Module J6

Références des nœuds

24.04

Table des matières

Sur ce document

Formation	Module J6
Titre	Références des nœuds
Révision	24.04
PDF	https://dali.bo/j6_pdf
EPUB	https://dali.bo/j6_epub
HTML	https://dali.bo/j6_html
Slides	https://dali.bo/j6_slides

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

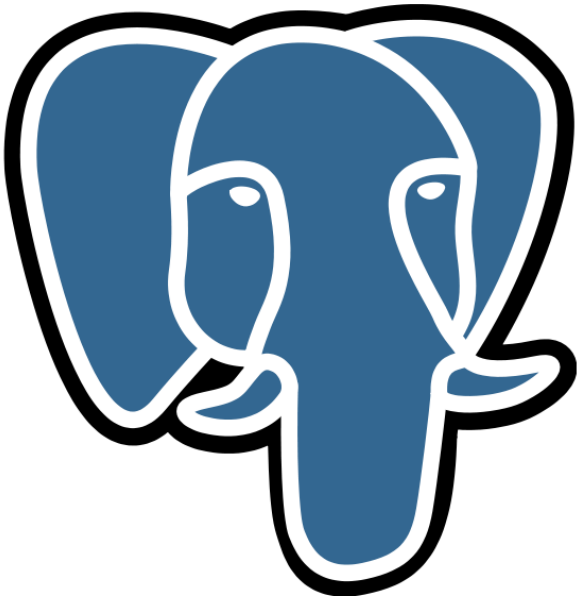
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Référence sur les nœuds d’exécution



1.1 INTRODUCTION

- Quatre types de nœuds
  - parcours (de table, d'index, de TID, etc.)
  - jointures (*Nested Loop*, *Sort*/*Merge Join*, *Hash Join*)
  - opérateurs sur des ensembles (*Append*, *Except*, *Intersect*, etc.)
  - et quelques autres (*Sort*, *Aggregate*, *Unique*, *Limit*, *Materialize*)

Un plan d'exécution est un arbre. Chaque nœud de l'arbre est une opération à effectuer par l'exécuteur. Le planificateur arrange les nœuds pour que le résultat final soit le bon, et qu'il soit récupéré le plus rapidement possible.

Il y a quatre types de nœuds :

- les parcours, qui permettent de lire les données dans les tables en passant :
  - soit par la table ;
  - soit par l'index ;
- les jointures, qui permettent de joindre deux ensembles de données ;
- les opérateurs sur des ensembles, qui là aussi vont joindre deux ensembles ou plus ;
- et les opérations sur un seul ensemble : tri, limite, agrégat, etc.

Cet annexe a pour but d'entrer dans le détail de chaque type de nœuds, ses avantages et inconvénients.

1.2 PARCOURS

- Ne prend rien en entrée
- Mais renvoie un ensemble de données
  - trié ou non, filtré ou non
- Exemples typiques
  - parcours séquentiel d'une table, avec ou sans filtrage des enregistrements produits
  - parcours par un index, avec ou sans filtrage supplémentaire

Les parcours sont les seules opérations qui lisent les données des tables (standards, temporaires ou non journalisées). Elles ne prennent donc rien en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

Il existe trois types de parcours que nous allons détailler :

- le parcours de table ;
- le parcours d'index ;
- le parcours de bitmap index ;

tous les trois pouvant recevoir des filtres supplémentaires en sortie.

Nous verrons aussi que PostgreSQL propose d'autres types de parcours.

1.2.1 Parcours de table

- Parcours séquentiel de la table (*Sequential Scan* ou *Seq Scan*)
  - parallélisation possible (*Parallel Seq Scan*)
- Aussi appelé *Full table scan* par d'autres SGBD
- La table est lue entièrement
  - même si seulement quelques lignes satisfont la requête
  - sauf pour `LIMIT` sans `ORDER BY`
- Séquentiellement, par bloc de 8 ko
- Optimisation : `synchronize_seqscans`

Le parcours le plus simple est le parcours séquentiel. La table est lue complètement, de façon séquentielle, par bloc de 8 ko. Les données sont lues dans l'ordre physique sur disque, donc les données ne sont pas envoyées triées au nœud supérieur.

Cela fonctionne dans tous les cas, car il n'y a besoin de rien de plus pour le faire : un parcours d'index nécessite un index, un parcours de table ne nécessite rien de plus que la table.

Le parcours de table est intéressant pour les performances dans deux cas :

- les très petites tables ;
- les grosses tables où la majorité des lignes doit être renvoyée.

Voici quelques exemples à partir de ce jeu de tests :

```
CREATE TABLE t1 (c1 integer);
INSERT INTO t1 (c1) SELECT generate_series(1, 100000);
ANALYZE t1;
```

Ici, nous faisons une lecture complète de la table. De ce fait, un parcours séquentiel sera plus rapide du fait de la rapidité de la lecture séquentielle des blocs :

```
EXPLAIN SELECT * FROM t1 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1443.00 rows=100000 width=4)
```

Le coût est relatif au nombre de blocs lus, au nombre de lignes décodées et à la valeur des paramètres `seq_page_cost` et `cpu_tuple_cost`. Si un filtre est ajouté, cela aura un coût supplémentaire dû à l'application du filtre sur toutes les lignes de la table (pour trouver celles qui correspondent à ce filtre) :

```
EXPLAIN SELECT * FROM t1 WHERE c1=1000 ;
```

```
                      QUERY PLAN
------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1693.00 rows=1 width=4)
   Filter: (c1 = 1000)
```

Ce coût supplémentaire dépend du nombre de lignes dans la table et de la valeur du paramètre `cpu_operator_cost` (défaut 0,0025) ou de la valeur du paramètre `COST` de la fonction appelée. L'exemple ci-dessus montre le coût (1693) en utilisant l'opérateur standard d'égalité. Maintenant, si on crée une fonction qui utilise cet opérateur (mais écrite en PL/pgSQL, cela reste invisible pour PostgreSQL), avec un coût forcé à 10 000, cela donne :

```
CREATE FUNCTION egal(integer,integer) RETURNS boolean LANGUAGE plpgsql AS $$
begin
return $1 = $2;
end
$$
COST 10000;

EXPLAIN SELECT * FROM t1 WHERE egal(c1, 1000) ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..2501443.00 rows=33333 width=4)
   Filter: egal(c1, 1000)
```

La ligne *Filter* indique le filtre réalisé. Le nombre de lignes indiqué par `rows=` est le nombre de lignes après filtrage. Pour savoir combien de lignes ne satisfont pas le prédicat de la clause `WHERE`, il faut exécuter la requête et donc utiliser l'option `EXPLAIN` :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM t1 WHERE c1=1000 ;

                      QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..1693.00 rows=1 width=4)
          (actual time=0.236..19.615 rows=1 loops=1)
   Filter: (c1 = 1000)
   Rows Removed by Filter: 99999
   Buffers: shared hit=443
 Planning time: 0.110 ms
 Execution time: 19.649 ms
```

Il s'agit de la ligne `Rows Removed by Filter`.

L'option `BUFFERS` permet en plus de savoir le nombre de blocs lus dans le cache et hors du cache.

Le calcul réalisé pour le coût final est le suivant :

```
SELECT
  round((
    current_setting('seq_page_cost')::numeric*relpages      +
    current_setting('cpu_tuple_cost')::numeric*reltuples    +
    current_setting('cpu_operator_cost')::numeric*reltuples
        )::numeric, 2)
  AS cout_final
FROM pg_class
WHERE relname='employes';
```

Si le paramètre `synchronize_seqscans` est activé (et il l'est par défaut), le processus qui entame une lecture séquentielle cherche en premier lieu si un autre processus ne ferait pas une lecture séquentielle de la même table. Si c'est le cas, Le second processus démarre son parcours de table à l'endroit où le premier processus est en train de lire, ce qui lui permet de profiter des données mises en cache par ce processus. L'accès au disque étant bien plus lent que l'accès mémoire, les processus restent naturellement synchronisés pour le reste du parcours de la table, et les lectures ne sont donc réalisées qu'une seule fois. Le début de la table restera à être lu indépendamment. Cette optimisation permet de diminuer le nombre de blocs lus par chaque processus en cas de lectures parallèles de la même table.

Il est possible, pour des raisons de tests, ou pour tenter de maintenir la compatibilité avec du code partant de l'hypothèse (erronée) que les données d'une table sont toujours retournées dans le même ordre, de désactiver ce type de parcours en positionnant le paramètre `synchronize_seqscans` à `off`.

Une nouvelle optimisation vient de la parallélisation. Depuis la version 9.6, il est possible d'obtenir un parcours de table parallélisé. Dans ce cas, le nœud s'appelle un *Parallel Seq Scan*. Le processus responsable de la requête demande l'exécution de plusieurs processus, appelés des *workers* qui auront tous pour charge de lire la table et d'appliquer le filtre. Chaque *worker* travaillera sur des blocs différents. Le prochain bloc à lire est enregistré en mémoire partagée. Quand un *worker* a terminé de travailler sur un bloc, il consulte la mémoire partagée pour connaître le prochain bloc à traiter, et incrémente ce numéro pour que le *worker* suivant puisse travailler sur un autre bloc. Il n'y a aucune assurance que chaque *worker* travaillera sur le même nombre de blocs. Voici un exemple de plan parallélisé pour un parcours de table :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT sum(c2) FROM t1 WHERE c1 BETWEEN 100000 AND 600000 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Finalize Aggregate  (cost=12196.94..12196.95 rows=1 width=8)
                    (actual time=91.886..91.886 rows=1 loops=1)
   Buffers: shared hit=1277
   ->  Gather  (cost=12196.73..12196.94 rows=2 width=8)
             (actual time=91.874..91.880 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=1277
         ->  Partial Aggregate  (cost=11196.73..11196.74 rows=1 width=8)
                           (actual time=83.760..83.760 rows=1 loops=3)
               Buffers: shared hit=4425
               ->  Parallel Seq Scan on t1
                              (cost=0.00..10675.00 rows=208692 width=4)
                     (actual time=12.727..62.608 rows=166667 loops=3)
                     Filter: ((c1 >= 100000) AND (c1 <= 600000))
                     Rows Removed by Filter: 166666
                     Buffers: shared hit=4425
 Planning time: 0.528 ms
 Execution time: 94.877 ms
```

Dans ce cas, le planificateur a prévu l'exécution de deux *workers*, et deux ont bien été lancés lors de l'exécution de la requête.

