Formation PERF2

Indexation & SQL Avancé

23.09

Table des matières

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

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.

[1] mailto:formation@dalibo.com

Licence Creative Commons CC-BY-NC-SA

Marques déposées

Sur ce document

Formation	Formation PERF2
Titre	Indexation & SQL Avancé
Révision	23.09
ISBN	N/A
PDF	https://dali.bo/perf2_pdf

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode
[3] https://www.postgresql.org/about/policies/trademarks/

EPUB	https://dali.bo/perf2_epub
HTML	https://dali.bo/perf2_html
Slides	https://dali.bo/perf2_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

1/ Techniques d’indexation

Photo de Maksym Kaharlytskyi[1], Unsplash licence

[1] https://unsplash.com/@qwitka

1.1 INTRODUCTION

- Qu'est-ce qu'un index ?
- Comment indexer une base ?
- Les différents types d'index

1.1.1 Objectifs

- Comprendre ce qu'est un index
- Maîtriser le processus de création d'index
- Connaître les différents types d'index et leurs cas d'usages

1.1.2 Introduction aux index

- Uniquement destinés à l'optimisation
- À gérer d'abord par le développeur
    - **Markus Winand** : *SQL Performance Explained*

Les index ne sont pas des objets qui font partie de la théorie relationnelle. Ils sont des objets physiques qui permettent d'accélérer l'accès aux données. Et comme ils ne sont que des moyens d'optimisation des accès, les index ne font pas non plus partie de la norme SQL. C'est d'ailleurs pour cette raison que la syntaxe de création d'index est si différente d'une base de données à une autre.

La création des index est à la charge du développeur ou du DBA, leur création n'est pas automatique, sauf exception.

Pour Markus Winand, c'est d'abord au développeur de poser les index, car c'est lui qui sait comment ses données sont utilisées. Un DBA d'exploitation n'a pas cette connaissance, mais il connaît généralement mieux les différents types d'index et leurs subtilités, et voit comment les requêtes réagissent en production. Développeur et DBA sont complémentaires dans l'analyse d'un problème de performance.

Le site de Markus Winand, Use the index, Luke[2], propose une version en ligne de son livre *SQL Performance Explained*, centré sur les index B-tree (les plus courants). Une version française est par ailleurs disponible sous le titre *SQL : au cœur des performances*.

1.1.3 Utilité d'un index

- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire tout récupérer dans l'index (*Index Only Scan*)
- Un index facilite :
  - certains tris
  - certains agrégats
- Obligatoires et automatique pour clés primaires & unicité
  - conseillé pour clés étrangères (FK)

Les index ne changent pas le résultat d'une requête, mais l'accélèrent. L'index permet de pointer l'endroit de la table où se trouve une donnée, pour y accéder directement. Parfois c'est toute une plage de l'index, voire sa totalité, qui sera lue, ce qui est généralement plus rapide que lire toute la table.

Le cas le plus favorable est l'*Index Only Scan* : toutes les données nécessaires sont contenues dans l'index, lui seul sera lu et PostgreSQL ne lira pas la table elle-même.

PostgreSQL propose différentes formes d'index :

- index classique sur une seule colonne d'une table ;
- index composite sur plusieurs colonnes d'une table ;
- index partiel, en restreignant les données indexées avec une clause `WHERE` ;
- index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table ;
- index couvrants, contenant plus de champs que nécessaire au filtrage, pour ne pas avoir besoin de lire la table, et obtenir un *Index Only Scan*.

La création des index est à la charge du développeur. Seules exceptions : ceux créés automatiquement quand on déclare des contraintes de clé primaire ou d'unicité. La création est alors automatique.

Les contraintes de clé étrangère imposent qu'il existe déjà une clé primaire sur la table pointée, mais ne crée pas d'index sur la table portant la clé.

[2] https://use-the-index-luke.com

1.1.4 Index et lectures

Un index améliore les SELECT

- Sans index :

```
=# SELECT * FROM test WHERE id = 10000;
Temps : 1760,017 ms
```

- Avec index :

```
=# CREATE INDEX idx_test_id ON test (id);
=# SELECT * FROM test WHERE id = 10000;
Temps : 27,711 ms
```

L'index est une structure de données qui permet d'accéder rapidement à l'information recherchée. À l'image de l'index d'un livre, pour retrouver un thème rapidement, on préférera utiliser l'index du livre plutôt que lire l'intégralité du livre jusqu'à trouver le passage qui nous intéresse. Dans une base de données, l'index a un rôle équivalent. Plutôt que de lire une table dans son intégralité, la base de données utilisera l'index pour ne lire qu'une faible portion de la table pour retrouver les données recherchées.

Pour la requête d'exemple (avec une table de 20 millions de lignes), on remarque que l'optimiseur n'utilise pas le même chemin selon que l'index soit présent ou non. Sans index, PostgreSQL réalise un parcours séquentiel de la table :

```
postgres=# EXPLAIN SELECT * FROM test WHERE id = 10000;

                              QUERY PLAN
-----------------------------------------------------------------------
 Gather  (cost=1000.00..193661.66 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on test  (cost=0.00..192661.56 rows=1 width=4)
         Filter: (id = 10000)
```

Lorsqu'il est présent, PostgreSQL l'utilise car l'optimiseur estime que son parcours ne récupérera qu'une seule ligne sur les 10 millions que compte la table :

```
postgres=# EXPLAIN SELECT * FROM test WHERE id = 10000;

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Only Scan using idx_test_id on test  (cost=0.44..8.46 rows=1 width=4)
   Index Cond: (id = 10000)
```

Mais l'index n'accélère pas seulement la simple lecture de données, il permet également d'accélérer les tris et les agrégations, comme le montre l'exemple suivant sur un tri :

```
postgres=# EXPLAIN SELECT id FROM test
           WHERE id BETWEEN 1000 AND 1200 ORDER BY id DESC;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan Backward using idx_test_id on test
                                                (cost=0.44..12.26 rows=191 width=4)
   Index Cond: ((id >= 1000) AND (id <= 1200))
```

1.1.5 Index : inconvénients

- L'index n'est pas gratuit !
- Ralentit les écritures
  - maintenance
- Place disque
- Compromis à trouver

La présence d'un index ralentit les écritures sur une table. En effet, il faut non seulement ajouter ou modifier les données dans la table, mais il faut également maintenir le ou les index de cette table.

Les index dégradent surtout les temps de réponse des insertions. Les mises à jour et les suppressions (UPDATE et DELETE) tirent en général parti des index pour retrouver les lignes concernées par les modifications. Le coût de maintenance de l'index est secondaire par rapport au coût de l'accès aux données.

Soit une table `test2` telle que :

```
CREATE TABLE test2 (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    valeur INTEGER,
    commentaire TEXT
);
```

La table est chargée avec pour seul index présent celui sur la clé primaire :

```
=# INSERT INTO test2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 35253,228 ms (00:35,253)
```

Un index supplémentaire est créé sur une colonne de type entier :

```
=# CREATE INDEX idx_test2_valeur ON test2 (valeur);
=# INSERT INTO test2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 44410,775 ms (00:44,411)
```

Un index supplémentaire est encore créé, mais cette fois sur une colonne de type texte :

```
=# CREATE INDEX idx_test2_commentaire ON test2 (commentaire);
=# INSERT INTO test2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 207075,335 ms (03:27,075)
```

On peut comparer ces temps à l'insertion dans une table similaire dépourvue d'index :

```
=# CREATE TABLE test3 AS SELECT * FROM test2;
=# INSERT INTO test3 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 14758,503 ms (00:14,759)
```

La table `test2` a été vidée préalablement pour chaque test.

Enfin, la place disque utilisée par ces index n'est pas négligeable :

```
# \di+ *test2*
                                 Liste des relations
 Schéma |          Nom           | Type  | Propriétaire | Table | Taille |  …
--------+------------------------+-------+--------------+-------+--------+-
 public | idx_test2_commentaire  | index | postgres     | test2 | 387 MB |
 public | idx_test2_valeur       | index | postgres     | test2 | 214 MB |
 public | test2_pkey             | index | postgres     | test2 | 214 MB |
```

```
# SELECT pg_size_pretty(pg_relation_size('test2')),
         pg_size_pretty(pg_indexes_size('test2')) ;
```

```
 pg_size_pretty | pg_size_pretty
----------------+----------------
 574 MB         | 816 MB
```

Pour ces raisons, on ne posera pas des index systématiquement avant de se demander s'ils seront utilisés. L'idéal est d'étudier les plans de ses requêtes et de chercher à optimiser.

1.1.6 Index : contraintes pratiques à la création

- Lourd…

```
CREATE INDEX ON matable ( macolonne ) ;              -- bloque les
 ↪  écritures !
CREATE INDEX CONCURRENTLY ON matable ( macolonne ) ;  -- ne bloque pas,
 ↪  peut échouer
```

- Si fragmentation :

```
REINDEX INDEX nomindex ;
REINDEX TABLE CONCURRENTLY nomtable ;
```

- Paramètres :
  - `maintenance_work_mem` (sinon : fichier temporaire !)
  - `max_parallel_maintenance_workers`

Création d'un index :

Bien sûr, la durée de création de l'index dépend fortement de la taille de la table. PostgreSQL va lire toutes les lignes et trier les valeurs rencontrées. Ce peut être lourd et impliquer la création de fichiers temporaires.

Si l'on utilise la syntaxe classique, toutes les écritures sur la table sont bloquées (mises en attente) pendant la durée de la création de l'index (verrou *ShareLock*). Les lectures restent possibles, mais cette contrainte est parfois rédhibitoire pour les grosses tables.

Clause CONCURRENTLY :

Ajouter le mot clé CONCURRENTLY permet de rendre la table accessible en écriture. Malheureusement, cela nécessite au minimum deux parcours de la table, et donc alourdit et ralentit la construction de l'index. Dans quelques cas défavorables, la création échoue et l'index existe mais devient invalide :

```
postgres=# \d tab
    Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
    Indexes:
    "idx" btree (col) INVALID
```

L'index est inutilisable et doit être supprimé et recréé, ou bien réindexé.

Pour les détails, voir la documentation officielle[3].

[3] https://docs.postgresql.fr/15/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY

De tels index invalides sont détectés avec cette requête, qui ne doit jamais rien ramener :

```
SELECT indexrelid::regclass AS index, indrelid::regclass AS table
FROM pg_index
WHERE indisvalid = false ;
```

Réindexation :

Comme les tables, les index sont soumis à la fragmentation. Celle-ci peut cependant monter assez haut sans grande conséquence pour les performances. De plus, le nettoyage des index est une des étapes des opérations de VACUUM[4].

Une reconstruction de l'index est automatique lors d'un `VACUUM FULL` de la table.

Certaines charges provoquent une fragmentation assez élevée, typiquement les tables gérant des files d'attente. Une réindexation reconstruit totalement l'index. Voici quelques variantes de l'ordre :

```
REINDEX INDEX pgbench_accounts_bid_idx ;  -- un seul index
REINDEX TABLE pgbench_accounts ;          -- tous les index de la table
REINDEX (VERBOSE) DATABASE pgbench ;      -- tous deux de la base, avec détails
```

À partir de la version 12, cet ordre accepte une clause `CONCURRENTLY` pour les mêmes raisons que le `CREATE INDEX` :

```
REINDEX (VERBOSE) INDEX CONCURRENTLY pgbench_accounts_bid_idx ;
```

Paramètres :

La rapidité de création d'un index dépend essentiellement de la mémoire accordée, définie dans `maintenance_work_mem`. Si elle ne suffit pas, le tri se fera dans des fichiers temporaires plus lents. Sur les serveurs modernes, le défaut de 64 Mo est ridicule, et on peut monter aisément à :

```
SET maintenance_work_mem = '2GB' ;
```

Attention de ne pas saturer la mémoire en cas de création simultanée de nombreux gros index (lors d'une restauration avec `pg_restore` notamment).

Si le serveur est bien doté en CPU, la parallélisation de la création d'index peut apporter un gain en temps appréciable. La valeur par défaut est :

```
SET max_parallel_maintenance_workers = 2 ;
```

et devrait même être baissée sur les plus petites configurations.

[4] https://dali.bo/m5_html#fonctionnement-de-vacuum

1.1.7 Types d'index dans PostgreSQL

- Défaut : B-tree classique (balancé)
- UNIQUE (préférer la contrainte)
- Mais aussi multicolonne, fonctionnel, partiel, couvrant
- Index spécialisés : hash, GiST, GIN, BRIN…

Par défaut un `CREATE INDEX` créera un index de type B-tree, de loin le plus courant. Il est stocké sous forme d'arbre balancé, avec de nombreux avantages :

- performances se dégradant peu avec la taille de l'arbre (les temps de recherche sont en O(log(n)), donc fonction du logarithme du nombre d'enregistrements dans l'index) ;
- excellente concurrence d'accès, avec très peu de contention entre processus qui insèrent simultanément.

Toutefois les B-tree ne permettent de répondre qu'à des questions très simples, portant sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre tout de même la majorité des cas.

Contrainte d'unicité et index :

Un index peut être déclaré `UNIQUE` pour provoquer une erreur en cas d'insertion de doublons. Mais on préférera généralement déclarer une *contrainte* d'unicité (notion fonctionnelle), qui techniquement, entraînera la création d'un index.

Par exemple, sur cette table `personne` :

```
$ CREATE TABLE personne (id int, nom text);

$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
```

on peut créer un index unique :

```
$ CREATE UNIQUE INDEX ON personne (id);

$ \d personne
                 Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "personne_id_idx" UNIQUE, btree (id)
```

La contrainte d'unicité est alors implicite. La suppression de l'index se fait sans bruit :

```
DROP INDEX personne_id_idx;
```

Définissons une contrainte d'unicité sur la colonne plutôt qu'un index :

```
ALTER TABLE personne ADD CONSTRAINT unique_id UNIQUE (id);
```

```
$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "unique_id" UNIQUE CONSTRAINT, btree (id)
```

Un index est également créé. La contrainte empêche sa suppression :

```
DROP INDEX unique_id ;
```

```
ERREUR:  n'a pas pu supprimer index unique_id car il est requis par contrainte
 unique_id sur table personne
ASTUCE : Vous pouvez supprimer contrainte unique_id sur table personne à la
 place.
```

Le principe est le même pour les clés primaires.

Indexation avancée :

Il faut aussi savoir que PostgreSQL permet de créer des index B-tree :

- sur plusieurs colonnes ;
- sur des résultats de fonction ;
- sur une partie des valeurs indexées ;
- intégrant des champs non indexés mais souvent récupérés avec les champs indexés (index couvrants).

D'autres types d'index que B-tree existent, destinés à certains types de données ou certains cas d'optimisation précis.

1.2 FONCTIONNEMENT D'UN INDEX

- Anatomie d'un index
- Les index « simples »
- Méthodologie
- Indexation avancée
- Outillage

1.2.1 Structure d'un index

- Analogie : index dans une publication scientifique
  - structure séparée, associant des clés (termes) à des localisations (pages)
  - même principe pour un index dans un SGBD
- Structure de données spécialisée, plusieurs types
- Existe en dehors de la table

Pour comprendre ce qu'est un index, l'index dans une publication scientifique au format papier offre une analogie simple.

Lorsque l'on recherche un terme particulier dans un ouvrage, il est possible de parcourir l'intégralité de l'ouvrage pour chercher les termes qui nous intéressent. Ceci prend énormément de temps, variable selon la taille de l'ouvrage. Ce type de recherche trouve son analogie sous la forme du parcours complet d'une table (*Seq Scan*).

Une deuxième méthode pour localiser ces différents termes consiste, si l'ouvrage en dispose, à utiliser l'index de celui-ci. Un tel index associe un terme à un ensemble de pages où celui-ci est présent. Ainsi, pour trouver le terme recherché, il est uniquement nécessaire de parcourir l'index (qui ne dépasse généralement pas quelques pages) à la recherche du terme, puis d'aller visiter les pages listées dans l'index pour extraire les informations nécessaires.

Dans un SGBD, le fonctionnement d'un index est très similaire à celui décrit ici. En effet, comme dans une publication, l'index est une structure de données à part, qui n'est pas strictement nécessaire à l'exploitation des informations, et qui est utilisée pour faciliter la recherche dans l'ensemble de données. Cette structure de données possède un coût de maintenance, dans les deux cas : toute modification des données peut entraîner des modifications afin de maintenir l'index à jour.

1.2.2 Un index n'est pas magique…

- Un index ne résout pas tout
- Importance de la conception du schéma de données
- Importance de l'écriture de requêtes SQL correctes

Bien souvent, la création d'index est vue comme le remède à tous les maux de performance subis par une application. Il ne faut pas perdre de vue que les facteurs principaux affectant les performances vont être liés à la conception du schéma de données, et à l'écriture des requêtes SQL.

Pour prendre un exemple caricatural, un schéma EAV (*Entity-Attribute-Value*, ou *entité-clé-valeur*) ne pourra jamais être performant, de part sa conception. Bien sûr, dans certains cas, une méthodologie pertinente d'indexation permettra d'améliorer un peu les performances, mais le problème réside là dans la conception même du schéma. Il est donc important dans cette phase de considérer la manière dont le modèle va influer sur les méthodes d'accès aux données, et les implications sur les performances.

De même, l'écriture des requêtes elles-mêmes conditionnera en grande partie les performances observées sur l'application. Par exemple, la mauvaise pratique (souvent mise en œuvre accidentellement via un ORM) dite du « N+1 » ne pourra être corrigée par une indexation correcte : celle-ci consiste à récupérer une collection d'enregistrement (une requête) puis d'effectuer une requête pour chaque enregistrement afin de récupérer les enregistrements liés (N requêtes). Dans ce type de cas, une jointure est bien plus performante. Ce type de comportement doit encore une fois être connu de l'équipe de développement, car il est plutôt difficile à détecter par une équipe d'exploitation.

De manière générale, avant d'envisager la création d'index supplémentaires, il convient de s'interroger sur les possibilités de réécriture des requêtes, voire du schéma.

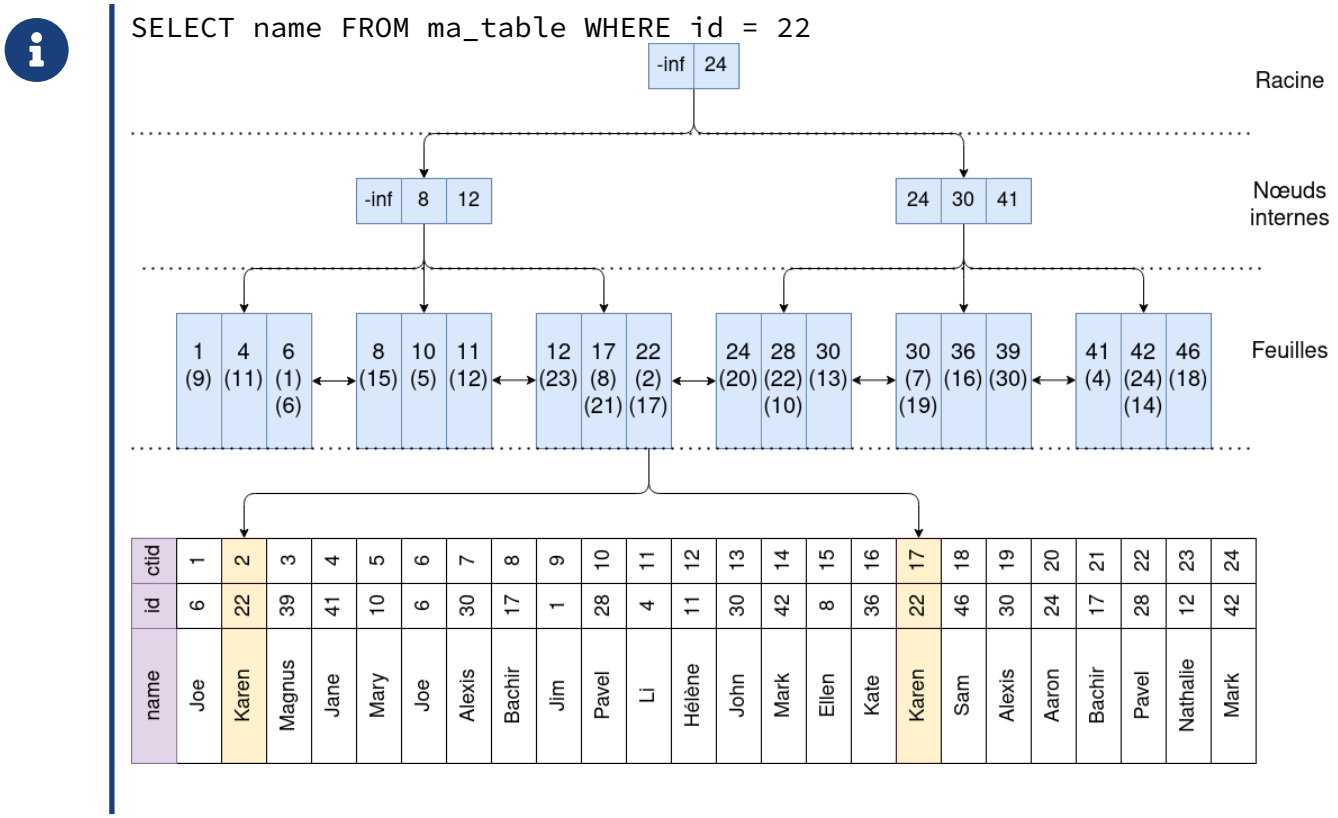
1.2.3 Index B-tree

- Type d'index le plus courant
  - et le plus simple
- Utilisable pour les contraintes d'unicité
- Supporte les opérateurs : <, <=, =, >=, >
- Supporte le tri
- Ne peut pas indexer des colonnes de plus de 2,6 ko

L'index B-tree est le plus simple conceptuellement parlant. Sans entrer dans les détails, un index B-tree est par définition équilibré : ainsi, quelle que soit la valeur recherchée, le coût est le même lors du parcours d'index. Ceci ne veut pas dire que toute requête impliquant l'index mettra le même temps ! En effet, si chaque clé n'est présente qu'une fois dans l'index, celle-ci peut être associée à une multitude de valeurs, qui devront alors être cherchées dans la table.

L'algorithme utilisé par PostgreSQL pour ce type d'index suppose que chaque page peut contenir au moins trois valeurs. Par conséquent, chaque valeur ne peut excéder un peu moins d'1/3 de bloc, soit environ 2,6 ko. La valeur en question correspond donc à la totalité des données de toutes les colonnes de l'index pour une seule ligne. Si l'on tente de créer ou maintenir un index sur une table ne satisfaisant pas ces prérequis, une erreur sera reportée, et la création de l'index (ou l'insertion/mise à jour de la ligne) échouera. Si un index de type B-tree est tout de même nécessaire sur les colonnes en question, il est possible de créer un index fonctionnel sur une fonction de hachage des valeurs. Dans un tel cas, seul l'opérateur = pourra bénéficier d'un parcours d'index.

1.2.4 Concrètement…



Ce schéma présente une vue simplifiée d'une table (en blanc, avec ses champs `id` et `name`) et d'un index B-tree sur `id` (en bleu), tel que le créerait :

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

Un index B-tree peut contenir trois types de nœuds :

- la racine : elle est unique c'est la base de l'arbre ;
- des nœuds internes : il peut y en avoir plusieurs niveaux ;
- des feuilles : elles contiennent :
  - les valeurs indexées (triées !) ;
  - les valeurs incluses (si applicable) ;
  - les positions physiques (`ctid`), ici entre parenthèses et sous forme abrégée, car la forme réelle est (numéro de bloc, position de la ligne dans le bloc) ;
  - l'adresse de la feuille précédente et de la feuille suivante.

La racine et les nœuds internes contiennent des enregistrements qui décrivent la valeur minimale de chaque bloc du niveau inférieur et leur adresse (`ctid`).

Lors de la création de l'index, il ne contient qu'une feuille. Lorsque cette feuille se remplit, elle se divise en deux et un nœud racine est créé au-dessus. Les feuilles se remplissent ensuite progressivement et

se séparent en deux quand elles sont pleines. Ce processus remplit progressivement la racine. Lorsque la racine est pleine, elle se divise en deux nœuds internes, et une nouvelle racine est crée au-dessus. Ce processus permet de garder un arbre équilibré.

Recherchons le résultat de :

```
SELECT name FROM ma_table WHERE id = 22
```

en passant par l'index.

- En parcourant la racine, on cherche un enregistrement dont la valeur est strictement supérieure à la valeur que l'on recherche. Ici, 22 est plus petit que 24 : on explore donc le nœud de gauche.
- Ce nœud référence trois nœuds inférieurs (ici des feuilles). On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud : pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre. Ici, 22 est plus grand que 12, on explore donc le nœud de droite au niveau inférieur.
- Un arbre B-tree peut bien évidemment avoir une profondeur plus grande, auquel cas l'étape précédente est répétée.
- Une fois arrivé sur une feuille, il suffit de la parcourir pour récupérer l'ensemble des positions physiques des lignes correspondants au critère. Ici, la feuille nous indique qu'à la valeur 22 correspondent deux lignes aux positions 2 et 17. Lorsque la valeur recherchée est supérieure ou égale à la plus grande valeur du bloc, PostgreSQL va également lire le bloc suivant. Ce cas de figure peut se produire si PostgreSQL a divisé une feuille en deux avant ou même pendant la recherche que nous exécutons. Ce serait par exemple le cas si on cherchait la valeur 30.
- Pour trouver les valeurs de `name`, il faut aller chercher dans la table même les lignes aux positions trouvées dans l'index. D'autre part, les informations de visibilité des lignes doivent aussi être trouvées dans la table. (Il existe des cas où la recherche peut éviter cette dernière étape : ce sont les *Index Only Scan*.)

Même en parcourant les deux structures de données, si la valeur recherchée représente une assez petite fraction des lignes totales, le nombre d'accès disques sera donc fortement réduit. En revanche, au lieu d'effectuer des accès séquentiels (pour lesquels les disques durs classiques sont relativement performants), il faudra effectuer des accès aléatoires, en *sautant* d'une position sur le disque à une autre. Le choix est fait par l'optimiseur.

