Module T1

Fonctionnalités avancées pour la performance

25.03

Table des matières

Sur ce document

Formation	Module T1
Titre	Fonctionnalités avancées pour la performance
Révision	25.03
PDF	https://dali.bo/t1_pdf
EPUB	https://dali.bo/t1_epub
HTML	https://dali.bo/t1_html
Slides	https://dali.bo/t1_slides
TP	https://dali.bo/t1_tp
TP (solutions)	https://dali.bo/t1_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 :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, 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, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, 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

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 13 à 17.

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/ Fonctionnalités avancées pour la performance

1.1 PRÉAMBULE

Quelques fonctionnalités, généralement liées aux performances.

1.1.1 Au menu

- Tables temporaires
- Tables non journalisées
- Colonnes générées
- JIT
- Recherche Full Text

1.2 TABLES TEMPORAIRES

1.2.1 Tables temporaires : utilité

```
CREATE TEMP TABLE  travail (…) ;
```

- N'existent que pendant la session
- Non journalisées : rapides !

Principe :

Sous PostgreSQL, les tables temporaires sont créées dans une session, et disparaissent à la déconnexion. Elles ne sont pas visibles par les autres sessions. Elles ne sont pas journalisées, ce qui est très intéressant pour les performances. Elles s'utilisent comme les autres tables, y compris pour l'indexation, les triggers, etc.

Les tables temporaires semblent donc idéales pour des tables de travail temporaires et « jetables ».

1.2.2 Tables temporaires : limites et paramétrage

- Ne pas en abuser !
- Ignorées par autovacuum
  - `ANALYZE` et `VACUUM` manuels !
- Paramétrage :
  - `temp_buffers` : cache disque pour les objets temporaires, par session, à augmenter ?

Cependant, il est déconseillé d'abuser des tables temporaires. En effet, leur création/destruction permanente entraîne une fragmentation importante des tables systèmes (en premier lieu `pg_catalog.pg_class`, `pg_catalog.pg_attribute`…), qui peuvent devenir énormes. Ce n'est jamais bon pour les performances, et peut nécessiter un `VACUUM FULL` des tables système !

Le démon autovacuum ne voit pas les tables temporaires ! Les statistiques devront donc être mises à jour manuellement avec `ANALYZE`, et il faudra penser à lancer `VACUUM` explicitement après de grosses modifications.

Aspect technique :

Les tables temporaires sont créées dans un schéma temporaire `pg_temp_…`, ce qui explique qu'elles ne sont pas visibles dans le schéma `public`.

Physiquement, par défaut, elles sont stockées sur le disque avec les autres données de la base, et non dans `base/pgsql_tmp` comme les fichiers temporaires. Il est possible de définir des tablespaces dédiés aux objets temporaires (fichiers temporaires et données des tables temporaires) à l'aide du paramètre `temp_tablespaces`, à condition de donner des droits `CREATE` dessus aux utilisateurs. Le nom du fichier d'une table temporaire est reconnaissable car il commence par `t`. Les éventuels index de la table suivent les même règles.

Exemple :

```
CREATE TEMP TABLE travail (x int PRIMARY KEY) ;

EXPLAIN (COSTS OFF, ANALYZE, BUFFERS, WAL)
INSERT INTO travail SELECT i FROM generate_series (1,1000000) i ;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Insert on travail (actual time=1025.752..1025.753 rows=0 loops=1)
   Buffers: shared hit=13, local hit=2172174 read=4 dirtied=7170 written=10246
   I/O Timings: read=0.012
   ->  Function Scan on generate_series i (actual time=77.112..135.624 rows=1000000
↪  loops=1)
 Planning Time: 0.028 ms
 Execution Time: 1034.984 ms

SELECT pg_relation_filepath ('travail') ;

 pg_relation_filepath
----------------------
 base/13746/t7_5148873

\d pg_temp_7.travail

                Table « pg_temp_7.travail »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 x       | integer |                 | not null  |
Index :
    "travail_pkey" PRIMARY KEY, btree (x)
```

Cache :

Dans les plans d'exécution avec `BUFFERS`, l'habituelle mention `shared` est remplacée par `local` pour les tables temporaires.

```
CREATE TEMP TABLE demotemp AS SELECT * FROM pg_class ;

EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT * FROM demotemp ;

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on demotemp (actual time=0.013..0.109 rows=602 loops=1)
   Buffers: local hit=16
 Planning:
   Buffers: shared hit=66 read=1
   I/O Timings: shared read=0.009
 Planning Time: 0.250 ms
 Execution Time: 0.170 ms
```

(Les `shared hit` concernent des accès aux tables systèmes pour la planification.)

En effet, leur cache disque dédié est au niveau de la session, non des *shared buffers*. Ce cache est défini par le paramètre `temp_buffers` (exprimé par session, et à 8 Mo par défaut). Ce paramètre peut être augmenté dans la session, avant la création de la table. Bien sûr, on risque de saturer la RAM en cas d'abus ou s'il y a trop de sessions, tout comme avec `work_mem`. Et comme la mémoire de `temp_buffers` n'est pas rendue avant la fin de la session, il faut éviter de maintenir inutilement ouvertes des sessions ayant créé des tables temporaires : elles bloquent de la mémoire.

À noter : ce cache n'empêche pas l'écriture des petites tables temporaires sur le disque.

Pour éviter de recréer perpétuellement la même table temporaire, une table *unlogged* (voir plus bas) sera sans doute plus indiquée. Le contenu de cette dernière sera aussi visible des autres sessions, ce qui est pratique pour suivre la progression d'un traitement, faciliter le travail de l'autovacuum, ou déboguer. Sinon, il est fréquent de pouvoir remplacer une table temporaire par une CTE (clause `WITH`) ou un tableau en mémoire.

L'extension pgtt[1] émule un autre type de table temporaire dite « globale » pour la compatibilité avec d'autres SGBD.

[1] https://github.com/darold/pgtt

1.3 TABLES NON JOURNALISÉES (UNLOGGED)

1.3.1 Tables non journalisées : utilité

- La durabilité est parfois accessoire :
  - tables temporaires et de travail
  - caches…
- Tables non journalisées
  - non répliquées, non restaurées
  - **remises à zéro en cas de crash**
- Respecter les contraintes

Une table *unlogged* est une table non journalisée. Comme la journalisation est responsable de la durabilité, une table non journalisée n'a pas cette garantie.

La table est systématiquement remise à zéro au redémarrage après un arrêt brutal. En effet, tout arrêt d'urgence peut entraîner une corruption des fichiers de la table ; et sans journalisation, il ne serait pas possible de la corriger au redémarrage et de garantir l'intégrité.

La non-journalisation de la table implique aussi que ses données ne sont pas répliquées vers des serveurs secondaires, et que les tables ne peuvent figurer dans une publication (réplication logique). En effet, les modes de réplication natifs de PostgreSQL utilisent les journaux de transactions. Pour la même raison, une restauration de sauvegarde PITR ne restaurera pas le contenu de la table. Le bon côté est qu'on allège la charge sur la sauvegarde et la réplication.

Les contraintes doivent être respectées même si la table *unlogged* est vidée : une table normale ne peut donc avoir de clé étrangère pointant vers une table *unlogged*. La contrainte inverse est possible, tout comme une contrainte entre deux tables *unlogged*.