1.2.2 Parcours d'index

- Parcours aléatoire de l'index
- Pour chaque enregistrement correspondant à la recherche
  - parcours non séquentiel de la table (pour vérifier la visibilité de la ligne)
- Gros gain en performance si filtre très sélectif
- Les lignes renvoyées sont triées
- Parallélisation possible
  - B-Tree uniquement
- Sur d'autres SGBD : `INDEX RANGE SCAN` + `TABLE ACCESS BY INDEX ROWID`

Parcourir une table prend du temps, surtout quand on cherche à ne récupérer que quelques lignes de cette table. Le but d'un index est donc d'utiliser une structure de données optimisée pour satisfaire une recherche particulière (on parle de prédicat).

Cette structure est un arbre. La recherche consiste à suivre la structure de l'arbre pour trouver le premier enregistrement correspondant au prédicat, puis suivre les feuilles de l'arbre jusqu'au dernier enregistrement vérifiant le prédicat. De ce fait, et étant donné la façon dont l'arbre est stocké sur disque, cela peut provoquer des déplacements de la tête de lecture.

L'autre problème des performances sur les index (mais cette fois, spécifique à PostgreSQL) est que les informations de visibilité des lignes sont uniquement stockées dans la table. Cela veut dire que, pour chaque élément de l'index correspondant au filtre, il va falloir lire la ligne dans la table pour vérifier si cette dernière est visible pour la transaction en cours. Il est de toute façons, pour la plupart des requêtes, nécessaire d'aller inspecter l'enregistrement de la table pour récupérer les autres colonnes nécessaires au bon déroulement de la requête, qui ne sont la plupart du temps pas stockées dans l'index. Ces enregistrements sont habituellement éparpillés dans la table, et retournés dans un ordre totalement différent de leur ordre physique par le parcours sur l'index. Cet accès à la table génère donc énormément d'accès aléatoires. Or, ce type d'activité est généralement le plus lent sur un disque magnétique. C'est pourquoi le parcours d'une large portion d'un index est très lent. PostgreSQL ne cherchera à utiliser un index que s'il suppose qu'il aura peu de lignes à récupérer.

Voici l'algorithme permettant un parcours d'index avec PostgreSQL :

- Pour tous les éléments de l'index :
  - chercher l'élément souhaité dans l'index ;
  - lorsqu'un élément est trouvé : vérifier qu'il est visible par la transaction en lisant la ligne dans la table et récupérer les colonnes supplémentaires de la table.

Cette manière de procéder est identique à ce que proposent d'autres SGBD sous les termes d'`INDEX RANGE SCAN`, suivi d'un `TABLE ACCESS BY INDEX ROWID`.

Un parcours d'index est donc très coûteux, principalement à cause des déplacements de la tête de lecture. Le paramètre lié au coût de lecture aléatoire d'une page est par défaut 4 fois supérieur à celui de la lecture séquentielle d'une page. Ce paramètre s'appelle `random_page_cost`. Un parcours d'index n'est préférable à un parcours de table que si la recherche ne va ramener qu'un très faible pourcentage de la table. Et dans ce cas, le gain possible est très important par rapport à un parcours séquentiel de table. Par contre, il se révèle très lent pour lire un gros pourcentage de la table (les accès aléatoires diminuent spectaculairement les performances).

Il est à noter que, contrairement au parcours de table, le parcours d'index renvoie les données triées. C'est le seul parcours à le faire. Il peut même servir à honorer la clause `ORDER BY` d'une requête. L'index est aussi utilisable dans le cas des tris descendants. Dans ce cas, le nœud est nommé *Index Scan Backward*. Ce renvoi de données triées est très intéressant lorsqu'il est utilisé en conjonction avec la clause `LIMIT`.

Il ne faut pas oublier aussi le coût de mise à jour de l'index. Si un index n'est pas utilisé, il coûte cher en maintenance (ajout des nouvelles entrées, suppression des entrées obsolètes, etc.).

Enfin, il est à noter que ce type de parcours est consommateur aussi en CPU.

Voici un exemple montrant les deux types de parcours et ce que cela occasionne comme lecture disque. Commençons par créer une table, lui insérer quelques données et lui ajouter un index :

```
CREATE TABLE t1 (c1 integer, c2 integer);
INSERT INTO t1 VALUES (1,2), (2,4), (3,6);
CREATE INDEX i1 ON t1(c1);
ANALYZE t1;
```

Essayons maintenant de lire la table avec un simple parcours séquentiel :

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c1=2;
```

```
                  QUERY PLAN
-------------------------------------------------
 Seq Scan on t1  (cost=0.00..1.04 rows=1 width=8)
        (actual time=0.020..0.023 rows=1 loops=1)
   Filter: (c1 = 2)
   Rows Removed by Filter: 2
   Buffers: shared hit=1
 Planning time: 0.163 ms
 Execution time: 0.065 ms
```

Seq Scan est le titre du nœud pour un parcours séquentiel. Profitons-en pour noter qu'il a fait de lui-même un parcours séquentiel. En effet, la table est tellement petite (8 ko) qu'utiliser l'index coûterait forcément plus cher. Grâce à l'option `BUFFERS`, nous savons que seul un bloc a été lu.

Pour faire un parcours d'index, nous allons désactiver les parcours séquentiels et réinitialiser les statistiques :

```
SET enable_seqscan TO off;
```

Il existe aussi un paramètre, appelé `enable_indexscan`, pour désactiver les parcours d'index.

Maintenant relançons la requête :

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c1=2;

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using i1 on t1  (cost=0.13..8.15 rows=1 width=8)
                   (actual time=0.117..0.121 rows=1 loops=1)
   Index Cond: (c1 = 2)
   Buffers: shared hit=1 read=1
 Planning time: 0.174 ms
 Execution time: 0.174 ms
```

Nous avons bien un parcours d'index. Vérifions les statistiques sur l'activité :

```
SELECT relname,
  heap_blks_read, heap_blks_hit,
  idx_blks_read, idx_blks_hit
FROM pg_statio_user_tables
WHERE relname='t1';

 relname | heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
---------+----------------+---------------+---------------+--------------
 t1      |              0 |             1 |             1 |            0
```

Une page disque a été lue dans l'index (colonne `idx_blks_read` à 1) et une autre a été lue dans la table (colonne `heap_blks_hit` à 1). Le plus impactant est l'accès aléatoire sur l'index et la table. Il serait bon d'avoir une lecture de l'index, puis une lecture séquentielle de la table. C'est le but du *Bitmap Index Scan*.

1.2.3 Parcours d'index bitmap

- *Bitmap Index Scan* / *Bitmap Heap Scan*
- Réduire les allers-retours index <-> table
  - trouver les blocs de l'index
  - lecture des blocs intéressantde la table
- Combiner plusieurs index en mémoire
  - nœud *BitmapAnd*
  - nœud *BitmapOr*
- Coût de démarrage généralement important (pas intéressant avec `LIMIT`)
- Parallélisation possible
- B-Tree uniquement
- Sensible à :
  - `effective_io_concurrency`

Principe :

D'autres SGBD connaissent les index bitmap, mais sous PostgreSQL, un index bitmap n'a aucune existence sur disque. Il est créé en mémoire lorsque son utilisation a un intérêt. Il se manifeste par le couple de noeuds *Bitmap Index Scan* et *Bitmap Heap Scan*.

Le principe est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table. Même avec un SSD, il évite d'aller chercher trop souvent les mêmes blocs et améliore l'utilisation du cache. Son principe est le suivant :

- lecture en une passe de l'index (*Bitmap Index Scan*) ;
- récupération des TID (*tuple id*) en mémoire (1 bit par ligne dans le cas idéal) ;
- tri des blocs à parcourir dans la table dans l'ordre physique de la table (pas dans l'ordre logique de l'index) ;
- lecture en une passe de la partie intéressante de la table (*Bitmap Heap Scan*).

Un bitmap est souvent utilisé quand il y a un grand nombre de valeurs à filtrer, notamment pour les clauses `IN` et `ANY`.

Ce type d'index présente un autre gros intérêt : pouvoir combiner plusieurs index en mémoire. Les bitmaps de TID obtenus se combinent facilement avec des opérations booléennes `AND` et `OR`.

