

Améliorer les temps de réponse des requêtes PostgreSQL avec EXPLAIN

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 2 Juillet 2009

<http://www.bortzmeyer.org/explain-postgresql.html>

Un des pièges de SQL est que c'est un langage de haut niveau. Le rapport entre la commande qu'on tape et le travail que doit faire la machine est beaucoup moins direct et beaucoup plus dur à saisir qu'avec l'assembleur. Il est donc fréquent qu'une requête SQL qui n'aie pas l'air bien méchante prenne des heures à s'exécuter. Heureusement, PostgreSQL dispose d'une excellent commande `EXPLAIN` qui explique, avant même qu'on exécute la commande, ce qu'elle va faire et ce qui va prendre du temps.

Un excellent exposé d'introduction est "*Explaining EXPLAIN*" (http://wiki.postgresql.org/wiki/Image:Explaining_EXPLAIN.pdf) mais, uniquement avec les transparents, c'est un peu court. Voyons des exemples d'utilisation d'`EXPLAIN` sur la base de données de StackOverflow (<http://www.bortzmeyer.org/stackoverflow-to-postgresql.html>).

Je commence par une requête triviale, `SELECT * FROM Votes` et je la fais précéder de `EXPLAIN` pour demander à PostgreSQL de m'expliquer :

```
so=> EXPLAIN SELECT * FROM Votes;
          QUERY PLAN
-----
Seq Scan on votes  (cost=0.00..36658.37 rows=2379537 width=16)
(1 row)
```

On apprend que PostgreSQL va faire une recherche séquentielle ("*Seq Scan*"), ce qui est normal, puisqu'il faudra tout trouver, que 2379537 tuples (lignes de la table SQL) sont à examiner. Et que le coût sera entre... zéro et 36658,37. Pourquoi zéro ? Parce que PostgreSQL affiche le coût de récupérer le **premier** tuple (important si on traite les données au fur et à mesure, ce qui est souvent le cas en OLTP) et celui de récupérer tous les tuples. (Utiliser `LIMIT` ne change rien, PostgreSQL calcule les coûts comme si `LIMIT` n'était pas là.)

Et le 36658,37 ? C'est 36658,37 quoi ? Pommes, bananes, euros, bons points ? Ce coût est exprimé dans une unité arbitraire qui dépend notamment de la puissance de la machine. Avant de lire toute la documentation sur le calcul des coûts, faisons une autre requête :

```
so=> EXPLAIN SELECT count(*) FROM Votes;
                QUERY PLAN
-----
Aggregate  (cost=42607.21..42607.22 rows=1 width=0)
-> Seq Scan on votes  (cost=0.00..36658.37 rows=2379537 width=0)
(2 rows)
```

Cette fois, le coût du premier tuple est le même que le coût total, ce qui est logique, puisque les fonctions d'agrégation comme `count ()` nécessitent de récupérer toutes les données.

Les coûts qu'affiche `EXPLAIN` ne sont que des coûts **estimés**, sans faire effectivement tourner la requête (donc, `EXPLAIN` est en général très rapide). Si on veut les temps réels, on peut ajouter le mot-clé `ANALYZE` :

```
so=> EXPLAIN ANALYZE SELECT * FROM Votes;
                QUERY PLAN
-----
Seq Scan on votes  (cost=0.00..36658.37 rows=2379537 width=16) (actual time=0.042..3036.302 rows=2379537 loops=1)
Total runtime: 5973.786 ms
(2 rows)
```

Cette fois, on a le temps effectif, trois secondes. On peut en déduire que, sur cette machine, l'unité de coût de PostgreSQL vaut environ 0,08 milli-secondes ($3036,302 / 36658,37$). Sur un serveur identique, mais avec des disques de modèle différent et un autre système de fichiers, on arrive à une valeur quasiment identique. Mais sur un simple portable, je mesure un facteur de 0,5 milli-secondes par unité de coût.

De toute façon, il faut faire **attention** : ce facteur d'échelle dépend de beaucoup de choses, à commencer par les réglages du SGBD (<http://www.postgresql.org/docs/current/interactive/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS>). Est-ce que ce ratio de 0,08 milli-secondes peut nous aider à prévoir le temps d'exécution d'une requête ? Voyons une requête plus compliquée qui compte le nombre de votes ayant eu lieu dans les 24 heures ayant suivi le message :

```
so=> EXPLAIN SELECT count(votes.id) FROM votes, posts WHERE votes.post = posts.id AND
so->                votes.creation <= posts.creation + interval '1 day';
                QUERY PLAN
-----
Aggregate  (cost=147911.94..147911.95 rows=1 width=4)
-> Hash Join  (cost=31161.46..145928.99 rows=793179 width=4)
    Hash Cond: (votes.post = posts.id)
    Join Filter: (votes.creation <= (posts.creation + '1 day'::interval))
-> Seq Scan on votes  (cost=0.00..36658.37 rows=2379537 width=12)
-> Hash  (cost=15834.65..15834.65 rows=881665 width=12)
    -> Seq Scan on posts  (cost=0.00..15834.65 rows=881665 width=12)
(7 rows)
```

PostgreSQL affiche un coût de 147911,95 soit 11,8 secondes. Et le résultat effectif est :

```
so=> EXPLAIN ANALYZE SELECT count(votes.id) FROM votes, posts WHERE votes.post = posts.id AND
                votes.creation <= posts.creation + interval '1 day';
                QUERY PLAN
-----
Aggregate  (cost=147911.94..147911.95 rows=1 width=4) (actual time=15125.455..15125.456 rows=1 loops=1)
...