À part ces limitations, les tables *unlogged* se comportent exactement comme les autres. Leur intérêt principal est d'être en moyenne 5 fois plus rapides à la mise à jour. Elles sont donc à réserver à des cas d'utilisation particuliers, comme :

- table de *spooling*/*staging* ;
- table de cache/session applicative ;
- table de travail partagée entre sessions ;
- table de travail systématiquement reconstruite avant utilisation dans le flux applicatif ;

- et de manière générale toute table contenant des données dont on peut accepter la perte sans impact opérationnel ou dont on peut regénérer aisément les données.

Les tables *unlogged* ne doivent pas être confondues avec les tables temporaires (non journalisées et visibles uniquement dans la session qui les a créées). Les tables *unlogged* ne sont pas ignorées par l'autovacuum (les tables temporaires le sont). Abuser des tables temporaires a tendance à générer de la fragmentation dans les tables système, alors que les tables *unlogged* sont en général créées une fois pour toutes.

1.3.2 Tables non journalisées : mise en place

```
CREATE UNLOGGED TABLE ma_table (col1 int …) ;
```

Une table *unlogged* se crée exactement comme une table journalisée classique, excepté qu'on rajoute le mot `UNLOGGED` dans la création.

1.3.3 Bascule d'une table en/depuis unlogged

```
ALTER TABLE table_normale SET UNLOGGED ;
```

- réécriture

```
ALTER TABLE table_unlogged SET LOGGED ;
```

- passage du contenu dans les WAL !

Il est possible de basculer une table à volonté de normale à *unlogged* et vice-versa.

Quand une table devient *unlogged*, on pourrait imaginer que PostgreSQL n'a rien besoin d'écrire. Malheureusement, pour des raisons techniques, la table doit tout de même être réécrite. Elle est défragmentée au passage, comme lors d'un `VACUUM FULL`. Ce peut être long pour une grosse table, et il faudra voir si le gain par la suite le justifie.

Les écritures dans les journaux à ce moment sont théoriquement inutiles, mais là encore des optimisations manquent et il se peut que de nombreux journaux soient écrits si les sommes de contrôles ou `wal_log_hints` sont activés. Par contre il n'y aura plus d'écritures dans les journaux lors des modifications de cette table, ce qui reste l'intérêt majeur.

Quand une table *unlogged* devient *logged* (journalisée), la réécriture a aussi lieu, et tout le contenu de la table est journalisé (c'est indispensable pour la sauvegarde PITR et pour la réplication notamment), ce qui génère énormément de journaux et peut prendre du temps.

Par exemple, une table modifiée de manière répétée pendant un batch, peut être définie *unlogged* pour des raisons de performance, puis basculée en *logged* en fin de traitement pour pérenniser son contenu.

1.4 COLONNES GÉNÉRÉES

1.4.1 Colonnes générées : principe

```
CREATE TABLE paquet (
  code       text          PRIMARY KEY, …
  livraison  timestamptz   DEFAULT now() + interval '3d',
  largeur    int,    longueur int,   profondeur int,
  volume     int
      GENERATED ALWAYS AS ( largeur * longueur * profondeur )
      STORED    CHECK (volume > 0.0)
  ) ;

-- Modification (PG17)
ALTER TABLE paquet ALTER COLUMN volume SET EXPRESSION AS …
--Réécriture !
```

1.4.2 Colonnes générées vs DEFAULT

- `DEFAULT`
  - expressions très simples, modifiables
- `GENERATED`
  - fonctions « immutables », ne dépendant **que** de la ligne
  - danger sinon (ex : pas pour dates de mises à jour)
  - (avant v17) difficilement modifiables
  - peuvent porter des contraintes

Les valeurs par défaut sont très connues mais limitées. PostgreSQL connaît aussi les colonnes générées (ou calculées).

La syntaxe est :

```
nomchamp  <type> GENERATED ALWAYS AS ( <expression> ) STORED ;
```

Les colonnes générées sont recalculées à chaque fois que les champs sur lesquels elles sont basées changent, donc aussi lors d'un `UPDATE` Ces champs calculés sont impérativement marqués `ALWAYS`,

c'est-à-dire obligatoires et non modifiables, et `STORED`, c'est-à-dire stockés sur le disque (et non re-calculés à la volée comme dans une vue). Ils ne doivent pas se baser sur d'autres champs calculés.

Un intérêt est que les champs calculés peuvent porter des contraintes, par exemple la clause `CHECK` ci-dessous, mais encore des clés étrangères ou unique.

Exemple :

```
CREATE TABLE paquet (
  code       text          PRIMARY KEY,
  reception  timestamptz   DEFAULT now(),
  livraison  timestamptz   DEFAULT now() + interval '3d',
  largeur    int,    longueur int,   profondeur int,
  volume     int
      GENERATED ALWAYS AS ( largeur * longueur * profondeur )
      STORED    CHECK (volume > 0.0)
  ) ;

INSERT INTO paquet (code, largeur, longueur, profondeur)
VALUES ('ZZ1', 3, 5, 10) ;

\x on

TABLE paquet ;

-[ RECORD 1 ]----------------------------
code       | ZZ1
reception  | 2024-04-19 18:02:41.021444+02
livraison  | 2024-04-22 18:02:41.021444+02
largeur    | 3
longueur   | 5
profondeur | 10
volume     | 150

-- Les champs DEFAULT sont modifiables
-- Changer la largeur va modifier le volume
UPDATE paquet
SET largeur=4,
    livraison = '2024-07-14'::timestamptz,
    reception = '2024-04-20'::timestamptz
WHERE code='ZZ1' ;

TABLE paquet ;

-[ RECORD 1 ]----------------------
code       | ZZ1
reception  | 2024-04-20 00:00:00+02
livraison  | 2024-07-14 00:00:00+02
largeur    | 4
longueur   | 5
profondeur | 10
volume     | 200

-- Le volume ne peut être modifié
UPDATE paquet
SET volume = 250
WHERE code = 'ZZ1' ;
```

```
ERROR:  column "volume" can only be updated to DEFAULT
DETAIL : Column "volume" is a generated column.
```

Expression immutable :

Avec `GENERATED`, l'expression du calcul doit être « immutable », c'est-à-dire ne dépendre **que** des autres champs de la même ligne, n'utiliser que des fonctions elles-mêmes immutables, et rien d'autre. Il n'est donc pas possible d'utiliser des fonctions comme `now()`, ni des fonctions de conversion de date dépendant du fuseau horaire, ou du paramètre de formatage de la session en cours (toutes choses autorisées avec `DEFAULT`), ni des appels à d'autres lignes ou tables…

La colonne calculée peut être convertie en colonne « normale » :

```
ALTER TABLE paquet ALTER COLUMN volume DROP EXPRESSION ;
```

Modifier l'expression n'est pas possible avant PostgreSQL 17, sauf à supprimer la colonne générée et en créer une nouvelle. Il faut alors recalculer toutes les lignes et réécrire toute la table, ce qui peut être très lourd.

À partir de PostgreSQL 17, l'expression est modifiable avec cette syntaxe :

```
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS  ( largeur * longueur * profondeur + 1 ) ;
```

Attention, la table est totalement bloquée le temps de la réécriture (verrou `AccessExclusiveLock`).

