Zeste de savoir

PostgreSQL : estimation rapide d'un COUNT(*)

30 juillet 2022

Table des matières

Introduction

Ce billet présente une technique peu orthodoxe pour estimer (très) rapidement le nombre de lignes (*rows*) dans une table, même avec un filtre `WHERE` sur des valeurs non indexées. Une explication technique du fonctionnement de l'astuce et de ses limites est proposée en guise de conclusion.

1. Compter avec `COUNT(*)`

Voici la configuration des tests qui suivront:

— PostgreSQL 14.3, *tunings* par défaut de la distribution;
— Fedora Linux 36 (noyau `5.18.13-200.fc36.x86_64`);
— disque SSD en NVMe;
— CPU `Intel(R) Core(TM) i7-10710U` en mode *performance*.

Les specs sont plutôt bonnes et les valeurs nominales peuvent finalement être acceptables dans l'absolu, même si nous allons voir comment augmenter drastiquement la vitesse des comptages.

Définissons une table de la façon suivante: 20 millions de lignes, chacune contenant, en plus de son ID, un nombre aléatoire entre 0 et 100 (colonne `g`). Cette colonne `g` n'est pas indexée.

```
create table grades (id serial, g int);

insert into grades (g)
    select
      random()*100
    from
      generate_series(1, 20000000); -- prend quelques secondes
```

Compter le tout avec un `count(*)` prend un peu plus d'une demi-seconde (**557 ms**) sur ma machine (il faut regarder la ligne `Execution Time: 556.957 ms`):

```
db=> explain analyze select count(*) from grades;

    QUERY PLAN
------------------------------------------------------------------------
    ------------------------------------------------------------------------
    --------------
 Finalize Aggregate  (cost=193663.38..193663.39 rows=1 width=8)
    (actual time=555.043..556.939 rows=1 loops=1)
   ->  Gather  (cost=193663.17..193663.38 rows=2 width=8) (actual
    time=554.956..556.934 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=192663.17..192663.18 rows=1
    width=8) (actual time=553.603..553.604 rows=1 loops=3)
               ->  Parallel Seq Scan on grades
    (cost=0.00..171829.73 rows=8333373 width=0) (actual
    time=0.053..346.266 rows=6666667 loops=3)

 Planning Time: 0.033 ms
 Execution Time: 556.957 ms
(8 rows)

(Résultat : 20 000 000)
```

Malgré l'index de la colonne `id` et l'absence de filtre `WHERE` qui pourrait nous laisser penser qu'il suffirait de regarder les métadonnées de l'arbre-B pour tout compter, il y a tout de même un parcours séquentiel (*Seq Scan*) sur l'intégralité des 20M lignes pour les compter. Fort heureusement, PostgreSQL parallélise les accès avec deux workers en l'occurrence (`Workers Launched: 2`).

Voyons voir le résultat d'une requête similaire, cette fois-ci en ajoutant des filtres pour connaître le nombre de valeurs `g` comprises dans le quart supérieur (entre 75 et 100):

```
select count(*) from grades where g between 75 and 100;
```

```
db=> explain analyze select count(*) from grades where g between
    75 and 100;

    QUERY PLAN
------------------------------------------------------------------------
    ------------------------------------------------------------------------
    --------------
 Finalize Aggregate  (cost=219845.64..219845.65 rows=1 width=8)
    (actual time=428.994..430.907 rows=1 loops=1)
```

```
   ->  Gather  (cost=219845.43..219845.64 rows=2 width=8) (actual
    time=428.900..430.901 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=218845.43..218845.44 rows=1
    width=8) (actual time=427.465..427.466 rows=1 loops=3)
               ->  Parallel Seq Scan on grades
    (cost=0.00..213496.60 rows=2139531 width=0) (actual
    time=0.129..370.624 rows=1699696 loops=3)
                     Filter: ((g >= 75) AND (g <= 100))
                     Rows Removed by Filter: 4966971

 Planning Time: 0.044 ms
 Execution Time: 430.924 ms
(10 rows)

(Résultat : 5 099 087)
```

Cette fois PostgreSQL a vérifié les clauses `((g >= 75) AND (g <= 100))` en chemin, mais le plan d'exécution reste globalement le même. Le plan d'exécution n'est donc pas surprenant, et est donc similaire au premier plan vu précédemment, avec des temps comparables: `Execution Time: 430.924 ms`.