Exemple :

Cet exemple utilise PostgreSQL 15 dans sa configuration par défaut. La table suivante possède trois champs indexés susceptibles de servir de critère de recherche :

```
CREATE UNLOGGED TABLE tbt
(i int GENERATED ALWAYS AS IDENTITY PRIMARY KEY, j int, k int, t text)  ;

INSERT INTO tbt (j,k,t)
SELECT (i / 1000) , i / 777, chr (64+ (i % 58))
FROM generate_series(1,10000000) i ;

CREATE INDEX tbt_j_idx ON tbt (j) ;
CREATE INDEX tbt_k_idx ON tbt (k) ;
CREATE INDEX tbt_t_idx ON tbt (t) ;

VACUUM ANALYZE tbt ;
```

Lors de la recherche sur les plusieurs critères, les lignes renvoyées par les *Bitmap Index Scan* peuvent être combinées :

```
-- pour la lisibilité des plans
SET max_parallel_workers_per_gather  TO 0 ;
SET jit TO off ;

EXPLAIN (ANALYZE, BUFFERS, VERBOSE, SETTINGS)
SELECT i, j, k, t  FROM tbt
WHERE j = 8
AND k = 10
AND t = 'a';

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

```
Bitmap Heap Scan on public.tbt  (cost=23.02..27.04 rows=1 width=14) (actual
↪  time=0.598..0.702 rows=9 loops=1)
  Output: i, j, k, t
  Recheck Cond: ((tbt.k = 10) AND (tbt.j = 8))
  Filter: (tbt.t = 'a'::text)
  Rows Removed by Filter: 538
  Heap Blocks: exact=4
  Buffers: shared read=11
  ->  BitmapAnd  (cost=23.02..23.02 rows=1 width=0) (actual time=0.557..0.558
↪  rows=0 loops=1)
        Buffers: shared read=7
        ->  Bitmap Index Scan on tbt_k_idx  (cost=0.00..10.62 rows=824 width=0)
↪  (actual time=0.501..0.501 rows=777 loops=1)
              Index Cond: (tbt.k = 10)
              Buffers: shared read=4
        ->  Bitmap Index Scan on tbt_j_idx  (cost=0.00..12.15 rows=1029 width=0)
↪  (actual time=0.053..0.053 rows=1000 loops=1)
              Index Cond: (tbt.j = 8)
              Buffers: shared read=3
Settings: jit = 'off', max_parallel_workers_per_gather = '0'
Planning Time: 0.114 ms
Execution Time: 0.740 ms
```

Dans le plan précédent[1] :

- deux *Bitmap Index Scan* parcourent séparément deux index, qui remontent l'un 777 lignes (toutes les lignes de la table où `k` vaut 10), l'autre 1000 lignes (où `j` vaut 8) ;
- le troisième index sur `t` est ignoré : il y a trop de lignes avec cette valeur (un décompte en trouverait 172 414), et surtout dispersées dans toute la table ;
- la combinaison des lignes remontées désigne seulement 4 blocs dans la table possédant des lignes correspondant aux deux critères (mention *Heap Blocks*) ;
- le *Bitmap Heap Scan* lit ces 4 blocs séquentiellement, et donc une seule fois chacun, et trouve 547 lignes ;
- la clause *Recheck* vérifie que ces lignes sont réellement visibles (ici rien n'est rejeté) ;
- il reste à appliquer le critère non géré par les index utilisés, soit `t = 'a'` : c'est le rôle de la clause *Filter*, qui écarte 538 lignes et n'en garde que 9.

Le coût de démarrage est généralement important à cause de la lecture préalable de l'index et du tri des TID. Ce type de parcours est donc moins intéressant quand on recherche un coût de démarrage faible (clause `LIMIT`, curseur…). Un parcours d'index simple sera généralement choisi dans ce cas.

Clause OR :

Les index sont également utiles avec une clause `OR` :

```
EXPLAIN (ANALYZE,BUFFERS, COSTS)
SELECT i, j, k, t  FROM tbt
WHERE j = 8
OR k = 10
OR t = 'a';
```

[1] https://explain.dalibo.com/plan/54bc5g7b3c9h76a7

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tbt  (cost=2039.91..59155.01 rows=174831 width=14) (actual
↪  time=27.860..385.638 rows=173623 loops=1)
   Recheck Cond: ((j = 8) OR (k = 10) OR (t = 'a'::text))
   Heap Blocks: exact=54054
   Buffers: shared hit=10 read=54199
   ->  BitmapOr  (cost=2039.91..2039.91 rows=174863 width=0) (actual
↪  time=12.514..12.515 rows=0 loops=1)
         Buffers: shared hit=4 read=151
         ->  Bitmap Index Scan on tbt_j_idx  (cost=0.00..12.18 rows=1033 width=0)
↪  (actual time=0.049..0.049 rows=1000 loops=1)
               Index Cond: (j = 8)
               Buffers: shared read=3
         ->  Bitmap Index Scan on tbt_k_idx  (cost=0.00..10.68 rows=833 width=0)
↪  (actual time=0.028..0.028 rows=777 loops=1)
               Index Cond: (k = 10)
               Buffers: shared hit=4
        ->  Bitmap Index Scan on tbt_t_idx  (cost=0.00..1885.92 rows=172998 width=0)
↪  (actual time=12.435..12.435 rows=172414 loops=1)
               Index Cond: (t = 'a'::text)
               Buffers: shared read=148
 Planning Time: 0.076 ms
 Execution Time: 394.014 ms
```

Ce plan[2] utilise cette fois les trois index. Au final, le *Bitmap Heap Scan* lit quand même toute la table ! En effet, il y a des `t='a'` dans tous les blocs (cas le plus défavorable). 98 % des comparaisons de critères sont tout de même évitées, et ce plan s'avère plus efficace qu'un parcours séquentiel, trois fois plus long sur la même machine.

Rôle du work_mem :

Si le `work_mem` est trop bas, PostgreSQL n'a plus la place de stocker un bit par ligne dans son tableau, mais utilise un bit par page. La mention *lossy* apparaît alors sur la ligne *Heap Blocks*, et toutes les lignes de la page doivent être vérifiées. Avec la requête précédente, la performance est cette fois pire qu'un parcours complet :

```
SET work_mem TO '256kB' ;

EXPLAIN (ANALYZE,BUFFERS, COSTS)
SELECT i, j, k, t  FROM tbt
WHERE j = 8
OR k = 10
OR t = 'a';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tbt  (cost=1955.42..224494.16 rows=167501 width=14) (actual
↪  time=8.987..1601.912 rows=173623 loops=1)
   Recheck Cond: ((j = 8) OR (k = 10) OR (t = 'a'::text))
   Rows Removed by Index Recheck: 9350021
   Heap Blocks: exact=2620 lossy=51434
   Buffers: shared read=54209
```

[2] https://explain.dalibo.com/plan/a5ebd41930fcccde

```
   ->  BitmapOr  (cost=1955.42..1955.42 rows=167532 width=0) (actual
↪  time=8.498..8.500 rows=0 loops=1)
         Buffers: shared read=155
         ->  Bitmap Index Scan on tbt_j_idx  (cost=0.00..12.19 rows=1034 width=0)
↪  (actual time=0.451..0.451 rows=1000 loops=1)
               Index Cond: (j = 8)
               Buffers: shared read=3
         ->  Bitmap Index Scan on tbt_k_idx  (cost=0.00..10.65 rows=828 width=0)
↪  (actual time=0.034..0.034 rows=777 loops=1)
               Index Cond: (k = 10)
               Buffers: shared read=4
        ->  Bitmap Index Scan on tbt_t_idx  (cost=0.00..1806.96 rows=165670 width=0)
↪  (actual time=8.011..8.011 rows=172414 loops=1)
               Index Cond: (t = 'a'::text)
               Buffers: shared read=148
 Planning Time: 0.089 ms
 Execution Time: 1610.028 ms
```

effective_io_concurrency :

Les parcours *Bitmap Heap Scan* sont sensibles au paramètre `effective_io_concurrency`, qu'il peut être très bénéfique d'augmenter. `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[3], 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[4].

Enfin, le paramètre `enable_bitmapscan` permet d'activer ou de désactiver l'utilisation des parcours d'index bitmap.

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

1.2.4 Parcours d'index seul

```
SELECT c1 FROM t1 WHERE c1<10
```

- Avant 9.2 : PostgreSQL devait lire l'index + la table
- À présent : le planificateur utilise la *Visibility Map*
  - nœud *Index Only Scan*
  - index B-Tree
  - index SP-GiST
  - index GiST => Types : point, box, inet, range

Voici un exemple sous PostgreSQL 9.1 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM generate_series(1,10000000) a;
INSERT 0 10000000
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                                (actual time=209.569..3314.717 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                              (actual time=197.177..197.177 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 3323.497 ms

b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                                 (actual time=48.620..269.907 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                                (actual time=35.780..35.780 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 273.761 ms
```

Donc 3 secondes pour la première exécution (avec un cache pas forcément vide), et 273 millisecondes pour la deuxième exécution (et les suivantes, non affichées ici).

Voici ce que cet exemple donne en 9.2 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM (select generate_series(1,10000000)) AS t(a);
INSERT 0 10000000
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                                 (cost=0.00..3084.77 rows=86656 width=11)
                                 (actual time=0.080..97.942 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
 Total runtime: 108.134 ms

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                                 (cost=0.00..3084.77 rows=86656 width=11)
                                 (actual time=0.024..26.954 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=347
 Total runtime: 34.352 ms
```

Donc, même à froid, il est déjà pratiquement trois fois plus rapide que la version 9.1, à chaud. La version 9.2 est dix fois plus rapide à chaud.

