DALIBO

L'expertise PostgreSQL

Index BRIN

Fonctionnement et usages possibles

Table des matières

Index BRIN

Photographie récupérée sur https://www.flickr.com/photos/hb1248/24864908014/ Prise par Heribert Bechen Licence CC BY-SA 2.0

1 Licence des slides

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

2 Auteur

- Adrien Nayrat
- Consultant PostgreSQL chez Dalibo
  - email : adrien.nayrat@dalibo.com
  - twitter : @Adrien_nayrat
  - blog : https://blog.anayrat.info/
- Contributeur : Guillaume Lelarge
- Hashtag de la journée : #PGDAY_fr

Ce document a été rédigé par Adrien Nayrat, consultant PostgreSQL chez Dalibo depuis juillet 2015.

Pour toute question regardant ce document, ou plus généralement, il est possible de le contacter, de préférence par mail.

3 Introduction

- Index BRIN
- Nouveauté de la version 9.5
  - Principe
  - Fonctionnement
  - Spécificités
  - Cas d'usage

Les index BRIN ont été développés par Alvaro Herrera, de la société 2ndQuadrant, qui a reçu un sponsoring de la communauté européenne pour le développement de fonctionnalités destinées spécifiquement au traitement de grosses volumétries. Les premières discussions sur ce type d'index datent de 2008 [1]. Après de nombreux travaux, et de non moins nombreuses discussions et recherches, les index BRIN ont été intégrés à la version 9.5.

Cette présentation a pour but de montrer le fonctionnement, ainsi que les avantages et inconvénients de ce type d'index. Des cas d'usages intéressants seront cités.

4 Principe d'un index

- Stocker l'emplacement d'enregistrements
  - réduire les opérations de lecture sur la table
- Tables volumineuses
  - index volumineux
  - coût : mémoire, disque, création/mise à jour de l'index

À la base, un index a pour but la localisation rapide d'un ensemble spécifique, et généralement restreint, d'enregistrements dans une table ou une vue matérialisée (ce qu'on appelle aussi une relation). Pour cela, un index stocke l'emplacement dans la relation de chaque valeur, avec la valeur associée. Une valeur identique peut être enregistrée plusieurs fois ou une seule fois, suivant l'algorithme utilisé. Mais de toute façon, plus une table est volumineuse, plus le nombre d'emplacements l'est aussi, et donc plus l'index est volumineux. Évidemment, plus l'index est volumineux, et plus l'utilisation des ressources (processus, mémoire, disque) est important.

1 Segment Exclusion et Minmax indexes

5 Différents types d'index

- B-Tree
  - par défaut
  - indexe toutes les valeurs, plusieurs fois en cas de valeurs identiques
- GIN
  - version 8.2
  - indexe toutes les valeurs une seule fois, même en cas de valeurs identiques

- GiST / SP-GiST
  - générique
  - recherche plein texte, PostGIS
- BRIN
  - version 9.5
- Hash
  - déconseillé
  - indexe la valeur hachée

La norme SQL ne décrit pas les index. Cet objet est lié à la recherche de performances et, de ce fait, plusieurs façons d'améliorer les performances peuvent exister.

L'index B-Tree est le plus simple. Toutes les valeurs sont indexées sous la forme d'un arbre. Cet arbre est balancé et trié, ce qui améliore fortement les recherches d'un groupe de valeurs contigues, mais aussi les opérations de tri. L'algorithme B-Tree présente néanmoins un inconvénient : une valeur présente plusieurs fois dans la relation apparaîtra plusieurs fois dans l'index. L'index est donc plus volumineux de ce fait, alors qu'il suffirait de n'enregistrer la valeur qu'une seule fois pour la liste d'emplacements.

C'est justement ce que propose l'index GIN, arrivant dans PostgreSQL avec la version 8.2. Là-aussi, toutes les valeurs sont indexées, sous la forme d'un arbre trié. Mais si une valeur est présente plusieurs fois dans la table, elle n'est présente qu'une seule fois dans l'index, et le champ de l'emplacement est remplacé par un tableau d'emplacement.

L'index GiST est un index généraliste, permettant d'indexer un peu tout et n'importe quoi. Il n'est pas le plus rapide pour des données scalaires, mais il est le seul disponible pour des données complexes, comme pour les types de données de la recherche plein texte ou ceux de la couche spatiale de PostgreSQL.

Les index Hash sont fortement déconseillées. Leur modification n'est pas tracée dans les journaux de transactions, ce qui est particulièrement gênant en cas de crash, de

sauvegardes PITR et de réplication.

6 Principe d'un index BRIN

- **B**lock **R**ange **IN**dex
- Résumé d'un ensemble de blocs
  - réduit la taille de l'index
  - permet d'exclure un ensemble de blocs lors d'une recherche
- Bénéficier d'avantages similaires au partitionnement
- Cible : fortes volumétries, Big data
  - index classiques volumineux
- Similaire aux *Storage Indexes* chez Oracle

Comme nous l'avons indiqué précédemment, un index B-Tree récupère toutes les valeurs, plusieurs fois si nécessaires. La conséquence est que plus une table est volumineuse, plus l'index B-Tree associé l'est aussi. Ceci va au détriment des performances, l'index devenant difficile à conserver en cache et de ce fait long à parcourir.

Le but de l'index BRIN est d'éviter cet effet. Pour cela, l'index n'enregistre pas toutes les valeurs, mais des intervalles de valeurs. Prenons comme exemple une colonne de type entier. Sur un bloc (de 8 Ko) d'une table, on peut avoir 300 lignes comprenant une colonne de type entier. Si on résume ce bloc à la valeur minimale et à la valeur maximale de la colonne, qu'on stocke uniquement ces deux valeurs dans l'index, avec le numéro du bloc, au lieu des 300 valeurs, on n'en aurait plus que deux, ce qui équivaudrait (de manière très naïve) à un index 150 fois plus petit. L'idée d'un index BRIN est là : résumer les valeurs par groupe de blocs. Quant à une recherche, l'intérêt de l'index est d'exclure les blocs dont les résumés ne correspondent pas aux valeurs recherchées. Les blocs restants sont à parcourir pour trouver les lignes correspondant réellement à la recherche. Comme les accès disques sont généralement plus coûteux que les décodages de lignes, les performances devraient être intéressantes, surtout sur les tables volumineuses.

D'une certaine façon, on peut comparer les index BRIN au partitionnement avec moins de contraintes pour la mise en place. Le partitionnement permet d'exclure des tables mais nécessite d'utiliser les notions d'héritage avec des tables parents et enfants. Il faut également mettre en place des triggers pour insérer les données dans les bonnes tables ainsi que des index dans chacune des tables enfants. Par ailleurs il n'est pas possible de mettre une contrainte d'unicité sur la table parent, il faudra utiliser des contraintes dans chaque table enfant afin d'éviter des doublons entre les tables.

Un index BRIN permettra d'exclure facilement des ensembles de blocs, il sera également possible de rajouter une contrainte d'unicité (avec un btree). En revanche, si on souhaite créer un index btree sur des portions de la table, le moteur devra

parcourir l'intégralité de la table pour la création de chaque index. Un axe optimisation serait d'utiliser l'index BRIN pour créer les index btree sur des portions de la table.

La cible principale de ce type d'index est donc les tables volumineuses où l'index B-tree sera défavorisé : une construction lente, une maintenance lourde, une recherche peu efficace.

Sources :

- http://www.oracle.com/technetwork/issue-archive/2011/11-may/o31exadata-354069.html
- https://axleproject2015.files.wordpress.com/2015/03/postgresql-performance-presentation-sfpgday2015.pdf
- https://axleproject.eu/2014/10/10/loading-tables-and-creating-b-tree-and-block-range-indexes/
- https://wiki.postgresql.org/wiki/Segment_Exclusion

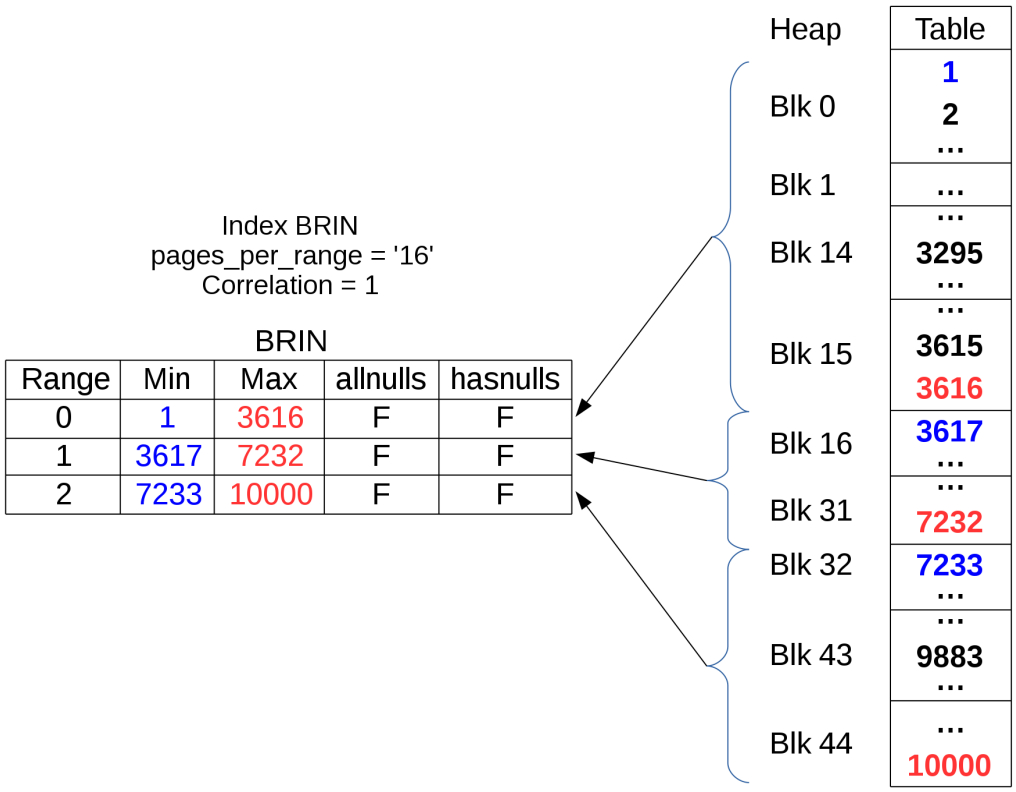
7 Fonctionnement

- Contient des résumés de groupe de blocs
  - un groupe de blocs est appelé un ***range***
- Stocke également 2 bits :
  - *allnulls* : tous les enregistrements d'un range sont *NULL*
  - *hasnulls* : le range contient au moins un enregistrement *NULL*

L'index ne contient pas des références à des enregistrements. Il référence des blocs, plus exactement des groupes de blocs. Pour rappel, un bloc dans PostgreSQL pèse 8 Ko. La table est découpée en groupe de 128 blocs par défaut (soit 1 Mo). Pour prendre un exemple, une table de 50 Go a 51200 enregistrements dans un index BRIN (à comparer avec un index B-tree qui contient autant d'enregistrements que de lignes), ce qui correspond à un fichier très petit.

