Module X2

Extensions PostgreSQL pour la performance

24.04

Table des matières

Sur ce document

Formation	Module X2
Titre	Extensions PostgreSQL pour la performance
Révision	24.04
PDF	https://dali.bo/x2_pdf
EPUB	https://dali.bo/x2_epub
HTML	https://dali.bo/x2_html
Slides	https://dali.bo/x2_slides
TP	https://dali.bo/x2_tp
TP (solutions)	https://dali.bo/x2_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

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/ Extensions PostgreSQL pour la performance

1.1 PRÉAMBULE

Ce module présente des extensions plus spécifiquement destinées à améliorer les performances.

1.2 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`.

1.3 PG_STAT_STATEMENTS

Capture en temps réel des requêtes

- Normalisation
- Indicateurs :
  - nombre d'exécutions,
  - nombre d'enregistrements retournés
  - temps cumulé d'exécution et d'optimisation
  - lectures/écritures en cache, demandées au système, tris
  - temps de lecture/écriture (`track_io_timing`)
  - écritures dans les journaux de transactions (v13)
  - temps de planning (désactivé par défaut, v13)
  - utilisation du JIT (v15)

Cette extension est fournie avec PostgreSQL et est parmi les plus populaires et les plus utiles.

Une fois installé, `pg_stat_statements` capture, à chaque exécution de requête, tous les compteurs ci-dessus et d'autres associés à cette requête (champ `query`), ci-dessous avec PostgreSQL 16 :

```
postgres=# \d pg_stat_statements
                    Vue « public.pg_stat_statements »
        Colonne         |       Type        | Collationnement | NULL-able | …
------------------------+-------------------+-----------------+-----------+--
 userid                 | oid               |                 |           |
 dbid                   | oid               |                 |           |
 toplevel               | boolean           |                 |           |
 queryid                | bigint            |                 |           |
 query                  | text              |                 |           |
 plans                  | bigint            |                 |           |
 total_plan_time        | double precision  |                 |           |
 min_plan_time          | double precision  |                 |           |
 max_plan_time          | double precision  |                 |           |
 mean_plan_time         | double precision  |                 |           |
 stddev_plan_time       | double precision  |                 |           |
 calls                  | bigint            |                 |           |
 total_exec_time        | double precision  |                 |           |
 min_exec_time          | double precision  |                 |           |
 max_exec_time          | double precision  |                 |           |
 mean_exec_time         | double precision  |                 |           |
 stddev_exec_time       | double precision  |                 |           |
 rows                   | bigint            |                 |           |
 shared_blks_hit        | bigint            |                 |           |
 shared_blks_read       | bigint            |                 |           |
 shared_blks_dirtied    | bigint            |                 |           |
 shared_blks_written    | bigint            |                 |           |
 local_blks_hit         | bigint            |                 |           |
```

```
local_blks_read         | bigint           |           |    |
local_blks_dirtied      | bigint           |           |    |
local_blks_written      | bigint           |           |    |
temp_blks_read          | bigint           |           |    |
temp_blks_written       | bigint           |           |    |
blk_read_time           | double precision |           |    |
blk_write_time          | double precision |           |    |
temp_blk_read_time      | double precision |           |    |
temp_blk_write_time     | double precision |           |    |
wal_records             | bigint           |           |    |
wal_fpi                 | bigint           |           |    |
wal_bytes               | numeric          |           |    |
jit_functions           | bigint           |           |    |
jit_generation_time     | double precision |           |    |
jit_inlining_count      | bigint           |           |    |
jit_inlining_time       | double precision |           |    |
jit_optimization_count  | bigint           |           |    |
jit_optimization_time   | double precision |           |    |
jit_emission_count      | bigint           |           |    |
jit_emission_time       | double precision |           |    |
```

Quelques champs peuvent manquer ou porter un autre nom dans les versions précédentes.

Les requêtes d'une même base et d'un même utilisateur sont normalisées (reconnues comme identiques même avec des paramètres différents).

Les champs sont détaillés dans https://dali.bo/h2_html#pg_stat_statements.

1.3.1 pg_stat_statements : mise en place

```
shared_preload_libraries = 'pg_stat_statements'

CREATE EXTENSION pg_stat_statements ; -- dans 1 ou plusieurs bases
SELECT * FROM pg_stat_statements ;
```

- Vue en mémoire partagée (volumétrie contrôlée)
- Pas d'échantillonnage, seulement des compteurs cumulés
  - `pg_stat_statements_reset()` ou PoWA

1.3.1.1 Installation et réinitialisation

Ce module nécessite un espace en mémoire partagée. Pour l'installer, il faut donc renseigner le paramètre suivant avant de redémarrer l'instance :

```
shared_preload_libraries = 'pg_stat_statements'
```

Il faut installer l'extension dans au moins une base (dont une à laquelle les développeurs auront aussi accès, car l'information les concerne au premier chef) :

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements ;
```

La vue `pg_stat_statements` retourne un instantané des compteurs au moment de l'interrogation depuis l'installation, depuis le dernier arrêt brutal, ou depuis le dernier appel à la fonction `pg_stat_statements_reset()`. Cette dernière fonction permet de réinitialiser les compteurs pour une base, un utilisateur, une requête, ou tout.

Deux méthodes d'utilisation sont donc possibles :

- effectuer un *reset* au début d'une période, puis interroger la vue `pg_stat_statements` à la fin de cette période ;
- capturer à intervalles réguliers le contenu de `pg_stat_statements` et visualiser les changements dans les compteurs : le projet PoWA[1] a été développé à cet effet.

La requête étant déjà analysée, cette opération supplémentaire n'ajoute qu'un faible surcoût (de l'ordre de 5 % sur une requête extrêmement courte), fixe, pour chaque requête.

Les données de l'extension sont stockées dans le PGDATA, sous `pg_stat_tmp` (même pour les versions récentes de PostgreSQL qui ne l'utilisent plus pour le `stats collector`), et un arrêt brutal peut mener à la perte du contenu.

1.3.1.2 Paramétrage

`pg_stat_statements` possède quelques paramètres[2].

Dès lors que l'extension est chargée en mémoire, la capture des compteurs est enclenchée, sauf si le paramètre `pg_stat_statements.track` est positionné à `none`. Celui-ci permet donc d'activer cette capture à la demande, sans qu'il soit nécessaire de redémarrer l'instance, ce qui peut s'avérer utile pour une instance avec beaucoup de requêtes très courtes (de type OLTP), et dont la rapidité est un élément critique : pour une telle instance, le surcoût lié à `pg_stat_statements` peut être jugé trop important pour que cette capture soit activée en permanence.

Sur un serveur chargé, il est déconseillé de réduire `pg_stat_statements.max` (nombre de requêtes différentes suivies, à 5000 par défaut), car le coût d'une désallocation n'est pas négligeable[3].

[1] https://powa.readthedocs.io/en/latest/
[2] https://docs.postgresql.fr/current/pgstatstatements.html#id-1.11.7.40.9
[3] https://yhuelf.github.io/2021/09/30/pg_stat_statements_bottleneck.html

1.3.2 pg_stat_statements : exemple 1

Requêtes les plus longues en temps cumulé :

```
SELECT r.rolname, d.datname, s.calls, s.total_exec_time,
       s.total_exec_time / s.calls AS avg_time, s.query
  FROM pg_stat_statements s
  JOIN pg_roles r     ON (s.userid=r.oid)
  JOIN pg_database d  ON (s.dbid = d.oid)
 ORDER BY s.total_exec_time DESC
 LIMIT 10 ;
```

La requête ci-dessus affiche les dix requêtes les plus longues en cumulé (même avec des paramètres différents), le nombre d'appels, le temps total, le temps moyen par appel. Les temps sont en millisecondes.

NB : pour une instance en version 12 ou antérieure, utiliser le champ `total_time`, qui inclut aussi le temps de planification.

1.3.3 pg_stat_statements : exemple 2

Requêtes les plus fréquemment appelées :

```
SELECT r.rolname, d.datname, s.calls, s.total_exec_time,
       s.total_exec_time / s.calls  AS avg_time, s.query
  FROM pg_stat_statements s
  JOIN pg_roles r     ON (s.userid=r.oid)
  JOIN pg_database d  ON (s.dbid = d.oid)
 ORDER BY s.calls DESC
 LIMIT 10;
```

Cette requête affiche les dix requêtes les plus fréquentes en nombre d'appels, et le temps moyen. Exemple de sortie, avec un peu de formatage :

```
\pset format wrapped
\pset columns 83

 SELECT r.rolname, d.datname,
       to_char (s.calls,'999G999FM') AS calls,
       s.total_exec_time * interval '1ms' AS total_exec_time,
       s.total_exec_time/s.calls * interval '1ms' AS avg_time,
       s.query
  FROM pg_stat_statements s
  JOIN pg_roles r     ON (s.userid=r.oid)
  JOIN pg_database d  ON (s.dbid = d.oid)
```

```
 ORDER BY s.calls DESC
 LIMIT 10  \gx

-[ RECORD 1 ]---+-----------------------------------------------------------------
rolname         | postgres
datname         | postgres
calls           | 329 021
total_exec_time | 00:00:01.617168
avg_time        | 00:00:00.000005
query           | SELECT pg_postmaster_start_time()
-[ RECORD 2 ]---+-----------------------------------------------------------------
rolname         | postgres
datname         | postgres
calls           | 316 192
total_exec_time | 24:19:01.780477
avg_time        | 00:00:00.276863
query           | SELECT                                                         +
                |             count(datid) as databases,                         +
                |             pg_size_pretty(sum(pg_database_size(               +
                |                 pg_database.datname))::bigint) as total_size,  +
                |             to_char(now(),$1) as time,                         +
                |             sum(xact_commit)::BIGINT as total_commit,          +
                |             sum(xact_rollback)::BIGINT as total_rollback       +
                |          FROM pg_database                                      +
                |          JOIN pg_stat_database ON (pg_database.oid = pg_stat_data.
                |.base.datid)                                                    +
                |          WHERE datistemplate = $2
-[ RECORD 3 ]---+-----------------------------------------------------------------
rolname         | postgres
datname         | postgres
calls           | 316 192
total_exec_time | 00:01:22.127931
avg_time        | 00:00:00.00026
query           | SELECT CASE sum(blks_hit+blks_read)                            +
                |           WHEN $1 THEN $2                                      +
                |           ELSE trunc(sum(blks_hit)/sum(blks_hit+blks_read)*$3)::.
                |.float                                                          +
                |          END AS hitratio                                       +
                |          FROM pg_stat_database
-[ RECORD 4 ]---+-----------------------------------------------------------------
rolname         | postgres
datname         | postgres
calls           | 316 192
total_exec_time | 00:00:02.82872
avg_time        | 00:00:00.000009
query           | SELECT buffers_alloc FROM pg_stat_bgwriter
-[ RECORD 5 ]---+-----------------------------------------------------------------
rolname         | postgres
datname         | postgres
calls           | 316 192
total_exec_time | 00:18:08.125136
avg_time        | 00:00:00.003441
query           | SELECT COUNT(*) AS nb FROM pg_stat_activity WHERE state != $1
-[ RECORD 6 ]---+-----------------------------------------------------------------
rolname         | postgres
datname         | pgbench_300_hdd
```