Utilisation d'une fonction :

Il est possible de créer sa propre fonction pour l'expression, qui doit aussi être immutable :

```
CREATE OR REPLACE FUNCTION volume (l int, h int, p int)
RETURNS int
AS $$
 SELECT l * h * p ;
$$
LANGUAGE sql
-- cette fonction dépend uniquement des données de la ligne donc :
PARALLEL SAFE
IMMUTABLE ;

-- Changement à partir de PostgreSQL v17
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS  ( volume (largeur, longueur, profondeur) );

-- Changement avant PostgreSQL 16
ALTER TABLE paquet DROP COLUMN volume ;
ALTER TABLE paquet ADD COLUMN volume int
  GENERATED ALWAYS AS ( volume (largeur, longueur, profondeur) )
  STORED;

TABLE paquet ;

-[ RECORD 1 ]----------------------
code       | ZZ1
reception  | 2024-04-20 00:00:00+02
livraison  | 2024-07-14 00:00:00+02
```

```
largeur     | 4
longueur    | 5
profondeur | 10
volume      | 200
```

Attention : modifier la fonction ne réécrit pas spontanément la table, il faut forcer la réécriture avec par exemple :

```
UPDATE paquet SET longueur = longueur ;
```

et ceci dans la même transaction que la modification de fonction. On pourrait imaginer de négliger cet `UPDATE` pour garder les valeurs déjà présentes qui suivraient d'anciennes règles… mais ce serait une erreur. En effet, les valeurs calculées ne figurent pas dans une sauvegarde logique, et en cas de restauration, tous les champs sont recalculés avec la dernière formule !

On préférera donc gérer l'expression dans la définition de la table dans les cas simples.

Un autre piège : il faut résister à la tentation de déclarer une fonction comme immutable sans la certitude qu'elle l'est bien (penser aux paramètres de session, aux fuseaux horaires…), sous peine d'incohérences dans les données.

Cas d'usage :

Les colonnes générées économisent la création de triggers, ou de vues de « présentation ». Elles facilitent la dénormalisation de données calculées dans une même table tout en garantissant l'intégrité.

Un cas d'usage courant est la dénormalisation d'attributs JSON pour les manipuler comme des champs de table classiques :

```
ALTER TABLE personnes
ADD COLUMN  lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED ;
```

L'accès au champ est notablement plus rapide que l'analyse systématique du champ JSON.

Par contre, les colonnes `GENERATED` ne sont **pas** un bon moyen pour créer des champs portant la dernière mise à jour. Certes, PostgreSQL ne vous empêchera pas de déclarer une fonction (abusivement) immutable utilisant `now()` ou une variante. Mais ces informations seront perdues en cas de restauration logique. Dans ce cas, les triggers restent une option plus complexe mais plus propre.

1.5 JIT

1.5.1 JIT : la compilation à la volée

- Compilation *Just In Time* des requêtes
- Utilise le compilateur LLVM
- Vérifier que l'installation est fonctionnelle
- Activé par défaut
  - sauf en v11 ; et absent auparavant

Une des nouveautés les plus visibles et techniquement pointues de la v11 est la « compilation à la volée » (*Just In Time compilation*, ou JIT) de certaines expressions dans les requêtes SQL. Le JIT n'est activé par défaut qu'à partir de la version 12.

Dans certaines requêtes, l'essentiel du temps est passé à décoder des enregistrements (*tuple deforming*), à analyser des clauses `WHERE`, à effectuer des calculs. En conséquence, l'idée du JIT est de transformer tout ou partie de la requête en un programme natif directement exécuté par le processeur.

Cette compilation est une opération lourde qui ne sera effectuée que pour des requêtes qui en valent le coup, donc qui dépassent un certain coût. Au contraire de la parallélisation, ce coût n'est pas pris en compte par le planificateur. La décision d'utiliser le JIT ou pas se fait une fois le plan décidé, si le coût calculé de la requête dépasse un certain seuil.

Le JIT de PostgreSQL s'appuie actuellement sur la chaîne de compilation LLVM, choisie pour sa flexibilité. L'utilisation nécessite un PostgreSQL compilé avec l'option `--with-llvm` et l'installation des bibliothèques de LLVM.

Sur Debian, avec les paquets du PGDG, les dépendances sont en place dès l'installation.

Sur Rocky Linux/Red Hat 8 et 9, l'installation du paquet dédié suffit :

```
# dnf install postgresql14-llvmjit
```

Sur CentOS/Red Hat 7, ce paquet supplémentaire nécessite lui-même des paquets du dépôt EPEL :

```
# yum install epel-release
# yum install postgresql14-llvmjit
```

Les systèmes CentOS/Red Hat 6 ne permettent pas d'utiliser le JIT.

Si PostgreSQL ne trouve pas les bibliothèques nécessaires, il ne renvoie pas d'erreur et continue sans tenter de JIT. Pour tester si le JIT est fonctionnel sur votre machine, il faut le chercher dans un plan quand on force son utilisation ainsi :

```
SET jit=on;
SET jit_above_cost TO 0 ;
EXPLAIN (ANALYZE) SELECT 1;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=4) (… rows=1 loops=1)
 Planning Time: 0.069 ms
 JIT:
   Functions: 1
   Options: Inlining false, Optimization false, Expressions true,
            Deforming true
   Timing:  Generation 0.123 ms, Inlining 0.000 ms, Optimization 0.187 ms,
            Emission 2.778 ms, Total 3.088 ms
 Execution Time: 3.952 ms
```

La documentation officielle est assez accessible : https://doc.postgresql.fr/current/jit.html

1.5.2 JIT : qu'est-ce qui est compilé ?

- *Tuple deforming*
- Évaluation d'expressions :
  - `WHERE`
  - agrégats, `GROUP BY`
- Appels de fonctions (*inlining*)
- Mais pas les jointures

Le JIT ne peut pas encore compiler toute une requête. La version actuelle se concentre sur des goulots d'étranglement classiques :

- le décodage des enregistrements (*tuple deforming*) pour en extraire les champs intéressants ;
- les évaluations d'expressions, notamment dans les clauses `WHERE` pour filtrer les lignes ;
- les agrégats, les `GROUP BY`…

Les jointures ne sont pas (encore ?) concernées par le JIT.

Le code résultant est utilisable plus efficacement avec les processeurs actuels qui utilisent les pipelines et les prédictions de branchement.

Pour les détails, on peut consulter notamment cette conférence très technique au FOSDEM 2018[2] par l'auteur principal du JIT, Andres Freund.

[2] https://archive.fosdem.org/2018/schedule/event/jiting_postgresql_using_llvm/

1.5.3 JIT : algorithme « naïf »

- `jit` (défaut : `on`)
- `jit_above_cost` (défaut : 100 000)
- `jit_inline_above_cost` (défaut : 500 000)
- `jit_optimize_above_cost` (défaut : 500 000)
- À comparer au coût de la requête… I/O comprises
- Seuils arbitraires !

De l'avis même de son auteur, l'algorithme de déclenchement du JIT est « naïf ». Quatre paramètres existent (hors débogage).

`jit = on` (défaut à partir de la v12) active le JIT **si** l'environnement technique évoqué plus haut le permet.