Chaque enregistrement contient aussi deux bits de statut. Le premier indique si tous les enregistrements sont NULL, le second si au moins un est NULL. Si tous les enregistrements sont NULL on peut exclure un range si on cherche une valeur non NULL. En revanche un range peut contenir des valeurs NULL et non NULL, si on cherche une valeur NULL il faudra lire les ranges correspondant aux bits *allnulls* et *hasnulls*

8 Fo

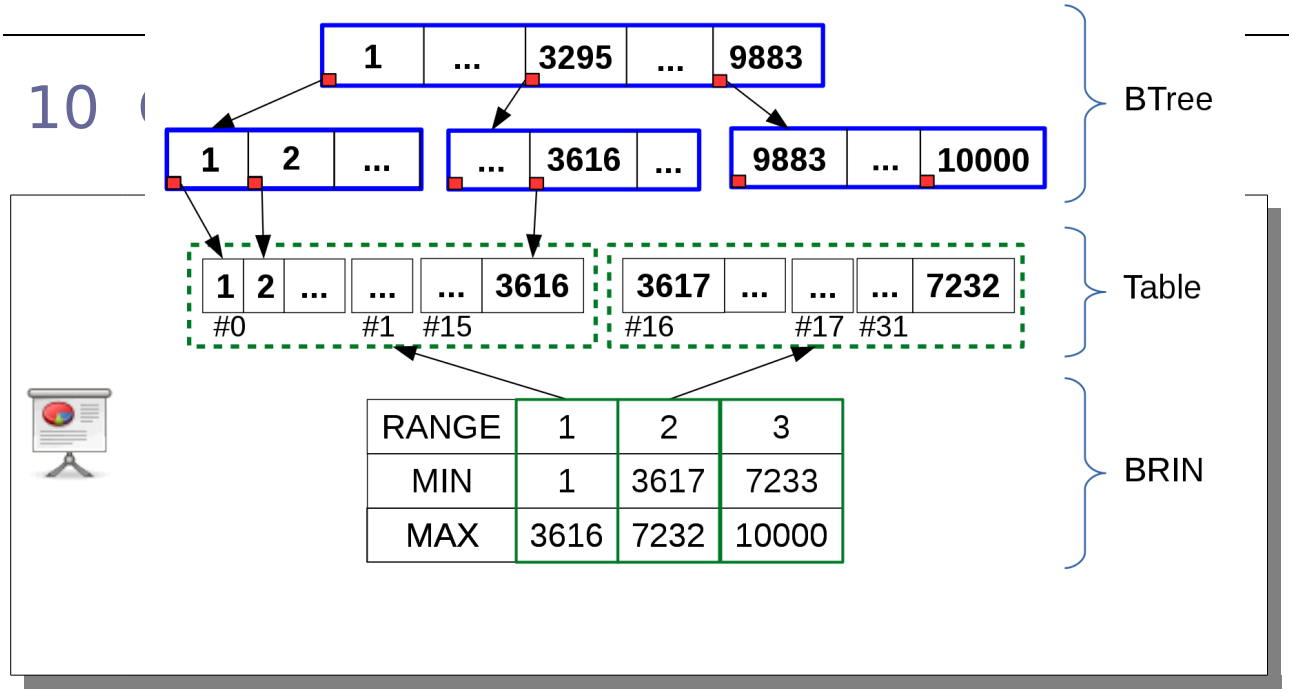


9 Fonctionnement

- Taille du *range* personnalisable
  - 128 blocs par défaut (1Mo)
- Range plus gros
  - index plus petit
  - création et maintenance plus efficaces
  - recherche plus longue (car plus de vérification)
- Parcours complet de l'index à chaque lecture

Le nombre de blocs d'un groupe est personnalisable, index par index, grâce au paramètre de stockage pages_per_range. Travailler par bloc permet d'obtenir un index plus petit. Plus le nombre de blocs par range est important, et plus l'index est petit. L'index est donc plus rapide à créer, et tient plus facilement en mémoire. Par contre, une recherche demandera à vérifier plus de blocs dans la table, ce qui sera préjudiciable aux performances de la recherche.

Il est important de noter qu'il est impossible de faire un parcours partiel de l'index. Un index BRIN sera toujours lu intégralement car il faut tester chaque résumé pour s'assurer si les valeurs recherchées appartiennent ou non aux résumés.



Voici un exemple un peu plus détaillé et plus proche de la réalité :

- la table contient 10000 valeurs ;
- l'index b-tree à un niveau ;
- l'index BRIN a été configuré avec un pages_per_range à 16 ;
- l'index b-tree contient 28 blocs alors que l'index BRIN rentre dans un seul bloc et il contient seulement 4 lignes.

L'exemple ci-dessous le montre.

On commence par créer une table, la peupler avec 10000 enregistrements. On finit en créant un index B-Tree :

```
CREATE TABLE t1 (c1 INT);
INSERT INTO t1 SELECT * FROM generate_series(1,10000);
CREATE INDEX ON t1 (c1);
```

En utilisant l'extension pageinspect, il est possible de décoder le bloc 0 de l'index B-tree pour récupérer les méta-données :

```
SELECT * FROM bt_metap('t1_c1_idx1');

 magic  | version | root | level | fastroot | fastlevel
--------+---------+------+-------+----------+-----------
 340322 |       2 |    3 |     1 |        3 |         1
```

La colonne root indique à quel bloc se trouve la racine de l'arbre. Ici, il s'agit du bloc 3. La colonne level nous précise qu'il n'y a qu'un seul niveau dans l'arbre.

Regardons maintenant les enregistrements du bloc 3 :

```
SELECT * FROM bt_page_items('t1_c1_idx1',3);

 itemoffset |  ctid   | itemlen | nulls | vars |          data
------------+---------+---------+-------+------+-------------------------
          1 |  (1,1)  |       8 | f     | f    |
          2 |  (2,1)  |      16 | f     | f    | 6f 01 00 00 00 00 00 00
          3 |  (4,1)  |      16 | f     | f    | dd 02 00 00 00 00 00 00
...
         28 | (29,1)  |      16 | f     | f    | 9b 26 00 00 00 00 00 00
```

La première ligne indique le bloc 1. La première valeur du bloc 2 est 367 (`6f 01`) alors que la première valeur du bloc 4 est 733 (`dd 02`). Il n'y a pas de bloc 3 car il s'agit de la racine.

Regardons ensuite le contenu du bloc 1 :

```
SELECT * FROM bt_page_items('t1_c1_idx1',1);

 itemoffset |  ctid   | itemlen | nulls | vars |          data
------------+---------+---------+-------+------+-------------------------
          1 | (1,141) |      16 | f     | f    | 6f 01 00 00 00 00 00 00
          2 | (0,1)   |      16 | f     | f    | 01 00 00 00 00 00 00 00
```

Les valeurs vont de de 1 à 367.

Et sur le bloc 2 :

```
itemoffset |  ctid   | itemlen | nulls | vars |          data
------------+---------+---------+-------+------+-------------------------
          1 | (3,55)  |      16 | f     | f    | dd 02 00 00 00 00 00 00
          2 | (1,141) |      16 | f     | f    | 6f 01 00 00 00 00 00 00
```

Les valeurs vont de 367 à 733.

Enfin, pour le bloc 29 :

```
SELECT * FROM bt_page_items('t1_c1_idx1',29);
 itemoffset |   ctid   | itemlen | nulls | vars |          data
------------+----------+---------+-------+------+-------------------------
          1 | (43,165) |      16 | f     | f    | 9b 26 00 00 00 00 00 00
...
        118 | (44,56)  |      16 | f     | f    | 10 27 00 00 00 00 00 00
```

Les valeurs vont de 9883 à 10000.

Créons maintenant un index BRIN avec un regroupement par ensemble de 16 blocs :

```
CREATE INDEX t1_brin_idx_16 ON t1 USING brin (c1) WITH (pages_per_range = 16);
```

En décodant le bloc 2 de l'index BRIN :

```
SELECT * FROM brin_page_items(get_raw_page('t1_brin_idx_16', 2),'t1_brin_idx_16');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |      value
```

```
------------+--------+--------+----------+----------+-------------+-----------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 3616}
          2 |     16 |      1 | f        | f        | f           | {3617 .. 7232}
          3 |     32 |      1 | f        | f        | f           | {7233 .. 10000}
```

Tous les enregistrements indexés tiennent sur un bloc. Ce bloc contient trois enregistrements. Le premier cible le premier ensemble de 16 blocs qui contient des valeurs allant de 1 à 3616, le deuxième cible le deuxième ensemble (valeurs de 3617 à 7232). Et le troisième les valeurs jusqu'à 10000.