Supposons désormais que nous souhaitions exécuter une requête sans filtre, mais exigeant un tri, du type :

```
SELECT id FROM ma_table ORDER BY id ;
```

L'index peut nous aider à répondre à cette requête. En effet, toutes les feuilles sont liées entre elles, et permettent ainsi un parcours ordonné. Il nous suffit donc de localiser la première feuille (la plus à gauche), et pour chaque clé, récupérer les lignes correspondantes. Une fois les clés de la feuille traitées, il suffit de suivre le pointeur vers la feuille suivante et de recommencer.

L'alternative consisterait à parcourir l'ensemble de la table, et trier toutes les lignes afin de les obtenir dans le bon ordre. Un tel tri peut être très coûteux, en mémoire comme en temps CPU. D'ailleurs, de tels tris débordent très souvent sur disque (via des fichiers temporaires) afin de ne pas garder l'intégralité des données en mémoire.

Pour les requêtes utilisant des opérateurs d'inégalité, on voit bien comment l'index peut là aussi être utilisé. Par exemple, pour la requête suivante :

```
SELECT * FROM ma_table WHERE id <= 10 AND id >= 4 ;
```

Il suffit d'utiliser la propriété de tri de l'index pour parcourir les feuilles, en partant de la borne inférieure, jusqu'à la borne supérieure.

Dernière remarque : ce schéma ne montre qu'une entrée d'index pour 22, bien qu'il pointe vers deux lignes. En fait, il y avait bien deux entrées pour 22 avant PostgreSQL 13. Depuis cette version, PostgreSQL sait dédupliquer les entrées.

1.2.5 Index multicolonnes

- Possibilité d'indexer plusieurs colonnes :

```
CREATE INDEX ON ma_table (id, name) ;
```

- Ordre des colonnes **primordial**
  - accès direct aux premières colonnes de l'index
  - pour les autres, PostgreSQL lira tout l'index ou ignorera l'index

Il est possible de créer un index sur plusieurs colonnes. Il faut néanmoins être conscient des requêtes supportées par un tel index. Admettons que l'on crée une table d'un million de lignes avec un index sur trois champs :

```
CREATE TABLE t1 (c1 int, c2 int, c3 int, c4 text);

INSERT INTO t1 (c1, c2, c3, c4)
SELECT i*10,j*5,k*20, 'text'||i||j||k
FROM generate_series (1,100) i
CROSS JOIN generate_series(1,100) j
CROSS JOIN generate_series(1,100) k ;

CREATE INDEX ON t1 (c1, c2, c3) ;

VACUUM ANALYZE t1 ;

-- Figer des paramètres pour l'exemple
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

L'index est optimal pour répondre aux requêtes portant sur les premières colonnes de l'index :

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 1000 and c2=500 and c3=2000 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..8.45 rows=1 width=22)
   Index Cond: ((c1 = 1000) AND (c2 = 500) AND (c3 = 2000))
```

Et encore plus quand l'index permet de répondre intégralement au contenu de la requête :

```
EXPLAIN SELECT c1,c2,c3 FROM t1 WHERE c1 = 1000 and c2=500 ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..6.33 rows=95 width=12)
   Index Cond: ((c1 = 1000) AND (c2 = 500))
```

Mais si les premières colonnes de l'index ne sont pas spécifiées, alors l'index devra être parcouru en grande partie.

Cela reste plus intéressant que parcourir toute la table, surtout si l'index est petit et contient toutes les données du SELECT. Mais le comportement dépend alors de nombreux paramètres, comme les statistiques, les estimations du nombre de lignes ramenées et les valeurs relatives de `seq_page_cost` et `random_page_cost` :

```
SET random_page_cost TO 0.1 ; SET seq_page_cost TO 0.1 ;  -- SSD

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (...) (...)
   Index Cond: (c3 = 2000)
   Buffers: shared hit=3899
 Planning:
   Buffers: shared hit=15
 Planning Time: 0.218 ms
 Execution Time: 67.081 ms
```

Noter que tout l'index a été lu.

Mais pour limiter les aller-retours entre index et table, PostgreSQL peut aussi décider d'ignorer l'index et de parcourir directement la table :

```
SET random_page_cost TO 4 ; SET seq_page_cost TO 1 ;  -- défaut (disque mécanique)

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                QUERY PLAN
-------------------------------------------------------------------------
Seq Scan on t1  (cost=0.00..18871.00 rows=9600 width=22) (...)
   Filter: (c3 = 2000)
   Rows Removed by Filter: 990000
   Buffers: shared hit=6371
 Planning Time: 0.178 ms
 Execution Time: 114.572 ms
```

Concernant les *range scans* (requêtes impliquant des opérateurs d'inégalité, tels que <, <=, >=, >), celles-ci pourront être satisfaites par l'index de manière quasi optimale si les opérateurs d'inégalité

sont appliqués sur la dernière colonne requêtée, et de manière sub-optimale s'ils portent sur les premières colonnes.

Cet index pourra être utilisé pour répondre aux requêtes suivantes de manière optimale :

```
SELECT * FROM t1 WHERE c1 = 20 ;
SELECT * FROM t1 WHERE c1 = 20 AND c2 = 50 AND c3 = 400 ;
SELECT * FROM t1 WHERE c1 = 10 AND c2 <= 4 ;
```

Il pourra aussi être utilisé, mais de manière bien moins efficace, pour les requêtes suivantes, qui bénéficieraient d'un index sur un ordre alternatif des colonnes :

```
SELECT * FROM t1 WHERE c1 = 100 AND c2 >= 80 AND c3 = 40 ;
SELECT * FROM t1 WHERE c1 < 100 AND c2 = 100 ;
```

Le plan de cette dernière requête est :

```
 Bitmap Heap Scan on t1  (cost=2275.98..4777.17 rows=919 width=22) (...)
   Recheck Cond: ((c1 < 100) AND (c2 = 100))
   Heap Blocks: exact=609
   Buffers: shared hit=956
   ->  Bitmap Index Scan on t1_c1_c2_c3_idx  (cost=0.00..2275.76 rows=919 width=0)
↪   (...)
         Index Cond: ((c1 < 100) AND (c2 = 100))
         Buffers: shared hit=347
 Planning Time: 0.227 ms
 Execution Time: 15.596 ms
```

Les index multicolonnes peuvent aussi être utilisés pour le tri comme dans les exemples suivants. Il n'y a pas besoin de trier (ce peut être très coûteux) puisque les données de l'index sont triées. Ici le cas est optimal puisque l'index contient toutes les données nécessaires :

```
SELECT * FROM t1 ORDER BY c1 ;
SELECT * FROM t1 ORDER BY c1, c2 ;
SELECT * FROM t1 ORDER BY c1, c2, c3 ;
```

Le plan de cette dernière requête est :

```
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..55893.66 rows=1000000 width=22)
↪   (...)
   Buffers: shared hit=1003834
 Planning Time: 0.282 ms
 Execution Time: 425.520 ms
```

Il est donc nécessaire d'avoir une bonne connaissance de l'application (ou de passer du temps à observer les requêtes consommatrices) pour déterminer comment créer des index multicolonnes pertinents pour un nombre maximum de requêtes.

1.3 MÉTHODOLOGIE DE CRÉATION D'INDEX

- On indexe pour une requête, ou idéalement une collection de requêtes
- On n'indexe pas « une table »

La première chose à garder en tête est que l'on indexe pas le schéma de données, c'est-à-dire les tables, mais en fonction de la charge de travail supportée par la base, c'est-à-dire les requêtes. En effet, comme nous l'avons vu précédemment, tout index superflu a un coût global pour la base de données, notamment pour les opérations DML.

1.3.1 L'index ? Quel index ?

- Identifier les requêtes nécessitant un index
- Créer les index permettant de répondre à ces requêtes
- Valider le fonctionnement, en rejouant la requête avec :

```
EXPLAIN (ANALYZE, BUFFERS)
```

La méthodologie elle-même est assez simple. Selon le principe qu'un index sert à une (ou des) requête(s), la première chose à faire consiste à identifier celles-ci. L'équipe de développement est dans une position idéale pour réaliser ce travail : elle seule peut connaître le fonctionnement global de l'application, et donc les colonnes qui vont être utilisées, ensemble ou non, comme cible de filtres ou de tris. Au delà de la connaissance de l'application, il est possible d'utiliser des outils tels que pgBadger, pg_stat_statements et PoWA pour identifier les requêtes particulièrement consommatrices, et qui pourraient donc potentiellement nécessiter un index. Ces outils seront présentés plus loin dans cette formation.

Une fois les requêtes identifiées, il est nécessaire de trouver les index permettant d'améliorer celles-ci. Ils peuvent être utilisés pour les opérations de filtrage (clause WHERE), de tri (clauses ORDER BY, GROUP BY) ou de jointures. Idéalement, l'étude portera sur l'ensemble des requêtes, afin notamment de pouvoir décider d'index multi-colonnes pertinents pour le plus grand nombre de requêtes, et éviter ainsi de créer des index redondants.

1.3.2 Index et clés étrangères

- Indexation des colonnes faisant référence à une autre
- Performances des DML
- Performances des jointures

De manière générale, l'ensemble des colonnes étant la source d'une clé étrangère devraient être indexées, et ce pour deux raisons.

La première concerne les jointures. Généralement, lorsque deux tables sont liées par des clés étrangères, il existe au moins certaines requêtes dans l'application joignant ces tables. La colonne « cible » de la clé étrangère est nécessairement indexée, c'est un prérequis dû à la contrainte unique nécessaire à celle-ci. Il est donc possible de la parcourir de manière triée.

La colonne source devrait être indexée elle aussi : en effet, il est alors possible de la parcourir de manière ordonnée, et donc de réaliser la jointure selon l'algorithme *Merge Join* (comme vu lors du module sur les plans d'exécution[5]), et donc d'être beaucoup plus rapide. Un tel index accélérera de la même manière les *Nested Loop*, en permettant de parcourir l'index une fois par ligne de la relation externe au lieu de parcourir l'intégralité de la table.

De la même manière, pour les DML sur la table cible, cet index sera d'une grande aide : pour chaque ligne modifiée ou supprimée, il convient de vérifier, soit pour interdire soit pour « cascader » la modification, la présence de lignes faisant référence à celle touchée.

S'il n'y a qu'une règle à suivre aveuglément ou presque, c'est bien celle-ci : les colonnes faisant partie d'une clé étrangère doivent être indexées !

Deux exceptions : les champs ayant une cardinalité très faible et homogène (par exemple, un champ homme/femme dans une population équilibrée) ; et ceux dont on constate l'inutilité après un certain temps, par des valeurs à zéro dans `pg_stat_user_indexes`.

[5] https://dali.bo/j0_html

1.4 INDEX INUTILISÉ

- C'est souvent tout à fait normal
- Utiliser l'index est-il rentable ?
- La requête est-elle compatible ?
- Bug de l'optimiseur : rare

C'est l'optimiseur SQL qui choisit si un index doit ou non être utilisé. Il est tout à fait possible que PostgreSQL décide qu'utiliser un index donné n'en vaut pas la peine par rapport à d'autres chemins. Il faut aussi savoir identifier les cas où l'index ne peut *pas* être utilisé.

L'optimiseur possède forcément quelques limitations. Certaines sont un compromis par rapport au temps que prendrait la recherche systématique de toutes les optimisations imaginables. Il y aussi le problème des estimations de volumétries, qui sont d'autant plus difficiles que la requête est complexe.

Quant à un vrai bug, si le cas peut être reproduit, il doit être remonté aux développeurs de PostgreSQL. D'expérience, c'est rarissime.

1.4.1 Index utilisable mais non utilisé

- L'optimiseur pense qu'il n'est pas rentable
  - sélectivité trop faible
  - meilleur chemin pour remplir d'autres critères
  - index redondant
  - *Index Only Scan* nécessite un VACUUM fréquent
- Les estimations de volumétries doivent être assez bonnes !
  - statistiques récentes, précises

Il existe plusieurs raisons pour que PostgreSQL néglige un index.

Sélectivité trop faible, trop de lignes :

Comme vu précédemment, le parcours d'un index implique à la fois des lectures sur l'index, et des lectures sur la table. Au contraire d'une lecture séquentielle de la table (*Seq Scan*), l'accès aux données via l'index nécessite des lectures aléatoires. Ainsi, si l'optimiseur estime que la requête nécessitera de

parcourir une grande partie de la table, il peut décider de ne pas utiliser l'index : l'utilisation de celui-ci serait alors trop coûteux.

Autrement dit, l'index n'est pas assez discriminant pour que ce soit la peine de faire des allers-retours entre lui et la table. Le seuil dépend entre autres des volumétries de la table et de l'index et du rapport entre les paramètres `random_page_cost` et `seq_page_cost` (respectivement 4 et 1 pour un disque dur classique peu rapide, et souvent 1 et 1 pour du SSD, voire moins).

Il y a un meilleur chemin :

Un index sur un champ n'est qu'un chemin parmi d'autres, en aucun cas une obligation, et une requête contient souvent plusieurs critères sur des tables différentes. Par exemple, un index sur un filtre peut être ignoré si un autre index permet d'éviter un tri coûteux, ou si l'optimiseur juge que faire une jointure avant de filtrer le résultat est plus performant.

Index redondant :

Il existe un autre index doublant la fonctionnalité de celui considéré. PostgreSQL favorise naturellement un index plus petit, plus rapide à parcourir. À l'inverse, un index plus complet peut favoriser plusieurs filtres, des tris, devenir couvrant…

VACUUM trop ancien :

Dans le cas précis des *Index Only Scan*, si la table n'a pas été récemment nettoyée, il y aura trop d'allers-retours avec la table pour vérifier les informations de visibilité (*heap fetches*). Un VACUUM permet de mettre à jour la *Visibility Map* pour éviter cela.

Statistiques périmées :

Il peut arriver que l'optimiseur se trompe quand il ignore un index. Des statistiques périmées sont une cause fréquente. Pour les rafraîchir :

```
ANALYZE (VERBOSE) nom_table;
```

Si cela résout le problème, ce peut être un indice que l'autovacuum ne passe pas assez souvent (voir `pg_stat_user_tables.last_autoanalyze`). Il faudra peut-être ajuster les paramètres `autovacuum_analyze_scale_factor` ou `autovacuum_analyze_threshold` sur les tables.

Statistiques pas assez fines :

Les statistiques sur les données peuvent être trop imprécises. Le défaut est un histogramme de 100 valeurs, basé sur 300 fois plus de lignes. Pour les grosses tables, augmenter l'échantillonnage sur les champs aux valeurs peu homogènes est possible :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500 ;
```

La valeur 500 n'est qu'un exemple. Monter beaucoup plus haut peut pénaliser les temps de planification. Ce sera d'autant plus vrai si on applique cette nouvelle valeur globalement, donc à tous les champs de toutes les tables (ce qui est certes le plus facile).

Estimations de volumétries trompeuses :

Par exemple, une clause WHERE sur deux colonnes corrélées (ville et code postal par exemple), mène à une sous-estimation de la volumétrie résultante par l'optimiseur, car celui-ci ignore le lien entre les deux champs.

À partir de la version 10, vous pouvez indiquer explicitement cette corrélation à PostgreSQL avec l'ordre CREATE STATISTICS (voir https://docs.postgresql.fr/current/sql-createstatistics.html).

1.4.2 Index inutilisable par la requête

- Pas le bon type (CAST plus ou moins explicite)
- Utilisation de fonctions, comme :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

- Pas les bons opérateurs
  - ex : LIKE 'critère%'
- Index invalide

Il faut toujours s'assurer que la requête est écrite correctement et permet l'utilisation de l'index.

Un index peut être inutilisable à cause d'une fonction plus ou moins explicite, ou encore d'un mauvais typage. Il arrive que le critère de filtrage ne peut remonter sur la table indexée à cause d'un CTE matérialisé, d'un DISTINCT, ou d'une vue complexe.

Voici quelques exemples d'index incompatible avec la clause WHERE :

Mauvais type :

Cela peut paraître contre-intuitif, mais certains transtypages ne permettent pas de garantir que les résultats d'un opérateur (par exemple l'égalité) seront les mêmes si les arguments sont convertis dans un type ou dans l'autre.

```
sql=# EXPLAIN SELECT * FROM clients WHERE client_id = 3::numeric;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on clients  (cost=0.00..2525.00 rows=500 width=51)
   Filter: ((client_id)::numeric = 3::numeric)

sql=# EXPLAIN SELECT * FROM clients WHERE client_id = 3;

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Scan using clients_pkey on clients  (cost=0.29..8.31 rows=1 width=51)
   Index Cond: (client_id = 3)
```

Aure exemple : vous avez créé un index B-tree sur un tableau ou un JSON, et vous exécutez une recherche sur un de ses éléments. Il faudra s'orienter vers un index de type GIN, par exemple.

Utilisation de fonction :

Une fonction est appliquée sur la colonne à indexer, comme dans cet exemple classique :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

PostgreSQL n'utilisera pas l'index sur `ma_date`. Il faut réécrire la requête ainsi :

```
SELECT * FROM ma_table WHERE ma_date >='2014-01-01' AND ma_date<'2015-01-01' ;
```

Dans cet autre exemple, on cherche les commandes dont la date tronquée au mois correspond au 1er janvier, c'est-à-dire aux commandes dont la date est entre le 1er et le 31 janvier. Pour un humain, la logique est évidente, mais l'optimiseur n'en a pas connaissance.

```
# EXPLAIN ANALYZE
  SELECT * FROM commandes
  WHERE date_trunc('month', date_commande) = '2015-01-01';

                                QUERY PLAN
-------------------------------------------------------------------------
Gather  (cost=1000.00..8160.96 rows=5000 width=51)
        (actual time=17.282..192.131 rows=4882 loops=1)
 Workers Planned: 3
 Workers Launched: 3
 ->  Parallel Seq Scan on commandes (cost=0.00..6660.96 rows=1613 width=51)
                            (actual time=17.338..177.896 rows=1220 loops=4)
     Filter: (date_trunc('month'::text,
                                    (date_commande)::timestamp with time zone)
                          = '2015-01-01 00:00:00+01'::timestamp with time zone)
     Rows Removed by Filter: 248780
 Planning time: 0.215 ms
 Execution time: 196.930 ms
```

Il faut plutôt écrire :

```
# EXPLAIN ANALYZE
  SELECT * FROM commandes
  WHERE date_commande BETWEEN '2015-01-01' AND '2015-01-31' ;

                            QUERY PLAN
-------------------------------------------------------------
 Index Scan using commandes_date_commande_idx on commandes
                                     (cost=0.42..118.82 rows=5554 width=51)
                               (actual time=0.019..0.915 rows=4882 loops=1)
   Index Cond: ((date_commande >= '2015-01-01'::date)
           AND (date_commande <= '2015-01-31'::date))
 Planning time: 0.074 ms
 Execution time: 1.098 ms
```

Dans certains cas, la réécriture est impossible (fonction complexe, code non modifiable…) et un index fonctionnel sera nécessaire.

Ces exemples semblent évidents, mais il peut être plus compliqué de trouver la cause du problème dans une grande requête d'un schéma mal connu dans l'urgence.

LIKE :

Si vous avez un index « normal » sur une chaîne texte, certaines recherches de type `LIKE` n'utiliseront pas l'index. En effet, il faut bien garder à l'esprit qu'un index ne sert qu'à certains opérateurs. Ceci est généralement indiqué correctement dans la documentation, mais pas forcément très intuitif. Pour plus de détails à ce sujet, se référer à la section correspondant aux classes d'opérateurs[6]. Si un opérateur non supporté est utilisé, l'index ne servira à rien :

```
sql=# CREATE INDEX ON fournisseurs (commentaire);
CREATE INDEX

sql=# EXPLAIN ANALYZE SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on fournisseurs  (cost=0.00..225.00 rows=1 width=45)
                           (actual time=0.045..1.477 rows=47 loops=1)
   Filter: (commentaire ~~ 'ipsum%'::text)
   Rows Removed by Filter: 9953
 Planning time: 0.085 ms
 Execution time: 1.509 ms
```

Nous verrons qu'il existe d'autre classes d'opérateurs, permettant d'indexer correctement la requête précédente.

Index invalide :

Dans le cas où un index a été construit ou ré-indexé avec la clause `CONCURRENTLY`, il peut arriver que l'opération échoue et l'index existe mais reste invalide. De tels index se repèrent :

```
SELECT indexrelid::regclass
FROM pg_index
WHERE indisvalid IS FALSE ;
```

Si cette requête trouve un index invalide, il doit être supprimé ou reconstruit.

[6] https://www.postgresql.org/docs/current/static/indexes-opclass.html

1.5 INDEXATION B-TREE AVANCÉE

De nombreuses possibilités d'indexation avancée :

- Index partiels
- Index fonctionnels
- Index couvrants
- Classes d'opérateur

1.5.1 Index partiels

- N'indexe qu'une partie des données :

```
CREATE INDEX on evenements (type) WHERE  traite IS FALSE ;
```

- Ne sert que si la clause exacte est respectée !
- Intérêt : index beaucoup plus petit

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

Pour prendre un exemple simple, imaginons un système de « queue », dans lequel des événements sont entrés, et qui disposent d'une colonne `traite` indiquant si oui ou non l'événement a été traité. Dans le fonctionnement normal de l'application, la plupart des requêtes ne s'intéressent qu'aux événements non traités :

```
CREATE TABLE evenements (
    id int primary key,
    traite bool NOT NULL,
    type text NOT NULL,
    payload text
);

-- 10 000 événements traités
INSERT INTO evenements (id, traite, type) (
    SELECT i,
        true,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
```

```
    FROM generate_series(1, 10000) as i);

-- et 10  non encore traités
INSERT INTO evenements (id, traite, type) (
    SELECT i,
        false,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
    FROM generate_series(10001, 10010) as i);

\d evenements

                Table « public.evenements »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 | not null  |
 traite  | boolean |                 | not null  |
 type    | text    |                 | not null  |
 payload | text    |                 |           |
Index :
    "evenements_pkey" PRIMARY KEY, btree (id)
```

Typiquement, différents applicatifs vont être intéressés par des événements d'un certain type, mais les événements déjà traités ne sont quasiment jamais accédés, du moins via leur état (une requête portant sur `traite IS true` sera exceptionnelle et ramènera l'essentiel de la table : un index est inutile).

Ainsi, on peut souhaiter indexer le type d'événement, mais uniquement pour les événements non traités :

```
CREATE INDEX index_partiel on evenements (type) WHERE NOT traite ;
```

Si on recherche les événements dont le type est « FACTURATION », sans plus de précision, l'index ne peut évidemment pas être utilisé :

```
EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION' ;

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on evenements  (cost=0.00..183.12 rows=50 width=69)
   Filter: (type = 'FACTURATION'::text)
```

En revanche, si la condition sur l'état de l'événement est précisée, l'index sera utilisé :

```
EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION' AND NOT traite ;

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on evenements  (cost=8.22..54.62 rows=25 width=69)
   Recheck Cond: ((type = 'FACTURATION'::text) AND (NOT traite))
   ->  Bitmap Index Scan on index_partiel  (cost=0.00..8.21 rows=25 width=0)
         Index Cond: (type = 'FACTURATION'::text)
```

Attention ! Les clauses de l'index et du `WHERE` doivent être **strictement identiques** ! Dans cet exemple, un critère `traite IS FALSE` à la place de `NOT traite` n'utilise pas l'index !

Sur ce jeu de données, on peut comparer la taille de deux index, partiels ou non :

```
CREATE INDEX index_complet ON evenements (type);

SELECT idxname, pg_size_pretty(pg_total_relation_size(idxname::text))
FROM (VALUES ('index_complet'), ('index_partiel')) as a(idxname);
```

```
    idxname    | pg_size_pretty
---------------+----------------
 index_complet | 88 kB
 index_partiel | 16 kB
```

Un index composé sur `(is_traite,type)` serait efficace, mais inutilement gros.

1.5.2 Index partiels : cas d'usage

- Données *chaudes* et *froides*
- Index dédié à une requête avec une condition fixe

Le cas typique d'utilisation d'un index partiel est celui de l'exemple précédent : une application avec des données *chaudes*, fréquemment accédées et traitées, et des données *froides*, qui sont plus destinées à de l'historisation ou de l'archivage. Par exemple, un système de vente en ligne aura probablement intérêt à disposer d'index sur les commandes dont l'état est différent de clôturé : en effet, un tel système effectuera probablement des requêtes fréquemment sur les commandes qui sont en cours de traitement, en attente d'expédition, en cours de livraison mais très peu sur des commandes déjà livrées, qui ne serviront alors plus qu'à de l'analyse statistique.

De manière générale, tout système est susceptible de bénéficier des index partiels s'il doit gérer des données à état dont seul un sous-ensemble de ces états est activement exploité par les requêtes à optimiser. Par exemple, toujours sur cette même table, des requêtes visant à faire des statistiques sur les expéditions pourraient tirer parti de cet index :

```
CREATE INDEX index_partiel_expes ON evenements (id) WHERE type = 'EXPEDITION' ;

EXPLAIN SELECT count(id) FROM evenements WHERE type = 'EXPEDITION' ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=106.68..106.69 rows=1 width=8)
   ->  Index Only Scan using index_partiel_expes on evenements  (cost=0.28..98.34
↪  rows=3337 width=4)
```

Nous avons mentionné précédemment qu'un index est destiné à satisfaire une requête ou un ensemble de requêtes. Donc, si une requête présente fréquemment des critères de ce type :

```
WHERE une_colonne = un_parametre_variable
  AND une_autre_colonne = une_valeur_fixe
```

alors il peut être intéressant de créer un index partiel pour les lignes satisfaisant le critère :

```
WHERE une_autre_colonne = une_valeur_fixe
```

Ces critères sont généralement très liés au fonctionnel de l'application : du point de vue de l'exploitation, il est souvent difficile d'identifier des requêtes dont une valeur est toujours fixe. Encore une fois, l'appropriation des techniques d'indexation par l'équipe de développement permet d'améliorer grandement les performances de l'application.

1.5.3 Index partiels : utilisation

- Éviter les index de type :