La compilation n'a cependant lieu que pour un coût de requête calculé d'au moins `jit_above_cost` (par défaut 100 000, une valeur élevée). Puis, si le coût atteint `jit_inline_above_cost` (500 000), certaines fonctions utilisées par la requête et supportées par le JIT sont intégrées dans la compilation. Si `jit_optimize_above_cost` (500 000) est atteint, une optimisation du code compilé est également effectuée. Ces deux dernières opérations étant longues, elles ne le sont que pour des coûts assez importants.

Ces seuils sont à comparer avec les coûts des requêtes, qui incluent les entrées-sorties, donc pas seulement le coût CPU. Ces seuils sont un peu arbitraires et nécessiteront sans doute un certain tuning en fonction de vos requêtes et de vos processeurs.

Des contre-performances dues au JIT ont déjà été observées, menant à monter les seuils. Le JIT est trop jeune pour que les développeurs de PostgreSQL eux-mêmes aient des règles d'ajustement des valeurs des différents paramètres. Il est fréquent de le désactiver ou de monter radicalement les seuils de déclenchement.

Un exemple de plan d'exécution sur une grosse table donne :

```
# EXPLAIN (ANALYZE) SELECT sum(x), count(id)
  FROM bigtable WHERE id + 2 > 500000  ;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=3403866.94..3403866.95 rows=1 width=16) (…)
   ->  Gather        (cost=3403866.19..3403866.90 rows=7 width=16)
                (actual time=11778.983..11784.235 rows=8 loops=1)
         Workers Planned: 7
         Workers Launched: 7
         ->  Partial Aggregate  (cost=3402866.19..3402866.20 rows=1 width=16)(…)
               ->  Parallel Seq Scan on bigtable  (…)
                     Filter: ((id + 2) > 500000)
```

```
                    Rows Removed by Filter: 62500
Planning Time: 0.047 ms
JIT:
  Functions: 42
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing:  Generation 5.611 ms, Inlining 422.019 ms, Optimization 229.956 ms,
           Emission 125.768 ms, Total 783.354 ms
Execution Time: 11785.276 ms
```

Le plan d'exécution est complété, à la fin, des informations suivantes :

- le nombre de fonctions concernées ;
- les temps de génération, d'inclusion des fonctions, d'optimisation du code compilé…

Dans l'exemple ci-dessus, on peut constater que ces coûts ne sont pas négligeables par rapport au temps total. Il reste à voir si ce temps perdu est récupéré sur le temps d'exécution de la requête… ce qui en pratique n'a rien d'évident.

Sans JIT, la durée de cette requête était d'environ 17 s. Ici le JIT est rentable.

1.5.4 Quand le JIT est-il utile ?

- Goulot d'étranglement au niveau CPU (pas I/O)
- Requêtes complexes (calculs, agrégats, appels de fonctions…)
- Beaucoup de lignes, filtres
- Assez longues pour « rentabiliser » le JIT
- Analytiques, pas ERP

Vu son coût élevé, le JIT n'a d'intérêt que pour les requêtes utilisant beaucoup le CPU et où il est le facteur limitant.

Ce seront donc surtout des requêtes analytiques agrégeant beaucoup de lignes, comprenant beaucoup de calculs et filtres, et non les petites requêtes d'un ERP.

Il n'y a pas non plus de mise en cache du code compilé.

Si gain il y a, il est relativement modeste en deçà de quelques millions de lignes, et devient de plus en plus important au fur et à mesure que la volumétrie augmente, à condition bien sûr que d'autres limites n'apparaissent pas (bande passante…).

Documentation officielle : https://docs.postgresql.fr/current/jit-decision.html

1.6 RECHERCHE PLEIN TEXTE

…ou FTS

1.6.1 Full Text Search : principe

- Recherche « à la Google » ; fonctions dédiées
- On n'indexe plus une chaîne de caractère mais
  - les mots (« lexèmes ») qui la composent
  - on peut rechercher sur chaque lexème indépendamment
- Les lexèmes sont soumis à des règles spécifiques à chaque langue
  - notamment termes courants
  - permettent une normalisation, des synonymes…

L'indexation FTS est un des cas les plus fréquents d'utilisation non-relationnelle d'une base de données : les utilisateurs ont souvent besoin de pouvoir rechercher une information qu'ils ne connaissent pas parfaitement, d'une façon floue :

- recherche d'un produit/article par rapport à sa description ;
- recherche dans le contenu de livres/documents…

PostgreSQL doit donc permettre de rechercher de façon efficace dans un champ texte. L'avantage de cette solution est d'être intégrée au SGBD. Le moteur de recherche est donc toujours parfaitement à jour avec le contenu de la base, puisqu'il est intégré avec le reste des transactions.

Le principe est de décomposer le texte en « lexèmes » propres à chaque langue. Cela implique donc une certaine forme de normalisation, et permettent aussi de tenir compte de dictionnaires de synonymes. Le dictionnaire inclue aussi les termes courants inutiles à indexer (*stop words*) propres à la langue (le, la, et, the, and, der, daß…).

Décomposition et recherche en plein texte utilisent des fonctions et opérateurs dédiés, ce qui nécessite donc une adaptation du code. Ce qui suit n'est qu'un premier aperçu. La recherche plein texte est un chapitre entier de la documentation officielle[3].

Adrien Nayrat a donné une excellente conférence sur le sujet au PGDay France 2017 à Toulouse[4] (slides[5]).

[3] https://docs.postgresql.fr/current/textsearch.html
[4] https://www.youtube.com/embed/9S5dBqMbw8A
[5] https://2017.pgday.fr/slides/nayrat_Le_Full_Text_Search_dans_PostgreSQL.pdf

1.6.2 Full Text Search : exemple

- Décomposition :

```
SELECT to_tsvector ('french',
                    'Longtemps je me suis couché de bonne heure');

               to_tsvector
-----------------------------------------
 'bon':7 'couch':5 'heur':8 'longtemp':1
```

- Recherche sur 2 mots :

```
SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('Valjean & Cosette');
```

- Recherche sur une phrase : `phrase_totsquery`

`to_tsvector` analyse un texte et le décompose en lexèmes, et non en mots. Les chiffres indiquent ici les positions et ouvrent la possibilité à des scores de proximité. Mais des indications de poids sont possibles.