11 Intérêt en lecture - 16 blocs

```
EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT c1 from brin_demo WHERE c1>10 AND c1<2000;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------
-----------------------------------------------
 Bitmap Heap Scan on public.brin_demo  (cost=32.08..504.47 rows=1959 width=4)
(actual time=0.062..0.656 rows=1989 loops=1)
   Output: c1
   Recheck Cond: ((brin_demo.c1 > 10) AND (brin_demo.c1 < 2000))
   Rows Removed by Index Recheck: 1627
   Heap Blocks: lossy=16
   Buffers: shared hit=18
   ->  Bitmap Index Scan on brin_demo_brin_idx  (cost=0.00..31.59 rows=1959 width=0)
(actual time=0.033..0.033 rows=160 loops=1)
         Index Cond: ((brin_demo.c1 > 10) AND (brin_demo.c1 < 2000))
         Buffers: shared hit=2
 Planning time: 0.237 ms
 Execution time: 0.796 ms
(11 rows)
```

Voici comment la table a été créée :

```
CREATE TABLE brin_demo (c1 INT);
INSERT INTO brin_demo SELECT * FROM generate_series(1,100000);
CREATE INDEX brin_demo_brin_idx ON brin_demo USING brin (c1) WITH (pages_per_range = 16);
```

2 blocs sont lus dans l'index, et 16 dans la table.

12 Intérêt en lecture - 128 blocs

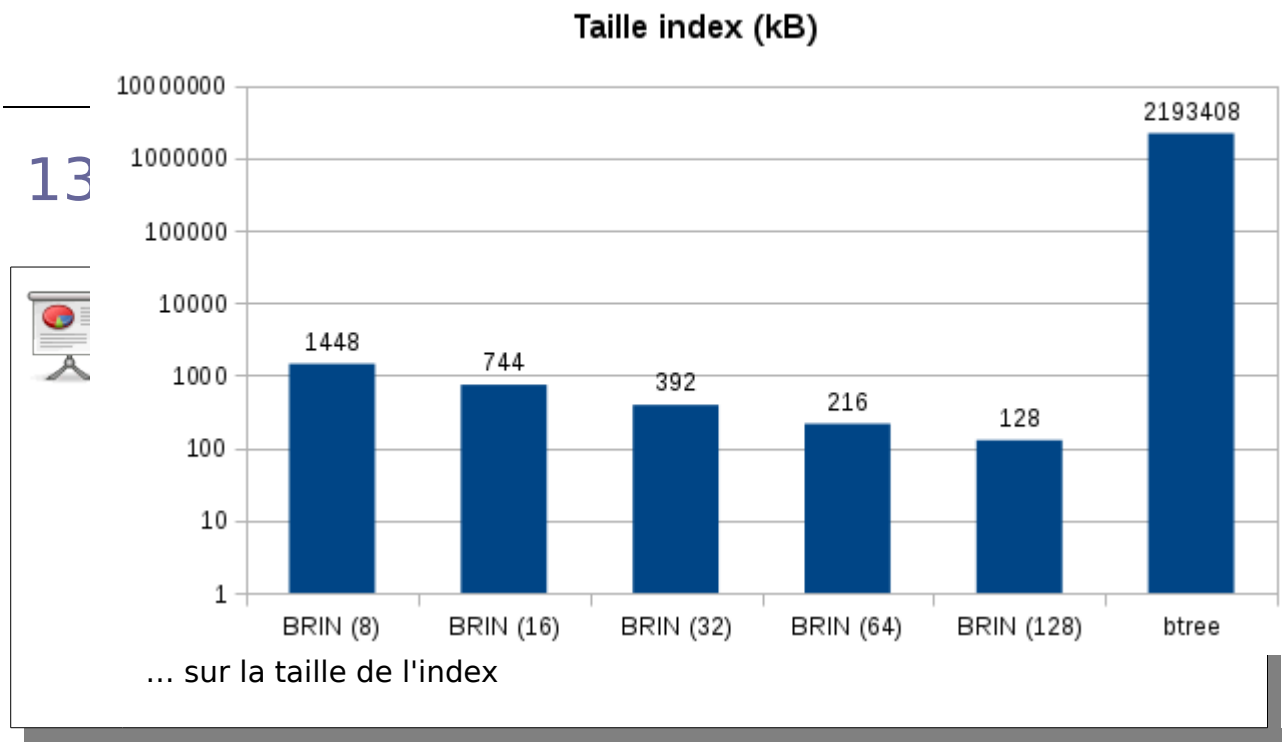
```
EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT c1 from brin_demo WHERE c1> 1 AND c1<2000;
                                                                 QUERY PLAN

-----------------------------------------------------------------------------------------
---------------------------------------------
 Bitmap Heap Scan on public.brin_demo  (cost=32.08..504.47 rows=1959 width=4)
(actual time=0.042..4.997 rows=1998 loops=1)
   Output: c1
   Recheck Cond: ((brin_demo.c1 > 1) AND (brin_demo.c1 < 2000))
   Rows Removed by Index Recheck: 26930
   Heap Blocks: lossy=128
   Buffers: shared hit=130
   ->  Bitmap Index Scan on brin_demo_brin_idx  (cost=0.00..31.59 rows=1959 width=0)
(actual time=0.027..0.027 rows=1280 loops=1)
         Index Cond: ((brin_demo.c1 > 1) AND (brin_demo.c1 < 2000))
         Buffers: shared hit=2
 Planning time: 0.064 ms
 Execution time: 5.134 ms
(11 rows)
```

Voici comment l'index a été re-créé :

```
DROP INDEX brin_demo_brin_idx;
CREATE INDEX brin_demo_brin_idx ON brin_demo USING brin (c1);
```

Cette fois, on a toujours 2 blocs lus dans l'index, mais 128 dans la table. En effet, dans les deux cas, tous les rensgitrements se trouvent dans un seul résumé. Le nombre de blocs lus dans la table dépend de la taille du résumé (16 dans le premier exemple, 128 dans le deuxième).

Une taille de range réduite permet d'avoir un index plus sélectif et de parcourir moins de blocs dans la table, le tout au prix d'un index plus volumineux.



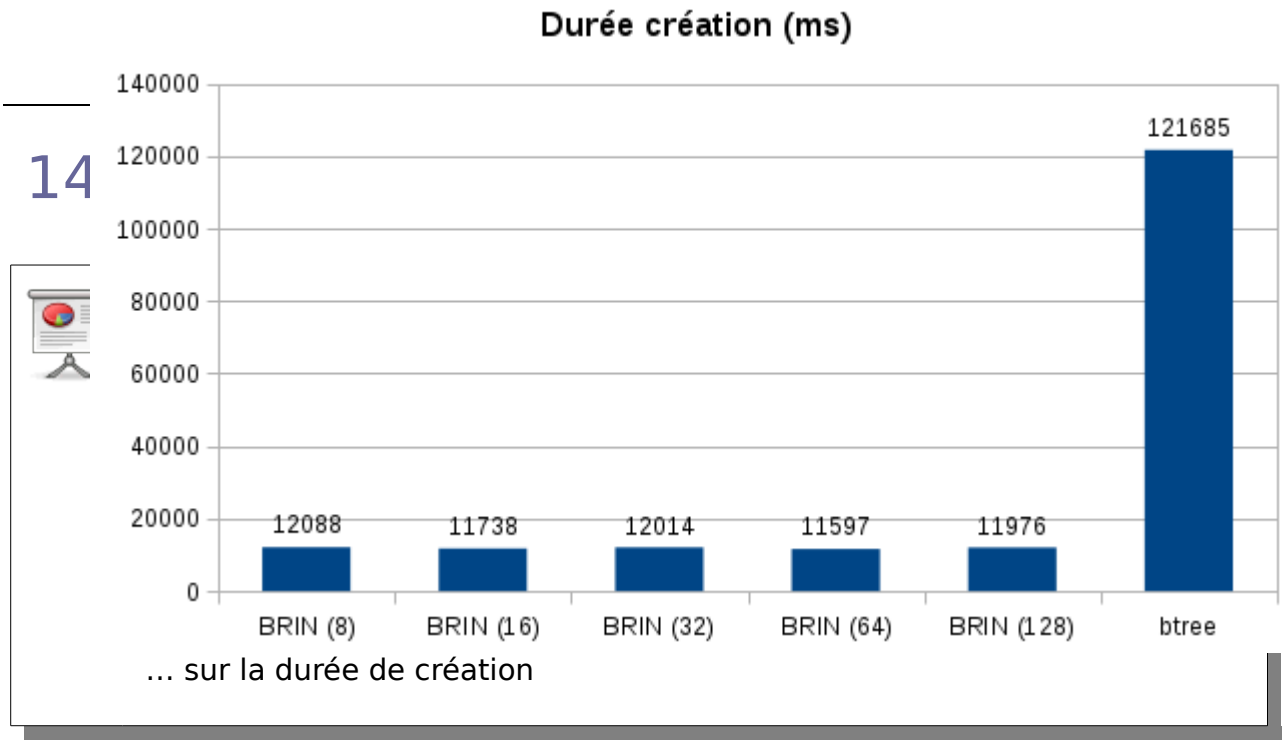
Ce graphique a été créé en utilisant le script de création d'objets ci-dessous :

```
CREATE TABLE brin_large (c1 INT);
INSERT INTO brin_large SELECT * FROM generate_series(1,100000000);
CREATE INDEX brin_large_brin_idx ON brin_large USING brin (c1);
CREATE INDEX brin_large_btree_idx ON brin_large USING btree (c1);
CREATE INDEX brin_large_brin_idx_8 ON brin_large USING brin (c1) WITH (pages_per_range = 8);
CREATE INDEX brin_large_brin_idx_16 ON brin_large USING brin (c1) WITH (pages_per_range = 16);
CREATE INDEX brin_large_brin_idx_32 ON brin_large USING brin (c1) WITH (pages_per_range = 32);
CREATE INDEX brin_large_brin_idx_64 ON brin_large USING brin (c1) WITH (pages_per_range = 64);
```

Et voici ce que donne la taille des objets (table et index) :

```
\dt+ brin_large
                    List of relations
 Schema |    Name     | Type  |  Owner   |  Size   | Description
--------+-------------+-------+----------+---------+-------------
 public | brin_large | table | postgres | 3458 MB |


\di+ brin_large*
                                  List of relations
 Schema |          Name           | Type  |  Owner   |   Table     |  Size   | Description
--------+-------------------------+-------+----------+-------------+---------+-------------
 public | brin_large_brin_idx     | index | postgres | brin_large | 128 kB  |
 public | brin_large_brin_idx_16 | index | postgres | brin_large | 744 kB  |
 public | brin_large_brin_idx_32 | index | postgres | brin_large | 392 kB  |
 public | brin_large_brin_idx_64 | index | postgres | brin_large | 216 kB  |
 public | brin_large_brin_idx_8  | index | postgres | brin_large | 1448 kB |
 public | brin_large_btree_idx    | index | postgres | brin_large | 2142 MB |
(6 rows)
```