```
CREATE INDEX ON matable ( champ_filtre ) WHERE champ_filtre = …
```

- Préférer :

```
CREATE INDEX ON matable ( champ_resultat ) WHERE champ_filtre = …
```

En général, un index partiel doit indexer une colonne différente de celle qui est filtrée (et donc connue). Ainsi, dans l'exemple précédent, la colonne indexée (`type`) n'est pas celle de la clause `WHERE`. On pose un critère, mais on s'intéresse aux types d'événements ramenés. Un autre index partiel pourrait porter sur `id WHERE NOT traite` pour simplement récupérer une liste des identifiants non traités de tous types.

L'intérêt est d'obtenir un index très ciblé et compact, et aussi d'économiser la place disque et la charge CPU de maintenance. Il faut tout de même que les index partiels soient notablement plus petits que les index « génériques » (au moins de moitié). Avec des index partiels spécialisés, il est possible de « précalculer » certaines requêtes critiques.

1.5.4 Index fonctionnels

- Un index sur a est inutilisable pour :

```
WHERE upper(a)='DUPOND'
```

- Indexer le résultat de la fonction :

```
CREATE INDEX mon_idx ON ma_table (upper(a)) ;
```

- Fonction impérativement IMMUTABLE !
- Ne pas oublier ANALYZE !
- La fonction ne doit jamais tomber en erreur !
- Modification de la fonction : réindexation !

Indexation de résultat de fonction :

À partir du moment où une clause WHERE applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement.

C'est comme demander à un dictionnaire Anglais vers Français : « Quels sont les mots dont la traduction en français est 'fenêtre' ? ». Le tri du dictionnaire ne correspond pas à la question posée. Il nous faudrait un index non plus sur les mots anglais, mais sur leur traduction en français.

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

L'exemple classique est l'indexation insensible à la casse : on crée un index sur UPPER (ou LOWER) de la chaîne à indexer, et on recherche les mots convertis à la casse souhaitée.

Il est facile de tomber involontairement dans ce cas, notamment avec des manipulations de dates. En général, il est résolu en plaçant la transformation du côté de la constante. Par exemple, la requête suivante retourne toutes les commandes de l'année 2011, mais la fonction `extract` est appliquée à la colonne `date_commande` (type `date`). L'optimiseur ne peut donc pas utiliser un index :

```
tpc=# EXPLAIN SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..5364.12 rows=844 width=77)
   Filter: (date_part('year'::text,
     (date_commande)::timestamp without time zone) = 2011::double precision)
```

En réécrivant le prédicat, l'index est bien utilisé :

```
tpc=# EXPLAIN SELECT * FROM commandes
WHERE date_commande BETWEEN '01-01-2011'::date AND '31-12-2011'::date;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on commandes  (cost=523.85..3302.80 rows=24530 width=77)
   Recheck Cond: ((date_commande >= '2011-01-01'::date)
                  AND (date_commande <= '2011-12-31'::date))
   ->  Bitmap Index Scan on idx_commandes_date_commande
            (cost=0.00..517.72 rows=24530 width=0)
         Index Cond: ((date_commande >= '2011-01-01'::date)
                      AND (date_commande <= '2011-12-31'::date))
```

Mais dans d'autres cas, une telle réécriture de la requête sera impossible.

On peut alors créer un index fonctionnel, dont la définition doit être **strictement** celle du `WHERE` :

```
CREATE INDEX ON commandes( extract('year' from date_commande) ) ;
```

Ceci fonctionne si `date_commande` est de type `date` ou `timestamp without timezone`.

Fonction immutable :

N'importe quelle fonction d'indexation n'est pas utilisable. Elle doit être notée `IMMUTABLE`, indiquant que la fonction retournera toujours le même résultat quand elle est appelée avec les mêmes arguments. En d'autres termes : la fonction ne dépend ni du contenu de la base, ni de la configuration, ni de l'environnement, ni du temps (comme `now()` ou `clock_timestamp()`), et n'a pas de comportement non-déterministe (comme `random()`). Si c'était le cas, l'endroit dans lequel la donnée devrait être insérée dans l'index serait potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Pour calculer l'année, il aurait été aussi possible d'appliquer un index avec la fonction `to_char`, cependant cette dernière n'est pas notée `IMMUTABLE` dans sa définition. Au moment de la création d'un tel index, PostgreSQL renvoie l'erreur suivante :

```
CREATE INDEX ON commandes ( to_char(date_commande,'YYYY') );

ERROR:  functions in index expression must be marked IMMUTABLE
```

En effet, `to_char()` n'est que « stable » car son résultat dépend des paramètres de session. En effet, `to_char(timestamp with time zone, text)` dépend du paramètre de session `timezone` : pour certains *timestamps* autour du Nouvel An, l'année dépend du fuseau horaire…

De même, si `date_commande` est de type `timestamp with time zone`, donc avec fuseau, la fonction précédente échoue :

```
CREATE INDEX ON commandes( extract('year' from date_commande) ) ;

ERROR:  functions in index expression must be marked IMMUTABLE
```

Au moins `extract` ou `date_trunc` sont-ils bien immutables avec les types sans fuseau horaires :

```
postgres=# \df+ extract
                                  Liste des fonctions
…   Nom    | Type… rés. |  Type de données des paramètres   | Type | Volatibilité |
… extract | numeric    | text, date                        | func | immutable    |
… extract | numeric    | text, interval                    | func | immutable    |
… extract | numeric    | text, timestamp without time zone | func | immutable    |
```

```
… extract | numeric    | text, timestamp with time zone    | func | stable       |
… extract | numeric    | text, time without time zone      | func | immutable    |
… extract | numeric    | text, time with time zone         | func | immutable    |
```

Il est possible de « tricher » en figeant le fuseau horaire via une fonction :

```
CREATE OR REPLACE FUNCTION annee_paris (t timestamptz)
RETURNS int
AS $$
    SELECT extract ('year' FROM (t AT TIME ZONE 'Europe/Paris')::timestamp) ;
$$ LANGUAGE sql  IMMUTABLE ;
```

Mais le critère dans le code devra changer :

```
SELECT * FROM commandes
WHERE annee_paris (date_commande) = 2021 ;
```

Au moins le nom de la fonction est-il un avertissement pour les utilisateurs dans d'autres fuseaux, même aussi près que Londres.

Statistiques :

Après la création de l'index fonctionnel, un `ANALYZE nom_table` est conseillé : en effet, l'optimiseur ne peut utiliser les statistiques déjà connues sur les valeurs dans la table pour un index fonctionnel. Il faut donc indexer les valeurs calculées.

Ces statistiques seront visibles dans la vue système `pg_stats` avec comme `tablename` le nom de l'index (et non celui de la table !).

Avertissements :

ATTENTION : la fonction ne doit jamais tomber en erreur ! Il ne faut pas tester que les données en place mais toutes celles susceptibles de se trouver dans la champ concerné. Sinon, des `ANALYZE` ou `VACUUM` pourraient échouer, avec de gros problèmes sur le long terme.

ATTENTION : si le contenu de la fonction est modifié avec `CREATE OR REPLACE FUNCTION`, il faudra impérativement réindexer. Sinon, les résultats des requêtes différeront selon qu'elles utiliseront ou non l'index !

1.5.5 Index couvrants

```
CREATE UNIQUE INDEX clients_idx1 ON clients (id_client) INCLUDE
↪  (nom_client) ;
```

- Répondent à la clause WHERE
- **ET** contiennent toutes les colonnes demandées par la requête :

```
SELECT id_client,nom_client FROM clients WHERE id_client > 100 ;
```

- Limite les visites à la table

Un parcours d'index classique (*Index Scan*) est en fait un aller/retour entre l'index et la table : on va chercher un enregistrement dans l'index, qui nous donne son adresse dans la table, on accède à cet enregistrement dans la table, puis on passe à l'entrée d'index suivante. Le coût en entrées-sorties peut être énorme : les données de la table sont habituellement éparpillées dans tous les blocs.

Un index couvrant (*covering index*) évite cela en plaçant dans l'index non seulement les champs servant de critères de recherche, mais aussi les champs résultats.

Un index couvrant peut ainsi permettre un *Index Only Scan* car il n'a plus besoin d'interroger la table. Pour pouvoir en bénéficier, il faut que toutes les colonnes retournées par la requête soient présentes dans l'index. De plus, les enregistrements cherchés étant contigus dans l'index (puisqu'il est trié), le nombre d'accès disque est bien plus faible, ce qui peut apporter des gains de performances énormes en sélection. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

Les index couvrants peuvent être explicitement déclarés à partir de la version 11 avec la clause INCLUDE :

```
CREATE TABLE t (id int NOT NULL, valeur int) ;

INSERT INTO t SELECT i, i*50 FROM generate_series(1,1000000) i;

CREATE UNIQUE INDEX t_pk ON t (id) INCLUDE (valeur) ;

VACUUM t ;

EXPLAIN ANALYZE SELECT valeur FROM t WHERE id = 555555 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Index Only Scan using t_pk on t  (cost=0.42..1.44 rows=1 width=4)
                                  (actual time=0.034..0.035 rows=1 loops=1)
   Index Cond: (id = 555555)
   Heap Fetches: 0
 Planning Time: 0.084 ms
 Execution Time: 0.065 ms
```

Dans cet exemple, il n'y a pas eu d'accès à la table. L'index est unique mais contient aussi la colonne `valeur`.

Noter le VACUUM, nécessaire pour garantir que la *visibility map* de la table est à jour et permet ainsi un *Index Only Scan* sans aucun accès à la table (clause *Heap Fetches* à 0).

Dans les versions antérieures à la 11, le principe reste valable : il suffit de déclarer les colonnes dans des index (`CREATE INDEX t_idx ON t (id, valeur)`). La clause `INCLUDE` a l'avantage de pouvoir se greffer sur des index uniques ou de clés et ainsi d'économiser des créations d'index, ainsi que d'éviter le tri des champs dans la clause `INCLUDE`.

1.5.6 Index couvrants : inconvénients et compatibilité

- Inconvénients :
  - index plus gros
  - limite d'enregistrement (2,6 ko)
  - pas de déduplication
- Compatibilité : B-tree (v11), GiST (v12), SP-GiST (v14), principe valable avant v11

Il faut garder à l'esprit que l'ajout de colonnes à un index augmente sa taille. Cela peut avoir un impact sur les performances des requêtes qui n'utilisent pas la colonne qui a été ajoutée. Il faut également être vigilant à ce que la taille des enregistrements avec les colonnes incluses ne dépassent pas 2,6 ko. Au-delà de cette valeur, les insertions ou mises à jour échouent. Enfin, la déduplication (apparue en version 13) n'est pas active sur les index couvrants.

Les méthodes d'accès aux index doivent inclure le support de cette fonctionnalité. C'est le cas pour le B-tree en version 11, pour le GiST en version 12 et pour le SP-GiST en version 14.

1.5.7 Classes d'opérateurs

- Un index utilise des opérateurs de comparaison
- Texte : différentes collations = différents tris… complexes
  - Index inutilisable sur :

```
WHERE col_varchar LIKE 'chaine%'
```

- Solution : opérateur `varchar_pattern_ops` :
  - force le tri caractère par caractère, sans la collation

```
CREATE INDEX idx1
ON ma_table (col_varchar varchar_pattern_ops)
```

- Plus généralement :
  - nombreux autres opérateurs pour d'autres types d'index

Il est tout à fait possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison pour l'indexation, dans des cas particuliers.

Le cas d'utilisation le plus fréquent d'utilisation dans PostgreSQL est la comparaison de chaîne `LIKE 'chaine%'`. L'indexation texte « classique » utilise la collation par défaut de la base (en France, généralement `fr_FR.UTF-8` ou `en_US.UTF-8`) ou la collation de la colonne de la table si elle diffère. Cette collation contient des notions de tri. Les règles sont différentes pour chaque collation. Les nouvelles collations sont à déclarer séparément dans chaque base.

Par exemple, le **ß** allemand se place entre **ss** et **t** (et ce, même en français). En danois, le tri est très particulier car le **å** et le **aa** apparaissent après le **z**.

```
-- Cette collation doit exister sur le système
CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls ORDER BY x COLLATE "da_DK";
```

```
 x
----
 s
 ss
 ß
 t
 zz
 å
 aa
```

Il faut être conscient que cela a une influence sur le résultat d'un filtrage :

```
WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls
WHERE x > 'z' COLLATE "da_DK" ;

 x
----
 aa
 å
 zz
```

Il serait donc très complexe de réécrire le `LIKE` en un `BETWEEN`, comme le font habituellement tous les SGBD : `col_texte LIKE 'toto%'` peut être réécrit comme `coltexte >= 'toto' and coltexte < 'totp'` en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur Unicode par exemple. Même si l'index est dans la bonne collation, il n'est pas facilement utilisable :

```
CREATE INDEX ON textes (livre) ;
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..525328.76 rows=75173 width=123)
   Workers Planned: 2
   ->  Parallel Seq Scan on textes  (cost=0.00..516811.46 rows=31322 width=123)
         Filter: (livre ~~ 'Les misérables%'::text)
```

La classe d'opérateurs `varchar_pattern_ops` sert à forcer ce comportement : l'index est construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient.

```
CREATE INDEX ON ma_table (col_varchar varchar_pattern_ops)
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                               QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using textes_livre_idx1 on textes  (cost=0.69..70406.87 rows=75173
↪  width=123)
   Index Cond: ((livre ~>=~ 'Les misérables'::text) AND (livre ~<~ 'Les
↪  misérablet'::text))
   Filter: (livre ~~ 'Les misérables%'::text)
```

Il devient alors trivial pour l'optimiseur de faire la réécriture. Cela convient pour un `LIKE 'toto%'`, car le début est fixe, et l'ordre de tri n'influe pas sur le résultat. Noter la clause `Filter` qui filtre en deuxième intention ce qui a pu être trouvé dans l'index.

Il existe quelques autres cas d'utilisation d'`opclass` alternatives, notamment pour utiliser d'autres types d'index que B-tree. Deux exemples :

- indexation d'un JSON (type `jsonb`) par un index GIN :

```
CREATE INDEX ON stock_jsonb USING gin (document_jsonb jsonb_path_ops);
```

- indexation de trigrammes de textes avec le module `pg_trgm` et des index GiST :

```
CREATE INDEX ON livres USING gist (text_data gist_trgm_ops);
```

1.5.8 Conclusion

- Responsabilité de l'indexation
- Compréhension des mécanismes
- Différents types d'index, différentes stratégies

L'indexation d'une base de données est souvent un sujet qui est traité trop tard dans le cycle de l'application. Lorsque celle-ci est gérée à l'étape du développement, il est possible de bénéficier de l'expérience et de la connaissance des développeurs. La maîtrise de cette compétence est donc idéalement transverse entre le développement et l'exploitation.

Le fonctionnement d'un index B-tree est somme toute assez simple, mais il est important de bien l'appréhender pour comprendre les enjeux d'une bonne stratégie d'indexation.

PostgreSQL fournit aussi d'autres types d'index moins utilisés, mais très précieux dans certaines situations : BRIN, GIN, GiST, etc.

1.6 QUIZ

https://dali.bo/j4_quiz

1.7 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 (similaire à Red Hat 8), Red Hat/CentOS 7, et Debian/Ubuntu.

Elle ne dure que quelques minutes.

ATTENTION : Red Hat et CentOS 6 et 7, comme Rocky 8, fournissent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` !
L'utilisation des dépôts du PGDG est donc obligatoire.

1.7.1 Sur Rocky Linux 8

Installation du dépôt communautaire :

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent peuvent être générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 15) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Il faut installer le dépôt et désactiver le module PostgreSQL par défaut :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 15 :

```
# dnf install -y postgresql15-server postgresql15-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Tout à fait optionnellement, une fonctionnalité avancée, le JIT (*Just In Time compilation*), nécessite un paquet séparé, qui lui-même nécessite des paquets du dépôt EPEL :

```
# dnf install postgresql15-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-15/bin/postgresql-15-setup initdb
# cat /var/lib/pgsql/15/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé.

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-15/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
PGDATA par défaut	`/var/lib/pgsql/15/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

```
# systemctl start postgresql-15
# systemctl stop postgresql-15
# systemctl reload postgresql-15
```

Test rapide de bon fonctionnement et connexion à psql

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au démarrage du système d'exploitation :

```
# systemctl enable postgresql-15
```

Consultation de l'état de l'instance :

```
# systemctl status postgresql-15
```

Ouverture du *firewall* pour le port 5432 :

Si le *firewall* est actif (dans le doute, consulter `systemctl status firewalld`) :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faudra installer les binaires pour chacune, et l'instance par défaut de chaque version vivra dans un sous-répertoire différent de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf`.

Si plusieurs instances d'une même version majeure (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faudra les installer dans des PGDATA différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un sous-répertoire de `/var/lib/pgsqsl/15/` (ou l'équivalent pour d'autres versions majeures).
- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-15.service \
        /etc/systemd/system/postgresql-15-secondaire.service
```

- Modification du fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/15/secondaire
```

- Option 1 : création d'une nouvelle instance vierge :

```
# /usr/pgsql-15/bin/postgresql-15-setup initdb postgresql-15-secondaire
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `postgresql.conf` (port !), `recovery.conf`…
- Commandes de maintenance :

```
# systemctl [start|stop|reload|status] postgresql-15-secondaire
# systemctl [enable|disable] postgresql-15-secondaire
```

- Ouvrir un port dans le firewall au besoin.

1.7.2 Sur Red Hat 7 / Cent OS 7

Fondamentalement, le principe reste le même qu'en version 8. Il faudra utiliser `yum` plutôt que `dnf`. Il n'y a pas besoin de désactiver de module AppStream. Le JIT (*Just In Time compilation*), nécessite un paquet séparé, qui lui-même nécessite des paquets du dépôt EPEL :

```
# yum install epel-release
# yum install postgresql15-llvmjit
```

La création de l'instance et la suite sont identiques.

1.7.3 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Installation du dépôt communautaire :

Référence : https://apt.postgresql.org/

- Import des certificats et de la clé :

```
# apt install curl ca-certificates gnupg
# curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | gpg --dearmor | \
          sudo tee /etc/apt/trusted.gpg.d/apt.postgresql.org.gpg >/dev/null
```

- Création du fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` (ici pour Debian 11 « bullseye » ; adapter au nom de code de la version de Debian ou Ubuntu correspondante : **stretch**, **bionic**, **focal**…) :

```
deb http://apt.postgresql.org/pub/repos/apt bullseye-pgdg main
```

Installation de PostgreSQL 15 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

```
# apt update
# apt install postgresql-common
```

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-15 postgresql-client-15
```

(Pour les versions 9.x, installer aussi le paquet `postgresql-contrib-9.x`).

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/15/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/15/main`
Fichiers de configuration	dans `/etc/postgresql/15/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils :

```
# pg_ctlcluster 15 main [start|stop|reload|status]
```

Démarrage de l'instance au lancement :

C'est en place par défaut, et modifiable dans `/etc/postgresql/15/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 15 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- les paramètres de création d'instance dans `/etc/postgresql-common/createcluster.conf` peuvent être modifiés, par exemple ici pour : les *checksums*, les messages en anglais, l'authentification sécurisée, le format des traces et un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C --auth-host=scram-sha-256
 ↪  --auth-local=peer'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- création de l'instance, avec possibilité là aussi de préciser certains paramètres du `postgresql.conf` voire de modifier les chemins des fichiers (déconseillé si vous pouvez l'éviter) :

```
# pg_createcluster 15 secondaire \
  --port=5433 \
  --datadir=/PGDATA/11/basedecisionnelle \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/11/basedecisionnelle/journaux
```

- démarrage :

```
# pg_ctlcluster 15 secondaire start
```

1.7.4 Accès à l'instance sur le serveur même

Par défaut, l'instance n'est accessible que par l'utilisateur système `postgres`, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (15.1)
Saisissez « help » pour l'aide.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (15.1)
Saisissez « help » pour l'aide.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, mise en place d'une authentification par mot de passe (`md5` par défaut) pour les accès à `localhost` :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

(une authentification `scram-sha-256` est plus conseillée mais elle impose que `password_encryption` soit à cette valeur dans `postgresql.conf` avant de définir les mots de passe).

- ajout d'un mot de passe à l'utilisateur `postgres` de l'instance ;

```
dalibo:~$ sudo -iu postgres psql
psql (15.1)
Saisissez « help » pour l'aide.
postgres=# \password
Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
postgres=# \q

dalibo:~$ psql -h localhost -U postgres
Mot de passe pour l'utilisateur postgres :
psql (15.1)
Saisissez « help » pour l'aide.
postgres=#
```

- pour se connecter sans taper le mot de passe, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

- ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/15/data/log` ou `/var/log/postgresql/`);
- toute modification de `pg_hba.conf` implique de recharger la configuration par une de ces trois méthodes selon le système :

```
root:~# systemctl reload postgresql-15

root:~# pg_ctlcluster 15 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf();'
```

1.8 TRAVAUX PRATIQUES

Cette série de question utilise la base **magasin**. La base **magasin** peut être téléchargée depuis https://dali.bo/tp_magasin (dump de 96 Mo, pour 667 Mo sur le disque au final). Elle s'importe de manière très classique (une erreur sur le schéma **public** déjà présent est normale), ici dans une base nommée aussi `magasin` :

```
curl -L https://dali.bo/tp_magasin -o magasin.dump
pg_restore -d magasin  magasin.dump
```

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

1.8.1 Index « simples »

But : Mettre en avant un cas d'usage d'un index « simple »

Considérons le cas d'usage d'une recherche de commandes par date. Le besoin fonctionnel est le suivant : renvoyer l'intégralité des commandes passées au mois de janvier 2014.

> Créer la requête affichant l'intégralité des commandes passées au mois de janvier 2014.

> Afficher le plan de la requête , en utilisant `EXPLAIN (ANALYZE, BUFFERS)`. Que constate-t-on ?

Nous souhaitons désormais afficher les résultats à l'utilisateur par ordre de date croissante.

> Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan. Que constate-t-on ?

Maintenant, nous allons essayer d'optimiser ces deux requêtes.

> Créer un index permettant de répondre à ces requêtes.

> Afficher de nouveau le plan des deux requêtes. Que constate-t-on ?

Maintenant, étudions l'impact des index pour une opération de jointure. Le besoin fonctionnel est désormais de lister toutes les commandes associées à un client (admettons, dont le `client_id` vaut 3), avec les informations du client lui-même.

> Écrire la requête affichant `commande.nummero_commande` et `client.type_client` pour `client_id = 3`. Afficher son plan. Que constate-t-on ?

Créer un index pour accélérer cette requête.

Afficher de nouveau son plan. Que constate-t-on ?

1.8.2 Sélectivité

But : Comprendre la sélectivité des index.

Écrire une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').

Ajouter un index sur la colonne `type_client`, et rejouer les requêtes précédentes.

Afficher leurs plans d'exécution. Que se passe-t-il ? Pourquoi ?

1.8.3 Index partiels

But : Mettre en avant un cas d'usage d'un index partiel

Sur la base fournie pour les TPs, les lots non livrés sont constamment requêtés. Notamment, un système d'alerte est mis en place afin d'assurer un suivi qualité sur les lots expédié depuis plus de 3 jours (selon la date d'expédition), mais non réceptionné (date de réception à NULL).

Écrire la requête correspondant à ce besoin fonctionnel (il est normal qu'elle ne retourne rien).

Afficher le plan d'exécution.

Quel index partiel peut-on créer pour optimiser ?

Afficher le nouveau plan d'exécution et vérifier l'utilisation du nouvel index.

1.8.4 Index fonctionnels

But : Mettre en avant un cas d'usage d'un index fonctionnel

Pour répondre aux exigences de stockage, l'application a besoin de pouvoir trouver rapidement les produits dont le volume est compris entre certaines bornes (nous négligeons ici le facteur de forme, qui est problématique dans le cadre d'un véritable stockage en entrepôt !).

Écrire une requête permettant de renvoyer l'ensemble des produits dont le volume ne dépasse pas 1 litre (les unités de longueur sont en mm, 1 litre = 1 000 000 mm^3).

Quel index permet d'optimiser cette requête ? (Indexer le résultat d'une nouvelle fonction est possible, mais pas obligatoire.)

1.8.5 Cas d'index non utilisés

But : Mettre en avant des cas d'index inutilisés

Un développeur cherche à récupérer les commandes dont le numéro d'expédition est 190774 avec cette requête :

```
SELECT * FROM lignes_commandes WHERE numero_lot_expedition = '190774'::numeric ;
```

Afficher le plan de la requête.

Créer un index pour améliorer son exécution.

L'index est-il utilisé ? Quel est le problème ?

Écrire une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

Créer un index pour améliorer l'exécution de cette requête.

Pourquoi celui-ci n'est-il pas utilisé ? (Conseil : regarder la vue `pg_stats`)

> Faire le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

1.9 TRAVAUX PRATIQUES (SOLUTIONS)

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

1.9.1 Index « simples »

Considérons le cas d'usage d'une recherche de commandes par date. Le besoin fonctionnel est le suivant : renvoyer l'intégralité des commandes passées au mois de janvier 2014.

> Créer la requête affichant l'intégralité des commandes passées au mois de janvier 2014.

Pour renvoyer l'ensemble de ces produits, la requête est très simple :

```
SELECT * FROM commandes date_commande
WHERE date_commande >= '2014-01-01'
AND date_commande < '2014-02-01';
```

> Afficher le plan de la requête , en utilisant `EXPLAIN (ANALYZE, BUFFERS)`. Que constate-t-on ?

Le plan de celle-ci est le suivant :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..25158.00 rows=19674 width=50)
                        (actual time=2.436..102.300 rows=19204 loops=1)
   Filter: ((date_commande >= '2014-01-01'::date)
            AND (date_commande < '2014-02-01'::date))
   Rows Removed by Filter: 980796
   Buffers: shared hit=10158
 Planning time: 0.057 ms
 Execution time: 102.929 ms
```

> Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan. Que constate-t-on ?