```
calls           | 79 534
total_exec_time | 00:03:44.82423
avg_time        | 00:00:00.002827
query           | select wait_event, wait_event_type, query from pg_stat_activity .
                |.where state =$1 and pid = $2
-[ RECORD 7 ]---+-----------------------------------------------------------------
rolname         | temboard_agent
datname         | postgres
calls           | 75 028
total_exec_time | 00:00:00.368735
avg_time        | 00:00:00.000005
query           | SELECT pg_postmaster_start_time()
-[ RECORD 8 ]---+-----------------------------------------------------------------
rolname         | temboard_agent
datname         | postgres
calls           | 72 091
total_exec_time | 00:04:02.992142
avg_time        | 00:00:00.003371
query           | SELECT COUNT(*) AS nb FROM pg_stat_activity WHERE state != $1
-[ RECORD 9 ]---+-----------------------------------------------------------------
rolname         | temboard_agent
datname         | postgres
calls           | 72 091
total_exec_time | 05:47:55.416569
avg_time        | 00:00:00.28957
query           | SELECT                                                         +
                |             count(datid) as databases,                         +
                |             pg_size_pretty(sum(pg_database_size(               +
                |                 pg_database.datname))::bigint) as total_size,  +
                |             to_char(now(),$1) as time,                         +
                |             sum(xact_commit)::BIGINT as total_commit,          +
                |             sum(xact_rollback)::BIGINT as total_rollback       +
                |         FROM pg_database                                       +
                |         JOIN pg_stat_database ON (pg_database.oid = pg_stat_data.
                |.base.datid)                                                    +
                |         WHERE datistemplate = $2
-[ RECORD 10 ]--+-----------------------------------------------------------------
rolname         | temboard_agent
datname         | postgres
calls           | 72 091
total_exec_time | 00:00:17.817369
avg_time        | 00:00:00.000247
query           | SELECT CASE sum(blks_hit+blks_read)                            +
                |           WHEN $1 THEN $2                                      +
                |           ELSE trunc(sum(blks_hit)/sum(blks_hit+blks_read)*$3)::.
                |.float                                                          +
                |         END AS hitratio                                        +
                |         FROM pg_stat_database
```

On voit qu'il y a beaucoup de requêtes de supervision, ce qui est logique. Il est donc conseillé de dédier un utilisateur à la supervision pour pouvoir filtrer aisément.

1.3.4 pg_stat_statements : exemple 3

Requêtes les plus consommatrices et *hit ratio* :

```
SELECT calls, total_exec_time, rows,
       100.0*shared_blks_hit
       /nullif(shared_blks_hit+shared_blks_read, 0) AS "hit %",
       query
FROM  pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 5 ;
```

Cette requête calcule le *hit ratio*, c'est-à-dire la proportion des blocs lus depuis le cache de PostgreSQL, pour les cinq plus grosses requêtes en temps cumulé. Dans l'idéal, ce ratio serait à 100 %.

1.4 AUTO_EXPLAIN

- Tracer les plans des requêtes lentes automatiquement
- Contrib officielle
- Mise en place globale (traces) :
  - globale :

```
shared_preload_libraries='auto_explain'   -- redémarrage !
ALTER DATABASE erp SET auto_explain.log_min_duration = '3s' ;
```

  - session :

```
LOAD 'auto_explain' ;
SET auto_explain.log_analyze TO true;
```

L'outil `auto_explain` est habituellement activé quand on a le sentiment qu'une requête devient subitement lente à certains moments, et qu'on suspecte que son plan diffère entre deux exécutions. Elle permet de tracer dans les journaux applicatifs, voire dans la console, le plan de la requête dès qu'elle dépasse une durée configurée.

C'est une « contrib » officielle de PostgreSQL (et non une extension). Tracer systématiquement le plan d'exécution d'une requête souvent répétée prend de la place, et est assez coûteux. C'est donc un outil à utiliser parcimonieusement. En général on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre `auto_explain.log_min_duration`. Par défaut, ce paramètre vaut -1 pour ne tracer aucun plan.

Comme dans un `EXPLAIN` classique, on peut activer les options (par exemple `ANALYZE` ou `TIMING` avec, respectivement, un `SET auto_explain.log_analyze TO true;` ou un `SET auto_explain.log_timing TO true;`) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'`EXPLAIN`, notamment : `auto_explain.log_buffers`, `auto_explain.log_settings`.

Quant à `auto_explain.sample_rate`, il permet de ne tracer qu'un échantillon des requêtes (voir la documentation[4]).

Pour utiliser `auto_explain` globalement, il faut charger la bibliothèque au démarrage dans le fichier `postgresql.conf` via le paramètre `shared_preload_libraries`.

```
shared_preload_libraries='auto_explain'
```

[4] https://docs.postgresql.fr/current/auto-explain.html

Après un redémarrage de l'instance, il est possible de configurer les paramètres de capture des plans d'exécution par base de données. Dans l'exemple ci-dessous, l'ensemble des requêtes sont tracées sur la base de données `bench`, qui est utilisée par `pgbench`.

```
ALTER DATABASE bench SET auto_explain.log_min_duration = '0';
ALTER DATABASE bench SET auto_explain.log_analyze = true;
```

Attention, l'activation des traces complètes sur une base de données avec un fort volume de requêtes peut être très coûteux.

Un benchmark `pgbench` est lancé sur la base de données `bench` avec 1 client qui exécute 1 transaction par seconde pendant 20 secondes :

```
pgbench -c1 -R1 -T20 bench
```

Les plans d'exécution de l'ensemble les requêtes exécutées par `pgbench` sont alors tracés dans les traces de l'instance.

```
2021-07-01 13:12:55.790 CEST [1705] LOG:  duration: 0.041 ms  plan:
    Query Text: SELECT abalance FROM pgbench_accounts WHERE aid = 416925;
    Index Scan using pgbench_accounts_pkey on pgbench_accounts
        (cost=0.42..8.44 rows=1 width=4) (actual time=0.030..0.032 rows=1 loops=1)
        Index Cond: (aid = 416925)
2021-07-01 13:12:55.791 CEST [1705] LOG:  duration: 0.123 ms  plan:
    Query Text: UPDATE pgbench_tellers SET tbalance = tbalance + -3201 WHERE tid = 19;
    Update on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                               (actual time=0.120..0.121 rows=0 loops=1)
        ->  Seq Scan on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                                         (actual time=0.040..0.058 rows=1 loops=1)
            Filter: (tid = 19)
            Rows Removed by Filter: 99
2021-07-01 13:12:55.797 CEST [1705] LOG:  duration: 0.116 ms  plan:
    Query Text: UPDATE pgbench_branches SET bbalance = bbalance + -3201 WHERE bid = 5;
    Update on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.112..0.114 rows=0 loops=1)
        ->  Seq Scan on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.036..0.038 rows=1 loops=1)
            Filter: (bid = 5)
            Rows Removed by Filter: 9
[...]
```

Pour utiliser `auto_explain` uniquement dans la session en cours, il faut penser à descendre au niveau de message `LOG` (défaut de `auto_explain`). On procède ainsi :

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0;
SET auto_explain.log_analyze = true;
SET client_min_messages to log;
SELECT count(*)
   FROM pg_class, pg_index
   WHERE oid = indrelid AND indisunique;
```

```
LOG:  duration: 1.273 ms  plan:
Query Text: SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
Aggregate  (cost=38.50..38.51 rows=1 width=8)
    (actual time=1.247..1.248 rows=1 loops=1)
  ->  Hash Join  (cost=29.05..38.00 rows=201 width=0)
          (actual time=0.847..1.188 rows=198 loops=1)
        Hash Cond: (pg_index.indrelid = pg_class.oid)
        ->  Seq Scan on pg_index  (cost=0.00..8.42 rows=201 width=4)
                      (actual time=0.028..0.188 rows=198 loops=1)
              Filter: indisunique
              Rows Removed by Filter: 44
        ->  Hash  (cost=21.80..21.80 rows=580 width=4)
            (actual time=0.726..0.727 rows=579 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 29kB
              ->  Seq Scan on pg_class  (cost=0.00..21.80 rows=580 width=4)
                            (actual time=0.016..0.373 rows=579 loops=1)
 count
-------
   198
```

`auto_explain` est aussi un moyen de suivre les plans au sein de fonctions. Par défaut, un plan n'indique les compteurs de blocs *hit*, *read*, *temp*… que de l'appel global à la fonction.

Une fonction simple en PL/pgSQL est définie pour récupérer le solde le plus élevé dans la table `pgbench_accounts` :

```
CREATE OR REPLACE function f_max_balance() RETURNS int AS $$
    DECLARE
        acct_balance int;
    BEGIN
        SELECT max(abalance)
        INTO acct_balance
        FROM pgbench_accounts;
        RETURN acct_balance;
    END;
$$ LANGUAGE plpgsql ;
```

Un simple `EXPLAIN ANALYZE` de l'appel de la fonction ne permet pas d'obtenir le plan de la requête `SELECT max(abalance) FROM pgbench_accounts` contenue dans la fonction :

```
EXPLAIN (ANALYZE,VERBOSE) SELECT f_max_balance();

                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=4) (actual time=49.214..49.216 rows=1 loops=1)
   Output: f_max_balance()
 Planning Time: 0.149 ms
 Execution Time: 49.326 ms
```

Par défaut, `auto_explain` ne va pas capturer plus d'information que la commande `EXPLAIN ANALYZE`. Le fichier log de l'instance capture le même plan lorsque la fonction est exécutée.

```
2021-07-01 15:39:05.967 CEST [2768] LOG:  duration: 42.937 ms  plan:
    Query Text: select f_max_balance();
    Result  (cost=0.00..0.26 rows=1 width=4)
            (actual time=42.927..42.928 rows=1 loops=1)
```

Il est cependant possible d'activer le paramètre `log_nested_statements` avant l'appel de la fonction, de préférence uniquement dans la ou les sessions concernées :

```
\c bench
SET auto_explain.log_nested_statements = true;
SELECT f_max_balance();
```

Le plan d'exécution de la requête SQL est alors visible dans les traces de l'instance :

```
2021-07-01 14:58:40.189 CEST [2202] LOG:  duration: 58.938 ms  plan:
Query Text: select max(abalance)
            from pgbench_accounts
Finalize Aggregate
(cost=22632.85..22632.86 rows=1 width=4)
(actual time=58.252..58.935 rows=1 loops=1)
    ->  Gather
        (cost=22632.64..22632.85 rows=2 width=4)
        (actual time=57.856..58.928 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate
                (cost=21632.64..21632.65 rows=1 width=4)
                (actual time=51.846..51.847 rows=1 loops=3)
            ->  Parallel Seq Scan on pgbench_accounts
                    (cost=0.00..20589.51 rows=417251 width=4)
                    (actual time=0.014..29.379 rows=333333 loops=3)
```

pgBadger est capable de lire les plans tracés par `auto_explain`, de les intégrer à son rapport et d'inclure un lien vers depesz.com[5] pour une version plus lisible.

