Saurez-vous sauver cette instance ?

Configuration PostgreSQL

1.2

Table des matières

CORRECTION

Les VM hébergeant les instances PostgreSQL disposent de 8 Go de RAM et 4 CPU.

Afin de faciliter la mise en place de cet atelier, les paramètres suivants ont été mis en place dans le fichier `postgresql.conf`. Ils ne doivent pas être modifiés.

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
log_temp_files = 0
log_autovacuum_min_duration = 0
log_error_verbosity = default
log_min_duration_statement = 0
lc_messages = 'C'
max_parallel_workers_per_gather = 0
jit = off
```

Optimisation de l'instance

autovacuum

```
autovacuum = off
```

Sur cette instance, l'autovacuum est désactivé. C'est une grâve erreur ! Pour rappel, l'autovacuum est un processus d'arrière plan qui empêche une fragmentation excessive des tables et des index. Il met également à jour les statistiques sur les données qui servent pour l'optimiseur de requêtes. Son rôle est donc particulièrement important pour garder de bonnes performances.

Dans notre cas, il est donc nécessaire de passer ce paramètre à on dans le fichier `postgresql.conf`.

```
autovacuum = on
```

Et de procéder au redémarrage de l'instance pour que la modification soit prise en compte.

On peu également lancer un `VACUUM ANALYZE` sur l'ensemble de la base `stack` afin de commencer le travail d'optimisation sans attendre le passage de l'autovacuum.

shared_buffers

```
shared_buffers = 300MB
```

Le paramètre `shared_buffers` définit la taille de la mémoire partagée de PostgreSQL. C'est ici que seront stockés les blocs disques nécessaires aux différents processus de PostgreSQL. Cette zone permet donc d'éviter de trop fréquents accès disques car ces accès sont lents.

Pour dimensionner `shared_buffers` sur un serveur dédié à PostgreSQL, la documentation officielle donne 25 % de la mémoire vive totale comme un bon point de départ et déconseille de dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

La VM sur laquelle nous nous trouvons possède 8 Go de RAM. Il est donc préférable de passer le paramètre `shared_buffers` à `2GB`.

Un redémarrage de l'instance est nécessaire pour prendre en compte ce changement.

maintenance_work_mem

```
maintenance_work_mem = 64MB
```

`maintenance_work_mem` définit la mémoire utilisable pour les opérations de maintenance comme `VACUUM`, `CREATE INDEX` et ajouts de clés étrangères. Il permet de grandement améliorer les performances des ces opérations en évitant des écritures sur disques et en évitant de devoir procéder à plusieurs passes.

Sa valeur par défaut est généralement trop petite. De plus, comme il ne concerne que certaines opérations bien précises, il a peu de chance de provoquer une sur-consommation mémoire contrairement au paramètre `work_mem`.

Pour notre VM avec 8 Go de RAM, il est possible de définir `maintenance_work_mem` à `512MB`. `autovacuum_work_mem` peut être positionner moins haut, afin d'éviter une saturation mémoire.

```
maintenance_work_mem = 512MB
autovacuum_work_mem = 256MB
```

Cette valeur n'est pas figée, dans le cas de grosses opérations de maintenance (`VACUUM FULL`, beaucoup de création d'index, etc) il est possible de l'augmenter pour une connexion spécifique. Il faudra simplement veiller à limiter l'activité sur l'instance pour laisser suffisamment de RAM disponible.

checkpoints

```
checkpoint_completion_target = 0.5
select
  num_timed,
  num_requested,
  (num_requested::numeric * 100 / num_timed)::numeric(5,2) ratio
```

```
from pg_stat_checkpointer;

 num_timed | num_requested |  ratio
-----------+---------------+---------
     17852 |           111 |  0.62
```

Dans l'idéal les checkpoints sont périodiques. Le temps maximum entre deux checkpoints est fixé par `checkpoint_timeou` (par défaut 5 minutes). Si `max_wal_size` est trop petit par rapport à l'activité en écriture, alors un checkpoint est déclenché afin de pouvoir supprimer ou recycler les segments WAL qui s'accumulent dans les répertoire `pg_wal` de l'instance.

La requête ci-dessus montre que moins de 1% des checkpoints ont été déclenchés sur cette dernière condition, il n'est donc pas nécessaire d'augmenter `max_wal_size`. Dans le cas contraire, il faudra trouver un compromis entre fréquence des checkpoints et durée du *crash recovery*.

effective_cache_size

```
effective_cache_size = 4GB
```

Ce paramètre indique à PostgreSQL une estimation de la taille du cache disque du système (total du `shared buffers` et du cache du système). Une bonne pratique est de positionner ce paramètre à 2/3 de la quantité totale de RAM du serveur. Sur un système Linux, il est possible de donner une estimation plus précise en s'appuyant sur la valeur de colonne `cached` de la commande `free`.

```
$ free -h
              total        used        free      shared  buff/cache   available
Mem:           1,8G        172M         92M        166M        1,5G        1,3G
Swap:          1,5G         18M        1,5G
```

Une valeur de 1,5 Go conviendrait donc mieux :

```
effective_cache_size = 1.5GB
```

random_page_cost

```
random_page_cost = 4
```

Ce paramètre permet de faire appréhender au planificateur le fait que les lectures non-séquentielles (autrement dit, avec déplacement de la tête de lecture pour les disques rotatifs) sont autrement plus coûteuses que les lectures séquentielles. Par défaut, ces premières ont un coût 4 fois plus important que ces dernières. Ce n'est qu'une estimation,

cela n'a pas à voir directement avec la vitesse des disques. Ça le prend en compte, mais ça prend aussi en compte l'effet du cache. Cette estimation peut être revue. Si elle est revue à la baisse, les lectures non-séquentielles seront estimées moins coûteuses et, par conséquent, les parcours d'index seront favorisés. La valeur 4 est une estimation basique. En cas d'utilisation de disques rapides, il ne faut pas hésiter à descendre un peu cette valeur (entre 2 et 3 par exemple). Si les données tiennent entièrement en cache, ou sont stockées sur des disques SSD, il est même possible de descendre encore plus cette valeur.

Il n'est pas toujours simple d'estimer la performance du stockage. Ici nous avons une VM hébergée chez un fournisseur de cloud, comment savoir ? Il est possible d'utiliser des utilitaires comme `fio` ou autres, pour ceux qui connaissent. Nous proposons ici une méthode plus orientée DBA :

Prenons un gros index sur notre base `stack` :

```
stack=# select pg_size_pretty(pg_relation_size('interaction_pkey'));
 pg_size_pretty
----------------
 278 MB
(1 row)

stack=# select pg_size_pretty(pg_relation_size('interaction'));
 pg_size_pretty
----------------
 247 MB
```

Après avoir vidé les caches système et postgres, nous mesurons le temps de lecture pour un parcours séquentiel :

```
stack=# EXPLAIN (analyze, buffers, costs off) SELECT id FROM interaction ORDER BY id DESC;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Sort (actual time=620.897..701.214 rows=495172 loops=1)
   Sort Key: id DESC
   Sort Method: external merge  Disk: 5824kB
   Buffers: shared read=31630, temp read=728 written=730
   I/O Timings: shared read=322.423, temp read=2.840 write=10.847
   ->  Seq Scan on interaction (actual time=0.860..471.915 rows=495172 loops=1)
         Buffers: shared read=31630
         I/O Timings: shared read=322.423
 Planning Time: 0.330 ms
 Execution Time: 742.561 ms
```

Ici, 31630 blocs sont lus en 322 ms, soit 98 blocs / ms.

En baissant `random_page_cost`, on passe sur un parcours index-only :

```
stack=# SET random_page_cost = 2;
SET
stack=# EXPLAIN (analyze, buffers, costs off) SELECT id FROM interaction ORDER BY id DESC;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Index Only Scan Backward using interaction_pkey on interaction (actual time=1.283..506.109 rows=495172 loops=1)
   Heap Fetches: 0
   Buffers: shared hit=1 read=2736
   I/O Timings: shared read=407.905
 Planning Time: 0.849 ms
 Execution Time: 538.156 ms
```

Ici 2736 blocs sont lus en 408 ms, soit 6,7 blocs / ms.

On voit donc que les lectures séquentielles restent beaucoup plus rapides, et on peut donc conserver le paramétrage par défaut, ou peut-être le baisser légèrement.

effective_io_concurrency / maintenance_io_concurrency

