



Comprendre EXPLAIN

Table des matières

Comprendre EXPLAIN.....	4
1 Licence des slides.....	5
2 Auteur.....	5
3 Comment puis-je comprendre EXPLAIN ?.....	6
4 Tout d'abord, nos objets.....	6
5 Essayons de lire notre table.....	7
6 Mettons à jour les statistiques.....	8
7 Que fait ANALYZE ?.....	8
8 Largeur de ligne.....	9
9 Nombre de lignes.....	10
10 Coût total.....	11
11 Mais que se passe-t-il vraiment ?.....	12
12 Et au niveau physique ?.....	13
13 Utilisons l'option BUFFERS.....	13
14 Et maintenant, avec un cache ?.....	14
15 Essayons avec un plus gros cache.....	15
16 Essayons une clause WHERE.....	16
17 Coût - un calcul plus compliqué.....	17
18 Nombre de lignes.....	18
19 Un index aiderait ici ?.....	18
20 Pourquoi ne pas utiliser l'index ?.....	19
21 Essayons de forcer l'utilisation de l'index.....	20
22 Essayons avec une autre requête.....	21
23 Compliquons le filtre.....	21
24 Essayons de filtrer sur c2 seulement.....	22
25 Ajoutons un index sur c2.....	22
26 Utilisons la classe d'opérateur.....	23
27 Essayons avec un index couvrant.....	24
28 Tous les nœuds (importants) de parcours.....	25
29 Essayons une clause ORDER BY.....	26
30 Est-on sûr de ne pas écrire sur disque ?.....	27
31 Donnons plus de mémoire pour le tri.....	27
32 Un index peut aussi aider.....	28
33 Revenons à c2.....	28
34 Et limitons le nombre de lignes.....	29
35 Ajoutons une autre table.....	30
36 ... et on joint les deux tables.....	30
37 Avoir un index est utile.....	31
38 Left join ?.....	32
39 Supprimons un index.....	32
40 Supprimons quelques lignes.....	33
41 Et avec une table vide ?.....	34
42 Toujours utilisé dans un CROSS JOIN.....	34
43 Tous les nœuds de jointure.....	35

44 Et pour les tables héritées ?.....	35
45 Lisons baz.....	36
46 Essayons avec un index.....	37
47 Essayons un agrégat.....	37
48 Essayons max().....	38
49 Max() est plus rapide avec un index.....	38
50 Fonctionne aussi avec les booléens en 9.2.....	39
51 Essayons GROUP BY.....	39
52 Avoir un gros work_mem aide.....	40
53 Ou un index.....	41
54 Tous les nœuds d'agrégat.....	41
55 Outils.....	42
56 pgAdmin.....	42
57 explain.depesz.com.....	43
58 Copie d'écran d'explain.depesz.com.....	43
59 pg_stat_plans.....	44
60 Exemple de pg_stat_plans.....	44
61 Conclusion.....	45
62 Question ?.....	45

Comprendre EXPLAIN



- Photographie récupérée sur <http://www.flickr.com/photos/crazygeorge/5578522008/>
- Prise par Andy Withers
- Licence CC BY-NC-ND 2.0

1 Licence des slides



- Creative Common BY-NC-SA
- Vous êtes libre
 - de partager
 - de modifier
- Sous les conditions suivantes
 - Attribution
 - Non commercial
 - Partage dans les mêmes conditions

2 Auteur



- Guillaume Lelarge
- Travail
 - Directeur technique de Dalibo
 - email: guillaume.lelarge@dalibo.com
- Communauté
 - pgAdmin, la documentation française, l'organisation de pgconf.eu, trésorier de PostgreSQL Europe, etc.
 - email: guillaume@lelarge.info
 - twitter: [@g_lelarge](https://twitter.com/g_lelarge)

3 Comment puis-je comprendre EXPLAIN ?



- EXPLAIN fait un peu peur
- Mais vous avez seulement besoin de quelques conseils pour l'appréhender
- C'est le thème de cette conférence
 - comprendre EXPLAIN
 - l'utiliser
 - l'aimer

EXPLAIN est une commande excellente qui vous donne un grand nombre d'informations. Cependant, il est souvent facile de se sentir perdu avec cet outil.

Et c'est bien dommage car il n'y a pas grand chose à savoir.

Cette conférence a donc pour but de vous montrer toutes les informations renvoyées par cette commande, de vous indiquer comment utiliser cette information, et comment corriger vos requêtes pour qu'elles fonctionnent plus rapidement.

4 Tout d'abord, nos objets



- Créons-les

```
CREATE TABLE foo (c1 integer, c2 text);
INSERT INTO foo
  SELECT i, md5(random()::text)
  FROM generate_series(1, 1000000) AS i;
```

Ces deux requêtes ajoutent une table et la peuplent avec des informations aléatoires. Cette table sera à la base de tous nos exemples.

5 Essayons de lire notre table

```
EXPLAIN SELECT * FROM foo;

          QUERY PLAN
-----
Seq Scan on foo (cost=0.00..18584.82
                rows=1025082 width=36)
```



- Coût
 - pour la première ligne..pour toutes les lignes
 - en unité de “coût de page”
- Rows
 - nombre de lignes
- Width
 - largeur moyenne d'une ligne (en octets)

Quand vous voulez lire une table, PostgreSQL dispose de plusieurs moyens pour le faire. Le plus simple est de lire séquentiellement le fichier correspondant, bloc par bloc.

C'est ce que montre ce plan d'exécution Vous avez aussi quelque informations statistiques données par le planificateur.

Le premier nombre correspond au coût pour obtenir la première ligne. Le deuxième nombre est le coût pour obtenir toutes les lignes, à partir d'un parcours séquentiel. Ces nombres ne sont pas en unité de temps. Ils sont dans une unité de coût de lecture d'une page. Vous pouvez habituellement comparer ces nombres au coût de lecture d'une page séquentielle (qui est de 1,0 par défaut).

Le troisième nombre est le nombre de lignes que le planificateur s'attend à obtenir lors d'un parcours séquentiel.

Le quatrième nombre est la largeur moyenne d'une ligne dans l'ensemble de lignes en résultat.

Ce que nous pouvons dire ici est que le nombre de lignes est un peu éloigné de la réalité. Pas énormément car l'autovacuum a probablement lancé un VACUUM lors des insertions.

6 Mettons à jour les statistiques



```
ANALYZE foo;
EXPLAIN SELECT * FROM foo;

          QUERY PLAN
-----
Seq Scan on foo (cost=0.00..18334.00
                rows=1000000 width=37)
```

- OK, bon nombre de lignes
- Mais comment sont calculées les statistiques ?

Donc nous devons mettre à jour les statistiques de la table et, pour cela, nous utilisons la commande ANALYZE. Comme je veux seulement les statistiques sur la table foo, je spécifie la table mais ce n'est pas une obligation.

Une fois que l'ANALYZE est terminé, si j'exécute la commande EXPLAIN une nouvelle fois, les informations statistiques du planificateur sont corrigées. Le nombre de lignes est correct, la largeur un peu supérieure, et le coût un peu moindre (le coût dépend du nombre de lignes, mais pas de la largeur de la ligne).

Tout ceci est très bien mais comment sont calculées toutes ces informations ? et pourquoi ANALYZE est-il si important ici ?