[5] https://explain.depesz.com/

1.5 PG_BUFFERCACHE

Qu'y a-t'il dans le cache de PostgreSQL ?
Fournit une vue :

- Pour chaque page (donc pour l'ensemble de l'instance)
  - fichier (donc objet) associé
  - OID base
  - fork (0 : table, 1 : FSM, 2 : VM)
  - numéro de bloc
  - isdirty
  - usagecount

Pour chaque entrée (bloc, par défaut de 8 ko) du cache disque de PostgreSQL, cette vue nous fournit les informations suivantes : le fichier (donc la table, l'index…), le bloc dans ce fichier, si ce bloc est synchronisé avec le disque (`isdirty` à `false`) ou s'il est « sale » (modifié en mémoire mais non synchronisé sur disque), et si ce bloc a été utilisé récemment (de 0 « plus utilisé dernièrement » à 5 « récemment utilisé »).

Cela permet donc de déterminer les *hot blocks* de la base, ou d'avoir une idée un peu plus précise du bon dimensionnement du cache : si rien n'atteint un `usagecount` de 5, le cache est manifestement trop petit : il n'est pas capable de détecter les pages devant impérativement rester en cache. Inversement, si vous avez énormément d'entrées à 0 et quelques pages avec des `usagecount` très élevés, toutes ces pages à 0 sont égales devant le mécanisme d'éviction du cache. Elles sont donc supprimées à peu près de la même façon que du cache du système d'exploitation. Le cache de PostgreSQL dans ce cas fait « double emploi » avec lui, et pourrait être réduit.

Attention toutefois avec les expérimentations sur les caches : il existe des effets de seuils. Un cache trop petit peut de la même façon qu'un cache trop grand avoir une grande fraction d'enregistrements avec un `usagecount` à 0. Par ailleurs, le cache bouge extrêmement rapidement par rapport à notre capacité d'analyse. Nous ne voyons qu'un instantané, qui peut ne pas refléter toute la réalité.

`isdirty` indique si un buffer est synchronisé avec le disque ou pas. Il est intéressant de vérifier qu'une instance dispose en permanence d'un certain nombre de buffers pour lesquels `isdirty` vaut `false` et pour lesquels `usagecount` vaut `0`. Si ce n'est pas le cas, c'est le signe :

- que `shared_buffers` est probablement trop petit (il n'arrive pas à contenir les modifications) ;
- que le `background_writer` n'est pas assez agressif.

De plus, avant la version 10, l'utilisation de cette extension est assez coûteuse car elle a besoin d'acquérir un verrou sur chaque page de cache inspectée. Chaque verrou est acquis pour une durée très courte, mais elle peut néanmoins entraîner une contention. L'impact a été diminué en version 10.

À titre d'exemple, cette requête affiche les dix plus gros objets de la base en cours en mémoire cache (dont, ici, deux index) :

```
SELECT c.relname,
       c.relkind,
       count(*) AS buffers,
       pg_size_pretty(count(*)*8192) as taille_mem
FROM   pg_buffercache b
INNER JOIN pg_class c
      ON b.relfilenode = pg_relation_filenode(c.oid)
        AND b.reldatabase IN (0, (SELECT oid FROM pg_database
                                  WHERE datname = current_database()))
GROUP BY c.relname, c.relkind
ORDER BY 3 DESC
LIMIT 5 ;
```

```
            relname              | relkind | buffers | taille_mem
---------------------------------+---------+---------+------------
 test_val_idx                    | i       |  162031 | 1266 MB
 test_pkey                       | i       |   63258 | 494 MB
 test                            | r       |   36477 | 285 MB
 pg_proc                         | r       |      47 | 376 kB
 pg_proc_proname_args_nsp_index | i       |      34 | 272 kB
```

On peut suivre la quantité de blocs *dirty* et l'*usagecount* avec une requête de ce genre, ici juste après une petite mise à jour de la table `test` :

```
SELECT
    relname,
    isdirty,
    usagecount,
    pinning_backends,
    count(bufferid)
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg%'
GROUP BY
        relname,
        isdirty,
        usagecount,
        pinning_backends
ORDER BY 1, 2, 3, 4 ;
```

```
    relname     | isdirty | usagecount | pinning_backends | count
----------------+---------+------------+------------------+--------
 brin_btree_idx | f       |          0 |                0 |      1
 brin_btree_idx | f       |          1 |                0 |   7151
 brin_btree_idx | f       |          2 |                0 |   3103
 brin_btree_idx | f       |          3 |                0 |  10695
 brin_btree_idx | f       |          4 |                0 | 141078
 brin_btree_idx | f       |          5 |                0 |      2
 brin_btree_idx | t       |          1 |                0 |      9
 brin_btree_idx | t       |          2 |                0 |      1
 brin_btree_idx | t       |          5 |                0 |     60
 test           | f       |          0 |                0 |  12371
 test           | f       |          1 |                0 |   6009
```

```
test           | f       |           2 |                 0 |   8466
test           | f       |           3 |                 0 |   1682
test           | f       |           4 |                 0 |   7393
test           | f       |           5 |                 0 |    112
test           | t       |           1 |                 0 |      1
test           | t       |           5 |                 0 |    267
test_pkey      | f       |           1 |                 0 |    173
test_pkey      | f       |           2 |                 0 |  27448
test_pkey      | f       |           3 |                 0 |   6644
test_pkey      | f       |           4 |                 0 |  10324
test_pkey      | f       |           5 |                 0 |   3420
test_pkey      | t       |           1 |                 0 |     57
test_pkey      | t       |           3 |                 0 |     81
test_pkey      | t       |           4 |                 0 |    116
test_pkey      | t       |           5 |                 0 |  15067
```

1.6 PG_PREWARM

- Charge des blocs en cache :

```
-- cache de PG
SELECT pg_prewarm ('pgbench_accounts', 'buffer') ;

-- cache de Linux (asynchrone)
SELECT pg_prewarm ('pgbench_accounts', 'prefetch') ;
-- cache (tous OS)
SELECT pg_prewarm ('pgbench_accounts', 'read') ;
```

- Ne pas oublier les index !
- N'interdit pas l'éviction
- Récupération du cache au redémarrage (v11)
  - avec un petit paramétrage

Grâce à l'extension pg_prewarm, intégrée à PostgreSQL, il est possible de pré-charger une table ou d'autres objets dans la mémoire de PostgreSQL, ou celle du système d'exploitation, pour améliorer les performances par la suite.

Par exemple, on charge la table `pgbench_accounts` dans le cache de PostgreSQL ainsi, et on le vérifie avec `pg_buffercache` :

```
CREATE EXTENSION IF NOT EXISTS pg_prewarm ;

SELECT pg_prewarm ('pgbench_accounts', 'buffer') ;
```

```
 pg_prewarm
------------
     163935
```

La valeur retournée correspond aux blocs chargés.

```
CREATE EXTENSION IF NOT EXISTS pg_buffercache ;

SELECT c.relname, count(*) AS buffers, pg_size_pretty(count(*)*8192) as taille_mem
FROM pg_buffercache b INNER JOIN pg_class c
                    ON b.relfilenode = pg_relation_filenode(c.oid)
GROUP BY c.relname ;
```

```
                relname                 | buffers | taille_mem
----------------------------------------+---------+------------
 …
 pgbench_accounts                       |  163935 | 1281 MB
 …
```

Il faut rappeler qu'une table ne se résume pas à ses données ! Il est au moins aussi intéressant de récupérer les index de la table en question :

```
SELECT pg_prewarm ('pgbench_accounts_pkey','buffer');
```

Si le cache de PostgreSQL ne suffit pas, celui du système peut être aussi préchargé :

```
SELECT pg_prewarm ('pgbench_accounts_pkey','read');
SELECT pg_prewarm ('pgbench_accounts_pkey','prefetch');  -- à préférer sur Linux
```

Charger une table en cache ne veut pas dire qu'elle va y rester ! Si les blocs chargés ne sont pas utilisés, ils seront évincés quand PostgreSQL aura besoin de faire de la place dans le cache, comme n'importe quels autres blocs.

Automatisation :

Cette extension peut sauvegarder le contenu du cache à intervalles réguliers ou lors de l'arrêt (propre) de PostgreSQL et le restaurer au redémarrage. Pour cela, paramétrer ceci :

```
shared_preload_libraries = 'pg_prewarm'
pg_prewarm.autoprewarm = on
pg_prewarm.autoprewarm_interval = '5min'
```

Les blocs concernés sont sauvés dans un fichier `autoprewarm.blocks` dans le répertoire PGDATA. Un *worker* nommé `autoprewarm leader` apparaîtra.

L'intérêt est de réduire énormément la phase de rechargement en cache des donnés actives après un redémarrage, accidentel ou non. En effet, une grosse base très active et aux disques un peu lents peut mettre longtemps à re-remplir son cache et à retrouver des performances acceptables. De plus, ne seront rechargées que les données en cache précédemment, donc à priori les parties de tables réellement actives.

Autres possibilités :

La documentation[6] décrit également comment charger :

- d'autres parties de la table comme la *visibility map* ;
- certains blocs précis.

[6] https://docs.postgresql.fr/current/pgprewarm.html

1.7 LANGAGES PROCÉDURAUX

- Procédures & fonctions en différents langages
- Par défaut : SQL, C et PL/pgSQL
- Extensions officielles : Perl, Python
- Mais aussi Java, Ruby, Javascript…
- Intérêts : fonctionnalités, performances

Les langages officiellement supportés par le projet sont :

- PL/pgSQL ;
- PL/Perl[7] ;
- PL/Python[8] ;
- PL/Tcl.

Voici une liste non exhaustive des langages procéduraux disponibles, à différents degrés de maturité :

- PL/sh[9] ;
- PL/R[10] ;
- PL/Java[11] ;
- PL/lolcode ;
- PL/Scheme ;
- PL/PHP ;
- PL/Ruby ;
- PL/Lua[12] ;
- PL/pgPSM ;
- PL/v8[13] (Javascript).