Autre exemple de décomposition d'une phrase :

```
SHOW default_text_search_config ;

 default_text_search_config
----------------------------
 pg_catalog.french

SELECT to_tsvector (
'La documentation de PostgreSQL est sur https://www.postgresql.org/') ;

                    to_tsvector
---------------------------------------------------
 'document':2 'postgresql':4 'www.postgresql.org':7
```

Les mots courts et le verbe « être » sont repérés comme termes trop courants, la casse est ignorée, même l'URL est décomposée en protocole et hôte. On peut voir en détail comment la FTS a procédé :

```
SELECT description, token, dictionary, lexemes
FROM ts_debug('La documentation de PostgreSQL est sur https://www.postgresql.org/') ;

   description    |       token        | dictionary  |       lexemes
------------------+--------------------+-------------+----------------------
 Word, all ASCII | La                 | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | documentation      | french_stem | {document}
 Space symbols   |                    | ¤           | ¤
```

```
 Word, all ASCII | de                 | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | PostgreSQL         | french_stem | {postgresql}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | est                | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | sur                | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Protocol head   | https://           | ¤           | ¤
 Host            | www.postgresql.org | simple      | {www.postgresql.org}
 Space symbols   | /                  | ¤           | ¤
```

Si l'on se trompe de langue, les termes courants sont mal repérés (et la recherche sera inefficace) :

```
SELECT to_tsvector ('english',
'La documentation de PostgreSQL est sur https://www.postgresql.org/');
```

```
                                  to_tsvector
-------------------------------------------------------------------------------
 'de':3 'document':2 'est':5 'la':1 'postgresql':4 'sur':6 'www.postgresql.org':7
```

Pour construire un critère de recherche, `to_tsquery` est nécessaire :

```
SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('Valjean & Cosette');
```

Les termes à chercher peuvent être combinés par `&`, `|` (ou), `!` (négation), `<->` (mots successifs), `<N>` (séparés par N lexèmes). `@@` est l'opérateur de correspondance. Il y en a d'autres[6].

Il existe une fonction `phraseto_tsquery` pour donner une phrase entière comme critère, laquelle sera décomposée en lexèmes :

```
SELECT livre, contenu FROM textes
WHERE
      livre ILIKE 'Les Misérables Tome V%'
AND ( to_tsvector ('french',contenu)
        @@ phraseto_tsquery('c''est la fautes de Voltaire')
  OR  to_tsvector ('french',contenu)
        @@ phraseto_tsquery('nous sommes tombés à terre')
    );
```

```
                     livre                      |           contenu
------------------------------------------------+---------------------------
…
Les misérables Tome V Jean Valjean, Hugo, Victor | Je suis tombé par terre,
Les misérables Tome V Jean Valjean, Hugo, Victor | C'est la faute à Voltaire,
```

[6] https://docs.postgresql.fr/current/functions-textsearch.html

1.6.3 Full Text Search : dictionnaires

- Configurations liées à la langue
  - basées sur des dictionnaires (parfois fournis)
  - dictionnaires filtrants (`unaccent`)
  - synonymes
- Extensible grâce à des sources extérieures
- Configuration par défaut : `default_text_search_config`

Les lexèmes, les termes courants, la manière de décomposer un terme… sont fortement liés à la langue.

Des configurations toutes prêtes sont fournies par PostgreSQL pour certaines langues :

```
# \dF
        Liste des configurations de la recherche de texte
   Schéma   |    Nom     |              Description
------------+------------+---------------------------------------
 pg_catalog | arabic     | configuration for arabic language
 pg_catalog | danish     | configuration for danish language
 pg_catalog | dutch      | configuration for dutch language
 pg_catalog | english    | configuration for english language
 pg_catalog | finnish    | configuration for finnish language
 pg_catalog | french     | configuration for french language
 pg_catalog | german     | configuration for german language
 pg_catalog | hungarian  | configuration for hungarian language
 pg_catalog | indonesian | configuration for indonesian language
 pg_catalog | irish      | configuration for irish language
 pg_catalog | italian    | configuration for italian language
 pg_catalog | lithuanian | configuration for lithuanian language
 pg_catalog | nepali     | configuration for nepali language
 pg_catalog | norwegian  | configuration for norwegian language
 pg_catalog | portuguese | configuration for portuguese language
 pg_catalog | romanian   | configuration for romanian language
 pg_catalog | russian    | configuration for russian language
 pg_catalog | simple     | simple configuration
 pg_catalog | spanish    | configuration for spanish language
 pg_catalog | swedish    | configuration for swedish language
 pg_catalog | tamil      | configuration for tamil language
 pg_catalog | turkish    | configuration for turkish language
```

La recherche plein texte est donc directement utilisable pour le français ou l'anglais et beaucoup d'autres langues européennes. La configuration par défaut dépend du paramètre `default_text_search_config`, même s'il est conseillé de toujours passer explicitement la configuration aux fonctions. Ce paramètre peut être modifié globalement, par session ou par un `ALTER DATABASE SET`.

En demandant le détail de la configuration `french`, on peut voir qu'elle se base sur des « dic-

tionnaires » pour chaque type d'élément qui peut être rencontré : mots, phrases mais aussi URL, entiers…

```
# \dF+ french
Configuration « pg_catalog.french » de la recherche de texte
Analyseur : « pg_catalog.default »
      Jeton      | Dictionnaires
-----------------+---------------
 asciihword      | french_stem
 asciiword       | french_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | french_stem
 hword_asciipart | french_stem
 hword_numpart   | simple
 hword_part      | french_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | french_stem
```

On peut lister ces dictionnaires :

```
# \dFd
            Liste des dictionnaires de la recherche de texte
   Schéma   |       Nom       |               Description
------------+-----------------+--------------------------------------------
…
 pg_catalog | english_stem    | snowball stemmer for english language
…
 pg_catalog | french_stem     | snowball stemmer for french language
…
 pg_catalog | simple          | simple dictionary: just lower case
                              | and check for stopword
…
```

Ces dictionnaires sont de type « Snowball[7] », incluant notamment des algorithmes différents pour chaque langue. Le dictionnaire `simple` n'est pas lié à une langue et correspond à une simple décomposition après passage en minuscule et recherche de termes courants anglais : c'est suffisant pour des éléments comme les URL.

D'autres dictionnaires peuvent être combinés aux existants pour créer une nouvelle configuration. Le principe est que les dictionnaires reconnaissent certains éléments, et transmettent aux suivants ce qu'ils n'ont pas reconnu. Les dictionnaires précédents, de type Snowball, reconnaissent tout et doivent donc être placés en fin de liste.

[7] https://snowballstem.org/

Par exemple, la contrib `unaccent` permet de faire une configuration négligeant les accents[8]. La contrib `dict_int` fournit un dictionnaire qui réduit la précision des nombres[9] pour réduire la taille de l'index. La contrib `dict_xsyn` permet de créer un dictionnaire pour gérer une liste de synonymes[10]. Mais les dictionnaires de synonymes peuvent être gérés manuellement[11]. Les fichiers adéquats sont déjà présents ou à ajouter dans `$SHAREDIR/tsearch_data/` (par exemple `/usr/pgsql-14/share/tsearch_data` sur Red Hat/CentOS ou `/usr/share/postgresql/14/tsearch_data` sur Debian).