```
effective_io_concurrency = 1
maintenance_io_concurrency = 10
```

Il a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (prefetch). Dans le cas d'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 ou RAID 10, n-1 s'il s'agit d'un RAID 5). Avec du SSD, il est possible de monter encore bien au-delà de cette valeur, étant donné la capacité de ce type de disque. La valeur maximale est de 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).

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

Là encore, il n'est pas toujours évident de connaître la performance du stockage. Testons en forçant artificiellement un parcours `Bitmap Scan`, le seul (à ce jour) à utiliser le *prefetching* :

```
stack=# EXPLAIN (analyze, buffers, costs off, settings) SELECT * FROM interaction WHERE id < 12284072;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on interaction (actual time=15.879..499.606 rows=140667 loops=1)
   Recheck Cond: (id < 12284072)
   Heap Blocks: exact=3978
   Buffers: shared hit=780 read=3978
   I/O Timings: shared read=361.553
   ->  Bitmap Index Scan on interaction_pkey (actual time=14.680..14.681 rows=140667 loops=1)
         Index Cond: (id < 12284072)
         Buffers: shared hit=780
 Settings: effective_io_concurrency = '1', enable_seqscan = 'off', enable_indexscan = 'off'
 Planning Time: 0.410 ms
 Execution Time: 510.137 ms
```

Ici on voit que 3978 blocs sont lus en dehors du cache, en 361 ms. Augmentons `effective_io_concurrency` :

```
stack=# SET effective_io_concurrency = 5;
SET
stack=# EXPLAIN (analyze, buffers, costs off, settings) SELECT * FROM interaction WHERE id < 12284072;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on interaction (actual time=13.049..271.611 rows=140667 loops=1)
   Recheck Cond: (id < 12284072)
```

```
   Heap Blocks: exact=3978
   Buffers: shared hit=780 read=3978
   I/O Timings: shared read=114.696
   ->  Bitmap Index Scan on interaction_pkey (actual time=11.645..11.645 rows=140667 loops=1)
         Index Cond: (id < 12284072)
         Buffers: shared hit=780
 Settings: effective_io_concurrency = '5', enable_seqscan = 'off', enable_indexscan = 'off'
 Planning Time: 0.401 ms
 Execution Time: 284.143 ms
```

Ici on voit que les mêmes 3978 blocs sont lus en dehors du cache, mais en seulement 114 ms. Les lectures sont plus que 3 fois plus rapides. Augmenter encore plus `effective_io_concurrency` ne semble pas avoir d'effet, nous restons donc avec cette valeur.

```
effective_io_concurrency = 5
```

Liens

Vous trouverez via le lien suivant le module de la formation PERF1 qui récapitule l'ensemble des bonnes pratiques pour la configuration du système et de PostgreSQL : module J1[1].

Optimisation des requêtes

Requête 1

```sql
SELECT * FROM users u
JOIN votes v ON (u.id = v.userid) AND v.votetypeid = 5
JOIN votes v2 ON (u.id = v2.userid) AND v2.votetypeid = 6
JOIN votes v3 ON (u.id = v3.userid) AND v3.votetypeid = 8
JOIN votes v4 ON (u.id = v4.userid) AND v4.votetypeid = 10
JOIN votes v5 ON (u.id = v5.userid) AND v5.votetypeid = 4;
```

On retrouve ici une requête avec de multiples jointures. Affichons le plan :

```
                                                            QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=54915.18..153109.34 rows=1 width=386) (actual time=3467.571..3467.610 rows=0 loops=1)
   Join Filter: (u.id = v5.userid)
   ->  Nested Loop  (cost=54914.76..153085.20 rows=1 width=346) (actual time=3467.570..3467.591 rows=0 loops=1)
         Join Filter: (u.id = v4.userid)
         ->  Hash Join  (cost=54914.33..153061.07 rows=1 width=306) (actual time=3467.569..3467.583 rows=0 loops=1)
               Hash Cond: (u.id = v3.userid)
               ->  Hash Join  (cost=49998.73..147643.39 rows=36 width=266) (actual time=3421.027..3421.039 rows=0 loops=1)
                     Hash Cond: (u.id = v2.userid)
                     ->  Hash Join  (cost=49974.59..108363.30 rows=923661 width=226) (actual time=1149.264..3236.052 rows=923508 loops=1)
                           Hash Cond: (v.userid = u.id)
                           ->  Seq Scan on votes v  (cost=0.00..19827.09 rows=923661 width=40) (actual time=0.024..352.296 rows=923508 loops=1)
                                 Filter: (votetypeid = 5)
```

[1] https://public.dalibo.com/exports/formation/manuels/modules/j1/j1.handout.html

```
                                Rows Removed by Filter: 28099
                             ->  Hash  (cost=17982.93..17982.93 rows=823093 width=186) (actual time=1147.762..1147.764 rows=823093 loops=1)
                                   Buckets: 32768  Batches: 64  Memory Usage: 1558kB
                                ->  Seq Scan on users u  (cost=0.00..17982.93 rows=823093 width=186) (actual time=0.872..526.647 rows=823093 loops=1)
                        ->  Hash  (cost=23.73..23.73 rows=32 width=40) (actual time=0.049..0.050 rows=5 loops=1)
                              Buckets: 1024  Batches: 1  Memory Usage: 9kB
                         ->  Index Scan using votes_type_idx on votes v2  (cost=0.42..23.73 rows=32 width=40) (actual time=0.032..0.036 rows=5 loops=1)
                                   Index Cond: (votetypeid = 6)
                  ->  Hash  (cost=4566.68..4566.68 rows=27914 width=40) (actual time=46.391..46.392 rows=28066 loops=1)
                        Buckets: 32768  Batches: 1  Memory Usage: 2230kB
                  ->  Index Scan using votes_type_idx on votes v3  (cost=0.42..4566.68 rows=27914 width=40) (actual time=0.075..34.423 rows=28066 loops=1)
                              Index Cond: (votetypeid = 8)
           ->  Index Scan using votes_type_idx on votes v4  (cost=0.42..23.73 rows=32 width=40) (never executed)
                 Index Cond: (votetypeid = 10)
      ->  Index Scan using votes_type_idx on votes v5  (cost=0.42..23.73 rows=32 width=40) (never executed)
            Index Cond: (votetypeid = 4)
 Planning Time: 1.381 ms
 Execution Time: 3467.959 ms
```

Affichons la description des tables `users`, et `votes` :

```
                                                    Table « public.users »
    Colonne     |            Type             | Collationnement | NULL-able |          Par défaut
----------------+-----------------------------+-----------------+-----------+------------------------------
 id             | bigint                      |                 | not null  | generated always as identity
 creationdate   | timestamp without time zone |                 | not null  |
 displayname    | character varying(100)      |                 | not null  |
 lastaccessdate | timestamp without time zone |                 |           |
 location       | text                        |                 |           |
 aboutme        | text                        |                 |           |
 age            | integer                     |                 |           |
Index :
    "users_pkey" PRIMARY KEY, btree (id)
    "user_created_at_idx" btree (creationdate) WITH (fillfactor='100')
    "user_display_idx" hash (displayname) WITH (fillfactor='100')
Référencé par :
    TABLE "badges" CONSTRAINT "fk_badges_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "comments" CONSTRAINT "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posthistory" CONSTRAINT "fk_posthistory_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posts" CONSTRAINT "fk_posts_owneruserid" FOREIGN KEY (owneruserid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "votes" CONSTRAINT "fk_votes_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
Méthode d'accès : heap