Tableau des langages supportés[14].

Pour qu'un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL. Les autres doivent être ajoutés à partir

[7] https://docs.postgresql.fr/current/plperl.html
[8] https://docs.postgresql.fr/current/plpython.html
[9] https://github.com/petere/plsh
[10] https://github.com/postgres-plr/plr
[11] https://tada.github.io/pljava/
[12] https://github.com/pllua/pllua
[13] https://github.com/plv8/plv8

des paquets de la distribution ou du PGDG, ou compilés à la main, puis l'extension installée dans la base :

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
-- etc.
```

Ces fonctions peuvent être utilisées dans des index fonctionnels et des triggers comme toute fonction SQL ou PL/pgSQL.

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n'est pas performant quand il s'agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage. S'ajoute l'intérêt de ne pas avoir à réécrire en PL/pgSQL des fonctions existantes dans d'autres langages ou d'accéder à des modules bien établis de ces langages.

1.7.1 Avantages & inconvénients

- PL/pgSQL plus performant pour l'accès aux données
- Chaque langage a son point fort
- Performances :
  - latence (pas d'allers-retours)
  - accès aux données depuis les fonctions
  - bibliothèques de chaque langage
  - index
- Langages *trusted* / *untrusted*

Il est courant de considérer que la logique métier (les fonctions) doit être intégralement dans l'applicatif, et pas dans la base de données. Même si l'on adopte ce point de vue, il faut savoir faire des exceptions pour prendre en compte les performances : une fonction, en PL/pgSQL ou un autre langage, exécutée dans la base de données économisera des aller-retours entre la base et le serveur applicatif, ce qui peut avoir un impact énorme (latence due à de nombreux ordres, ou durée de transfert des résultats intermédiaires).

Une fonction en Perl ou Python complexe peut servir aussi de critère d'indexation, pour des gains parfois énormes.

Le PL/pgSQL[15] est le mieux intégré des langages (avec le C), D'autres langages peuvent subir une pénalité due à la communication avec l'interpréteur (car c'est bien celui présent sur le serveur qui est utilisé). Cependant, ils peuvent apporter des fonctionnalités qui manquent à PostgreSQL : PL/R, bibliothèques numériques NumPy et Scipy de Python…

Pour des raisons de sécurité, on distingue des langages *trusted* et *untrusted*. Un langage *trusted* est disponible pour tous les utilisateurs de la base, n'autorise pas l'accès à des données normalement inaccessibles à l'utilisateur, mais quelques fonctionnalités ont pu être supprimées (interaction avec l'environnement notamment). Un langage *untrusted* n'a pas ces limites et les fonctions ne peuvent être créées que par un super-utilisateur. PL/pgSQL est *trusted*. PL/Python n'existe qu'en *untrusted* (l'extension pour la version 3 se nomme `plpython3u`). PL/Perl existe dans les deux versions (extensions `plperl` et `plperlu`).

[15] https://www.postgresql.org/docs/current/plpgsql-overview.html

1.8 HLL

- `COUNT(DISTINCT)` est notoirement lent
- Principe d'HyperLogLog :
  - travail sur des hachages, avec perte
  - estimation statistique
  - non exact, mais beaucoup plus rapide
- Exemple :

```
SELECT mois, hll_cardinality(hll_add_agg(hll_hash_text( id )))
FROM voyages ;
```

- Type `hll` pour pré-agréger des données

Les décomptes de valeurs distinctes sont une opération assez courante dans certains domaines : décompte de visiteurs distincts d'un site web ou d'un lieu, de patients d'un hôpital, de voyageurs, etc. Or `COUNT(DISTINCT)` est notoirement lent quand on fait face à un grand nombre de valeurs distinctes, à cause de la déduplication des valeurs, du maintien d'un espace pour le décompte, du besoin fréquent de fichiers temporaires…

Le principe de HyperLogLog est de ne pas opérer de calculs exacts mais de compiler un hachage des données rencontrées, avec perte, et donc beaucoup de manière plus compacte ; puis d'étudier la répartition statistique des valeurs rencontrées, et d'en déduire la volumétrie approximative. En effet, dans beaucoup de contexte, il n'est pas forcément utile de connaître le nombre *exact* de clients, de passagers… Une approximation peut répondre à beaucoup de besoins. En fonction de l'imprécision acceptée, on peut économiser beaucoup de mémoire et de temps (un gain d'un facteur supérieur à 10 est fréquent).

Une extension dédiée existe, à présent maintenue par Citusdata. Le source est sur Github[16], et on trouvera les paquets dans les dépôts communautaires habituels.

La bibliothèque doit être préchargée dans chaque session pour être exécuté par l'optimiseur pour influencer les plans générés :

```
shared_preload_libraries = 'hll'
```

Puis charger l'extension dans la base concernée :

```
CREATE EXTENSION hll ;
```

On peut alors immédiatement remplacer un `COUNT(DISTINCT id)` par cet équivalent :

[16] https://github.com/citusdata/postgresql-hll

```
SELECT mois, hll_cardinality(hll_add_agg(hll_hash_text( id )))
FROM   matable ;
```

Concrètement, l'identifiant à trier est haché (il y a une fonction dédiée par type). Puis ces hachages sont agrégés en un ensemble par la fonction `hll_add_agg()`. Ensuite, la fonction `hll_cardinality()` estime le nombre de valeurs distinctes originales à partir de cet ensemble.

Le paramétrage par défaut est déjà pertinent pour des cardinalités jusqu'au billion (10^{12}) d'après la documentation[17], avec une erreur de l'ordre du pour cent. La précision de l'estimation peut être ajustée de manière générale, ou bien comme paramètre à la fonction de création de l'ensemble, comme dans ces exemples (ici avec les valeurs par défaut) :

```
SELECT hll_set_defaults(11, 5, -1, 1) ;
```

```
SELECT hll_cardinality(hll_add_agg(hll_hash_text( id ), 11, 5, -1, 1 ))
FROM matable ;
```

Les deux premiers paramètres sont les plus importants : le nombre de registres utilisés (de 4 à 31, chaque incrément de 1 doublant la taille mémoire requise), et la taille des registres en bits (de 1 à 8). Des valeurs trop grandes risquent de rendre l'estimation inutilisable (résultat `NaN`).

Dans le monde décisionnel, il est fréquent de créer des tables d'agrégat avec des résultats précalculés sur un jour ou un mois. Cela ne fonctionne que partiellement pour des `COUNT(DISTINCT)` : par exemple, on ne peut sommer le nombre de voyageurs distincts de chaque mois pour calculer celui sur l'année, ce sont peut-être les mêmes clients toute l'année. L'extension apporte donc aussi un type `hll` destiné à stocker des résultats agrégés issus d'un appel à `hll_add_agg()`. On agrège le contenu de ces champs `hll` avec la fonction `hll_union_agg()`, et on peut procéder à l'estimation sur l'ensemble avec `hll_cardinality`.

[17] https://github.com/citusdata/postgresql-hll

1.9 QUIZ

https://dali.bo/x2_quiz

1.10 TRAVAUX PRATIQUES

1.10.1 Indexation de pattern avec les varchar_patterns et pg_trgm

But : Indexer des patterns avec les varchar_patterns et 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 ».

1.10.2 auto_explain

But : Capturer les plans d'exécutions automatiquement avec auto_explain

Installer le module `auto_explain` (documentation : https://docs.postgresql.fr/current/auto-explain.html).

Exécuter des requêtes sur n'importe quelle base de données, et inspecter les traces générées.

Passer le niveau de messages de sa session (`client_min_messages`) à `log`.

1.10.3 pg_stat_statements

But : Analyser les performances des requêtes avec pg_stat_statements

- pg_stats_statements nécessite une bibliothèque préchargée. La positionner dans le fichier `postgresql.conf`, redémarrer PostgreSQL et créer l'extension.

- Inspecter le contenu de l'extension `pg_stat_statements` (`\dx` et `\dx+`).

- Vérifier que le serveur est capable d'activer la mesure de la durée des entrées-sorties avec `pg_test_timing`. Puis l'activer (`track_io_timing`), sans oublier de redémarrer PostgreSQL.
- Depuis un autre terminal, créer une base **pgbench** (si pas déjà disponible), l'initialiser (même si elle existait), et lancer une activité dessus :

```
# en tant qu'utilisateur postgres
createdb -e pgbench
/usr/pgsql-16/bin/pgbench -i -s135 pgbench
/usr/pgsql-16/bin/pgbench -c5 -j1 pgbench -T 600 -P1
```

- Dans la vue `pg_stat_statements`, récupérer les 5 requêtes les plus gourmandes en temps cumulé sur l'instance et leur nombre de lignes.

Quelle est la requête générant le plus d'écritures directes sur disques (*written*) ? Et en temps d'écriture ?

Quel est le *hit ratio* des requêtes les plus fréquentes ?

1.10.4 PL/Python, import de page web et compression

But : Importer et stocker une page web au format compressé avec PL/Python

Sur la base du code suivant en python 3 utilisant un des modules standard (documentation : https://docs.python.org/3/library/urllib.request.html), créer une fonction PL/Python récupérant le code HTML d'une page web avec un simple `SELECT pageweb('https://www.postgresql.org/')` :

```
import urllib.request
f = urllib.request.urlopen('https://www.postgresql.org/')
print (f.read().decode('utf-8'))
```

Stocker le résultat dans une table.

Puis stocker cette page en compression maximale dans un champ `bytea`, en passant par une fonction python inspirée du code suivant (documentation : https://docs.python.org/3/library/bz2.html) :

```
import bz2
```

```
compressed_data = bz2.compress(data, compresslevel=9)
```

Écrire la fonction de décompression avec la fonction python `bz2.decompress`.

Utiliser ensuite `convert_from( bytea, 'UTF8')` pour récupérer un `text`.

1.10.5 PL/Perl et comparaison de performances

But : Exécuter un traitement performant avec PL/Perl

Ce TP s'inspire d'un billet de blog de Daniel Vérité[18], qui a publié le code des fonctions sur le wiki PostgreSQL sous licence PostgreSQL. Le principe est d'implémenter un remplacement en masse de nombreuses chaînes de caractères par d'autres. Une fonction codée en PL/perl peut se révéler plus rapide qu'une autre en PL/pgSQL.

Il utilise la base de données contenant des livres issus du projet Gutenberg, dans sa version complète qui contient *Les Misérables* de Victor Hugo. 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
```

Créer la fonction `multi_replace` en PL/pgSQL à partir du wiki PostgreSQL : https://wiki.postgresql.org/wiki/Multi_Replace_plpgsql

Récupérer la fonction en PL/perl sur le même wiki : https://wiki.postgresql.org/wiki/Multi_Replace_Perl.