Essayons maintenant en désactivant les parcours d'index seul :

```
b1=# SET enable_indexonlyscan TO off;
SET
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                                 (actual time=29.256..2992.289 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=346 read=43834 written=2022
```

```
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                              (actual time=27.004..27.004 rows=89432 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=346
 Total runtime: 3000.502 ms
```

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000 ;

                               QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                               (actual time=23.533..1141.754 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=2 read=44178
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                              (actual time=21.592..21.592 rows=89432 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=2 read=344
 Total runtime: 1146.538 ms
```

On retombe sur les performances de la version 9.1.

Maintenant, essayons avec un cache vide (niveau PostgreSQL et système) :

- en 9.1

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000 ;

                               QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                              (actual time=126.624..9750.245 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Buffers: shared hit=2 read=44250
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                            (actual time=112.542..112.542 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=2 read=346
 Total runtime: 9765.670 ms
```

- en 9.2 :

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000 ;

                               QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                                (cost=0.00..3084.77 rows=86656 width=11)
                                (actual time=11.592..63.379 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=2 read=345
 Total runtime: 70.188 ms
```

La version 9.1 met 10 secondes à exécuter la requête, alors que la version 9.2 ne met que 70 millisecondes (elle est donc 142 fois plus rapide).

Voir aussi cet article de blog[5].

1.2.5 Parcours : autres

- *TID Scan*
- *Function Scan*
- *Values*
- *Result*

Il existe d'autres parcours, bien moins fréquents ceci dit.

TID est l'acronyme de *Tuple ID*. C'est en quelque sorte un pointeur vers une ligne. Un *TID Scan* est un parcours de *TID*. Ce type de parcours est généralement utilisé en interne par PostgreSQL. Il est possible de le désactiver via le paramètre `enable_tidscan`.

```
b1=# EXPLAIN SELECT * FROM pg_class WHERE ctid = '(1,1)';

                        QUERY PLAN
-----------------------------------------------------------
 Tid Scan on pg_class  (cost=0.00..4.01 rows=1 width=265)
   TID Cond: (ctid = '(1,1)'::tid)
```

Un *Function Scan* est utilisé par les fonctions renvoyant des ensembles (appelées SRF pour *Set Returning Functions*). En voici un exemple :

```
b1=# EXPLAIN SELECT * FROM generate_series(1, 1000) ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Function Scan on generate_series  (cost=0.00..10.00 rows=1000 width=4)
```

`VALUES` est une clause de l'instruction `INSERT`, mais `VALUES` peut aussi être utilisé comme une table dont on spécifie les valeurs. Par exemple :

```
b1=# VALUES (1), (2);

 column1
---------
       1
       2
(2 rows)

b1=# SELECT * FROM (VALUES ('a', 1), ('b', 2), ('c', 3)) AS tmp(c1, c2);
```

[5] https://pgsnaga.blogspot.com/2011/10/index-only-scans-and-heap-block-reads.html

```
 c1 | c2
----+----
 a  |  1
 b  |  2
 c  |  3
(3 rows)
```

Le planificateur utilise un nœud spécial appelé *Values Scan* pour indiquer un parcours sur cette clause :

```
b1=# EXPLAIN
b1-# SELECT *
b1-# FROM (VALUES ('a', 1), ('b', 2), ('c', 3))
b1-# AS tmp(c1, c2) ;

                          QUERY PLAN
--------------------------------------------------------------
 Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=36)
```

Enfin, le nœud *Result* n'est pas à proprement parler un nœud de type parcours. Il y ressemble dans le fait qu'il ne prend aucun ensemble de données en entrée et en renvoie un en sortie. Son but est de renvoyer un ensemble de données suite à un calcul. Par exemple :

```
b1=# EXPLAIN SELECT 1+2 ;

               QUERY PLAN
----------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=0)
```

1.3 JOINTURES

- Prend 2 ensembles de données en entrée
  - *inner* (interne)
  - *outer* (externe)
- Et renvoie un seul ensemble de données
- Exemples typiques :
  - *Nested Loop*, *Merge Join*, *Hash Join*

Le but d'une jointure est de grouper deux ensembles de données pour n'en produire qu'un seul. L'un des ensembles est appelé ensemble interne (*inner set*), l'autre est appelé ensemble externe (*outer set*).

Le planificateur de PostgreSQL est capable de traiter les jointures grâce à trois nœuds :

- *Nested Loop*, une boucle imbriquée ;
- *Merge Join*, un parcours des deux ensembles triés ;
- *Hash Join*, une jointure par tests des données hachées.

1.3.1 Nested Loops

Boucles imbriquées

- Pour chaque ligne de la relation externe
  - pour chaque ligne de la relation interne
    - si la condition de jointure est avérée : émettre la ligne en résultat
- L'ensemble externe n'est parcouru qu'une fois
- L'ensemble interne est parcouru pour chaque ligne de l'ensemble externe
  - un index utilisable sur l'ensemble interne augmente fortement les performances !

Étant donné le pseudo-code indiqué ci-dessus, on s'aperçoit que l'ensemble externe n'est parcouru qu'une fois alors que l'ensemble interne est parcouru pour chaque ligne de l'ensemble externe. Le coût de ce nœud est donc proportionnel à la taille des ensembles. Il est intéressant pour les petits

ensembles de données, et encore plus lorsque l'ensemble interne dispose d'un index satisfaisant la condition de jointure.

En théorie, il s'agit du type de jointure le plus lent, mais il a un gros intérêt : il n'est pas nécessaire de trier les données ou de les hacher avant de commencer à traiter les données. Il a donc un coût de démarrage très faible, ce qui le rend très intéressant si cette jointure est couplée à une clause `LIMIT`, ou si le nombre d'itérations (donc le nombre d'enregistrements de la relation externe) est faible.

Il est aussi très intéressant, car il s'agit du seul nœud capable de traiter des jointures sur des conditions différentes de l'égalité ainsi que des jointures de type `CROSS JOIN`.

Voici un exemple avec deux parcours séquentiels :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
    WHERE pg_class.relnamespace=pg_namespace.oid ;
```

```
                                QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.00..37.18 rows=281 width=307)
   Join Filter: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
   ->  Materialize  (cost=0.00..1.09 rows=6 width=117)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
```

Et un exemple avec un parcours séquentiel et un parcours d'index :

```
b1=# SET random_page_cost TO 0.5;
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
    WHERE pg_class.relnamespace=pg_namespace.oid;
```

```
                                QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.00..33.90 rows=281 width=307)
   ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
   ->  Index Scan using pg_class_relname_nsp_index on pg_class
       (cost=0.00..4.30 rows=94 width=194)
         Index Cond: (relnamespace = pg_namespace.oid)
```

Le paramètre `enable_nestloop` permet d'activer ou de désactiver ce type de nœud.

1.3.2 Merge Join

Jointure d'ensembles triés

- Trier l'ensemble interne
- Trier l'ensemble externe
- Tant qu'il reste des lignes dans un des ensembles
  - lire les deux ensembles en parallèle
  - si la condition de jointure est avérée : émettre la ligne
- Parcourir les deux ensembles triés (d'où *Sort-Merge Join*)
- Ne gère que les conditions avec égalité
- Produit un ensemble résultat trié
- Le plus rapide sur de gros ensembles de données

Contrairement au *Nested Loop*, le *Merge Join* ne lit qu'une fois chaque ligne, sauf pour les valeurs dupliquées. C'est d'ailleurs son principal atout.

L'algorithme est assez simple. Les deux ensembles de données sont tout d'abord triés, puis ils sont parcourus ensemble. Lorsque la condition de jointure est vraie, la ligne résultante est envoyée dans l'ensemble de données en sortie.

L'inconvénient de cette méthode est que les données en entrée doivent être triées. Trier les données peut prendre du temps, surtout si les ensembles de données sont volumineux. Cela étant dit, le *Merge Join* peut s'appuyer sur un index pour accélérer l'opération de tri (ce sera alors forcément un *Index Scan*). Une table clusterisée peut aussi accélérer l'opération de tri. Néanmoins, il faut s'attendre à avoir un coût de démarrage important pour ce type de nœud, ce qui fait qu'il sera facilement disqualifié si une clause `LIMIT` est à exécuter après la jointure.

Le gros avantage du tri sur les données en entrée est que les données reviennent triées. Cela peut avoir son avantage dans certains cas.