14



... sur la durée de création

Voici de nouveau les ordres SQL de création des index, avec le chronométrage activé dans psql :

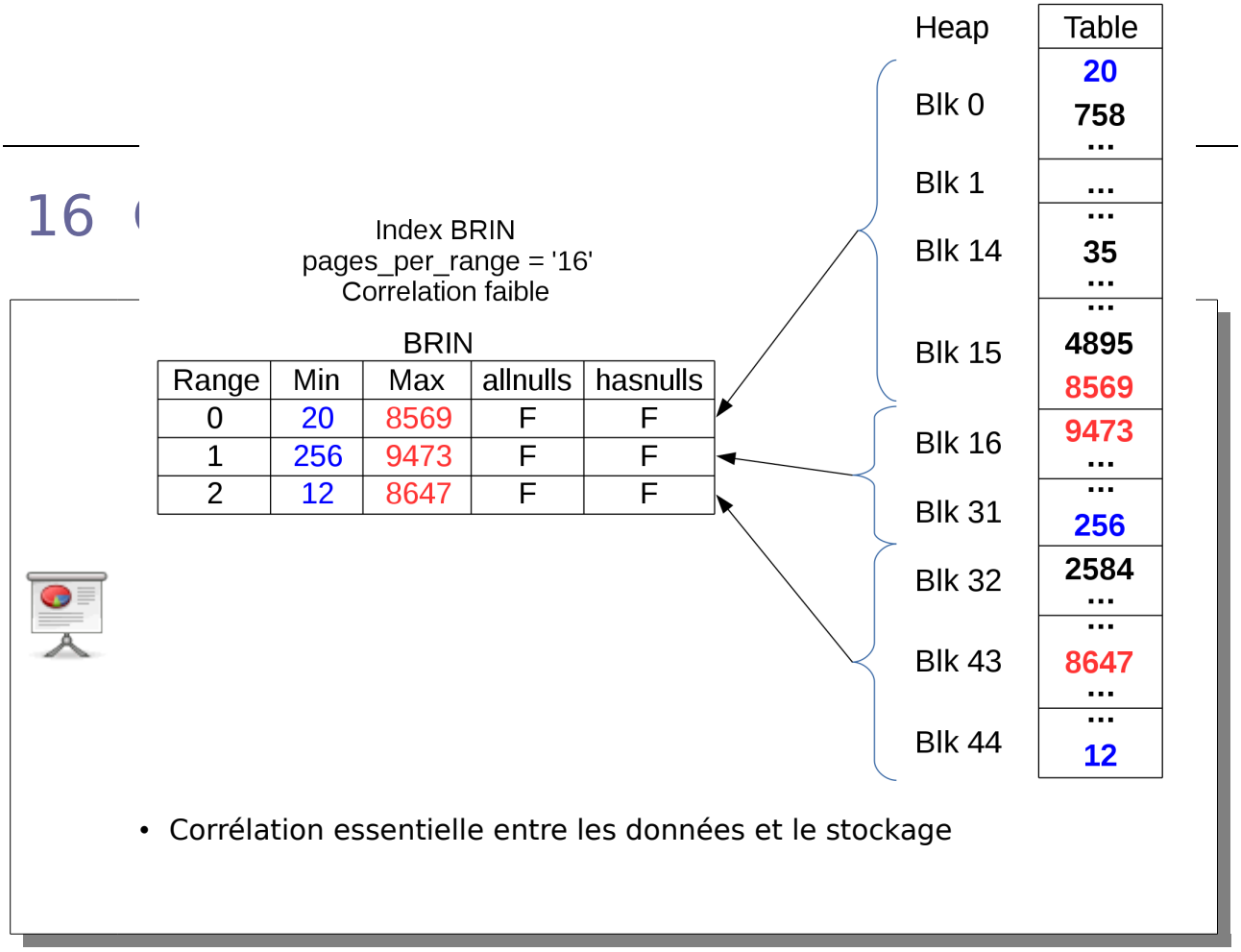
```
pgday=# CREATE INDEX brin_large_brin_idx ON brin_large USING brin (c1);
CREATE INDEX
Time: 11976.777 ms

pgday=# CREATE INDEX brin_large_btree_idx ON brin_large USING btree (c1);
CREATE INDEX
Time: 121685.474 ms

pgday=# CREATE INDEX brin_large_brin_idx_8 ON brin_large USING brin (c1) WITH (pages_per_range =
8);
CREATE INDEX
Time: 12087.568 ms

pgday=# CREATE INDEX brin_large_brin_idx_16 ON brin_large USING brin (c1) WITH (pages_per_range =
16);
CREATE INDEX
Time: 11738.022 ms

pgday=# CREATE INDEX brin_large_brin_idx_32 ON brin_large USING brin (c1) WITH (pages_per_range =
32);
CREATE INDEX
Time: 12013.699 ms

pgday=# CREATE INDEX brin_large_brin_idx_64 ON brin_large USING brin (c1) WITH (pages_per_range =
64);
CREATE INDEX
Time: 11597.166 ms
```

15 Classes d'opérateur

- Types qui peuvent être triés linéairement (pour obtenir min/max)
- Stocke les valeurs min et max pour les classes d'opérateurs `minmax`
  - int, numeric, text, bit, date ...
- Stocke l'enveloppe pour les classes d'opérateurs `inclusion`
  - box, inet ...
  - `bounding box` pour les types géométrique
  - masque réseau pour type inet
- Au total, 30 types supportés
  - extensible!

Nous n'avons pour l'instant évoqué que le cas simple du type integer. Mais un index BRIN accepte d'autres types de données. Tout dépend des classes d'opérateurs disponibles. Les classes `minmax` vont permettre de stocker les valeurs minimale et maximale d'une colonne particulière. Cela fonctionne pour les entiers, mais aussi pour les numériques, le texte, les dates... bref, tout ce qui est classable. Il existe aussi les classes d'inclusion. Cela concerne notamment les `bounding box` dans le cas des types géométriques, mais aussi les masques réseaux pour le type inet.

Au total, 30 types de données sont supportés (voir http://www.postgresql.org/docs/9.5/static/brin-builtin-opclasses.html pour la liste des types supportés), et il est toujours possible d'étendre le support de BRIN à d'autres types de données. Julien Rouhaud, Ronan Dunklau, Giuseppe Broccolo ont développé des classes d'opérateur pour deux types de données dans PostGIS, la couche spatiale de PostgreSQL.



- Corrélation essentielle entre les données et le stockage

L'exemple suivant se base sur des données réparties aléatoirement (donc sans corrélation entre valeur et emplacement physique).

```
CREATE TABLE brin_random (c1 INT);
INSERT INTO brin_random SELECT trunc(random() * 90 + 1) AS i FROM generate_series(1,100000);
CREATE INDEX brin_random_brin_idx_16 ON brin_random USING brin (c1) WITH (pages_per_range = 16);

SELECT pg_relation_size('brin_random')/(8*1024);
 ?COLUMN?
----------
      443
(1 ROW)

EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT c1 FROM brin_random WHERE c1> 10 AND c1<20;
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------
-----------------------------------
 Bitmap Heap Scan ON public.brin_random  (cost=17.12..488.71 ROWS=500 width=4) (actual
TIME=0.068..10.502 ROWS=10058 loops=1)
   Output: c1
   Recheck Cond: ((brin_random.c1 > 10) AND (brin_random.c1 < 20))
   ROWS Removed BY INDEX Recheck: 89942
   Heap Blocks: lossy=443
   Buffers: shared hit=445
   ->  Bitmap INDEX Scan ON brin_random_brin_idx_16  (cost=0.00..17.00 ROWS=500 width=0) (actual
TIME=0.053..0.053 ROWS=4480 loops=1)
         INDEX Cond: ((brin_random.c1 > 10) AND (brin_random.c1 < 20))
         Buffers: shared hit=2
 Planning TIME: 0.077 ms
 Execution TIME: 10.862 ms
(11 ROWS)
```

```
TIME: 11.233 ms
```

La ligne contenant `lossy=443` montre que 443 blocs ont été lus au niveau de la table. Il se trouve que la table contient exactement 443 blocs. Autrement dit, le moteur a parcouru toute la table. L'index n'a aucun intérêt dans ce cas.

La requête suivante permet de confirmer la faible corrélation de la colonne c1 :

```
SELECT tablename,correlation FROM pg_stats WHERE tablename='brin_random';
```

Actuellement, la corrélation n'est pas prise en compte dans le calcul du coût :

- http://doxygen.postgresql.org/index__selfuncs_8h.html#aa732367fc3b041ae0a0c5a377e2b1027
- http://www.postgresql.org/message-id/20151116135239.GV614468@alvherre.pgsql

