Module J5

Indexation avancée

24.04

Table des matières

Sur ce document

Formation	Module J5
Titre	Indexation avancée
Révision	24.04
PDF	https://dali.bo/j5_pdf
EPUB	https://dali.bo/j5_epub
HTML	https://dali.bo/j5_html
Slides	https://dali.bo/j5_slides
TP	https://dali.bo/j5_tp
TP (solutions)	https://dali.bo/j5_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

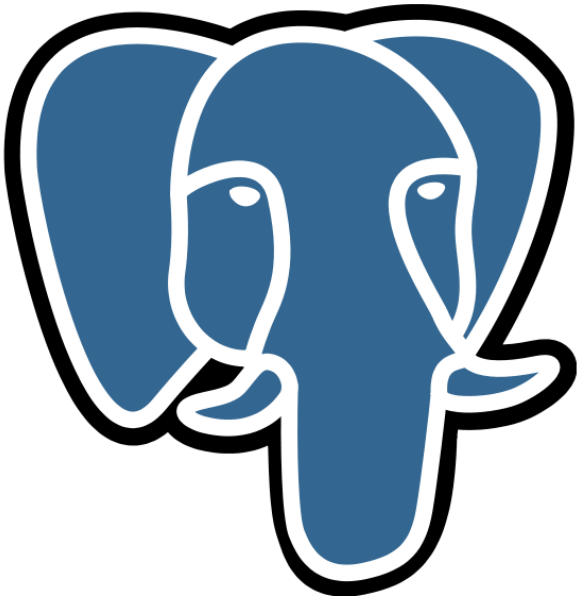
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Indexation avancée



1.1 INDEX AVANCÉS

De nombreuses fonctionnalités d'indexation sont disponibles dans PostgreSQL :

- Index B-tree
  - multicolonnes
  - fonctionnels
  - partiels
  - couvrants
- Classes d'opérateurs
- Indexation GIN, GiST, BRIN, hash, bloom
- Indexation de motifs
- Indexation multicolonne

PostgreSQL fournit de nombreux types d'index, afin de répondre à des problématiques de recherches complexes.

1.2 INDEX B-TREE (RAPPELS)

- Index B-tree fonctionnel, multicolonne, partiel, couvrant :
  - requête cible :

```
SELECT col4 FROM ma_table
WHERE col3<12 and f(col1)=7 and col2 LIKE 'toto%' ;
```

  - index dédié :

```
CREATE INDEX idx_adv ON ma_table (f(col1), col2 varchar_pattern_ops)
INCLUDE (col4) WHERE col3<12 ;
```

- Rappel : un index est coûteux à maintenir !

Rappelons que l'index classique est créé comme ceci :

```
CREATE INDEX mon_index ON ma_table(ma_colonne) ;
```

B-tree (en arbre équilibré) est le type d'index le plus fréquemment utilisé.

Toutes les fonctionnalités vues précédemment peuvent être utilisées simultanément. Il est parfois tentant de créer des index très spécialisés grâce à toutes ces fonctionnalités, comme dans l'exemple ci-dessus. Mais il ne faut surtout pas perdre de vue qu'un index est une structure lourde à mettre à jour, comparativement à une table. Une table avec un seul index est environ 3 fois plus lente qu'une table nue, et chaque index ajoute le même surcoût. Il est donc souvent plus judicieux d'avoir des index pouvant répondre à plusieurs requêtes différentes, et de ne pas trop les spécialiser. Il faut trouver un juste compromis entre le gain à la lecture et le surcoût à la mise à jour.

1.3 INDEX GIN

Un autre type d'index

- Définition
- Utilisation avec des données non structurées
- Utilisation avec des données scalaires
- Mise à jour

Les index B-tree sont les plus utilisés, mais PostgreSQL propose d'autres types d'index. Le type GIN est l'un des plus connus.

1.3.1 GIN : définition & données non structurées

GIN : *Generalized Inverted iNdex*

- Index inversé généralisé
  - les champs sont décomposés en éléments (par API)
  - l'index associe une valeur à la liste de ses adresses
- Pour chaque entrée du tableau
  - liste d'adresses où le trouver
- Utilisation principale : données non scalaires
  - tableaux, listes, non structurées…

Un index inversé est une structure classique, utilisée le plus souvent dans l'indexation *Full Text*. Le principe est de décomposer un document en sous-structures, et ce sont ces éléments qui seront indexées. Par exemple, un document sera décomposé en la liste de ses mots, et chaque mot sera une clé de l'index. Cette clé fournira la liste des documents contenant ce mot. C'est l'inverse d'un index B-tree classique, qui va lister chacune des occurrences d'une valeur et y associer sa localisation.

Par analogie : dans un livre de cuisine, un index classique permettrait de chercher « Crêpes au caramel à l'armagnac » et « Sauce caramel et beurre salé », alors qu'un index GIN contiendrait « caramel », « crêpes », « armagnac », « sauce », « beurre »…

Pour plus de détails sur la structure elle-même, cet article Wikipédia[1] est une lecture conseillée.

[1] https://fr.wikipedia.org/wiki/Index_invers%C3%A9

Dans l'implémentation de PostgreSQL, un index GIN est construit autour d'un index B-tree des éléments indexés, et à chacun est associé soit une simple liste de pointeurs vers la table (*posting list*) pour les petites listes, soit un pointeur vers un arbre B-tree contenant ces pointeurs (*posting tree*). La *pending list* est une optimisation des écritures (voir plus bas).

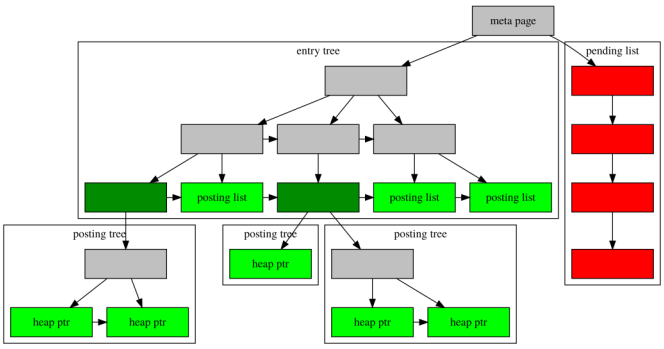


Figure 1/ .1: Schéma des index GIN (documentation officielle de PostgreSQL, licence PostgreSQL)

Les index GIN de PostgreSQL sont « généralisés », car ils sont capables d'indexer n'importe quel type de données, à partir du moment où on leur fournit les différentes fonctions d'API permettant le découpage et le stockage des différents *items* composant la donnée à indexer. En pratique, ce sont les opérateurs indiqués à la création de l'index, parfois implicites, qui contiendront la logique nécessaire.

Les index GIN sont des structures lentes à la mise à jour. Par contre, elles sont extrêmement efficaces pour les interrogations multicritères, ce qui les rend très appropriées pour l'indexation *Full Text*, des champs `jsonb`…

1.3.2 GIN et les tableaux

Quels tableaux contiennent une valeur donnée ?

- Opérateurs : `@>`, `<@`, `=`, `&&`, `?`, `?|`, `?&`

```
SELECT * FROM matable WHERE a @> ARRAY[42] ;

CREATE INDEX ON tablo USING gin (a);
```

- Champs concaténés : transformer en tableaux

```
SELECT * FROM voitures
WHERE regexp_split_to_array(caracteristiques,',')
          @> '{"toit ouvrant","climatisation"}'  ;

CREATE INDEX idx_attributs_array ON voitures
USING gin ( regexp_split_to_array(caracteristiques,',') ) ;
```

GIN et champ structuré :

Comme premier exemple d'indexation d'un champ structuré, prenons une table listant des voitures[2], dont un champ `caracteristiques` contient une liste d'attributs séparés par des virgules. Ceci ne respecte bien entendu pas la première forme normale.

Cette liste peut être transformée facilement en tableau avec `regexp_split_to_array`. Des opérateurs de manipulation peuvent alors être utilisés, comme : `@>` (contient), `<@` (contenu par), `&&` (a des éléments en communs). Par exemple, pour chercher les voitures possédant deux caractéristiques données, la requête est :

```
SELECT * FROM voitures
WHERE regexp_split_to_array(caracteristiques,',')
      @> '{"toit ouvrant","climatisation"}'    ;
```

```
 immatriculation | modele |                      caracteristiques
-----------------+--------+-------------------------------------------------------
 XB-025-PH       | clio   | toit ouvrant,climatisation
 RC-561-BI       | megane | regulateur de vitesse,boite automatique,
                 |        | toit ouvrant,climatisation,…
 LU-190-KO       | megane | toit ouvrant,climatisation,4 roues motrices
 SV-193-YR       | megane | climatisation,abs,toit ouvrant
 FG-432-FZ       | kangoo | climatisation,jantes aluminium,regulateur de vitesse,
                 |        | toit ouvrant
 …
```

avec ce plan :

```
                              QUERY PLAN
-----------------------------------------------------------------------------
Seq Scan on voitures  (cost=0.00..1406.20 rows=1 width=96)
  Filter: (regexp_split_to_array(caracteristiques, ','::text) @> '{"toit
↪  ouvrant",climatisation}'::text[])
```

Pour accélérer la recherche, le tableau de textes peut être directement indexé avec un index GIN, ici un index fonctionnel :

```
CREATE INDEX idx_attributs_array ON voitures
USING gin (regexp_split_to_array(caracteristiques,',')) ;
```

On ne précise pas d'opérateur, celui par défaut pour les tableaux convient.

Le plan devient :

```
 Bitmap Heap Scan on voitures  (cost=40.02..47.73 rows=2 width=96)
   Recheck Cond: (regexp_split_to_array(caracteristiques, ','::text) @> '{"toit
↪  ouvrant",climatisation}'::text[])
   ->  Bitmap Index Scan on idx_attributs_array  (cost=0.00..40.02 rows=2 width=0)
         Index Cond: (regexp_split_to_array(caracteristiques, ','::text) @> '{"toit
↪  ouvrant",climatisation}'::text[])
```

GIN et tableau :

GIN supporte nativement les tableaux des types scalaires (`int`, `float`, `text`, `date`…) :

[2] https://dali.bo/tp_voitures

```
CREATE TABLE tablo (i int, a int[]) ;
INSERT INTO tablo SELECT i, ARRAY[i, i+1] FROM generate_series(1,100000) i ;
```

Un index B-tree classique permet de rechercher un tableau identique à un autre, mais pas de chercher un tableau qui contient une valeur scalaire **à l'intérieur** du tableau :

```
EXPLAIN (COSTS OFF) SELECT * FROM tablo WHERE a = ARRAY[42,43] ;
```

```
                  QUERY PLAN
-----------------------------------------------
 Bitmap Heap Scan on tablo
   Recheck Cond: (a = '{42,43}'::integer[])
   ->  Bitmap Index Scan on tablo_a_idx
         Index Cond: (a = '{42,43}'::integer[])
```

```
SELECT * FROM tablo WHERE a @> ARRAY[42] ;
```

```
 i  |    a
----+---------
 41 | {41,42}
 42 | {42,43}
```

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT * FROM tablo WHERE a @> ARRAY[42] ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tablo (actual time=0.023..19.322 rows=2 loops=1)
   Filter: (a @> '{42}'::integer[])
   Rows Removed by Filter: 99998
   Buffers: shared hit=834
 Planning:
   Buffers: shared hit=6 dirtied=1
 Planning Time: 0.107 ms
 Execution Time: 19.337 ms
```

L'indexation GIN permet de chercher des valeurs figurant **à l'intérieur** des champs indexés :

```
CREATE INDEX ON tablo USING gin (a);
```

pour un résultat beaucoup plus efficace :

```
 Bitmap Heap Scan on tablo (actual time=0.010..0.010 rows=2 loops=1)
   Recheck Cond: (a @> '{42}'::integer[])
   Heap Blocks: exact=1
   Buffers: shared hit=5
   ->  Bitmap Index Scan on tablo_a_idx1 (actual time=0.007..0.007 rows=2 loops=1)
         Index Cond: (a @> '{42}'::integer[])
         Buffers: shared hit=4
 Planning:
   Buffers: shared hit=23
 Planning Time: 0.121 ms
 Execution Time: 0.023 ms
```

1.3.3 GIN pour les JSON et les textes

- JSON :
  - opérateur `jsonb_path_ops` ou `jsonb_ops`
- Indexation de trigrammes
  - extensions `pg_trgm` (opérateur dédié `gin_trgm_ops`)
- Recherche *Full Text*
  - indexe les vecteurs

Le principe est le même pour des JSON, s'ils sont bien stockés dans un champ de type `jsonb`. Les recherches de l'existence d'une clé à la racine du document ou tableau JSON sont réalisées avec les opérateurs `?`, `?|` et `?&`.