7 Que fait ANALYZE ?



- Lit des lignes aléatoires dans la table
 - 300*default_statistics_target
- Calcule des statistiques pour chaque colonne
- Valeurs les plus fréquentes et histogramme
 - dépend de default_statistics_target
 - peut être personnalisé pour chaque colonne
- Enregistre les statistiques dans pg_statistic
- Essayez la vue pg_stats
 - plus simple à lire
 - et plein d'informations très intéressantes

ANALYZE lit une partie de chaque table dans la base de données (sauf si vous indiquez une table spécifique auquel cas il ne lit qu'une partie de cette table). L'échantillon lu est pris au hasard. Sa taille dépend de la valeur du paramètre `default_statistics_target`. Il lira autant de lignes que 300 fois cette valeur. Par défaut, `default_statistics_target` vaut 100 (depuis la version 8.4). Donc, par défaut, il lira 30000 lignes au hasard dans la table.

Sur cet échantillon, il calcule quelques informations statistiques, comme le pourcentage de valeurs NULL, la largeur moyenne d'une ligne, le nombre de valeurs distinctes, etc. Il enregistre aussi les valeurs les plus fréquentes et leur fréquences. Le nombre de valeurs dépend de la valeur du paramètre `default_statistics_target`. Vous pouvez configurer cette valeur pour chaque ligne avec une requête `ALTER TABLE`.

Chaque information statistique est enregistrés dans le catalogue système `pg_statistic`. Ce catalogue est difficile à interpréter, donc il existe une vue bien plus simple, appelée `pg_stats`. Nous verrons certaines de ces colonnes lors du reste de la conférence.

Nous allons maintenant voir comment le planificateur calcule les informations statistiques données par la commande EXPLAIN.

8 Largeur de ligne



```
SELECT sum(avg_width) AS width
FROM pg_stats
WHERE tablename='foo';
```

```
width
-----
    37
```

- Taille moyenne (en octets) d'une ligne dans l'ensemble des données
- Plus c'est petit, mieux c'est

L'information de largeur est la largeur moyenne d'une ligne dans l'ensemble de lignes en résultat. Comme la commande ANALYZE récupère la largeur moyenne de chaque colonne de chaque table, nous pouvons obtenir cette information en additionnant la valeur de chaque colonne d'une table. Ceci donne la requête SQL dans ce slide.

9 Nombre de lignes

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

```
reltuples
```

```
-----  
1e+06
```



- Toutes les méta-données des relations apparaissent dans le catalogue pg_class
- Informations les plus intéressantes
 - reltuples
 - relpages
 - relallvisible

Le nombre de lignes est facile à obtenir. Le catalogue système pg_class est un catalogue de toutes les relations (tables, index, séquences, etc) dans la base de données. Ce catalogue contient les méta-données sur les relations. Il contient aussi quelques informations statistiques, comme le nombre estimé de lignes (colonne reltuples), le nombre estimé de blocs disques (relpages), et le nombre de blocs contenant seulement des blocs visibles par toutes les transactions en cours (relallvisible).

Là, nous sommes intéressés par reltuples.

10 Coût total

- Dans un parcours séquentiel, l'exécuteur a besoin :
 - de lire tous les blocs disques de la relation foo
 - de vérifier chaque ligne de chaque bloc pour filtrer les lignes "invisibles"



```
SELECT relpages*current_setting('seq_page_cost')::float4
+ reltuples*current_setting('cpu_tuple_cost')::float4
  AS total_cost
FROM pg_class
WHERE relname='foo';

total_cost
-----
      18334
```

Le coût total est un peu plus difficile à calculer. Quand PostgreSQL fait un parcours séquentiel, il fait en gros deux choses. Tout d'abord, il doit lire tous les blocs disques de la table. Pour chaque bloc, il a besoin de trouver et de vérifier chaque ligne. Donc il y a un coût pour la lecture de chaque bloc. Le fichier postgresql.conf indique le coût de lecture d'un bloc séquentiel avec le paramètre `seq_page_cost`. Le coût de lecture de tous les blocs d'une table est la multiplication de la valeur du paramètre `seq_page_cost` avec le nombre de blocs de la relation (`relpages` dans `pg_class`). Puis, pour vérifier chaque ligne, il y a aussi un coût associé dans le fichier `postgresql.conf` : `cpu_tuple_cost`. Tout ce qu'il doit faire revient à multiplier ce coût avec le nombre de lignes (`reltuples` dans `pg_class`). Enfin, il ajoute ces deux coûts, et c'est terminé.

Donc tout va bien et nous savons comment le planificateur calcule les nombres pour cette requête.

11 Mais que se passe-t-il vraiment ?

- Utilisons l'option ANALYZE



```
EXPLAIN (ANALYZE) SELECT * FROM foo;
QUERY PLAN
-----
Seq Scan on foo (cost=0.00..18334.00
                rows=1000000 width=37)
                (actual time=0.016..96.074
                 rows=1000000 loops=1)
Total runtime: 132.573 ms
```

- 3 nouvelles informations
 - durée réelle d'exécution en millisecondes
 - nombre réel de lignes
 - et nombre de boucles
- Attention, la requête est réellement exécutée !

Mais vous voulez certainement savoir ce qu'il se passe réellement quand vous exécutez la requête. Il est généralement préférable d'avoir la durée réelle d'exécution et le nombre de lignes exact.

Nous pouvons obtenir cela en ajoutant l'option ANALYZE à la commande EXPLAIN. Notez les parenthèses qui entourent l'option. C'est la nouvelle façon pour indiquer les options de la commande EXPLAIN. Cela existe depuis la version 9.0.

Ce résultat montre trois nouvelles informations provenant de l'exécution réelle de la requête. Tout d'abord, nous obtenons la durée réelle d'exécution pour récupérer la première ligne, puis celle pour les récupérer toutes. Nous avons aussi le vrai nombre de lignes du résultat et le nombre de boucles. Les durées d'exécution sont en millisecondes, donc le parcours séquentiel a pris ici 96 millisecondes pour récupérer un million de lignes. L'estimation du nombre de lignes était en fait exacte.

Faites attention en utilisant l'option ANALYZE avec toute commande DML. Un EXPLAIN ANALYZE UPDATE va réellement mettre à jour quelques lignes. Ce sera pareil pour DELETE et INSERT. Assurez-vous de les englober dans une transaction explicite, pour pouvoir annuler les modifications.

12 Et au niveau physique ?

- Tout d'abord, vidons tous les caches



```
$ /etc/init.d/postgresql stop
$ sync
$ echo 3 > /proc/sys/vm/drop_caches
$ /etc/init.d/postgresql start
```

Maintenant, voyons ce qui arrive au niveau physique.

13 Utilisons l'option BUFFERS



```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo;
```

QUERY PLAN


```
-----
Seq Scan on foo (cost=0.00..18334.00
                rows=1000000 width=37)
    (actual time=14.638..507.726
     rows=1000000 loops=1)
    Buffers: shared read=8334
Total runtime: 638.084 ms
```

- BUFFERS, nouvelle option de la version 9.0
- Nombre de blocs lus et écrits
 - dans le cache PostgreSQL (hit)
 - et en dehors (read)

Nous utilisons une autre option de la commande EXPLAIN. Cette option, appelée BUFFERS, est disponible depuis la version 9.0. Elle ajoute quelques nouvelles informations relatives aux blocs disques : nombre de blocs lus dans le cache de PostgreSQL, nombre de blocs lus par Linux, nombre de blocs written, nombre de blocs pour les objets temporaires, etc.

Sur cet exemple, "shared read" correspond au nombre de blocs lus en dehors du cache de PostgreSQL. C'est logique, nous venons de démarrer PostgreSQL, il a donc un cache vide. PostgreSQL a demandé l'accès à 8334 blocs pour lire la table complète.