Cet exemple le montre :

```
CREATE INDEX brin_random_btree_idx ON brin_random USING btree (c1);
CLUSTER brin_random USING brin_random_btree_idx;
ANALYZE brin_random ;

SELECT tablename,correlation FROM pg_stats WHERE tablename='brin_random';
  tablename   | correlation
--------------+-------------
 brin_random |           1
(1 ROW)

EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT c1 FROM brin_random WHERE c1> 10 AND c1<20;
                                                                 QUERY PLAN

-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------
 INDEX ONLY Scan USING brin_random_btree_idx ON public.brin_random  (cost=0.29..363.15 ROWS=10143
width=4) (actual TIME=0.021..2.036 ROWS=10058 loops=1)
   Output: c1
   INDEX Cond: ((brin_random.c1 > 10) AND (brin_random.c1 < 20))
   Heap Fetches: 10058
   Buffers: shared hit=45 READ=29
 Planning TIME: 0.166 ms
 Execution TIME: 2.468 ms
(7 ROWS)

TIME: 2.889 ms

DROP INDEX brin_random_btree_idx ;

EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT c1 FROM brin_random WHERE c1> 10 AND c1<20;
                                                             QUERY PLAN

-----------------------------------------------------------------------------------------------------------
--------------------------------------
 Bitmap Heap Scan ON public.brin_random  (cost=115.97..711.11 ROWS=10143 width=4) (actual
TIME=0.067..1.624 ROWS=10058 loops=1)
   Output: c1
   Recheck Cond: ((brin_random.c1 > 10) AND (brin_random.c1 < 20))
   ROWS Removed BY INDEX Recheck: 790
   Heap Blocks: lossy=48
   Buffers: shared hit=50
   ->  Bitmap INDEX Scan ON brin_random_brin_idx_16  (cost=0.00..113.43 ROWS=10143 width=0) (actual
TIME=0.026..0.026 ROWS=480 loops=1)
         INDEX Cond: ((brin_random.c1 > 10) AND (brin_random.c1 < 20))
```

```
        Buffers: shared hit=2
 Planning TIME: 0.104 ms
 Execution TIME: 2.048 ms
(11 ROWS)

TIME: 2.444 ms

SELECT * FROM brin_page_items(get_raw_page('brin_random_brin_idx_16', 2),
'brin_random_brin_idx_16');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   VALUE
------------+--------+--------+----------+----------+-------------+------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 4}
          2 |     16 |      1 | f        | f        | f           | {4 .. 7}
          3 |     32 |      1 | f        | f        | f           | {7 .. 10}
          4 |     48 |      1 | f        | f        | f           | {10 .. 13}
          5 |     64 |      1 | f        | f        | f           | {13 .. 17}
          6 |     80 |      1 | f        | f        | f           | {17 .. 20}
          7 |     96 |      1 | f        | f        | f           | {20 .. 23}
          8 |    112 |      1 | f        | f        | f           | {23 .. 27}
          9 |    128 |      1 | f        | f        | f           | {27 .. 30}
         10 |    144 |      1 | f        | f        | f           | {30 .. 33}
         11 |    160 |      1 | f        | f        | f           | {33 .. 36}
         12 |    176 |      1 | f        | f        | f           | {36 .. 40}
         13 |    192 |      1 | f        | f        | f           | {40 .. 43}
         14 |    208 |      1 | f        | f        | f           | {43 .. 46}
         15 |    224 |      1 | f        | f        | f           | {46 .. 49}
         16 |    240 |      1 | f        | f        | f           | {49 .. 53}
         17 |    256 |      1 | f        | f        | f           | {53 .. 56}
         18 |    272 |      1 | f        | f        | f           | {56 .. 59}
         19 |    288 |      1 | f        | f        | f           | {59 .. 62}
         20 |    304 |      1 | f        | f        | f           | {62 .. 66}
         21 |    320 |      1 | f        | f        | f           | {66 .. 69}
         22 |    336 |      1 | f        | f        | f           | {69 .. 72}
         23 |    352 |      1 | f        | f        | f           | {72 .. 76}
         24 |    368 |      1 | f        | f        | f           | {76 .. 79}
         25 |    384 |      1 | f        | f        | f           | {79 .. 82}
         26 |    400 |      1 | f        | f        | f           | {82 .. 85}
         27 |    416 |      1 | f        | f        | f           | {85 .. 88}
         28 |    432 |      1 | f        | f        | f           | {88 .. 90}
(28 ROWS)
```

Un CLUSTER permet de réorganiser la table afin qu'elle soit corrélée par rapport au stockage. Ceci met en évidence qu'avec une corrélation importante il y a moins de blocs parcourus.

17 Index multi-colonnes

- Multi-colonnes possible
- Colonnes corrélées avec l'emplacement physique

```
CREATE TABLE brin_multicol (id serial, val text);

INSERT INTO brin_multicol (val) SELECT md5(i::text) FROM generate_series(1,100000) i;
CREATE INDEX brin_multicol_brin_idx ON brin_multicol USING brin (id,val);
```

```
\dt+ brin_multicol
\di+ brin_multicol*

SELECT pg_relation_size('brin_multicol') /(8*1024);
 ?COLUMN?
----------
      834
(1 ROW)

SELECT * FROM brin_page_items(get_raw_page('brin_multicol_brin_idx', 2),'brin_multicol_brin_idx');

SELECT tablename,attname,correlation FROM pg_stats WHERE tablename='brin_multicol';
   tablename    | attname | correlation
----------------+---------+-------------
 brin_multicol | id      |           1
 brin_multicol | val     |   0.0050295
(2 ROWS)

EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT * FROM brin_multicol WHERE
val='a684eceee76fc522773286a895bc8436' AND id=10;
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------
--------------------------------
 Bitmap Heap Scan ON public.brin_multicol  (cost=12.00..16.02 ROWS=1 width=37) (actual
TIME=2.098..2.098 ROWS=0 loops=1)
   Output: id, val
   Recheck Cond: ((brin_multicol.id = 10) AND (brin_multicol.val =
'a684eceee76fc522773286a895bc8436'::text))
   ROWS Removed BY INDEX Recheck: 15360
   Heap Blocks: lossy=128
   Buffers: shared hit=136
   ->  Bitmap INDEX Scan ON brin_multicol_brin_idx  (cost=0.00..12.00 ROWS=1 width=0) (actual
TIME=0.056..0.056 ROWS=1280 loops=1)
         INDEX Cond: ((brin_multicol.id = 10) AND (brin_multicol.val =
'a684eceee76fc522773286a895bc8436'::text))
         Buffers: shared hit=8
 Planning TIME: 0.060 ms
 Execution TIME: 2.122 ms
(11 ROWS)
```

Seuls 128 blocs sont lus. En revanche si on filtre uniquement sur val (très faible corrélation), toute la table est parcourue :

```
EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT * FROM brin_multicol WHERE
val='a684eceee76fc522773286a895bc8436';
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------
--------------------------------
 Bitmap Heap Scan ON public.brin_multicol  (cost=12.01..16.02 ROWS=1 width=37) (actual
TIME=0.097..11.695 ROWS=1 loops=1)
   Output: id, val
   Recheck Cond: (brin_multicol.val = 'a684eceee76fc522773286a895bc8436'::text)
   ROWS Removed BY INDEX Recheck: 99999
   Heap Blocks: lossy=834
   Buffers: shared hit=836
   ->  Bitmap INDEX Scan ON brin_multicol_brin_idx  (cost=0.00..12.01 ROWS=1 width=0) (actual
TIME=0.078..0.078 ROWS=8960 loops=1)
         INDEX Cond: (brin_multicol.val = 'a684eceee76fc522773286a895bc8436'::text)
         Buffers: shared hit=2
 Planning TIME: 0.048 ms
 Execution TIME: 11.718 ms
(11 ROWS)
```

⇒ multicolonne utile si les colonnes sont corrélées avec leur emplacement (donc entres elles).

18 Cas d'utilisation

- Tables volumineuses
- Index classiques qui ne tiennent pas en RAM
- Corrélation entre les données et le stockage
- Tables avec peu de modifications
- Exemple : timeseries

19 Cas d'utilisation : performance

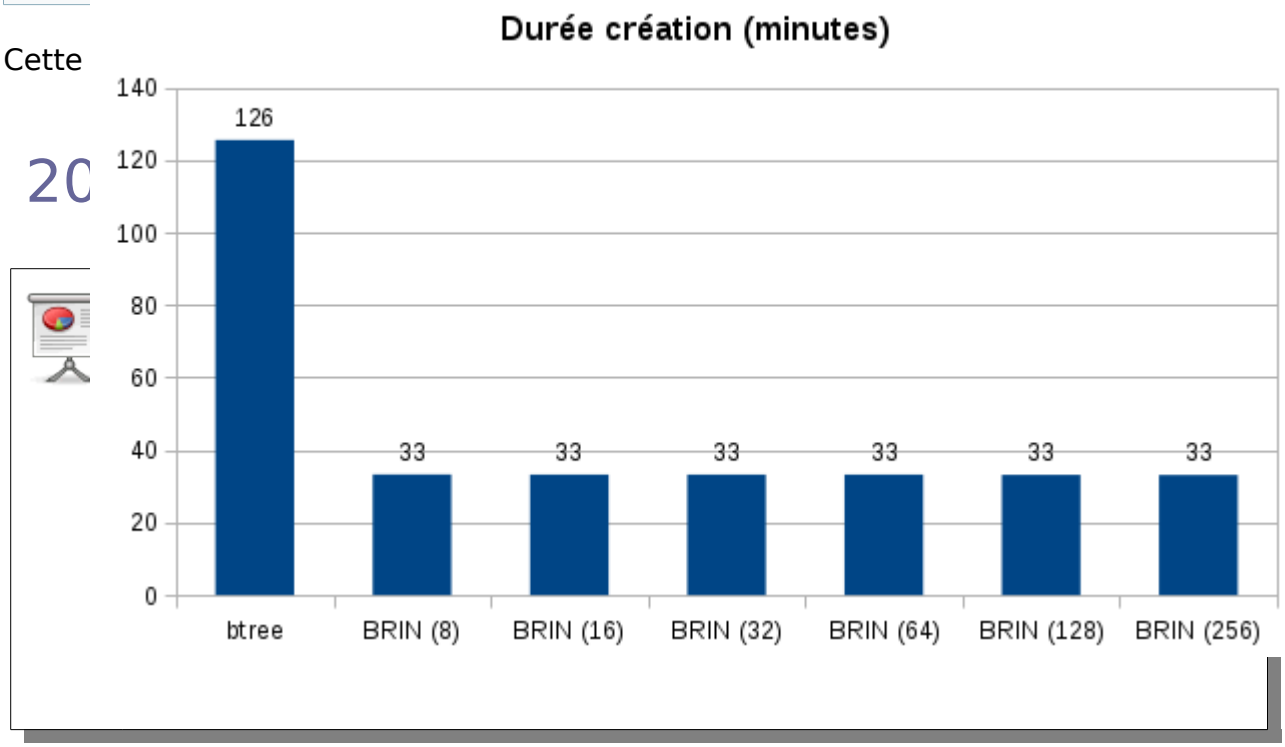
- Meilleures dans les cas où il faut lire beaucoup de blocs d'un index btree
- Exemple :
  - système de mesure avec 100 sondes et une mesure toutes les secondes
  - sur une année, 3 milliards de lignes (>150Go)
- Chiffres présentés indicatifs
  - menez vos propres tests avec vos données !