```
SELECT x, x ? 'b' AS "b existe", x ? 'c' AS "c existe"
FROM (VALUES ('{ "b" : { "c" : "ccc" }}'::jsonb)) AS F(x) ;
```

```
          x          | b existe | c existe
---------------------+----------+----------
 {"b": {"c": "ccc"}} | t        | f
```

Les recherches sur la présence d'une valeur dans un document JSON avec les opérateurs `@>`, `@?` ou `@@` peuvent être réalisées avec la classe d'opérateur par défaut (`json_ops`), mais aussi la classe d'opérateur `jsonb_path_ops`, donc au choix :

```
CREATE INDEX idx_prs ON personnes USING gin (proprietes jsonb_ops) ;

CREATE INDEX idx_prs ON personnes USING gin (proprietes jsonb_path_ops) ;
```

La classe `jsonb_path_ops` est plus performante pour ce genre de recherche et génère des index plus compacts lorsque les clés apparaissent fréquemment dans les données. Par contre, elle ne permet pas d'effectuer efficacement des recherches de structure JSON vide du type : `{ "a" : {} }`. Dans ce dernier cas, PostgreSQL devra faire, au mieux, un parcours de l'index complet, au pire un parcours séquentiel de la table. Le choix de la meilleure classe pour l'index dépend fortement de la typologie des données.

La documentation officielle[3] entre plus dans le détail.

L'extension `pg_trgm` utilise aussi les index GIN, pour permettre des recherches de type :

```
SELECT * FROM ma_table
WHERE ma_col_texte LIKE '%ma_chaine1%ma_chaine2%' ;
```

[3] https://docs.postgresql.fr/current/datatype-json.html#JSON-INDEXING

L'extension fournit un opérateur dédié pour l'indexation (voir plus loin).

La recherche *Full Text* est généralement couplée à un index GIN pour indexer les `tsvector` (voir le module T1[4]).

1.3.4 GIN & données scalaires

- Données scalaires aussi possibles
  - avec l'extension `btree_gin`
- GIN compresse quand les données se répètent
  - alternative à bitmap !

Grâce à l'extension `btree_gin`, fournie avec PostgreSQL, l'indexation GIN peut aussi s'appliquer aux scalaires. Il est ainsi possible d'indexer un ensemble de colonnes, par exemple d'entiers ou de textes (voir exemple plus loin). Cela peut servir dans les cas où une requête multicritères peut porter sur de nombreux champs différents, dont aucun n'est obligatoire. Un index B-tree est là moins adapté, voire inutilisable.

Un autre cas d'utilisation traditionnel est celui des index dits *bitmap*. Les index bitmap sont très compacts, mais ne permettent d'indexer que peu de valeurs différentes. Un index bitmap est une structure utilisant 1 bit par enregistrement pour chaque valeur indexable. Par exemple, on peut définir un index bitmap sur le sexe : deux valeurs seulement (voire quatre si on autorise NULL ou non-binaire) sont possibles. Indexer un enregistrement nécessitera donc un ou deux bits. Le défaut des index *bitmap* est que l'ajout de nouvelles valeurs est très peu performant car l'index nécessite d'importantes réécritures à chaque ajout. De plus, l'ajout de données provoque une dégradation des performances puisque la taille par enregistrement devient bien plus grosse.

Les index GIN permettent un fonctionnement sensiblement équivalent au *bitmap* : chaque valeur indexable contient la liste des enregistrements répondant au critère, et cette liste a l'intérêt d'être compressée. Par exemple, sur une base créée avec pgbench, de taille 100, avec les options par défaut :

```
CREATE EXTENSION btree_gin ;

CREATE INDEX pgbench_accounts_gin_idx on pgbench_accounts USING gin (bid);

EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT * FROM pgbench_accounts WHERE bid=5 ;

                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on pgbench_accounts (actual time=7.505..19.931 rows=100000 loops=1)
   Recheck Cond: (bid = 5)
```

[4] https://dali.bo/t1_html

```
   Heap Blocks: exact=1640
   Buffers: shared hit=1657
   ->  Bitmap Index Scan on pgbench_accounts_gin_idx (actual time=7.245..7.245
↪  rows=100000 loops=1)
         Index Cond: (bid = 5)
         Buffers: shared hit=17
 Planning:
   Buffers: shared hit=2
 Planning Time: 0.080 ms
 Execution Time: 24.090 ms
```

Dans ce cas précis qui renvoie de nombreuses lignes, l'utilisation du GIN est même aussi efficace que le B-tree ci-dessous, car l'index GIN est mieux compressé et a besoin de lire moins de blocs :

```
CREATE INDEX pgbench_accounts_btree_idx ON pgbench_accounts (bid);

EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT * FROM pgbench_accounts WHERE bid=5 ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_btree_idx on pgbench_accounts (actual
↪  time=0.008..19.138 rows=100000 loops=1)
   Index Cond: (bid = 5)
   Buffers: shared hit=1728
 Planning:
   Buffers: shared hit=18
 Planning Time: 0.117 ms
 Execution Time: 23.369 ms
```

L'index GIN est en effet environ 5 fois plus compact que le B-tree dans cet exemple simple, avec 100 valeurs distinctes de `bid` :

```
pgbench_300=# \di+ pgbench_accounts_*
                    Liste des relations
 Schéma |            Nom             | … | Méthode d'accès | Taille | Description
--------+----------------------------+---+-----------------+--------+-------------
 public | pgbench_accounts_bid_idx   | … | btree           | 66 MB  |
 public | pgbench_accounts_gin_idx   | … | gin             | 12 MB  |
```

Ce ne serait pas le cas avec des valeurs toutes différentes. (Avant la version 13, la différence de taille est encore plus importante car les index B-tree ne disposent pas encore de la déduplication des clés.)

Avant de déployer un index GIN, il faut vérifier l'impact du GIN sur les performances en insertions et mises à jour et l'impact sur les requêtes.

1.3.5 GIN : mise à jour

- Création lourde
  - `maintenance_work_mem` élevé
- Mise à jour lente
  - d'où l'option `fastupdate`
  - … à désactiver si temps de réponse instable !

Les index GIN sont lourds à créer et à mettre à jour. Une valeur élevée de `maintenance_work_mem` est conseillée.

L'option `fastupdate` permet une mise à jour bien plus rapide. Elle est activée par défaut. PostgreSQL stocke alors les mises à jour de l'index dans une *pending list* qui est intégrée en bloc, notamment lors d'un `VACUUM`. Sa taille peut être modifiée par le paramètre `gin_pending_list_limit`, par défaut à 4 Mo, et au besoin surchargeable sur chaque index.

L'inconvénient de cette technique est que le temps de réponse de l'index devient instable : certaines recherches peuvent être très rapides et d'autres très lentes. Le seul moyen d'accélérer ces recherches est de désactiver `fastupdate`. Cela permet en plus d'éviter la double écriture dans les journaux de transactions. Mais il y a un impact : les mises à jour de l'index sont bien plus lentes.

Il faut donc faire un choix. Si l'on conserve l'option `fastupdate`, il faut surveiller la fréquence de passage de l'autovacuum. L'appel manuel à la fonction `gin_clean_pending_list()` est une autre option.

Pour les détails, voir la documentation[5].

[5] https://www.postgresql.org/docs/current/static/gin-implementation.html#GIN-FAST-UPDATE%3E

1.4 INDEX GIST

GiST : *Generalized Search Tree*

- Arbre de recherche généralisé
- Indexation non plus des valeurs mais de la véracité de prédicats
- Moins performants que B-tree (moins sélectifs)
- Moins lourds que GIN

Initialement, les index GiST sont un produit de la recherche de l'université de Berkeley. L'idée fondamentale est de pouvoir indexer non plus les valeurs dans l'arbre B-tree, mais plutôt la véracité d'un prédicat : « *ce prédicat est vrai sur telle sous-branche* ». On dispose donc d'une API permettant au type de données d'informer le moteur GiST d'informations comme : « *quel est le résultat de la fusion de tel et tel prédicat* » (pour pouvoir déterminer le prédicat du nœud parent), quel est le surcoût d'ajout de tel prédicat dans telle ou telle partie de l'arbre, comment réaliser un *split* (découpage) d'une page d'index, déterminer la distance entre deux prédicats, etc.

Tout ceci est très virtuel, et rarement re-développé par les utilisateurs. Par contre, certaines extensions et outils intégrés utilisent ce mécanisme.

Il faut retenir qu'un index GiST est moins performant qu'un B-tree si ce dernier est possible.

L'utilisation se recoupant en partie avec les index GIN, il faut noter que :

- les index GiST ont moins tendance à se fragmenter que les GIN, même si cela dépend énormément du type de mises à jour ;
- les index GiST sont moins lourds à maintenir ;
- mais généralement moins performants.

1.4.1 GiST : cas d'usage

Le GiST indexe à peu près n'importe quoi

- géométries (PostGIS)
- intervalles, adresses IP, FTS…

D'autres usages recouvrent en partie ceux de GIN.

Un index GiST permet d'indexer n'importe quoi, quelle que soit la dimension, le type, tant qu'on peut utiliser des prédicats sur ce type.

Il est disponible pour les types natifs suivants :

- géométriques (`box` , `circle` , `point` , `poly`) : le projet PostGIS utilise les index GiST massivement, pour répondre efficacement à des questions complexes telles que « *quelles sont les routes qui coupent le Rhône* ? », « *quelles sont les villes adjacentes à Toulouse* ? », « *quels sont les restaurants situés à moins de 3 km de la Nationale 12* ? » ;
- `range` (d'`int` , de `timestamp` …) ;
- adresses IP/CIDR.
- *Full Text Search*.

Une partie des cas d'utilisation des index GiST recouvre ceux des index GIN. Nous verrons que les index GiST sont notamment utilisés par :

- les contraintes d'exclusion ;
- les recherches multicolonnes ;
- l'extension `pg_trgm` (dans ce cas, GiST est moins efficace que GIN pour la recherche exacte, mais permet de rapidement trouver les N enregistrements les plus proches d'une chaîne donnée, sans tri, et est plus compact).

1.4.2 GiST & KNN

- **KNN** = *K-Nearest neighbours* (K-plus proches voisins)
  - c'est-à-dire :

```
SELECT …
ORDER BY ma_colonne <-> une_référence LIMIT 10 ;
```

- Très utile pour la recherche de mots ressemblants, géographique
  - Exemple :

```
SELECT   p, p <-> point(18,36)
FROM     mes_points
ORDER BY p <-> point(18, 36)
LIMIT    4 ;
```

Les index GiST supportent les requêtes de type *K-plus proche voisins*, et permettent donc de répondre extrêmement rapidement à des requêtes telles que :

- quels sont les dix restaurants les plus proches d'un point particulier ?
- quels sont les 5 mots ressemblant le plus à « éphélant » ? afin de proposer des corrections à un utilisateur ayant commis une faute de frappe (par exemple).

Une convention veut que l'opérateur distance soit généralement nommé « `<->` », mais rien n'impose ce choix.

On peut prendre l'exemple d'indexation ci-dessus, avec le type natif `point` :

```
CREATE TABLE mes_points (p point);

INSERT INTO mes_points (SELECT point(i, j)
FROM generate_series(1, 100) i, generate_series(1,100) j WHERE random() > 0.8);

CREATE INDEX ON mes_points USING gist (p);
```

Pour trouver les 4 points les plus proches du point ayant pour coordonnées (18,36), on peut utiliser la requête suivante :

```
SELECT p,
       p <-> point(18,36)
FROM   mes_points
ORDER BY p <-> point(18, 36)
LIMIT 4;
```

```
    p    |     ?column?
---------+------------------
 (18,37) |                1
 (18,35) |                1
 (16,36) |                2
 (16,35) | 2.23606797749979
```

Cette requête utilise bien l'index GiST créé plus haut :

```
                          QUERY PLAN
-------------------------------------------------------------
 Limit  (cost=0.14..0.49 rows=4 width=16)
        (actual time=0.049..0.052 rows=4 loops=1)
   ->  Index Scan using mes_points_p_idx on mes_points
            (cost=0.14..176.72 rows=2029 width=16)
            (actual time=0.047..0.049 rows=4 loops=1)
         Order By: (p <-> '(18,36)'::point)
 Planning time: 0.050 ms
 Execution time: 0.075 ms
```

Les index SP-GiST sont compatibles avec ce type de recherche depuis la version 12.

1.4.3 GiST & Contraintes d'exclusion

Contrainte d'exclusion : une extension du concept d'unicité

- Unicité :
  - `n-uplet1 = n-uplet2` interdit dans une table
- Contrainte d'exclusion :
  - `n-uplet1 op n-uplet2` interdit dans une table
  - `op` est n'importe quel opérateur indexable par GiST
  - Exemple :

```
CREATE TABLE circles
    ( c circle,
      EXCLUDE USING gist (c WITH &&));
```

  - Exemple : réservations de salles

Premiers exemples :

Les contraintes d'unicité sont une forme simple de contraintes d'exclusion. Si on prend l'exemple :

```
CREATE TABLE foo (
    id int,
    nom text,
    EXCLUDE (id WITH =)
);
```

cette déclaration est équivalente à une contrainte UNIQUE sur `foo.id`, mais avec le mécanisme des contraintes d'exclusion. Ici, la contrainte s'appuie toujours sur un index B-tree. Les NULL sont toujours permis, exactement comme avec une contrainte UNIQUE.

On peut également poser une contrainte unique sur plusieurs colonnes :