Par exemple, en utilisant le fichier d'exemple `$SHAREDIR/tsearch_data/synonym_sample.syn`, dont le contenu est :

```
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

on peut définir un dictionnaire de synonymes, créer une nouvelle configuration reprenant `french`, et y insérer le nouveau dictionnaire en premier élément :

```
CREATE TEXT SEARCH DICTIONARY messynonymes (template=synonym,
 ↪  synonyms='synonym_sample');

CREATE TEXT SEARCH CONFIGURATION french2 (copy=french);

ALTER TEXT SEARCH CONFIGURATION french2
ALTER MAPPING FOR asciiword,hword,asciihword,word
WITH messynonymes, french_stem ;
```

À l'usage :

```
SELECT to_tsvector ('french2', 'PostgreSQL s''abrège en pgsql ou Postgres') ;

         to_tsvector
-------------------------
 'abreg':3 'pgsql':1,5,7
```

Les trois versions de « PostgreSQL » ont été reconnues.

Pour une analyse plus fine, on peut ajouter d'autres dictionnaires linguistiques depuis des sources extérieures (Ispell, OpenOffice…). Ce n'est pas intégré par défaut à PostgreSQL mais la procédure est dans la documentation[12].

Des « thesaurus » peuvent être même être créés pour remplacer des expressions par des synonymes (et identifier par exemple « le meilleur SGBD » et « PostgreSQL »).

[8] https://docs.postgresql.fr/current/unaccent.html
[9] https://docs.postgresql.fr/current/dict-int.html
[10] https://docs.postgresql.fr/current/dict-xsyn.html
[11] https://docs.postgresql.fr/current/textsearch-dictionaries.html#textsearch-synonym-dictionary
[12] https://docs.postgresql.fr/current/textsearch-dictionaries.html

1.6.4 Full Text Search : stockage & indexation

- Stocker `to_tsvector (champtexte)`
  - colonne mise à jour par trigger
  - ou colonne générée (v12)
- Indexation GIN ou GiST

Principe :

Sans indexation, une recherche FTS fonctionne, mais parcourra entièrement la table. L'indexation est possible, avec GIN ou GiST. On peut stocker le vecteur résultat de `to_tsvector` dans une autre colonne de la table, et c'est elle qui sera indexée. Jusque PostgreSQL 11, il est nécessaire de le faire manuellement, ou d'écrire un trigger pour cela. À partir de PostgreSQL 12, on peut utiliser une colonne générée (il est nécessaire de préciser la configuration FTS), qui sera stockée sur le disque :

```
-- Attention, ceci réécrit la table
ALTER TABLE textes
ADD COLUMN vecteur tsvector
GENERATED ALWAYS AS (to_tsvector ('french', contenu)) STORED ;
```

Les critères de recherche porteront sur la colonne `vecteur` :

```
SELECT * FROM textes
WHERE vecteur @@ to_tsquery ('french','Roméo <2> Juliette');
```

Cette colonne sera ensuite indexée par GIN pour avoir des temps d'accès corrects :

```
CREATE INDEX on textes USING gin (vecteur) ;
```

Alternative : index fonctionnel

Plus simplement, il peut suffire de créer juste un index fonctionnel sur `to_tsvector ('french', contenu)`. On épargne ainsi l'espace du champ calculé dans la table.

Par contre, l'index devra porter sur le critère de recherche exact, sinon il ne sera pas utilisable. Cela n'est donc pertinent que si la majorité des recherches porte sur un nombre très restreint de critères, et il faudra un index par critère.

```
CREATE INDEX idx_fts ON public.textes
USING gin (to_tsvector('french'::regconfig, contenu))

SELECT * FROM textes
WHERE to_tsvector ('french', contenu) @@ to_tsquery ('french','Roméo <2> Juliette');
```

Exemple complet de mise en place de FTS :

- Création d'une configuration de dictionnaire dédiée avec dictionnaire français, sans accent, dans une table de dépêches :

```
CREATE TEXT SEARCH CONFIGURATION depeches (COPY= french);

CREATE EXTENSION unaccent ;

ALTER TEXT SEARCH CONFIGURATION depeches ALTER MAPPING FOR
hword, hword_part, word WITH unaccent,french_stem;
```

- Ajout d'une colonne vectorisée à la table `depeche`, avec des poids différents pour le titre et le texte, ici gérée manuellement avec un trigger.

```
CREATE TABLE depeche (id int, titre text, texte text) ;

ALTER TABLE depeche ADD vect_depeche tsvector;

UPDATE depeche
SET vect_depeche =
(setweight(to_tsvector('depeches',coalesce(titre,'')), 'A') ||
setweight(to_tsvector('depeches',coalesce(texte,'')), 'C'));

CREATE FUNCTION to_vectdepeche( )
 RETURNS trigger
 LANGUAGE plpgsql
 -- common options:  IMMUTABLE  STABLE  STRICT  SECURITY DEFINER
AS $function$
BEGIN
  NEW.vect_depeche :=
    setweight(to_tsvector('depeches',coalesce(NEW.titre,'')), 'A') ||
    setweight(to_tsvector('depeches',coalesce(NEW.texte,'')), 'C');
  return NEW;
END
$function$;

CREATE TRIGGER trg_depeche before INSERT OR update ON depeche
FOR EACH ROW execute procedure to_vectdepeche();
```

- Création de l'index associé au vecteur :

```
 CREATE INDEX idx_gin_texte ON depeche USING gin(vect_depeche);
```

- Collecte des statistiques sur la table :

```
ANALYZE depeche ;
```

- Utilisation basique :

```
SELECT titre,texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle');
SELECT titre,texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle & médecin');
```

- Tri par pertinenence :

```
SELECT titre,texte
FROM depeche
WHERE vect_depeche @@ to_tsquery('depeches','varicelle & médecin')
ORDER BY ts_rank_cd(vect_depeche, to_tsquery('depeches','varicelle & médecin'));
```

- Cette requête peut s'écrire aussi ainsi :

```
SELECT titre,ts_rank_cd(vect_depeche,query) AS rank
FROM depeche, to_tsquery('depeches','varicelle & médecin') query
WHERE query@@vect_depeche
ORDER BY rank DESC ;
```

1.6.5 Full Text Search sur du JSON

- Vectorisation possible des JSON

```
SELECT info FROM commandes c
WHERE to_tsvector ('french', c.info) @@ to_tsquery('papier') ;

                              info
-------------------------------------------------------------------
 {"items": {"qté": 5, "produit": "Rame papier normal A4"},
            "client": "Benoît Delaporte"}
 {"items": {"qté": 5, "produit": "Pochette Papier dessin A3"},
            "client": "Lucie Dumoulin"}
```

Une recherche FTS est directement possible sur des champs JSON. Voici un exemple :

```
CREATE TABLE commandes (info jsonb);

INSERT INTO commandes (info)
VALUES
   (
      '{ "client": "Jean Dupont",
         "articles": {"produit": "Enveloppes A4","qté": 24}}'
   ),
   (
      '{ "client": "Jeanne Durand",
         "articles": {"produit": "Imprimante","qté": 1}}'
   ),
   (
      '{ "client": "Benoît Delaporte",
         "items": {"produit": "Rame papier normal A4","qté": 5}}'
   ),
   (
      '{ "client": "Lucie Dumoulin",
         "items": {"produit": "Pochette Papier dessin A3","qté": 5}}'
   );
```

La décomposition par FTS donne :

```
SELECT to_tsvector('french', info) FROM commandes ;
```

```
                  to_tsvector
-----------------------------------------------
 'a4':5 'dupont':2 'envelopp':4 'jean':1
 'durand':2 'imprim':4 'jeann':1
 'a4':4 'benoît':6 'delaport':7 'normal':3 'papi':2 'ram':1
 'a3':4 'dessin':3 'dumoulin':7 'luc':6 'papi':2 'pochet':1
```

Une recherche sur « papier » donne :

```
SELECT info FROM commandes c
WHERE to_tsvector ('french', c.info) @@ to_tsquery('papier') ;
```