Vérifier que les deux fonctions ont le même nom mais des types de paramètres différents.

Le test va consister à transposer tous les noms et lieux des *Misérables* de Victor Hugo dans une version américaine :

- charger la base du projet Gutenberg si elle n'est pas déjà en place.

[18] https://blog-postgresql.verite.pro/2020/01/22/multi-replace.html

- créer une table `miserables` reprenant tous les livres dont le titre commence par « Les misérables ».

Tester le bon fonctionnement avec ces requêtes :

```
SELECT multi_replace (contenu,'{"Valjean":"Valjohn", "Cosette":"Lucy"}'::jsonb)
FROM miserables
WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5 ;

SELECT multi_replace(contenu, '{Valjean,Cosette}', '{Valjohn, Lucy}' )
FROM miserables
WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5 ;
```

Pour faciliter la modification, prévoir une table pour stocker les critères :

```
CREATE TABLE remplacement (j jsonb, old_t text[], new_t text[]) ;
```

Insérer par exemple les données suivantes :

```
 INSERT INTO remplacement (j)
SELECT '{"Valjean":"Valjohn", "Jean Valjean":"John Valjohn",
"Cosette":"Lucy", "Fantine":"Fanny", "Javert":"Green",
"Thénardier":"Thenardy", "Éponine":"Sharon", "Azelma":"Azealia",
"Marius":"Marc", "Gavroche":"Garry", "Enjolras":"Joker",
"Notre-Dame":"Empire State Building", "Victor Hugo":"Victor Hugues",
"Hugo":"Hugues", "Fauchelevent":"Dropwind", "Bouchart":"Butcher",
"Célestine":"Celeste","Mabeuf":"Myoax", "Leblanc":"White",
"Combeferre":"Combiron", "Magloire":"Glory",
"Gillenormand":"Jillnorthman", "France":"États-Unis",
"Paris":"New York", "Louis Philippe":"Andrew Jackson" }'::jsonb  ;
```

Copier le contenu sous forme de tableau de caractères dans les autres champs :

```
UPDATE remplacement
SET old_t = noms_old , new_t = noms_new
FROM (SELECT array_agg (key) AS noms_old, array_agg (value) AS noms_new
      FROM (
               SELECT (jsonb_each_text (j)).* FROM remplacement
       ) j1
      ) j2 ;
```

Comparer la performance des deux fonctions suivantes :

```
\pset pager off
-- fonction en PL/perl
EXPLAIN (ANALYZE, BUFFERS)
SELECT  multi_replace (contenu, (SELECT j FROM remplacement))
FROM miserables ;

-- fonction en PL/pgSQL
EXPLAIN (ANALYZE, BUFFERS)
```

```
SELECT  multi_replace (contenu,
                       (SELECT old_t FROM remplacement),
                      (SELECT new_t FROM remplacement) )
FROM miserables ;
```

1.10.6 hll

But : Estimer le nombre de valeurs distinctes plus rapidement avec hll

- Installer l'extension `hll` dans la base de données de test :
- le paquet est `hll_14` ou `postgresql-14-hll` (ou l'équivalent pour les autres numéros de versions) selon la distribution ;
- l'extension se nomme `hll` ;
- elle nécessite d'être préalablement déclarée dans `shared_preload_libraries`.

- Créer un jeu de données simulant des voyages en transport en commun, par passager selon la date :

```
CREATE TABLE voyages
(voyage_id      bigint  GENERATED ALWAYS AS IDENTITY,
 passager_id    text,
 d              date
) ;

INSERT INTO voyages (passager_id, d)
SELECT sem+mod(i, sem+1) ||'-'|| mod(i,77777) AS passager_id, d
FROM generate_series (0,51) sem,
      LATERAL
        (SELECT i,
         '2019-01-01'::date + sem * interval '7 days' + i * interval '2s' AS d
       FROM generate_series (1,
         (case when sem in (31,32,33) then 0 else 22 end +abs(30-sem))*5000 ) i
       ) j
;
```

- Activer l'affichage du temps (`timing`).
- Désactiver JIT et le parallélisme.
- Passer la mémoire de tri à 1 Go.
- Précharger la table dans le cache de PostgreSQL.

- Calculer, par mois, le nombre exact de voyages et de passagers **distincts**.
- Dans le plan de la requête, chercher où est perdu le temps.

- Calculer, pour l'année, le nombre exact de voyages et de passagers **distincts**.

- Recompter les passagers dans les deux cas en remplaçant le `COUNT(DISTINCT)` par cette expression : `hll_cardinality(hll_add_agg(hll_hash_text(passager_id)))::int`

- Réexécuter les requêtes après modification du paramétrage de `hll` :

```
SELECT hll_set_defaults(17, 5, -1, 0);
```

- Créer une table d'agrégat par mois avec un champ d'agrégat `hll` et la remplir.

À partir de cette table d'agrégat :

- calculer le nombre moyen mensuel de passagers distincts,
- recalculer le nombre de passagers distincts sur l'année à partir de cette table d'agrégat.

Avec une fonction de fenêtrage sur `hll_union_agg`, calculer une moyenne glissante sur 3 mois du nombre de passagers distincts.

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

1.11.1 Indexation de pattern 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
```

1.11.2 auto_explain

> Installer le module `auto_explain` (documentation : https://docs.postgresql.fr/current/auto-explain.html).

Dans le fichier `postgresql.conf`, chargement du module et activation globale pour *toutes* les requêtes (ce qu'on évitera de faire en production) :

```
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = 0
```

Redémarrer PostgreSQL.

> Exécuter des requêtes sur n'importe quelle base de données, et inspecter les traces générées.

Le plan de la moindre requête (même un `\d+`) doit apparaître dans la trace.

> Passer le niveau de messages de sa session (`client_min_messages`) à `log`.

Il est possible de recevoir les messages directement dans sa session. Tous les messages de log sont marqués d'un niveau de priorité. Les messages produits par `auto_explain` sont au niveau `log`. Il suffit donc de passer le paramètre `client_min_messages` au niveau `log`.

Positionner le paramètre de session comme ci-dessous, ré-exécuter la requête.

```
SET client_min_messages TO log;
SELECT…
```

1.11.3 pg_stat_statements

- pg_stats_statements nécessite une bibliothèque préchargée. La positionner dans le fichier `postgresql.conf`, redémarrer PostgreSQL et créer l'extension.

Si une autre extension (ici `auto_explain`) est également présente, on peut les lister ainsi :

```
shared_preload_libraries = 'auto_explain,pg_stat_statements'
```

Redémarrer PostgreSQL.

Dans la base **postgres** (par exemple), créer l'extension :

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements ;
```

- Inspecter le contenu de l'extension `pg_stat_statements` (`\dx` et `\dx+`).

```
 \dx+ pg_stat_statements

Objets dans l'extension « pg_stat_statements »
         Description d'objet
-------------------------------------
 function pg_stat_statements(boolean)
 function pg_stat_statements_reset()
 view pg_stat_statements
```

- Vérifier que le serveur est capable d'activer la mesure de la durée des entrées-sorties avec `pg_test_timing`. Puis l'activer (`track_io_timing`), sans oublier de redémarrer PostgreSQL.

pg_test_timing[19] est livré avec PostgreSQL :

```
/usr/pgsql-16/bin/pg_test_timing

Testing timing overhead for 3 seconds.
Per loop time including overhead: 33.24 ns
Histogram of timing durations:
  < us   % of total      count
     1     97.25509   87770521
     2      2.72390    2458258
     4      0.00072        646
     8      0.00244       2200
    16      0.00984       8882
    32      0.00328       2958
    64      0.00298       2689
   128      0.00099        892
   256      0.00055        499
   512      0.00016        141
  1024      0.00006         53
  2048      0.00000          1
```

[19] https://docs.postgresql.fr/current/pgtesttiming.html

Si le temps de mesure n'est que de quelques dizaines de nanosecondes, c'est OK. (C'est le cas sur presque toutes les machines et systèmes d'exploitation actuels, mais il y a parfois des surprises.) Sinon, éviter de faire ce qui suit sur un serveur de production. Sur une machine de formation, ce n'est pas un problème.

Dans le fichier `postgresql.conf`, positionner :

```
track_io_timing = on
```

Changer ce paramètre nécessite de redémarrer PostgreSQL.

> – Depuis un autre terminal, créer une base **pgbench** (si pas déjà disponible), l'initialiser (même si elle existait), et lancer une activité dessus :
>
> ```
> # en tant qu'utilisateur postgres
> createdb -e pgbench
> /usr/pgsql-16/bin/pgbench -i -s135 pgbench
> /usr/pgsql-16/bin/pgbench -c5 -j1 pgbench -T 600 -P1
> ```

```
createdb -e pgbench

SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE pgbench;

/usr/pgsql-16/bin/pgbench -i -s135 pgbench

…
creating tables...
generating data (client-side)...
13500000 of 13500000 tuples (100%) done (elapsed 10.99 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 14.97 s (drop tables 0.00 s, create tables 0.02 s, client-side generate 11.04
 ↪  s, vacuum 0.34 s, primary keys 3.57 s).

/usr/pgsql-16/bin/pgbench -c5 -j1 pgbench -T 600 -P1

pgbench (16.2)
starting vacuum...end.
progress: 1.0 s, 2364.9 tps, lat 2.078 ms stddev 1.203, 0 failed
progress: 2.0 s, 2240.0 tps, lat 2.221 ms stddev 0.871, 0 failed
…
```

On a donc 5 clients qui vont mettre à jour la base à raison de 2000 transactions par seconde (valeur très dépendante des CPUs et des disques).

> – Dans la vue `pg_stat_statements`, récupérer les 5 requêtes les plus gourmandes en temps cumulé sur l'instance et leur nombre de lignes.

```
SELECT calls, query, rows,
total_exec_time*interval '1ms' AS tps_total
FROM pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 5
\gx
```

Le résultat va dépendre de l'historique de votre instance, et du temps déroulé depuis le lancement de `pgbench`, mais c'est probablement proche de ceci :

```
-[ RECORD 1 ]-------------------------------------------------------------
calls      | 879669
query      | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
rows       | 879669
tps_total | 00:02:56.184131
-[ RECORD 2 ]-------------------------------------------------------------
calls      | 879664
query      | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
rows       | 879664
tps_total | 00:00:44.803628
-[ RECORD 3 ]-------------------------------------------------------------
calls      | 879664
query      | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
rows       | 879664
tps_total | 00:00:12.196055
-[ RECORD 4 ]-------------------------------------------------------------
calls      | 1
query      | copy pgbench_accounts from stdin with (freeze on)
rows       | 13500000
tps_total | 00:00:10.698976
-[ RECORD 5 ]-------------------------------------------------------------
calls      | 879664
query      | SELECT abalance FROM pgbench_accounts WHERE aid = $1
rows       | 879664
tps_total | 00:00:06.530169
```

Noter que l'unique `COPY` pour créer la base dure plus que les centaines de milliers d'occurences de la cinquième requête.