Ajoutons la clause ORDER BY :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01'
ORDER BY date_commande;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=26561.15..26610.33 rows=19674 width=50)
       (actual time=103.895..104.726 rows=19204 loops=1)
```

```
   Sort Key: date_commande
   Sort Method: quicksort  Memory: 2961kB
   Buffers: shared hit=10158
   ->  Seq Scan on commandes  (cost=0.00..25158.00 rows=19674 width=50)
                              (actual time=2.801..102.181
                               rows=19204 loops=1)
         Filter: ((date_commande >= '2014-01-01'::date)
                  AND (date_commande < '2014-02-01'::date))
         Rows Removed by Filter: 980796
         Buffers: shared hit=10158
 Planning time: 0.096 ms
 Execution time: 105.410 ms
```

On constate ici que lors du parcours séquentiel, 980 796 lignes ont été lues, puis écartées car ne correspondant pas au prédicat, nous laissant ainsi avec un total de 19 204 lignes. Les valeurs précises peuvent changer, les données étant générées aléatoirement. De plus, le tri a été réalisé en mémoire. On constate de plus que 10 158 blocs ont été parcourus, ici depuis le cache, mais ils auraient pu l'être depuis le disque.

> Créer un index permettant de répondre à ces requêtes.

Création de l'index :

```
CREATE INDEX idx_commandes_date_commande ON commandes(date_commande);
```

> Afficher de nouveau le plan des deux requêtes. Que constate-t-on ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01';
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using idx_commandes_date_commande on commandes
        (cost=0.42..822.60 rows=19674 width=50)
        (actual time=0.015..3.311 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
                AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
 Planning time: 0.074 ms
 Execution time: 4.133 ms
```

Le temps d'exécution a été réduit considérablement : la requête est 25 fois plus rapide. On constate notamment que seuls 254 blocs ont été parcourus.

Pour la requête avec la clause ORDER BY, nous obtenons le plan d'exécution suivant :

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using idx_commandes_date_commande on commandes
        (cost=0.42..822.60 rows=19674 width=50)
        (actual time=0.032..3.378 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
                AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
```

```
 Planning time: 0.516 ms
 Execution time: 4.049 ms
```

Celui-ci est identique ! En effet, l'index permettant un parcours trié, l'opération de tri est ici « gratuite ».

> Écrire la requête affichant `commande.nummero_commande` et `client.type_client` pour `client_id = 3`. Afficher son plan. Que constate-t-on ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT numero_commande, type_client FROM commandes
      INNER JOIN clients ON commandes.client_id = clients.client_id
    WHERE clients.client_id = 3;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..22666.42 rows=11 width=101)
              (actual time=8.799..80.771 rows=14 loops=1)
   Buffers: shared hit=10161
   ->  Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.017..0.018 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=3
   ->  Seq Scan on commandes  (cost=0.00..22658.00 rows=11 width=50)
                              (actual time=8.777..80.734 rows=14 loops=1)
         Filter: (client_id = 3)
         Rows Removed by Filter: 999986
         Buffers: shared hit=10158
 Planning time: 0.281 ms
 Execution time: 80.853 ms
```

> Créer un index pour accélérer cette requête.

```
CREATE INDEX ON commandes (client_id) ;
```

> Afficher de nouveau son plan. Que constate-t-on ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
      INNER JOIN clients on commandes.client_id = clients.client_id
      WHERE clients.client_id = 3;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=4.80..55.98 rows=11 width=101)
              (actual time=0.064..0.189 rows=14 loops=1)
   Buffers: shared hit=23
   ->  Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.032..0.032 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=6
   ->  Bitmap Heap Scan on commandes  (cost=4.51..47.56 rows=11 width=50)
                                      (actual time=0.029..0.147
                                       rows=14 loops=1)
         Recheck Cond: (client_id = 3)
```

```
         Heap Blocks: exact=14
         Buffers: shared hit=17
         ->  Bitmap Index Scan on commandes_client_id_idx
                    (cost=0.00..4.51 rows=11 width=0)
                    (actual time=0.013..0.013 rows=14 loops=1)
               Index Cond: (client_id = 3)
               Buffers: shared hit=3
 Planning time: 0.486 ms
 Execution time: 0.264 ms
```

On constate ici un temps d'exécution divisé par 160 : en effet, on ne lit plus que 17 blocs pour la commande (3 pour l'index, 14 pour les données) au lieu de 10 158.

1.9.2 Sélectivité

> Écrire une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').

Les requêtes :

```
SELECT * FROM clients WHERE type_client = 'P';
SELECT * FROM clients WHERE type_client = 'E';
```

> Ajouter un index sur la colonne `type_client`, et rejouer les requêtes précédentes.

Pour créer l'index :

```
CREATE INDEX ON clients (type_client);
```

> Afficher leurs plans d'exécution. Que se passe-t-il ? Pourquoi ?

Les plans d'éxécution :

```
EXPLAIN ANALYZE SELECT * FROM clients WHERE type_client = 'P';

                             QUERY PLAN
---------------------------------------------------------------------
 Seq Scan on clients  (cost=0.00..2276.00 rows=89803 width=51)
                      (actual time=0.006..12.877 rows=89800 loops=1)
   Filter: (type_client = 'P'::bpchar)
   Rows Removed by Filter: 10200
 Planning time: 0.374 ms
 Execution time: 16.063 ms

EXPLAIN ANALYZE SELECT * FROM clients WHERE type_client = 'E';

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=154.50..1280.84 rows=8027 width=51)
                              (actual time=2.094..4.287 rows=8111 loops=1)
   Recheck Cond: (type_client = 'E'::bpchar)
   Heap Blocks: exact=1026
```

```
   ->  Bitmap Index Scan on clients_type_client_idx
         (cost=0.00..152.49 rows=8027 width=0)
         (actual time=1.986..1.986 rows=8111 loops=1)
       Index Cond: (type_client = 'E'::bpchar)
 Planning time: 0.152 ms
 Execution time: 4.654 ms
```

L'optimiseur sait estimer, à partir des statistiques (consultables via la vue `pg_stats`), qu'il y a approximativement 89 000 clients particuliers, contre 8 000 clients entreprise.

Dans le premier cas, la majorité de la table sera parcourue, et renvoyée : il n'y a aucun intérêt à utiliser l'index.

Dans l'autre, le nombre de lignes étant plus faible, l'index est bel et bien utilisé (via un *Bitmap Scan*, ici).

1.9.3 Index partiels

> Sur la base fournie pour les TPs, les lots non livrés sont constamment requêtés. Notamment, un système d'alerte est mis en place afin d'assurer un suivi qualité sur les lots expédié depuis plus de 3 jours (selon la date d'expédition), mais non réceptionné (date de réception à NULL).
>
> Écrire la requête correspondant à ce besoin fonctionnel (il est normal qu'elle ne retourne rien).

La requête est la suivante :

```
SELECT * FROM lots
    WHERE date_reception IS NULL
    AND   date_expedition < now() - '3d'::interval;
```

> Afficher le plan d'exécution.

Le plans (ci-dessous avec ANALYZE) opère un *Seq Scan* parallélisé, lit et rejette toutes les lignes, ce qui est évidemment lourd :

```
                          QUERY PLAN
---------------------------------------------------------------
 Gather  (cost=1000.00..17764.65 rows=1 width=43) (actual time=28.522..30.993 rows=0
↪  loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on lots  (cost=0.00..16764.55 rows=1 width=43) (actual
↪  time=24.887..24.888 rows=0 loops=3)
         Filter: ((date_reception IS NULL) AND (date_expedition < (now() - '3
↪  days'::interval)))
         Rows Removed by Filter: 335568
 Planning Time: 0.421 ms
 Execution Time: 31.012 ms
```

> Quel index partiel peut-on créer pour optimiser ?

On peut optimiser ces requêtes sur les critères de recherche à l'aide des index partiels suivants :

```
CREATE INDEX ON lots (date_expedition) WHERE date_reception  IS NULL;
```

> Afficher le nouveau plan d'exécution et vérifier l'utilisation du nouvel index.

```
EXPLAIN (ANALYZE)
SELECT * FROM lots
    WHERE date_reception IS NULL
    AND   date_expedition < now() - '3d'::interval;

                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using lots_date_expedition_idx on lots  (cost=0.13..4.15 rows=1
 ↪  width=43) (actual time=0.008..0.009 rows=0 loops=1)
   Index Cond: (date_expedition < (now() - '3 days'::interval))
 Planning Time: 0.243 ms
 Execution Time: 0.030 ms
```

Il est intéressant de noter que seul le test sur la condition indexée (`date_expedition`) est présent dans le plan : la condition `date_reception IS NULL` est implicitement validée par l'index partiel.

Attention, il peut être tentant d'utiliser une formulation de la sorte pour ces requêtes :

```
SELECT * FROM lots
WHERE date_reception IS NULL
AND   now() - date_expedition > '3d'::interval;
```

D'un point de vue logique, c'est la même chose, mais l'optimiseur n'est pas capable de réécrire cette requête correctement. Ici, le nouvel index sera tout de même utilisé, le volume de lignes satisfaisant au critère étant très faible, mais il ne sera pas utilisé pour filtrer sur la date :

```
EXPLAIN (ANALYZE) SELECT * FROM lots
      WHERE date_reception IS NULL
      AND   now() - date_expedition > '3d'::interval;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using lots_date_expedition_idx on lots
        (cost=0.12..4.15 rows=1 width=43)
        (actual time=0.007..0.007 rows=0 loops=1)
   Filter: ((now() - (date_expedition)::timestamp with time zone) >
           '3 days'::interval)
 Planning time: 0.204 ms
 Execution time: 0.132 ms
```

La ligne importante et différente ici concerne le `Filter` en lieu et place du `Index Cond` du plan précédent. Ici tout l'index partiel (certes tout petit) est lu intégralement et les lignes testées une à une.

C'est une autre illustration des points vus précédemment sur les index non utilisés.

1.9.4 Index fonctionnels

> Écrire une requête permettant de renvoyer l'ensemble des produits dont le volume ne dépasse pas 1 litre (les unités de longueur sont en mm, 1 litre = 1 000 000 mm^3).

Concernant le volume des produits, la requête est assez simple :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000 ;
```

> Quel index permet d'optimiser cette requête ? (Indexer le résultat d'une nouvelle fonction est possible, mais pas obligatoire.)

On peut aussi tout simplement créer l'index de cette façon, sans avoir besoin d'une fonction :

```
CREATE INDEX ON produits((longueur * hauteur * largeur));
```

En général, il est plus propre de créer une fonction. Il faut que cette une fonction soit `IMMUTABLE` :

```
CREATE OR REPLACE function volume(p produits) RETURNS numeric
AS $$
 SELECT p.longueur * p.hauteur * p.largeur;
$$ language SQL
IMMUTABLE ;
```

On peut ensuite indexer le résultat de cette fonction :

```
CREATE INDEX ON produits (volume(produits));
```

Il est ensuite possible d'écrire la requête de plusieurs manières, la fonction étant ici écrite en SQL et non en PL/pgSQL ou autre langage procédural :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000 ;
SELECT * FROM produits WHERE volume(produits) < 1000000 ;
```

En effet, l'optimiseur est capable de « regarder » à l'intérieur de la fonction SQL pour déterminer que les clauses sont les mêmes, ce qui n'est pas vrai pour les autres langages.

De part l'origine « relationnel-objet » de PostgreSQL, on peut même écrire la requête de la manière suivante :

```
SELECT * FROM produits WHERE produits.volume < 1000000;
```

1.9.5 Cas d'index non utilisés

> Afficher le plan de la requête.

```
SELECT * FROM lignes_commandes WHERE numero_lot_expedition = '190774'::numeric;

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM lignes_commandes
                            WHERE numero_lot_expedition = '190774'::numeric;
```

```
                     QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on lignes_commandes
                        (cost=0.00..89331.51 rows=15710 width=74)
                        (actual time=0.024..1395.705 rows=6 loops=1)
   Filter: ((numero_lot_expedition)::numeric = '190774'::numeric)
   Rows Removed by Filter: 3141961
   Buffers: shared hit=97 read=42105
 Planning time: 0.109 ms
 Execution time: 1395.741 ms
```

Le moteur fait un parcours séquentiel et retire la plupart des enregistrements pour n'en conserver que 6.

> Créer un index pour améliorer son exécution.

```
CREATE INDEX ON lignes_commandes (numero_lot_expedition);
```

> L'index est-il utilisé ? Quel est le problème ?

L'index n'est pas utilisé à cause de la conversion `bigint` vers `numeric`. Il est important d'utiliser les bons types :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM lignes_commandes
WHERE numero_lot_expedition = '190774' ;
```

```
                          QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using lignes_commandes_numero_lot_expedition_idx
    on lignes_commandes
        (cost=0.43..8.52 rows=5 width=74)
        (actual time=0.054..0.071 rows=6 loops=1)
   Index Cond: (numero_lot_expedition = '190774'::bigint)
   Buffers: shared hit=1 read=4
 Planning time: 0.325 ms
 Execution time: 0.100 ms
```

Sans conversion la requête est bien plus rapide. Faites également le test sans index, le *Seq Scan* sera également plus rapide, le moteur n'ayant pas à convertir toutes les lignes parcourues.

> Écrire une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM lignes_commandes
                                  WHERE quantite BETWEEN 1 AND 8;
```

```
                           QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on lignes_commandes
        (cost=0.00..89331.51 rows=2504357 width=74)
        (actual time=0.108..873.666 rows=2512740 loops=1)
   Filter: ((quantite >= 1) AND (quantite <= 8))
   Rows Removed by Filter: 629227
```

```
   Buffers: shared hit=16315 read=25887
 Planning time: 0.369 ms
 Execution time: 1009.537 ms
```

> Créer un index pour améliorer l'exécution de cette requête.

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

> Pourquoi celui-ci n'est-il pas utilisé ? (Conseil : regarder la vue `pg_stats`)

La table `pg_stats` nous donne des informations de statistiques. Par exemple, pour la répartition des valeurs pour la colonne quantite:

```
SELECT * FROM pg_stats
WHERE tablename='lignes_commandes' AND attname='quantite'
\gx

…
n_distinct             | 10
most_common_vals       | {0,6,1,8,2,4,7,9,5,3}
most_common_freqs      | {0.1037,0.1018,0.101067,0.0999333,0.0999,0.0997,
                           0.0995,0.0992333,0.0978333,0.0973333}
…
```

Ces quelques lignes nous indiquent qu'il y a 10 valeurs distinctes et qu'il y a environ 10 % d'enregistrements correspondant à chaque valeur.

Avec le prédicat `quantite BETWEEN 1 and 8`, le moteur estime récupérer environ 80 % de la table. Il est donc bien plus coûteux de lire l'index et la table pour récupérer 80 % de la table. C'est pourquoi le moteur fait un *Seq Scan* qui moins coûteux.

> Faire le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

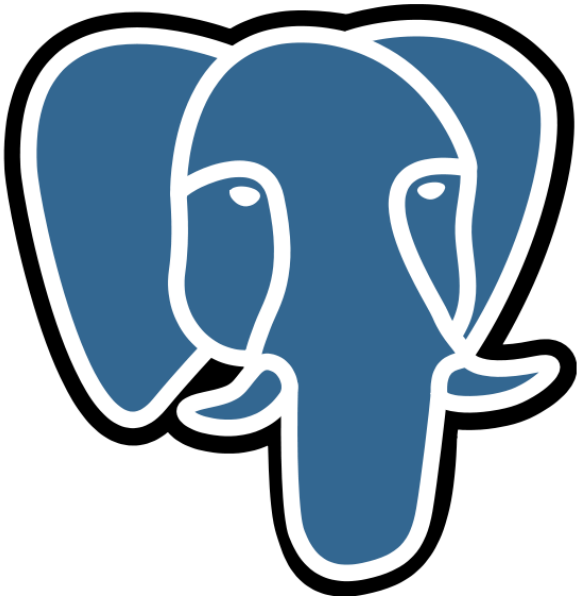
```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM lignes_commandes
                                  WHERE quantite BETWEEN 1 AND 4;

                          QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on lignes_commandes
            (cost=26538.09..87497.63 rows=1250503 width=74)
            (actual time=206.705..580.854 rows=1254886 loops=1)
   Recheck Cond: ((quantite >= 1) AND (quantite <= 4))
   Heap Blocks: exact=42202
   Buffers: shared read=45633
   ->  Bitmap Index Scan on lignes_commandes_quantite_idx
            (cost=0.00..26225.46 rows=1250503 width=0)
            (actual time=194.250..194.250 rows=1254886 loops=1)
         Index Cond: ((quantite >= 1) AND (quantite <= 4))
         Buffers: shared read=3431
 Planning time: 0.271 ms
 Execution time: 648.414 ms
(9 rows)
```

Cette fois, la sélectivité est différente et le nombre d'enregistrements moins élevé. Le moteur passe donc par un parcours d'index.

Cet exemple montre qu'on indexe selon une requête et non selon une table.

2/ Indexation avancée



2.1 INDEX AVANCÉS

De nombreuses fonctionnalités d'indexation sont disponibles dans PostgreSQL :

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

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

2.2 INDEX B-TREE (RAPPELS)

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

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

  - index dédié :

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

- Rappel : un index est coûteux à maintenir !

Rappelons que l'index classique est créé comme ceci :

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

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

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

2.3 INDEX GIN

Un autre type d'index

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

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

2.3.1 GIN : définition & données non structurées

GIN : *Generalized Inverted iNdex*

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

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

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

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

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

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

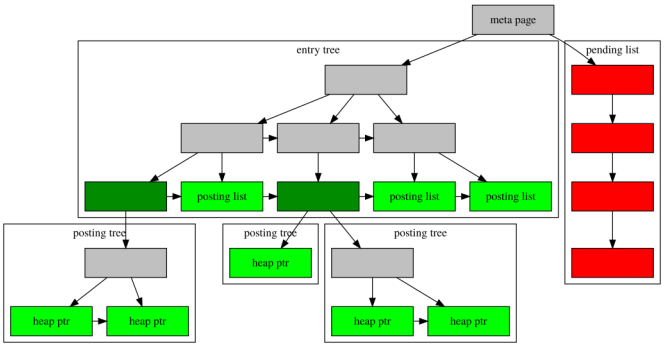


Figure 2/ .1: Schéma des index GIN (documentation officielle de PostgreSQL, licence PostgreSQL)

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

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

2.3.2 GIN et les tableaux

Quels tableaux contiennent une valeur donnée ?

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

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

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

- Champs concaténés : transformer en tableaux

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

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

GIN et champ structuré :

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

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

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

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

avec ce plan :

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

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

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

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

Le plan devient :

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

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

GIN et tableau :

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

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

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

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

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

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

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

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

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

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

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

pour un résultat beaucoup plus efficace :

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

2.3.3 GIN pour les JSON et les textes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.4 GIN & données scalaires

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

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

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

Les index GIN permettent un fonctionnement sensiblement équivalent au *bitmap* : chaque valeur indexable contient la liste des enregistrements répondant au critère, et cette liste a l'intérêt d'être compressée.

Par exemple, sur une base créée avec pgbench, de taille 100, avec les options par défaut :

```
CREATE EXTENSION btree_gin ;

CREATE INDEX pgbench_accounts_gin_idx on pgbench_accounts USING gin (bid);

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

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

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

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

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

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

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


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

Par contre les tailles des index GIN sont généralement inférieures s'il y a peu de valeurs distinctes (une centaine ici) :

```
pgbench_300=# \di+ pgbench_accounts_*
                   Liste des relations
 Schéma |            Nom             | … | Méthode d'accès | Taille | Description
--------+----------------------------+---+-----------------+--------+-------------
 public | pgbench_accounts_btree_idx | … | gin             | 66 MB  |
 public | pgbench_accounts_bid_idx1  | … | btree           | 12 MB  |
```

L'index GIN est donc environ 5 fois plus compact que le B-tree dans cet exemple simple. Ce ne serait pas le cas avec des valeurs toutes différentes.

Avant la version 13, la différence de taille est encore plus importante car les index B-tree ne disposent pas encore de la déduplication des clés.

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

2.3.5 GIN : mise à jour

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

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

L'option `fastupdate` permet une mise à jour bien plus rapide. Elle est activée par défaut. PostgreSQL stocke alors les mises à jour de l'index dans une *pending list* qui est intégrée en bloc, notamment lors d'un VACUUM. Sa taille peut être modifiée par le paramètre `gin_pending_list_limit`, par défaut à 4 Mo, et au besoin surchargeable sur chaque index (avant la 9.5, la limite était `work_mem`).

L'inconvénient de cette technique est que le temps de réponse de l'index devient instable : certaines recherches peuvent être très rapides et d'autres très lentes. Le seul moyen d'accélérer ces recherches est de désactiver cette option. Cela permet en plus de diminuer drastiquement les écritures dans les journaux de transactions en cas de mises à jour massives. Mais cela a un gros impact : les mises à jour de l'index sont bien plus lentes.

Il faut donc faire un choix. Si on conserve l'option `fastupdate`, il faut surveiller la fréquence de passage de l'autovacuum. À partir de la version 9.6, l'appel manuel à la fonction `gin_clean_pending_list()` est une autre option.

Pour les détails, voir la documentation[5].

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

2.4 INDEX GIST

GiST : *Generalized Search Tree*

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

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

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

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

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

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

2.4.1 GiST : cas d'usage

- Indexer à peu près n'importe quoi
  - géométriques (PostGIS !)
  - intervalles, adresses IP, FTS…
- Multi-colonnes dans n'importe quel ordre

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

Types natifs :

Il est disponible pour les types natifs suivants :

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

Multi colonnes :

GiST est aussi intéressant si on a besoin d'indexer plusieurs colonnes sans trop savoir dans quel ordre on va les accéder. On peut faire un index GiST multi-colonnes, et voir si ses performances sont satisfaisantes.

Pour indexer des scalaires, il faudra utiliser l'extension `btree_gist`.

```
CREATE EXTENSION IF NOT EXISTS btree_gist ;

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

CREATE INDEX ON demo_gist USING gist (i,j,k,l);

VACUUM ANALYZE demo_gist ;
```

La table pèse 50 Mo, et l'index GiST 70 Mo.

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT * FROM demo_gist WHERE i=17 ;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gist (actual time=6.027..21.000 rows=27027 loops=1)
   Recheck Cond: (i = 17)
   Heap Blocks: exact=6370
   Buffers: shared hit=6834
   ->  Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=5.109..5.109
↪  rows=27027 loops=1)
         Index Cond: (i = 17)
         Buffers: shared hit=464
 Planning:
   Buffers: shared hit=17
 Planning Time: 0.100 ms
 Execution Time: 22.143 ms

EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT * FROM demo_gist WHERE k=17 ;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gist (actual time=24.769..32.570 rows=10310 loops=1)
   Recheck Cond: (k = 17)
   Heap Blocks: exact=6370
```

```
   Buffers: shared hit=10096
   ->  Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=23.736..23.736
↪  rows=10310 loops=1)
         Index Cond: (k = 17)
         Buffers: shared hit=3726
 Planning Time: 0.061 ms
 Execution Time: 33.014 ms
```

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

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_gist (actual time=3.501..3.991 rows=83 loops=1)
   Recheck Cond: ((j = 17) AND (l = 17))
   Heap Blocks: exact=83
   Buffers: shared hit=436
   ->  Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=3.485..3.486 rows=83
↪  loops=1)
         Index Cond: ((j = 17) AND (l = 17))
         Buffers: shared hit=353
 Planning Time: 0.476 ms
 Execution Time: 4.260 ms
```

Un index B-tree sur ces mêmes 4 colonnes serait certes plus rapide à créer, plus petit, et au moins aussi efficace si la recherche porte au moins sur le premier champ, mais il serait inutilisable ou peu efficace si la recherche ne concerne que les autres champs.

Les index GiST peuvent donc être très intéressants dans les recherches multicritères.