```
CREATE TABLE foo (
    nom text,
    naissance date,
    EXCLUDE (nom WITH =, naissance WITH =)
);
```

Intérêt :

L'intérêt des contraintes d'exclusion est qu'on peut utiliser des index d'un autre type que les B-tree, comme les GiST ou les hash, et surtout des opérateurs autres que l'égalité, ce qui permet de couvrir des cas que les contraintes habituelles ne savent pas traiter.

Par exemple, une contrainte UNIQUE ne permet pas d'interdire que deux enregistrements de type intervalle aient des bornes qui se chevauchent. Cependant, il est possible de le faire avec une contrainte d'exclusion.

Exemples :

L'exemple suivante implémente la contrainte que deux objets de type `circle` (cercle) ne se chevauchent pas. Or ce type ne s'indexe qu'avec du GiST :

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
INSERT INTO circles(c) VALUES ('10, 4, 10');
INSERT INTO circles(c) VALUES ('8, 3, 8');

ERROR:  conflicting key value violates exclusion constraint "circles_c_excl"
DETAIL : Key (c)=(<(8,3),8>) conflicts with existing key (c)=(<(10,4),10>).
```

Un autre exemple très fréquemment proposé est celui de la réservation de salles de cours sur des plages horaires qui ne doivent pas se chevaucher :

```
CREATE TABLE reservation
(
  salle      TEXT,
  professeur TEXT,
  durant    tstzrange);

CREATE EXTENSION btree_gist ;

ALTER TABLE reservation ADD CONSTRAINT test_exclude EXCLUDE
USING gist (salle WITH =,durant WITH &&);

INSERT INTO reservation (professeur,salle,durant) VALUES
( 'marc', 'salle techno', '[2010-06-16 09:00:00, 2010-06-16 10:00:00)');
INSERT INTO reservation (professeur,salle,durant) VALUES
( 'jean', 'salle techno', '[2010-06-16 10:00:00, 2010-06-16 11:00:00)');
INSERT INTO reservation (professeur,salle,durant) VALUES
( 'jean', 'salle informatique', '[2010-06-16 10:00:00, 2010-06-16 11:00:00)');

INSERT INTO reservation (professeur,salle,durant) VALUES
( 'michel', 'salle techno', '[2010-06-16 10:30:00, 2010-06-16 11:00:00)');

ERROR:  conflicting key value violates exclusion constraint "test_exclude"
DETAIL : Key (salle, durant)=(salle techno,
              ["2010-06-16 10:30:00+02","2010-06-16 11:00:00+02"))
         conflicts with existing key
         (salle, durant)=(salle techno,
                          ["2010-06-16 10:00:00+02","2010-06-16 11:00:00+02")).
```

On notera que, là encore, l'extension `btree_gist` permet d'utiliser l'opérateur `=` avec un index GiST, ce qui nous permet d'utiliser `=` dans une contrainte d'exclusion.

Cet exemple illustre la puissance du mécanisme. Il est quasiment impossible de réaliser la même opération sans contrainte d'exclusion, à part en verrouillant intégralement la table, ou en utilisant le mode d'isolation *serializable*, qui a de nombreuses implications plus profondes sur le fonctionnement de l'application.

Autres fonctionnalités :

Enfin, précisons que les contraintes d'exclusion supportent toutes les fonctionnalités avancées que l'on est en droit d'attendre d'un système comme PostgreSQL : mode différé (*deferred*), application de la contrainte à un sous-ensemble de la table (permet une clause `WHERE`), ou utilisation de fonctions/expressions en place de références de colonnes.

1.5 GIN, GIST & PG_TRGM

- Indexation des recherches `LIKE '%critère%'`
- Similarité basée sur des trigrammes

```
CREATE EXTENSION pg_trgm;
SELECT similarity('bonjour','bnojour');

 similarity
------------
   0.333333
```

- Indexation (GIN ou GiST) :

```
CREATE INDEX test_trgm_idx ON test_trgm
  USING gist (text_data gist_trgm_ops);
```

Ce module permet de décomposer en trigramme les chaînes qui lui sont proposées :

```
 SELECT show_trgm('hello');

             show_trgm
 ---------------------------------
  {"  h","  he",ell,hel,llo,"lo "}
```

Une fois les trigrammes indexés, on peut réaliser de la recherche floue, ou utiliser des clauses `LIKE` malgré la présence de jokers (`%`) n'importe où dans la chaîne. À l'inverse, les indexations simples, de type B-tree, ne permettent des recherches efficaces que dans un cas particulier : si le seul joker de la chaîne est à la fin de celle ci (`LIKE 'hello%'` par exemple). Contrairement à la *Full Text Search*, la recherche par trigrammes ne réclame aucune modification des requêtes.

```
CREATE EXTENSION pg_trgm;

CREATE TABLE test_trgm (text_data text);

INSERT INTO test_trgm(text_data)
VALUES ('hello'), ('hello everybody'),
('helo young man'),('hallo!'),('HELLO !');
INSERT INTO test_trgm SELECT 'hola' FROM generate_series(1,1000);

CREATE INDEX test_trgm_idx ON test_trgm
  USING gist (text_data gist_trgm_ops);

SELECT text_data FROM test_trgm
 WHERE text_data like '%hello%';

    text_data
-----------------
```

```
 hello
 hello everybody
```

Cette dernière requête passe par l'index `test_trgm_idx`, malgré le `%` initial :

```
EXPLAIN (ANALYZE)
SELECT text_data FROM test_trgm
WHERE  text_data like '%hello%' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
Index Scan using test_trgm_gist_idx on test_trgm
   (cost=0.41..0.63 rows=1 width=8) (actual time=0.174..0.204 rows=2 loops=1)
   Index Cond: (text_data ~~ '%hello%'::text)
   Rows Removed by Index Recheck: 1
 Planning time: 0.202 ms
 Execution time: 0.250 ms
```

On peut aussi utiliser un index GIN (comme pour le *Full Text Search*). Les index GIN ont l'avantage d'être plus efficaces pour les recherches exhaustives. Mais l'indexation pour la recherche des k éléments les plus proches (on parle de recherche k-NN) n'est disponible qu'avec les index GiST .

```
SELECT text_data, text_data <-> 'hello'
 FROM test_trgm
 ORDER BY text_data <-> 'hello'
 LIMIT 4;
```

nous retourne par exemple les deux enregistrements les plus proches de « hello » dans la table `test_trgm`.

1.6 INDEXATION MULTICOLONNE : GIN, GIST & BLOOM

- Multi-colonnes dans n'importe quel ordre
- GIN ou GiST
  - extensions `btree_gist` ou `btree_gin`
- Ou bloom ?
- Quel est le meilleur ?
  - ça dépend…

GiST comme GIN sont intéressants si on a besoin d'indexer plusieurs colonnes, sans trop savoir quelles colonnes seront interrogées.

Pour indexer des scalaires, il faut utiliser les extensions `btree_gist` ou `btree_gin`.

```
CREATE EXTENSION IF NOT EXISTS btree_gist ;
CREATE EXTENSION IF NOT EXISTS btree_gin ;
```

Ce premier jeu de données utilisera des données qui se répètent beaucoup (de basse cardinalité) et les requêtes ramèneront de nombreuses lignes :

```
CREATE TABLE demo_gist (n int, i int, j int, k int, l int,
                        filler char(50) default ' ') ;
CREATE TABLE demo_gin  (n int, i int, j int, k int, l int,
                        filler char(50) default ' ') ;
CREATE INDEX demo_gist_idx ON demo_gist USING gist (i,j,k,l) ;
CREATE INDEX demo_gin_idx ON demo_gin USING gin (i,j,k,l) ;

INSERT INTO demo_gist
SELECT n, mod(n,37) AS i, mod(n,53) AS j, mod (n, 97) AS k, mod(n,229) AS l
FROM generate_series (1,1000000) n ;

INSERT INTO demo_gin
SELECT n, mod(n,37) AS i, mod(n,53) AS j, mod (n, 97) AS k, mod(n,229) AS l
FROM generate_series (1,1000000) n ;
```

Même en ne fournissant pas la première colonne des index, les index GIN et GiST sont utilisables :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_gist WHERE j=17 AND l=17  ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gist (actual time=1.615..1.662 rows=83 loops=1)
   Recheck Cond: ((j = 17) AND (l = 17))
   Heap Blocks: exact=83
   Buffers: shared hit=434
```

```
   ->  Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=1.607..1.607 rows=83
↪  loops=1)
         Index Cond: ((j = 17) AND (l = 17))
         Buffers: shared hit=351
 Planning Time: 0.026 ms
 Execution Time: 1.673 ms
```

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_gin WHERE j=17 AND l=17  ;
```

```
Bitmap Heap Scan on demo_gin (actual time=0.436..0.491 rows=83 loops=1)
   Recheck Cond: ((j = 17) AND (l = 17))
   Heap Blocks: exact=83
   Buffers: shared hit=100
   ->  Bitmap Index Scan on demo_gin_i_j_k_l_idx (actual time=0.427..0.427 rows=83
↪  loops=1)
         Index Cond: ((j = 17) AND (l = 17))
         Buffers: shared hit=17
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.031 ms
 Execution Time: 0.503 ms
```

Le GIN est ici plus efficace car le nombre de blocs d'index balayés est plus bas. En effet, avec une basse cardinalité, la compression du GIN joue à plein. (Pour ces tables identiques de 97 Mo sous PostgreSQL 16, l'index GiST fait 62 Mo, et le GIN 19 Mo seulement.)

Si la première colonne `i` était systématiquement fournie à la requête, on pourrait se contenter d'un index B-tree (30 Mo ici) ; mais il serait peu efficace pour les autres requêtes : la requête précédente donnerait souvent lieu à un *Seq Scan* parallélisé, bien que parfois un *Bitmap Scan* puisse apparaître avec des performances satisfaisantes.

Toujours dans ce cas précis, le temps de création de l'index GIN est aussi meilleur que le GiST d'un facteur deux au moins (et équivalent au B-tree), mais ce temps est très sensible à la valeur de `maintenance_work_mem`.

À l'inverse, le GiST est bien plus performant que le GIN dans d'autres types de requêtes commme celle-ci avec `BETWEEN` :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_gist
WHERE j BETWEEN 17 AND 21 AND l BETWEEN 17 AND 21  ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gist (actual time=8.419..8.895 rows=2059 loops=1)
   Recheck Cond: ((j >= 17) AND (j <= 21) AND (l >= 17) AND (l <= 21))
   Heap Blocks: exact=757
   Buffers: shared hit=1744
   ->  Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=8.355..8.355
↪  rows=2059 loops=1)
         Index Cond: ((j >= 17) AND (j <= 21) AND (l >= 17) AND (l <= 21))
         Buffers: shared hit=987
 Planning Time: 0.030 ms
 Execution Time: 8.958 ms
```

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_gin
WHERE j BETWEEN 17 AND 21 AND l BETWEEN 17 AND 21  ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gin (actual time=68.281..69.313 rows=2059 loops=1)
   Recheck Cond: ((j >= 17) AND (j <= 21) AND (l >= 17) AND (l <= 21))
   Heap Blocks: exact=757
   Buffers: shared hit=1760
   ->  Bitmap Index Scan on demo_gin_i_j_k_l_idx (actual time=68.174..68.174
↪  rows=2059 loops=1)
         Index Cond: ((j >= 17) AND (j <= 21) AND (l >= 17) AND (l <= 21))
         Buffers: shared hit=1003
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.056 ms
 Execution Time: 69.435 ms
```

Un index GiST permet aussi l'*Index Scan* et parfois l'*Index Only Scan*, au contraire d'un index GIN, qui se limitera toujours à un *Bitmap Scan*. L'intérêt varie selon les requêtes :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT j,l FROM demo_gist WHERE j=17 AND l=17 ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using demo_gist_i_j_k_l_idx on demo_gist (actual time=0.043..1.563
↪  rows=83 loops=1)
   Index Cond: ((j = 17) AND (l = 17))
   Heap Fetches: 0
   Buffers: shared hit=352
 Planning Time: 0.024 ms
 Execution Time: 1.572 ms

EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT j,l FROM demo_gin WHERE j=17 AND l=17 ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gin (actual time=0.699..0.796 rows=83 loops=1)
   Recheck Cond: ((j = 17) AND (l = 17))
   Heap Blocks: exact=83
   Buffers: shared hit=100
   ->  Bitmap Index Scan on demo_gin_i_j_k_l_idx (actual time=0.683..0.683 rows=83
↪  loops=1)
         Index Cond: ((j = 17) AND (l = 17))
         Buffers: shared hit=17
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.069 ms
 Execution Time: 0.819 ms
```

Si la cardinalité est élevée (les données sont toutes différentes), l'index GIN perd son avantage en taille. En effet, si l'on remplace les données précédentes par un jeu de données toutes différentes :