> Quelle est la requête générant le plus d'écritures directes sur disques (*written*) ? Et en temps d'écriture ?

Pour les *written*, il faut tenir compte des trois sources : blocs du cache partagé, blocs des *backends*, fichiers temporaires.

```
SELECT calls,
       pg_size_pretty(8192::numeric
       * (shared_blks_written+local_blks_written+temp_blks_written)) AS written,
       pg_size_pretty(8192::numeric*shared_blks_written)  AS shared_written,
       pg_size_pretty(8192::numeric*temp_blks_written)    AS temp_written,
       blk_write_time * interval '1ms'      AS blk_write_time,
       temp_blk_write_time * interval '1ms' AS temp_blk_write_time,
      query
FROM pg_stat_statements
ORDER BY shared_blks_written+local_blks_written+temp_blks_written DESC LIMIT 3 ;

-[ RECORD 1 ]-------+--------------------------------------------------------------
calls               | 2400667
written             | 15 GB
shared_written      | 15 GB
temp_written        | 0 bytes
```

```
blk_write_time       | 00:00:11.840499
temp_blk_write_time  | 00:00:00
query                | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE a.
                     |.id = $2
-[ RECORD 2 ]--------+-----------------------------------------------------------
calls                | 1
written              | 3442 MB
shared_written       | 3442 MB
temp_written         | 0 bytes
blk_write_time       | 00:00:00
temp_blk_write_time  | 00:00:00
query                | copy pgbench_accounts from stdin with (freeze on)
-[ RECORD 3 ]--------+-----------------------------------------------------------
calls                | 1
written              | 516 MB
shared_written       | 0 bytes
temp_written         | 516 MB
blk_write_time       | 00:00:00
temp_blk_write_time  | 00:00:00
query                | alter table pgbench_accounts add primary key (aid)
```

Il y a donc beaucoup d'écritures directes. C'est le signe que le cache en écriture de PostgreSQL est insuffisant (la base fait 2 Go, à peu près intégralement balayée, et le `shared_buffers` par défaut ne fait que 128 Mo) ou que le `background writer` doit être modifié pour nettoyer plus souvent les blocs *dirty*.

On note que l'`UPDATE` et le `COPY` ont écrit des blocs qui auraient dû passer uniquement par le cache, alors le `ALTER TABLE`, lui, a essentiellement écrit un fichier temporaire (c'est logique lors d'une création d'index).

Avec des *shared buffers* plus importants, les `shared_written` sont quasiment absents. Ils proviennent essentiellement d'ordres lourds comme `COPY`.

> Quel est le *hit ratio* des requêtes les plus fréquentes ?

```
SELECT calls, total_exec_time,
       round(100.0*shared_blks_hit
       /nullif(shared_blks_hit+shared_blks_read, 0),2) AS "hit %",
       query
FROM  pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 5 ;
```

```
-[ RECORD 1 ]---+-----------------------------------------------------------------
calls           | 2400667
total_exec_time | 464702.3557850064
hit %           | 73.24
query           | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =.
                |. $2
-[ RECORD 2 ]---+-----------------------------------------------------------------
calls           | 2400658
total_exec_time | 141310.02034101041
hit %           | 100.00
query           | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =.
                |. $2
```

```
-[ RECORD 3 ]---+-----------------------------------------------------------------
calls           | 2400659
total_exec_time | 34201.65339700031
hit %           | 100.00
query           | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = .
                |.$2
-[ RECORD 4 ]---+-----------------------------------------------------------------
calls           | 2400661
total_exec_time | 16494.857696000774
hit %           | 100.00
query           | SELECT abalance FROM pgbench_accounts WHERE aid = $1
-[ RECORD 5 ]---+-----------------------------------------------------------------
calls           | 2400656
total_exec_time | 11685.776115000388
hit %           | 100.00
query           | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES.
                |. ($1, $2, $3, $4, CURRENT_TIMESTAMP)
```

On constate que le *hit ratio* est parfait, sauf la première requête. C'est logique, car la table `pgbench_accounts` ne tient pas dans le cache par défaut et elle est balayée à peu près entièrement par les requêtes de pgbench.

1.11.4 PL/Python, import de page web et compression

> Sur la base du code suivant en python 3 utilisant un des modules standard (documentation : https://docs.python.org/3/library/urllib.request.html), créer une fonction PL/Python récupérant le code HTML d'une page web avec un simple `SELECT pageweb('https://www.postgresql.org/')` :
>
> ```
> import urllib.request
> f = urllib.request.urlopen('https://www.postgresql.org/')
> print (f.read().decode('utf-8'))
> ```

Il faut bien évidemment que PL/Python soit installé. D'abord le paquet, ici sous Rocky Linux 8 avec PostgreSQL 14 :

```
 # dnf install postgresql14-plpython3
```

Sur Debian et dérivés ce sera :

```
 # apt install postgresql-plpython3-14
```

Puis, dans la base de données concernée :

```
CREATE EXTENSION plpython3u ;
```

La fonction PL/Python est :

```
CREATE OR REPLACE FUNCTION pageweb (url text)
  RETURNS text
AS $$
import urllib.request
f = urllib.request.urlopen(url)
return f.read().decode('utf-8')
$$ LANGUAGE plpython3u COST 10000;
```

Évidemment, il ne s'agit que d'un squelette ne gérant pas les erreurs, les redirections, etc.

> Stocker le résultat dans une table.

On vérifie ainsi le bon fonctionnement :

```
CREATE TABLE pagesweb (url text, page text, pagebz2 bytea, page2 text ) ;

INSERT INTO pagesweb (url, page)
SELECT 'https://www.postgresql.org/', pageweb('https://www.postgresql.org/') ;
```

> Puis stocker cette page en compression maximale dans un champ `bytea`, en passant par une fonction python inspirée du code suivant (documentation : https://docs.python.org/3/library/bz2.html) :
>
> ```
> import bz2
> compressed_data = bz2.compress(data, compresslevel=9)
> ```

```
import bz2
c=bz2.compress(data, compresslevel=9)
```

Même si la page récupérée est en texte, la fonction python exige du binaire, donc le champ en entrée sera du `bytea` :

```
-- version pour bytea
CREATE OR REPLACE FUNCTION bz2 (objet bytea)
  RETURNS bytea
AS $$
import bz2
return bz2.compress(objet, compresslevel=9)
$$ LANGUAGE plpython3u IMMUTABLE COST 1000000;
```

On peut faire la conversion depuis `text` à l'appel ou modifier la fonction pour qu'elle convertisse d'elle-même. Mais le plus confortable est de créer une fonction SQL de même nom qui se chargera de la conversion. Selon le type en paramètre, l'une ou l'autre fonction sera appelée.

```
-- fonction d'enrobage pour s'épargner une conversion explicite en bytea
CREATE OR REPLACE FUNCTION bz2 (objet text)
  RETURNS bytea
AS $$
SELECT bz2(objet::bytea) ;
$$ LANGUAGE sql IMMUTABLE ;
```

Compression de la page :

```
UPDATE pagesweb
SET pagebz2 = bz2 (page) ;
```

NB : PostgreSQL stocke déjà les textes longs sous forme compressée (mécanisme du TOAST).
Tout ceci n'a donc d'intérêt que pour gagner quelques octets supplémentaires, ou si le `.bz2` doit être réutilisé directement. Noter que l'on utilise ici uniquement des fonctionnalités standards de PostgreSQL et python3, sans module extérieur à la fiabilité inconnue.
De plus, les données ne quittent pas le serveur, épargnant du trafic réseau.

Écrire la fonction de décompression avec la fonction python `bz2.decompress`.

```
CREATE OR REPLACE FUNCTION bz2d (objet bytea)
  RETURNS bytea
AS $$
import bz2
return bz2.decompress(objet)
$$ LANGUAGE plpython3u IMMUTABLE COST 1000000;
```

Utiliser ensuite `convert_from( bytea, 'UTF8')` pour récupérer un `text`.

```
CREATE OR REPLACE FUNCTION bz2_to_text (objetbz2 bytea)
  RETURNS text
```

```
AS $$
SELECT convert_from( bz2d(objetbz2), 'UTF8')
$$ LANGUAGE sql IMMUTABLE ;
```

Vérification que l'on obtient au final le même texte qu'avant compression :

```
UPDATE pagesweb
SET page2 = bz2_to_text( pagebz2 )
;
-- Vérification que la page décompressée est identique à l'originale
SELECT count(*) AS pages,
       count(*) FILTER (WHERE page = page2) AS pages_identique
FROM   pagesweb ;
```

```
 pages | pages_identique
-------+-----------------
     1 |               1
```

1.11.5 PL/Perl et comparaison de performances

> Créer la fonction `multi_replace` en PL/pgSQL à partir du wiki PostgreSQL : https://wiki.postgresql.org/wiki/Multi_Replace_plpgsql

Le code sur le wiki est le suivant :

```
/* This function quotes characters that may be interpreted as special
   in a regular expression.
   It's used by the function below and declared separately for clarity. */
CREATE FUNCTION quote_meta(text) RETURNS text AS $$
  SELECT regexp_replace($1, '([\[\]\\\^\$\.\|\?\*\+\(\)])', '\\\1', 'g');
$$ LANGUAGE SQL strict immutable;

/* Substitute a set of substrings within a larger string.
   When several strings match, the longest wins.
   Similar to php's strtr(string $str, array $replace_pairs).
   Example:
   select multi_replace('foo and bar is not foobar',
            '{"bar":"foo", "foo":"bar", "foobar":"foobar"}'::jsonb);
   => 'bar and foo is not foobar'
 */
CREATE FUNCTION multi_replace(str text, substitutions jsonb)
RETURNS text
AS $$
DECLARE
 rx text;
 s_left text;
 s_tail text;
 res text:='';
BEGIN
 SELECT string_agg(quote_meta(term), '|' )
 FROM jsonb_object_keys(substitutions) AS x(term)
   WHERE term <> ''
 INTO rx;

 IF (COALESCE(rx, '') = '') THEN
   -- the loop on the RE can't work with an empty alternation
   RETURN str;
 END IF;

 rx := concat('^(.*?)(', rx, ')(.*)$'); -- match no more than 1 row

 loop
   s_tail := str;
   SELECT
       concat(matches[1], substitutions->>matches[2]),
       matches[3]
    FROM
      regexp_matches(str, rx, 'g') AS matches
    INTO s_left, str;

   exit WHEN s_left IS NULL;
   res := res || s_left;
```

```
 END loop;

 res := res || s_tail;
 RETURN res;

END
$$ LANGUAGE plpgsql strict immutable;
```

> Récupérer la fonction en PL/perl sur le même wiki : https://wiki.postgresql.org/wiki/Multi_Replace_Perl.