Mais encore… :

Nous verrons enfin que les index GiST sont en outre utilisés par :

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

2.4.2 GiST & KNN

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

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

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

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

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

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

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

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

```
CREATE TABLE mes_points (p point);

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

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

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

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

```
    p    |     ?column?
---------+------------------
 (18,37) |                1
 (18,35) |                1
```

```
 (16,36) |                2
 (16,35) | 2.23606797749979
```

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

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

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

2.4.3 GiST & Contraintes d'exclusion

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

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

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

  - Exemple : réservations de salles

Premiers exemples :

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

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

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

On peut également poser une contrainte unique sur plusieurs colonnes :

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

Intérêt :

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

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

Exemples :

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

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

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

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

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

CREATE EXTENSION btree_gist ;

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

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

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

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

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

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

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

Autres fonctionnalités :

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

2.5 GIN, GIST & PG_TRGM

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

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

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

- Indexation (GIN ou GiST) :

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

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

```
 SELECT show_trgm('hello');

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

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

```
CREATE EXTENSION pg_trgm;

CREATE TABLE test_trgm (text_data text);

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

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

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

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

```
 hello
 hello everybody
```

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

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

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

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

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

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

2.6 INDEX BRIN

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

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

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

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

2.6.0.1 Exemple

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

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

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

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

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

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

On peut consulter le contenu de cet index[6], et constater que chacune de ses entrées liste les valeurs de age par paquet de 128 blocs (cette valeur peut se changer) :

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

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

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

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

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

Testons une requête avec uniquement ce petit index BRIN :

```
DROP INDEX brin_demo_btree_idx ;

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

```
                                QUERY PLAN
---------------------------------------------------------------------------
```

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

```
 Aggregate (actual time=4.838..4.839 rows=1 loops=1)
   Buffers: shared hit=130
   ->  Bitmap Heap Scan on brin_demo (actual time=0.241..3.530 rows=22212 loops=1)
         Recheck Cond: (age = 87)
         Rows Removed by Index Recheck: 6716
         Heap Blocks: lossy=128
         Buffers: shared hit=130
         ->  Bitmap Index Scan on brin_demo_brin_idx (actual time=0.024..0.024
↪  rows=1280 loops=1)
               Index Cond: (age = 87)
               Buffers: shared hit=2
 Planning:
   Buffers: shared hit=5
 Planning Time: 0.084 ms
 Execution Time: 4.873 ms
```

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

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

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

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

VACUUM ANALYZE brin_demo ;

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

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

```
         67 |   8448 |      1 | f          | f          | f           | {86 .. 88}
         68 |   8576 |      1 | f          | f          | f           | {88 .. 89}
         69 |   8704 |      1 | f          | f          | f           | {89 .. 90}
         70 |   8832 |      1 | f          | f          | f           | {1 .. 90}

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

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

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

Pour rendre son intérêt à l'index, il faut reconstruire la table avec les données dans le bon ordre avec la commande CLUSTER[7]. Hélas, c'est une opération au moins aussi lourde et bloquante qu'un VACUUM FULL. De plus, le tri de la table ne peut se faire par l'index BRIN, et il faut recréer un index B-tree au moins le temps de l'opération.

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

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

On revient alors à la situation de départ.

2.6.0.2 Utilité d'un index BRIN

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

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

- que les données soit naturellement triées dans la table, et le restent (série temporelle, décisionnel avec imports réguliers…) ;
- que la table soit à « insertion seule » pour conserver l'ordre physique ;
- ou que l'on ait la disponibilité nécessaire pour reconstruire régulièrement la table avec CLUSTER et éviter que les performances se dégradent au fil du temps.

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

2.6.0.3 Index BRIN sur clé composée

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

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

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

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

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

Pour voir son contenu :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
SET enable_seqscan TO on ;

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

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

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

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

```
DROP INDEX test_brin_idx;

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

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

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

2.6.0.4 Maintenance et consultation

Les lignes ajoutées à la table après la création de l'index ne sont pas forcément intégrées au résumé tout de suite. Il faut faire attention à ce que le VACUUM passe assez souvent[8]. Cela dit, la maintenance d'un BRIN lors d'écritures est plus légères qu'un gros B-tree.

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

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

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

Pour consulter la répartition des valeurs comme nous l'avons fait plus haut, il faut utiliser pageinspect[9]. Pour une table `brin_demo` dix fois plus grosse, et des plages de 16 blocs :

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

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

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

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

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

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

2.6.0.5 Plus d'informations sur les BRIN

- Conférence d'Adrien Nayrat au PGDay France 2016 (Lille, 31 mai 2016) : vidéo[10], texte[11].
- Documentation officielle[12].

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

2.7 INDEX HASH

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

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

Une conséquence de ce principe est qu'il est impossible de parcourir des plages de valeurs, seule l'égalité **exacte** à un critère peut être recherchée. Cet exemple utilise la base du projet Gutenberg[13] :

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

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

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

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

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

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

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

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

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

2.8 OUTILS

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

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

Tous les outils suivants sont disponibles dans les paquets diffusés par le PGDG sur yum.postgresql.org[14] ou apt.postgresql.org[15].

2.8.1 Identifier les requêtes

- pgBadger
- pg_stat_statements
- PoWA

Pour identifier les requêtes les plus lentes, et donc potentiellement nécessitant une réécriture ou un nouvel index, pgBadger[16] permet d'analyser les logs une fois ceux-ci configurés pour tracer toutes les requêtes. Des exemples figurent dans notre formation DBA1[17].

Pour une vision cumulative, voire temps réel, de ces requêtes, l'extension `pg_stat_statements`, fournie avec les « contrib » de PostgreSQL, permet de garder trace des N requêtes les plus fréquemment exécutées, et calcule le temps d'exécution total de chacune d'entre elles, ainsi que les accès au cache de PostgreSQL ou au système de fichiers. Son utilisation est détaillée dans notre module X2[18].

Le projet PoWA[19] exploite ces statistiques en les historisant, et en fournissant une interface web permettant de les exploiter.

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

2.8.2 Identifier les prédicats et des requêtes liées

- Extension pg_qualstats
  - avec PoWa

Pour identifier les prédicats (clause `WHERE` ou condition de jointure à identifier en priorité), l'extension pg_qualstats[20] permet de pousser l'analyse offerte par pg_stat_statements au niveau du prédicat lui-même. Ainsi, on peut détecter les requêtes filtrant sur les mêmes colonnes, ce qui peut aider notamment à déterminer des index multi-colonnes ou des index partiels.

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

2.8.3 Extension HypoPG

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

Cette extension est disponible sur GitHub[21] et dans les paquets du PGDG. Il existe trois fonctions principales et une vue :

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

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

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

[20] https://github.com/powa-team/pg_qualstats
[21] https://github.com/HypoPG/hypopg

```
CREATE EXTENSION hypopg;

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

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

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

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

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

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

SELECT * FROM hypopg_list_indexes;

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

SELECT * FROM hypopg_reset();

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

(1 row)

CREATE INDEX ON employes_big(prenom);

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

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

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

2.8.4 Étude des index à créer

- PoWA peut utiliser HypoPG

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

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

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

2.9 QUIZ

https://dali.bo/j5_quiz

2.10 TRAVAUX PRATIQUES

2.10.1 Indexation de motifs avec les varchar_patterns et pg_trgm

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

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

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

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

opérateur	fonction
!~*	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 ».

2.10.2 Index GIN comme bitmap

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

Ce TP utilise la base de données **tpc**. La base **tpc** peut être téléchargée depuis https://dali.bo/tp_tpc (dump de 31 Mo, pour 267 Mo sur le disque au final). Auparavant créer les utilisateurs depuis le script sur https://dali.bo/tp_tpc_roles.

```
$ psql < tpc_roles.sql           # Exécuter le script de création des rôles
$ createdb --owner tpc_owner tpc # Création de la base
$ pg_restore -d tpc tpc.dump     # Une erreur sur un schéma 'public' existant est
 ↪  normale
```

Les mots de passe sont dans le script. Pour vous connecter :

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

Créer deux index sur `lignes_commandes(quantite)` :

- un de type B-tree
- et un GIN.
- puis deux autres sur `lignes_commandes(fournisseur_id)`.

Comparer leur taille.

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

2.10.3 Index GIN et critères multicolonnes

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

Créer une table avec 4 colonnes de 50 valeurs :

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

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

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

2.10.4 HypoPG

But : Créer un index hypothétique avec HypoPG

Pour la clarté des plans, désactiver le JIT.

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

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

(Ne pas oublier VACUUM ANALYZE.)

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

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

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

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

Créer un index fonctionnel hypothétique pour faciliter la requête suivante :

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

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

2.11 TRAVAUX PRATIQUES (SOLUTIONS)

2.11.1 Indexation de motifs avec les varchar_patterns et pg_trgm

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
                               QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using textes_reverse_idx on textes
                              (cost=0.56..377770.76 rows=104728 width=123)
                         (actual time=0.083..0.098 rows=2 loops=1)
   Index Cond: ((reverse(contenu) ~>=~ 'erviv tE'::text)
```

```
           AND (reverse(contenu) ~<~ 'erviv tF'::text))
   Filter: (reverse(contenu) ~~ 'erviv tE%'::text)
 Planning Time: 1.903 ms
 Execution Time: 0.421 ms
```

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

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

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

Pour installer l'extension `pg_trgm` :

```
CREATE EXTENSION pg_trgm;
```

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

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

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

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

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

Puis insensible à la casse :

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

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=77.01..6479.68 rows=1679 width=123)
                     (actual time=13.135..23.145 rows=1214 loops=1)
   Recheck Cond: (contenu ~~* '%Valjean%'::text)
   Heap Blocks: exact=353
   ->  Bitmap Index Scan on idx_textes_trgm
```

```
                                  (cost=0.00..76.59 rows=1679 width=0)
                        (actual time=12.779..12.779 rows=1214 loops=1)
         Index Cond: (contenu ~~* '%Valjean%'::text)
 Planning Time: 2.047 ms
 Execution Time: 23.444 ms
```

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

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

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

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

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

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

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

2.11.2 Index GIN comme bitmap

Ce TP utilise la base **magasin**. La base **magasin** peut être téléchargée depuis https://dali.bo/tp_magasin (dump de 96 Mo, pour 667 Mo sur le disque au final). Elle s'importe de manière très classique (une

erreur sur le schéma **public** déjà présent est normale), ici dans une base nommée aussi `magasin` :

```
curl -L https://dali.bo/tp_magasin -o magasin.dump
pg_restore -d magasin  magasin.dump
```

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

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

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

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

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

CREATE EXTENSION btree_gin;

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

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

> Comparer leur taille.

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

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

```
               indexname               | pg_size_pretty
---------------------------------------+----------------
 lignes_commandes_fournisseur_id_btree | 22 MB
 lignes_commandes_fournisseur_id_gin   | 15 MB
 lignes_commandes_pkey                 | 94 MB
 lignes_commandes_quantite_btree       | 21 MB
 lignes_commandes_quantite_gin         | 3848 kB
```

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

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

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

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

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

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

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

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

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

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

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

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

```
 Planning:
   Buffers: shared hit=13
 Planning Time: 0.198 ms
 Execution Time: 305.030 ms
```

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

2.11.3 Index GIN et critères multicolonnes

> Créer une table avec 4 colonnes de 50 valeurs :

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

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

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

```
VACUUM ijkl;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
DROP INDEX ijkl_gin ;

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

                              QUERY PLAN
----------------------------------------------------------------------
 Gather (actual time=34.861..713.474 rows=2500 loops=1)
```

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

2.11.4 HypoPG

> Pour la clarté des plans, désactiver le JIT.

```
SET jit TO off ;
```

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

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

VACUUM ANALYZE log10 ;
```

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

> - On se demande si créer un index sur `i`, `(i,j)` ou `(j,i)` serait utile pour les deux requêtes suivantes :
>
> ```
> SELECT i, min(j), max(j) FROM log10 GROUP BY i ;
> SELECT max(j) FROM log10 WHERE i = 6 ;
> ```
>
> - Installer l'extension hypopg (paquets `hypopg_14` ou `postgresql-14-hypopg`).
> - Créer des index hypothétiques (y compris un partiel) et choisir un seul index.

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

```
sudo dnf install hypopg_14
```

Sur Debian et dérivés :

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

Puis installer l'extension dans la base concernée :

```
CREATE EXTENSION hypopg;
```

Création des différents index hypothétiques qui pourraient servir :

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

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

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

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

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

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

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

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

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

```
SELECT * FROM hypopg_list_indexes;
```

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

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

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=1000.08..392727.62 rows=5 width=20)
   Group Key: i
   ->  Gather Merge  (cost=1000.08..392727.49 rows=10 width=20)
         Workers Planned: 2
         ->  Partial GroupAggregate  (cost=0.06..391726.32 rows=5 width=20)
               Group Key: i
               ->  Parallel Index Only Scan
                   using <78054>btree_log10_i_j on log10
                   (cost=0.06..357004.01 rows=4629634 width=12)
```

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

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.08..0.09 rows=1 width=8)
   InitPlan 1 (returns $0)
```

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

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

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

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

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

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

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

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

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

                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate      (cost=564931.67..564931.68 rows=1 width=8)
                                          (actual rows=1 loops=1)
   ->  Gather  (cost=564931.46..564931.67 rows=2 width=8)
                                 (actual rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=563931.46..563931.47 rows=1 width=8)
                                               (actual rows=1 loops=3)
               ->  Parallel Seq Scan on log10
                              (cost=0.00..562921.43 rows=404013 width=8)
                                    (actual rows=333333 loops=3)
                     Filter: (i = 6)
                     Rows Removed by Filter: 3370370
 Planning Time: 0.414 ms
 Execution Time: 2322.879 ms

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

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

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

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

Si on simule la présence de cet index fonctionnel :

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

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

on constate que l'optimiseur le néglige.

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

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

on constate qu'il est alors utilisé :

```
Gather  (cost=3243.57..343783.72 rows=119630 width=4) (actual rows=112233 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Bitmap Heap Scan on log10
                          (cost=2243.57..330820.72 rows=49846 width=4)
                                    (actual rows=37411 loops=3)
        Recheck Cond: (mod(j, '99'::bigint) = 55)
        Rows Removed by Index Recheck: 470869
        Heap Blocks: exact=18299 lossy=23430
        ->  Bitmap Index Scan on log10_mod_idx
                              (cost=0.00..2213.66 rows=119630 width=0)
                                    (actual rows=112233 loops=1)
              Index Cond: (mod(j, '99'::bigint) = 55)
Planning Time: 1.216 ms
Execution Time: 541.668 ms
```

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

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

NB : on peut penser aussi à un index couvrant :

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

en fonction des requêtes réelles à optimiser.

3/ Extensions PostgreSQL pour la performance

3.1 PRÉAMBULE

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

3.2 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`.

3.3 PG_STAT_STATEMENTS

Capture en temps réel des requêtes :

- Vue, en mémoire partagée (volumétrie contrôlée)
- Par requête
  - nombre d'exécution, 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)
  - Pas d'échantillonnage, seulement des compteurs cumulés
- Installation :
  - `shared_preload_libraries = 'pg_stat_statements'`

`pg_stat_statements` capture, à chaque exécution de requête, tous les compteurs ci-dessus. 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.

`pg_stat_statements` fournit une vue (du même nom), qui retourne un instantané des compteurs au moment de l'interrogation, ainsi qu'une fonction `pg_stat_statements_reset`. Deux méthodes 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 intervalle régulier le contenu de `pg_stat_statements` et visualiser les changements dans les compteurs : le projet PoWA[1] a été développé à cet effet.

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'
```

Dès lors que l'extension est chargée en mémoire, la capture des compteurs est enclenchée, sauf si le paramètre `pg_stat_statements.track` est positionnée 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.

[1] https://powa.readthedocs.io/en/latest/

Même si la vue porte sur toutes les bases de l'instance, son installation et celle des fonctions utilitaires impliquent l'installation de l'extension dans la base où l'on veut les utiliser :

```
CREATE EXTENSION pg_stat_statements ;
```

`pg_stat_statements` possède quelques paramètres[2]. 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].

3.3.1 pg_stat_statements : exemple 1

Requêtes les plus longues en temps cumulé :

```
SELECT r.rolname, d.datname, s.calls, s.total_exec_time,
       s.calls / s.total_exec_time 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;
```

3.3.2 pg_stat_statements : exemple 2

Requêtes les plus fréquemment appelées :

```
SELECT r.rolname, d.datname, s.calls, s.total_exec_time,
       s.calls / s.total_exec_time 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;
```

[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

3.4 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'
```

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`.

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

```
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/

3.5 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
```

3.6 PG_PREWARM

- Charge des blocs en cache :

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

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

- 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 :

À partir de la version 11, 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

3.7 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] (version 2 et 3) ;
- 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.

3.7.1 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

3.8 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 du tri 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 :

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

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

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 pré-calculés 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

3.9 QUIZ

https://dali.bo/x2_quiz

3.10 TRAVAUX PRATIQUES

3.10.1 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 charger dans une base préexistante avec `pg_restore` et créera juste une table nommée `textes`.

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 ».

3.10.2 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`.

3.10.3 pg_stat_statements

But : Analyser les performances des requêtes avec pg_stat_statements

La base **cave** peut être téléchargée depuis https://dali.bo/tp_cave (dump de 2,6 Mo, pour 71 Mo sur le disque au final) et importée ainsi :

```
$ psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
$ psql -c "CREATE DATABASE cave OWNER caviste"
$ pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

- 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+`).

- Exécuter une requête coûteuse (la récupération du nombre de bouteilles de chaque appellation en stock par exemple).
- Dans la vue `pg_stat_statements`, récupérer les 5 requêtes les plus gourmandes en temps cumulé sur l'instance.

- Vérifier que le serveur soit capable d'activer la mesure de la durée des entrées-sorties avec `pg_test_timing`.

- Activer la mesure des temps d'exécution des entrées-sorties, redémarrer PostgreSQL (pour vider son cache), remettre la vue `pg_stat_statements` à 0, et ré-exécuter la requête « lourde » précédente.

3.10.4 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 page-web('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`.

3.10.5 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 charger dans une base préexistante avec `pg_restore` et créera juste une table nommée `textes`.

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.
- 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 ;
```

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

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 ;
```

3.10.6 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.

3.11 TRAVAUX PRATIQUES (SOLUTIONS)

3.11.1 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
```

3.11.2 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…
```

3.11.3 pg_stat_statements

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

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

Redémarrer PostgreSQL.

Créer l'extension :

```
CREATE EXTENSION 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
```

- Exécuter une requête coûteuse (la récupération du nombre de bouteilles de chaque appellation en stock par exemple).
- Dans la vue `pg_stat_statements`, récupérer les 5 requêtes les plus gourmandes en temps cumulé sur l'instance.

```
SELECT appellation.libelle,
       sum(stock.nombre)
FROM appellation
JOIN vin ON appellation.id=vin.appellation_id
JOIN stock ON vin.id=stock.vin_id
GROUP BY appellation.libelle;
```

```
SELECT * FROM pg_stat_statements ORDER BY total_exec_time desc LIMIT 5;
```

La colonne `total_exec_time` apparaît en version 13. Elle avait auparavant pour nom `total_time`.

- Vérifier que le serveur soit capable d'activer la mesure de la durée des entrées-sorties avec `pg_test_timing`.

```
$ /usr/pgsql-14/bin/pg_test_timing
```

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 34.23 nsec
Histogram of timing durations:
< usec   % of total      count
     1     96.58665   84647529
     2      3.41157    2989865
     4      0.00044        387
     8      0.00080        702
    16      0.00052        455
    32      0.00002         16
    64      0.00000          1
   128      0.00000          1
   256      0.00000          0
   512      0.00000          1
```

Si le temps de mesure est de quelques dizaines de nanosecondes, c'est OK. Sinon, éviter de faire ce qui suit sur un serveur de production. Sur une machine de formation, ce n'est pas un problème.

- Activer la mesure des temps d'exécution des entrées-sorties, redémarrer PostgreSQL (pour vider son cache), remettre la vue `pg_stat_statements` à 0, et ré-exécuter la requête « lourde » précédente.

Dans le fichier `postgresql.conf`, positionner :

```
track_io_timing = on
```

Redémarrer PostgreSQL.

Exécuter :

```
SELECT pg_stat_statements_reset() ;
```

Ré-exécuter la requête, et vérifier dans `pg_stat_statements` que les colonnes `blk_read_time` et `blk_write_time` sont maintenant mises à jour.

3.11.4 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
```

3.11.5 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.

3.11.6 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. Noter que tous les accès se font en mémoire (*shared hits*), y compris le tri des 765 Mo, le cas est donc idéal. L'essentiel du temps est perdu dans l'opération `GroupAggregate`.

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 turnover 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
```

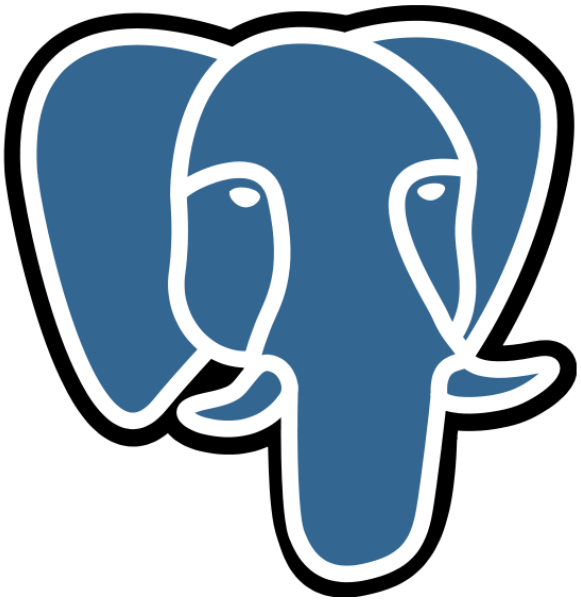
```
           hll_cardinality(hll_union_agg(passagers_hll)
                    OVER (ORDER BY mois ASC ROWS 2 PRECEDING) )::bigint
         ELSE null END  AS nb_d_passagers_3_mois_glissants
FROM voyages_mois
;
```

```
    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.

4/ Partitionnement sous PostgreSQL



- Ses principes et intérêts
- Historique
- Les différents types

4.1 PRINCIPE & INTÉRÊTS DU PARTITIONNEMENT

- Faciliter la maintenance de gros volumes
  - VACUUM (FULL), réindexation, déplacements, sauvegarde logique…
- Performances
  - parcours complet sur de plus petites tables
  - statistiques par partition plus précises
  - purge par partitions entières
  - pg_dump parallélisable
  - tablespaces différents (données froides/chaudes)
- Attention à la maintenance sur le code

Maintenir de très grosses tables peut devenir fastidieux, voire impossible : `VACUUM FULL` trop long, espace disque insuffisant, autovacuum pas assez réactif, réindexation interminable… Il est aussi aberrant de conserver beaucoup de données d'archives dans des tables lourdement sollicitées pour les données récentes.

Le partitionnement consiste à séparer une même table en plusieurs sous-tables (partitions) manipulables en tant que tables à part entière.

Maintenance

La maintenance s'effectue sur les partitions plutôt que sur l'ensemble complet des données. En particulier, un `VACUUM FULL` ou une réindexation peuvent s'effectuer partition par partition, ce qui permet de limiter les interruptions en production, et lisser la charge. `pg_dump` ne sait pas paralléliser la sauvegarde d'une table volumineuse et non partitionnée, mais parallélise celle de différentes partitions d'une même table.

C'est aussi un moyen de déplacer une partie des données dans un autre *tablespace* pour des raisons de place, ou pour déporter les parties les moins utilisées de la table vers des disques plus lents et moins chers.

Parcours complet de partitions

Certaines requêtes (notamment décisionnelles) ramènent tant de lignes, ou ont des critères si complexes, qu'un parcours complet de la table est souvent privilégié par l'optimiseur.

Un partitionnement, souvent par date, permet de ne parcourir qu'une ou quelques partitions au lieu de l'ensemble des données. C'est le rôle de l'optimiseur de choisir la partition (*partition pruning*), par exemple celle de l'année en cours, ou des mois sélectionnés.

Suppression des partitions

La suppression de données parmi un gros volume peut poser des problèmes d'accès concurrents ou de performance, par exemple dans le cas de purges.

En configurant judicieusement les partitions, on peut résoudre cette problématique en supprimant une partition (`DROP TABLE nompartition ;`), ou en la *détachant* (`ALTER TABLE table_partitionnee DETACH PARTITION nompartition ;`) pour l'archiver (et la réattacher au besoin) ou la supprimer ultérieurement.

D'autres optimisations sont décrites dans ce billet de blog d'Adrien Nayrat[1] : statistiques plus précises au niveau d'une partition, réduction plus simple de la fragmentation des index, jointure par rapprochement des partitions…

La principale difficulté d'un système de partitionnement consiste à partitionner avec un impact minimal sur la maintenance du code par rapport à une table classique.

[1] https://blog.anayrat.info/2021/09/01/cas-dusages-du-partitionnement-natif-dans-postgresql/

4.2 PARTITIONNEMENT APPLICATIF

- Intégralement géré au niveau applicatif
- Complexité pour le développeur
- Intégrité des liens entre les données ?
- Réinvention de la roue

L'application peut gérer le partitionnement elle-même, par exemple en créant des tables numérotées par mois, année… Le moteur de PostgreSQL ne voit que des tables classiques et ne peut assurer l'intégrité entre ces données.

C'est au développeur de réinventer la roue : choix de la table, gestion des index… La lecture des données qui concerne plusieurs tables peut devenir délicate.

Ce modèle extrêmement fréquent est bien sûr à éviter.

4.3 MÉTHODES DE PARTITIONNEMENT INTÉGRÉES À POSTGRESQL

- Partitionnement par héritage (historique, < v10)
- Partitionnement déclaratif (>=v10, préférer v13+)

Un partitionnement entièrement géré par le moteur, n'existe réellement que depuis la version 10 de PostgreSQL. Il a été grandement amélioré en versions 11 et 12, en fonctionnalités comme en performances.

Jusqu'à PostgreSQL 9.6 n'existaient que le partitionnement dit par héritage, nettement moins flexible, et bien sûr le partitionnement géré intégralement par l'applicatif.

4.4 PARTITIONNEMENT PAR HÉRITAGE

- Syntaxe :

```
CREATE TABLE primates (debout boolean) INHERITS (mammiferes) ;
```

- Table mère :
  - définie normalement
- Tables filles :
  - héritent des propriétés de la table mère
  - mais pas des contraintes, index et droits
  - colonnes supplémentaires possibles
- Insertion applicative ou par trigger (lent !)

PostgreSQL permet de créer des tables qui héritent les unes des autres.

L'héritage d'une table mère transmet les propriétés suivantes à la table fille :

- les colonnes, avec le type et les valeurs par défaut ;
- les contraintes CHECK.

Les tables filles peuvent ajouter leurs propres colonnes. Par exemple :

```
CREATE TABLE animaux (nom text PRIMARY KEY);

INSERT INTO animaux VALUES ('Éponge');
INSERT INTO animaux VALUES ('Ver de terre');


CREATE TABLE cephalopodes (nb_tentacules integer)
INHERITS (animaux);

INSERT INTO cephalopodes VALUES ('Poulpe', 8);

CREATE TABLE vertebres (
    nb_membres integer
)
INHERITS (animaux);

CREATE TABLE tetrapodes () INHERITS (vertebres);

ALTER TABLE ONLY tetrapodes
ALTER COLUMN nb_membres
SET DEFAULT 4;
```

```
CREATE TABLE poissons (eau_douce boolean)
INHERITS (tetrapodes);

INSERT INTO poissons (nom, eau_douce) VALUES ('Requin', false);
INSERT INTO poissons (nom, nb_membres, eau_douce) VALUES ('Anguille', 0, false);
```

La table `poissons` possède les champs des tables dont elle hérite :