                                                    Table « public.votes »
   Colonne    |            Type             | Collationnement | NULL-able |          Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 postid       | bigint                      |                 | not null  |
 votetypeid   | bigint                      |                 | not null  |
 userid       | bigint                      |                 | not null  |
 creationdate | timestamp without time zone |                 | not null  |
Index :
    "votes_pkey" PRIMARY KEY, btree (id)
    "votes_creation_date_idx" btree (creationdate) WITH (fillfactor='100')
    "votes_post_id_idx" hash (postid) WITH (fillfactor='100')
    "votes_type_idx" btree (votetypeid) WITH (fillfactor='100')
Contraintes de clés étrangères :
    "fk_votes_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_votes_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_votes_votetypeid" FOREIGN KEY (votetypeid) REFERENCES votetypes(id) ON UPDATE CASCADE ON DELETE CASCADE
Méthode d'accès : heap
```

Il y a bien un index sur le champ `id` de la table `users` car c'est la clé primaire. Sur la table `votes`, il y a un index sur le champ `votetypeid` qui est bien utilisé mais qui ne suffit pas à optimiser le temps d'exécution de la requête.

La première modification serait d'ajouter un index sur le champ `userid` de la table `votes` afin d'améliorer la rapidité des jointures. Pour être encore plus performant, cet index pourrait également stocker le champ `votetypeid` pour accélérer la résolution des instructions `votetypeid = x`.

Créons l'index suivant :

```sql
create index on votes (votetypeid,userid);
```

Le nouveau plan :

```
                                                                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=20.01..87377.92 rows=1 width=386) (actual time=2077.023..2077.036 rows=0 loops=1)
   Join Filter: (u.id = v5.userid)
   ->  Nested Loop  (cost=19.58..87377.42 rows=1 width=346) (actual time=2077.022..2077.034 rows=0 loops=1)
         Join Filter: (u.id = v4.userid)
         ->  Nested Loop  (cost=19.16..87376.93 rows=1 width=306) (actual time=2077.022..2077.032 rows=0 loops=1)
               ->  Merge Join  (cost=18.73..87357.89 rows=36 width=266) (actual time=2077.020..2077.029 rows=0 loops=1)
                     Merge Cond: (u.id = v2.userid)
                     ->  Merge Join  (cost=18.31..84976.66 rows=923661 width=226) (actual time=0.125..1945.798 rows=910754 loops=1)
                           Merge Cond: (u.id = v.userid)
                  ->  Index Scan using users_pkey on users u  (cost=0.42..31138.79 rows=823093 width=186) (actual time=0.044..349.562 rows=717548 loops=1)
                  ->  Index Scan using votes_votetypeid_userid_idx on votes v  (cost=0.42..40237.84 rows=923661 width=40) (actual time=0.051..1029.857 rows=91075
                                 Index Cond: (votetypeid = 5)
                     ->  Materialize  (cost=0.42..71.61 rows=32 width=40) (actual time=0.041..0.070 rows=5 loops=1)
                  ->  Index Scan using votes_votetypeid_userid_idx on votes v2  (cost=0.42..71.53 rows=32 width=40) (actual time=0.037..0.056 rows=5 loops=1)
                                 Index Cond: (votetypeid = 6)
               ->  Index Scan using votes_votetypeid_userid_idx on votes v3  (cost=0.42..0.51 rows=2 width=40) (never executed)
                     Index Cond: ((votetypeid = 8) AND (userid = u.id))
         ->  Index Scan using votes_votetypeid_userid_idx on votes v4  (cost=0.42..0.48 rows=1 width=40) (never executed)
               Index Cond: ((votetypeid = 10) AND (userid = v.userid))
   ->  Index Scan using votes_votetypeid_userid_idx on votes v5  (cost=0.42..0.48 rows=1 width=40) (never executed)
         Index Cond: ((votetypeid = 4) AND (userid = v.userid))
 Planning Time: 2.513 ms
 Execution Time: 2077.275 ms
```

L'amélioration n'est pas flagrante, nous allons donc regarder la configuration de PostgreSQL. Le paramètre `join_collapse_limit` détermine la limite en nombre de tables à partir de laquelle l'optimiseur ne cherchera plus à traiter tous les cas possibles de réordonnancement des jointures.

Voyons sa valeur :

```
stack=# show join_collapse_limit;
 join_collapse_limit
---------------------
 2
```

Notre requête compte 6 tables. Nous allons remettre `join_collapse_limit` à sa valeur par défaut qui est 8 :

```sql
stack=# SET join_collapse_limit = 8;
```

Le plan devient :

```
                                                                                       QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=26.26..61.18 rows=1 width=386) (actual time=0.194..0.205 rows=0 loops=1)
   Join Filter: (u.id = v5.userid)
   ->  Nested Loop  (cost=25.83..60.69 rows=1 width=346) (actual time=0.187..0.197 rows=0 loops=1)
         ->  Nested Loop  (cost=25.41..60.16 rows=1 width=306) (actual time=0.187..0.196 rows=0 loops=1)
               ->  Nested Loop  (cost=24.98..57.68 rows=1 width=266) (actual time=0.186..0.196 rows=0 loops=1)
```

```
                    ->  Hash Join  (cost=24.56..49.24 rows=1 width=80) (actual time=0.186..0.194 rows=0 loops=1)
                          Hash Cond: (v2.userid = v4.userid)
                       ->  Index Scan using votes_type_idx on votes v2  (cost=0.42..23.73 rows=32 width=40) (actual time=0.081..0.086 rows=5 loops=1)
                                Index Cond: (votetypeid = 6)
                          ->  Hash  (cost=23.73..23.73 rows=32 width=40) (actual time=0.070..0.071 rows=1 loops=1)
                                Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Index Scan using votes_type_idx on votes v4  (cost=0.42..23.73 rows=32 width=40) (actual time=0.055..0.057 rows=1 loops=1)
                                Index Cond: (votetypeid = 10)
                    ->  Index Scan using users_pkey on users u  (cost=0.42..8.44 rows=1 width=186) (never executed)
                          Index Cond: (id = v2.userid)
              ->  Index Scan using votes_userid_votetypeid_idx on votes v  (cost=0.42..1.81 rows=67 width=40) (never executed)
                    Index Cond: ((userid = u.id) AND (votetypeid = 5))
        ->  Index Scan using votes_userid_votetypeid_idx on votes v3  (cost=0.42..0.51 rows=2 width=40) (never executed)
              Index Cond: ((userid = u.id) AND (votetypeid = 8))
  ->  Index Scan using votes_userid_votetypeid_idx on votes v5  (cost=0.42..0.48 rows=1 width=40) (never executed)
        Index Cond: ((userid = v.userid) AND (votetypeid = 4))
 Planning Time: 10.023 ms
 Execution Time: 0.678 ms
```

Grâce à l'augmentation du paramètre `join_collapse_limit`, l'optimiseur a pu réordonner les jointures afin d'obtenir un plan plus performant. On peut également constater que le temps de plannification (`Planning Time`) est quatorze fois plus long en augmentant `join_collapse_limit`. Il faut donc être vigilant en cas d'augmentation globale de ce paramètre car les temps de planification d'autres requêtes peuvent augmenter drastiquement.

Requête 2

```sql
select * from posthistory where extract('year' from creationdate) = 2010;
```

Voyons son plan d'exécution :

```
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
Seq Scan on posthistory  (cost=0.00..715986.77 rows=46272 width=503) (actual time=0.081..18183.154 rows=31967 loops=1)
   Filter: (date_part('year'::text, creationdate) = '2010'::double precision)
   Rows Removed by Filter: 9222351
 Planning Time: 0.559 ms
 Execution Time: 18186.291 ms
(5 lignes)
```

On constate que la requête réalise un `Seq Scan` et qu'elle ramène 31967 lignes

Regardons la définition de la table `posthistory` :

```
                                      Table « public.posthistory »
      Colonne      |            Type             | Collationnement | NULL-able |          Par défaut
-------------------+-----------------------------+-----------------+-----------+-------------------------------
 id                | bigint                      |                 | not null  | generated always as identity
 posthistorytypeid | bigint                      |                 | not null  |
 postid            | bigint                      |                 | not null  |
 creationdate      | timestamp without time zone |                 | not null  |
 userid            | bigint                      |                 | not null  |
 posttext          | text                        |                 |           |
 jsonfield         | json                        |                 |           |
Index :
    "posthistory_pkey" PRIMARY KEY, btree (id)
    "ph_post_type_id_idx" btree (posthistorytypeid) WITH (fillfactor='100')
    "ph_postid_idx" hash (postid) WITH (fillfactor='100')
    "ph_userid_idx" btree (userid) WITH (fillfactor='100')
Contraintes de clés étrangères :
    "fk_posthistory_posthistorytypeid" FOREIGN KEY (posthistorytypeid) REFERENCES posthistorytypes(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_posthistory_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_posthistory_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Le champ `creationdate` est de type `timestamp without time zone` et aucun index n'est créé sur celui-ci. Nous allons donc le créer pour voir si cela améliore le comportement :

```
create index on posthistory (creationdate);
```

Analysons de nouveau le plan d'exécution de la requête :

```
                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Seq Scan on posthistory  (cost=0.00..715986.77 rows=46272 width=503) (actual time=0.091..16936.830 rows=31967 loops=1)
   Filter: (date_part('year'::text, creationdate) = '2010'::double precision)
   Rows Removed by Filter: 9222351
 Planning Time: 1.323 ms
 Execution Time: 16939.975 ms
(5 lignes)
```

Malheureusement, aucun changement.