Évidemment, il faudra l'extension dédiée au langage Perl :

```
 # dnf install postgresql14-plperl
```

```
CREATE EXTENSION plperl;
```

Le code de la fonction est :

```
CREATE FUNCTION multi_replace(string text, orig text[], repl text[])
RETURNS text
AS $BODY$
  my ($string, $orig, $repl) = @_;
  my %subs;

  if (@$orig != @$repl) {
     elog(ERROR, "array sizes mismatch");
  }
  if (ref @$orig[0] eq 'ARRAY' || ref @$repl[0] eq 'ARRAY') {
     elog(ERROR, "array dimensions mismatch");
  }

  @subs{@$orig} = @$repl;

  my $re = join "|", map quotemeta,
     sort { (length($b) <=> length($a)) } keys %subs;
  $re = qr/($re)/;

  $string =~ s/$re/$subs{$1}/g;
  return $string;
$BODY$ language plperl strict immutable;
```

> Vérifier que les deux fonctions ont le même nom mais des types de paramètres différents.

```
 \df multi_replace
                        Liste des fonctions
Schéma |      Nom      | …résultat |    Type … paramètres      | Type
-------+---------------+-----------+---------------------------+------
public | multi_replace | text      | string text,              | func
       |               |           | orig text[], repl text[]  |
public | multi_replace | text      | str text,                 | func
       |               |           | substitutions jsonb       |
```

PostgreSQL sait quelle fonction appeler selon les paramètres fournis.

Le test va consister à transposer tous les noms et lieux des *Misérables* de Victor Hugo dans une version américaine :

> Le test va consister à transposer tous les noms et lieux des *Misérables* de Victor Hugo dans une version américaine :
>
> - charger la base du projet Gutenberg si elle n'est pas déjà en place.
> - créer une table `miserables` reprenant tous les livres dont le titre commence par « Les misérables ».

```
CREATE TABLE miserables as select * from textes
  WHERE livre LIKE 'Les misérables%' ;
```

Cette table fait 68 000 lignes.

> Tester le bon fonctionnement avec ces requêtes :
>
> ```
> SELECT multi_replace (contenu,'{"Valjean":"Valjohn", "Cosette":"Lucy"}'::jsonb)
> FROM miserables
> WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5 ;
>
> SELECT multi_replace(contenu, '{Valjean,Cosette}', '{Valjohn, Lucy}' )
> FROM miserables
> WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5 ;
> ```

Le texte affiché doit comporter « Jean Valjohn » et « Lucy ».

> Pour faciliter la modification, prévoir une table pour stocker les critères :
>
> ```
> CREATE TABLE remplacement (j jsonb, old_t text[], new_t text[]) ;
> ```

> Insérer par exemple les données suivantes :
>
> ```
>  INSERT INTO remplacement (j)
> SELECT '{"Valjean":"Valjohn", "Jean Valjean":"John Valjohn",
> "Cosette":"Lucy", "Fantine":"Fanny", "Javert":"Green",
> "Thénardier":"Thenardy", "Éponine":"Sharon", "Azelma":"Azealia",
> "Marius":"Marc", "Gavroche":"Garry", "Enjolras":"Joker",
> "Notre-Dame":"Empire State Building", "Victor Hugo":"Victor Hugues",
> "Hugo":"Hugues", "Fauchelevent":"Dropwind", "Bouchart":"Butcher",
> "Célestine":"Celeste","Mabeuf":"Myoax", "Leblanc":"White",
> "Combeferre":"Combiron", "Magloire":"Glory",
> "Gillenormand":"Jillnorthman", "France":"États-Unis",
> "Paris":"New York", "Louis Philippe":"Andrew Jackson" }'::jsonb  ;
> ```

> Copier le contenu sous forme de tableau de caractères dans les autres champs :
>
> ```
> UPDATE remplacement
> SET old_t = noms_old , new_t = noms_new
> FROM (SELECT array_agg (key) AS noms_old, array_agg (value) AS noms_new
>       FROM (
>                 SELECT (jsonb_each_text (j)).* FROM remplacement
> ```

```
        ) j1
       ) j2 ;
```

On vérifie le contenu :

```
SELECT * FROM remplacement \gx
```

Comparer la performance des deux fonctions suivantes :

```
\pset pager off
-- fonction en PL/perl
EXPLAIN (ANALYZE, BUFFERS)
SELECT  multi_replace (contenu, (SELECT j FROM remplacement))
FROM miserables ;

-- fonction en PL/pgSQL
EXPLAIN (ANALYZE, BUFFERS)
SELECT  multi_replace (contenu,
                    (SELECT old_t FROM remplacement),
                   (SELECT new_t FROM remplacement) )
FROM miserables ;
```

```
\pset pager off

EXPLAIN (ANALYZE, BUFFERS)
SELECT  multi_replace (contenu, (SELECT j FROM remplacement))
FROM miserables ;

EXPLAIN (ANALYZE, BUFFERS)
SELECT  multi_replace (contenu,
                     (SELECT old_t FROM remplacement),
                     (SELECT new_t FROM remplacement) )
FROM miserables ;
```

Selon les performances de la machine, les résultats peuvent varier, mais la première (en PL/perl) est probablement plus rapide. La fonction en PL/perl montre son intérêt quand il y a beaucoup de substitutions.

1.11.6 hll

- Installer l'extension `hll` dans la base de données de test :
- le paquet est `hll_14` ou `postgresql-14-hll` (ou l'équivalent pour les autres numéros de versions) selon la distribution ;
- l'extension se nomme `hll` ;
- elle nécessite d'être préalablement déclarée dans `shared_preload_libraries`.

Sur Rocky Linux et autres dérivés Red Hat :

```
 # dnf install hll_14
```

Sur Debian et dérivés :

```
 # apt install postgresql-14-hll
```

Modifier `postgresql.conf` ainsi afin que la bibliothèque soit préchargée dès le démarrage du serveur :

```
shared_preload_libraries = 'hll'
```

Redémarrer PostgreSQL.

Installer l'extension dans la base :

```
# CREATE EXTENSION hll ;
```

- Créer un jeu de données simulant des voyages en transport en commun, par passager selon la date :

```
CREATE TABLE voyages
(voyage_id      bigint  GENERATED ALWAYS AS IDENTITY,
 passager_id    text,
 d              date
) ;

INSERT INTO voyages (passager_id, d)
SELECT sem+mod(i, sem+1) ||'-'|| mod(i,77777) AS passager_id, d
FROM generate_series (0,51) sem,
      LATERAL
        (SELECT i,
         '2019-01-01'::date + sem * interval '7 days' + i * interval '2s' AS d
       FROM generate_series (1,
         (case when sem in (31,32,33) then 0 else 22 end +abs(30-sem))*5000 ) i
       ) j
;
```

Cette table de 9 millions de voyages étalés de janvier à décembre 2019 pèse 442 Mo.

- Activer l'affichage du temps (`timing`).
- Désactiver JIT et le parallélisme.
- Passer la mémoire de tri à 1 Go.
- Précharger la table dans le cache de PostgreSQL.

```
\timing on
SET max_parallel_workers_per_gather TO 0 ;
SET jit TO off ;
SET work_mem TO '1GB';

CREATE EXTENSION pg_prewarm ;

SELECT pg_prewarm('voyages') ;
```

- Calculer, par mois, le nombre exact de voyages et de passagers **distincts**.
- Dans le plan de la requête, chercher où est perdu le temps.

```
SELECT
    date_trunc('month', d)::date AS mois,
    COUNT(*) AS nb_voyages,
    count(DISTINCT passager_id) AS nb_d_passagers_mois
FROM voyages
GROUP BY 1 ORDER BY 1 ;
```

```
    mois    | nb_voyages | nb_d_passagers_mois
------------+------------+---------------------
 2019-01-01 |    1139599 |              573853
 2019-02-01 |     930000 |              560840
 2019-03-01 |     920401 |              670993
 2019-04-01 |     793199 |              613376
 2019-05-01 |     781801 |              655970
 2019-06-01 |     570000 |              513439
 2019-07-01 |     576399 |              518478
 2019-08-01 |     183601 |              179913
 2019-09-01 |     570000 |              527994
 2019-10-01 |     779599 |              639944
 2019-11-01 |     795401 |              728657
 2019-12-01 |     830000 |              767419
(12 lignes)

Durée : 57301,383 ms (00:57,301)
```

Le plan de cette même requête avec `EXPLAIN (ANALYZE, BUFFERS)` est :

```
                                QUERY PLAN
---------------------------------------------------------------------------
 GroupAggregate       (cost=1235334.73..1324038.21 rows=230 width=20)
                 (actual time=11868.776..60192.466 rows=12 loops=1)
   Group Key: ((date_trunc('month'::text, (d)::timestamp with time zone))::date)
   Buffers: shared hit=56497
   ->  Sort     (cost=1235334.73..1257509.59 rows=8869946 width=12)
             (actual time=5383.305..5944.522 rows=8870000 loops=1)
         Sort Key:
             ((date_trunc('month'::text, (d)::timestamp with time zone))::date)
```

```
         Sort Method: quicksort  Memory: 765307kB
         Buffers: shared hit=56497
         ->  Seq Scan on voyages  (cost=0.00..211721.06 rows=8869946 width=12)
                         (actual time=0.055..3714.690 rows=8870000 loops=1)
               Buffers: shared hit=56497
 Planning Time: 0.439 ms
 Execution Time: 60278.583 ms
```

Le plan est visible sur https://explain.dalibo.com/plan/Hj (pour PostgreSQL 14). Il suppose que `shared_buffers` est assez grand pour que tous tous les accès se fassent en mémoire (*shared hits*). Le `work_mem` élevé permet que le tri des 765 Mo soit aussi en mémoire. Le cas est donc idéal. L'essentiel du temps est perdu en tri.

Pour donner une idée de la lourdeur d'un `COUNT(DISTINCT)` : un décompte non distinct (qui revient à calculer le nombre de voyages) prend sur la même machine 5 secondes, même moins si le parallélisme est utilisé, mais ce qu'un `COUNT (DISTINCT)` ne permet pas.

> – Calculer, pour l'année, le nombre exact de voyages et de passagers **distincts**.

```
SELECT COUNT(*) AS nb_voyages,
       COUNT(DISTINCT passager_id) AS nb_d_passagers_annee
FROM voyages;
```

```
 nb_voyages | nb_d_passagers_annee
------------+----------------------
    8870000 |              4731210

Durée : 60396,816 ms (01:00,397)
```

On a donc plusieurs millions de voyages chaque mois, répartis sur quelques centaines de milliers de passagers mensuels, qui ne totalisent que 4,7 millions de personnes distinctes. Il y a donc un fort turn-over tout au long de l'année sans que ce soit un renouvellement complet d'un mois sur l'autre.