Pour donner un exemple, voici un script SQL de création et peuplement d'une table :

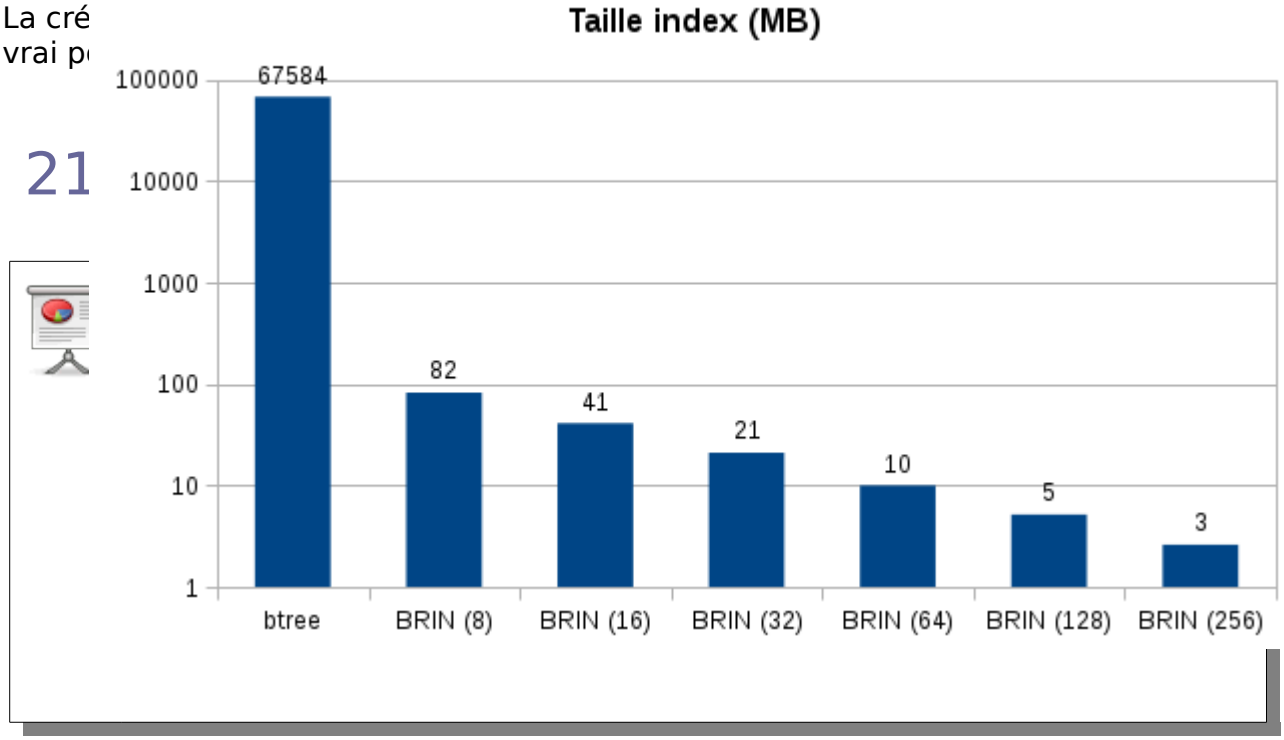
```
CREATE OR REPLACE FUNCTION random_string(LENGTH INTEGER) RETURNS text AS
$$
DECLARE
  chars text[] :=
'{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,a,b,c,d,e,f,g,h,i,j,k,l,m
,n,o,p,q,r,s,t,u,v,w,x,y,z}';
  RESULT text := '';
  i INTEGER := 0;
BEGIN
  IF LENGTH < 0 THEN
    RAISE EXCEPTION 'Given length cannot be less than 0';
  END IF;
  FOR i IN 1..LENGTH LOOP
    RESULT := RESULT || chars[1+random()*(array_length(chars, 1)-1)];
  END LOOP;
  RETURN RESULT;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE probe (id serial PRIMARY KEY, name text);
INSERT INTO probe (name ) SELECT random_string(5) FROM generate_series(1,100);

CREATE UNLOGGED TABLE DATA AS
  WITH generation AS (
    SELECT '2015-01-01'::TIMESTAMP + i * INTERVAL '1 second' AS date_metric,sonde::text,random() AS
metric
    FROM generate_series(0, 3600*24*365) i,
```

```
    LATERAL (SELECT name FROM probe) sonde)
SELEC
```

Cette



La cré
vrai p



Le script suivant crée un grand nombre d'index en changeant simplement la valeur du paramètre pages_per_range :

```
CREATE INDEX metro_btree_idx ON DATA USING btree (date_metric);
CREATE INDEX metro_brin_idx_8 ON DATA USING brin (date_metric) WITH (pages_per_range = 8);
CREATE INDEX metro_brin_idx_16 ON DATA USING brin (date_metric) WITH (pages_per_range = 16);
CREATE INDEX metro_brin_idx_32 ON DATA USING brin (date_metric) WITH (pages_per_range = 32);
CREATE INDEX metro_brin_idx_64 ON DATA USING brin (date_metric) WITH (pages_per_range = 64);
CREATE INDEX metro_brin_idx_128 ON DATA USING brin (date_metric); -- équivalent à un
pages_per_range = 128
CREATE INDEX metro_brin_idx_256 ON DATA USING brin (date_metric) WITH (pages_per_range = 256);
```

Le résultat en durée de création et en taille est indiqué dans le tableau ci-dessous :

Index	Durée création (minutes)	Taille (MB)
btree	126	67584
BRIN (8)	33	82
BRIN (16)	33	41
BRIN (32)	33	21
BRIN (64)	33	10
BRIN (128)	33	5
BRIN (256)	33	3

22 Cas d'utilisation : performance

- Quelques tests de lecture
  - 100 résultats (1 secondes de mesure)
  - 227 millions de résultats (1 mois de mesures)
  - 777 millions de résultats (3 mois de mesures)
  - 1.3 milliards de résultats (5 mois de mesures)

```
SELECT date_metric,sonde,metric
FROM DATA
WHERE date_metric = '2015-05-01 00:00:00'::TIMESTAMP;
```

Lignes	BRIN		Btree		Gain	
	Durée	Blocs lus	Durée	Blocs lus	Durée	Volume données lues
100	24 ms	697	0.06 ms	7	Btree (x400)	Btree (x100)
267 millions	170 s	13 Go	228 s	18 Go	BRIN (x1.3)	BRIN (x1.4)
777 millions	8 min	38 Go	11 min	54 Go	BRIN	BRIN (x1.4)

					(x1.37)	
1.3 milliard	13 min	63 Go	32 min (seqs-can) 18min	153 Go (se-qscan) 90 Go	BRIN (x2) vs seqscan BRIN (1.4x) vs Btree	BRIN (x2.4) vs seqscan BRIN (1.4x) vs Btree

Les index Btree et BRIN sont désactivés successivement pour comparer leurs durées d'exécution et le volume de données lues.

Chercher un faible nombre d'enregistrements : métriques sur une seconde ⇒ 100 lignes :

```
SELECT date_metric,sonde,metric FROM DATA WHERE date_metric = '2015-05-01 00:00:00'::TIMESTAMP;
```

- Sans les index btree :
  - le moteur choisit metro_brin_idx_256
  - durée d'exécution : 24.878 ms
  - blocs lus : shared hit=698
- Sans les index BRIN :
  - le moteur choisit metro_btree_idx
  - durée d'exécution : 0.061 ms
  - blocs lus : shared hit=7

Chercher un nombre important d'enregistrements : métriques sur 1 mois ⇒ 267 millions de lignes :

```
SELECT date_metric,sonde,metric FROM DATA WHERE date_metric BETWEEN '2015-05-01
00:00:00'::TIMESTAMP AND '2015-06-01 00:00:00'::TIMESTAMP;
```

- Sans les index btree :
  - le moteur choisit metro_brin_idx_256
  - durée d'exécution : 170 s
  - blocs lus : shared hit=115 read=1706567 (~13Go)
- Avec index btree :
  - le moteur choisit metro_btree_idx
  - durée d'exécution : 228 s
  - blocs lus : shared read=2437797 (~18Go)

Métriques sur 3 mois ⇒ 777 millions de lignes :

```
SELECT date_metric,sonde,metric FROM DATA WHERE date_metric BETWEEN '2015-01-01
00:00:00'::TIMESTAMP AND '2015-04-01 00:00:00'::TIMESTAMP;
```

- Sans les index btree :