À partir du moment où une clause `WHERE` applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement. C'est comme essayer d'utiliser l'index d'un livre en anglais pour trouver les pages contenant les mots dont la traduction en français est « fenêtre ». L'index de répond pas à la question posée. Il nous faudrait un index non plus sur les mots anglais, mais sur leur traduction en français.

C'est exactement ce que font les index fonctionnels : ils indexent le résultat d'une fonction appliquée à l'enregistrement.

Créons l'index fonctionnel :

```
create index on posthistory(extract('year' from creationdate));
```

Regardons le plan :

```
                                                             QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on posthistory  (cost=871.04..142466.06 rows=46272 width=503) (actual time=10.447..53.581 rows=31967 loops=1)
   Recheck Cond: (date_part('year'::text, creationdate) = '2010'::double precision)
   Heap Blocks: exact=2243
   ->  Bitmap Index Scan on posthistory_date_part_idx  (cost=0.00..859.47 rows=46272 width=0) (actual time=9.810..9.816 rows=31967 loops=1)
         Index Cond: (date_part('year'::text, creationdate) = '2010'::double precision)
 Planning Time: 1.810 ms
 Execution Time: 56.831 ms
```

On obtient grâce à cet index une nette amélioration du temps d'exécution.

Requête 3

```
SELECT COUNT(ph.posthistorytypeid) AS nb, pht.name AS name
FROM posthistory ph JOIN posthistorytypes pht ON (ph.posthistorytypeid = pht.id)
WHERE ph.creationdate < '2011-01-01' GROUP BY 2;
```

Regardons le plan de la requête :

```
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=696813.34..697180.01 rows=20 width=22) (actual time=13964.060..13976.620 rows=15 loops=1)
   Group Key: pht.name
   ->  Sort  (cost=696813.34..696935.50 rows=48862 width=22) (actual time=13963.649..13968.025 rows=31967 loops=1)
         Sort Key: pht.name
         Sort Method: quicksort  Memory: 3266kB
         ->  Hash Join  (cost=1.45..693007.87 rows=48862 width=22) (actual time=0.133..13929.049 rows=31967 loops=1)
               Hash Cond: (ph.posthistorytypeid = pht.id)
               ->  Seq Scan on posthistory ph  (cost=0.00..692850.97 rows=48862 width=8) (actual time=0.031..13912.577 rows=31967 loops=1)
                     Filter: (creationdate < '2011-01-01 00:00:00'::timestamp without time zone)
                     Rows Removed by Filter: 9222351
               ->  Hash  (cost=1.20..1.20 rows=20 width=22) (actual time=0.058..0.075 rows=20 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 10kB
                     ->  Seq Scan on posthistorytypes pht  (cost=0.00..1.20 rows=20 width=22) (actual time=0.018..0.029 rows=20 loops=1)
 Planning Time: 1.253 ms
 Execution Time: 13977.056 ms
```

A première vue, la requête utilise un `Seq Scan` pour répondre à la clause `WHERE`. Elle réalise également un `Seq Scan` sur la table `posthistorytypes` pour faire la jointure.

Reprenons la définition des tables `posthistory` et `posthistorytypes` :

```
                                          Table « public.posthistory »
      Colonne      |            Type             | Collationnement | NULL-able |          Par défaut
-------------------+-----------------------------+-----------------+-----------+------------------------------
 id                | bigint                      |                 | not null  | generated always as identity
 posthistorytypeid | bigint                      |                 | not null  |
 postid            | bigint                      |                 | not null  |
 creationdate      | timestamp without time zone |                 | not null  |
 userid            | bigint                      |                 | not null  |
 posttext          | text                        |                 |           |
 jsonfield         | json                        |                 |           |
Index :
    "posthistory_pkey" PRIMARY KEY, btree (id)
    "ph_post_type_id_idx" btree (posthistorytypeid) WITH (fillfactor='100')
    "ph_userid_idx" btree (userid) WITH (fillfactor='100')
    "ph_postid_idx" hash (postid) WITH (fillfactor='100')
Contraintes de clés étrangères :
    "fk_posthistory_posthistorytypeid" FOREIGN KEY (posthistorytypeid) REFERENCES posthistorytypes(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_posthistory_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_posthistory_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE


                        Table « public.posthistorytypes »
 Colonne |  Type  | Collationnement | NULL-able |          Par défaut
---------+--------+-----------------+-----------+------------------------------
 id      | bigint |                 | not null  | generated always as identity
 name    | text   |                 | not null  |
Index :
    "posthistorytypes_pkey" PRIMARY KEY, btree (id)
Référencé par :
   TABLE "posthistory" CONSTRAINT "fk_posthistory_posthistorytypeid" FOREIGN KEY (posthistorytypeid) REFERENCES posthistorytypes(id) ON UPDATE CASCADE ON DELETE
Méthode d'accès : heap
```

Rien à signaler sur la table `posthistorytypes`, il y a bien un index sur le champ `id` car c'est une clé primaire. En revanche, sur la table `posthistory`, aucun index n'est présent sur le champ `creationdate`.

Voyons si la création de celui-ci change quelque chose :

```sql
create index on posthistory (creationdate);
```

Le plan d'exécution devient celui-ci :

```
                                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=5927.45..5927.65 rows=20 width=22) (actual time=163.843..163.855 rows=15 loops=1)
   Group Key: pht.name
   Batches: 1  Memory Usage: 24kB
   ->  Hash Join  (cost=1.89..5623.86 rows=60717 width=22) (actual time=1.061..143.954 rows=31967 loops=1)
         Hash Cond: (ph.posthistorytypeid = pht.id)
      -> Index Scan using posthistory_creationdate_idx on posthistory ph  (cost=0.43..5429.25 rows=60717 width=8) (actual time=0.570..115.495 rows=31967 loops=1
               Index Cond: (creationdate < '2011-01-01 00:00:00'::timestamp without time zone)
         ->  Hash  (cost=1.20..1.20 rows=20 width=22) (actual time=0.453..0.456 rows=20 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 10kB
               ->  Seq Scan on posthistorytypes pht  (cost=0.00..1.20 rows=20 width=22) (actual time=0.401..0.434 rows=20 loops=1)
 Planning Time: 3.252 ms
 Execution Time: 163.986 ms
```

Cette fois, la requête utilise bien un index pour répondre à la clause WHERE. Le `Seq Scan` sur la table `posthistorytypes` n'est en soit pas génant car la table ne compte que 20 lignes. Même si un index est bien présent sur la clé primaire de cette table, l'optimiseur ne l'utilisera jamais vu le peu de lignes qu'elle contient.

Le temps de réponse est encore élevé, regardons s'il n'est pas possible de faire mieux.

Une possibilité serait de réaliser un index multicolonnes afin d'inclure le champ `posthistorytypeid` et ainsi éviter les accès à la table `posthistory`.