14 Et maintenant, avec un cache ?



```

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo;

          QUERY PLAN
-----
Seq Scan on foo  (cost=0.00..18334.00
                 rows=1000000 width=37)
                 (actual time=0.046..115.908
                 rows=1000000 loops=1)
  Buffers: shared hit=32 read=8302
Total runtime: 171.803 ms

SELECT current_setting('shared_buffers') AS shared_buffers,
       pg_size_pretty(pg_table_size('foo')) AS table_size;

 shared_buffers | table_size
-----+-----
      32MB      |    65 MB

```

Exécutons une nouvelle fois la requête, cette fois avec un cache à chaud. Nous voyons maintenant que nous lisons 32 blocs dans le cache de PostgreSQL et les 8302 autres en dehors du cache de PostgreSQL. C'est énorme.

Une des raisons pour lesquelles il se comporte ainsi est que notre table est plus grosse que le cache de PostgreSQL. Notre table fait 65 Mo, plus du double de la taille du cache. Mais la vraie raison est que PostgreSQL utilise une optimisation appelée "ring buffer". Cela signifie qu'un seul SELECT ne peut pas utiliser le cache entièrement pour lui; La même chose arrive avec VACUUM. Il essaie de bien se comporter avec les autres sessions. Avec un cache plus gros, nous obtiendrons le même comportement mais une partie plus importante de la table sera placée dans le cache.

Notez cependant que la requête est trois fois plus rapide qu'avec un cache vide. Avoir un gros cache disque au niveau Linux aide beaucoup.

15 Essayons avec un plus gros cache

- Avec `shared_buffers` à 320MB



```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo;
      QUERY PLAN
-----
Seq Scan on foo (cost=0.00..18334.00
                rows=1000000 width=37)
    (actual time=15.009..532.372
     rows=1000000 loops=1)
    Buffers: shared read=8334
Total runtime: 646.289 ms

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo;
      QUERY PLAN
-----
Seq Scan on foo (cost=0.00..18334.00
                rows=1000000 width=37)
    (actual time=0.014..100.762
     rows=1000000 loops=1)
    Buffers: shared hit=8334
Total runtime: 147.999 ms
```

Plaçons 320 Mo dans le cache de PostgreSQL. Puis vidons le cache Linux et redémarrons PostgreSQL.

À la première exécution, tout est lu du disque et la requête prend 646 ms à s'exécuter. À la deuxième exécution, tout se trouve dans le cache disque de PostgreSQL et la requête ne prend plus que 147 ms.

16 Essayons une clause WHERE

```
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
```

```
QUERY PLAN
```

```
-----  
Seq Scan on foo (cost=0.00..20834.00  
                rows=999579 width=37)  
  Filter: (c1 > 500)
```



- Nouvelle ligne : Filter: (c1 > 500)
- Le coût est plus important
 - moins de lignes
 - mais plus de travail à cause du filtre
- Le nombre de lignes n'est pas exact mais suffisamment précis

Cette fois, nous utilisons une clause `WHERE`. Nous n'avons pas d'index sur cette colonne (et en fait, nous n'avons pas du tout d'index pour cette table), donc le seul moyen de répondre à cette requête est d'utiliser un parcours séquentiel. Lors de la lecture de chaque bloc et de la vérification de la visibilité de chaque ligne, PostgreSQL vérifie aussi que la valeur de la colonne `c1` est supérieure à 500. Dans le cas contraire, il ignore la ligne. C'est la signification de la nouvelle ligne "Filter".

Même si nous finissons avec moins de lignes, PostgreSQL a plus de travail à faire car il doit vérifier la valeur de la colonne `c1` sur chaque ligne de la table. Donc le coût estimé sera plus important que celui de la requête sans filtre.

Comme vous pouvez le voir, le nombre estimé de lignes n'est pas exact. Cependant, il est suffisamment bon.

17 Coût - un calcul plus compliqué

- L'exécuteur
 - lit tous les blocs de la relation foo
 - filtre les lignes d'après la clause WHERE
 - vérifie la visibilité des lignes restantes




```
SELECT
  relpages*current_setting('seq_page_cost')::float4
+ reltuples*current_setting('cpu_tuple_cost')::float4
+ reltuples*current_setting('cpu_operator_cost')::float4
  AS total_cost
FROM pg_class
WHERE relname='foo';

total_cost
-----
      20834
```

Il est un peu plus compliqué de calculer le coût. Quand PostgreSQL exécute cette requête, il lit tous les blocs de la table. Donc nous avons toujours le coût de lecture avec cette formule : nombre de blocs multiplié par le coût de l'accès séquentiel à un bloc. Il doit aussi vérifier la visibilité de chaque ligne, donc l'ancienne formule est toujours valable : nombre de lignes multiplié par le `cpu_tuple_cost`. La seule différence est dans le filter : nous devons utiliser l'opérateur sur la colonne pour chaque ligne. Nous avons aussi un paramètre qui indique le coût d'utilisation d'un opérateur pour une ligne (paramètre nommé `cpu_operator_cost`). Nous utiliserons cet opérateur sur chaque ligne, donc nous avons besoin de cette nouvelle formule : nombre de lignes multiplié par `cpu_operator_cost`.

Ajoutez chaque coût spécifique, et on obtient le coût total de la requête.

18 Nombre de lignes



```

SELECT histogram_bounds
FROM pg_stats
WHERE tablename='foo' AND attname='c1';

          histogram_bounds
-----
{57,10590,19725,30449,39956,50167,59505,...,999931}

SELECT round(
(
(10590.0-500)/(10590-57)
+
(current_setting('default_statistics_target')::int4 - 1)
)
* 10000.0
) AS rows;

999579


```

Le calcul de l'estimation du nombre de lignes est encore plus complexe.

PostgreSQL utilise un histogramme des valeurs pour obtenir une estimation du nombre de lignes. Cette histogramme est divisé en 100 parties contenant le même nombre de lignes. 100 est la valeur par défaut du paramètre `default_statistics_target`. La table contient 1 million de lignes, donc chaque partie contient 10000 lignes.

La première partie contient des valeurs entre 57 et 10590. Comme nous cherchons des valeurs après 500, cela signifie que nous avons seulement un pourcentage de cette partie, et toutes les autres parties séparément. La formule sur le slide reflète cela, et donne le nombre de lignes estimées par le planificateur.

19 Un index aiderait ici ?



```

CREATE INDEX ON foo(c1);
EXPLAIN SELECT * FROM foo WHERE c1 > 500;

          QUERY PLAN
-----
Seq Scan on foo  (cost=0.00..20834.00
                 rows=999529 width=37)
  Filter: (c1 > 500)

```

- Non...

Habituellement, tout le monde pense qu'un index aiderait à exécuter rapidement une requête si nous avons une clause `WHERE`.

Dans notre cas, c'est faux. Rappelez-vous, la table contient 1 million de lignes et nous ignorons seulement 500 lignes. Il est bien préférable de filtrer avec le CPU que d'utiliser un index.

20 Pourquoi ne pas utiliser l'index ?



```
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;

          QUERY PLAN
-----
Seq Scan on foo (cost=0.00..20834.00
                rows=999493 width=37)
    (actual time=0.125..147.613
     rows=999500 loops=1)
  Filter: (c1 > 500)
  Rows Removed by Filter: 500
  Total runtime: 184.901 ms
```

- Nouvelle ligne : Rows Removed by Filter: 500
 - affichée seulement avec l'option ANALYZE
 - à partir de la version 9.2
- Il doit lire 99,95% de la table !

Si vous n'avez aucune connaissance sur cette table, la version 9.2 vous aidera plus. Essayons un `EXPLAIN ANALYZE` sur la table `foo` en 9.2. Notez la nouvelle ligne "Rows Removed by Filter". 500 lignes sont supprimées par le filtre et nous finissons avec 999500 dans l'ensemble de résultats. Nous devons lire 99,95% de la table ! Un index ne sera jamais utilisé dans ce cas.