Voici un exemple pour ce nœud :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
    WHERE pg_class.relnamespace=pg_namespace.oid ;

                                QUERY PLAN
------------------------------------------------------------------------
 Merge Join  (cost=23.38..27.62 rows=281 width=307)
   Merge Cond: (pg_namespace.oid = pg_class.relnamespace)
   ->  Sort  (cost=1.14..1.15 rows=6 width=117)
         Sort Key: pg_namespace.oid
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
   ->  Sort  (cost=22.24..22.94 rows=281 width=194)
         Sort Key: pg_class.relnamespace
         ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
```

Le paramètre `enable_mergejoin` permet d' activer ou de désactiver ce type de nœud.

1.3.3 Hash Join

Jointure par hachage

- Calculer le hachage de chaque ligne de l'ensemble interne
- Tant qu'il reste des lignes dans l'ensemble externe
  - hacher la ligne lue
  - comparer ce hachage aux lignes hachées de l'ensemble interne
  - si une correspondance est trouvée : émettre la ligne
- Ne gère que les conditions avec égalité
- Idéal pour joindre une grande table à une petite table
- Coût de démarrage important à cause du hachage de la table

La vérification de la condition de jointure peut se révéler assez lente dans beaucoup de cas : elle nécessite un accès à un enregistrement par un index ou un parcours de la table interne à chaque itération dans un *Nested Loop* par exemple. Le *Hash Join* cherche à supprimer ce problème en créant une table de hachage de la table interne. Cela sous-entend qu'il faut au préalable calculer le hachage de chaque ligne de la table interne. Ensuite, il suffit de parcourir la table externe, hacher chaque ligne l'une après l'autre et retrouver le ou les enregistrements de la table interne pouvant correspondre à la valeur hachée de la table externe. On vérifie alors qu'ils répondent bien aux critères de jointure (il peut y avoir des collisions dans un hachage, ou des prédicats supplémentaires à vérifier).

Ce type de nœud est très rapide à condition d'avoir suffisamment de mémoire pour stocker le résultat du hachage de l'ensemble interne. Le paramétrage de `work_mem` et `hash_mem_multiplier` (à partir de la 13) peut donc avoir un gros impact. De même, diminuer le nombre de colonnes récupérées permet de diminuer la mémoire à utiliser pour le hachage, et donc d'améliorer les performances d'un *Hash Join*. Cependant, si la mémoire est insuffisante, il est possible de travailler par groupes de lignes (*batch*). L'algorithme est alors une version améliorée de l'algorithme décrit plus haut, permettant justement de travailler en partitionnant la table interne (on parle de *Hybrid Hash Join*). Il est à noter que ce type de nœud est souvent idéal pour joindre une grande table à une petite table.

Le coût de démarrage peut se révéler important à cause du hachage de la table interne. Il ne sera probablement pas utilisé par l'optimiseur si une clause `LIMIT` est à exécuter après la jointure.

Attention, les données retournées par ce nœud ne sont pas triées.

De plus, ce type de nœud peut être très lent si l'estimation de la taille des tables est mauvaise.

Voici un exemple de *Hash Join* :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
    WHERE pg_class.relnamespace=pg_namespace.oid ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=1.14..15.81 rows=281 width=307)
   Hash Cond: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
   ->  Hash  (cost=1.06..1.06 rows=6 width=117)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
```

Le paramètre `enable_hashjoin` permet d' activer ou de désactiver ce type de nœud.

1.3.4 Suppression d'une jointure

```
SELECT pg_class.relname, pg_class.reltuples
FROM pg_class
LEFT JOIN pg_namespace
       ON pg_class.relnamespace=pg_namespace.oid;
```

- Un index unique existe sur la colonne `oid` de `pg_namespace`
- Jointure inutile
  - sa présence ne change pas le résultat

Sur la requête ci-dessus, la jointure est inutile. En effet, il existe un index unique sur la colonne `oid` de la table `pg_namespace`. De plus, aucune colonne de la table `pg_namespace` ne va apparaître dans le résultat. Autrement dit, que la jointure soit présente ou non, cela ne va pas changer le résultat. Dans ce cas, il est préférable de supprimer la jointure. Si le développeur ne le fait pas, PostgreSQL le fera.

Par exemple, PostgreSQL 8.4 donnait ce plan :

```
b1=# EXPLAIN SELECT pg_class.relname, pg_class.reltuples
    FROM pg_class
    LEFT JOIN pg_namespace ON pg_class.relnamespace=pg_namespace.oid ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Left Join  (cost=1.14..12.93 rows=244 width=68)
   Hash Cond: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..8.44 rows=244 width=72)
   ->  Hash  (cost=1.06..1.06 rows=6 width=4)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=4)
```

Et la même requête exécutée à partir de PostgreSQL 9.0 :

```
b1=# EXPLAIN SELECT pg_class.relname, pg_class.reltuples
    FROM pg_class
    LEFT JOIN pg_namespace ON pg_class.relnamespace=pg_namespace.oid ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=72)
```

On constate que la jointure est ignorée.

Ce genre de requête peut fréquemment survenir surtout avec des générateurs de requêtes comme les ORM. L'utilisation de vues imbriquées peut aussi être la source de ce type de problème.

1.3.5 Ordre de jointure

- Trouver le bon ordre de jointure est un point clé dans la recherche de performances
- Nombre de possibilités en augmentation factorielle avec le nombre de tables
- Si petit nombre, recherche exhaustive
- Sinon, utilisation d'heuristiques et de GEQO (`geqo_threshold`)
  - limite le temps de planification et l'utilisation de mémoire
  - `join_collapse_limit` , `from_collapse_limit` : limites de 8 tables

Sur une requête comme `SELECT * FROM a, b, c...` , les tables `a` , `b` et `c` ne sont pas forcément jointes dans cet ordre. PostgreSQL teste différents ordres pour obtenir les meilleures performances.

Prenons comme exemple la requête suivante :

```
SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
```

Avec une table `a` contenant un million de lignes, une table `b` n'en contenant que 1000 et une table `c` en contenant seulement 10, et une configuration par défaut, son plan d'exécution est celui-ci :

```
b1=# EXPLAIN SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=1.23..18341.35 rows=1 width=12)
   Join Filter: (a.id = b.id)
   ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
   ->  Materialize  (cost=1.23..18176.37 rows=10 width=8)
         ->  Hash Join  (cost=1.23..18176.32 rows=10 width=8)
               Hash Cond: (a.id = c.id)
               ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
               ->  Hash  (cost=1.10..1.10 rows=10 width=4)
                     ->  Seq Scan on c  (cost=0.00..1.10 rows=10 width=4)
```

Le planificateur préfère joindre tout d' abord la table `a` à la table `c`, puis son résultat à la table `b`. Cela lui permet d'avoir un ensemble de données en sortie plus petit (donc moins de consommation mémoire) avant de faire la jointure avec la table `b`.

Cependant, si PostgreSQL se trouve face à une jointure de 25 tables, le temps de calculer tous les plans possibles en prenant en compte l'ordre des jointures sera très important. En fait, plus le nombre de tables jointes est important, et plus le temps de planification va augmenter. Il est nécessaire de prévoir une échappatoire à ce système. En fait, il en existe plusieurs. Les paramètres `from_collapse_limit` et `join_collapse_limit` permettent de spécifier une limite en nombre de tables. Si cette limite est dépassée, PostgreSQL ne cherchera plus à traiter tous les cas possibles de réordonnancement des jointures. Par défaut, ces deux paramètres valent 8, ce qui fait que, dans notre exemple, le planificateur a bien cherché à changer l'ordre des jointures. En configurant ces paramètres à une valeur plus basse, le plan va changer :

```
b1=# SET join_collapse_limit TO 2;
SET
b1=# EXPLAIN SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=27.50..18363.62 rows=1 width=12)
   Join Filter: (a.id = c.id)
   ->  Hash Join  (cost=27.50..18212.50 rows=1000 width=8)
         Hash Cond: (a.id = b.id)
         ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
         ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
               ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
   ->  Materialize  (cost=0.00..1.15 rows=10 width=4)
         ->  Seq Scan on c  (cost=0.00..1.10 rows=10 width=4)
```

Avec un `join_collapse_limit` à 2, PostgreSQL décide de ne pas tester l'ordre des jointures. Le plan fourni fonctionne tout aussi bien, mais son estimation montre qu'elle semble être moins performante (coût de 18 363 au lieu de 18 341 précédemment).

Pour des requêtes avec de très nombreuses tables (décisionnel…), pour ne pas avoir à réordonner les tables dans la clause `FROM`, on peut monter les valeurs de `join_collapse_limit` ou `from_collapse_limit` le temps de la session ou pour un couple utilisateur/base précis. Le faire au niveau global risque de faire exploser les temps de planification d'autres requêtes.

Une autre technique mise en place pour éviter de tester tous les plans possibles est GEQO (*GEnetic Query Optimizer*). Cette technique est très complexe, et dispose d'un grand nombre de paramètres que très peu savent réellement configurer. Comme tout algorithme génétique, il fonctionne par introduction de mutations aléatoires sur un état initial donné. Il permet de planifier rapidement une requête complexe, et de fournir un plan d'exécution acceptable. Il se déclenche lorsque le nombre de tables dans la clause `FROM` est supérieure ou égale à la valeur du paramètre `geqo_threshold`, qui vaut 12 par défaut.

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe[6]). Tant que le paramètre `geqo_seed` ainsi que les autres paramètres contrô-

[6] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f

lant GEQO restent inchangés, le plan obtenu pour une requête donnée restera inchangé. Il est possible de faire varier la valeur de `geqo_seed` pour obtenir d'autres plans (voir la documentation officielle[7] pour approfondir ce point).

[7] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116279

1.4 OPÉRATIONS ENSEMBLISTES

- Prend un ou plusieurs ensembles de données en entrée
- Et renvoie un ensemble de données
- Concernent principalement les requêtes sur des tables partitionnées ou héritées
- Exemples typiques
  - *Append*
  - *Intersect*
  - *Except*

Ce type de nœuds prend un ou plusieurs ensembles de données en entrée et renvoie un seul ensemble de données. Cela concerne surtout les requêtes visant des tables partitionnées ou héritées.

1.4.1 Append

- Prend plusieurs ensembles de données
- Sortie non triée
- Utilisation :
  - tables héritées (dont partitionnement)
  - `UNION ALL` et des `UNION`
  - NB : `UNION` sans `ALL` élimine les doublons (tri !)
- Opération parallélisable (v11)

Un nœud `Append` a pour but de concaténer plusieurs ensembles de données pour n'en faire qu'un, non trié. Ce type de nœud est utilisé dans les requêtes concaténant explicitement des tables (clause `UNION`) ou implicitement (requêtes sur une table mère d'un héritage ou une table partitionnée).