```
TRUNCATE TABLE demo_gin ;
TRUNCATE TABLE demo_gist ;
INSERT INTO demo_gin
SELECT n, n AS i, 100e6+n AS j, 200e6+n AS k, 300e6+n AS l
FROM generate_series (1,1000000) n ;
INSERT INTO demo_gist
SELECT n, n AS i, 100e6+n AS j, 200e6+n AS k, 300e6+n AS l
FROM generate_series (1,1000000) n ;
```

la taille de l'index GIN passe à 218 Mo (plus que la table), alors que l'index GiST ne monte qu'à 85 Mo (et un index B-tree resterait à 30 Mo). Cela ne rend pas forcément l'index GIN moins efficace dans l'absolu, mais a un effet défavorable sur le cache.

Index bloom :

Il existe encore un type d'index, rarement utilisé : l'index Bloom[6], qui réclame l'installation de l'extension `bloom` (livrée avec PostgreSQL). Cet index est basé sur les filtres bloom[7], de nature probabiliste. Les lignes retournées par l'index doivent être revérifiées dans la table, ce qui impose un *Recheck* systématique. L'index est rapide à générer, et encore plus petit que les index ci-dessus (15 Mo ici), mais cet index doit être complètement parcouru. Les performances sont donc moins bonnes qu'avec GiST ou GIN. Il est aussi limité aux entiers et chaînes de caractères, et à une recherche sur l'égalité (donc il est inadapaté aux critères `BETWEEN` et `LIKE`). Si ses performances suffisent, un index bloom peut être utile dans le cas où de nombreuses colonnes sont susceptibles d'être interrogées, car il évite de créer d'autres index B-tree ou GIN coûteux en place. Sur une table identique à celles ci-dessus, le plan est :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_bloom WHERE j=100000017 AND l=300000017  ;

                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on demo_bloom (actual time=3.732..3.753 rows=1 loops=1)
   Recheck Cond: ((j = 100000017) AND (l = 300000017))
   Rows Removed by Index Recheck: 27
   Heap Blocks: exact=28
   Buffers: shared hit=1961 read=28
   ->  Bitmap Index Scan on demo_bloom_i_j_k_l_idx (actual time=3.721..3.722 rows=28
↪  loops=1)
         Index Cond: ((j = 100000017) AND (l = 300000017))
         Buffers: shared hit=1933 read=28
 Planning Time: 0.042 ms
 Execution Time: 3.767 ms
```

Choix du type d'index pour une indexation multicolonne :

Les exemples ci-dessus viennent d'une instance avec la configuration par défaut de PostgreSQL. Rappelons que les paramètres `effective_io_concurrency`, `seq_page_cost` et `random_page_cost` (liés aux disques) et `effective_cache_size` (lié à la mémoire), influent fortement sur le choix d'un parcours *Index Scan*, *Bitmap Scan* ou *Seq Scan* quand la requête balaie beaucoup de lignes. La répar-

[6] https://docs.postgresql.fr/current/bloom.html
[7] https://fr.wikipedia.org/wiki/Filtre_de_Bloom

tition physique des données joue aussi (sur l'efficacité du cache comme sur celle des *Index Scan* et *Bitmap Scan*.) Le type de données à indexer, leur longueur, l'opérateur… a aussi une importance.

Au final, pour une indexation multicolonne d'une table, il faudra tester les types d'index disponibles avec la charge, le paramétrage et les requêtes réelles, des données réelles, et arbitrer en tenant compte des tailles des index (donc de l'impact sur le cache), des durées de génération, des performances pures et des performances en pratique acceptables…

1.7 INDEX BRIN

BRIN : **B**lock **R**ange **IN**dex

```
CREATE INDEX brin_demo_brin_idx ON brin_demo USING brin (age)
WITH (pages_per_range=16) ;
```

- Valeurs corrélées à leur emplacement physique
- Calcule des plages de valeur par groupe de blocs
  - index très compact
- Pour :
  - grosses volumétries
  - corrélation entre emplacement et valeur
  - penser à `CLUSTER`

Un index BRIN ne stocke pas les valeurs de la table, mais quelles plages de valeurs se rencontrent dans un ensemble de blocs. Cela réduit la taille de l'index et permet d'exclure un ensemble de blocs lors d'une recherche.

1.7.0.1 Exemple

Soit une table `brin_demo` de 2 millions de personnes, triée par âge :

```
CREATE TABLE brin_demo (id int, age int);
SET work_mem TO '300MB' ;
INSERT INTO brin_demo
SELECT id, trunc(random() * 90 + 1)  AS age
FROM   generate_series(1,2e6) id
ORDER  BY age ;
```

```
=# \dt+ brin_demo
                            List of relations
 Schema |   Name     | Type  |  Owner   | Persistence |  Size   | Description
--------+------------+-------+----------+-------------+---------+-------------
 public | brin_demo | table | postgres | permanent   |   69 MB |
```

Un index BRIN va contenir une plage des valeurs pour chaque bloc. Dans notre exemple, l'index contiendra la valeur minimale et maximale de plusieurs blocs. La conséquence est que ce type d'index prend très peu de place et il peut facilement tenir en RAM (réduction des opérations des disques). Il est aussi plus rapide à construire et maintenir.

```
CREATE INDEX brin_demo_btree_idx ON brin_demo USING btree (age);
CREATE INDEX brin_demo_brin_idx  ON brin_demo USING brin (age);
```

```
=# \di+ brin_demo*
                          List of relations
 Schema |          Name          | Type  | … |    Table     | Persistence | Size  | …
--------+------------------------+-------+---+--------------+-------------+-------+---
 public | brin_demo_brin_idx     | index |   | brin_demo    | permanent   | 48 kB |
 public | brin_demo_btree_idx    | index |   | brin_demo    | permanent   | 13 MB |
```

On peut consulter le contenu de cet index[8], et constater que chacune de ses entrées liste les valeurs de `age` par paquet de 128 blocs (cette valeur peut se changer) :

```
CREATE EXTENSION IF NOT EXISTS pageinspect ;
SELECT *
FROM brin_page_items(get_raw_page('brin_demo_brin_idx', 2),'brin_demo_brin_idx');
```

```
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 2}
          2 |    128 |      1 | f        | f        | f           | {2 .. 3}
          3 |    256 |      1 | f        | f        | f           | {3 .. 4}
          4 |    384 |      1 | f        | f        | f           | {4 .. 6}
          5 |    512 |      1 | f        | f        | f           | {6 .. 7}
          6 |    640 |      1 | f        | f        | f           | {7 .. 8}
          7 |    768 |      1 | f        | f        | f           | {8 .. 10}
          8 |    896 |      1 | f        | f        | f           | {10 .. 11}
          9 |   1024 |      1 | f        | f        | f           | {11 .. 12}
…
         66 |   8320 |      1 | f        | f        | f           | {85 .. 86}
         67 |   8448 |      1 | f        | f        | f           | {86 .. 88}
         68 |   8576 |      1 | f        | f        | f           | {88 .. 89}
         69 |   8704 |      1 | f        | f        | f           | {89 .. 90}
         70 |   8832 |      1 | f        | f        | f           | {90 .. 90}
```

La colonne `blknum` indique le début de la tranche de blocs, et `value` la plage de valeurs rencontrées. Les personnes de 87 ans sont donc présentes uniquement entre les blocs 8448 à 8575, ce qui se vérifie :

```
SELECT min (ctid), max (ctid) FROM brin_demo WHERE age = 87 ;
```

```
    min     |    max
------------+-----------
 (8457,170) | (8556,98)
```

Testons une requête avec uniquement ce petit index BRIN :

```
DROP INDEX brin_demo_btree_idx ;

EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT count(*) FROM brin_demo WHERE age = 87 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate (actual time=4.838..4.839 rows=1 loops=1)
   Buffers: shared hit=130
   ->  Bitmap Heap Scan on brin_demo (actual time=0.241..3.530 rows=22212 loops=1)
         Recheck Cond: (age = 87)
```

[8] https://docs.postgresql.fr/current/pageinspect.html#id-1.11.7.33.8

```
         Rows Removed by Index Recheck: 6716
         Heap Blocks: lossy=128
         Buffers: shared hit=130
         ->  Bitmap Index Scan on brin_demo_brin_idx (actual time=0.024..0.024
↪  rows=1280 loops=1)
               Index Cond: (age = 87)
               Buffers: shared hit=2
 Planning:
   Buffers: shared hit=5
 Planning Time: 0.084 ms
 Execution Time: 4.873 ms
```

On constate que l'index n'est consulté que sur 2 blocs. Le nœud *Bitmap Index Scan* renvoie les 128 blocs contenant des valeurs entre 86 et 88, et ces blocs sont récupérés dans la table (*heap*). 6716 lignes sont ignorées, et 22 212 conservées. Le temps de 4,8 ms est bon.

Certes, un index B-tree, bien plus gros, aurait fait encore mieux dans ce cas précis, qui est modeste. Mais plus la table est énorme, plus une requête en ramène une proportion importante, plus les aller-retours entre index et table sont pénalisants, et plus l'index BRIN devient compétitif, en plus de rester très petit.

Par contre, si la table se fragmente, même un peu, les lignes d'un même `age` se retrouvent réparties dans toute la table, et l'index BRIN devient bien moins efficace :

```
UPDATE brin_demo
SET    age=age+0
WHERE  random()>0.99 ;  -- environ 20000 lignes
VACUUM brin_demo ;
UPDATE brin_demo
SET    age=age+0
WHERE  age > 80 AND random()>0.90 ;  -- environ 22175 lignes

VACUUM ANALYZE brin_demo ;

SELECT *
FROM brin_page_items(get_raw_page('brin_demo_brin_idx', 2),'brin_demo_brin_idx');
```

```
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 81}
          2 |    128 |      1 | f        | f        | f           | {2 .. 81}
          3 |    256 |      1 | f        | f        | f           | {3 .. 81}
          4 |    384 |      1 | f        | f        | f           | {4 .. 81}
…
         45 |   5632 |      1 | f        | f        | f           | {58 .. 87}
         46 |   5760 |      1 | f        | f        | f           | {59 .. 87}
         47 |   5888 |      1 | f        | f        | f           | {60 .. 87}
…
         56 |   7040 |      1 | f        | f        | f           | {72 .. 89}
         57 |   7168 |      1 | f        | f        | f           | {73 .. 89}
         58 |   7296 |      1 | f        | f        | f           | {75 .. 89}
…
         67 |   8448 |      1 | f        | f        | f           | {86 .. 88}
         68 |   8576 |      1 | f        | f        | f           | {88 .. 89}
         69 |   8704 |      1 | f        | f        | f           | {89 .. 90}
         70 |   8832 |      1 | f        | f        | f           | {1 .. 90}
```

```
EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT count(*) FROM brin_demo WHERE age = 87 ;

                              QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate (actual time=71.053..71.055 rows=1 loops=1)
   Buffers: shared hit=3062
   ->  Bitmap Heap Scan on brin_demo (actual time=2.451..69.851 rows=22303 loops=1)
         Recheck Cond: (age = 87)
         Rows Removed by Index Recheck: 664141
         Heap Blocks: lossy=3060
         Buffers: shared hit=3062
         ->  Bitmap Index Scan on brin_demo_brin_idx (actual time=0.084..0.084
↪  rows=30600 loops=1)
               Index Cond: (age = 87)
               Buffers: shared hit=2
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.069 ms
 Execution Time: 71.102 ms
```

3060 blocs et 686 444 lignes, la plupart inutiles, ont été lus dans la table (en gros, un tiers de celles-ci). Cela ne devient plus très intéressant par rapport à un parcours complet de la table.

Pour rendre son intérêt à l'index, il faut reconstruire la table avec les données dans le bon ordre avec la commande `CLUSTER`[9]. Hélas, c'est une opération au moins aussi lourde et bloquante qu'un `VACUUM FULL`. De plus, le tri de la table ne peut se faire par l'index BRIN, et il faut recréer un index B-tree au moins le temps de l'opération.

```
CREATE INDEX brin_demo_btree_idx ON brin_demo USING btree (age);
CLUSTER brin_demo USING brin_demo_btree_idx;
SELECT *
FROM brin_page_items(get_raw_page('brin_demo_brin_idx', 2),'brin_demo_brin_idx') ;

 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 2}
          2 |    128 |      1 | f        | f        | f           | {2 .. 3}
…
         68 |   8576 |      1 | f        | f        | f           | {88 .. 89}
         69 |   8704 |      1 | f        | f        | f           | {89 .. 90}
         70 |   8832 |      1 | f        | f        | f           | {90 .. 90}
```

On revient alors à la situation de départ.

1.7.0.2 Utilité d'un index BRIN

Pour qu'un index BRIN soit utile, il faut donc :

- que les données soit naturellement triées dans la table, et le restent (série temporelle, décisionnel avec imports réguliers…) ;
- que la table soit à « insertion seule » pour conserver l'ordre physique ;

[9] https://docs.postgresql.fr/current/sql-cluster.html

- ou que l'on ait la disponibilité nécessaire pour reconstruire régulièrement la table avec `CLUSTER` et éviter que les performances se dégradent au fil du temps.

Sous ces conditions, les BRIN sont indiqués si l'on a des problèmes de volumétrie, ou de temps d'écritures dus aux index B-tree, ou pour éviter de partitionner une grosse table dont les requêtes ramènent une grande proportion.