21 Essayons de forcer l'utilisation de l'index



```
SET enable_seqscan TO off;  
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;  
SET enable_seqscan TO on;
```

QUERY PLAN

```
-----  
Index Scan using foo_c1_idx on foo  
  (cost=0.00..36801.25 rows=999480 width=37)  
   (actual time=0.099..186.982 rows=999500 loops=1)  
    Index Cond: (c1 > 500)  
Total runtime: 221.354 ms  
(3 rows)
```

- OK, mais lent
- Le planificateur avait raison
 - l'index n'a aucun intérêt ici

Pour s'assurer que cela coûte plus, essayons de forcer le planificateur à utiliser un parcours d'index. Pour cela, nous avons toute sorte de paramètres "enable_*". Celui dont nous avons besoin est le paramètre `enable_seqscan` que nous configurons à `off`. Comprenez bien que cela ne désactive pas complètement les parcours séquentiels (si vous n'avez pas d'index, le moteur ne peut pas faire autrement que d'utiliser un parcours séquentiel), mais cela rendra beaucoup plus difficile l'utilisation d'un parcours séquentiel (cela se fait en ajoutant un coût énorme à ce type de nœud).

Donc quand nous configurons `enable_seqscan` à `off`, il utilisera un index. Mais vous pouvez voir que cela prend un peu plus de temps d'utiliser l'index (221 ms) que de lire la table complète et de la filtrer (184 ms).

22 Essayons avec une autre requête



```
EXPLAIN SELECT * FROM foo WHERE c1 < 500;
```

```
QUERY PLAN
```

```
-----
Index Scan using foo_c1_idx on foo
(cost=0.00..24.59 rows=471 width=37)
Index Cond: (c1 < 500)
```

- Nouvelle ligne : Index Cond: (c1 < 500)
- Il lit toujours la table pour obtenir les informations de visibilité

Maintenant, changeons le filtre. Cette fois, nous avons un parcours d'index. Nous avons ce nœud car nous lisons un petit nombre de lignes (seulement 499 sur 1 million).

23 Compliquons le filtre



```
EXPLAIN SELECT * FROM foo
WHERE c1 < 500 AND c2 LIKE 'abcd%';
```

```
QUERY PLAN
```

```
-----
Index Scan using foo_c1_idx on foo
(cost=0.00..25.77 rows=71 width=37)
Index Cond: (c1 < 500)
Filter: (c2 ~ 'abcd% '::text)
```

- “Index Cond”, et “Filter”

Maintenant essayons un autre filtre. Nous obtenons un parcours d'index sur la colonne c1 et un filtre sur les lignes lues dans la table pour obtenir au final que les lignes contenant “abcd” au début de la colonne c2.

24 Essayons de filtrer sur c2 seulement



```
EXPLAIN (ANALYZE)
SELECT * FROM foo WHERE c2 LIKE 'abcd%';

          QUERY PLAN
-----
Seq Scan on foo (cost=0.00..20834.00
                rows=100 width=37)
    (actual time=21.435..134.153
     rows=15 loops=1)
    Filter: (c2 ~ 'abcd% '::text)
    Rows Removed by Filter: 999985
    Total runtime: 134.198 ms
```

- 999985 lignes supprimées par le filtre
- Seulement 15 lignes en résultat
 - donc un index sera forcément intéressant...

Ici, nous filtrons seulement sur c2. Nous n'avons pas d'index sur c2, donc PostgreSQL fait un parcours séquentiel. Ce que nous notons ici que le parcours séquentiel lit un million de lignes (999985+15), et n'en conserve que 15. Un index sur c2 aiderait vraiment.

25 Ajoutons un index sur c2



```
CREATE INDEX ON foo(c2);
EXPLAIN (ANALYZE) SELECT * FROM foo
WHERE c2 LIKE 'abcd%';

          QUERY PLAN
-----
Seq Scan on foo (cost=0.00..20834.00
                rows=100 width=37)
    (actual time=22.189..143.467
     rows=15 loops=1)
    Filter: (c2 ~ 'abcd% '::text)
    Rows Removed by Filter: 999985
    Total runtime: 143.512 ms
```

- Cela ne fonctionne pas ! à cause de l'encodage
 - nous avons besoin d'une classe d'opérateur

Donc ajoutons un index sur c2. Une fois que nous exécutons EXPLAIN, nous pouvons voir que nous avons toujours un parcours séquentiel. L'index ne semble pas intéressant pour répondre à la requête. C'est étonnant.

En fait, le problème vient du fait que l'index ne correspond pas à la demande. La colonne c2 contient des valeurs dans l'encodage UTF8. Quand vous utilisez un autre encodage que C, vous devez indiquer la classe d'opérateur (varchar_pattern_ops, text_pattern_ops, etc.) à la création de l'index.

26 Utilisons la classe d'opérateur



```
CREATE INDEX ON foo(c2 text_pattern_ops);
EXPLAIN SELECT * FROM foo WHERE c2 LIKE 'abcd%';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on foo  (cost=7.29..57.91
                          rows=100 width=37)
  Filter: (c2 ~~ 'abcd%'::text)
    -> Bitmap Index Scan on foo_c2_idx1
        (cost=0.00..7.26 rows=13 width=0)
        Index Cond: ((c2 ~>~ 'abcd'::text)
                     AND (c2 ~<~ 'abce'::text))
```

- Nouveau type de nœud : Bitmap Index Scan

Une fois qu'il existe un index doté de la bonne classe d'opérateur, le planificateur peut utiliser l'index pour accélérer l'obtention des résultats de la requête.

Notez aussi le nouveau type de nœud. Le "Bitmap Index Scan" est une autre façon d'utiliser un index. Et il en existe un autre en 9.2.

27 Essayons avec un index couvrant



```
EXPLAIN SELECT c1 FROM foo WHERE c1 < 500;
```

```
QUERY PLAN
```

```
-----  
Index Only Scan using foo_c1_idx on foo  
(cost=0.00..26.40 rows=517 width=4)  
Index Cond: (c1 < 500)
```

- À partir de la 9.2
- Ne lit pas la table pour les informations de visibilité

Dans certaines requêtes, toutes les informations redondantes enregistrées dans l'index sont les seules informations nécessaires pour répondre à une requête. La requête dans cette slide en est un exemple parfait. Dans l'index `foo_c1_idx`, nous avons la valeur de la colonne `c1`, donc elle peut être utilisée pour faire le filtre et pour donner la valeur dans l'ensemble de résultats. Nous utilisons habituellement le terme `index couvrant` dans de tels cas.

Le nœud "Index Only Scan" est utilisé quand un index couvrant est disponible. C'est plus rapide qu'un parcours d'index standard car il n'est pas nécessaire de lire la table pour savoir si une ligne est visible ou non.

C'est une fonctionnalité de la version 9.2.

28 Tous les nœuds (importants) de parcours




- Parcours séquentiel
 - lit toute la table séquentiellement
- Parcours d'index
 - lit l'index pour filtrer la clause WHERE
 - et la table pour filtrer les lignes "invisibles"
- Parcours d'index bitmap
 - pareil
 - mais lit complètement l'index, puis la table
 - plus rapide avec un grand nombre de lignes
- Parcours d'index couvrant (Index Only Scan)
 - pour les index couvrants

Nous avons vu tous les nœuds majeurs de parcours : parcours séquentiel, parcours d'index, parcours de bitmap d'index, parcours d'index couvrants. Ils sont tous intéressants et utiles pour répondre aux différents types de requêtes.

Il existe d'autres types de nœuds de parcours, comme celui sur les fonctions et sur le VALUES, mais nous n'aurons pas le temps d'aller dans ce type de détails lors de cette conférence.