Tout cela reste quand même assez long, si nous voulions obtenir la distribution en quatre parties, il faudrait presque une bonne seconde d'attente (**985 ms**) avec la requête suivante[1]:

```sql
select
    count(*) filter (where g between 0  and 25) as fst,
    count(*) filter (where g between 26  and 50) as snd,
    count(*) filter (where g between 51  and 75) as trd,
    count(*) filter (where g between 76  and 100) as fth
from grades ;
```

```
db=> explain analyze select
    count(*) filter (where g between 0  and 25) as fst,
    count(*) filter (where g between 26  and 50) as snd,
    count(*) filter (where g between 51  and 75) as trd,
    count(*) filter (where g between 76  and 100) as fth
 from grades ;

[...]

 Planning Time: 0.042 ms
 Execution Time: 984.510 ms
(8 rows)
```

Résultat (à retenir pour la suite):

fst (0–25)	snd (26–50)	trd (51–75)	fth (76–100)
5 100 815	5 001 831	4 998 524	4 898 830

2. Le Planning Time

Avez-vous remarqué le `Planning Time` ridiculement petit par rapport à l'`Execution Time` qui reste souvent sous la milliseconde? Le `EXPLAIN ANALYZE` calcule le *planning time*, mais exécute aussi effectivement la requête pour avoir l'*execution time*. Pour se passer de la seconde analyse, un simple `EXPLAIN` (sans `ANALYZE`) le permettra et est presque instantané. Étant donné que la requête ne sera pas exécutée effectivement, nous pouvons nous permettre de requêter directement les résultats avec `select *` plutôt que `select count(*)`:

```
db=> explain select * from grades where g between 76 and 100;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on grades  (cost=0.00..388496.00 rows=4911233 width=8)
   Filter: ((g >= 76) AND (g <= 100))
(2 rows)
```

Le résultat est instantané et quelque chose saute aux yeux: ce `rows=4911233`. C'est drôle, nous sommes très proches de la valeur exacte, à savoir 4 898 830 comme vu précédemment. Notre erreur est de 0,25 %. Nous avons économisé plus de 500 ms d'exécution au prix d'une erreur de 0,25 %. Ce n'est pas si mal, c'est même parfaitement acceptable dans bon nombre de situations.

PostgreSQL permet de retourner le résultat sous forme JSON, ce qui permet même d'extraire cette valeur de façon programmatique:

```
db=> explain (format json) select * from grades where g between 75
    and 100;
                  QUERY PLAN
----------------------------------------------
 [                                           +
   {                                         +
     "Plan": {                               +
       "Node Type": "Seq Scan",              +
       "Parallel Aware": false,              +
       "Relation Name": "grades",            +
       "Alias": "grades",                    +
       "Startup Cost": 0.00,                 +
       "Total Cost": 388496.00,              +
       "Plan Rows": 4911233,                 +
```

```
        "Plan Width": 8,                          +
        "Filter": "((g >= 76) AND (g <= 100))"+
      }                                           +
    }                                             +
 ]
(1 row)
```

Il n'y a plus qu'à parser cela et extraire le champ `[0]["Plan"]["Plan Rows"]` pour implémenter l'estimation rapide.

Magique, n'est-ce pas?

3. Explication et limites

En se penchant en détail sur le fonctionnement interne de PostgreSQL, quelques éléments d'explication nous apparaissent.

PostgreSQL utilise le mécanisme du MVCC (Multiversion Concurrency Control) pour écrire ou lire les données de façon concurrente en gérant les conflits en se passant d'un *lock* global (quand c'est possible) qui amoindrirait les performances sur les workloads typiques. Cela signifie qu'il est possible de lire les données d'une table sans être bloqué par une écriture concurrente, et *vice versa*, le tout en garantissant une cohérence *in fine*.