1.7.0.3 Index BRIN sur clé composée

Prenons un autre exemple avec plusieurs colonnes et un type `text` :

```
CREATE TABLE test (id serial PRIMARY KEY, val text);
INSERT INTO test (val) SELECT md5(i::text) FROM generate_series(1, 10000000) i;
```

La colonne `id` sera corrélée (c'est une séquence), la colonne `md5` ne sera pas du tout corrélée. L'index BRIN porte sur les deux colonnes :

```
CREATE INDEX test_brin_idx ON test USING brin (id,val);
```

Pour une table de 651 Mo, l'index ne fait ici que 104 ko.

Pour voir son contenu :

```
SELECT itemoffset, blknum, attnum,value
FROM brin_page_items(get_raw_page('test_brin_idx', 2),'test_brin_idx')
LIMIT 4 ;
```

```
 itemoffset | blknum | attnum |                  value
------------+--------+--------+----------------------------------------
          1 |      0 |      1 | {1 .. 15360}
          1 |      0 |      2 | {00003e3b9e5336685200ae85d21b4f5e ..
 ↪  fffb8ef15de06d87e6ba6c830f3b6284}
          2 |    128 |      1 | {15361 .. 30720}
          2 |    128 |      2 | {00053f5e11d1fe4e49a221165b39abc9 ..
 ↪  fffe9f664c2ddba4a37bcd35936c7422}
```

La colonne `attnum` correspond au numéro d'attribut du champ dans la table. L'`id` est bien corrélé aux numéros de bloc, contrairement à la colonne `val`. Ce que nous confirme bien la vue `pg_stats` :

```
SELECT tablename, attname, correlation
FROM   pg_stats WHERE  tablename='test' ORDER BY attname ;
```

```
 tablename | attname | correlation
-----------+---------+-------------
 test      | id      |           1
 test      | val     |  0.00528745
```

Si l'on teste la requête suivante, on s'aperçoit que PostgreSQL effectue un parcours complet (*Seq Scan*) de façon parallélisée ou non, et n'utilise donc pas l'index BRIN. Pour comprendre pourquoi, essayons de l'y forcer :

```
SET enable_seqscan TO off ;
SET max_parallel_workers_per_gather TO 0;

EXPLAIN (BUFFERS,ANALYZE) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
    AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=721.46..234055.46 rows=2642373 width=37) (actual
↪  time=2.558..1622.646 rows=2668675 loops=1)
   Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
              AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Rows Removed by Index Recheck: 7331325
   Heap Blocks: lossy=83334
   Buffers: shared hit=83349
   ->  Bitmap Index Scan on test_brin_idx  (cost=0.00..60.86 rows=10000000 width=0)
↪  (actual time=2.523..2.523 rows=833340 loops=1)
         Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                  AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
         Buffers: shared hit=15
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.079 ms
 Execution Time: 1703.018 ms
```

83 334 blocs sont lus (651 Mo) soit l'intégralité de la table ! Il est donc logique que PostgreSQL préfère d'entrée un *Seq Scan* (parcours complet).

Pour que l'index BRIN soit utile pour ce critère, il faut là encore trier la table avec une commande `CLUSTER`, ce qui nécessite un index B-tree :

```
CREATE INDEX test_btree_idx ON test USING btree (val);

\di+ test_btree_idx
                              List of relations
 Schema |      Name      | Type  |  Owner   | Table |  Size  | Description
--------+----------------+-------+----------+-------+--------+-------------
 cave   | test_btree_idx | index | postgres | test  | 563 MB |
```

Notons au passage que cet index B-tree est presque aussi gros que notre table !

Après la commande `CLUSTER`, notre table est bien corrélée avec `val` (mais plus avec `id`) :

```
CLUSTER test USING test_btree_idx ;
ANALYZE test;
SELECT tablename, attname, correlation
FROM   pg_stats WHERE  tablename='test' ORDER BY attname ;

 tablename | attname | correlation
-----------+---------+----------------
 test      | id      |  -0.0023584804
 test      | val     |              1
```

La requête après le cluster utilise alors l'index BRIN :

```
SET enable_seqscan TO on ;

EXPLAIN (BUFFERS,ANALYZE) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
    AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';

                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=712.28..124076.96 rows=2666839 width=37) (actual
↪  time=1.460..540.250 rows=2668675 loops=1)
   Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
              AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Rows Removed by Index Recheck: 19325
   Heap Blocks: lossy=22400
   Buffers: shared hit=22409
   ->  Bitmap Index Scan on test_brin_idx  (cost=0.00..45.57 rows=2668712 width=0)
↪  (actual time=0.520..0.520 rows=224000 loops=1)
         Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                   AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
         Buffers: shared hit=9
 Planning:
   Buffers: shared hit=18 read=2
   I/O Timings: read=0.284
 Planning Time: 0.468 ms
 Execution Time: 630.124 ms
```

22 400 blocs sont lus dans la table, soit 175 Mo. Dans la table triée, l'index BRIN devient intéressant. Cette remarque vaut aussi si PostgreSQL préfère un *Index Scan* au plan précédent (notamment si `random_page_cost` vaut moins du 4 par défaut).

On supprime notre index BRIN et on garde l'index B-tree :

```
DROP INDEX test_brin_idx;

EXPLAIN (BUFFERS,ANALYZE) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
AND     'eccbc87e4b5ce2fe28308fd9f2a7baf3';

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using test_btree_idx on test  (cost=0.56..94786.34 rows=2666839
↪  width=37) (actual time=0.027..599.185 rows=2668675 loops=1)
   Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
             AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Buffers: shared hit=41306
 Planning:
   Buffers: shared hit=12
 Planning Time: 0.125 ms
 Execution Time: 675.624 ms
```

La durée est ici similaire, mais le nombre de blocs lus est double, ce qui est une conséquence de la taille de l'index.

1.7.0.4 Maintenance et consultation

Les lignes ajoutées à la table après la création de l'index ne sont pas forcément intégrées au résumé tout de suite. Il faut faire attention à ce que le `VACUUM` passe assez souvent[10]. Cela dit, la maintenance d'un BRIN lors d'écritures est plus légères qu'un gros B-tree.

Pour modifier la granularité de l'index BRIN, il faut utiliser le paramètre `pages_per_range` à la création :

```
CREATE INDEX brin_demo_brin_idx ON brin_demo
USING brin (age) WITH (pages_per_range=16) ;
```

Les calculs de plage de valeur se feront alors par paquets de 16 blocs, ce qui est plus fin tout en conservant une volumétrie dérisoire. Sur la table `brin_demo`, l'index ne fait toujours que 56 ko. Par contre, la requête d'exemple parcourra un peu moins de blocs inutiles. La plage est à ajuster en fonction de la finesse des données et de leur répartition.

Pour consulter la répartition des valeurs comme nous l'avons fait plus haut, il faut utiliser pageinspect[11]. Pour une table `brin_demo` dix fois plus grosse, et des plages de 16 blocs :

```
SELECT * FROM brin_metapage_info(get_raw_page('brin_demo_brin_idx', 0));
```

```
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |            16 |              5
```

On retrouve le paramètre de plages de 16 blocs, et la range map commence au bloc 5. Par ailleurs, `pg_class.relpages` indique 20 blocs. Nous avons donc les bornes des pages de l'index à consulter :

```
SELECT * FROM generate_series (6,19) p,
  LATERAL (SELECT * FROM brin_page_items(get_raw_page('brin_demo_brin_idx',
 ↪  p),'brin_demo_brin_idx') ) b
ORDER BY blknum ;
```

```
p  | itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
---+------------+--------+--------+----------+----------+-------------+------------
 18 |        273 |      0 |      1 | f        | f        | f           | {1 .. 1}
 18 |        274 |     16 |      1 | f        | f        | f           | {1 .. 1}
 18 |        275 |     32 |      1 | f        | f        | f           | {1 .. 1}
 18 |        276 |     48 |      1 | f        | f        | f           | {1 .. 1}
…
 19 |        224 |  88432 |      1 | f        | f        | f           | {90 .. 90}
 19 |        225 |  88448 |      1 | f        | f        | f           | {90 .. 90}
 19 |        226 |  88464 |      1 | f        | f        | f           | {90 .. 90}
 19 |        227 |  88480 |      1 | f        | f        | f           | {90 .. 90}
(5531 lignes)
```

[10] https://docs.postgresql.fr/current/brin-intro.html#BRIN-OPERATION
[11] https://docs.postgresql.fr/current/pageinspect.html#id-1.11.7.33.8

1.7.0.5 Plus d'informations sur les BRIN

- Conférence d'Adrien Nayrat au PGDay France 2016 (Lille, 31 mai 2016) : vidéo[12], texte[13].
- Documentation officielle[14].

[12] https://youtu.be/g3tSRyeN1TY
[13] https://kb.dalibo.com/conferences/index_brin/index_brin_pgday
[14] https://docs.postgresql.fr/current/brin.html

1.8 INDEX HASH

- Basés sur un hash
- Tous types de données, quelle que soit la taille
- Ne gèrent que `=`
  - donc ni `<`, `>`, `!=` …
- Mais plus compacts
- Ne pas utiliser avant v10 (non journalisés)

Les index hash contiennent des hachages de tout type de données. Cela leur permet d'être relativement petits, même pour des données de gros volume, et d'être une alternative aux index B-tree qui ne peuvent pas indexer des objets de plus de 2,7 ko.

Une conséquence de ce principe est qu'il est impossible de parcourir des plages de valeurs, seule l'égalité **exacte** à un critère peut être recherchée. Cet exemple utilise la base du projet Gutenberg[15] :

```
EXPLAIN (ANALYZE,BUFFERS,COSTS OFF)
SELECT * FROM textes
-- attention au nombre exact d'espaces
WHERE contenu = '    Maître corbeau, sur un arbre perché' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using textes_contenu_hash_idx on textes (actual time=0.049..0.050 rows=1
 ↪  loops=1)
   Index Cond: (contenu = '    Maître corbeau, sur un arbre perché'::text)
   Buffers: shared hit=3
 Planning Time: 0.073 ms
 Execution Time: 0.072 ms
```

Les index hash restent plus longs à créer que des index B-tree. Ils ne sont plus petits qu'eux que si les champs indexés sont gros. Par exemple, dans la même table `texte`, de 3 Go, le nom de l'œuvre (`livre`) est court, mais une ligne de texte (`contenu`) peut faire 3 ko (ici, on a dû purger les trois lignes trop longues pour être indexées par un B-tree) :

```
SELECT pg_size_pretty(pg_relation_size(indexname::regclass)) AS taille,
indexdef FROM pg_indexes
WHERE indexname like 'texte%' ;
```

```
taille  |                                indexdef
--------+-------------------------------------------------------------------------
733 MB  | CREATE INDEX textes_contenu_hash_idx ON public.textes USING hash (contenu)
1383 MB | CREATE INDEX textes_contenu_idx ON public.textes USING btree (contenu
 ↪  varchar_pattern_ops)
```

[15] https://dali.bo/tp_gutenberg

```
1033 MB | CREATE INDEX textes_livre_hash_idx ON public.textes USING hash (livre)
155 MB  | CREATE INDEX textes_livre_idx ON public.textes USING btree (livre
↪  varchar_pattern_ops)
```

On réservera donc les index hash à l'indexation de grands champs, éventuellement binaires, notamment dans des requêtes recherchant la présence d'un objet. Ils peuvent vous éviter de gérer vous-même un hachage du champ.

Les index hash n'étaient pas journalisés avant la version 10, leur utilisation y était donc une mauvaise idée (corruption à chaque arrêt brutal, pas de réplication…). Ils étaient aussi peu performants par rapport à des index B-tree. Ceci explique le peu d'utilisation de ce type d'index jusqu'à maintenant.

1.9 OUTILS

- Pour identifier des requêtes
- Pour identifier des prédicats et des requêtes liées
- Pour valider un index

Différents outils permettent d'aider le développeur ou le DBA à identifier plus facilement les index à créer. On peut classer ceux-ci en trois groupes, selon l'étape de la méthodologie à laquelle ils s'appliquent.

Tous les outils suivants sont disponibles dans les paquets diffusés par le PGDG sur yum.postgresql.org[16] ou apt.postgresql.org[17].

1.9.1 Identifier les requêtes

- pgBadger
- pg_stat_statements
- PoWA

Pour identifier les requêtes les plus lentes, et donc potentiellement nécessitant une réécriture ou un nouvel index, pgBadger[18] permet d'analyser les logs une fois ceux-ci configurés pour tracer toutes les requêtes. Des exemples figurent dans notre formation DBA1[19].

Pour une vision cumulative, voire temps réel, de ces requêtes, l'extension `pg_stat_statements`, fournie avec les « contrib » de PostgreSQL, permet de garder trace des N requêtes les plus fréquemment exécutées, et calcule le temps d'exécution total de chacune d'entre elles, ainsi que les accès au cache de PostgreSQL ou au système de fichiers. Son utilisation est détaillée dans notre module X2[20].

Le projet PoWA[21] exploite ces statistiques en les historisant, et en fournissant une interface web permettant de les exploiter.