29 Essayons une clause ORDER BY



```

DROP INDEX foo_c1_idx;
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;

          QUERY PLAN
-----
Sort
(cost=172682.84..175182.84 rows=1000000 width=37)
(actual time=612.793..729.135 rows=1000000 loops=1)
Sort Key: c1
Sort Method: external sort  Disk: 45952kB
-> Seq Scan on foo  (cost=0.00..18334.00
                    rows=1000000 width=37)
                    (actual time=0.017..111.068
                    rows=1000000 loops=1)
Total runtime: 772.685 ms

```


Il existe plusieurs façons de répondre à un “ORDER BY” (ou, plus généralement, de faire un tri).

Le moyen le plus simple (ou un qui ne nécessite pas d'autres objets dans la table) est un tri effectué par le processeur lors de l'exécution de la requête. Ceci prend du temps CPU et de la mémoire. Dans cet exemple, cela nécessite tellement de mémoire que PostgreSQL se résout à faire un tri sur disque. Il utilise 45 Mo. Comme il écrit beaucoup de données sur le disque, cela prend beaucoup de temps. Le temps passé à faire le parcours séquentiel (111 ms) est bien moindre que le temps passé à faire le tri (618 ms, résultat de la soustraction de 729 ms avec 111 ms).

C'est aussi la première fois que nous avons un plan à deux nœuds. Le premier nœud exécuté est le parcours séquentiel, et le dernier est le tri. Les premiers nœuds exécutés sont ceux qui sont le plus à droite, et les derniers sont ceux les plus à gauche.

Notez aussi que nous avons deux informations supplémentaires : la clé du tri et la méthode du tri.

30 Est-on sûr de ne pas écrire sur disque ?



```


EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo ORDER BY c1;

          QUERY PLAN
-----
Sort
 (cost=172682.84..175182.84 rows=1000000 width=37)
 (actual time=632.796..756.958 rows=1000000 loops=1)
 Sort Key: c1
 Sort Method: external sort  Disk: 45952kB
 Buffers: shared hit=3910 read=4424,
          temp read=5744 written=5744
-> Seq Scan on foo  (cost=0.00..18334.00
                    rows=1000000 width=37)
                    (actual time=0.134..110.617
                    rows=1000000 loops=1)
                    Buffers: shared hit=3910 read=4424
 Total runtime: 811.308 ms

```

Pour s'assurer que PostgreSQL écrit bien sur le disque les données temporaires pour le tri, nous pouvons utiliser l'option BUFFERS de la commande EXPLAIN. La ligne intéressante ici est "temp read=5744 written=5744". PostgreSQL écrit 5744 blocs de données temporaires, puis les lit. 5744 blocs de 8 Ko font bien les 45 Mo que nous attendions.

31 Donnons plus de mémoire pour le tri



```


SET work_mem TO '200MB';
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;

          QUERY PLAN
-----
Sort
 (cost=117991.84..120491.84 rows=1000000 width=37)
 (actual time=552.401..598.984 rows=1000000 loops=1)
 Sort Key: c1
 Sort Method: quicksort  Memory: 102702kB
-> Seq Scan on foo  (cost=0.00..18334.00
                    rows=1000000 width=37)
                    (actual time=0.014..102.907
                    rows=1000000 loops=1)
 Total runtime: 637.525 ms

```

PostgreSQL peut aussi faire le tri en mémoire mais nous lui devons lui donner suffisamment de mémoire à utiliser. Cela se fait via le paramètre `work_mem`. Par défaut, ce paramètre est à 1 Mo. Ici, nous le configurons à 200 Mo, et nous ré-exécutons la requête. PostgreSQL bascule sur un algorithme quicksort en mémoire et c'est plus rapide (et c'est même encore plus rapide en 9.2, grâce à Peter Geoghegan et son utilisation d'un quicksort optimisé).

32 Un index peut aussi aider



```


CREATE INDEX ON foo(c1);
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;

          QUERY PLAN
-----
Index Scan using foo_c1_idx on foo
 (cost=0.00..34318.35 rows=1000000 width=37)
 (actual time=0.030..179.810 rows=1000000 loops=1)
Total runtime: 214.442 ms

```

Un index contient les données triées, donc un autre moyen de faire un “ORDER BY” est de lire un index. Dans cet exemple, nous voyons que PostgreSQL préfère lire l'index que de trier les données en mémoire. Et le résultat est encore plus rapide : trois fois plus rapide.

33 Revenons à c2



```

DROP INDEX foo_c2_idx1;
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM foo WHERE c2 LIKE 'ab%';


          QUERY PLAN
-----
Seq Scan on foo
 (cost=0.00..20834.00 rows=100 width=37)
 (actual time=0.066..134.149 rows=3978 loops=1)
  Filter: (c2 ~ 'ab%':text)
  Rows Removed by Filter: 996022
  Buffers: shared hit=8334
Total runtime: 134.367 ms

```

Supprimons l'index de la colonne c2, et utilisons EXPLAIN ANALYZE sur une nouvelle requête.

Dans cet exemple, nous voulons toutes les lignes qui ont la chaîne 'ab' au début de la valeur de la colonne c2. Comme il n'existe pas d'index, PostgreSQL fait un parcours séquentiel qui rejette 996002 lignes.

34 Et limitons le nombre de lignes



```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM foo WHERE c2 LIKE 'ab%' LIMIT 10;

          QUERY PLAN
-----
Limit
(cost=0.00..2083.40 rows=10 width=37)
(actual time=0.065..0.626 rows=10 loops=1)
  Buffers: shared hit=19
    -> Seq Scan on foo
        (cost=0.00..20834.00 rows=100 width=37)
        (actual time=0.063..0.623 rows=10 loops=1)
          Filter: (c2 ~ 'ab% '::text)
          Rows Removed by Filter: 2174
          Buffers: shared hit=19
Total runtime: 0.652 ms
```

Avec une clause LIMIT, nous obtenons le même parcours séquentiel mais nous remarquons qu'il rejette seulement 2174 lignes. Quand un nœud Limit est présent, le parcours est prévenu qu'il n'a pas besoin de récupérer toutes les lignes mais seulement les X premières (dans cet exemple, les 10 premières). Du coup, PostgreSQL doit lire 2184 lignes avant de récupérer les dix lignes dont la colonne c2 commence avec 'ab'.

Cela fonctionne aussi avec un parcours d'index.

35 Ajoutons une autre table

- Créons-la



```
CREATE TABLE bar (c1 integer, c2 boolean);
INSERT INTO bar
SELECT i, i%2=1
FROM generate_series(1, 500000) AS i;
```

- Et on met à jour les statistiques

```
ANALYZE bar;
```

Nous avons besoin de créer une autre table pour tester les jointures. Nous insérons aussi quelques lignes avec des valeurs aléatoires pour peupler la table. Enfin, nous lançons ANALYZE pour disposer de bonnes statistiques.

36 ... et on joint les deux tables



```
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;


QUERY PLAN
-----
Hash Join  (cost=15417.00..63831.00 rows=500000 width=42)
           (actual time=133.820..766.437 rows=500000 ...)
  Hash Cond: (foo.c1 = bar.c1)
    -> Seq Scan on foo
        (cost=0.00..18334.00 rows=1000000 width=37)
        (actual time=0.009..107.492 rows=1000000 loops=1)
    -> Hash  (cost=7213.00..7213.00 rows=500000 width=5)
        (actual time=133.166..133.166 rows=500000 loops=1)
        Buckets: 4096 Batches: 32 Memory Usage: 576kB
        -> Seq Scan on bar
            (cost=0.00..7213.00 rows=500000 width=5)
            (actual time=0.011..49.898 rows=500000 loops=1)
Total runtime: 782.830 ms
```

Ceci est une jointure simple. Il existe différents moyens de faire des jointures. Là, nous avons un HashJoin. L'idée derrière la jointure par hachage est de hacher une table et, pour chaque ligne de l'autre table, hacher la ligne et la comparer aux hachages de l'autre table. Avec une telle jointure, vous pouvez seulement avoir un opérateur d'égalité pour la jointure.