Supposons que la table `t1` est une table mère. Plusieurs tables héritent de cette table : `t1_0`, `t1_1`, `t1_2` et `t1_3`. Voici ce que donne un `SELECT` sur la table mère :

```
b1=# EXPLAIN SELECT * FROM t1 ;

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

```
 Result  (cost=0.00..89.20 rows=4921 width=36)
   ->  Append  (cost=0.00..89.20 rows=4921 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
         ->  Seq Scan on t1_0 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_1 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_2 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_3 t1  (cost=0.00..22.30 rows=1230 width=36)
```

Nouvel exemple avec un filtre sur la clé de partitionnement :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 off

b1=# EXPLAIN SELECT * FROM t1 WHERE c1>250;

                              QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=0.00..101.50 rows=1641 width=36)
   ->  Append  (cost=0.00..101.50 rows=1641 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_0 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_1 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_2 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_3 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
```

Le paramètre `constraint_exclusion` permet d'éviter de parcourir les tables filles qui ne peuvent pas accueillir les données qui nous intéressent. Pour que le planificateur comprenne qu'il peut ignorer certaines tables filles, ces dernières doivent avoir des contraintes `CHECK` qui assurent le planificateur qu'elles ne peuvent pas contenir les données en question :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 on

b1=# EXPLAIN SELECT * FROM t1 WHERE c1>250 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=0.00..50.75 rows=821 width=36)
   ->  Append  (cost=0.00..50.75 rows=821 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_2 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_3 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
```

Une requête utilisant `UNION ALL` passera aussi par un nœud *Append* :

```
b1=# EXPLAIN SELECT 1 UNION ALL SELECT 2 ;

                      QUERY PLAN
------------------------------------------------------
 Result  (cost=0.00..0.04 rows=2 width=4)
   ->  Append  (cost=0.00..0.04 rows=2 width=4)
         ->  Result  (cost=0.00..0.01 rows=1 width=0)
         ->  Result  (cost=0.00..0.01 rows=1 width=0)
```

`UNION ALL` récupère toutes les lignes des deux ensembles de données, même en cas de doublon. Pour n'avoir que les lignes distinctes, il est possible d'utiliser `UNION` *sans* la clause `ALL` mais cela entraîne une déduplication des données, ce qui est souvent coûteux :

```
b1=# EXPLAIN SELECT 1 UNION SELECT 2 ;

                         QUERY PLAN
------------------------------------------------------------
 Unique  (cost=0.05..0.06 rows=2 width=0)
   ->  Sort  (cost=0.05..0.06 rows=2 width=0)
         Sort Key: (1)
         ->  Append  (cost=0.00..0.04 rows=2 width=0)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
```

L'utilisation involontaire de `UNION` au lieu de `UNION ALL` est un problème de performance très fréquent.

Le paramètre `enable_partition_pruning` permet d'activer l'élagage des partitions de la même manière que `constraint_exclusion` pour les tables implémentant l'héritage. Il peut prendre deux valeurs `on` ou `off`. Cette fonctionnalité est activée par défaut.

```
                                    Partitioned table "public.tpart"
 Column |  Type   | Collation | Nullable | Default | Storage  | Compression | Stats
 ↪  target | Description
--------+---------+-----------+----------+---------+----------+-------------+-------
 ↪  -------+-------------
 i      | integer |           |          |         | plain    |             |
 t      | text    |           |          |         | extended |             |
Partition key: RANGE (i)
Partitions: part1 FOR VALUES FROM (0) TO (100),
            part2 FOR VALUES FROM (100) TO (200),
            part3 FOR VALUES FROM (200) TO (300)
```

Avec l'élagage activé, on observe que seules les partitions dont la contrainte `CHECK` correspond au prédicat sont visitées. À la différence du partitionnement par héritage, la table mère n'est pas scannée car elle ne contient pas de données.

```
b2=# EXPLAIN SELECT * FROM tpart WHERE i > 100;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Append  (cost=0.00..5.50 rows=199 width=36)
   ->  Seq Scan on part2 tpart_1  (cost=0.00..2.25 rows=99 width=36)
         Filter: (i > 100)
   ->  Seq Scan on part3 tpart_2  (cost=0.00..2.25 rows=100 width=36)
         Filter: (i > 100)
(5 rows)
```

En désactivant l'élagage, toutes les partitions sont visitées.

```
b2=# SET enable_partition_pruning TO off;
b2=# EXPLAIN SELECT * FROM tpart WHERE i > 100;

                              QUERY PLAN
----------------------------------------------------------------------
 Append  (cost=0.00..7.75 rows=200 width=36)
   ->  Seq Scan on part1 tpart_1  (cost=0.00..2.25 rows=1 width=36)
         Filter: (i > 100)
   ->  Seq Scan on part2 tpart_2  (cost=0.00..2.25 rows=99 width=36)
         Filter: (i > 100)
   ->  Seq Scan on part3 tpart_3  (cost=0.00..2.25 rows=100 width=36)
         Filter: (i > 100)
(7 rows)
```

À partir de la version 11, les fils d'un nœud *Append* sont parallélisables.

1.4.2 MergeAppend

- Append avec optimisation
- Sortie triée
- Utilisation :
  - `UNION ALL`, partitionnement/héritage
  - avec parcours triés
  - idéal avec `LIMIT`

Le nœud *MergeAppend* est une optimisation spécifiquement conçue pour le partitionnement. Elle permet de répondre plus efficacement aux requêtes effectuant un tri sur un `UNION ALL`, soit explicite, soit induit par héritage ou partitionnement. Considérons la requête suivante :

```
SELECT *
FROM (
 SELECT t1.a, t1.b FROM t1
 UNION ALL
 SELECT t2.a, t2.c FROM t2
```

```
) t
ORDER BY a;
```

Il est facile de répondre à cette requête si l'on dispose d'un index sur les colonnes `a` des tables `t1` et `t2` : il suffit de parcourir chaque index en parallèle (assurant le tri sur `a`), en renvoyant la valeur la plus petite.

Pour comparaison, avant la 9.1 et l'introduction du nœud *MergeAppend*, le plan obtenu était celui-ci :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=24129.64..24629.64 rows=200000 width=22)
       (actual time=122.705..133.403 rows=200000 loops=1)
   Sort Key: t1.a
   Sort Method:  quicksort  Memory: 21770kB
   ->  Result  (cost=0.00..6520.00 rows=200000 width=22)
               (actual time=0.013..76.527 rows=200000 loops=1)
         ->  Append  (cost=0.00..6520.00 rows=200000 width=22)
                     (actual time=0.012..54.425 rows=200000 loops=1)
               ->  Seq Scan on t1  (cost=0.00..2110.00 rows=100000 width=23)
                                   (actual time=0.011..19.379 rows=100000 loops=1)
               ->  Seq Scan on t2  (cost=0.00..4410.00 rows=100000 width=22)
                                   (actual time=1.531..22.050 rows=100000 loops=1)
 Total runtime: 141.708 ms
```

Depuis la 9.1, l'optimiseur est capable de détecter qu'il existe un **parcours paramétré**, renvoyant les données triées sur la clé demandée (`a`), et utilise la stratégie *MergeAppend* :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Merge Append  (cost=0.72..14866.72 rows=300000 width=23)
               (actual time=0.040..76.783 rows=300000 loops=1)
   Sort Key: t1.a
   ->  Index Scan using t1_pkey on t1  (cost=0.29..3642.29 rows=100000 width=22)
                                       (actual time=0.014..18.876 rows=100000 loops=1)
   ->  Index Scan using t2_pkey on t2  (cost=0.42..7474.42 rows=200000 width=23)
                                       (actual time=0.025..35.920 rows=200000 loops=1)
 Total runtime: 85.019 ms
```

Cette optimisation est d'autant plus intéressante si l'on utilise une clause `LIMIT`.

Sans *MergeAppend*, avec `LIMIT 5` :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=9841.93..9841.94 rows=5 width=22)
        (actual time=119.946..119.946 rows=5 loops=1)
   ->  Sort  (cost=9841.93..10341.93 rows=200000 width=22)
             (actual time=119.945..119.945 rows=5 loops=1)
         Sort Key: t1.a
         Sort Method:  top-N heapsort  Memory: 25kB
         ->  Result  (cost=0.00..6520.00 rows=200000 width=22)
                     (actual time=0.008..75.482 rows=200000 loops=1)
               ->  Append  (cost=0.00..6520.00 rows=200000 width=22)
```

```
                          (actual time=0.008..53.644 rows=200000 loops=1)
                    ->  Seq Scan on t1
                              (cost=0.00..2110.00 rows=100000 width=23)
                              (actual time=0.006..18.819 rows=100000 loops=1)
                    ->  Seq Scan on t2
                              (cost=0.00..4410.00 rows=100000 width=22)
                              (actual time=1.550..22.119 rows=100000 loops=1)
 Total runtime: 119.976 ms
```

Avec *MergeAppend* :

```
 Limit  (cost=0.72..0.97 rows=5 width=23)
        (actual time=0.055..0.060 rows=5 loops=1)
   ->  Merge Append  (cost=0.72..14866.72 rows=300000 width=23)
                     (actual time=0.053..0.058 rows=5 loops=1)
         Sort Key: t1.a
         ->  Index Scan using t1_pkey on t1
                        (cost=0.29..3642.29 rows=100000 width=22)
                        (actual time=0.033..0.036 rows=3 loops=1)
         ->  Index Scan using t2_pkey on t2
                        (cost=0.42..7474.42 rows=200000 width=23) =
                        (actual time=0.019..0.021 rows=3 loops=1)
 Total runtime: 0.117 ms
```

On voit ici que chacun des parcours d'index renvoie 3 lignes, ce qui est suffisant pour renvoyer les 5 lignes ayant la plus faible valeur pour `a`.