[16] https://yum.postgresql.org
[17] https://apt.postgresql.org
[18] https://pgbadger.darold.net
[19] https://dali.bo/h1_html#pgbadger
[20] https://dali.bo/x2_html#pg_stat_statements
[21] https://powa.readthedocs.io/en/latest/

1.9.2 Identifier les prédicats et des requêtes liées

- Extension pg_qualstats
  - avec PoWa

Pour identifier les prédicats (clause `WHERE` ou condition de jointure à identifier en priorité), l'extension pg_qualstats[22] permet de pousser l'analyse offerte par pg_stat_statements au niveau du prédicat lui-même. Ainsi, on peut détecter les requêtes filtrant sur les mêmes colonnes, ce qui peut aider notamment à déterminer des index multi-colonnes ou des index partiels.

De même que pg_stat_statements, cette extension peut être historisée et exploitée par le biais du projet PoWA.

1.9.3 Extension HypoPG

- Extension PostgreSQL
- Création d'index hypothétiques pour tester leur intérêt
  - avant de les créer pour de vrai
- Limitations : surtout B-Tree, statistiques

Cette extension est disponible sur GitHub[23] et dans les paquets du PGDG. Il existe trois fonctions principales et une vue :

- `hypopg_create_index()` pour créer un index hypothétique ;
- `hypopg_drop_index()` pour supprimer un index hypothétique particulier ou `hypopg_reset()` pour tous les supprimer ;
- `hypopg_list_indexes` pour les lister.

Un index hypothétique n'existe que dans la session, ni en mémoire ni sur le disque, mais le planificateur le prendra en compte dans un `EXPLAIN` simple (évidemment pas un `EXPLAIN ANALYZE`). En quittant la session, tous les index hypothétiques restants et créés sur cette session sont supprimés.

[22] https://github.com/powa-team/pg_qualstats
[23] https://github.com/HypoPG/hypopg

L'exemple suivant est basé sur la base dont le script peut être téléchargé sur https://dali.bo/tp_employes_services.

```
CREATE EXTENSION hypopg;

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
         Filter: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_create_index('CREATE INDEX ON employes_big(prenom)');

 indexrelid |            indexname
------------+---------------------------------
      24591 | <24591>btree_employes_big_prenom

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                          QUERY PLAN
---------------------------------------------------------------------
 Index Scan using <24591>btree_employes_big_prenom on employes_big
                                (cost=0.05..4.07 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_list_indexes;

 indexrelid |            indexname             | nspname |   relname    | amname
------------+----------------------------------+---------+--------------+--------
      24591 | <24591>btree_employes_big_prenom | public  | employes_big | btree

SELECT * FROM hypopg_reset();

 hypopg_reset
--------------

(1 row)

CREATE INDEX ON employes_big(prenom);

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                      QUERY PLAN
------------------------------------------------------
 Index Scan using employes_big_prenom_idx on employes_big
                       (cost=0.42..4.44 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
```

Le cas idéal d'utilisation est l'index B-Tree sur une colonne. Un index fonctionnel est possible, mais, faute de statistiques disponibles avant la création réelle de l'index, les estimations peuvent être fausses. Les autres types d'index sont moins bien ou non supportées.

1.9.4 Étude des index à créer

- PoWA peut utiliser HypoPG

Le projet PoWA propose une fonctionnalité, encore rudimentaire, de suggestion d'index à créer, en se basant sur HypoPG, pour répondre à la question « Quel serait le plan d'exécution de ma requête si cet index existait ? ».

L'intégration d'HypoPG dans PoWA permet là aussi une souplesse d'utilisation, en présentant les plans espérés avec ou sans les index suggérés.

Ensuite, en ouvrant l'interface de PoWA, on peut étudier les différentes requêtes, et les suggestions d'index réalisées par l'outil. À partir de ces suggestions, on peut créer les nouveaux index, et enfin relancer le bench pour constater les améliorations de performances.

1.10 QUIZ

https://dali.bo/j5_quiz

1.11 TRAVAUX PRATIQUES

Tous les TP se basent sur la configuration par défaut de PostgreSQL, sauf précision contraire.

1.11.1 Indexation de motifs avec les varchar_patterns et pg_trgm

But : Indexer des motifs à l'aide de l'opérateur `varchar_pattern_ops` et de l'extension `pg_trgm`

Ces exercices nécessitent une base contenant une quantité de données importante.

On utilisera donc le contenu de livres issus du projet Gutenberg. La base est disponible en deux versions : complète sur https://dali.bo/tp_gutenberg (dump de 0,5 Go, table de 21 millions de lignes dans 3 Go) ou https://dali.bo/tp_gutenberg10 pour un extrait d'un dizième. Le dump peut se restaurer par exemple dans une nouvelle base, et contient juste une table nommée `textes`.

```
curl -kL https://dali.bo/tp_gutenberg -o /tmp/gutenberg.dmp
createdb gutenberg
pg_restore -d gutenberg /tmp/gutenberg.dmp
# le message sur le schéma public exitant est normale
rm -- /tmp/gutenberg.dmp
```

Pour obtenir des plans plus lisibles, on désactive JIT et parallélisme :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
```

Créer un index simple sur la colonne `contenu` de la table.

Rechercher un enregistrement commençant par « comme disent » : l'index est-il utilisé ?

Créer un index utilisant la classe `text_pattern_ops`. Refaire le test.

On veut chercher les lignes finissant par « Et vivre ». Indexer `reverse(contenu)` et trouver les lignes.

Installer l'extension `pg_trgm`, puis créer un index GIN spécialisé de recherche dans les chaînes. Rechercher toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis insensible.

Si vous avez des connaissances sur les expression rationnelles, utilisez aussi ces trigrammes pour des recherches plus avancées. Les opérateurs sont :

opérateur	fonction
~	correspondance sensible à la casse
~*	correspondance insensible à la casse
!~	non-correspondance sensible à la casse
!~*	non-correspondance insensible à la casse

Rechercher toutes les lignes contenant « Fantine » OU « Valjean » : on peut utiliser une expression rationnelle.

Rechercher toutes les lignes mentionnant à la fois « Fantine » ET « Valjean ». Une formulation d'expression rationnelle simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

1.11.2 Index GIN comme bitmap

But : Comparer l'utilisation des index B-tree et GIN

Ce TP utilise la base de données **tpc**. La base **tpc** (dump de 31 Mo, pour 267 Mo sur le disque au final) et ses utilisateurs peuvent être installés comme suit :

```
curl -kL https://dali.bo/tp_tpc -o /tmp/tpc.dump
curl -kL https://dali.bo/tp_tpc_roles -o /tmp/tpc_roles.sql
# Exécuter le script de création des rôles
psql < /tmp/tpc_roles.sql
# Création de la base
createdb --owner tpc_owner tpc
# L'erreur sur un schéma 'public' existant est normale
pg_restore -d tpc /tmp/tpc.dump
rm -- /tmp/tpc.dump /tmp/tpc_roles.sql
```

Les mots de passe sont dans le script `/tmp/tpc_roles.sql`. Pour vous connecter :

```
$ psql -U tpc_admin -h localhost -d tpc
```

Créer deux index sur `lignes_commandes(quantite)` :

- un de type B-tree
- et un GIN.

- puis deux autres sur `lignes_commandes(fournisseur_id)`.

Comparer leur taille.

Comparer l'utilisation des deux types d'index avec `EXPLAIN` avec des requêtes sur `fournisseur_id = 1014`, puis sur `quantite = 4`.

1.11.3 Index GIN et critères multicolonnes

But : Comparer l'utilisation des index B-tree et GIN sur des critères multicolonnes

Créer une table avec 4 colonnes de 50 valeurs :

```
CREATE UNLOGGED TABLE ijkl
AS SELECT i,j,k,l
FROM generate_series(1,50) i
CROSS JOIN generate_series (1,50) j
CROSS JOIN generate_series(1,50) k
CROSS JOIN generate_series (1,50) l ;
```

Créer un index B-tree et un index GIN sur ces 4 colonnes.

Comparer l'utilisation pour des requêtes portant sur `i`, `j`, `k` & `l`, puis `i` & `k`, puis `j` & `l`.

1.11.4 HypoPG

But : Créer un index hypothétique avec HypoPG

Pour la clarté des plans, désactiver le JIT.

Créer la table suivante, où la clé `i` est très déséquilibrée :

```
CREATE UNLOGGED TABLE log10 AS
SELECT i,j, i+10 AS k,  now() AS d, lpad(' ',300,' ') AS filler
FROM generate_series (0,7) i,
LATERAL (SELECT * FROM generate_series(1, power(10,i)::bigint ) j ) jj ;
```

(Ne pas oublier `VACUUM ANALYZE`.)

- On se demande si créer un index sur `i`, `(i,j)` ou `(j,i)` serait utile pour les deux requêtes suivantes :

```
SELECT i, min(j), max(j) FROM log10 GROUP BY i ;
SELECT max(j) FROM log10 WHERE i = 6 ;
```

- Installer l'extension `hypopg` (paquets `hypopg_14` ou `postgresql-14-hypopg`).
- Créer des index hypothétiques (y compris un partiel) et choisir un seul index.

Comparer le plan de la deuxième requête avant et après la création réelle de l'index.

Créer un index fonctionnel hypothétique pour faciliter la requête suivante :

```
SELECT k FROM log10 WHERE mod(j,99) = 55 ;
```

Quel que soit le résultat, le créer quand même et voir s'il est utilisé.

1.12 TRAVAUX PRATIQUES (SOLUTIONS)

1.12.1 Indexation de motifs avec les varchar_patterns et pg_trgm

> Créer un index simple sur la colonne `contenu` de la table.

```
CREATE INDEX ON textes(contenu);
```

Il y aura une erreur si la base `textes` est dans sa version complète, un livre de Marcel Proust dépasse la taille indexable maximale :

```
ERROR:  index row size 2968 exceeds maximum 2712 for index "textes_contenu_idx"
ASTUCE : Values larger than 1/3 of a buffer page cannot be indexed.
Consider a function index of an MD5 hash of the value, or use full text indexing.
```

Pour l'exercice, on supprime ce livre avant d'indexer la colonne :

```
DELETE FROM textes where livre = 'Les Demi-Vierges, Prévost, Marcel';
CREATE INDEX ON textes(contenu);
```

> Rechercher un enregistrement commençant par « comme disent » : l'index est-il utilisé ?

Le plan exact peut dépendre de la version de PostgreSQL, du paramétrage exact, d'éventuelles modifications à la table. Dans beaucoup de cas, on obtiendra :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
VACUUM ANALYZE textes;

EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on textes  (cost=0.00..669657.38 rows=1668 width=124)
            (actual time=305.848..6275.845 rows=47 loops=1)
   Filter: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Filter: 20945503
 Planning Time: 1.033 ms
 Execution Time: 6275.957 ms
```

C'est un `Seq Scan` : l'index n'est pas utilisé !

Dans d'autres cas, on aura ceci (avec PostgreSQL 12 et la version complète de la base ici) :

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';

                            QUERY PLAN
------------------------------------------------------------------
 Index Scan using textes_contenu_idx on textes (…)
   Index Cond: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Index Recheck: 110
   Buffers: shared hit=28 read=49279
   I/O Timings: read=311238.192
 Planning Time: 0.352 ms
 Execution Time: 313481.602 ms
```

C'est un `Index Scan` mais il ne faut pas crier victoire : l'index est parcouru entièrement (50 000 blocs !). Il ne sert qu'à lire toutes les valeurs de `contenu` en lisant moins de blocs que par un `Seq Scan` de la table. Le choix de PostgreSQL entre lire cet index et lire la table dépend notamment du paramétrage et des tailles respectives.

Le problème est que l'index sur `contenu` utilise la collation `C` et non la collation par défaut de la base, généralement `en_US.UTF-8` ou `fr_FR.UTF-8`. Pour contourner cette limitation, PostgreSQL fournit deux classes d'opérateurs : `varchar_pattern_ops` pour `varchar` et `text_pattern_ops` pour `text`.

> Créer un index utilisant la classe `text_pattern_ops`. Refaire le test.