Sur cet exemple, PostgreSQL fait un parcours séquentiel de la table bar, et calcule le hachage de toutes ses lignes. Puis il fait un parcours séquentiel de foo et, pour chaque ligne, il calcule le hachage de la ligne et le compare à celui de la table bar hachée. Si une correspondance est faite, la ligne est placée dans l'ensemble de résultat. Dans le cas contraire, la ligne est ignorée.

Cela fonctionne bien avec suffisamment de mémoire pour conserver la table hachée en mémoire.

37 Avoir un index est utile




```
CREATE INDEX ON bar(c1);
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;

-----
QUERY PLAN
-----
Merge Join (cost=0.82..39993.03 rows=500000 width=42)
  (actual time=0.039..393.172 rows=500000 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
    -> Index Scan using foo_c1_idx on foo
        (cost=0.00..34318.35 rows=1000000 width=37)
        (actual time=0.018..103.613 rows=500001 loops=1)
    -> Index Scan using bar_c1_idx on bar
        (cost=0.00..15212.80 rows=500000 width=5)
        (actual time=0.013..101.863 rows=500000 loops=1)
  Total runtime: 411.022 ms
```

Si vous ajoutez un index sur la colonne c1 de la table bar, PostgreSQL verra deux index. Il existe un moyen pour obtenir des données triées rapidement et un bon moyen pour joindre deux tables est d'assembler deux ensembles de données triés. Avec ce nouvel index, PostgreSQL peut rapidement lire les données triées pour les deux tables. Du coup, il préfère une jointure par fusion.

38 Left join ?



```


EXPLAIN (ANALYZE)
SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;

          QUERY PLAN
-----
Merge Left Join
(cost=0.82..58281.15 rows=1000000 width=42)
(actual time=0.036..540.328 rows=1000000 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
    -> Index Scan using foo_c1_idx on foo
        (cost=0.00..34318.35 rows=1000000 width=37)
        (actual time=0.016..191.645 rows=1000000 loops=1)
    -> Index Scan using bar_c1_idx on bar
        (cost=0.00..15212.80 rows=500000 width=5)
        (actual time=0.012..93.921 rows=500000 loops=1)
  Total runtime: 573.966 ms
(5 rows)

```

Cet exemple remplace le simple JOIN avec un LEFT JOIN. Le résultat est un nœud Merge Left Join à la place d'un Merge Join.

39 Supprimons un index



```

DELETE FROM bar WHERE c1>500; DROP INDEX bar_c1_idx;
ANALYZE bar;
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;

          QUERY PLAN
-----
Merge Join (cost=2240.53..2265.26 rows=500 width=42)
(actual time=40.433..40.740 rows=500 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
    -> Index Scan using foo_c1_idx on foo
        (cost=0.00..34318.35 rows=1000000 width=37)
        (actual time=0.016..0.130 rows=501 loops=1)
    -> Sort (cost=2240.41..2241.66 rows=500 width=5)
        (actual time=40.405..40.432 rows=500 loops=1)
        Sort Key: bar.c1
        Sort Method: quicksort  Memory: 48kB
        -> Seq Scan on bar
            (cost=0.00..2218.00 rows=500 width=5)
            (actual time=0.020..40.297 rows=500 loops=1)
  Total runtime: 40.809 ms

```

Pour prouver que le nœud Merge Join nécessite des données pré-triées, nous allons

supprimer l'index sur bar et supprimer la plupart des valeurs de bar. Comme bar contient peu de lignes, le tri sera rapide. Dans cet exemple, PostgreSQL choisit de faire un parcours séquentiel sur bar, puis de trier les données avec un quicksort en mémoire. Avec le parcours d'index sur la table foo, il a tout ce qui lui faut pour obtenir un Merge Join rapide.

40 Supprimons quelques lignes




```
DELETE FROM foo WHERE c1>1000;
ANALYZE;
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;

          QUERY PLAN
-----
Nested Loop  (cost=10.79..7413.13 rows=500 width=42)
  (actual time=0.034..3.287 rows=500 loops=1)
  -> Seq Scan on bar  (cost=0.00..8.00 rows=500 width=5)
      (actual time=0.006..0.131 rows=500 loops=1)
  -> Bitmap Heap Scan on foo
      (cost=10.79..14.80 rows=1 width=37)
      (actual time=0.005..0.005 rows=1 loops=500)
      Recheck Cond: (c1 = bar.c1)
      -> Bitmap Index Scan on foo_c1_idx
          (cost=0.00..10.79 rows=1 width=0)
          (actual time=0.003..0.003 rows=1 loops=500)
          Index Cond: (c1 = bar.c1)
Total runtime: 3.381 ms
```

Avec plus de lignes supprimées, PostgreSQL basculera sur un Nested Loop.

41 Et avec une table vide ?



```


TRUNCATE bar;
ANALYZE;
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1>bar.c1;

          QUERY PLAN
-----
Nested Loop
(cost=0.00..21405802.11 rows=776666667 width=42)
(actual time=0.004..0.004 rows=0 loops=1)
-> Seq Scan on bar
   (cost=0.00..33.30 rows=2330 width=5)
   (actual time=0.002..0.002 rows=0 loops=1)
-> Index Scan using foo_c1_idx on foo
   (cost=0.00..5853.70 rows=333333 width=37)
   (never executed)
   Index Cond: (c1 > bar.c1)
Total runtime: 0.049 ms

```

Sans ligne, PostgreSQL pense toujours qu'il y a quelques lignes (bien peu, mais quelques unes quand même). Donc il choisit un Nested Loop. Cette fois, le point intéressant est que le parcours d'index n'est jamais exécuté car il n'y a réellement aucune ligne dans la table bar.

42 Toujours utilisé dans un CROSS JOIN



```

EXPLAIN SELECT * FROM foo CROSS JOIN bar ;

          QUERY PLAN
-----
Nested Loop
(cost=0.00..1641020027.00 rows=100000000000 width=42)
-> Seq Scan on foo
   (cost=0.00..18334.00 rows=1000000 width=37)
-> Materialize
   (cost=0.00..2334.00 rows=100000 width=5)
   -> Seq Scan on bar
      (cost=0.00..1443.00 rows=100000 width=5)

```

Nested Loop est aussi le type de nœud utilisé pour un CROSS JOIN. En fait, CROSS JOIN ne peut rien utiliser d'autres.

43 Tous les nœuds de jointure



- Nested Loop
 - pour les petites tables
 - rapide à démarrer, lent à finir
- Merge Join
 - tri, puis fusion
 - lent à démarrer sans index
 - plus rapide sur les gros volumes de données
- Hash Join
 - seulement sur une jointure d'égalité
 - très rapide avec suffisamment de mémoire
 - mais lent à démarrer

Maintenant, nous avons vu tous les nœuds de jointure. PostgreSQL n'en connaît pas plus. Les trois disponibles sont suffisantes pour réaliser chaque type de jointure.

44 Et pour les tables héritées ?

- Notre table :

```
CREATE TABLE baz(c1 timestamp, c2 text);
```

- La première table héritée et ses données :



```
CREATE TABLE baz_2012(
  CHECK(c1 BETWEEN '2012-01-01'
        AND '2012-12-31')
) INHERITS(baz);

INSERT INTO baz_2012
SELECT now()-i*interval '1 day', 'line '||i
FROM generate_series(1, 200) AS i;
```

- Et deux autres tables héritées
 - baz_2011, baz_2010

Les tables héritées sont principalement utilisées avec l'héritage mais peuvent aussi être utilisées dans certains cas particuliers.