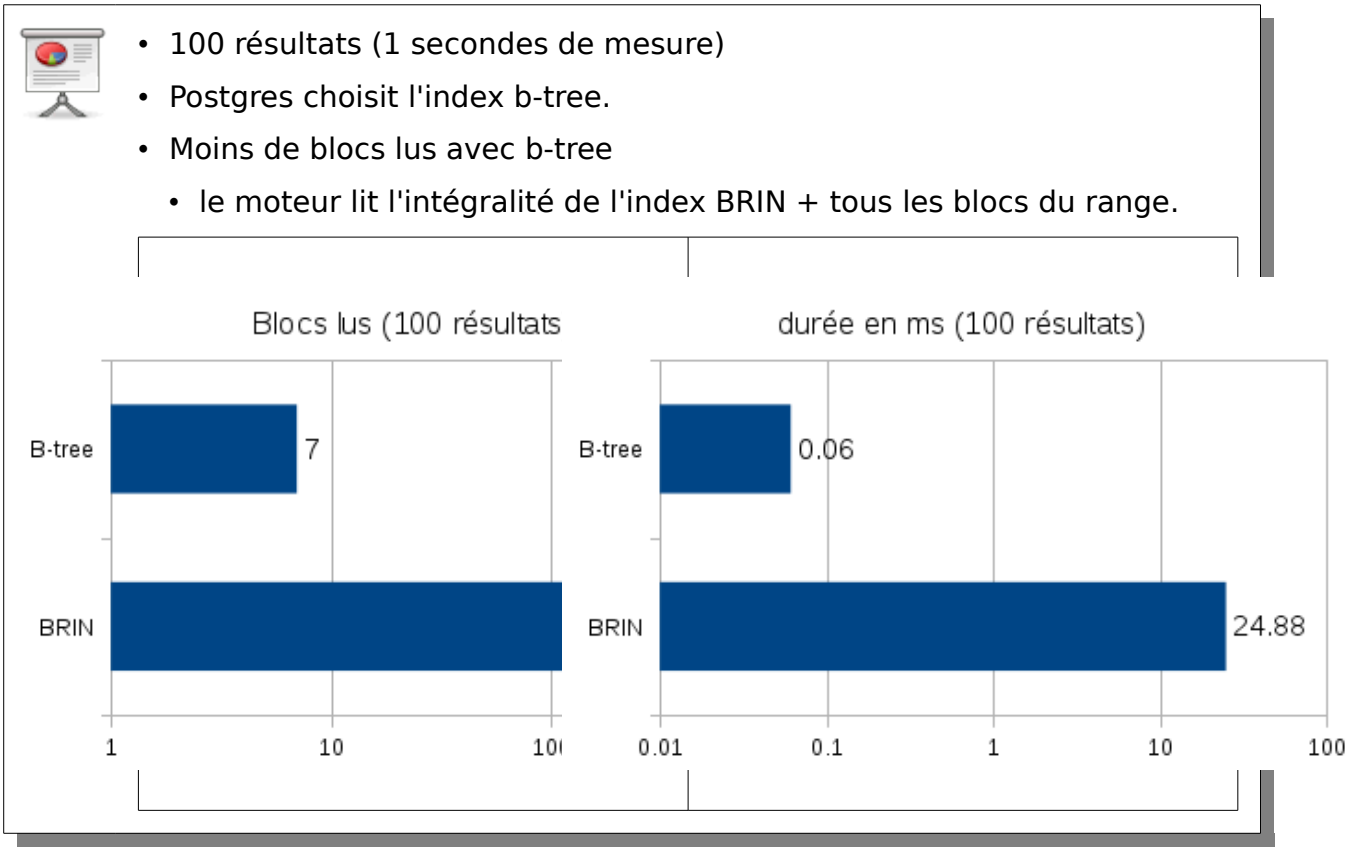
  - le moteur choisit metro_brin_idx_256
  - durée d'exécution : 8 minutes
  - blocs lus : shared hit=115 read=4953415 (~38Go)
- Sans les index BRIN :
  - le moteur choisit metro_btree_idx
  - durée d'exécution : 11 minutes
  - blocs lus : shared hit=4 read=7077457 (~54Go)

Métriques sur 5 mois ⇒ 1.3 milliard de lignes :

```
SELECT date_metric,sonde,metric FROM DATA WHERE date_metric BETWEEN '2015-01-01
00:00:00'::TIMESTAMP AND '2015-06-01 00:00:00'::TIMESTAMP;
```

- Sans les index btree :
  - le moteur choisit metro_brin_idx_256
  - durée d'exécution : 13 minutes
  - blocs lus : shared hit=116 read=8310342 (~63Go)
- Sans les index BRIN :
  - le moteur choisit seqscan (parcours de toute la table)
  - durée d'exécution : 32 minutes
  - blocs lus : shared hit=523766 (~4Go) read=19562859 (~149Go)
- Sans les index BRIN, seqscan désactivés :
  - le moteur choisit metro_btree_idx
  - durée d'exécution : 18 minutes
  - blocs lus : shared hit=4 read=11874400 (~90Go)

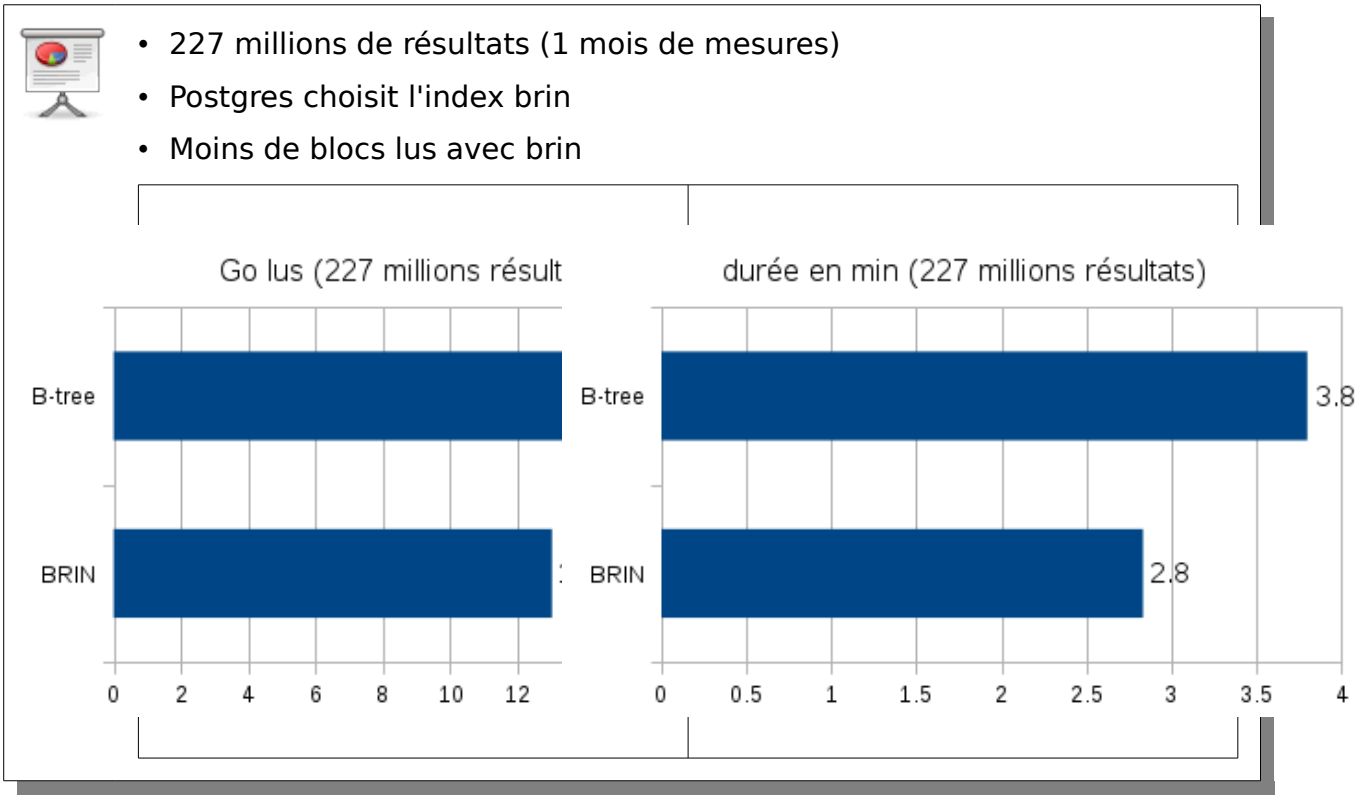
23 Cas d'utilisation : performance



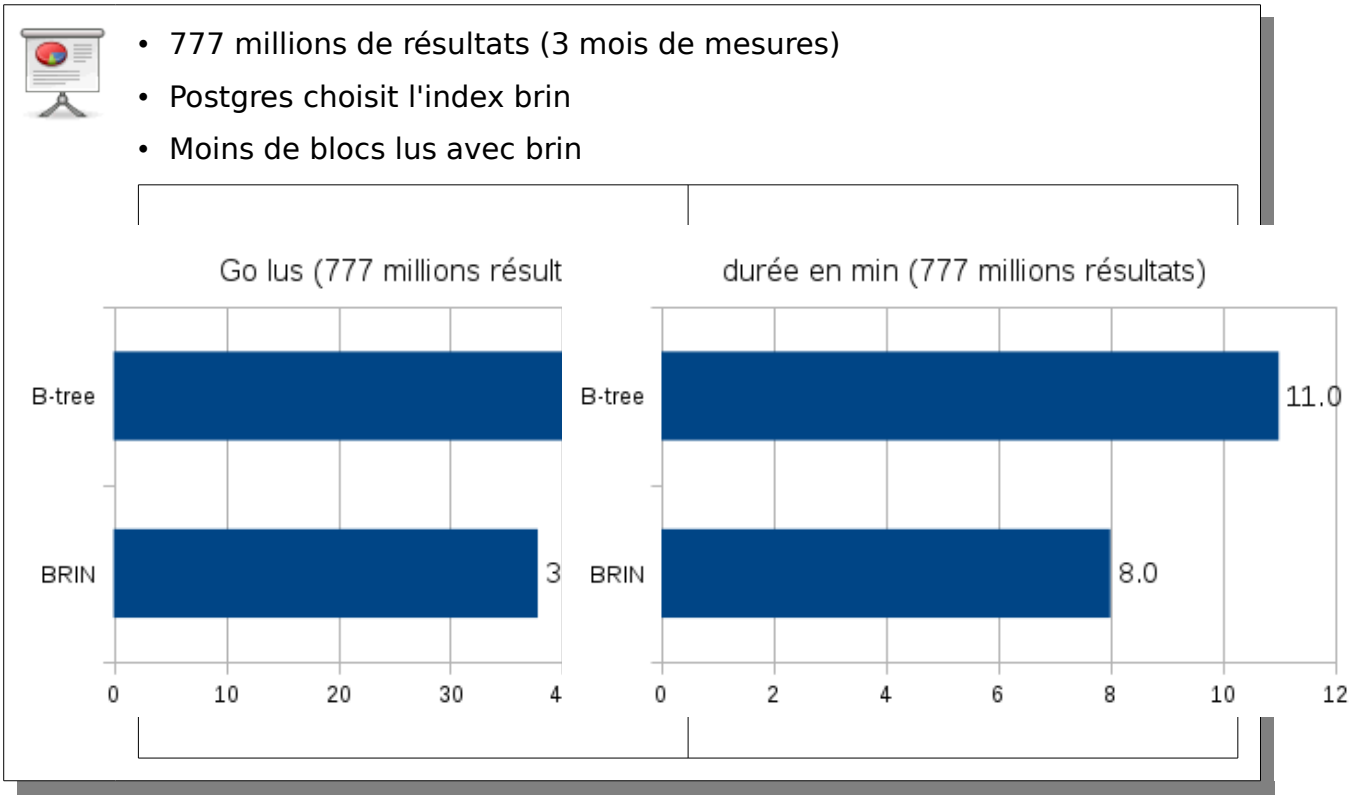
Dans ce cas, la lecture en utilisant l'index b-tree est bien plus rapide qu'avec l'index

BRIN. En effet, le moteur commence par lire l'intégralité de l'index BRIN avant d'aller lire l'intégralité des blocs correspondant au range.