```
\d+ poissons
                                    Table "public.poissons"
   Column    |  Type   | Collation | Nullable | Default | Storage  | Stats target |
 ↪  Description
-------------+---------+-----------+----------+---------+----------+--------------+--
 ↪  -----------
 nom         | text    |           | not null |         | extended |              |
 nb_membres  | integer |           |          | 4       | plain    |              |
 eau_douce   | boolean |           |          |         | plain    |              |
Inherits: tetrapodes
Access method: heap
```

On peut créer toute une hiérarchie avec des branches parallèles, chacune avec ses colonnes propres :

```
CREATE TABLE reptiles (venimeux boolean)
INHERITS (tetrapodes);

INSERT INTO reptiles VALUES ('Crocodile', 4, false);
INSERT INTO reptiles VALUES ('Cobra', 0, true);

CREATE TABLE mammiferes () INHERITS (tetrapodes);

CREATE TABLE cetartiodactyles (
    cornes boolean,
    bosse boolean
)
INHERITS (mammiferes);

INSERT INTO cetartiodactyles VALUES ('Girafe', 4, true, false);
INSERT INTO cetartiodactyles VALUES ('Chameau', 4, false, true);

CREATE TABLE primates (debout boolean)
INHERITS (mammiferes);

INSERT INTO primates (nom, debout) VALUES ('Chimpanzé', false);
INSERT INTO primates (nom, debout) VALUES ('Homme', true);

\d+ primates

                                    Table "public.primates"
   Column    |  Type   | Collation | Nullable | Default | Storage  | Stats target |
    ↪  Description
-------------+---------+-----------+----------+---------+----------+--------------+--
 ↪  -----------
 nom         | text    |           | not null |         | extended |              |
 nb_membres  | integer |           |          | 4       | plain    |              |
 debout      | boolean |           |          |         | plain    |              |
```

```
Inherits: mammiferes
Access method: heap
```

On remarquera que la clé primaire manque. En effet, l'héritage ne transmet pas :

- les contraintes d'unicité et référentielles ;
- les index ;
- les droits.

Chaque table possède ses propres lignes :

```
SELECT * FROM poissons ;
```

```
   nom    | nb_membres | eau_douce
----------+------------+-----------
 Requin   |          4 | f
 Anguille |          0 | f
```

Par défaut une table affiche aussi le contenu de ses tables filles et les colonnes communes :

```
SELECT * FROM animaux ORDER BY 1 ;
```

```
   nom
--------------
 Anguille
 Chameau
 Chimpanzé
 Cobra
 Crocodile
 Éponge
 Girafe
 Homme
 Poulpe
 Requin
 Ver de terre
```

```
SELECT * FROM tetrapodes ORDER BY 1 ;
```

```
    nom    | nb_membres
-----------+------------
 Anguille  |          0
 Chameau   |          4
 Chimpanzé |          4
 Cobra     |          0
 Crocodile |          4
 Girafe    |          4
 Homme     |          4
 Requin    |          4
```

```
EXPLAIN SELECT * FROM tetrapodes ORDER BY 1 ;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Sort  (cost=420.67..433.12 rows=4982 width=36)
   Sort Key: tetrapodes.nom
   ->  Append  (cost=0.00..114.71 rows=4982 width=36)
         ->  Seq Scan on tetrapodes  (cost=0.00..0.00 rows=1 width=36)
```

```
        ->  Seq Scan on poissons  (cost=0.00..22.50 rows=1250 width=36)
        ->  Seq Scan on reptiles  (cost=0.00..22.50 rows=1250 width=36)
        ->  Seq Scan on mammiferes  (cost=0.00..0.00 rows=1 width=36)
        ->  Seq Scan on cetartiodactyles  (cost=0.00..22.30 rows=1230 width=36)
        ->  Seq Scan on primates  (cost=0.00..22.50 rows=1250 width=36)
```

Pour ne consulter que le contenu de la table sans ses filles :

```
SELECT * FROM ONLY animaux ;
```

```
     nom
--------------
 Éponge
 Ver de terre
```

En conséquence, on a bien affaire à des tables indépendantes. Rien n'empêche d'avoir des doublons entre la table mère et la table fille. Cela empêche aussi bien sûr la mise en place de clé étrangère, puisqu'une clé étrangère s'appuie sur une contrainte d'unicité de la table référencée. Lors d'une insertion, voire d'une mise à jour, le choix de la table cible se fait par l'application ou un trigger sur la table mère.

Il faut être vigilant à bien recréer les contraintes et index manquants ainsi qu'à attribuer les droits sur les objets de manière adéquate. L'une des erreurs les plus fréquentes est d'oublier de créer les contraintes, index et droits qui n'ont pas été transmis.

Ce type de partitionnement est un héritage des débuts de PostgreSQL, à l'époque de la mode des « bases de donnée objet ». Dans la pratique, dans les versions antérieures à la version 10, l'héritage était utilisé pour mettre en place le partitionnement. La maintenance des index, des contraintes et la nécessité d'un trigger pour aiguiller les insertions vers la bonne table fille, ne facilitaient pas la maintenance. Les performances en écritures étaient bien en-deçà des tables classiques ou du nouveau partitionnement déclaratif (voir comparaison plus bas).

Si le partitionnement par héritage fonctionne toujours sur les versions récentes de PostgreSQL, il est déconseillé pour les nouveaux développements.

4.5 PARTITIONNEMENT DÉCLARATIF

- Préférer version 13+
- Mise en place et administration simplifiées (intégrées au moteur)
- Gestion automatique des lectures et écritures
- Partitions
  - attacher/détacher une partition
  - contrainte implicite de partitionnement
  - expression possible pour la clé de partitionnement
  - sous-partitions possibles
  - partition par défaut

Le partitionnement déclaratif est le système à privilégier de nos jours. Apparu en version 10, il est à présent mûr.

Son but est de permettre une mise en place et une administration simples des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL, comme `CREATE TABLE` et `ALTER TABLE`, pour attacher (`ATTACH PARTITION`) et détacher des partitions (`DETACH PARTITION`).

Au niveau de la simplification de la mise en place par rapport à l'ancien partitionnement par héritage, on peut noter qu'il n'est pas nécessaire de créer une fonction *trigger* ni d'ajouter des *triggers* pour gérer les insertions et mises à jour. Le routage est géré de façon automatique en fonction de la définition des partitions, au besoin vers une partition par défaut. Contrairement au partitionnement par héritage, la table partitionnée ne contient pas elle-même de ligne, ce n'est qu'une coquille vide. Du fait de ce routage automatique, les durées d'insertions ne subissent pas de pénalité.

4.5.1 Partitionnement par liste

- Liste de valeurs par partition
- Clé de partitionnement forcément mono-colonne
- Syntaxe :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1) ;

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
…
```

Il est possible de partitionner une table par valeurs. Ce type de partitionnement fonctionne uniquement avec une clé de partitionnement mono-colonne.

Voici un exemple de création d'une table partitionnée et de ces partitions :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);

CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
```

Les noms des partitions sont à définir par l'utilisateur, il n'y a pas d'automatisme ni de convention particulière.

Et voici quelques insertions de données :

```
=# INSERT INTO t1 VALUES (1);
INSERT 0 1

=# INSERT INTO t1 VALUES (2);
INSERT 0 1

=# INSERT INTO t1 VALUES (5);
INSERT 0 1
```

Lors de l'insertion, les données sont correctement redirigées vers leur partition, comme le montre cette requête :

```
SELECT tableoid::regclass, * FROM t1;

 tableoid | c1 | c2
----------+----+----
 t1_a     |  1 |
 t1_a     |  2 |
 t1_b     |  5 |
```

Il est aussi possible d'interroger les partitions séparément :

```
SELECT * FROM t1_a ;

 c1 | c2
----+----
  1 |
  2 |
```

Si aucune partition correspondant à la clé insérée n'est trouvée et qu'aucune partition par défaut n'est déclarée (fonctionnalité disponible qu'à partir de la version 11), une erreur se produit.

```
=# INSERT INTO t1 VALUES (0);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

=# INSERT INTO t1 VALUES (6);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).
```

Si la clé de partitionnement d'une ligne est modifiée par un UPDATE, la ligne change automatiquement de partition (sauf en version 10, où ce n'est pas implémenté et provoque une erreur).

4.5.2 Partitionnement par intervalle

- Intervalle de valeurs par partition
- Clé de partitionnement mono- ou multi-colonnes
- Bornes « infinies » :
  - MINVALUE / MAXVALUE
- Syntaxe :

```
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);

CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
…
```

Voici un exemple de création de la table partitionnée et de deux partitions :

```
=# CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE

=# CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) to (100);
CREATE TABLE
```

```
=# CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
```

Le MAXVALUE indique la valeur maximale du type de données : `t2_2` acceptera donc tous les entiers supérieurs ou égaux à 100.

Noter que les bornes supérieures des partitions sont **exclues** ! La valeur 100 ira donc dans la seconde partition.

Lors de l'insertion, les données sont redirigées vers leur partition, s'il y en a une :

```
=# INSERT INTO t2 VALUES (0);
ERROR:  no PARTITION OF relation "t2" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

=# INSERT INTO t2 VALUES (10, 'dix');
INSERT 0 1

=# INSERT INTO t2 VALUES (100, 'cent');
INSERT 0 1

=# INSERT INTO t2 VALUES (10000, 'dix mille');
INSERT 0 1

=# SELECT * FROM t2 ;

  c1   |    c2
-------+-----------
    10 | dix
   100 | cent
 10000 | dix mille
(3 lignes)

=# SELECT * FROM t2_2 ;

  c1   |    c2
-------+-----------
   100 | cent
 10000 | dix mille
(2 lignes)
```

La colonne système `tableoid` permet de connaître la partition d'où provient une ligne :

```
=# SELECT ctid, tableoid::regclass, * FROM t2 ;

 ctid  | tableoid |  c1   |    c2
-------+----------+-------+-----------
 (0,1) | t2_1     |    10 | dix
 (0,1) | t2_2     |   100 | cent
 (0,2) | t2_2     | 10000 | dix mille
```

4.5.3 Partitionnement par hachage

- À partir de la version 11
- Hachage de valeurs par partition
  - indiquer un modulo et un reste
- Clé de partitionnement mono- ou multi-colonnes
- Syntaxe :

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);

CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder
 ↪  0);
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder
 ↪  1);
CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder
 ↪  2);
```

Voici comment partitionner par hachage une table en trois partitions :

```
=# CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);
CREATE TABLE

=# CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 0);
CREATE TABLE

=# CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 1);
CREATE TABLE

=# CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 2);
CREATE TABLE
```

Une grosse insertion de données répartira les données de manière équitable entre les différentes partitions :

```
=# INSERT INTO t3 SELECT generate_series(1, 1000000);
INSERT 0 1000000

=# SELECT relname, count(*)
FROM t3
JOIN pg_class ON t3.tableoid=pg_class.oid
GROUP BY 1;

 relname | count
---------+--------
 t3_1    | 333263
 t3_2    | 333497
 t3_3    | 333240
```

4.5.4 Clé de partitionnement multi-colonnes

- Clé sur plusieurs colonnes acceptée
  - si partitionnement par intervalle ou hash, pas pour liste
- Syntaxe :

```
CREATE TABLE t1(c1 integer, c2 text, c3 date)
PARTIONNED BY RANGE (c1, c3) ;

CREATE TABLE t1_a PARTITION OF t1
FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11') ;
…
```

Quand on utilise le partitionnement par intervalle, il est possible de créer les partitions en utilisant plusieurs colonnes.

On profitera de l'exemple ci-dessous pour montrer l'utilisation conjointe de tablespaces différents.

Commençons par créer les tablespaces :

```
=# CREATE TABLESPACE ts0 LOCATION '/tablespaces/ts0';
CREATE TABLESPACE

=# CREATE TABLESPACE ts1 LOCATION '/tablespaces/ts1';
CREATE TABLESPACE

=# CREATE TABLESPACE ts2 LOCATION '/tablespaces/ts2';
CREATE TABLESPACE

=# CREATE TABLESPACE ts3 LOCATION '/tablespaces/ts3';
CREATE TABLESPACE
```

Créons maintenant la table partitionnée et deux partitions :

```
=# CREATE TABLE t2(c1 integer, c2 text, c3 date not null)
       PARTITION BY RANGE (c1, c3);
CREATE TABLE

=# CREATE TABLE t2_1 PARTITION OF t2
       FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11')
       TABLESPACE ts1;
CREATE TABLE

=# CREATE TABLE t2_2 PARTITION OF t2
       FOR VALUES FROM (100,'2017-08-11') TO (200, '2017-08-12')
       TABLESPACE ts2;
CREATE TABLE
```

La borne supérieure étant exclue, la valeur `(100, '2017-08-11')` fera donc partie de la seconde partition.

Si les valeurs sont bien comprises dans les bornes :

```
=# INSERT INTO t2 VALUES (1, 'test', '2017-08-10');
INSERT 0 1

=# INSERT INTO t2 VALUES (150, 'test2', '2017-08-11');
INSERT 0 1
```

Si la valeur pour `c1` est trop petite :

```
=# INSERT INTO t2 VALUES (0, 'test', '2017-08-10');
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (0, 2017-08-10).
```

Si la valeur pour `c3` (colonne de type date) est antérieure :

```
=# INSERT INTO t2 VALUES (1, 'test', '2017-08-09');
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (1, 2017-08-09).
```

Les valeurs spéciales `MINVALUE` et `MAXVALUE` permettent de ne pas indiquer de valeur de seuil limite. Les partitions `t2_0` et `t2_3` pourront par exemple être déclarées comme suit et permettront d'insérer les lignes qui étaient ci-dessus en erreur.

```
=# CREATE TABLE t2_0 PARTITION OF t2
       FOR VALUES FROM (MINVALUE, MINVALUE) TO (1,'2017-08-10')
       TABLESPACE ts0;

=# CREATE TABLE t2_3 PARTITION OF t2
       FOR VALUES FROM (200,'2017-08-12') TO (MAXVALUE, MAXVALUE)
       TABLESPACE ts3;
```

Enfin, on peut consulter la table `pg_class` afin de vérifier la présence des différentes partitions :

```
=# ANALYZE t2;

ANALYZE

=# SELECT relname,relispartition,relkind,reltuples
           FROM pg_class WHERE relname LIKE 't2%';

 relname | relispartition | relkind | reltuples
---------+----------------+---------+-----------
 t2      | f              | p       |         0
 t2_0    | t              | r       |         2
 t2_1    | t              | r       |         1
 t2_2    | t              | r       |         1
 t2_3    | t              | r       |         0
```

4.5.5 Performances en insertion

t1 (non partitionnée) :

```
INSERT INTO t1 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 7774.443 ms (00:07.774)
```

t2 (nouveau partitionnement) :

```
INSERT INTO t2 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 8062.570 ms (00:08.063)
```

t3 (ancien partitionnement par héritage) :

```
INSERT INTO t3 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 68928.431 ms (01:08.928)
```

La table *t1* est une table non partitionnée. Elle a été créée comme suit :

```
CREATE TABLE t1 (c1 integer, c2 text);
```

La table `t2` est une table partitionnée utilisant les nouvelles fonctionnalités de la version 10 de PostgreSQL :

```
CREATE TABLE t2 (c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (      0) TO ( 1000000);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (1000000) TO ( 2000000);
CREATE TABLE t2_3 PARTITION OF t2 FOR VALUES FROM (2000000) TO ( 3000000);
CREATE TABLE t2_4 PARTITION OF t2 FOR VALUES FROM (3000000) TO ( 4000000);
CREATE TABLE t2_5 PARTITION OF t2 FOR VALUES FROM (4000000) TO ( 5000000);
CREATE TABLE t2_6 PARTITION OF t2 FOR VALUES FROM (5000000) TO ( 6000000);
CREATE TABLE t2_7 PARTITION OF t2 FOR VALUES FROM (6000000) TO ( 7000000);
CREATE TABLE t2_8 PARTITION OF t2 FOR VALUES FROM (7000000) TO ( 8000000);
CREATE TABLE t2_9 PARTITION OF t2 FOR VALUES FROM (8000000) TO ( 9000000);
CREATE TABLE t2_0 PARTITION OF t2 FOR VALUES FROM (9000000) TO (10000000);
```

Enfin, la table *t3* est une table utilisant l'ancienne méthode de partitionnement :

```
CREATE TABLE t3 (c1 integer, c2 text);
CREATE TABLE t3_1 (CHECK (c1 BETWEEN       0 AND  999999)) INHERITS (t3);
CREATE TABLE t3_2 (CHECK (c1 BETWEEN 1000000 AND 1999999)) INHERITS (t3);
CREATE TABLE t3_3 (CHECK (c1 BETWEEN 2000000 AND 2999999)) INHERITS (t3);
CREATE TABLE t3_4 (CHECK (c1 BETWEEN 3000000 AND 3999999)) INHERITS (t3);
CREATE TABLE t3_5 (CHECK (c1 BETWEEN 4000000 AND 4999999)) INHERITS (t3);
CREATE TABLE t3_6 (CHECK (c1 BETWEEN 5000000 AND 5999999)) INHERITS (t3);
CREATE TABLE t3_7 (CHECK (c1 BETWEEN 6000000 AND 6999999)) INHERITS (t3);
CREATE TABLE t3_8 (CHECK (c1 BETWEEN 7000000 AND 7999999)) INHERITS (t3);
CREATE TABLE t3_9 (CHECK (c1 BETWEEN 8000000 AND 8999999)) INHERITS (t3);
```

```
CREATE TABLE t3_0 (CHECK (c1 BETWEEN 9000000 AND 9999999)) INHERITS (t3);

CREATE OR REPLACE FUNCTION insert_into() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.c1    BETWEEN       0 AND  999999 THEN
    INSERT INTO t3_1 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 1000000 AND 1999999 THEN
    INSERT INTO t3_2 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 2000000 AND 2999999 THEN
    INSERT INTO t3_3 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 3000000 AND 3999999 THEN
    INSERT INTO t3_4 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 4000000 AND 4999999 THEN
    INSERT INTO t3_5 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 5000000 AND 5999999 THEN
    INSERT INTO t3_6 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 6000000 AND 6999999 THEN
    INSERT INTO t3_7 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 7000000 AND 7999999 THEN
    INSERT INTO t3_8 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 8000000 AND 8999999 THEN
    INSERT INTO t3_9 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 9000000 AND 9999999 THEN
    INSERT INTO t3_0 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_insert_t3 BEFORE INSERT ON t3 FOR EACH ROW EXECUTE PROCEDURE
 ↪  insert_into();
```

4.5.6 Partition par défaut

- Pour le partitionnement par liste et par intervalle
- Toutes les données n'allant pas dans les partitions définies iront dans la partition par défaut

```
CREATE TABLE t2_autres PARTITION OF t2 DEFAULT ;
```

Ajouter une partition par défaut permet de ne plus avoir d'erreur au cas où une partition n'est pas définie.

En voici un exemple à partir de la table `t1` définie ci-dessus :

```
=# INSERT INTO t1 VALUES (0);
```

```
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

=# INSERT INTO t1 VALUES (6);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).

=# CREATE TABLE t1_defaut PARTITION OF t1 DEFAULT;
CREATE TABLE

=# INSERT INTO t1 VALUES (0);
INSERT 0 1

=# INSERT INTO t1 VALUES (6);
INSERT 0 1

=# SELECT tableoid::regclass, * FROM t1;

 tableoid  | c1 | c2
-----------+----+----
 t1_a      |  1 |
 t1_a      |  2 |
 t1_b      |  5 |
 t1_defaut |  0 |
 t1_defaut |  6 |
```

Comme la partition par défaut risque d'être parcourue intégralement à chaque ajout d'une nouvelle partition, il vaut mieux la garder de petite taille.

Un partitionnement par hachage ne peut posséder de table par défaut.

4.5.7 Attacher une partition

```
ALTER TABLE … ATTACH PARTITION … FOR VALUES …;
```

- La table doit préexister
- Vérification du respect de la contrainte par les données existantes
  - parcours complet de la table
  - potentiellement lent !
    * …sauf si ajout préalable d'une contrainte CHECK identique
- Si la partition par défaut a des données qui iraient dans cette partition :
  - erreur à l'ajout de la nouvelle partition
  - détacher la partition par défaut
  - ajouter la nouvelle partition
  - déplacer les données de l'ancienne partition par défaut
  - ré-attacher la partition par défaut

Ajouter une table comme partition d'une table partitionnée est possible mais cela nécessite de vérifier que la contrainte de partitionnement est valide pour toute la table attachée, et que la partition par défaut ne contient pas de données qui devraient figurer dans cette nouvelle partition.

Cela résulte en un parcours complet de la table attachée, et de la partition par défaut si elle existe, ce qui sera d'autant plus lent qu'elles sont volumineuses.

Ce peut être très coûteux en disque, mais le plus gros problème est la durée du verrou sur la table partitionnée, pendant toute cette opération. Il est donc conseillé d'ajouter une contrainte CHECK adéquate **avant** l'ATTACH : la durée du verrou sera raccourcie d'autant.

Si des lignes pour cette nouvelle partition figurent déjà dans la partition par défaut, des opérations supplémentaires sont à réaliser pour les déplacer. Ce n'est pas automatique.

4.5.8 Détacher une partition

```
ALTER TABLE … DETACH PARTITION …
```

- Simple et rapide
- Mais nécessite un verrou exclusif
  - option CONCURRENTLY (v14+)

Détacher une partition est beaucoup plus rapide qu'en attacher une. En effet, il n'est pas nécessaire de procéder à des vérifications sur les données des partitions.

Cependant, il est nécessaire d'acquérir un verrou exclusif sur la table partitionnée, ce qui peut prendre du temps si des transactions sont en cours d'exécution. L'option CONCURRENTLY (à partir de PostgreSQL 14) mitige le problème malgré quelques restrictions[2].

La partition détachée devient alors une table tout à fait classique. Elle conserve les index, contraintes, etc. dont elle a pu hériter de la table partitionnée originale.

4.5.9 Supprimer une partition

```
DROP TABLE nom_partition ;
```

Une partition étant une table, supprimer la table revient à supprimer la partition, et bien sûr les données qu'elle contient. Il n'y a pas besoin de la détacher explicitement auparavant.

L'opération est simple et rapide, mais demande un verrou exclusif.

[2] https://docs.postgresql.fr/14/sql-altertable.html#SQL-ALTERTABLE-DETACH-PARTITION

4.5.10 Fonctions de gestion et vues système

- Disponibles à partir de la v12
  - `\dP`
  - `pg_partition_tree ('logs')` : liste entière des partitions
  - `pg_partition_root ('logs_2019')` : racine d'une partition
  - `pg_partition_ancestors ('logs_201901')` : parents d'une partition

Voici le jeu de tests pour l'exemple qui suivra. Il illustre également l'utilisation de sous-partitions (ici sur la même clé, mais cela n'a rien d'obligatoire).

```
-- Table partitionnée
CREATE TABLE logs (dreception timestamptz, contenu text) PARTITION BY
 ↪  RANGE(dreception);

-- Partition 2018, elle-même partitionnée
CREATE TABLE logs_2018 PARTITION OF logs FOR VALUES FROM ('2018-01-01') TO
 ↪  ('2019-01-01')
                       PARTITION BY range(dreception);
-- Sous-partitions 2018
CREATE TABLE logs_201801 PARTITION OF logs_2018 FOR VALUES FROM ('2018-01-01') TO
 ↪  ('2018-02-01');
CREATE TABLE logs_201802 PARTITION OF logs_2018 FOR VALUES FROM ('2018-02-01') TO
 ↪  ('2018-03-01');
…
-- Idem en 2019
CREATE TABLE logs_2019 PARTITION OF logs FOR VALUES FROM ('2019-01-01') TO
 ↪  ('2020-01-01')
                       PARTITION BY range(dreception);
CREATE TABLE logs_201901 PARTITION OF logs_2019 FOR VALUES FROM ('2019-01-01') TO
 ↪  ('2019-02-01');
…
```

Et voici le test des différentes fonctions :

```
=# SELECT pg_partition_root('logs_2019');

 pg_partition_root
-------------------
 logs

=# SELECT pg_partition_root('logs_201901');

 pg_partition_root
-------------------
 logs
```

```
=# SELECT pg_partition_ancestors('logs_2018');

 pg_partition_ancestors
------------------------
 logs_2018
 logs

=# SELECT pg_partition_ancestors('logs_201901');

 pg_partition_ancestors
------------------------
 logs_201901
 logs_2019
 logs

=# SELECT * FROM pg_partition_tree('logs');

    relid    | parentrelid | isleaf | level
-------------+-------------+--------+-------
 logs        |             | f      |     0
 logs_2018   | logs        | f      |     1
 logs_2019   | logs        | f      |     1
 logs_201801 | logs_2018   | t      |     2
 logs_201802 | logs_2018   | t      |     2
 logs_201901 | logs_2019   | t      |     2
```

Noter les propriétés de « feuille » (*leaf*) et le niveau de profondeur dans le partitionnement.

Sous psql, `\d` affichera toutes les tables, partitions comprises, ce qui peut vite encombrer l'affichage.

À partir de la version 12 du client, `\dP` affiche uniquement les tables et index partitionnés :

```
=# \dP
                Liste des relations partitionnées
 Schéma |   Nom    | Propriétaire |        Type         |  Table
--------+----------+--------------+---------------------+----------
 public | logs     | postgres     | table partitionnée |
 public | t2       | postgres     | index partitionné  | bigtable
```

La table système `pg_partitioned_table`[3] permet des requêtes plus complexes. Le champ `pg_class.relpartbound`[4] contient les définitions des clés de partitionnement.

4.5.11 Indexation

- Propagation automatique (v11+)
- Index supplémentaires par partition possibles
- Clés étrangères entre tables partitionnées (v12+)

[3] https://www.postgresql.org/docs/current/catalog-pg-partitioned-table.html
[4] https://www.postgresql.org/docs/14/catalog-pg-class.html

À partir de la version 11, les index sont propagés de la table mère aux partitions : tout index créé sur la table partitionnée sera automatiquement créé sur les partitions existantes. Toute nouvelle partition disposera des index de la table partitionnée. La suppression d'un index se fait sur la table partitionnée et concernera toutes les partitions. Il n'est pas possible de supprimer un tel index d'une seule partition.