Créons l'index :

```
create index on posthistory (creationdate,posthistorytypeid);
```

Le plan d'exécution devient celui-ci :

```
                                                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=2489.18..2489.38 rows=20 width=22) (actual time=48.779..48.824 rows=15 loops=1)
   Group Key: pht.name
   Batches: 1  Memory Usage: 24kB
   ->  Hash Join  (cost=1.89..2185.59 rows=60717 width=22) (actual time=0.271..33.683 rows=31967 loops=1)
         Hash Cond: (ph.posthistorytypeid = pht.id)
      -> Index Only Scan using posthistory_creationdate_posthistorytypeid_idx on posthistory ph  (cost=0.43..1990.98 rows=60717 width=8) (actual time=0.072..19.
               Index Cond: (creationdate < '2011-01-01 00:00:00'::timestamp without time zone)
               Heap Fetches: 0
         ->  Hash  (cost=1.20..1.20 rows=20 width=22) (actual time=0.125..0.128 rows=20 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 10kB
               ->  Seq Scan on posthistorytypes pht  (cost=0.00..1.20 rows=20 width=22) (actual time=0.087..0.097 rows=20 loops=1)
 Planning Time: 7.961 ms
 Execution Time: 49.257 ms
```

On passe donc d'un `Index Scan` à un `Index Only Scan` pour la clause WHERE ce qui est beaucoup plus rapide puisqu'aucun accès à la table `posthistory` n'est nécessaire.

Requête 4

```
SELECT * FROM interaction WHERE interac = 22;
```

Le plan :

```
                                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on interaction  (cost=971.44..30830.70 rows=10712 width=24) (actual time=10.503..1190.156 rows=10172 loops=1)
   Recheck Cond: (interac = 22)
   Heap Blocks: exact=5062
   ->  Bitmap Index Scan on interaction_interac_idx  (cost=0.00..968.76 rows=10712 width=0) (actual time=8.214..8.220 rows=10172 loops=1)
         Index Cond: (interac = 22)
 Planning Time: 3.850 ms
 Execution Time: 1193.905 ms
```

A première vue rien d'anormal ici. Voyons la description de la table `interaction` :

```
                                      Table « public.interaction »
   Colonne    |            Type             | Collationnement | NULL-able |          Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 creationdate | timestamp without time zone |                 |           |
 interac      | bigint                      |                 |           |
Index :
    "interaction_pkey" PRIMARY KEY, btree (id)
    "interaction_interac_idx" btree (interac)
```

Rien d'anormal non plus. Comme tout semble correct, nous allons nous pencher sur l'index `interaction_interac_idx` pour voir son niveau de fragmentation (`bloat`).

La requête suivante est disponible pour avoir une estimation du niveau de bloat des index : requête bloat[2]. Cette requête nécessite d'être `superuser` et d'avoir des statistiques à jour.

Voici le résultat :

```
 current_database | schemaname |    tablename     |          indexname            | real_size_ | fillfactor | extra_size_ | bloat_size_ | bloat_ratio_
------------------+------------+------------------+-------------------------------+------------+------------+-------------+-------------+--------------
 stack            | public     | interaction      | interaction_pkey              | 257 MB     |         90 | 247 MB      | 246 MB      |         95.7
 stack            | public     | interaction      | interaction_interac_idx       | 74 MB      |         90 | 64 MB       | 63 MB       |         85.2
```

On constate que l'index `interaction_interac_idx` utilisé par la requête est fragmenté à 85.2 %. Cette fragmentation dégrade fortement les performances des index *B-tree*. La seule solution disponible ici est la réindexation de l'index en question ou de toute la table pour traiter en même temps la clé primaire.

Notez qu'une façon plus fiable et précise d'obtenir la fragmentation serait d'utiliser l'extension `pgstattuple`, mais il y a un impact non-négligeable en consommation d'I/O.

Attention avec la réindexation car celle-ci pose un verrou empêchant la planification de toutes les requêtes sur la table concernée. Il est possible de reindexer sans blocage avec la clause `CONCURRENTLY`, mais celle-ci est plus lente et nécessite de vérifier la validité de l'index à la fin de l'opération.

Réindexons `interaction_interac_idx` :

```sql
reindex index interaction_interac_idx;
```

Le nouveau plan :

[2] https://github.com/ioguix/pgsql-bloat-estimation/blob/master/btree/btree_bloat-superuser.sql

```
                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on interaction  (cost=123.07..29877.11 rows=10664 width=24) (actual time=4.125..41.418 rows=10172 loops=1)
   Recheck Cond: (interac = 22)
   Heap Blocks: exact=5062
   ->  Bitmap Index Scan on interaction_interac_idx  (cost=0.00..120.40 rows=10664 width=0) (actual time=2.656..2.662 rows=10172 loops=1)
         Index Cond: (interac = 22)
 Planning Time: 1.096 ms
 Execution Time: 42.839 ms
```

Le parcours de l'index est maintenant beaucoup plus rapide.

Requête 5

```sql
select id from posts where closeddate is not null;
```

Le plan :

```
                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on posts  (cost=0.00..361445.85 rows=87452 width=8) (actual time=1.152..8313.831 rows=88398 loops=1)
   Filter: (closeddate IS NOT NULL)
   Rows Removed by Filter: 3142425
 Planning Time: 0.317 ms
 Execution Time: 8340.057 ms
```

On réalise donc ici un `Seq Scan` pour répondre à la clause `WHERE`. Voyons la description de la table `posts` :

```
                                            Table « public.posts »
   Colonne    |            Type             | Collationnement | NULL-able |         Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 posttypeid   | bigint                      |                 | not null  |
 creationdate | timestamp without time zone |                 | not null  |
 body         | text                        |                 |           |
 owneruserid  | bigint                      |                 | not null  |
 title        | text                        |                 |           |
 closeddate   | timestamp without time zone |                 |           |
 jsonfield    | jsonb                       |                 |           |
Index :
    "posts_pkey" PRIMARY KEY, btree (id)
    "posts_creation_date_idx" btree (creationdate) WITH (fillfactor='100')
    "posts_owner_user_id_idx" hash (owneruserid) WITH (fillfactor='100')
    "posts_post_type_id_idx" btree (posttypeid) WITH (fillfactor='100')
Contraintes de clés étrangères :
    "fk_posts_owneruserid" FOREIGN KEY (owneruserid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_posts_posttypeid" FOREIGN KEY (posttypeid) REFERENCES posttypes(id) ON UPDATE CASCADE ON DELETE CASCADE
Référencé par :
    TABLE "comments" CONSTRAINT "fk_comments_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posthistory" CONSTRAINT "fk_posthistory_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "postlinks" CONSTRAINT "fk_postlinks_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "postlinks" CONSTRAINT "fk_postlinks_relatedpostid" FOREIGN KEY (relatedpostid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posttags" CONSTRAINT "fk_posttags_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "votes" CONSTRAINT "fk_votes_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
Méthode d'accès : heap
```

Aucun index n'est créé sur le champ `closeddate`. Testons donc en premier un index sur ce champ.

```sql
create index on posts (closeddate);
```

Le nouveau plan donne ceci :

```
                                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
Index Scan using posts_closeddate_idx on posts  (cost=0.43..76067.23 rows=87448 width=8) (actual time=0.858..7725.912 rows=88398 loops=1)
   Index Cond: (closeddate IS NOT NULL)
 Planning Time: 0.782 ms
 Execution Time: 7750.620 ms
```

Aucun changement ! On passe bien par le nouvel index pour répondre à la clause `WHERE`. Mais celui-ci n'améliore pas la situation. Voyons ce que ça donne en utilisant un index partiel.

Un index partiel ne couvre qu'une partie des enregistrements. Il est donc beaucoup plus petit. En contrepartie, il ne pourra être utilisé que si sa condition est définie dans la requête.

Testons l'index partiel suivant :

```sql
create index on posts (id) where closeddate is not null;
```

Le nouveau plan :

```
                                                           QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using posts_id_idx on posts  (cost=0.29..2280.01 rows=87448 width=8) (actual time=0.155..27.092 rows=88398 loops=1)
   Heap Fetches: 0
 Planning Time: 1.947 ms
 Execution Time: 35.077 ms
```

Le gain est flagrant. En plus d'avoir un index plus petit, on passe d'un `Index Scan` à un `Index Only Scan`. L'index porte sur le même champ que l'instruction `SELECT` et il répond déjà à la clause `WHERE`. il n'est donc pas nécessaire de récupérer des données dans la table `posts`.