> – Recompter les passagers dans les deux cas en remplaçant le `COUNT(DISTINCT)` par cette expression : `hll_cardinality(hll_add_agg(hll_hash_text(passager_id)))::int`

Les ID des passagers sont hachés, agrégés, et le calcul de cardinalité se fait sur l'ensemble complet.

```
SELECT
    date_trunc('month', d)::date AS mois,
    hll_cardinality(hll_add_agg(hll_hash_text(passager_id)))::int
     AS nb_d_passagers_mois
FROM voyages
GROUP BY 1 ORDER BY 1 ;
```

```
    mois    | nb_d_passagers_mois
------------+---------------------
 2019-01-01 |              563372
 2019-02-01 |              553182
 2019-03-01 |              683411
 2019-04-01 |              637927
 2019-05-01 |              670292
```

```
 2019-06-01 |              505151
 2019-07-01 |              517140
 2019-08-01 |              178431
 2019-09-01 |              527655
 2019-10-01 |              632810
 2019-11-01 |              708418
 2019-12-01 |              766208
(12 lignes)

Durée : 4556,646 ms (00:04,557)
```

L'accélération est foudroyante (facteur 10 ici). Les chiffres sont différents, mais très proches (écart souvent inférieur à 1 %, au maximum 2,8 %).

Le plan indique un parcours de table et un agrégat par hachage :

```
 Sort (actual time=5374.025..5374.025 rows=12 loops=1)
   Sort Key: ((date_trunc('month'::text, (d)::timestamp with time zone))::date)
   Sort Method: quicksort  Memory: 25kB
   Buffers: shared hit=56497
   ->  HashAggregate (actual time=5373.793..5374.009 rows=12 loops=1)
         Group Key: (date_trunc('month'::text, (d)::timestamp with time zone))::date
         Buffers: shared hit=56497
         ->  Seq Scan on voyages (actual time=0.020..3633.757 rows=8870000 loops=1)
               Buffers: shared hit=56497
 Planning Time: 0.122 ms
 Execution Time: 5374.062 ms
```

Pour l'année, on a un résultat similaire :

```
SELECT
    hll_cardinality (hll_add_agg (hll_hash_text (passager_id)))::int
     AS nb_d_passagers_annee
FROM voyages;

 nb_d_passagers_annee
----------------------
              4645096
(1 ligne)
Durée : 1461,006 ms (00:01,461)
```

L'écart est de 1,8 % pour une durée réduite d'un facteur 40. Cet écart est-il acceptable pour les besoins applicatifs ? C'est un choix fonctionnel. On peut d'ailleurs agir dessus.

- Réexécuter les requêtes après modification du paramétrage de `hll` :

```
SELECT hll_set_defaults(17, 5, -1, 0);
```

Les défauts sont : `log2m=11, regwidth=5, expthresh=-1, sparseon=1`.

La requête mensuelle dure à peine plus longtemps (environ 6 s sur la machine de test) pour un écart par rapport à la réalité de l'ordre de 0,02 à 0,6 %.

La requête sur l'année dure environ le même temps pour seulement 0,2 % d'erreur cette fois :

```
 nb_d_passagers_annee
----------------------
               4741645
(1 ligne)
Durée : 1122,149 ms (00:01,122)
```

Selon les cas et après des tests soigneux, on testera donc l'intérêt de modifier ces paramètres tels que décrits sur le site du projet : https://github.com/citusdata/postgresql-hll

- Créer une table d'agrégat par mois avec un champ d'agrégat `hll` et la remplir.

```
CREATE TABLE voyages_mois
(mois              date,
 nb_exact_passagers_mois int,
 passagers_hll     hll
)  ;

INSERT INTO voyages_mois
SELECT
    date_trunc('month', d)::date,
    COUNT(DISTINCT passager_id),
    hll_add_agg (hll_hash_text (passager_id))
FROM voyages
GROUP BY 1;
```

Cette table d'agrégat n'a que 12 lignes mais contient un champ de type `hll` agrégeant les `passager_id` de ce mois. Sa taille n'est que d'1 Mo :

```
hll=# \d+
                                 Liste des relations
 Schéma |          Nom           |   Type   | Propriétaire |   Taille   | …
--------+------------------------+----------+--------------+------------+--
 public | voyages                | table    | postgres     | 442 MB     |
 public | voyages_mois           | table    | postgres     | 1072 kB    |
 public | voyages_voyage_id_seq | séquence | postgres     | 8192 bytes |
```

À partir de cette table d'agrégat :

- calculer le nombre moyen mensuel de passagers distincts,
- recalculer le nombre de passagers distincts sur l'année à partir de cette table d'agrégat.

La fonction pour agréger des champs de type `hll` est `hll_union_agg`. La requête est donc :

```
SELECT  AVG(nb_exact_passagers_mois)::int AS passagers_mois_moyen,
        hll_cardinality(hll_union_agg(passagers_hll))::int AS nb_passagers_annuels
        FROM voyages_mois ;

 passagers_mois_moyen | nb_passagers_annuels
----------------------+----------------------
               579240 |              4741645
(1 ligne)
Temps : 17,391 ms
```

L'extension HyperLogLog permet donc d'utiliser des tables d'agrégat pour un `COUNT(DISTINCT)`. De manière presque instantanée, on retrouve la même estimation presque parfaite que ci-dessus. Il aurait été impossible de la recalculer depuis la table d'agrégat (au contraire de la moyenne par mois, ou d'une somme du nombre de voyages).

> Avec une fonction de fenêtrage sur `hll_union_agg`, calculer une moyenne glissante sur 3 mois du nombre de passagers distincts.

```
SELECT  mois,
        nb_exact_passagers_mois,
        CASE WHEN ROW_NUMBER() OVER() > 2 THEN
           hll_cardinality(hll_union_agg(passagers_hll)
                     OVER (ORDER BY mois ASC ROWS 2 PRECEDING) )::bigint
         ELSE null END  AS nb_d_passagers_3_mois_glissants
FROM voyages_mois
;
```

```
    mois    | nb_exact_passagers_mois | nb_d_passagers_3_mois_glissants
------------+-------------------------+---------------------------------
 2019-01-01 |                  573853 |
 2019-02-01 |                  560840 |
 2019-03-01 |                  670993 |                         1463112
 2019-04-01 |                  613376 |                         1439444
 2019-05-01 |                  655970 |                         1485437
 2019-06-01 |                  513439 |                         1368534
 2019-07-01 |                  518478 |                         1333368
 2019-08-01 |                  179913 |                         1018605
 2019-09-01 |                  527994 |                         1057278
 2019-10-01 |                  639944 |                         1165308
 2019-11-01 |                  728657 |                         1579378
 2019-12-01 |                  767419 |                         1741934
(12 lignes)

Temps : 78,522 ms
```

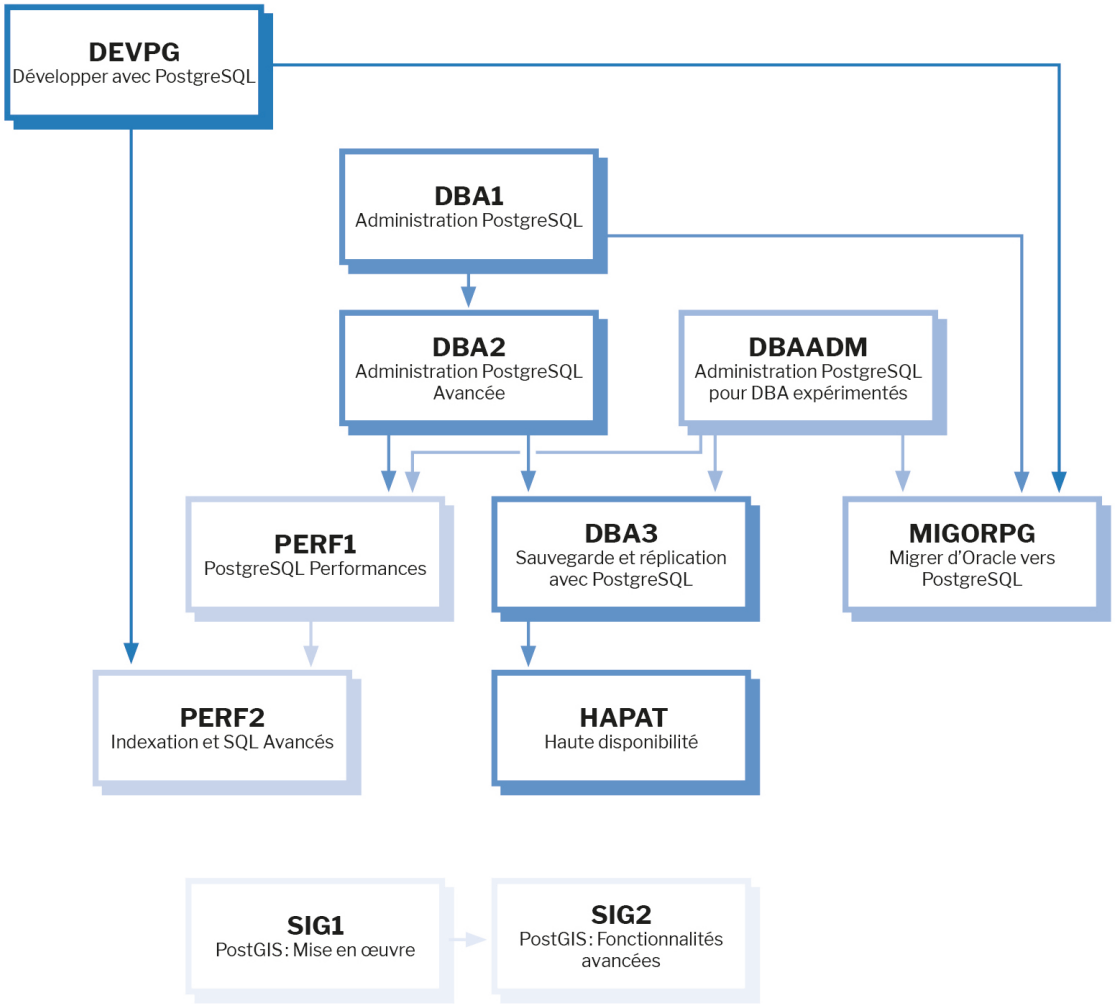
Un `COUNT(DISTINCT)` avec une fonction de fenêtrage n'est en pratique pas faisable, en tout cas pas aussi aisément, et bien plus lentement.

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.