Gérer des index manuellement sur certaines partitions est possible. Par exemple, on peut n'avoir besoin de certains index que sur les partitions de données récentes, et ne pas les créer sur des partitions de données d'archives. (Et avec PostgreSQL 10, les index doivent ainsi être gérés à la main, partition par partition : cette version ne permet donc pas de gérer des clés primaires sur toute la table.)

Toujours à partir de PostgreSQL 11, on peut créer une clé primaire ou unique sur une table partitionnée (mais elle devra contenir toutes les colonnes de la clé de partitionnement) ; ainsi qu'une clé étrangère d'une table partitionnée *vers* une table normale.

Cependant, il faut au minimum PostgreSQL 12 pour pouvoir créer une clé étrangère *vers* une table partitionnée. Par exemple, si des tables `ventes` et `lignes_ventes` sont partitionnées, poser une clé étrangère entre les deux échouera avant PostgreSQL 12 :

```
ALTER TABLE lignes_ventes
ADD CONSTRAINT lignes_ventes_ventes_fk
FOREIGN KEY (vente_id) REFERENCES ventes(vente_id) ;
```

(En version 11, la colonne `lignes_ventes.vente_id` peut tout de même être indexée, et il reste possible d'ajouter manuellement des contraintes entre les partitions une à une, si les clés de partitionnement sont compatibles.)

4.5.12 Opérations de maintenance

- Changement de tablespace
- autovacuum/analyze
  - sur les partitions comme sur toute table
- `VACUUM`, `VACUUM FULL`, `ANALYZE`
  - sur table mère : redescendent sur les partitions
- `REINDEX`
  - avant v14 : uniquement par partition
- `ANALYZE`
  - prévoir aussi sur la table mère (manuellement…)

Les opérations de maintenance profitent grandement du fait de pouvoir scinder les opérations en autant d'étapes qu'il y a de partitions.

Des données « froides » peuvent être déplacées dans un autre tablespace sur des disques moins chers, partition par partition, ce qui est impossible avec une table monolithique :

```
ALTER TABLE pgbench_accounts_8 SET TABLESPACE hdd ;
```

L'autovacuum et l'autoanalyze fonctionnent normalement et indépendamment sur chaque partition, comme sur les tables classiques. Ainsi ils peuvent se déclencher plus souvent sur les partitions actives. Par rapport à une grosse table monolithique, il y a moins souvent besoin de régler l'autovacuum.

Les ordres `ANALYZE` et `VACUUM` peuvent être effectués sur une partition, mais aussi sur la table partitionnée, auquel cas l'ordre redescendra en cascade sur les partitions (l'option `VERBOSE` permet de le vérifier). Les statistiques seront calculées par partition, donc plus précises.

Reconstruire une table partitionnée avec `VACUUM FULL` se fera généralement partition par partition. Dans certains cas, l'opération devient possible seulement grâce au partitionnement : le verrou sur une table monolithique serait trop long, ou l'espace disque total serait insuffisant.

Au-dessus des partitions, noter cependant ces spécificités sur les tables partitionnées :

REINDEX :

À partir de PostgreSQL 14, un `REINDEX` sur la table partitionnée réindexe toutes les partitions automatiquement. Dans les versions précédentes, il faut réindexer partition par partition.

ANALYZE :

L'autovacuum ne crée pas spontanément de statistiques sur les données pour la table partitionnée dans son ensemble, mais uniquement partition par partition. Pour obtenir des statistiques sur toute la table partitionnée, il faut exécuter manuellement :

```
ANALYZE table_partitionnée ;
```

4.5.13 Intérêts du partitionnement déclaratif

- Souple
- Performant
- Intégration au moteur

Par rapport à l'ancien système, le partitionnement déclaratif n'a que des avantages : rapidité d'insertion, souplesse dans le choix du partitionnement, intégration au moteur (ce qui garantit l'intégrité des données)…

4.5.14 Limitations du partitionnement déclaratif et versions

- Temps de planification ! Max ~100 partitions si transactions courtes
- Pas de création automatique des partitions
- Pas d'héritage multiple, schéma fixe
- Partitions distantes sans propagation d'index
- PostgreSQL >= 13 conseillée !
  - v10 : ni partition par défaut, ni propagation des index & contraintes
  - v10/11 : pas de clé étrangère *vers* une table partitionnée
  - v10/11/12 : pas de triggers `BEFORE UPDATE … FOR EACH ROW`
  - contournement : travailler par partition

Une table partitionnée ne peut être convertie en table classique, ni vice-versa. (Par contre, une table classique peut être attachée comme partition, ou une partition détachée).

Les partitions ont forcément le même schéma de données que leur partition mère.

Leur création n'est pas automatisée : il faut les créer par avance manuellement, et éventuellement prévoir une partition par défaut pour les cas qui ont pu être oubliés.

Les clés de partition ne doivent pas se recouvrir. Les contraintes ne peuvent s'exercer qu'au sein d'une même partition : les clés d'unicité doivent donc inclure toute la clé de partitionnement, les contraintes d'exclusion ne peuvent vérifier toutes les partitions.

Il n'y a pas de notion d'héritage multiple.

Une limitation sérieuse du partitionnement tient au temps de planification qui augmente très vite avec le nombre de partitions, même petites. En général, on considère qu'il ne faut pas dépasser 100 partitions si l'on ne veut pas pénaliser les transactions courtes.

Cela est moins un problème pour les requêtes longues (analytiques). Pour contourner cette limite, il est possible de manipuler directement les partitions, s'il est facile pour le développeur (ou le générateur de code…) de trouver leur nom.

L'ordre `CLUSTER`, pour réécrire une table dans l'ordre d'un index donné, ne fonctionne pas pour les tables partitionnées ; il doit être exécuté manuellement table par table.

Il est possible d'attacher comme partitions des tables distantes, généralement déclarée avec `postgres_fdw` ; cependant la propagation d'index ne fonctionnera pas sur ces tables. Il faudra les créer manuellement sur les instances distantes. (Restriction supplémentaire en version 10 : les

partitions distantes ne sont accessibles qu'en lecture, si accédées *via* la table mère.) Un TRUNCATE d'une table distante n'est pas possible avant PostgreSQL 14.

Les partitions par défaut n'existent pas en version 10.

Les limitations sur les index et clés primaires et étrangères avant la version 12 ont été évoquées plus haut.

Les triggers de lignes ne se propagent pas en version 10. En v11, on peut créer des triggers AFTER UPDATE … FOR EACH ROW, mais les BEFORE UPDATE … FOR EACH ROW ne peuvent toujours pas être créés sur la table mère. Il reste là encore la possibilité de les créer partition par partition au besoin. À partir de la version 13, les triggers BEFORE UPDATE … FOR EACH ROW sont possibles, mais il ne permettent pas de modifier la partition de destination.

Enfin, la version 10 ne permet pas de faire une mise à jour (UPDATE) d'une ligne où la clé de partitionnement est modifiée de telle façon que la ligne doit changer de partition. Il faut faire un DELETE et un INSERT à la place. La version 11 gère mieux ce cas en déplaçant directement la ligne dans la bonne partition.

Toujours en version 10 uniquement, un UPDATE sur une ligne ne peut encore la faire changer de changer de partition : il faut faire un DELETE et un INSERT à la place.

On constate que des limitations évoquées plus haut dépendent des versions de PostgreSQL. Si le partitionnement vous intéresse, il est conseillé d'utiliser une version la plus récente possible, au moins PostgreSQL 13.

4.6 EXTENSIONS & OUTILS

- Extension pg_partman[5]
  - automatisation
- Extensions dédiées à un domaine :
  - timescaledb
  - citus

L'extension pg_partman[6], de Crunchy Data, est un complément aux systèmes de partitionnement de PostgreSQL. Elle est apparue d'abord pour automatiser le partitionnement par héritage. Elle peut être utile avec le partitionnement déclaratif, pour simplifier la maintenance d'un partitionnement sur une échelle temporelle ou de valeurs (par *range*).

PostgresPro proposait un outil nommé pg_pathman[7], à présent déprécié en faveur du partitionnement déclaratif intégré à PostgreSQL.

timescaledb est une extension spécialisée dans les séries temporelles. Basée sur le partitionnement par héritage, elle vaut surtout pour sa technique de compression et ses utilitaires. La version communautaire sur Github[8] ne comprend pas tout ce qu'offre la version commerciale.

citus[9] est une autre extension commerciale. Le principe est de partitionner agressivement les tables sur plusieurs instances, et d'utiliser simultanément les processeurs, disques de toutes ces instances (*sharding*). Citus gère la distribution des requêtes, mais pas la maintenance des instances PostgreSQL supplémentaires. L'éditeur Citusdata a été racheté par Microsoft, qui le propose à présent dans Azure. En 2022, l'entièreté du code est passée sous licence libre[10]. Le gain de performance peut être impressionnant, mais attention : certaines requêtes se prêtent très mal au *sharding*.

[6] https://github.com/pgpartman/pg_partman
[7] https://github.com/postgrespro/pg_pathman
[8] https://github.com/timescale/timescaledb
[9] https://github.com/citusdata/citus
[10] https://www.citusdata.com/blog/2022/06/17/citus-11-goes-fully-open-source/

4.7 CONCLUSION

- Le partitionnement déclaratif est mûr
- Préférer une version récente de PostgreSQL

Le partitionnement par héritage n'a plus d'utilité pour la plupart des applications.

Le partitionnement déclaratif apparu en version 10 est mûr dans les dernières versions. Il introduit une complexité supplémentaire, mais peut rendre de grands services quand la volumétrie augmente.

4.8 QUIZ

https://dali.bo/v1_quiz

4.9 TRAVAUX PRATIQUES

4.9.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Nous travaillons sur la base **cave**. La base **cave** peut être téléchargée depuis https://dali.bo/tp_cave (dump de 2,6 Mo, pour 71 Mo sur le disque au final) et importée ainsi :

```
$ psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
$ psql -c "CREATE DATABASE cave OWNER caviste"
$ pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

Nous allons partitionner la table `stock` sur l'année.

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions) :

```
-- Création de lignes en 2001-2005
INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
-- purge des lignes prédédentes
DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

4.9.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

4.9.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

NB : Pour le TP, la base sera d'échelle 10 (environ 168 Mo). Des échelles 100 ou 1000 seraient plus réalistes.

Dans une fenêtre en arrière-plan, laisser tourner un processus pgbench avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

4.9.2.2 Partitionnement par *hash*

Pour partitionner la table pgbench_accounts par *hash* sur la colonne aid sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis pgbench_accounts ;
- la substitution de la table partitionnée à la table originale.

Tester et exécuter.

Supprimer l'ancienne table pgbench_accounts_old.

4.9.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

La table pgbench_history se remplit avec le temps. Elle doit être partitionnée par date (champ mtime). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.

- Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.

NB : Pour éviter de coder des dates en dur, il est possible, avec psql, d'utiliser une variable :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;
```

Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

4.9.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

4.9.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre `pgbench_accounts` et `pgbench_history`. Voir les contraintes créées.

Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

4.9.2.6 Index global

On veut créer un index sur `pgbench_history (aid)`.

Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

4.10 TRAVAUX PRATIQUES (SOLUTIONS)

4.10.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions).

```
INSERT INTO stock
SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock
GROUP BY vin_id, contenant_id, 2001 + annee % 5 ;

DELETE FROM stock WHERE annee<2001 ;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

```
ALTER TABLE stock RENAME TO stock_old;
CREATE TABLE stock(LIKE stock_old) PARTITION BY LIST (annee);
```

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

```
CREATE TABLE stock_2001 PARTITION of stock FOR VALUES IN (2001) ;
CREATE TABLE stock_2002 PARTITION of stock FOR VALUES IN (2002) ;
CREATE TABLE stock_2003 PARTITION of stock FOR VALUES IN (2003) ;
CREATE TABLE stock_2004 PARTITION of stock FOR VALUES IN (2004) ;
CREATE TABLE stock_2005 PARTITION of stock FOR VALUES IN (2005) ;
```

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

```
INSERT INTO stock SELECT * FROM stock_old;
```

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

```
ANALYZE;
```

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

```
SELECT count(*) FROM stock_2001;
SELECT count(*) FROM ONLY stock;
```

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE annee=2002;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Append  (cost=0.00..417.36 rows=18192 width=16) (...)
   ->  Seq Scan on stock_2002  (cost=0.00..326.40 rows=18192 width=16) (...)
         Filter: (annee = 2002)
 Planning Time: 0.912 ms
 Execution Time: 21.518 ms
```

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

```
CREATE UNIQUE INDEX ON stock (vin_id,contenant_id,annee);
```

Les autres index ne servent à rien sur les partitions : `idx_stock_annee` est évidemment inutile, mais `idx_stock_vin_annee` aussi, puisqu'il est inclus dans l'index unique que nous venons de créer.

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE vin_id=1725 AND annee=2003 ;
```

```
 Append  (cost=0.29..4.36 rows=3 width=16) (...)
   ->  Index Scan using stock_2003_vin_id_contenant_id_annee_idx on stock_2003 (...)
         Index Cond: ((vin_id = 1725) AND (annee = 2003))
 Planning Time: 1.634 ms
 Execution Time: 0.166 ms
```

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

```
UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

```
ERROR:  duplicate key value violates unique constraint
↪  "stock_2004_vin_id_contenant_id_annee_idx"
DETAIL:  Key (vin_id, contenant_id, annee)=(1725, 1, 2004) already exists.
```

C'est une violation de contrainte unique, qui est une erreur normale : nous avons déjà un enregistrement de stock pour ce vin pour l'année 2004.

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

```
DELETE FROM stock WHERE annee=2004 and vin_id=1725;

UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

```
DROP TABLE stock_2001 ;
```

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;

ERROR:  no partition of relation "stock" found for row
DETAIL:  Partition key of the failing row contains (annee) = (2006).
```

Il n'existe pas de partition définie pour l'année 2006, cela échoue donc.

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

```
CREATE TABLE stock_default PARTITION OF stock DEFAULT ;

INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;
```

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

```
CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

ERROR:  updated partition constraint for default partition "stock_default"
        would be violated by some row
```

Cela échoue car des enregistrements présents dans la partition par défaut répondent à cette nouvelle contrainte de partitionnement.

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

```
BEGIN ;

ALTER TABLE stock DETACH PARTITION stock_default;

CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

INSERT INTO stock SELECT * FROM stock_default WHERE annee = 2006 ;

DELETE FROM stock_default WHERE annee = 2006 ;

ALTER TABLE stock ATTACH PARTITION stock_default DEFAULT ;

COMMIT ;
```

4.10.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

4.10.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 10 pgbench
```

Dans une fenêtre en arrière-plan, laisser tourner un processus `pgbench` avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

```
$ /usr/pgsql-14/bin/pgbench -n -T3600 -c20 -j2 --debug pgbench
```

L'activité est à ajuster en fonction de la puissance de la machine. Laisser l'affichage défiler dans une fenêtre pour bien voir les blocages.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

Un verrou en attente peut bloquer les opérations d'autres transactions venant après. On peut annuler l'opération à partir d'un certain seuil pour éviter ce phénomène :

```
pgbench=# SET lock_timeout TO '10s' ;
```

Cela ne concerne cependant pas les opérations une fois que les verrous sont acquis. On peut garantir qu'un ordre donné ne durera pas plus d'une certaine durée :

```
SET statement_timeout TO '10s' ;
```

En fonction de la rapidité de la machine et des données à déplacer, cette interruption peut être tolérable ou non.

4.10.2.2 Partitionnement par *hash*

Pour partitionner la table `pgbench_accounts` par *hash* sur la colonne `aid` sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis `pgbench_accounts` ;
- la substitution de la table partitionnée à la table originale.

> Tester et exécuter.

Le champ `aid` n'a pas de signification, un partitionnement par *hash* est adéquat.

Le script peut être le suivant :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

BEGIN ;

-- Nouvelle table partitionnée

CREATE TABLE pgbench_accounts_part (LIKE pgbench_accounts INCLUDING ALL)
PARTITION BY HASH (aid) ;

CREATE TABLE pgbench_accounts_1 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 0 ) ;

CREATE TABLE pgbench_accounts_2 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 1 ) ;

CREATE TABLE pgbench_accounts_3 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 2 ) ;

-- Transfert des données

-- Bloquer les accès à la table le temps du transfert
-- (sinon risque de perte de données !)
LOCK TABLE pgbench_accounts ;

-- Copie des données
INSERT INTO pgbench_accounts_part
SELECT * FROM pgbench_accounts ;

-- Substitution par renommage
ALTER TABLE pgbench_accounts RENAME TO pgbench_accounts_old ;
ALTER TABLE pgbench_accounts_part RENAME TO pgbench_accounts ;

-- Contrôle

\d+

-- On ne validera qu'après contrôle
-- (pendant ce temps les sessions concurrentes restent bloquées !)

COMMIT ;
```

À la moindre erreur, la transaction tombe en erreur. Il faudra demander manuellement ROLLBACK.

Si la durée fixée par `statement_timeout` est dépassée, on aura cette erreur :

```
ERROR:  canceling statement due to statement timeout
Time: 10115.506 ms (00:10.116)
```

Surtout, le traitement pgbench reprend en arrière-plan. On peut alors relancer le script corrigé plus tard.

Si tout se passe bien, un `\d+` renvoie ceci :

```
                          Liste des relations
 Schéma |         Nom          |        Type        | Propriétaire | Taille  | …
--------+----------------------+--------------------+--------------+---------+--
 public | pgbench_accounts     | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_2   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_3   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_old | table              | postgres     | 130 MB  |
 public | pgbench_branches     | table              | postgres     | 136 kB  |
 public | pgbench_history      | table              | postgres     | 5168 kB |
 public | pgbench_tellers      | table              | postgres     | 216 kB  |
```

On peut vérifier rapidement que les valeurs de `aid` sont bien réparties entre les 3 partitions :

```
SELECT aid FROM pgbench_accounts_1 LIMIT 3 ;

 aid
-----
   2
   6
   8

SELECT aid FROM pgbench_accounts_2 LIMIT 3 ;

 aid
-----
   3
   7
  10

SELECT aid FROM pgbench_accounts_3 LIMIT 3 ;

 aid
-----
   1
   9
  11
```

Après la validation du script, on voit apparaître les lignes dans les nouvelles partitions :

```
SELECT relname, n_live_tup
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_accounts%' ;

       relname        | n_live_tup
----------------------+------------
 pgbench_accounts_old |    1000002
 pgbench_accounts_1   |     333263
 pgbench_accounts_2   |     333497
 pgbench_accounts_3   |     333240
```

> Supprimer l'ancienne table `pgbench_accounts_old`.

```
DROP TABLE pgbench_accounts_old ;
```

4.10.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

> La table `pgbench_history` se remplit avec le temps. Elle doit être partitionnée par date (champ `mtime`). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.
>
> - Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.
>
> NB : Pour éviter de coder des dates en dur, il est possible, avec `psql`, d'utiliser une variable :
>
> ```
> SELECT ( now()+ interval '60s') AS date_frontiere \gset
> SELECT :'date_frontiere'::timestamptz ;
> ```

La « date frontière » doit être dans le futur (proche). En effet, pgbench va modifier les tables en permanence, on ne sait pas exactement à quel moment la transition aura lieu (et de toute façon on ne maîtrise pas les valeurs de `mtime`) : il continuera donc à écrire dans l'ancienne table, devenue partition, pendant encore quelques secondes.

Cette date est arbitrairement à 1 minute dans le futur, pour dérouler le script manuellement :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
```

Et on peut réutiliser cette variable ainsi ;

```
SELECT :'date_frontiere'::timestamptz ;
```

Le script peut être celui-ci :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;

BEGIN ;

-- Nouvelle table partitionnée
CREATE TABLE pgbench_history_part (LIKE pgbench_history INCLUDING ALL)
PARTITION BY RANGE (mtime) ;

-- Des partitions pour les prochaines minutes
```

```
CREATE TABLE pgbench_history_1
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz  )
             TO (:'date_frontiere'::timestamptz + interval '1min' ) ;

CREATE TABLE pgbench_history_2
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz + interval '1min' )
             TO (:'date_frontiere'::timestamptz + interval '2min' ) ;

-- Au cas où le service perdure au-delà des partitions prévues,
-- on débordera dans cette table
CREATE TABLE pgbench_history_default
PARTITION OF pgbench_history_part DEFAULT ;

-- Jusqu'ici pgbench continue de tourner en arrière plan

-- La table devient une simple partition
-- Ce renommage pose un verrou, les sessions pgbench sont bloquées
ALTER TABLE pgbench_history RENAME TO pgbench_history_orig ;

ALTER TABLE pgbench_history_part
ATTACH PARTITION pgbench_history_orig
FOR VALUES FROM (MINVALUE) TO (:'date_frontiere'::timestamptz) ;

-- Contrôle
\dP

-- Substitution de la table partitionnée à celle d'origine.
ALTER TABLE pgbench_history_part RENAME TO pgbench_history ;

-- Contrôle
\d+ pgbench_history


COMMIT ;
```

> Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

Pour surveiller le contenu des tables jusqu'à la transition :

```
SELECT relname, n_live_tup, now()
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_history%' ;

\watch 3
```

Un \d+ doit renvoyer ceci :

```
                                 Liste des relations
 Schéma |           Nom           |        Type        | Propriétaire | Taille  | …
--------+-------------------------+--------------------+--------------+---------+--
 public | pgbench_accounts        | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1      | table              | postgres     | 44 MB   |
 public | pgbench_accounts_2      | table              | postgres     | 44 MB   |
```

```
 public | pgbench_accounts_3      | table              | postgres     | 44 MB   |
 public | pgbench_branches        | table              | postgres     | 136 kB  |
 public | pgbench_history         | table partitionnée | postgres     | 0 bytes |
 public | pgbench_history_1       | table              | postgres     | 672 kB  |
 public | pgbench_history_2       | table              | postgres     | 0 bytes |
 public | pgbench_history_default | table              | postgres     | 0 bytes |
 public | pgbench_history_orig    | table              | postgres     | 8736 kB |
 public | pgbench_tellers         | table              | postgres     | 216 kB  |
```

4.10.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

```
ALTER TABLE pgbench_history
DETACH PARTITION pgbench_history_orig ;

-- On pourrait faire le DROP directement

DROP TABLE pgbench_history_orig ;
```

4.10.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre pgbench_accounts et pgbench_history. Voir les contraintes créées.

NB : les clés étrangères entre tables partitionnées ne sont pas disponibles avant PostgreSQL 12.

```
SET lock_timeout TO '3s' ;
SET statement_timeout TO '10s' ;

CREATE INDEX ON pgbench_history (aid) ;

ALTER TABLE pgbench_history
ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid) REFERENCES
 ↪  pgbench_accounts ;
```

On voit que chaque partition porte un index comme la table mère. La contrainte est portée par chaque partition.

```
pgbench=# \d+ pgbench_history
  Table partitionnée « public.pgbench_history »
 …
Clé de partition : RANGE (mtime)
Index :
    "pgbench_history_aid_idx" btree (aid)
Contraintes de clés étrangères :
    "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Partitions: pgbench_history_1 FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                              TO ('2020-02-14 17:42:08.298445'),
            pgbench_history_2 FOR VALUES FROM ('2020-02-14 17:42:08.298445')
                              TO ('2020-02-14 17:43:08.298445'),
            pgbench_history_default DEFAULT
```

```
pgbench=# \d+ pgbench_history_1
   Table « public.pgbench_history_1 »
…
Partition de : pgbench_history FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                                          TO ('2020-02-14 17:42:08.298445')
Contrainte de partition : ((mtime IS NOT NULL)
    AND(mtime >= '2020-02-14 17:41:08.298445'::timestamp without time zone)
    AND (mtime < '2020-02-14 17:42:08.298445'::timestamp without time zone))
Index :
    "pgbench_history_1_aid_idx" btree (aid)
Contraintes de clés étrangères :
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                            FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Méthode d'accès : heap
```

> Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

Le `statement_timeout` peut être un problème :

```
SET
pgbench=# ALTER TABLE pgbench_history
          ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid)
          REFERENCES pgbench_accounts ;
ERROR:  canceling statement due to statement timeout
```

On peut créer les contraintes séparément sur les tables. Cela permet de ne poser un verrou sur la partition active (sans doute `pgbench_history_default`) que pendant le strict minimum de temps (les autres partitions de `pgbench_history` ne sont pas utilisées).

```
SET statement_timeout to '1s' ;
ALTER TABLE pgbench_history_1 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_2 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_default ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

La contrainte au niveau global sera alors posée presque instantanément :

```
ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

4.10.2.6 Index global

> On veut créer un index sur `pgbench_history (aid)`.
>
> Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

Construire un index de manière concurrente (clause `CONCURRENTLY`) permet de ne pas bloquer la table en écriture pendant la création de l'index, qui peut être très longue. Mais il n'est pas possible de le faire sur la table partitionnée :

```
CREATE INDEX CONCURRENTLY ON pgbench_history (aid) ;

ERROR:  cannot create index on partitioned table "pgbench_history" concurrently
```

Mais on peut créer l'index sur chaque partition séparément :

```
CREATE INDEX CONCURRENTLY ON pgbench_history_1 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_2 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_default (aid) ;
```

S'il y a beaucoup de partitions, on peut générer dynamiquement ces ordres :

```
SELECT 'CREATE INDEX CONCURRENTLY ON ' ||
       c.oid::regclass::text || ' (aid) ; '
FROM pg_class c
WHERE relname like 'pgbench_history%' AND relispartition \gexec
```

Comme lors de toute création concurrente, il faut vérifier que les index sont bien valides : la requête suivante ne doit rien retourner.