Requête 6

```sql
SELECT id FROM posts WHERE title LIKE 'pi%';
```

Voyons le plan :

```
                                                QUERY PLAN
----------------------------------------------------------------------------------------------------------
 Seq Scan on posts  (cost=0.00..369521.29 rows=138 width=8) (actual time=16.042..8124.413 rows=79 loops=1)
   Filter: (title ~~ 'pi%'::text)
   Rows Removed by Filter: 3230744
 Planning Time: 9.438 ms
 Execution Time: 8124.641 ms
```

Cette requête effectue une recherche sur le motif `pi%` en utlisant un `Seq Scan`.

Reprenons la description de la table `posts` :

```
                                      Table « public.posts »
   Colonne    |            Type             | Collationnement | NULL-able |          Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 posttypeid   | bigint                      |                 | not null  |
 creationdate | timestamp without time zone |                 | not null  |
 body         | text                        |                 |           |
 owneruserid  | bigint                      |                 | not null  |
 title        | text                        |                 |           |
 closeddate   | timestamp without time zone |                 |           |
 jsonfield    | jsonb                       |                 |           |
Index :
    "posts_pkey" PRIMARY KEY, btree (id)
    "posts_creation_date_idx" btree (creationdate) WITH (fillfactor='100')
    "posts_owner_user_id_idx" hash (owneruserid) WITH (fillfactor='100')
    "posts_post_type_id_idx" btree (posttypeid) WITH (fillfactor='100')
Contraintes de clés étrangères :
    "fk_posts_owneruserid" FOREIGN KEY (owneruserid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_posts_posttypeid" FOREIGN KEY (posttypeid) REFERENCES posttypes(id) ON UPDATE CASCADE ON DELETE CASCADE
Référencé par :
    TABLE "comments" CONSTRAINT "fk_comments_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posthistory" CONSTRAINT "fk_posthistory_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "postlinks" CONSTRAINT "fk_postlinks_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "postlinks" CONSTRAINT "fk_postlinks_relatedpostid" FOREIGN KEY (relatedpostid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posttags" CONSTRAINT "fk_posttags_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "votes" CONSTRAINT "fk_votes_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
Méthode d'accès : heap
```

Le champ `title` est de type `text` et n'est pas indexé. Essayons d'abord cela :

```
create index on posts (title);
```

Le nouveau plan :

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Seq Scan on posts  (cost=0.00..369521.29 rows=138 width=8) (actual time=18.314..7848.768 rows=79 loops=1)
   Filter: (title ~~ 'pi%'::text)
   Rows Removed by Filter: 3230744
 Planning Time: 6.324 ms
 Execution Time: 7849.054 ms
```

Aucun changement, l'index n'est même pas utilisé.

En fait, ce comportement est normal. Chaque index est associé à une classe d'opérateur qui identifie les opérateurs que l'index doit utiliser sur une colonne. Par exemple, un index *B-tree* sur une colonne de type `text` utilise la classe `text_ops`. Cette classe d'opérateurs comprend des fonctions de comparaison pour les valeurs de type `text`. En pratique, la classe d'opérateurs par défaut est généralement suffisante.

Mais pour nous, ce n'est pas le cas. L'utilisation du mot clé `LIKE` dans la requête rend inutilisatble l'index créé précédemment.

Pour que l'instruction `LIKE` puisse bénéficier d'un index, il va falloir utiliser une classe d'opérateur spécifique : `text_pattern_ops`. À la différence des classes d'opérateurs par défaut, `text_pattern_ops` compare strictement caractère par caractère plutôt que suivant les règles de tri spécifiques aux paramètres régionaux (*locale*), ce qui permet de transformer un `LIKE 'pi%'` en `>= 'pi' AND < 'pj'`, et d'utiliser un index btree standard.

Créons cet index :

```sql
create index on posts (title text_pattern_ops);
```

Affichons le nouveau plan :

```
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Index Scan using posts_title_idx1 on posts  (cost=0.56..8.58 rows=138 width=8) (actual time=0.653..21.529 rows=79 loops=1)
   Index Cond: ((title ~>=~ 'pi'::text) AND (title ~<~ 'pj'::text))
   Filter: (title ~~ 'pi%'::text)
 Planning Time: 5.664 ms
 Execution Time: 21.702 ms
```

Le gain est énorme. L'index est cette fois bien utilisé.

Requête 7

```sql
SELECT * FROM comments WHERE creationdate BETWEEN '2021-01-01' AND '2021-02-01';
```

Le plan d'exécution :

```
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on comments  (cost=0.00..243299.51 rows=53261 width=220) (actual time=4426.044..4831.616 rows=55746 loops=1)
  Filter: ((creationdate >= '2021-01-01 00:00:00'::timestamp without time zone) AND (creationdate <= '2021-02-01 00:00:00'::timestamp without time zone))
   Rows Removed by Filter: 5689821
 Planning Time: 2.009 ms
 Execution Time: 4838.818 ms
```

La requête exécute un `Seq Scan` et retourne 55746 lignes. On constate également qu'une grande quantité de lignes est supprimée par la clause `WHERE`. On peut donc supposer qu'un index est nécessaire.

Voyons la description de la table `comments` :

```
                                    Table « public.comments »
   Colonne    |            Type             | Collationnement | NULL-able |          Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 postid       | bigint                      |                 | not null  |
 text         | text                        |                 |           |
 creationdate | timestamp without time zone |                 | not null  |
 userid       | bigint                      |                 | not null  |
 jsonfield    | json                        |                 |           |
Index :
    "comments_pkey" PRIMARY KEY, btree (id)
    "cmnts_postid_idx" hash (postid) WITH (fillfactor='100')
    "comments_creationdate_idx" btree (creationdate) INVALID
Contraintes de clés étrangères :
    "fk_comments_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Il y a bien un index sur le champ `creationdate`. Cependant, en regardant de plus près on remarque le mot clé `INVALID` qui indique que celui-ci n'est pas utilisable. Ce problème peut apparaitre notamment lors des opérations de création d'index ou de réindexation avec la clause `CONCURRENTLY`.

La requête suivante permet de remonter l'ensemble des index invalides d'une base de données :

```
SELECT  indrelid::regclass, indexrelid::regclass FROM pg_index WHERE indisvalid ='f';
 indrelid |        indexrelid
----------+--------------------------
 comments | comments_creationdate_idx
```

On y retrouve bien l'index `comments_creationdate_idx` sur la table `comments`.

L'outils `pgCluu`[3] permet également de récupérer la liste des index invalides.

Pour corriger ce problème il va falloir réaliser une réindexation de l'index invalide.

```
reindex index comments_creationdate_idx;
```

Vérifions s'il n'y plus d'index invalide :

```
SELECT  indrelid::regclass, indexrelid::regclass FROM pg_index WHERE indisvalid ='f';
 indrelid | indexrelid
----------+------------
```

Testons de nouveau le plan :

```
                                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------
Index Scan using comments_creationdate_idx on comments  (cost=0.43..3138.53 rows=53261 width=220) (actual time=0.184..77.822 rows=55746 loops=1)
  Index Cond: ((creationdate >= '2021-01-01 00:00:00'::timestamp without time zone) AND (creationdate <= '2021-02-01 00:00:00'::timestamp without time zone))
 Planning Time: 1.246 ms
 Execution Time: 83.664 ms
```

La requête bénéficie bien de l'index.

Requête 8

```
select * from comments where jsonfield is not null;
```

Cette requête ressemble beaucoup à la requête 5. Voyons son plan d'exécution :

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on comments  (cost=0.00..214571.67 rows=1 width=220) (actual time=4672.189..4672.231 rows=0 loops=1)
   Filter: (jsonfield IS NOT NULL)
   Rows Removed by Filter: 5745567
 Planning Time: 0.368 ms
 Execution Time: 4672.405 ms
```

[3] https://github.com/darold/pgcluu

Un `Seq Scan` avec beaucoup de lignes supprimées par la clause `WHERE`, il est donc fort probable qu'un index soit manquant ou non utilisé. Affichons la description de la table `comments` :

```
                                    Table « public.comments »
   Colonne    |            Type             | Collationnement | NULL-able |          Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 postid       | bigint                      |                 | not null  |
 text         | text                        |                 |           |
 creationdate | timestamp without time zone |                 | not null  |
 userid       | bigint                      |                 | not null  |
 jsonfield    | json                        |                 |           |
Index :
    "comments_pkey" PRIMARY KEY, btree (id)
    "cmnts_postid_idx" hash (postid) WITH (fillfactor='100')
    "comments_creationdate_idx" btree (creationdate) INVALID
Contraintes de clés étrangères :
    "fk_comments_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Aucun index sur le champ `jsonfield`. Est-ce qu'un index classique sur ce champ permettrait d'améliorer la requête ?