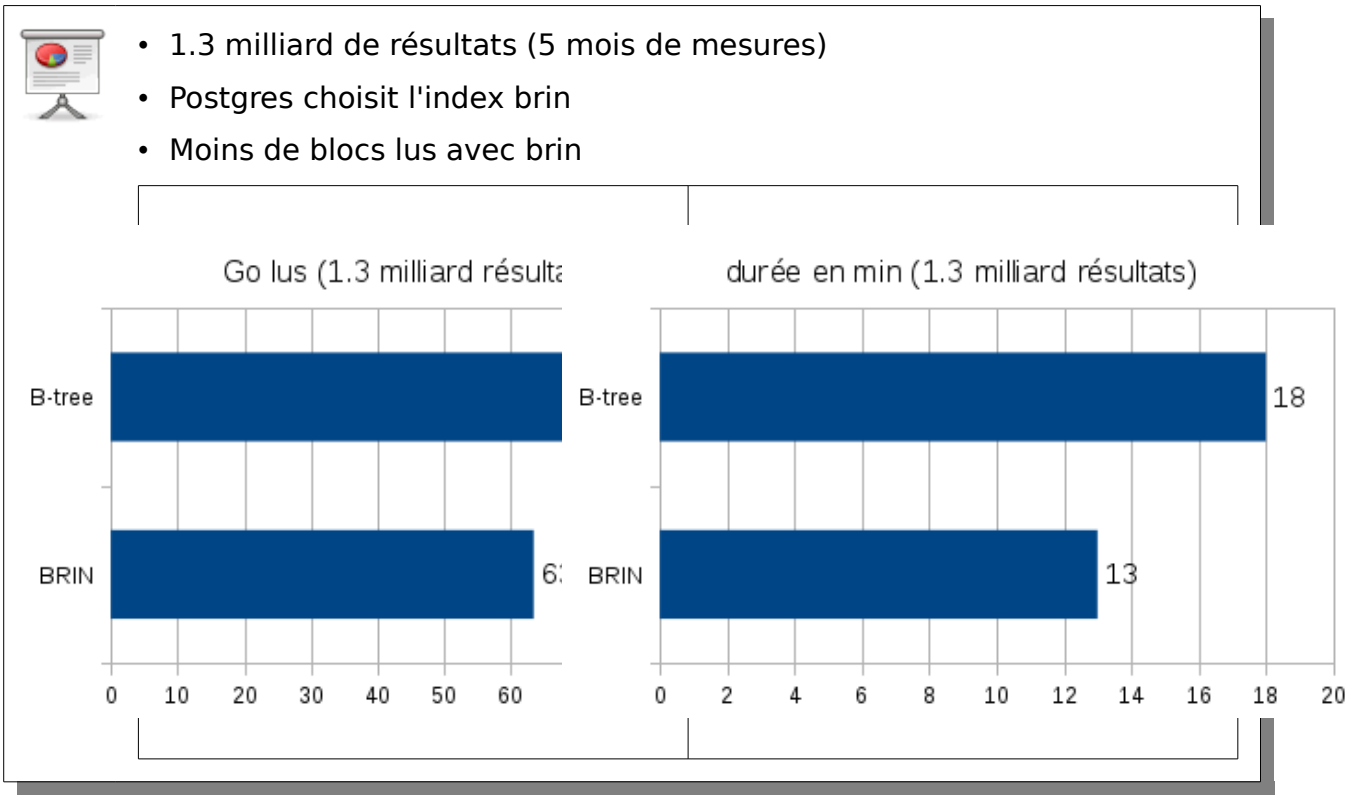
24 Cas d'utilisation : performance



25 Cas d'utilisation : performance



26 Cas d'utilisation : performance



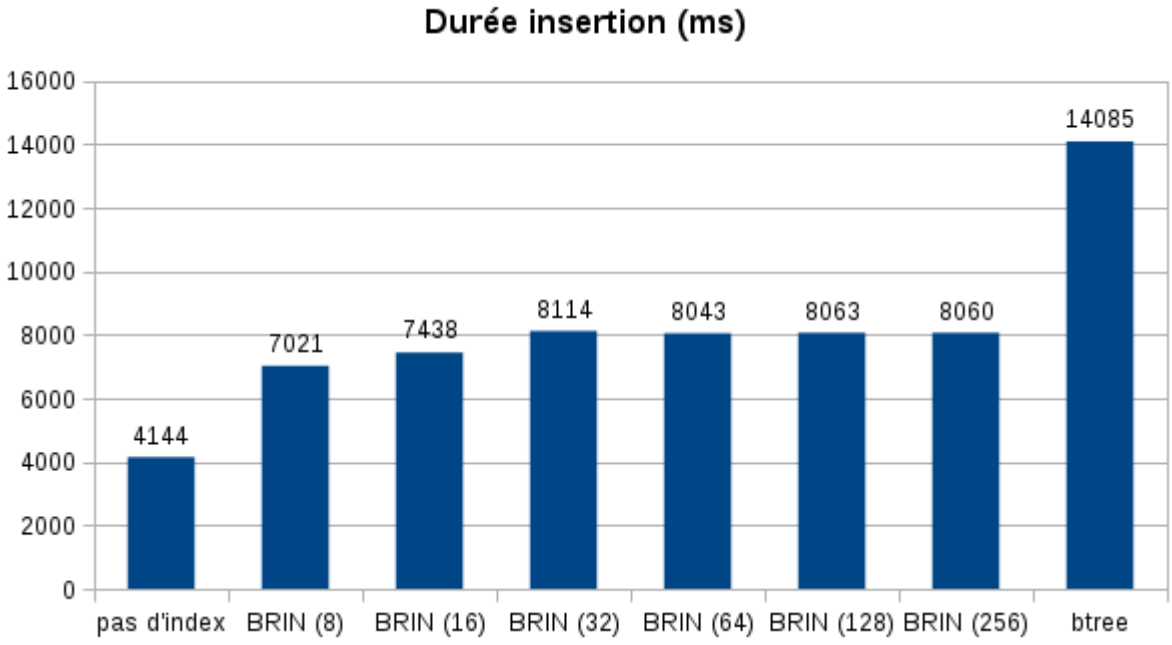
27 Cas d'utilisation : insertion

- Lors de modification, le moteur ne met à jour que les ranges déjà présents dans l'index
- Pour le mettre à jour :
  - VACUUM (ou autovacuum)
  - Fonction `brin_summarize_new_values()`
- Moins lourd à mettre à jour que les index B-Tree
- Exemple : insertion de 10 millions de lignes
- Chiffres indicatifs

Les chiffres sont simplement indicatifs. En effet, les tests ont été menés sur un PC portable. Les index et la base sont sur un seul et même disque.

Sur le
autov

28



Voici le script de mise à jour de l'index :

```
\timing
CREATE UNLOGGED TABLE brin_demo_2 (c1 INT);

INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);

TRUNCATE brin_demo_2;
```

```
CREATE INDEX brin_demo_2_brin_idx ON brin_demo_2 USING brin (c1);
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_brin_idx;
TRUNCATE brin_demo_2;

CREATE INDEX brin_demo_2_brin_idx ON brin_demo_2 USING brin (c1) WITH (pages_per_range = 256);
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_brin_idx;
TRUNCATE brin_demo_2;

CREATE INDEX brin_demo_2_brin_idx ON brin_demo_2 USING brin (c1) WITH (pages_per_range = 64);
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_brin_idx;
TRUNCATE brin_demo_2;

CREATE INDEX brin_demo_2_brin_idx ON brin_demo_2 USING brin (c1) WITH (pages_per_range = 32);
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_brin_idx;
TRUNCATE brin_demo_2;

CREATE INDEX brin_demo_2_brin_idx ON brin_demo_2 USING brin (c1) WITH (pages_per_range = 16);
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_brin_idx;
TRUNCATE brin_demo_2;

CREATE INDEX brin_demo_2_brin_idx ON brin_demo_2 USING brin (c1) WITH (pages_per_range = 8);
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_brin_idx;
TRUNCATE brin_demo_2;

CREATE INDEX brin_demo_2_btree_idx ON brin_demo_2 (c1) ;
INSERT INTO brin_demo_2 SELECT * FROM generate_series(1,10000000);
DROP INDEX brin_demo_2_btree_idx;
TRUNCATE brin_demo_2;
```

Et voici le tableau récapitulant les résultats :

Type d'index	Durée (ms)
Pas d'index	4144
BRIN 8	7021
BRIN 16	7438
BRIN 32	8114
BRIN 64	8043
BRIN 128	8063
BRIN 256	8060
BTREE	14085

29 Conclusion

- Index très compact
- Lecture de l'intégralité de l'index
- Plus lent en lecture qu'un index b-tree
  - sauf cas qui nécessitent de lire beaucoup de blocs dans l'index b-tree
- Surcoût pour les insertions
  - plus faible qu'avec un b-tree
- Utile sur les tables volumineuse avec peu de modification
  - application type *Data warehouse*
- Prérequis : Forte corrélation avec l'emplacement physique

BRIN permet d'avoir des index très petits, même pour des tables volumineuses. Un index BRIN est toujours parcouru complètement. Un petit index B-tree sera généralement plus performant, mais dès que l'index grossit, BRIN va montrer de meilleures performances, d'autant plus que les données sont corrélées. Les insertions sont aussi plus coûteuses avec un index BRIN. La niche principale des index BRIN reste les tables de grosses volumétries, ce qui fait qu'il est particulièrement indiqué pour des applications de type Data warehouse.

30 Questions ?

- C'est le moment !
- Sinon :
  - email : adrien.nayrat@dalibo.com
  - twitter : @Adrien_nayrat
  - blog : https://blog.anayrat.info/
- Présentation : http://bit.ly/1XWpnBh
- http://www.dalibo.com