```
DROP INDEX textes_contenu_idx;
CREATE INDEX ON textes(contenu text_pattern_ops);

EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using textes_contenu_idx1 on textes
                               (cost=0.56..8.58 rows=185 width=130)
                        (actual time=0.530..0.542 rows=4 loops=1)
   Index Cond: ((contenu ~>=~ 'comme disent'::text)
            AND (contenu ~<~ 'comme disenu'::text))
   Filter: (contenu ~~ 'comme disent%'::text)
   Buffers: shared hit=4 read=4
 Planning Time: 1.112 ms
 Execution Time: 0.618 ms
```

On constate que comme l'ordre choisi est l'ordre ASCII, l'optimiseur sait qu'après « comme disent », c'est « comme disenu » qui apparaît dans l'index.

Noter que `Index Cond` contient le filtre utilisé pour l'index (réexprimé sous forme d'inégalités en collation `C`) et `Filter` un filtrage des résultats de l'index.

> On veut chercher les lignes finissant par « Et vivre ». Indexer `reverse(contenu)` et trouver les lignes.

Cette recherche n'est possible avec un index B-Tree qu'en utilisant un index sur fonction :

```
CREATE INDEX ON textes(reverse(contenu) text_pattern_ops);
```

Il faut ensuite utiliser ce `reverse` systématiquement dans les requêtes :

```
EXPLAIN (ANALYZE)
SELECT * FROM textes WHERE reverse(contenu) LIKE reverse('%Et vivre') ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using textes_reverse_idx on textes
```

```
                          (cost=0.56..377770.76 rows=104728 width=123)
                       (actual time=0.083..0.098 rows=2 loops=1)
   Index Cond: ((reverse(contenu) ~>=~ 'erviv tE'::text)
            AND (reverse(contenu) ~<~ 'erviv tF'::text))
   Filter: (reverse(contenu) ~~ 'erviv tE%'::text)
 Planning Time: 1.903 ms
 Execution Time: 0.421 ms
```

On constate que le résultat de `reverse(contenu)` a été directement utilisé par l'optimiseur. La requête est donc très rapide. On peut utiliser une méthode similaire pour la recherche insensible à la casse, en utiliser `lower()` ou `upper()`.

Toutefois, ces méthodes ne permettent de filtrer qu'au début ou à la fin de la chaîne, ne permettent qu'une recherche sensible ou insensible à la casse, mais pas les deux simultanément, et imposent aux développeurs de préciser `reverse`, `lower`, etc. partout.

> Installer l'extension `pg_trgm`, puis créer un index GIN spécialisé de recherche dans les chaînes. Rechercher toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis insensible.

Pour installer l'extension `pg_trgm` :

```
CREATE EXTENSION pg_trgm;
```

Pour créer un index GIN sur la colonne `contenu` :

```
CREATE INDEX idx_textes_trgm ON textes USING gin (contenu gin_trgm_ops);
```

Recherche des lignes contenant « Valjean » de façon sensible à la casse :

```
EXPLAIN (ANALYZE)
SELECT * FROM textes WHERE contenu LIKE '%Valjean%' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=77.01..6479.68 rows=1679 width=123)
                      (actual time=11.004..14.769 rows=1213 loops=1)
   Recheck Cond: (contenu ~~ '%Valjean%'::text)
   Rows Removed by Index Recheck: 1
   Heap Blocks: exact=353
   ->  Bitmap Index Scan on idx_textes_trgm
                               (cost=0.00..76.59 rows=1679 width=0)
                      (actual time=10.797..10.797 rows=1214 loops=1)
         Index Cond: (contenu ~~ '%Valjean%'::text)
 Planning Time: 0.815 ms
 Execution Time: 15.122 ms
```

Puis insensible à la casse :

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ILIKE '%Valjean%';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=77.01..6479.68 rows=1679 width=123)
```

```
                     (actual time=13.135..23.145 rows=1214 loops=1)
   Recheck Cond: (contenu ~~* '%Valjean%'::text)
   Heap Blocks: exact=353
   ->  Bitmap Index Scan on idx_textes_trgm
                                (cost=0.00..76.59 rows=1679 width=0)
                      (actual time=12.779..12.779 rows=1214 loops=1)
         Index Cond: (contenu ~~* '%Valjean%'::text)
 Planning Time: 2.047 ms
 Execution Time: 23.444 ms
```

On constate que l'index a été nettement plus long à créer, et que la recherche est plus lente. La contrepartie est évidemment que les trigrammes sont infiniment plus souples. On constate aussi que le `LIKE` a dû encore filtrer 1 enregistrement après le parcours de l'index : en effet l'index trigramme est insensible à la casse, il ramène donc trop d'enregistrements, et une ligne avec « VALJEAN » a dû être filtrée.

> Rechercher toutes les lignes contenant « Fantine » OU « Valjean » : on peut utiliser une expression rationnelle.

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ~ 'Valjean|Fantine';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=141.01..6543.68 rows=1679 width=123)
                     (actual time=159.896..174.173 rows=1439 loops=1)
   Recheck Cond: (contenu ~ 'Valjean|Fantine'::text)
   Rows Removed by Index Recheck: 1569
   Heap Blocks: exact=1955
   ->  Bitmap Index Scan on idx_textes_trgm
                                (cost=0.00..140.59 rows=1679 width=0)
                      (actual time=159.135..159.135 rows=3008 loops=1)
         Index Cond: (contenu ~ 'Valjean|Fantine'::text)
 Planning Time: 2.467 ms
 Execution Time: 174.284 ms
```

> Rechercher toutes les lignes mentionnant à la fois « Fantine » ET « Valjean ». Une formulation d'expression rationnelle simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

```
EXPLAIN ANALYZE SELECT * FROM textes
  WHERE contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)' ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=141.01..6543.68 rows=1679 width=123)
                       (actual time=26.825..26.897 rows=8 loops=1)
   Recheck Cond: (contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)'::text)
   Heap Blocks: exact=6
   ->  Bitmap Index Scan on idx_textes_trgm
                                  (cost=0.00..140.59 rows=1679 width=0)
                         (actual time=26.791..26.791 rows=8 loops=1)
         Index Cond: (contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)'::text)
 Planning Time: 5.697 ms
 Execution Time: 26.992 ms
```

1.12.2 Index GIN comme bitmap

Ce TP utilise la base **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin  /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

> Créer deux index sur `lignes_commandes(quantite)` :
>
> - un de type B-tree
> - et un GIN.
> - puis deux autres sur `lignes_commandes(fournisseur_id)`.

Il est nécessaire d'utiliser l'extension `btree_gin` afin d'indexer des types scalaires (`int`…) avec un GIN :

```
CREATE INDEX ON lignes_commandes USING gin (quantite);

ERROR:  data type bigint has no default operator class for access method "gin"
HINT:  You must specify an operator class for the index
    or define a default operator class for the data type.

CREATE EXTENSION btree_gin;

CREATE INDEX lignes_commandes_quantite_gin ON lignes_commandes
    USING gin (quantite) ;
CREATE INDEX lignes_commandes_quantite_btree ON lignes_commandes
    (quantite) ;  -- implicitement B-tree

CREATE INDEX lignes_commandes_fournisseur_id_gin ON lignes_commandes
    USING gin (fournisseur_id) ;
CREATE INDEX lignes_commandes_fournisseur_id_btree ON lignes_commandes
    (fournisseur_id) ; -- implicitement B-tree
```

> Comparer leur taille.

Ces index sont compressés, ainsi la clé n'est indexée qu'une fois. Le gain est donc intéressant dès lors que la table comprend des valeurs identiques.

```
SELECT indexname, pg_size_pretty(pg_relation_size(indexname::regclass))
FROM pg_indexes
WHERE indexname LIKE 'lignes_commandes%'
ORDER BY 1 ;

               indexname               | pg_size_pretty
---------------------------------------+----------------
 lignes_commandes_fournisseur_id_btree | 22 MB
```

```
 lignes_commandes_fournisseur_id_gin   | 15 MB
 lignes_commandes_pkey                 | 94 MB
 lignes_commandes_quantite_btree       | 21 MB
 lignes_commandes_quantite_gin         | 3848 kB
```

(Ces valeurs ont été obtenues avec PostgreSQL 13. Une version antérieure affichera des index B-tree nettement plus gros, car le stockage des valeurs dupliquées y est moins efficace. Les index GIN seront donc d'autant plus intéressants.)

Noter qu'il y a peu de valeurs différentes de `quantite`, et beaucoup plus de `fournisseur_id`, ce qui explique les différences de tailles d'index :

```
SELECT COUNT(DISTINCT fournisseur_id), COUNT(DISTINCT quantite)
  FROM magasin.lignes_commandes ;

 count | count
-------+-------
  1811 |    10
```

Les index GIN sont donc plus compacts. Sont-il plus efficace à l'utilisation ?

> Comparer l'utilisation des deux types d'index avec `EXPLAIN` avec des requêtes sur `fournisseur_id = 1014`, puis sur `quantite = 4`.

Pour récupérer une valeur précise avec peu de valeurs, PostgreSQL préfère les index B-Tree :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM lignes_commandes
  WHERE fournisseur_id = 1014 ;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on lignes_commandes  (cost=20.97..5469.07 rows=1618 width=74)
                                       (actual time=0.320..9.132 rows=1610 loops=1)
   Recheck Cond: (fournisseur_id = 1014)
   Heap Blocks: exact=1585
   Buffers: shared hit=407 read=1185
   ->  Bitmap Index Scan on lignes_commandes_fournisseur_id_btree
                                       (cost=0.00..20.56 rows=1618 width=0)
                                       (actual time=0.152..0.152 rows=1610 loops=1)
         Index Cond: (fournisseur_id = 1014)
         Buffers: shared hit=3 read=4
 Planning:
   Buffers: shared hit=146 read=9
 Planning Time: 2.269 ms
 Execution Time: 9.250 ms
```

À l'inverse, la recherche sur les quantités ramène plus de valeurs, et, là, PostgreSQL estime que le GIN est plus favorable :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM lignes_commandes WHERE quantite = 4 ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on lignes_commandes  (cost=2844.76..48899.60 rows=308227 width=74)
                                    (actual time=37.904..293.243 rows=313674 loops=1)
   Recheck Cond: (quantite = 4)
   Heap Blocks: exact=42194
   Buffers: shared hit=532 read=41712 written=476
   ->  Bitmap Index Scan on lignes_commandes_quantite_gin
                                       (cost=0.00..2767.70 rows=308227 width=0)
                                     (actual time=31.164..31.165 rows=313674 loops=1)
         Index Cond: (quantite = 4)
         Buffers: shared hit=50
 Planning:
   Buffers: shared hit=13
 Planning Time: 0.198 ms
 Execution Time: 305.030 ms
```

En cas de suppression de l'index GIN, l'index B-Tree reste utilisable. Il sera plus long à lire. Cependant, en fonction de sa taille, de celle de la table, de la valeur de `random_page_cost` et `seq_page_cost`, PostgreSQL peut décider de ne pas l'utiliser.

1.12.3 Index GIN et critères multicolonnes

> Créer une table avec 4 colonnes de 50 valeurs :
>
> ```
> CREATE UNLOGGED TABLE ijkl
> AS SELECT i,j,k,l
> FROM generate_series(1,50) i
> CROSS JOIN generate_series (1,50) j
> CROSS JOIN generate_series(1,50) k
> CROSS JOIN generate_series (1,50) l ;
> ```

Les 264 Mo de cette table contiennent 6,25 millions de lignes.

Comme après tout import, ne pas oublier d'exécuter un `VACUUM` :

```
VACUUM ijkl;
```

> Créer un index B-tree et un index GIN sur ces 4 colonnes.

Là encore, `btree_gin` est obligatoire pour indexer un type scalaire, et on constate que l'index GIN est plus compact :

```
CREATE INDEX ijkl_btree on ijkl (i,j,k,l) ;
CREATE EXTENSION btree_gin ;
CREATE INDEX ijkl_gin on ijkl USING gin (i,j,k,l) ;
```

```
# \di+ ijkl*
                              Liste des relations
 Schéma |    Nom     | Type  | Propriétaire | Table | Taille | Description
--------+------------+-------+--------------+-------+--------+-------------
 public | ijkl_btree | index | postgres     | ijkl  | 188 MB |
 public | ijkl_gin   | index | postgres     | ijkl  | 30 MB  |
```

> Comparer l'utilisation pour des requêtes portant sur `i`, `j`, `k` & `l`, puis `i` & `k`, puis `j` & `l`.

Le premier critère porte idéalement sur toutes les colonnes de l'index B-tree : celui-ci est très efficace et réclame peu d'accès. Toutes les colonnes retournées font partie de l'index : on profite donc en plus d'un `Index Only Scan` :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
  SELECT * FROM ijkl WHERE i=10 AND j=20 AND k=30 AND l=40 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using ijkl_btree on ijkl (… rows=1 loops=1)
   Index Cond: ((i = 10) AND (j = 20) AND (k = 30) AND (l = 40))
   Heap Fetches: 0
   Buffers: shared hit=1 read=3
 Planning Time: 0.330 ms
 Execution Time: 0.400 ms
```

Tant que la colonne `i` est présente dans le critère, l'index B-tree reste avantageux même s'il doit balayer plus de blocs (582 ici !) :

```
# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
  SELECT * FROM ijkl WHERE i=10 AND k=30 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using ijkl_btree on ijkl (… rows=2500 loops=1)
   Index Cond: ((i = 10) AND (k = 30))
   Heap Fetches: 0
   Buffers: shared hit=102 read=480 written=104
 Planning Time: 0.284 ms
 Execution Time: 29.452 ms
```

Par contre, dès que la première colonne de l'index B-tree (`i`) manque, celui-ci devient beaucoup moins intéressant (quoique pas inutilisable, mais il y a des chances qu'il faille le parcourir complètement). L'index GIN devient alors intéressant par sa taille réduite :

```
# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
  SELECT * from ijkl WHERE j=20 AND k=40 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on ijkl (… rows=2500 loops=1)
   Recheck Cond: ((j = 20) AND (k = 40))
   Heap Blocks: exact=713
   Buffers: shared hit=119 read=670
   ->  Bitmap Index Scan on ijkl_gin (… rows=2500 loops=1)
         Index Cond: ((j = 20) AND (k = 40))
         Buffers: shared hit=76
 Planning Time: 0.382 ms
 Execution Time: 28.987 ms
```

L'index GIN est obligé d'aller vérifier la visibilité des lignes dans la table, il ne supporte pas les *Index Only Scan*.