1.5 AUTRES NŒUDS

- Nœud *HashSetOp Except*
  - `EXCEPT` et `EXCEPT ALL`
- Nœud *HashSetOp Intersect*
  - `INTERSECT` et `INTERSECT ALL`

La clause `UNION` permet de concaténer deux ensembles de données. Les clauses `EXCEPT` et `INTERSECT` permettent de supprimer une partie de deux ensembles de données.

Voici un exemple basé sur `EXCEPT` :

```
b1=# EXPLAIN SELECT oid FROM pg_proc
    EXCEPT  SELECT oid FROM pg_proc ;

                              QUERY PLAN
----------------------------------------------------------------------
 HashSetOp Except  (cost=0.00..219.39 rows=2342 width=4)
   ->  Append  (cost=0.00..207.68 rows=4684 width=4)
         ->  Subquery Scan on "*SELECT* 1"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
         ->  Subquery Scan on "*SELECT* 2"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
```

Et un exemple basé sur `INTERSECT` :

```
b1=# EXPLAIN   SELECT oid FROM pg_proc
    INTERSECT SELECT oid FROM pg_proc ;

                              QUERY PLAN
----------------------------------------------------------------------
 HashSetOp Intersect  (cost=0.00..219.39 rows=2342 width=4)
   ->  Append  (cost=0.00..207.68 rows=4684 width=4)
         ->  Subquery Scan on "*SELECT* 1"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
         ->  Subquery Scan on "*SELECT* 2"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
```

1.5.1 Divers

- Prend un ensemble de données en entrée
- Et renvoie un ensemble de données
- Exemples typiques
  - *Sort*
  - *Aggregate*
  - *Unique*
  - *Limit*
  - *InitPlan*, *SubPlan*

Tous les autres nœuds que nous allons voir prennent un seul ensemble de données en entrée et en renvoient un aussi. Ce sont des nœuds d'opérations simples comme le tri, l'agrégat, l'unicité, la limite, etc.

1.5.2 Tris

- Sort
- Incremental Sort

1.5.2.1 Sort

- Utilisé pour le `ORDER BY`
  - Mais aussi `DISTINCT`, `GROUP BY`, `UNION`
  - Les jointures de type *Merge Join*
- Gros délai de démarrage
- Trois types de tri
  - en mémoire, tri *quicksort*
  - en mémoire, tri *top-N heapsort* (si `LIMIT`)
  - sur disque

PostgreSQL peut faire un tri de trois façons.

Les deux premières sont manuelles. Il lit toutes les données nécessaires et les trie en mémoire. La quantité de mémoire utilisable dépend du paramètre `work_mem`. S'il n'a pas assez de mémoire, il utilisera un stockage sur disque. La rapidité du tri dépend principalement de la mémoire utilisable mais aussi de la puissance des processeurs. Le tri effectué est un tri *quicksort* sauf si une clause `LIMIT` existe, auquel cas, le tri sera un *top-N heapsort*. La troisième méthode est de passer par un index B-Tree. En effet, ce type d'index stocke les données de façon triée. Dans ce cas, PostgreSQL n'a pas besoin de mémoire.

Le choix entre ces trois méthodes dépend principalement de `work_mem`. En fait, le pseudo-code ci-dessous explique ce choix :

```
Si les données de tri tiennent dans work_mem
    Si une clause LIMIT est présente
        Tri top-N heapsort
    Sinon
        Tri quicksort
Sinon
    Tri sur disque
```

Voici quelques exemples :

- un tri externe :

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=150385.45..153040.45 rows=1062000 width=4)
       (actual time=807.603..941.357 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: external sort  Disk: 17608kB
   ->  Seq Scan on t2  (cost=0.00..15045.00 rows=1062000 width=4)
                 (actual time=0.050..143.918 rows=1000000 loops=1)
 Total runtime: 1021.725 ms
```

- un tri en mémoire :

```
b1=# SET work_mem TO '100MB';
SET
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;

                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=121342.45..123997.45 rows=1062000 width=4)
       (actual time=308.129..354.035 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: quicksort  Memory: 71452kB
   ->  Seq Scan on t2  (cost=0.00..15045.00 rows=1062000 width=4)
                 (actual time=0.088..142.787 rows=1000000 loops=1)
 Total runtime: 425.160 ms
```

- un tri en mémoire :

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id LIMIT 10000 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=85863.56..85888.56 rows=10000 width=4)
        (actual time=271.674..272.980 rows=10000 loops=1)
   ->  Sort  (cost=85863.56..88363.56 rows=1000000 width=4)
             (actual time=271.671..272.240 rows=10000 loops=1)
         Sort Key: id
         Sort Method: top-N heapsort  Memory: 1237kB
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                             (actual time=0.031..146.306 rows=1000000 loops=1)
 Total runtime: 273.665 ms
```

- un tri par un index :

```
b1=# CREATE INDEX ON t2(id);
CREATE INDEX
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id ;

                               QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t2_id_idx on t2
    (cost=0.00..30408.36 rows=1000000 width=4)
    (actual time=0.145..308.651 rows=1000000 loops=1)
 Total runtime: 355.175 ms
```

Les paramètres `enable_sort` et `enable_incremental_sort` permettent de défavoriser l'utilisation d'un tri, respectivement non incrémental ou incrémental. Dans ce cas, le planificateur tendra à préférer l'utilisation d'un index, qui retourne des données déjà triées.

Augmenter la valeur du paramètre `work_mem` aura l'effet inverse : favoriser un tri plutôt que l'utilisation d'un index.

1.5.2.2 Incremental Sort

- Utilisé lorsqu'un index existe sur les premières colonnes du tri
  - `ORDER BY`, `DISTINCT`, `GROUP BY`, `UNION`
  - Les jointures de type *Merge Join*
- Délai de démarrage réduit

Lorsqu'un tri est réalisé sur plusieurs colonnes, si aucun index ne permet de réaliser un tri classique, PostgreSQL peut essayer d'utiliser un index existant sur une des premières colonnes du tri pour réaliser un pré-tri.

Il pourra alors réaliser un tri sur les colonnes suivantes en tirant parti des groupes établis avec l'index utilisé. Dans ce cas, le délai de démarrage est réduit ce qui peut améliorer les performances lorsque la requête contient une clause `LIMIT`.

```
=# EXPLAIN (ANALYZE, COSTS OFF) SELECT * FROM clients ORDER BY id, ddn;

                              QUERY PLAN
-----------------------------------------------------------------------------
 Incremental Sort (actual time=0.170..0.171 rows=0 loops=1)
   Sort Key: id, ddn
   Presorted Key: id
   Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 25kB  Peak Memory:
↪  25kB
   ->  Index Scan using clients_pkey on clients (actual time=0.008..0.008 rows=0
↪  loops=1)
 Planning Time: 0.209 ms
 Execution Time: 0.214 ms
(7 rows)
```

Le `DISTINCT` est géré depuis la version 16.

1.5.3 Aggregate

- Agrégat complet
- Pour un seul résultat

Il existe plusieurs façons de réaliser un agrégat :

- l'agrégat standard ;
- l'agrégat par tri des données ;
- et l'agrégat par hachage.

ces deux derniers sont utilisés quand la clause `SELECT` contient des colonnes en plus de la fonction d'agrégat.

Par exemple, pour un seul résultat `COUNT(*)`, nous aurons ce plan d'exécution :

```
b1=# EXPLAIN SELECT count(*) FROM pg_proc ;

                           QUERY PLAN
-----------------------------------------------------------------
 Aggregate  (cost=86.28..86.29 rows=1 width=0)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=0)
```

Seul le parcours séquentiel est possible ici car `COUNT()` doit compter toutes les lignes.

Autre exemple avec une fonction d'agrégat `max` :

```
b1=# EXPLAIN SELECT max(proname) FROM pg_proc ;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Aggregate  (cost=92.13..92.14 rows=1 width=64)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
```

Il existe une autre façon de récupérer la valeur la plus petite ou la plus grande : passer par l'index. Ce sera très rapide car l'index est trié.

```
b1=# EXPLAIN SELECT max(oid) FROM pg_proc ;

                            QUERY PLAN
-------------------------------------------------------------------------
 Result  (cost=0.13..0.14 rows=1 width=0)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.13 rows=1 width=4)
           ->  Index Scan Backward using pg_proc_oid_index on pg_proc
               (cost=0.00..305.03 rows=2330 width=4)
                 Index Cond: (oid IS NOT NULL)
```

1.5.4 HashAggregate

- Hachage de chaque n-uplet de regroupement (`GROUP BY`)
- Accès direct à chaque n-uplet pour appliquer fonction d'agrégat
- Intéressant si l'ensemble des valeurs distinctes tient en mémoire, dangereux sinon

Voici un exemple de ce type de nœud :

```
b1=# EXPLAIN SELECT proname, count(*) FROM pg_proc GROUP BY proname ;

                            QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=92.13..111.24 rows=1911 width=64)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
```

Le hachage occupe de la place en mémoire, le plan n'est choisi que si PostgreSQL estime que si la table de hachage générée tient dans `work_mem`. **C'est le seul type de nœud qui peut dépasser work_mem :** la seule façon d'utiliser le *HashAggregate* est en mémoire, il est donc agrandi s'il est trop petit. Cependant, 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.