```

soit 15 secondes au lieu des 11 prévues, ce qui n'est pas trop mal, pour une fonction de coût qui agrège beaucoup de choses différentes.

Si on trouve le résultat de EXPLAINANALYZE un peu dur à lire, on peut le copier-coller dans <http://explain-analyze.info/> et on a alors une présentation plus jolie.

Que voit-on d'autre sur la sortie de EXPLAIN pour cette requête? Que les coûts s'additionnent (regardez bien l'indentation des lignes commençant par -> pour voir comment faire l'addition). La recherche séquentielle la plus interne (sur la table Posts) coûte 15834,65, le hachage coûte le même prix (il n'a rien ajouté), l'autre recherche séquentielle (sur la table Votes) a un coût de 36658,37 et ces deux coûts vont s'ajouter, ainsi que le filtrage et le test de jointure, pour donner 145928,99, le coût de la jointure. Ensuite, un petit coût supplémentaire restera à payer pour l'agrégat (count ()) nous emmenant à 147911,95.

L'addition des coûts se voit encore mieux sur cette requête :

```
so=> EXPLAIN SELECT count(votes.id)*100/(SELECT count(votes.id) FROM Posts, Votes
so(>
      WHERE Votes.post = Posts.id) AS percent
so->
      FROM votes, posts WHERE
so->
      votes.post = posts.id AND
so->
      votes.creation > (posts.creation + interval '1 day')::date;
      QUERY PLAN
-----
Aggregate (cost=287472.95..287472.97 rows=1 width=4)
  InitPlan
    -> Aggregate (cost=133612.15..133612.16 rows=1 width=4)
      -> Hash Join (cost=30300.46..127663.31 rows=2379537 width=4)
        Hash Cond: (public.votes.post = public.posts.id)
        -> Seq Scan on votes (cost=0.00..36658.37 rows=2379537 width=8)
        -> Hash (cost=15834.65..15834.65 rows=881665 width=4)
          -> Seq Scan on posts (cost=0.00..15834.65 rows=881665 width=4)
      -> Hash Join (cost=31161.46..151877.84 rows=793179 width=4)
        Hash Cond: (public.votes.post = public.posts.id)
        Join Filter: (public.votes.creation > ((public.posts.creation + '1 day')::interval)::date)
        -> Seq Scan on votes (cost=0.00..36658.37 rows=2379537 width=12)
        -> Hash (cost=15834.65..15834.65 rows=881665 width=12)
          -> Seq Scan on posts (cost=0.00..15834.65 rows=881665 width=12)
(14 rows)
```

dont le temps d'exécution effectif est de 30 secondes, très proche des 23 secondes calculées en multipliant le coût total de 287472,97 par le facteur d'échelle de 0,08 millisecondes.

Plusieurs essais avec des bases très différentes montrent que le facteur d'échelle (la valeur en millisecondes d'une unité de coût) est plutôt stable. Voici un résultat avec la base de DNSmezzo (<http://www.dnsmezzo.net/>):

```
dnsmezzo2=> EXPLAIN ANALYZE SELECT DISTINCT substr(registered_domain,1,43) AS domain,count(registered_domain) AS
      QUERY PLAN
-----
Unique (cost=496460.23..496460.31 rows=10 width=12) (actual time=41233.482..42472.861 rows=234779 loops=1)
 -> Sort (cost=496460.23..496460.26 rows=10 width=12) (actual time=41233.478..41867.259 rows=234780 loops=1)
   Sort Key: (count(registered_domain)), (substr(registered_domain, 1, 43))
   Sort Method: external merge Disk: 11352kB
 -> HashAggregate (cost=496459.91..496460.06 rows=10 width=12) (actual time=38954.395..39462.324 rows=
   -> Bitmap Heap Scan on dns_packets (cost=197782.55..496245.34 rows=42914 width=12) (actual time=
     Recheck Cond: ((rcode = 3) AND (file = 3))
```

<http://www.bortzmeyer.org/explain-postgresql.html>

```
Filter: (NOT query)
-> BitmapAnd (cost=197782.55..197782.55 rows=87519 width=0) (actual time=34590.468..34590.468)
    -> Bitmap Index Scan on rcode_idx (cost=0.00..74228.75 rows=1250275 width=0) (actual time=34590.468..34590.468)
        Index Cond: (rcode = 3)
    -> Bitmap Index Scan on pcap_idx (cost=0.00..123532.10 rows=2083791 width=0) (actual time=34590.468..34590.468)
        Index Cond: (file = 3)
Total runtime: 42788.863 ms
(14 rows)
```

Quelques ressources intéressantes si on veut accélérer ses requêtes PostgreSQL :

- "*Performance Tips*" (<http://www.postgresql.org/docs/current/interactive/performance-tips.html>) dans la documentation officielle.
- "*Performance Optimization*" (http://wiki.postgresql.org/wiki/Performance_Optimization) sur le Wiki officiel.
- "*A Quick Look at How To Optimize PostgreSQL Queries*" (<http://www.mohawksoft.org/?q=node/56>).