Sans lui, la seule alternative serait un *Seq Scan* qui parcourrait les 33 784 blocs de la table :

```
DROP INDEX ijkl_gin ;

EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
  SELECT * from ijkl WHERE j=20 AND k=40 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Gather (actual time=34.861..713.474 rows=2500 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=2260 read=31524
   ->  Parallel Seq Scan on ijkl (… rows=833 loops=3)
         Filter: ((j = 20) AND (k = 40))
         Rows Removed by Filter: 2082500
         Buffers: shared hit=2260 read=31524
 Planning Time: 1.464 ms
 Execution Time: 713.796 ms
```

1.12.4 HypoPG

> Pour la clarté des plans, désactiver le JIT.

```
SET jit TO off ;
```

> Créer la table suivante, où la clé `i` est très déséquilibrée :
>
> ```
> CREATE UNLOGGED TABLE log10 AS
> SELECT i,j, i+10 AS k,  now() AS d, lpad(' ',300,' ') AS filler
> FROM generate_series (0,7) i,
> LATERAL (SELECT * FROM generate_series(1, power(10,i)::bigint ) j ) jj ;
> ```
>
> (Ne pas oublier `VACUUM ANALYZE`.)

```
CREATE UNLOGGED TABLE log10 AS
SELECT i,j, i+10 AS k,  now() AS d, lpad(' ',300,' ') AS filler
FROM generate_series (0,7) i,
LATERAL (SELECT * FROM generate_series(1, power(10,i)::bigint ) j ) jj ;

VACUUM ANALYZE log10 ;
```

Cette table fait 11,1 millions de lignes et presque 4 Go.

> – On se demande si créer un index sur `i`, `(i,j)` ou `(j,i)` serait utile pour les deux requêtes suivantes :
>
> ```
> SELECT i, min(j), max(j) FROM log10 GROUP BY i ;
> SELECT max(j) FROM log10 WHERE i = 6 ;
> ```

- Installer l'extension `hypopg` (paquets `hypopg_14` ou `postgresql-14-hypopg`).
- Créer des index hypothétiques (y compris un partiel) et choisir un seul index.

D'abord installer le paquet de l'extension. Sur Rocky Linux et autres dérivés Red Hat :

```
sudo dnf install hypopg_14
```

Sur Debian et dérivés :

```
sudo apt install postgresql-14-hypopg
```

Puis installer l'extension dans la base concernée :

```
CREATE EXTENSION hypopg;
```

Création des différents index hypothétiques qui pourraient servir :

```
SELECT  hypopg_create_index ('CREATE INDEX ON log10 (i)' );
```

```
     hypopg_create_index
-------------------------------
 (78053,<78053>btree_log10_i)
```

```
SELECT * FROM  hypopg_create_index ('CREATE INDEX ON log10 (i,j)' );
```

```
 indexrelid |       indexname
------------+------------------------
      78054 | <78054>btree_log10_i_j
```

```
SELECT * FROM  hypopg_create_index ('CREATE INDEX ON log10 (j,i)' );
```

```
 indexrelid |       indexname
------------+------------------------
      78055 | <78055>btree_log10_j_i
```

```
SELECT * FROM  hypopg_create_index ('CREATE INDEX ON log10 (j) WHERE i=6' );
```

```
 indexrelid |      indexname
------------+----------------------
      78056 | <78056>btree_log10_j
```

On vérifie qu'ils sont tous actifs dans cette session :

```
SELECT * FROM hypopg_list_indexes;
```

```
 indexrelid |       indexname        | nspname | relname | amname
------------+------------------------+---------+---------+--------
      78053 | <78053>btree_log10_i   | public  | log10   | btree
      78054 | <78054>btree_log10_i_j | public  | log10   | btree
      78055 | <78055>btree_log10_j_i | public  | log10   | btree
      78056 | <78056>btree_log10_j   | public  | log10   | btree
```

```
EXPLAIN SELECT i, min(j), max(j) FROM log10 GROUP BY i ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=1000.08..392727.62 rows=5 width=20)
   Group Key: i
```

```
   ->  Gather Merge  (cost=1000.08..392727.49 rows=10 width=20)
         Workers Planned: 2
         ->  Partial GroupAggregate  (cost=0.06..391726.32 rows=5 width=20)
               Group Key: i
               ->  Parallel Index Only Scan
                   using <78054>btree_log10_i_j on log10
                   (cost=0.06..357004.01 rows=4629634 width=12)
```

```
EXPLAIN SELECT max(j) FROM log10 WHERE i = 6 ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Result  (cost=0.08..0.09 rows=1 width=8)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.05..0.08 rows=1 width=8)
           ->  Index Only Scan Backward using <78056>btree_log10_j on log10
                                  (cost=0.05..31812.59 rows=969631 width=8)
                 Index Cond: (j IS NOT NULL)
```

Les deux requêtes n'utilisent pas le même index. Le partiel (`<78056>btree_log10_j`) ne conviendra évidemment pas à toutes les requêtes, on voit donc ce qui se passe sans lui :

```
SELECT * FROM hypopg_drop_index(78056);
 hypopg_drop_index
-------------------
 t
```

```
EXPLAIN SELECT max(j) FROM log10 WHERE i = 6 ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Result  (cost=0.10..0.11 rows=1 width=8)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.06..0.10 rows=1 width=8)
           ->  Index Only Scan Backward using <78054>btree_log10_i_j on log10
                                     (cost=0.06..41660.68 rows=969631 width=8)
                 Index Cond: ((i = 6) AND (j IS NOT NULL))
```

C'est presque aussi bon. L'index sur `(i,j)` semble donc convenir aux deux requêtes.

> Comparer le plan de la deuxième requête avant et après la création réelle de l'index.

Bien sûr, un `EXPLAIN (ANALYZE)` négligera ces index qui n'existent pas réellement :

```
  EXPLAIN (ANALYZE,TIMING OFF)
  SELECT max(j) FROM log10 WHERE i = 6 ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Finalize Aggregate      (cost=564931.67..564931.68 rows=1 width=8)
                                       (actual rows=1 loops=1)
   ->  Gather  (cost=564931.46..564931.67 rows=2 width=8)
                                  (actual rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
```

```
        ->  Partial Aggregate  (cost=563931.46..563931.47 rows=1 width=8)
                                                  (actual rows=1 loops=3)
              ->  Parallel Seq Scan on log10
                                 (cost=0.00..562921.43 rows=404013 width=8)
                                               (actual rows=333333 loops=3)
                    Filter: (i = 6)
                    Rows Removed by Filter: 3370370
 Planning Time: 0.414 ms
 Execution Time: 2322.879 ms
```

```
CREATE INDEX ON log10 (i,j) ;
```

Et le nouveau plan est cohérent avec l'estimation d'HypoPG :

```
 Result  (cost=0.60..0.61 rows=1 width=8) (actual rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.56..0.60 rows=1 width=8) (actual rows=1 loops=1)
           ->  Index Only Scan Backward using log10_i_j_idx on log10
               (cost=0.56..34329.16 rows=969630 width=8) (actual rows=1 loops=1)
                 Index Cond: ((i = 6) AND (j IS NOT NULL))
                 Heap Fetches: 0
 Planning Time: 1.070 ms
 Execution Time: 0.239 ms
```

> Créer un index fonctionnel hypothétique pour faciliter la requête suivante :
>
> ```
> SELECT k FROM log10 WHERE mod(j,99) = 55 ;
> ```
>
> Quel que soit le résultat, le créer quand même et voir s'il est utilisé.

Si on simule la présence de cet index fonctionnel :

```
SELECT * FROM  hypopg_create_index ('CREATE INDEX ON log10 ( mod(j,99))');
EXPLAIN SELECT k FROM log10 WHERE mod(j,99) = 55 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Gather  (cost=1000.00..581051.11 rows=55556 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on log10  (cost=0.00..574495.51 rows=23148 width=4)
         Filter: (mod(j, '99'::bigint) = 55)
```

on constate que l'optimiseur le néglige.

Si on le crée quand même, sans oublier de mettre à jour les statistiques :

```
CREATE INDEX ON log10 ( mod(j,99) ) ;
ANALYZE log10 ;
```

on constate qu'il est alors utilisé :

```
 Gather  (cost=3243.57..343783.72 rows=119630 width=4) (actual rows=112233 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Bitmap Heap Scan on log10
                          (cost=2243.57..330820.72 rows=49846 width=4)
```

```
                                        (actual rows=37411 loops=3)
        Recheck Cond: (mod(j, '99'::bigint) = 55)
        Rows Removed by Index Recheck: 470869
        Heap Blocks: exact=18299 lossy=23430
        ->  Bitmap Index Scan on log10_mod_idx
                              (cost=0.00..2213.66 rows=119630 width=0)
                                        (actual rows=112233 loops=1)
              Index Cond: (mod(j, '99'::bigint) = 55)
 Planning Time: 1.216 ms
 Execution Time: 541.668 ms
```

La différence tient à la volumétrie attendue qui a doublé après l'`ANALYZE` : il y a à présent des statistiques sur les résultats de la fonction qui n'étaient pas disponibles sans la création de l'index, comme on peut le constater avec :

```
SELECT * FROM pg_stats WHERE tablename = 'log10' ;
```

NB : on peut penser aussi à un index couvrant :

```
SELECT * FROM
hypopg_create_index (
  'CREATE INDEX ON log10 ( mod(j,99) ) INCLUDE(k)'
);
```

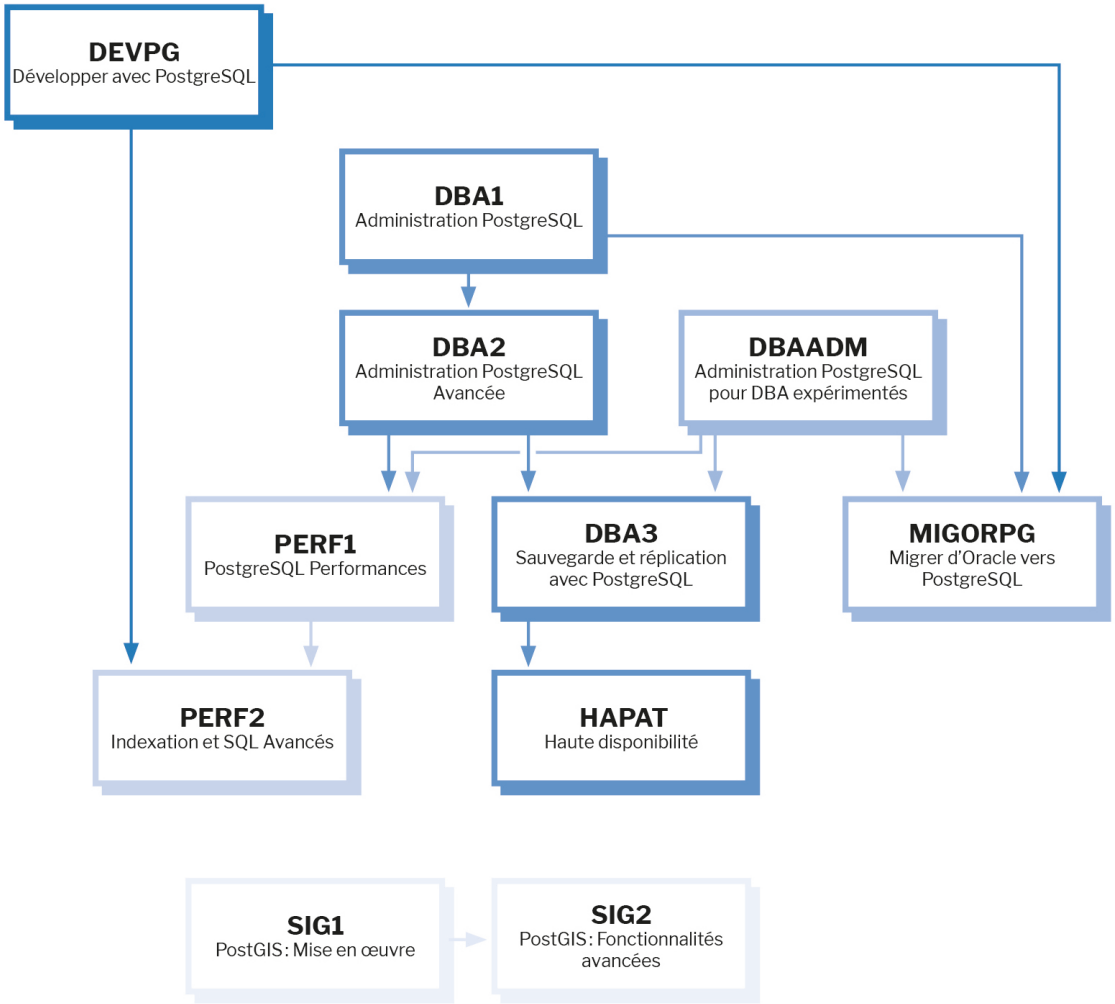
en fonction des requêtes réelles à optimiser.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.