               ->  Hash (actual time=376.019..376.025 rows=35 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 10kB
                     ->  Nested Loop (actual time=309.851..376.006 rows=35 loops=1)
                           ->  Nested Loop (actual time=309.845..375.889 rows=35
↪  loops=1)
                                 ->  Hash Join (actual time=309.809..375.767 rows=10
↪  loops=1)
                                       Hash Cond: (factures.numero_commande =
↪  commandes.numero_commande)
```

```
                                        ->  Seq Scan on factures (actual
↪  time=0.011..85.450 rows=1000000 loops=1)
                                      ->  Hash (actual time=205.640..205.644 rows=10
↪  loops=1)
                                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                      ->  Hash Join (actual time=48.891..205.625
↪  rows=10 loops=1)
                                              Hash Cond: (commandes.client_id =
↪  cl.client_id)
                                              ->  Seq Scan on commandes (actual
↪  time=0.003..92.731 rows=1000000 loops=1)
                                          ->  Hash (actual time=27.823..27.826
↪  rows=1 loops=1)
                                                Buckets: 1024  Batches: 1
↪  Memory Usage: 9kB
                                                ->  Nested Loop (actual
↪  time=16.526..27.823 rows=1 loops=1)
                                                      ->  Hash Join (actual
↪  time=16.509..27.804 rows=1 loops=1)
                                                            Hash Cond:
↪  (cl.contact_id = ct.contact_id)
                                                            ->  Seq Scan on
↪  clients cl (actual time=0.002..6.978 rows=100000 loops=1)
                                                          ->  Hash (actual
↪  time=11.785..11.786 rows=1 loops=1)
                                                               Buckets: 1024
↪  Batches: 1  Memory Usage: 9kB
                                                               ->  Seq Scan
↪  on contacts ct (actual time=4.188..11.781 rows=1 loops=1)
                                                                     Filter:
↪  ((login)::text = 'Beatty_Brahem'::text)
                                                                     Rows
↪  Removed by Filter: 110004
                                                  ->  Index Only Scan using
↪  types_clients_pkey on types_clients (actual time=0.013..0.013 rows=1 loops=1)
                                                            Index Cond:
↪  (type_client = cl.type_client)
                                                            Heap Fetches: 1
                                  ->  Index Scan using lignes_commandes_pkey on
↪  lignes_commandes lc (actual time=0.008..0.009 rows=4 loops=10)
                                        Index Cond: (numero_commande =
↪  factures.numero_commande)
                            ->  Index Scan using lots_pkey on lots l (actual
↪  time=0.002..0.002 rows=1 loops=35)
                                  Index Cond: (numero_lot = lc.numero_lot_expedition)
          ->  Hash (actual time=0.008..0.008 rows=1 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on transporteurs (actual time=0.006..0.007 rows=1 loops=1)
                     Filter: ((nom)::text = 'Royal Air Drone'::text)
                     Rows Removed by Filter: 4
Planning Time: 1.543 ms
Execution Time: 573.169 ms
```

(Plan complet sur https://explain.dalibo.com/plan/suz)

Le plan redevient très voisin du précédent, sans forcément être aussi optimal que celui ci-dessus. Mais

l'inefficacité majeure est corrigée.

La conclusion de cette exercice est que, lorsque c'est possible, il vaut mieux mettre en première jointure les tables portant les critères les plus discriminants. Dans le cas où des requêtes contenant de nombreuses jointures sont générées dynamiquement, qu'elles sont fréquentes, et si le temps de planification est ridicule par rapport au gain de l'exécution, alors il est envisageable de monter globalement `join_collapse_limit` (NB: il est aussi possible de positionner ce paramètre sur le rôle de l'utilisateur ou encore sur les paramètres de la base).

1.14.5 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 grosses 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** ;

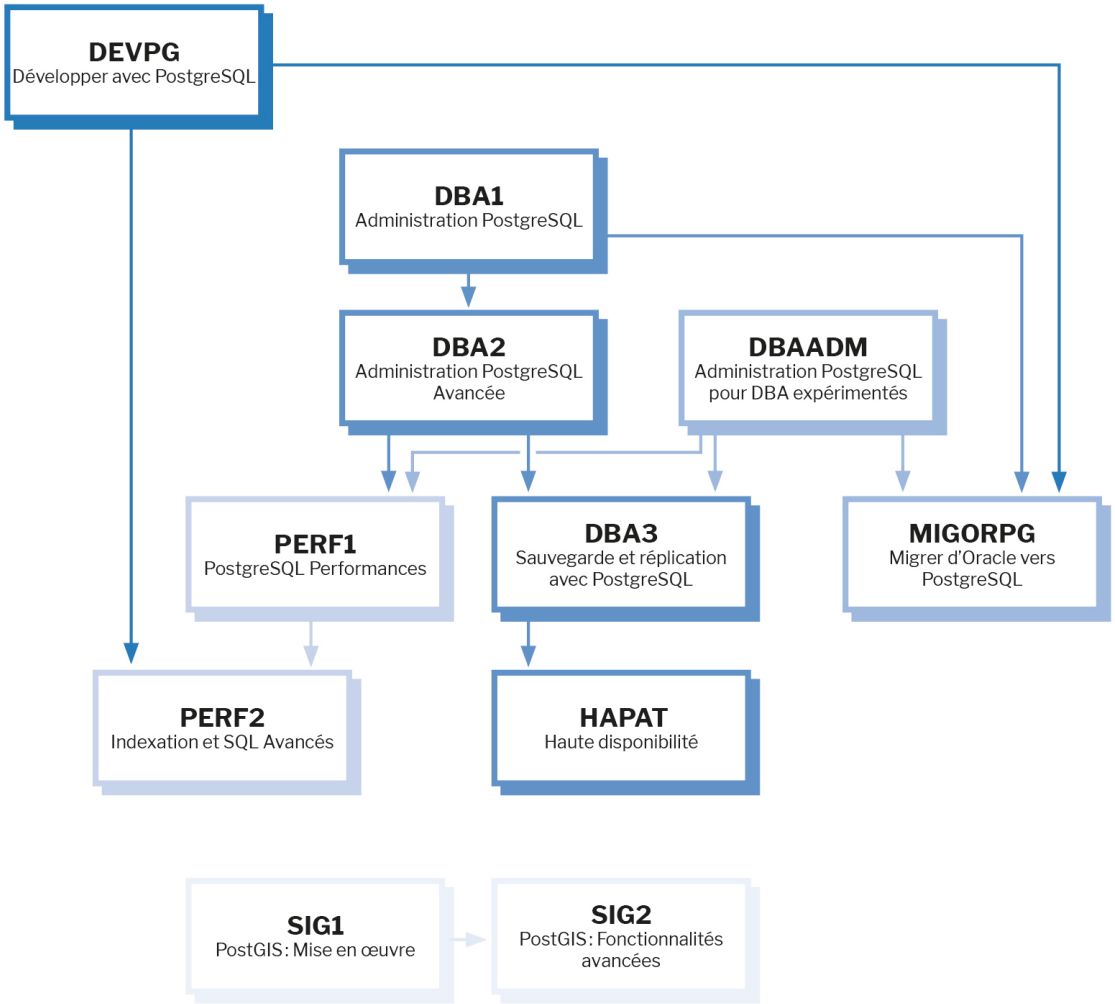
- que la ré-écriture de requête est souvent complexe
- néanmoins, surveillez un certain nombre de choses :
  - transtypages implicites suspects ;
  - jointures externes inutiles ;
  - sous-requêtes imbriquées ;
  - jointures inutiles (données constantes).

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.