Lors de la lecture des données (ce qui constitue une transaction), il faut donc vérifier que chacune des lignes lues appartient bien au *snapshot* de la transaction en cours (si les lignes sont bien versionnées). Une ligne qui «appartiendrait» à une autre transaction concurrente ne doit ainsi pas être comptabilisée afin de retourner un résultat cohérent.

Cette vérification a un coût non négligeable quand le nombre de lignes à vérifier est important, c'est ce que nous payons ici. Le wiki du PostgreSQL nous dit[2]:

> The basic SQL standard query to count the rows in a table is :
>
> ```sql
> SELECT count(*) FROM table_name;
> ```
>
> This can be rather slow because PostgreSQL has to check visibility for all rows, due to the MVCC model.
>
> *https://wiki.postgresql.org/wiki/Count_estimate*

Pour avoir une estimation rapide et se passer des vérifications qui nous ralentissent, nous accédons alors aux données statistiques internes de PostgreSQL qui servent à élaborer les plans retournés pas un `EXPLAIN`. Ce sont ces statistiques qui lui permettent d'organiser des plans d'exécution efficaces, elles sont mises à jour au fur et à mesure de l'évolution des données, mais pas forcément en temps réel (PostgreSQL s'en occupe lui-même).

Limites

Mais une question se pose alors. Comme nous l'avons vu, la lenteur s'explique par la vérification de chaque ligne dans le cadre du MVCC. Tout semble indiquer que la méthode alternative outrepasse bien cette vérification en se basant uniquement sur les statistiques internes. Mais cela ne risquerait-il pas alors de dégrader la précision de l'estimation si jamais des phases d'écritures concurrentes massives, y compris les `UPDATE`, avaient lieu?

Pour rappel, PostgreSQL ne met pas à jour les lignes *in situ*, il en créées de nouvelles et marquent les anciennes comme obsolètes, lesquelles peuvent tout de même rester visibles pour les transactions antérieures qui sont encore en cours d'exécution. Quand les anciennes lignes ne sont plus nécessaires (si plus aucune transaction n'en n'a besoin), elles peuvent être supprimées physiquement et les statistiques internes se mettent à jour. Mais cette opération coûteuse n'est pas systématiquement instantanée.

Illustrons cela en mettant à jour l'intégralité des lignes de la table en incrémentant chaque `g` de 1:

```
update grades set g = g+1; -- met quelques bonnes secondes
```

Si nous relançons notre estimation, le résultat est surprenant:

```
db=> explain select * from grades where g between 76 and 100;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on grades  (cost=0.00..776989.12 rows=9822419 width=8)
   Filter: ((g >= 76) AND (g <= 100))
(2 rows)
```

Il y a 9 822 419 lignes contre 4 911 233 avant, c'est presque le double! C'est cohérent avec ce qui vient d'être vu: sans vérifier si chaque ligne est encore à jour dans le contexte d'une transaction donnée, on compte tout sans distinction, à savoir l'à jour et l'obsolète ce qui fait à peu près le double.

Nous l'avons évoqué, PostgreSQL sait faire le ménage et évincer les lignes obsolètes qui n'auront plus besoin d'être là pour d'autres transactions ultérieures. Il s'agit d'un *vacuum*[3], habituellement lancé périodiquement, mais que l'on peut aussi lancer manuellement pour ne pas attendre:

```
vacuum;
```

Voyons le résultat après avoir fait passé l'aspirateur:

```
db=> explain select * from grades where g between 76 and 100;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on grades  (cost=0.00..475472.26 rows=5214400 width=8)
   Filter: ((g >= 76) AND (g <= 100))
(2 rows)
```

Nous avons 5 214 400 lignes, voilà qui est plus raisonnable.

Sur un workload qui met à jour en masse et régulièrement les données déjà insérées, cette technique est donc à disqualifier. La commande `VACUUM` peut être assez lente sur des gros jeux de données. Il n'est donc pas si intéressant de systématiquement y avoir recours en vue d'obtenir une estimation des lignes juste après.

1. La variante présentée semble plus rapide que `select sum(case when g between 0 and 25 then 1 else 0 end) as fst, ...`
2. *Count estimate*
3. *VACUUM—garbage-collect and optionally analyze a database*