```
                                          info
-------------------------------------------------------------------------------
 {"items": {"qté": 5, "produit": "Rame papier normal A4"}, "client": "Benoît
 ↪  Delaporte"}
 {"items": {"qté": 5, "produit": "Pochette Papier dessin A3"}, "client": "Lucie
 ↪  Dumoulin"}
```

Plus d'information chez Depesz : Full Text Search support for json and jsonb[13].

[13] https://www.depesz.com/2017/04/04/waiting-for-postgresql-10-full-text-search-support-for-json-and-jsonb/

1.7 QUIZ

https://dali.bo/t1_quiz

1.8 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/t1_solutions.

1.8.1 Tables non journalisées

But : Tester les tables non journalisées

Afficher le nom du journal de transaction courant.

Créer une base **pgbench** vierge, de taille 80 (environ 1,2 Go). Les tables doivent être en mode **unlogged**.

Afficher la liste des objets **unlogged** dans la base **pgbench**.

Afficher le nom du journal de transaction courant. Que s'est-il passé ?

Passer l'ensemble des tables de la base **pgbench** en mode **logged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé ?

Repasser toutes les tables de la base **pgbench** en mode **unlogged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé ?

Réinitialiser la base **pgbench** toujours avec une taille 80 mais avec les tables en mode **logged**. Que constate-t-on ?

Réinitialiser la base **pgbench** mais avec une taille de 10. Les tables doivent être en mode **unlogged**.

Compter le nombre de lignes dans la table `pgbench_accounts`.

Simuler un crash de l'instance PostgreSQL.

Redémarrer l'instance PostgreSQL.

Compter le nombre de lignes dans la table `pgbench_accounts`. Que constate-t-on ?

1.8.2 Indexation Full Text

But : Tester l'indexation *Full Text*

Vous aurez besoin de la base **textes**. La base **gutenberg** 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
```

Ce TP utilise la version complète de la base **textes** basée sur le projet Gutenberg. Un index GIN va permettre d'utiliser la *Full Text Search* sur la table **textes**.

Créer un index GIN sur le vecteur du champ `contenu` (fonction `to_tsvector`).

Quelle est la taille de cet index ?

Quelle performance pour trouver « Fantine » (personnage des *Misérables* de Victor Hugo) dans la table ? Le résultat contient-il bien « Fantine » ?

Trouver les lignes qui contiennent à la fois les mots « affaire » et « couteau » et voir le plan.

1.9 TRAVAUX PRATIQUES (SOLUTIONS)

1.9.1 Tables non journalisées

> Afficher le nom du journal de transaction courant.

```
SELECT pg_walfile_name(pg_current_wal_lsn()) ;
```

```
     pg_walfile_name
--------------------------
 000000010000000100000024
```

> Créer une base **pgbench** vierge, de taille 80 (environ 1,2 Go). Les tables doivent être en mode **unlogged**.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 80 --unlogged-tables pgbench
```

```
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
8000000 of 8000000 tuples (100%) done (elapsed 4.93 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 8.84 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 5.02 s,
vacuum 1.79 s, primary keys 2.02 s).
```

> Afficher la liste des objets **unlogged** dans la base **pgbench**.

```
SELECT relname FROM pg_class
WHERE relpersistence = 'u' ;
```

```
        relname
-----------------------
 pgbench_accounts
 pgbench_branches
 pgbench_history
 pgbench_tellers
 pgbench_branches_pkey
 pgbench_tellers_pkey
 pgbench_accounts_pkey
```

Les 3 objets avec le suffixe **pkey** correspondent aux clés primaires des tables créées par **pgbench**. Comme elles dépendent des tables, elles sont également en mode **unlogged**.

> Afficher le nom du journal de transaction courant. Que s'est-il passé ?

```
SELECT pg_walfile_name(pg_current_wal_lsn()) ;
```

```
     pg_walfile_name
--------------------------
 000000010000000100000024
```

Comme l'initialisation de **pgbench** a été réalisée en mode **unlogged**, aucune information concernant les tables et les données qu'elles contiennent n'a été inscrite dans les journaux de transaction. Donc le journal de transaction est toujours le même.

> Passer l'ensemble des tables de la base **pgbench** en mode **logged**.

```
ALTER TABLE pgbench_accounts SET LOGGED;
ALTER TABLE pgbench_branches SET LOGGED;
ALTER TABLE pgbench_history SET LOGGED;
ALTER TABLE pgbench_tellers SET LOGGED;
```

> Afficher le nom du journal de transaction courant. Que s'est-il passé ?

```
SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
     pg_walfile_name
--------------------------
 000000010000000100000077
```

Comme toutes les tables de la base **pgbench** ont été passées en mode **logged**, une réécriture de celles-ci a eu lieu (comme pour un `VACUUM FULL`). Cette réécriture additionnée au mode **logged** a entraîné une forte écriture dans les journaux de transaction. Dans notre cas, 83 journaux de transaction ont été consommés, soit approximativement 1,3 Go d'utilisé sur disque.

Il faut donc faire particulièrement attention à la quantité de journaux de transaction qui peut être générée lors du passage d'une table du mode **unlogged** à **logged**.

> Repasser toutes les tables de la base **pgbench** en mode **unlogged**.

```
ALTER TABLE pgbench_accounts SET UNLOGGED;
ALTER TABLE pgbench_branches SET UNLOGGED;
ALTER TABLE pgbench_history SET UNLOGGED;
ALTER TABLE pgbench_tellers SET UNLOGGED;
```

> Afficher le nom du journal de transaction courant. Que s'est-il passé ?

```
SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
     pg_walfile_name
--------------------------
 000000010000000100000077
```

Le processus est le même que précedemment, mais, lors de la réécriture des tables, aucune information n'est stockée dans les journaux de transaction.

> Réinitialiser la base **pgbench** toujours avec une taille 80 mais avec les tables en mode **logged**. Que constate-t-on ?

```
$ /usr/pgsql-14/bin/pgbench -i -s 80 -d pgbench
```

```
dropping old tables...
creating tables...
generating data (client-side)...
8000000 of 8000000 tuples (100%) done (elapsed 9.96 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 16.60 s (drop tables 0.11 s, create tables 0.00 s, client-side generate 10.12
↪  s,
vacuum 2.87 s, primary keys 3.49 s).
```

On constate que le temps mis par **pgbench** pour initialiser sa base est beaucoup plus long en mode **logged** que **unlogged**. On passe de 8,84 secondes en **unlogged** à 16,60 secondes en mode **logged**. Cette augmentation du temps de traitement est due à l'écriture dans les journaux de transaction.

> Réinitialiser la base **pgbench** mais avec une taille de 10. Les tables doivent être en mode **unlogged**.

```
$ /usr/pgsql-14/bin/pgbench -i -s 10 -d pgbench --unlogged-tables
```

```
dropping old tables...
creating tables...
generating data (client-side)...
1000000 of 1000000 tuples (100%) done (elapsed 0.60 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 1.24 s (drop tables 0.02 s, create tables 0.02 s, client-side generate 0.62 s,
vacuum 0.27 s, primary keys 0.31 s).
```

> Compter le nombre de lignes dans la table `pgbench_accounts`.

```
SELECT count(*) FROM pgbench_accounts ;
```

```
  count
---------
 1000000
```

> Simuler un crash de l'instance PostgreSQL.