A priori non. Nous sommes ici dans un cas similaire à la requête 5. Le filtre de la requête ne change pas et vu le peu de résultats retournés, un index partiel parait idéal.

```
create index on comments (id) WHERE jsonfield is not null;
```

Le nouveau plan :

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using comments_id_idx on comments  (cost=0.12..4.14 rows=1 width=220) (actual time=0.021..0.028 rows=0 loops=1)
 Planning Time: 1.732 ms
 Execution Time: 0.163 ms
```

Comme l'index est très petit et qu'il répond parfaitement à la clause `WHERE` le gain est énorme.

Requête 9

```
SELECT u.displayname AS name, max(c.creationdate) AS date_last_comment
FROM comments c JOIN users u ON (c.userid = u.id)
WHERE userid = 664306 GROUP BY 1;
```

Affichons le plan :

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
GroupAggregate  (cost=228940.44..228942.43 rows=1 width=19) (actual time=4169.577..4169.590 rows=1 loops=1)
   Group Key: u.displayname
   ->  Sort  (cost=228940.44..228941.10 rows=263 width=19) (actual time=4169.497..4169.509 rows=6 loops=1)
         Sort Key: u.displayname
         Sort Method: quicksort  Memory: 25kB
         ->  Nested Loop  (cost=0.42..228929.87 rows=263 width=19) (actual time=3098.829..4169.421 rows=6 loops=1)
```

```
              ->  Index Scan using users_pkey on users u  (cost=0.42..8.44 rows=1 width=19) (actual time=2.006..2.014 rows=1 loops=1)
                    Index Cond: (id = 664306)
              ->  Seq Scan on comments c  (cost=0.00..228918.80 rows=263 width=16) (actual time=3096.800..4167.373 rows=6 loops=1)
                    Filter: (userid = 664306)
                    Rows Removed by Filter: 5745561
 Planning Time: 10.318 ms
 Execution Time: 4169.774 ms
```

Le principal problème ici vient du `Seq Scan` réalisé sur la table `comments` pour répondre à la clause WHERE. Seulement 6 lignes sont retournées et 5745561 sont supprimées par le clause WHERE.

Reprenons la description de la table `comments` :

```
                                        Table « public.comments »
   Colonne    |            Type             | Collationnement | NULL-able |          Par défaut
--------------+-----------------------------+-----------------+-----------+------------------------------
 id           | bigint                      |                 | not null  | generated always as identity
 postid       | bigint                      |                 | not null  |
 text         | text                        |                 |           |
 creationdate | timestamp without time zone |                 | not null  |
 userid       | bigint                      |                 | not null  |
 jsonfield    | json                        |                 |           |
Index :
    "comments_pkey" PRIMARY KEY, btree (id)
    "cmnts_postid_idx" hash (postid) WITH (fillfactor='100')
    "comments_creationdate_idx" btree (creationdate) INVALID
Contraintes de clés étrangères :
    "fk_comments_postid" FOREIGN KEY (postid) REFERENCES posts(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Aucun index n'est présent sur le champ `userid`. La source de nos lenteurs vient donc à priori de là. Testons la création de l'index suivant :

```sql
create index on comments (userid);
```

Le nouveau plan :

```
                                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=721.18..721.19 rows=1 width=19) (actual time=0.810..0.832 rows=1 loops=1)
   Group Key: u.displayname
   Batches: 1  Memory Usage: 24kB
   ->  Nested Loop  (cost=0.86..719.87 rows=263 width=19) (actual time=0.537..0.744 rows=6 loops=1)
         ->  Index Scan using users_pkey on users u  (cost=0.42..8.44 rows=1 width=19) (actual time=0.178..0.198 rows=1 loops=1)
               Index Cond: (id = 664306)
         ->  Index Scan using comments_userid_idx on comments c  (cost=0.43..708.80 rows=263 width=16) (actual time=0.271..0.451 rows=6 loops=1)
               Index Cond: (userid = 664306)
 Planning Time: 2.867 ms
 Execution Time: 1.210 ms
```

La clause WHERE bénéficie maintenant d'un index et le gain est sans appel.

Requête 10

```sql
SELECT DISTINCT location, age, displayname
FROM users
WHERE creationdate > '2021-01-01'
ORDER BY 1, 2;
```

Le plan d'exécution :

```
                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=13845.17..14792.60 rows=82410 width=32) (actual time=1203.924..1358.021 rows=86971 loops=1)
   ->  Sort  (cost=13845.17..14082.03 rows=94743 width=32) (actual time=1203.908..1317.814 rows=91387 loops=1)
         Sort Key: location, age, displayname
         Sort Method: external merge  Disk: 2192kB
         ->  Index Scan using user_created_at_idx on users  (cost=0.42..3745.84 rows=94743 width=32) (actual time=5.692..114.933 rows=91387 loops=1)
               Index Cond: (creationdate > '2021-01-01 00:00:00'::timestamp without time zone)
 Planning Time: 3.239 ms
 Execution Time: 1373.213 ms
```

Un index est bien utilisé pour la clause WHERE et il n'y a pas de jointure. La piste d'un index manquant est donc à écarter.

Affichons un plan plus détaillé avec la requête suivante :

```sql
explain (analyze,buffers) SELECT DISTINCT location, age, displayname
FROM users
WHERE creationdate > '2021-01-01'
ORDER BY 1, 2;
```

Le plan avec l'option `buffers` :

```
                                                              QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=13845.17..14792.60 rows=82410 width=32) (actual time=1166.625..1331.309 rows=86971 loops=1)
   Buffers: shared hit=2132, temp read=274 written=275
   ->  Sort  (cost=13845.17..14082.03 rows=94743 width=32) (actual time=1166.620..1286.378 rows=91387 loops=1)
         Sort Key: location, age, displayname
         Sort Method: external merge  Disk: 2192kB
         Buffers: shared hit=2132, temp read=274 written=275
         ->  Index Scan using user_created_at_idx on users  (cost=0.42..3745.84 rows=94743 width=32) (actual time=0.101..64.209 rows=91387 loops=1)
               Index Cond: (creationdate > '2021-01-01 00:00:00'::timestamp without time zone)
               Buffers: shared hit=2132
 Planning Time: 0.446 ms
 Execution Time: 1342.109 ms
```

On peut constater que des données sont écrites sur disques : `Disk: 2192kB`. Cette écriture est provoquée par la clause ORDER BY de la requête.

Pour rappel, chaque connexion dispose d'une zone mémoire pour réaliser des opérations de tris et de hachages. S'il est nécessaire d'utiliser une quantité de mémoire supérieure à cette zone, le contenu de celle-ci est stocké sur disque pour permettre sa réutilisation.

La taille de cette mémoire est fixée par le paramètre `work_mem`.

Regardons la taille de ce paramètre :

```
stack=# show work_mem;
 work_mem
----------
 4MB
```

Modifions la taille du `work_mem` à `10MB` :

```sql
set work_mem = '10MB';
```

Affichons le nouveau plan :

```
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=12009.51..12215.53 rows=82410 width=32) (actual time=266.499..279.899 rows=86971 loops=1)
   Sort Key: location, age
   Sort Method: quicksort  Memory: 7875kB
   Buffers: shared hit=2132
   ->  HashAggregate  (cost=4456.41..5280.51 rows=82410 width=32) (actual time=134.155..176.356 rows=86971 loops=1)
         Group Key: location, age, displayname
         Batches: 1  Memory Usage: 8209kB
         Buffers: shared hit=2132
         ->  Index Scan using user_created_at_idx on users  (cost=0.42..3745.84 rows=94743 width=32) (actual time=0.080..47.930 rows=91387 loops=1)
               Index Cond: (creationdate > '2021-01-01 00:00:00'::timestamp without time zone)
               Buffers: shared hit=2132
 Planning Time: 0.425 ms
 Execution Time: 292.186 ms
```

Les écritures sur disques ont disparu. Par ailleurs, le plan a changé, un nœud `HashAggregate` est apparu, suivi d'un nœud `Sort`, alors qu'on avait précédemment un `Sort` suivi d'un `Unique`.

Faites très attention à la consommation mémoire que peut engendrer le paramètre `work_mem` car cette quantité de mémoire peut-être allouée par connexion, voire par nœud de requête et par *worker* (pour les requêtes parallélisées) !

Requête 11