Ici, nous créons la table maître, et trois tables héritées. Nous les peuplons avec 200 lignes chacune.

45 Lisons baz



```
EXPLAIN (ANALYZE) SELECT * FROM baz WHERE c1
BETWEEN '2012-06-08' AND '2012-06-10';


-----
QUERY PLAN
-----
Result (cost=0.00..21.00 rows=11 width=18)
-> Append (cost=0.00..21.00 rows=11 width=18)
    -> Seq Scan on baz
        (cost=0.00..0.00 rows=1 width=40)
        Filter: ((c1 >= '2012-06-08...')
                AND (c1 <= '2012-06-10...'))
    -> Seq Scan on baz_2012 baz
        (cost=0.00..21.00 rows=10 width=16)
        Filter: ((c1 >= '2012-06-08...')
                AND (c1 <= '2012-06-10...'))
```

- Une seule partition parcourue
- et la table principale

Avec une lecture des données correspondant à quelques jours de 2012, nous voyons que PostgreSQL parcourt seulement la table maître et la table correspondant à l'année 2012. Il ne parcourt pas les autres tables. Les contraintes disponibles sur chaque table garantissent à PostgreSQL que seul baz_2012 contient des données sur l'année 2012.

Ceci fonctionnera seulement si le paramètre `constraint_exclusion` est activé. Donc ne le désactivez pas.

46 Essayons avec un index



```


CREATE INDEX ON baz_2012(c1);
INSERT INTO baz_2012 <a few times>
EXPLAIN (ANALYZE) SELECT * FROM baz WHERE c1='2012-06-10';

-----
QUERY PLAN
-----
Result  (cost=0.00..8.27 rows=2 width=28)
        (actual time=0.014..0.014 rows=0 loops=1)
-> Append (cost=0.00..8.27 rows=2 width=28)
    (actual time=0.013..0.013 rows=0 loops=1)
    -> Seq Scan on baz
        (cost=0.00..0.00 rows=1 width=40)
        (actual time=0.002..0.002 rows=0 loops=1)
        Filter: (c1 = '2012-06-10...')
    -> Index Scan using baz_2012_c1_idx
        on baz_2012 baz
        (cost=0.00..8.27 rows=1 width=16)
        (actual time=0.011..0.011 rows=0 loops=1)
        Index Cond: (c1 = '2012-06-10...')
Total runtime: 0.052 ms

```

Cette optimisation fonctionne aussi avec un index. Elle peut utiliser certains index spécifiques sur chaque table.

47 Essayons un agrégat



```

EXPLAIN SELECT count(*) FROM foo;

-----
QUERY PLAN
-----
Aggregate
(cost=20834.00..20834.01 rows=1 width=0)
-> Seq Scan on foo
    (cost=0.00..18334.00 rows=1000000 width=0)

```

C'est la requête la plus simple que vous pouvez avoir avec un agrégat, et probablement la plus utilisée.

Il existe un seul moyen de faire ceci : un parcours séquentiel pour compter toutes les lignes. Et c'est ce que fait PostgreSQL avec le nœud Aggregate.

48 Essayons max()



```

DROP INDEX foo_c2_idx;
EXPLAIN (ANALYZE) SELECT max(c2) FROM foo;
      QUERY PLAN
-----
Aggregate
(cost=20834.00..20834.01 rows=1 width=33)
(actual time=257.813..257.814 rows=1 loops=1)
-> Seq Scan on foo
   (cost=0.00..18334.00 rows=1000000 width=33)
   (actual time=0.011..89.625 rows=1000000 loops=1)
Total runtime: 257.856 ms

```

- Très moche

max() est une autre fonction d'agrégat vraiment utilisée. Il existe deux façons d'obtenir la valeur maximale d'une colonne.

La plus simple (mais aussi la plus lente) est de lire toutes les lignes et de conserver la plus grande valeur des lignes visibles.

49 Max() est plus rapide avec un index



```

CREATE INDEX ON foo(c2);
EXPLAIN (ANALYZE) SELECT max(c2) FROM foo;
      QUERY PLAN
-----
Result
(cost=0.08..0.09 rows=1 width=0)
(actual time=0.183..0.184 rows=1 loops=1)
  InitPlan 1 (returns $0)
    -> Limit (cost=0.00..0.08 rows=1 width=33)
        (actual time=0.177..0.178 rows=1 loops=1)
          -> Index Only Scan Backward using foo_c2_idx
              on foo
              (cost=0.00..79677.29 rows=1000000 width=33)
              (actual time=0.176..0.176 rows=1 loops=1)
              Index Cond: (c2 IS NOT NULL)
              Heap Fetches: 1
Total runtime: 0.223 ms

```


- Really better

L'autre façon est d'utiliser un index. Comme un index dispose de valeurs déjà triées, vous avez seulement besoin de récupérer la valeur la plus grande. Cela sous-entend

un parcours d'index (inverse si vous voulez la plus grosse valeur) et de limiter le parcours à la lecture de la première ligne.

En 9.2, il est aussi possible de passer par un Index Only Scan.

50 Fonctionne aussi avec les booléens en 9.2



```


CREATE INDEX ON foo(c2);
EXPLAIN SELECT bool_and(c2) FROM bar;

              QUERY PLAN
-----
Result  (cost=0.04..0.05 rows=1 width=0)
  InitPlan 1 (returns $0)
    -> Limit  (cost=0.00..0.04 rows=1 width=1)
        -> Index Only Scan using bar_c2_idx on bar
            (cost=0.00..21447.34 rows=500000 width=1)
            Index Cond: (c2 IS NOT NULL)

```

Ce type de micro-optimisation fonctionne aussi avec les booléens en version 9.2. Jamais vu utilisé en production, mais cela fonctionne :)

51 Essayons GROUP BY



```

DROP INDEX foo_c2_idx;
EXPLAIN (ANALYZE)
  SELECT c2, count(*) FROM foo GROUP BY c2;


              QUERY PLAN
-----
GroupAggregate
(cost=172682.84..190161.00 rows=997816 width=33)
(actual time=4444.907..5674.155 rows=999763 loops=1)
 -> Sort
    (cost=172682.84..175182.84 rows=1000000 width=33)
    (actual time=4444.893..5368.587 rows=1000000
     loops=1)
      Sort Key: c2
      Sort Method: external merge  Disk: 42080kB
    -> Seq Scan on foo
        (cost=0.00..18334.00 rows=1000000 width=33)
        (actual time=0.014..147.936 rows=1000000
         loops=1)
Total runtime: 5715.402 ms

```

Faire un décompte (count(*)) pour chaque valeur distincte d'une colonne peut se faire de deux façons. La plus simple revient à trier les données, et à les grouper suivant leur valeurs.

Dans cet exemple, le planificateur n'a pas d'index pour trier les données, donc il retourne à un parcours séquentiel suivi d'un tri manuel. Le tri est trop gros pour rester en mémoire, donc il utilise un tri externe. Après le tri, il a simplement besoin de grouper les valeurs non distinctes, et calculer le décompte. Ce qui prend le temps ici est le tri (5,3 secondes pour une requête de 5,6 secondes). Avoir un work_mem plus gros peut aider, avoir un index aide à coup sûr.

52 Avoir un gros work_mem aide




```
SET work_mem TO '200MB';
EXPLAIN (ANALYZE)
  SELECT c2, count(*) FROM foo GROUP BY c2;

              QUERY PLAN
-----
HashAggregate
(cost=23334.00..33312.16 rows=997816 width=33)
(actual time=581.653..976.936 rows=999763 loops=1)
-> Seq Scan on foo
   (cost=0.00..18334.00 rows=1000000 width=33)
   (actual time=0.021..96.977 rows=1000000 loops=1)
Total runtime: 1019.810 ms
```

Avec un work_mem plus gros, PostgreSQL bascule sur un HashAggregate car le hachage peut rester en mémoire.