```
$ ps -ef | grep postmaster
```

```
postgres   697  1  0 14:32 ?   00:00:00 /usr/pgsql-14/bin/postmaster -D ...
```

```
$ kill -9 697
```

Ne faites jamais un `kill -9` sur un processus de l'instance PostgreSQL en production, bien sûr !

> Redémarrer l'instance PostgreSQL.

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data start
```

> Compter le nombre de lignes dans la table `pgbench_accounts`. Que constate-t-on ?

```
SELECT count(*) FROM pgbench_accounts ;

 count
-------
     0
```

Lors d'un crash, PostgreSQL remet tous les objets **unlogged** à zéro.

1.9.2 Indexation Full Text

> Créer un index GIN sur le vecteur du champ `contenu` (fonction `to_tsvector`).

```
textes=# CREATE INDEX idx_fts ON textes
USING gin (to_tsvector('french',contenu));
CREATE INDEX
```

> Quelle est la taille de cet index ?

La table « pèse » 3 Go (même si on pourrait la stocker de manière beaucoup plus efficace). L'index GIN est lui-même assez lourd dans la configuration par défaut :

```
textes=# SELECT pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 593 MB
(1 ligne)
```

> Quelle performance pour trouver « Fantine » (personnage des *Misérables* de Victor Hugo) dans la table ? Le résultat contient-il bien « Fantine » ?

```
textes=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');

                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=107.94..36936.16 rows=9799 width=123)
                          (actual time=0.423..1.149 rows=326 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu)
                 @@ '''fantin'''::tsquery)
   Heap Blocks: exact=155
   Buffers: shared hit=159
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..105.49 rows=9799 width=0)
                             (actual time=0.210..0.211 rows=326 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu)
                      @@ '''fantin'''::tsquery)
         Buffers: shared hit=4
 Planning Time: 1.248 ms
 Execution Time: 1.298 ms
```

On constate donc que le *Full Text Search* est très efficace du moins pour le *Full Text Search* + GIN : trouver 1 mot parmi plus de 100 millions avec 300 enregistrements correspondants dure 1,5 ms (cache chaud).

Si l'on compare avec une recherche par trigramme (extension `pg_trgm` et index GIN), c'est bien meilleur. À l'inverse, les trigrammes permettent des recherches floues (orthographe approximative), des recherches sur autre chose que des mots, et ne nécessitent pas de modification de code.

Par contre, la recherche n'est pas exacte, « Fantin » est fréquemment trouvé. En fait, le plan montre que c'est le vrai critère retourné par `to_tsquery('french','fantine')` et transformé en `'fantin'::tsquery`. Si l'on tient à ce critère précis il faudra ajouter une clause plus classique `contenu LIKE '%Fantine%'` pour filtrer le résultat après que le FTS ait « dégrossi » la recherche.

> Trouver les lignes qui contiennent à la fois les mots « affaire » et « couteau » et voir le plan.

10 lignes sont ramenées en quelques millisecondes :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','affaire  & couteau')
;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=36.22..154.87 rows=28 width=123)
                       (actual time=6.642..6.672 rows=10 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu)
                 @@ '''affair'' & ''couteau'''::tsquery)
   Heap Blocks: exact=10
   Buffers: shared hit=53
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..36.21 rows=28 width=0)
                            (actual time=6.624..6.624 rows=10 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu)
                     @@ '''affair'' & ''couteau'''::tsquery)
         Buffers: shared hit=43
 Planning Time: 0.519 ms
 Execution Time: 6.761 ms
```

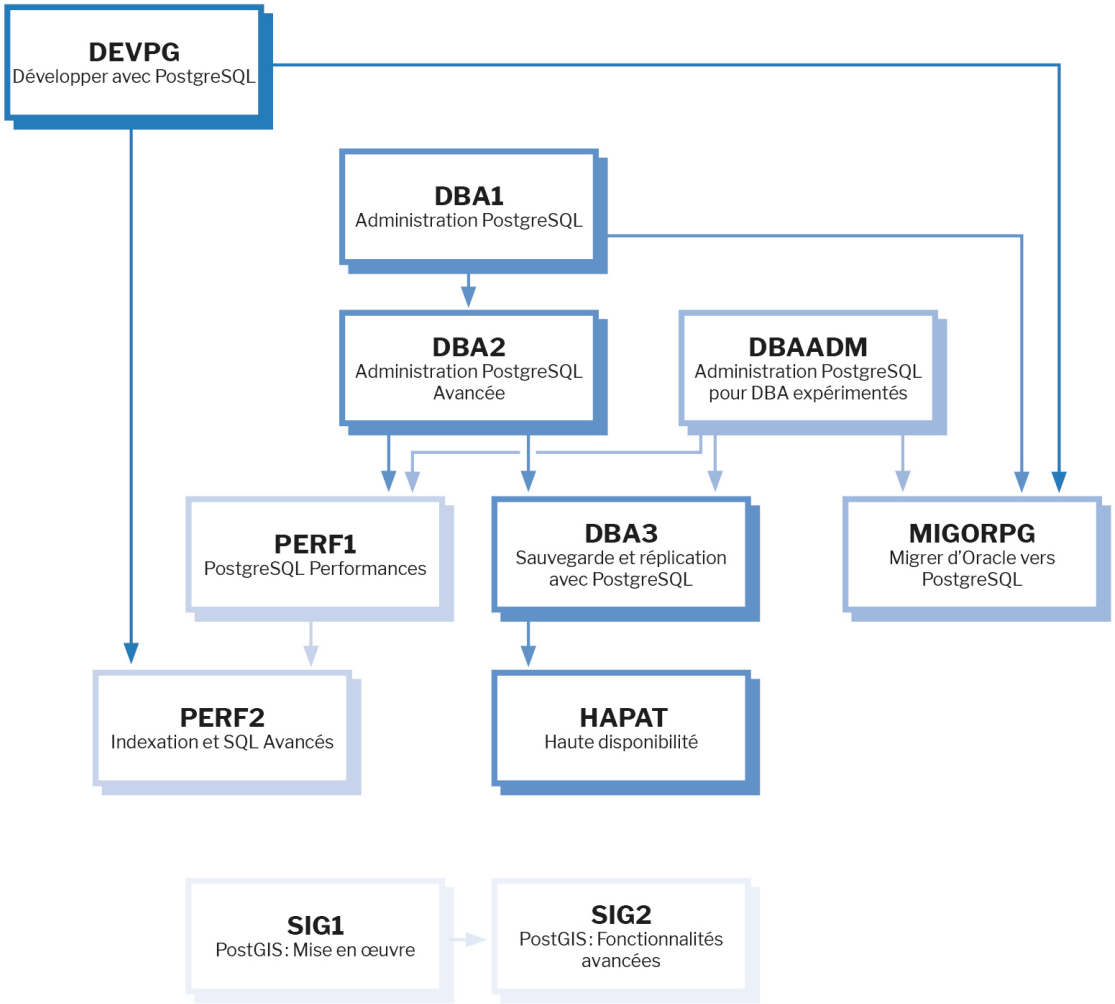
Noter que les pluriels « couteaux » et « affaires » figurent parmi les résultats puisque la recherche porte sur les lexèmes `'affair'' & ''couteau'`.

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.