```sql
select * from users where upper(displayname) = 'FRANZ';
```

Le plan :

```
                                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------
Seq Scan on users  (cost=0.00..22098.40 rows=4115 width=186) (actual time=7.176..1192.570 rows=10 loops=1)
   Filter: (upper((displayname)::text) = 'FRANZ'::text)
   Rows Removed by Filter: 823083
 Planning Time: 0.309 ms
 Execution Time: 1192.728 ms
```

La requête réalise ici un `Seq Scan` et filtre énormement de lignes pour peu de résultats retournés. La piste d'un index manquant ou non utilisé est donc à prioriser.

Affichons la description de la table `users` :

```
                                          Table « public.users »
    Colonne     |            Type             | Collationnement | NULL-able |          Par défaut
----------------+-----------------------------+-----------------+-----------+------------------------------
 id             | bigint                      |                 | not null  | generated always as identity
 creationdate   | timestamp without time zone |                 | not null  |
```

```
 displayname    | character varying(100)      |                | not null  |
 lastaccessdate | timestamp without time zone |                |           |
 location       | text                        |                |           |
 aboutme        | text                        |                |           |
 age            | integer                     |                |           |
Index :
    "users_pkey" PRIMARY KEY, btree (id)
    "user_created_at_idx" btree (creationdate) WITH (fillfactor='100')
    "user_display_idx" hash (displayname) WITH (fillfactor='100')
Référencé par :
    TABLE "badges" CONSTRAINT "fk_badges_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "comments" CONSTRAINT "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posthistory" CONSTRAINT "fk_posthistory_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posts" CONSTRAINT "fk_posts_owneruserid" FOREIGN KEY (owneruserid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "votes" CONSTRAINT "fk_votes_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Un index est bien présent sur le champ `displayname` mais il n'est pas utilisé. Notre requête est très similaire à la requête 2. Comme la clause `WHERE` porte sur le résultat d'une fonction l'index n'est plus utilisable pour ce champ.

Pour résoudre ce problème, il va falloir créer un index fonctionnel qui index le résultat de la fonction en question :

```
create index on users (upper(displayname));
```

Affichons le nouveau plan :

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on users  (cost=88.32..7725.98 rows=4115 width=186) (actual time=0.304..0.501 rows=10 loops=1)
   Recheck Cond: (upper((displayname)::text) = 'FRANZ'::text)
   Heap Blocks: exact=10
   ->  Bitmap Index Scan on users_upper_idx  (cost=0.00..87.29 rows=4115 width=0) (actual time=0.274..0.280 rows=10 loops=1)
         Index Cond: (upper((displayname)::text) = 'FRANZ'::text)
 Planning Time: 1.282 ms
 Execution Time: 0.639 ms
```

L'index est cette fois-ci bien utilisé.

Requête 12

```
SELECT displayname AS name, location FROM users WHERE location LIKE 'Reims%';
```

Le plan :

```
                                            QUERY PLAN
---------------------------------------------------------------------------------------------------
 Seq Scan on users  (cost=0.00..20040.66 rows=9 width=28) (actual time=39.950..227.295 rows=4 loops=1)
   Filter: (location ~~ 'Reims%'::text)
   Rows Removed by Filter: 823089
 Planning Time: 0.365 ms
 Execution Time: 227.374 ms
```

Nous sommes ici dans un cas similaire à la requête 6. L'utilisation du mot clé `LIKE` dans la requête rend inutilisable un index utilisant la classe d'opérateur définit par défaut.

Voyons la définition de la table `users` :

```
                                        Table « public.users »
    Colonne     |            Type             | Collationnement | NULL-able |          Par défaut
----------------+-----------------------------+-----------------+-----------+------------------------------
 id             | bigint                      |                 | not null  | generated always as identity
 creationdate   | timestamp without time zone |                 | not null  |
 displayname    | character varying(100)      |                 | not null  |
 lastaccessdate | timestamp without time zone |                 |           |
 location       | text                        |                 |           |
 aboutme        | text                        |                 |           |
 age            | integer                     |                 |           |
Index :
    "users_pkey" PRIMARY KEY, btree (id)
    "user_created_at_idx" btree (creationdate) WITH (fillfactor='100')
    "user_display_idx" hash (displayname) WITH (fillfactor='100')
Référencé par :
    TABLE "badges" CONSTRAINT "fk_badges_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "comments" CONSTRAINT "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posthistory" CONSTRAINT "fk_posthistory_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posts" CONSTRAINT "fk_posts_owneruserid" FOREIGN KEY (owneruserid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "votes" CONSTRAINT "fk_votes_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Aucun index n'est présent sur le champ `location`. Il faut donc en créer un mais en modifiant la classe d'opérateur pour qu'il puisse être utilisé par la clause `LIKE`.

```sql
create index on users (location text_pattern_ops);
```

Le nouveau plan :

```
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Index Scan using users_location_idx on users  (cost=0.42..8.45 rows=9 width=28) (actual time=0.146..0.220 rows=4 loops=1)
   Index Cond: ((location ~>=~ 'Reims'::text) AND (location ~<~ 'Reimt'::text))
   Filter: (location ~~ 'Reims%'::text)
 Planning Time: 0.906 ms
 Execution Time: 0.297 ms
```

L'index est bien utilisé par la clause `LIKE`.

Requête 13

```sql
select * from users where age * age > 2500;
```

Le plan :

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on users  (cost=0.00..22098.40 rows=274364 width=186) (actual time=265.790..265.811 rows=28 loops=1)
   Filter: ((age * age) > 2500)
   Rows Removed by Filter: 823065
 Planning Time: 0.172 ms
 Execution Time: 265.854 ms
```

Aucun index n'est utilisé ici. Affichons la description de la table `users` :

```
                                        Table « public.users »
    Colonne     |            Type             | Collationnement | NULL-able |          Par défaut
----------------+-----------------------------+-----------------+-----------+------------------------------
 id             | bigint                      |                 | not null  | generated always as identity
 creationdate   | timestamp without time zone |                 | not null  |
 displayname    | character varying(100)      |                 | not null  |
 lastaccessdate | timestamp without time zone |                 |           |
 location       | text                        |                 |           |
 aboutme        | text                        |                 |           |
 age            | integer                     |                 |           |
Index :
    "users_pkey" PRIMARY KEY, btree (id)
    "user_created_at_idx" btree (creationdate) WITH (fillfactor='100')
    "user_display_idx" hash (displayname) WITH (fillfactor='100')
Référencé par :
    TABLE "badges" CONSTRAINT "fk_badges_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "comments" CONSTRAINT "fk_comments_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posthistory" CONSTRAINT "fk_posthistory_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "posts" CONSTRAINT "fk_posts_owneruserid" FOREIGN KEY (owneruserid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
    TABLE "votes" CONSTRAINT "fk_votes_userid" FOREIGN KEY (userid) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Aucun index sur le champ age. Comme pour les requêtes 2 et 11, la clause WHERE filtre sur le résultat d'une fonction. Il va donc falloir créer un index fonctionnel.

```sql
create index on users ((age*age));
```

Le nouveau plan :

```
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on users  (cost=3058.75..16926.21 rows=274364 width=186) (actual time=0.177..0.183 rows=28 loops=1)
   Recheck Cond: ((age * age) > 2500)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on users_expr_idx  (cost=0.00..2990.16 rows=274364 width=0) (actual time=0.157..0.157 rows=28 loops=1)
         Index Cond: ((age * age) > 2500)
 Planning Time: 1.042 ms
 Execution Time: 0.291 ms
```