53 Ou un index



```

RESET work_mem;
CREATE INDEX ON foo(c2);a
EXPLAIN (ANALYZE)
  SELECT c2, count(*) FROM foo GROUP BY c2;
          
```

QUERY PLAN

```

GroupAggregate
(cost=0.00..92155.45 rows=997816 width=33)
(actual time=0.108..1439.023 rows=999763 loops=1)
->  Index Only Scan using foo_c2_idx on foo
    (cost=0.00..77177.29 rows=1000000 width=33)
    (actual time=0.095..1043.090 rows=1000000 loops=1)
        Heap Fetches: 1000000
Total runtime: 1475.775 ms
          
```

Une index ne peut pas aider dans le cas du hachage. Donc, avec un index et un petit work_mem, nous retournons au GroupAggregate. Mais cette fois, il utilise un parcours d'index à la place d'un parcours séquentiel suivi d'un tri.

54 Tous les nœuds d'agrégat



- Aggregate
- GroupAggregate
- HashAggregate

Ce sont tous les nœuds d'agrégat disponibles pour le planificateur.

55 Outils



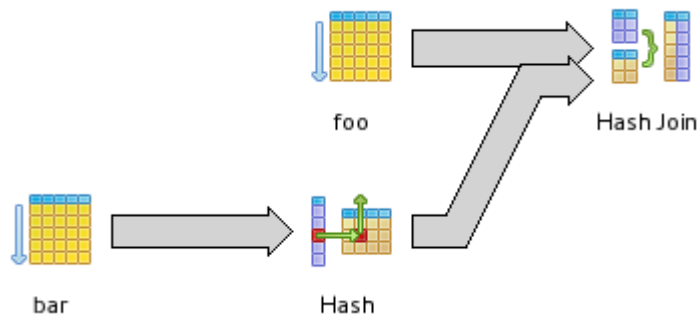
- Pas beaucoup d'outils
- mais certains sont bien utiles
- pgAdmin
- explain.depesz.com
- pg_stat_plans

Il n'existe malheureusement pas beaucoup d'outils pour nous aider avec les plans d'exécution. Nous avons pgAdmin et son affichage graphique du plan d'exécution, explain.depesz.com et ses tableaux colorisés, et enfin pg_stat_plans.

56 pgAdmin



- Bon outil de requêtage
- EXPLAIN graphique



- Permet de trouver immédiatement le premier nœud exécuté et la façon dont ils sont connectés
- Une flèche plus grosse indique plus de données

pgAdmin est un bon outil d'administration. Son outil de requêtage a beaucoup de fonctionnalités, l'une d'entre elles étant l'affichage graphique d'un plan d'exécution. C'est, à ma connaissance, le seul outil à le faire.

Cela aide vraiment à trouver rapidement le premier nœud exécuté et à visualiser comment tous les nœuds d'un plan sont connectés les uns aux autres. La taille des

flèches a une signification : plus vous obtenez de onnées, plus elles sont grosses.It really helps to understand which node is the first to be executed, and how they are all connected. The size of the arrows has a meaning: the more data you got, the bigger they are.

57 explain.depesz.com



- Encore mieux que l'EXPLAIN graphique de pgAdmin
- Vous pouvez l'installer localement
 - un module Perl
 - un serveur web mojolicious
- Vous donne la durée d'exécution exclusive de chaque nœud
 - probablement la meilleure information donnée par cet outil
- Des couleurs pour trouver rapidement le problème dans le plan

explain.depesz.com est un site web où vous pouvez poster le plan d'exécution d'une requête provenant directement d'un EXPLAIN. Les informations les plus intéressantes sont la durée exclusive pour chaque nœud. Vous pouvez évidemment le calculer vous-même mais c'est plus simple quand c'est fait pour vous. De plus, des couleurs permettent de trouver rapidement les points faibles du plan d'exécution.

Si vous ne voulez pas poster vos plans d'exécution pour éviter de laisser filtrer des informations importantes, vous pouvez aussi l'installer sur un de vos serveurs. Vous avez un mo

58 C



exclusive	inclusive	rows x	rows	loops	node
1428.146	9459.431	↑ 1.0	20	1	→ GroupAggregate (cost=119575.55..125576.11 rows=20 width=23) (actual time=6912.523..9459.431 rows=20 loops=1) Buffers: shared hit=30 read=12306, temp read=6600 written=6598
4709.032	8031.285	↑ 1.0	600036	1	→ Sort (cost=119575.55..121075.64 rows=600036 width=23) (actual time=6817.015..8031.285 rows=600036 loops=1) Sort Key: c.name Sort Method: external merge Disk: 20160kB Buffers: shared hit=30 read=12306, temp read=6274 written=6272
2043.571	3322.253	↑ 1.0	600036	1	→ Hash Join (cost=9416.95..37376.03 rows=600036 width=23) (actual time=407.974..3322.253 rows=600036 loops=1) Hash Cond: (b.item_id = i.id) Buffers: shared hit=30 read=12306, temp read=994 written=992
870.898	870.898	↑ 1.0	600036	1	→ Seq Scan on bids b (cost=0.00..11001.36 rows=600036 width=8) (actual time=0.009..870.898 rows=600036 loops=1) Buffers: shared hit=2 read=4999
94.573	407.784	↑ 1.0	50000	1	→ Hash (cost=8522.95..8522.95 rows=50000 width=19) (actual time=407.784..407.784 rows=50000 loops=1) Buckets: 4096 Batches: 2 Memory Usage: 989kB Buffers: shared hit=28 read=7307, temp written=111

59 pg_stat_plans



- Nouvelle extension de Peter Geoghegan
- Encore meilleure que pg_stat_statements
 - vous pouvez maintenant avoir le plan de la requête

pg_stat_plans est une extension assez récente, vu qu'elle est sortie il y a seulement quelques mois. Elle est écrite par Peter Geoghegan, ancien 2ndQuadrant, nouvel employé d'Heroku.

Cette extension va un peu plus loin que pg_stat_statements car elle vous donne le plan d'exécution de vos dernières requêtes exécutées.

60 Exemple de pg_stat_plans



```
SELECT pg_stat_plans_explain(planid, userid, dbid),
       planid, last_startup_cost, last_total_cost
FROM pg_stat_plans
WHERE planid = 3002758564;
-[ RECORD 1 ]-----+-----
pg_stat_plans_explain | GroupAggregate
                      | (cost=0.00..92155.45 rows=997816
                      | width=33)
                      |   -> Index Only Scan
                      |       using foo_c2_idx
                      |       on foo
                      |       (cost=0.00..77177.29
                      |       rows=1000000
                      |       width=33)
planid                 | 3002758564
last_startup_cost     | 0
last_total_cost       | 92155.450671524
```

Sur cet exemple, nous avons seulement récupéré le plan, son coût de démarrage et son coût total. Vous pouvez avoir plus d'informations comme `had_our_search_path` et `query_valid`.

Vous devriez lire la documentation sur github:

https://github.com/2ndQuadrant/pg_stat_plans, c'est une extension très intéressante.

61 Conclusion



- Le planificateur de PostgreSQL est réellement impressionnant
 - même s'il a toujours des défauts
- EXPLAIN vous donne un grand nombre de pistes pour comprendre
 - ce que l'exécuteur fait
 - et pourquoi il le fait
- Avec une meilleure compréhension, vous pourrez plus facilement corriger vos requêtes

62 Question ?



- S'il ne me reste pas de temps pour prendre des questions :
 - je suis ici toute la journée
 - je suis joignable sur guillaume@lelarge.info