Le paramètre `enable_hashagg` permet d'activer et de désactiver l'utilisation de ce type de nœud.

1.5.5 GroupAggregate

- Reçoit des données déjà triées
- Parcours des données
  - regroupement du groupe précédent arrivé à une donnée différente

Voici un exemple de ce type de nœud :

```
b1=# EXPLAIN SELECT proname, count(*) FROM pg_proc GROUP BY proname ;

                              QUERY PLAN
-----------------------------------------------------------------------
 GroupAggregate  (cost=211.50..248.17 rows=1911 width=64)
   ->  Sort  (cost=211.50..217.35 rows=2342 width=64)
         Sort Key: proname
         ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
```

Un parcours d'index est possible pour remplacer le parcours séquentiel et le tri.

1.5.6 Unique

- Reçoit des données déjà triées
- Parcours des données
  - renvoi de la donnée précédente une fois arrivé à une donnée différente
- Résultat trié

Le nœud `Unique` permet de ne conserver que les lignes différentes. L'opération se réalise en triant les données, puis en parcourant le résultat trié. Là aussi, un index aide à accélérer ce type de nœud.

En voici un exemple :

```
b1=# EXPLAIN SELECT DISTINCT pronamespace FROM pg_proc ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Unique  (cost=211.57..223.28 rows=200 width=4)
   ->  Sort  (cost=211.57..217.43 rows=2343 width=4)
         Sort Key: pronamespace
         ->  Seq Scan on sample4  (cost=0.00..80.43 rows=2343 width=4)
```

1.5.7 Limit

- Limiter le nombre de résultats renvoyés
- Utilisation :
  - `LIMIT` et `OFFSET` dans une requête `SELECT`
  - fonctions `min()` et `max()` quand il n'y a pas de clause `WHERE` et qu'il y a un index
- Le nœud précédent sera de préférence un nœud dont le coût de démarrage est peu élevé (*Seq Scan*, *Nested Loop*)

Voici un exemple de l'utilisation d'un nœud *Limit* :

```
b1=# EXPLAIN SELECT 1 FROM pg_proc LIMIT 10 ;
                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=0.00..0.34 rows=10 width=0)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=0)
```

1.5.8 Memoize

- Apparu en version 14
- Cache de résultat
- Utilisable par la table interne des *Nested Loop*
- Utile si :
  - peu de valeurs distinctes dans l'ensemble interne
  - beaucoup de valeurs dans l'ensemble externe
  - peu de correspondance entre les deux ensembles
- Paramètres : `work_mem` `hash_mem_multiplier`

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* généré par une jointure classique ou `LATERAL`.

Le cas idéal pour cette optimisation concerne des jointures où de large portions des lignes de l'ensemble interne de la jointure n'ont pas de correspondance dans l'ensemble externe. Dans ce

genre de cas, un *Hash Join* serait moins efficace car il devrait calculer la clé de hachage de valeurs qui ne seront jamais utilisées ; et le *Merge Join* devrait ignorer un grand nombre de lignes dans son parcours de la table interne.

L'intérêt du cache de résultat augmente lorsqu'il y a peu de valeurs distinctes dans l'ensemble interne et que le nombre de valeurs dans l'ensemble externe est grand, ce qui provoque beaucoup de boucles. Ce nœud est donc très sensible aux statistiques sur le nombre de valeurs distinctes (`ndistinct`).

Cette fonctionnalité utilise une table de hashage pour stocker les résultats. Cette table est dimensionnée grâce aux paramètres `work_mem` et `hash_mem_multiplier` . Si le cache se remplit, les valeurs les plus anciennes sont exclues du cache.

Exemple :

```
CREATE TABLE t1(i int, j int);
CREATE TABLE t2(k int, l int);
INSERT INTO t2 SELECT x % 20,x FROM generate_series(1, 3000000) AS F(x);
INSERT INTO t1 SELECT x,x FROM generate_series(1, 300000) AS F(x);
CREATE INDEX ON t1(j);
ANALYZE t1,t2;

EXPLAIN (TIMING off, COSTS off, SUMMARy off, ANALYZE)
  SELECT * FROM t1 INNER JOIN t2 ON t1.j = t2.k;

                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop (actual rows=950 loops=1)
   ->  Seq Scan on t2 (actual rows=1000 loops=1)
   ->  Memoize (actual rows=1 loops=1000)
         Cache Key: t2.k
         Cache Mode: logical
         Hits: 980  Misses: 20  Evictions: 0  Overflows: 0  Memory Usage: 3kB
         ->  Index Scan using t1_j_idx on t1 (actual rows=1 loops=20)
               Index Cond: (j = t2.k)
```

On voit ici que le cache fait 3 ko. Il a permis de stocker 20 valeurs et de faire 980 accès au cache sur 1000 accès au total. Aucune valeur n'a été exclue du cache.

En désactivant ce nœud, on bascule sur un *Hash Join*:

```
SET enable_memoize TO off;
EXPLAIN (TIMING off, COSTS off, SUMMARY off, ANALYZE)
  SELECT * FROM t1 INNER JOIN t2 ON t1.j = t2.k;

                     QUERY PLAN
-----------------------------------------------------
 Hash Join (actual rows=950 loops=1)
   Hash Cond: (t2.k = t1.j)
   ->  Seq Scan on t2 (actual rows=1000 loops=1)
   ->  Hash (actual rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 48kB
         ->  Seq Scan on t1 (actual rows=1000 loops=1)
```

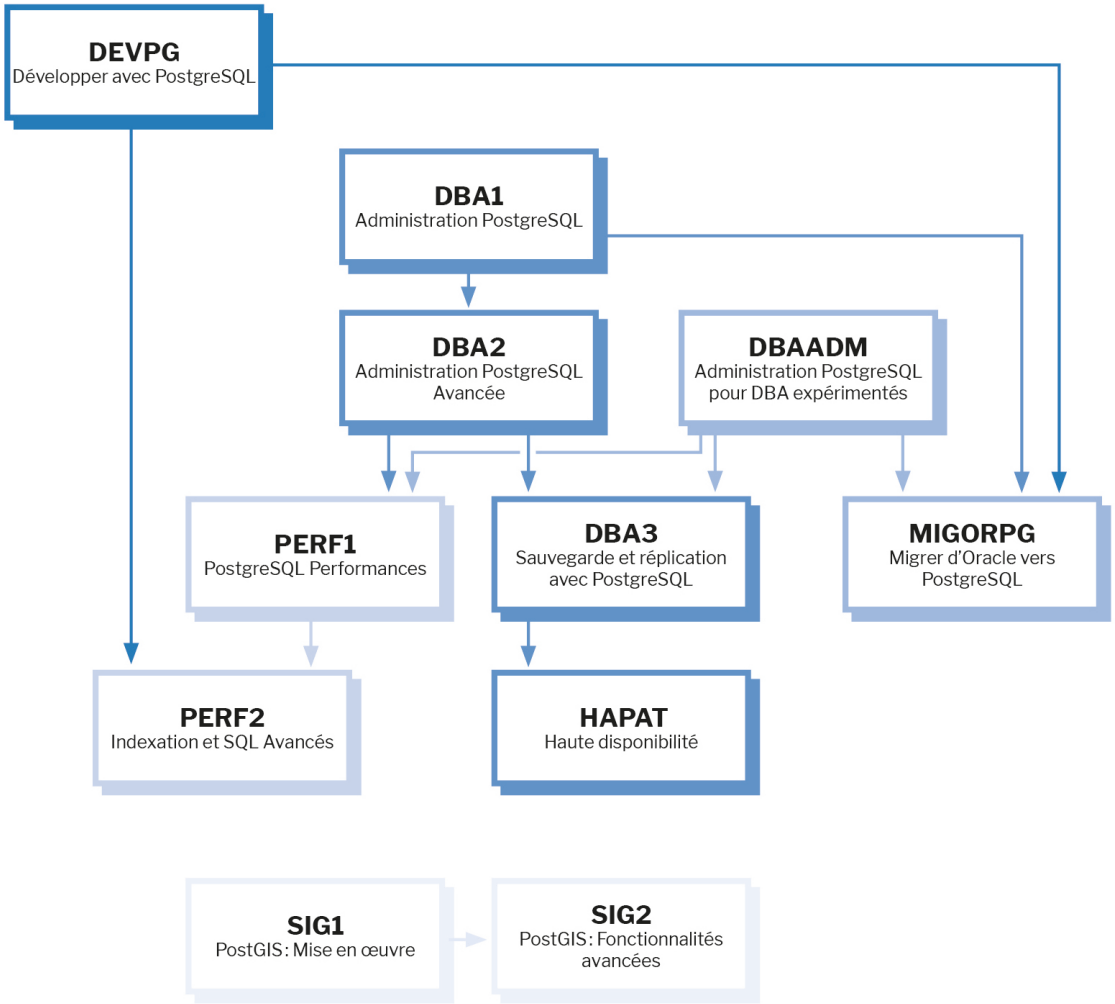
Dans ce petit exemple, le gain est nul, mais pour de grosses jointures, il peut être conséquent.

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.