```
SELECT indexrelid::regclass FROM pg_index WHERE NOT  indisvalid ;
```

Enfin on crée l'index au niveau de la table partitionnée : il réutilise les index existants et sera donc créé presque instantanément :

```
CREATE INDEX ON pgbench_history(aid) ;

pgbench=# \d+ pgbench_history
..
Partition key: RANGE (mtime)
Indexes:
    "pgbench_history_aid_idx" btree (aid)
…
```

5/ Types avancés

PostgreSQL offre des types avancés :

- Composés :
    - `hstore`
    - JSON : `json`, `jsonb`
    - XML
- Pour les objets binaires :
    - `bytea`
    - Large Objects

5.1 TYPES COMPOSÉS : GÉNÉRALITÉS

- Un champ = plusieurs attributs
- De loin préférable à une table Entité/Attribut/Valeur
- Uniquement si le modèle relationnel n'est pas assez souple
- 3 types dans PostgreSQL :
  - `hstore` : clé/valeur
  - `json` : JSON, stockage texte, validation syntaxique, fonctions d'extraction
  - `jsonb` : JSON, stockage binaire, accès rapide, fonctions d'extraction, de requêtage, indexation avancée

Ces types sont utilisés quand le modèle relationnel n'est pas assez souple, donc s'il est nécessaire d'ajouter dynamiquement des colonnes à la table suivant les besoins du client, ou si le détail des attributs d'une entité n'est pas connu (modélisation géographique par exemple), etc.

La solution traditionnelle est de créer des tables entité/attribut de ce format :

```
CREATE TABLE attributs_sup (entite int, attribut text, valeur text);
```

On y stocke dans `entite` la clé de l'enregistrement de la table principale, dans `attribut` la colonne supplémentaire, et dans `valeur` la valeur de cet attribut. Ce modèle présente l'avantage évident de résoudre le problème. Les défauts sont par contre nombreux :

- Les attributs d'une ligne peuvent être totalement éparpillés dans la table `attributs_sup` : récupérer n'importe quelle information demandera donc des accès à de nombreux blocs différents.
- Il faudra plusieurs requêtes (au moins deux) pour récupérer le détail d'un enregistrement, avec du code plus lourd côté client pour fusionner le résultat des deux requêtes, ou bien une requête effectuant des jointures (autant que d'attributs, sachant que le nombre de jointures complexifie énormément le travail de l'optimiseur SQL) pour retourner directement l'enregistrement complet.

Toute recherche complexe est très inefficace : une recherche multi-critères sur ce schéma va être extrêmement peu performante. Les statistiques sur les valeurs d'un attribut deviennent nettement moins faciles à estimer pour PostgreSQL. Quant aux contraintes d'intégrité entre valeurs, elles deviennent pour le moins complexes à gérer.

Les types `hstore`, `json` et `jsonb` permettent de résoudre le problème autrement. Ils permettent de stocker les différentes entités dans un seul champ pour chaque ligne de l'entité. L'accès aux attributs se fait par une syntaxe ou des fonctions spécifiques.

Il n'y a même pas besoin de créer une table des attributs séparée : le mécanisme du « TOAST » permet de déporter les champs volumineux (texte, JSON, `hstore`…) dans une table séparée gérée par Post-

greSQL, éventuellement en les compressant, et cela de manière totalement transparente. On y gagne donc en simplicité de développement.

5.2 HSTORE

Stocker des données non structurées

- Extension
- Stockage Clé/Valeur, uniquement texte
- Binaire
- Indexable
- Plusieurs opérateurs disponibles

hstore est une extension, fournie en « contrib ». Elle est donc systématiquement disponible. L'installer permet d'utiliser le type de même nom. On peut ainsi stocker un ensemble de clés/valeurs, exclusivement textes, dans un unique champ.

Ces champs sont indexables et peuvent recevoir des contraintes d'intégrité (unicité, non recouvrement…).

Les `hstore` ne permettent par contre qu'un modèle « plat ». Il s'agit d'un pur stockage clé-valeur. Si vous avez besoin de stocker des informations davantage orientées document, vous devrez vous tourner vers un type JSON.

Ce type perd donc de son intérêt depuis que PostgreSQL 9.4 a apporté le type `jsonb`. Il lui reste sa simplicité d'utilisation.

5.2.1 hstore : exemple

```
CREATE EXTENSION hstore ;

CREATE TABLE animaux (nom text, caract hstore);
INSERT INTO animaux VALUES ('canari','pattes=>2,vole=>oui');
INSERT INTO animaux VALUES ('loup','pattes=>4,carnivore=>oui');
INSERT INTO animaux VALUES ('carpe','eau=>douce');

CREATE INDEX idx_animaux_donnees ON animaux
                                 USING gist (caract);

SELECT *, caract->'pattes' AS nb_pattes FROM animaux
WHERE caract@>'carnivore=>oui';

 nom  |              caract                | nb_pattes
------+------------------------------------+-----------
 loup | "pattes"=>"4", "carnivore"=>"oui" | 4
```

Les ordres précédents installent l'extension, créent une table avec un champ de type `hstore`, insèrent trois lignes, avec des attributs variant sur chacune, indexent l'ensemble avec un index GiST, et enfin recherchent les lignes où l'attribut `carnivore` possède la valeur `t`.

```
# SELECT * FROM animaux ;

  nom   |              caract
--------+----------------------------------
 canari | "vole"=>"oui", "pattes"=>"2"
 loup   | "pattes"=>"4", "carnivore"=>"oui"
 carpe  | "eau"=>"douce"
```

Les différentes fonctions disponibles sont bien sûr dans la documentation[1].

Par exemple :

```
# UPDATE animaux SET caract = caract||'poil=>t'::hstore
  WHERE nom = 'loup' ;

# SELECT * FROM animaux WHERE caract@>'carnivore=>oui';

 nom  |                      caract
------+-------------------------------------------------
 loup | "poil"=>"t", "pattes"=>"4", "carnivore"=>"oui"
```

Il est possible de convertir un `hstore` en tableau :

```
# SELECT hstore_to_matrix(caract) FROM animaux
  WHERE caract->'vole' = 'oui';

    hstore_to_matrix
-------------------------
 { {vole,oui},{pattes,2} }
```

ou en JSON :

```
# SELECT caract::jsonb FROM animaux
  WHERE (caract->'pattes')::int > 2;

                       caract
-----------------------------------------------------
 {"pattes": "4", "poil": "t", "carnivore": "oui"}
```

L'indexation de ces champs peut se faire avec divers types d'index. Un index unique n'est possible qu'avec un index B-tree classique. Les index GIN ou GiST sont utiles pour rechercher des valeurs d'un attribut. Les index hash ne sont utiles que pour des recherches d'égalité d'un champ entier ; par contre ils sont très compacts. (Rappelons que les index hash sont inutilisables avant PostgreSQL 10).

[1] https://docs.postgresql.fr/current/hstore.html

5.3 JSON

```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "postalCode": "75009"
  }
}
```

- `json` : format texte
- `jsonb` : format binaire, à préférer
- `jsonpath` : SQL/JSON paths (PG 12+)

Le format JSON[2] est devenu extrêmement populaire. Au-delà d'un simple stockage clé/valeur, il permet de stocker des tableaux, ou des hiérarchies, de manière plus simple et lisible qu'en XML. Par exemple, pour décrire une personne, on peut utiliser cette structure :

```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "state": "",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    {
      "type": "personnel",
      "number": "06 12 34 56 78"
    },
    {
      "type": "bureau",
      "number": "07 89 10 11 12"
    }
  ],
  "children": [],
  "spouse": null
}
```

Historiquement, le JSON est apparu dans PostgreSQL 9.2, mais n'est vraiment utilisable qu'avec

[2] https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

l'arrivée du type `jsonb` (binaire) dans PostgreSQL 9.4. Les opérateurs SQL/JSON path[3] ont été ajoutés dans PostgreSQL 12, suite à l'introduction du JSON dans le standard SQL:2016. Ils permettent de spécifier des parties d'un champ JSON.

5.3.1 Type json

- Type texte
- Validation du format JSON
- Fonctions de manipulation JSON
  - Mais ré-analyse du champ pour chaque appel de fonction
  - Indexation comme un simple texte
- => Réservé au stockage à l'identique
  - Sinon préférer `jsonb`

Le type natif `json`, dans PostgreSQL, n'est rien d'autre qu'un habillage autour du type texte. Il valide à chaque insertion/modification que la donnée fournie est une syntaxe JSON valide.

Le stockage est exactement le même qu'une chaîne de texte, et utilise le mécanisme du « TOAST », qui compresse au besoin les plus grands champs, de manière transparente pour l'utilisateur.

Toutefois, le fait que la donnée soit validée comme du JSON permet d'utiliser des fonctions de manipulation, comme l'extraction d'un attribut, la conversion d'un JSON en enregistrement, de façon systématique sur un champ sans craindre d'erreur.

On préférera généralement le type binaire `jsonb`, pour les performances et ses fonctionnalités supplémentaires. Au final, l'intérêt de `json` est surtout de conserver un objet JSON sous sa forme originale.

Les exemples suivants avec `jsonb` sont aussi applicables à `json`. La plupart des fonctions et opérateurs existent dans les deux versions.

[3] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

5.3.2 Type jsonb

- **JSON** au format **B**inaire
- Indexation GIN
- Gestion du langage JSON Path (v12+)

Le type `jsonb` permet de stocker les données dans un format binaire optimisé. Ainsi, il n'est plus nécessaire de désérialiser l'intégralité du document pour accéder à une propriété.

Pour un exemple extrême (document JSON d'une centaine de Mo), voici une comparaison des performances entre les deux types `json` et `jsonb` pour la récupération d'un champ sur 1300 lignes :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_json;

                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on test_json  (cost=0.00..26.38 rows=1310 width=32)
                        (actual time=893.454..912.168 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.021 ms
 Execution time: 912.194 ms

EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_jsonb;

                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on test_jsonb  (cost=0.00..26.38 rows=1310 width=32)
                         (actual time=77.707..84.148 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.026 ms
 Execution time: 84.177 ms
```

5.3.3 JSON : Exemple d'utilisation

```
CREATE TABLE personnes (datas jsonb );

INSERT INTO personnes  (datas) VALUES ('
{
  "firstName": "Jean",
  "lastName": "Valjean",
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    { "number": "06 12 34 56 78" },
    {"type": "bureau",
     "number": "07 89 10 11 12"}
  ],
  "children": [],
  "spouse": null
}
'),
('
{
  "firstName": "Georges",
  "lastName": "Durand",
  "address": {
    "streetAddress": "27 rue des Moulins",
    "city": "Châteauneuf",
    "postalCode": "45990"
  },
  "phoneNumbers": [
    { "number": "06 21 34 56 78" },
    {"type": "bureau",
     "number": "07 98 10 11 12"}
  ],
  "children": [],
  "spouse": null
}
');
```

Un champ de type `jsonb` (ou `json`) accepte tout champ JSON directement.

5.3.4 JSON : Affichage de champs

```
SELECT datas->>'firstName' AS prenom,    -- text
       datas->'address'    AS addr       -- jsonb
       FROM personnes ;

SELECT datas #>> '{address,city}' AS villes FROM personnes ; -- text

SELECT datas['address']['city'] as villes from personnes ;  -- jsonb, v14

SELECT datas['address']->>'city' as villes from personnes ;  -- text, v14

SELECT jsonb_array_elements (datas->'phoneNumbers')->>'number' FROM
 ↪  personnes;
```

Le type json dispose de nombreuses fonctions de manipulation et d'extraction. Les opérateurs ->> et -> renvoient respectivement une valeur au format texte, et au format JSON.

```
# SELECT datas->>'firstName' AS prenom,
       datas->'address'      AS addr
       FROM personnes
       \gdesc
 Column | Type
--------+-------
 prenom | text
 addr   | jsonb

# \g
 prenom  |                        addr
---------+-----------------------------------------------------
 Jean    | {                                                  +
         |     "streetAddress": "43 rue du Faubourg Montmartre",+
         |     "city": "Paris",                               +
         |     "state": "",                                   d
         |     "postalCode": "75002"                          +
         |   }
 Georges | {                                                  +
         |     "streetAddress": "27 rue des Moulins",         +
         |     "city": "Châteauneuf",                         +
         |     "postalCode": "45990"                          +
         |   }
```

L'équivalent existe avec des chemins, avec #> et #>> :

```
# SELECT datas #>> '{address,city}' AS villes FROM personnes ;

   villes
-------------
 Paris
 Châteauneuf
```

Depuis la version 14, une autre syntaxe plus claire est disponible, plus simple, et qui renvoie du JSON :

```
SELECT datas['address']['city'] as villes from personnes ;
```

```
    villes
---------------
 "Paris"
 "Châteauneuf"
```

`jsonb_array_elements` permet de parcourir des tableaux de JSON :

```
# SELECT jsonb_array_elements (datas->'phoneNumbers')->>'number'
         AS numero
  FROM personnes ;
```

```
     numero
----------------
 06 12 34 56 78
 07 89 10 11 12
 06 21 34 56 87
 07 98 10 11 13
```

5.3.5 Conversions jsonb / relationnel

- Construire un objet JSON depuis un ensemble :
  - `json_object_agg()`
- Construire un ensemble de tuples depuis un objet JSON :
  - `jsonb_each()`
  - `jsonb_to_record()`
- Manipuler des tableaux :
  - `jsonb_array_elements()`
  - `jsonb_to_recordset()`

Les fonctions permettant de construire du `jsonb`, ou de le manipuler de manière ensembliste permettent une très forte souplesse. Il est aussi possible de déstructurer des tableaux, mais il est compliqué de requêter sur leur contenu.

Par exemple, si l'on souhaite filtrer des documents de la sorte pour ne ramener que ceux dont une catégorie est `categorie` :

```
{
  "id": 3,
```

```
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
```

```
CREATE TABLE json_table (id serial, document jsonb);
INSERT INTO json_table (document) VALUES ('
{
  "id": 3,
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
');
```

```
SELECT document->'id'
FROM json_table j,
LATERAL jsonb_array_elements(document #> '{sous_document, mon_tableau}')
    AS elements_tableau
WHERE elements_tableau->>'categorie' = 'categorie';
```

Ce type de requête devient rapidement compliqué à écrire, et n'est pas indexable.

5.3.6 JSON : performances

Inconvénients par rapport à un modèle normalisé :

- Perte d'intégrité (types, contraintes, FK…)
- Complexité du code
- Pas de statistiques sur les clés JSON !
- Pas forcément plus léger en disque
  - clés répétées
- Lire 1 champ = lire tout le JSON
  - voire accès table TOAST
- Mise à jour : tout ou rien

Les champs JSON sont très pratiques quand le schéma est peu structuré. Mais la complexité supplémentaire de code nuit à la lisibilité des requêtes. En termes de performances, ils sont coûteux, pour

les raisons que nous allons voir.

Les contraintes d'intégrité sur les types, les tailles, les clés étrangères… ne sont pas disponibles. Les contraintes protègent de nombreux bugs, mais elles sont aussi une aide précieuse pour l'optimiseur.

Chaque JSON récupéré l'est en bloc. Si un seul champ est récupéré, PostgreSQL devra charger tout le JSON et le décomposer. Cela peut même coûter un accès supplémentaire à une table TOAST pour les gros JSON. Rappelons que le mécanisme du TOAST permet à PostgreSQL de compresser à la volée un grand champ texte, binaire, JSON… et/ou de le déporter dans une table annexe interne, le tout étant totalement transparent pour l'utilisateur. Pour les détails, voir cet extrait de la formation DBA2[4].

Il n'y a pas de mise à jour partielle : changer un champ implique de décomposer tout le JSON pour le réécrire entièrement (et parfois en le *détoastant*/*retoastant*).

Un gros point noir est l'absence de statistiques propres aux clés du JSON. Le planificateur va avoir beaucoup de mal à estimer les cardinalités des critères.

Suivant le modèle, il peut y avoir une perte de place, puisque les clés sont répétées entre chaque champ JSON, et non normalisées dans des tables séparées.

Ces inconvénients sont à mettre en balance avec les intérêts du JSON (surtout : éviter des lignes avec trop de champs toujours à NULL, si même on les connaît), les fréquences de lecture et mises à jour des JSON, et les modalités d'utilisation des champs.

Certaines de ces limites peuvent être réduites par les techniques ci-dessous.

5.3.7 jsonb : indexation (1/2)

- Index fonctionnel sur un champ précis
  - bonus : statistiques

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
ANALYZE personnes ;
```

- Champ dénormalisé:
  - champ normal, indexable, facile à utiliser
  - statistiques

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

Index fonctionnel :

[4] https://dali.bo/m4_html#mécanisme-toast

L'extraction d'une partie d'un JSON est en fait une fonction immutable, donc indexable. Un index fonctionnel permet d'accéder directement à certaines propriétés, par exemple :

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
```

Mais il ne fonctionnera que s'il y a une clause WHERE avec cette expression exacte. Pour un champ fréquemment utilisé pour des recherches, c'est le plus efficace.

On n'oubliera pas de lancer un ANALYZE pour calculer les statistiques après création de l'index fonctionnel. Même si l'index est peu discriminant, on obtient ainsi de bonnes statistiques sur son critère. Un VACUUM permet aussi les *Index Only Scan* quand c'est possible.

Champ dénormalisé :

Une autre possibilité est de dénormaliser le champ JSON intéressant dans un champ séparé de la table, géré automatiquement, et indexable :

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

ANALYZE personnes ;

CREATE INDEX ON personnes (lastname) ;
```

Ce champ coûte un peu d'espace disque supplémentaire, mais il peut être intéressant pour la lisibilité du code, la facilité d'utilisation avec certains outils ou pour certains utilisateurs. Dans le cas des gros JSON, il peut aussi éviter quelques allers-retours vers la table TOAST.

5.3.8 jsonb : indexation (2/2)

- Indexation « schemaless » grâce au GIN :

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

Index GIN :

Les champs `jsonb` peuvent tirer parti de fonctionnalités avancées de PostgreSQL, notamment les index GIN, et ce via deux classes d'opérateurs.

Même si l'opérateur par défaut GIN pour `jsonb` supporte plus d'opérations, il est souvent suffisant, et surtout plus efficace, de choisir l'opérateur `jsonb_path_ops` (voir les détails[5]) :

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

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

`jsonb_path_ops` supporte notamment l'opérateur « contient » (@>) :

```
# EXPLAIN (ANALYZE) SELECT datas->>'firstName' FROM personnes
  WHERE datas @> '{"lastName": "Dupont"}'::jsonb ;

                            QUERY PLAN
------------------------------------------------------------------
Bitmap Heap Scan on personnes  (cost=2.01..3.02 rows=1 width=32)
                      (actual time=0.018..0.019 rows=1 loops=1)
  Recheck Cond: (datas @> '{"lastName": "Dupont"}'::jsonb)
  Heap Blocks: exact=1
  ->  Bitmap Index Scan on idx_prs  (cost=0.00..2.01 rows=1 width=0)
                           (actual time=0.010..0.010 rows=1 loops=1)
        Index Cond: (datas @> '{"lastName": "Dupont"}'::jsonb)
Planning Time: 0.052 ms
Execution Time: 0.104 ms
```

Ce type d'index est moins efficace qu'un index fonctionnel B-tree classique, mais il est idéal quand la clé de recherche n'est pas connue, et que n'importe quel champ du JSON est un critère. De plus il est compressé.

Un index GIN ne permet cependant pas d'*Index Only Scan*.

Surtout, un index GIN ne permet pas de recherches sur des opérateurs B-tree classiques (<, <=, >, >=), ou sur le contenu de tableaux. On est obligé pour cela de revenir au monde relationnel, ou se rabattre sur les index fonctionnels vus plus haut. Il est donc préférable d'utiliser les opérateurs spécifiques, comme « contient » (@>).

5.3.9 SQL/JSON & JSONpath

- SQL:2016 introduit SQL/JSON et le langage JSON Path
- JSON Path :
  - langage de recherche pour JSON
  - concis, flexible, plus rapide
  - inclus dans PostreSQL 12 pour l'essentiel
  - exemple :

```
SELECT jsonb_path_query (datas, '$.phoneNumbers[*] ? (@.type ==
↪  "bureau") ')
  FROM personnes ;
```

JSON path facilite la recherche et le parcours dans les documents JSON complexes. Il évite de parcourir manuellement les nœuds du JSON.

Par exemple, une recherche peut se faire ainsi :

```
SELECT datas->>'firstName' AS prenom
FROM personnes
WHERE  datas  @@ '$.lastName == "Durand"' ;

 prenom
----------
 Georges
```

Mais l'intérêt est d'extraire facilement des parties d'un tableau :

```
SELECT jsonb_path_query (datas, '$.phoneNumbers[*] ? (@.type == "bureau") ')
FROM   personnes ;

               jsonb_path_query
-----------------------------------------------
 {"type": "bureau", "number": "07 89 10 11 12"}
 {"type": "bureau", "number": "07 98 10 11 13"}
```

On trouvera d'autres exemples dans la présentation de Postgres Pro dédié à la fonctionnalité lors la parution de PostgreSQL 12[6], ou dans un billet de Michael Paquier[7].

5.3.10 Extension jsQuery

- Fournit un « langage de requête » sur JSON
  - similaire aux jsonpath (PG 12+), mais dès PG 9.4
- Indexation GIN

```
SELECT document->'id'
FROM json_table j
WHERE j.document @@ 'sous_document.mon_tableau.#.categorie = categorie' ;
```

L'extension jsquery fournit un opérateur @@ (« correspond à la requête jsquery »), similaire à l'opérateur @@ de la recherche plein texte. Celui-ci permet de faire des requêtes évoluées sur un document JSON, optimisable facilement grâce à la méthode d'indexation supportée.

jsquery permet de requêter directement sur des champs imbriqués, en utilisant même des jokers pour certaines parties.

Le langage en lui-même est relativement riche, et fournit un système de *hints* pour pallier à certains problèmes de la collecte de statistiques, qui devrait être améliorée dans le futur.

Il supporte aussi les opérateurs différents de l'égalité :

[6] https://www.postgresql.eu/events/pgconfeu2019/sessions/session/2555/slides/221/jsonpath-pgconfeu-2019.pdf
[7] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

```
SELECT *
FROM json_table j
WHERE j.document @@ 'ville.population > 10000';
```

Le périmètre est très proche des expressions jsonpath apparues dans PostgreSQL 12, qui, elles, se basent sur le standard SQL:2016. Les auteurs sont d'ailleurs les mêmes. Voir cet article pour les détails[8], ou le dépôt github[9]. La communauté fournit des paquets.

[8] https://habr.com/en/company/postgrespro/blog/500440/
[9] https://github.com/akorotkov/jsquery

5.4 XML

- Type `xml`
  - stocke un document XML
  - valide sa structure
- Quelques fonctions et opérateurs disponibles :
  - `XMLPARSE`, `XMLSERIALIZE`, `query_to_xml`, `xmlagg`
  - `xpath` (XPath 1.0 uniquement)

Le type `xml`, inclus de base, vérifie que le XML inséré est un document « bien formé », ou constitue des fragments de contenu (« content »). L'encodage UTF-8 est impératif. Il y a quelques limitations par rapport aux dernières versions du standard, XPath et XQuery[10]. Le stockage se fait en texte, donc bénéficie du mécanisme de compression TOAST.

Il existe quelques opérateurs et fonctions de validation et de manipulations, décrites dans la documentation du type xml[11] ou celle des fonctions[12]. Par contre, une simple comparaison est impossible et l'indexation est donc impossible directement. Il faudra passer par une expression XPath.

À titre d'exemple : `XMLPARSE` convertit une chaîne en document XML, `XMLSERIALIZE` procède à l'opération inverse.

```
CREATE TABLE liste_cd (catalogue xml) ;
\d liste_cd

                  Table « public.liste_cd »
  Colonne  | Type | Collationnement | NULL-able | Par défaut
-----------+------+-----------------+-----------+------------
 catalogue | xml  |                 |           |

INSERT INTO liste_cd
SELECT XMLPARSE ( DOCUMENT
$$<?xml version="1.0" encoding="UTF-8"?>
<CATALOG>
  <CD>
    <TITLE>The Times They Are a-Changin'</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <YEAR>1964</YEAR>
  </CD>
  <CD>
    <TITLE>Olympia 1961</TITLE>
```

[10] https://docs.postgresql.fr/current/xml-limits-conformance.html
[11] https://docs.postgresql.fr/current/datatype-xml.html
[12] https://docs.postgresql.fr/current/functions-xml.html

```
    <ARTIST>Jacques Brel</ARTIST>
    <COUNTRY>France</COUNTRY>
    <YEAR>1962</YEAR>
  </CD>
</CATALOG> $$ ) ;
--- Noter le $$ pour délimiter une chaîne contenant une apostrophe

SELECT XMLSERIALIZE (DOCUMENT catalogue AS text) FROM liste_cd;

                  xmlserialize
-------------------------------------------------
 <?xml version="1.0" encoding="UTF-8"?>          +
 <CATALOG>                                       +
   <CD>                                          +
     <TITLE>The Times They Are a-Changin'</TITLE>+
     <ARTIST>Bob Dylan</ARTIST>                  +
     <COUNTRY>USA</COUNTRY>                      +
     <YEAR>1964</YEAR>                           +
   </CD>                                         +
   <CD>                                          +
     <TITLE>Olympia 1961</TITLE>                 +
     <ARTIST>Jacques Brel</ARTIST>               +
     <COUNTRY>France</COUNTRY>                   +
     <YEAR>1962</YEAR>                           +
   </CD>                                         +
 </CATALOG>
(1 ligne)
```

Il existe aussi `query_to_xml` pour convertir un résultat de requête en XML, `xmlagg` pour agréger des champs XML, ou `xpath` pour extraire des nœuds suivant une expression XPath 1.0.

NB : l'extension xml2[13] est dépréciée et ne doit pas être utilisée dans les nouveaux projets.

[13] https://docs.postgresql.fr/current/xml2.html

5.5 OBJETS BINAIRES

- Souvent une mauvaise idée…
- 2 méthodes
  - `bytea` : type binaire
  - *Large Objects* : manipulation comme un fichier

PostgreSQL permet de stocker des données au format binaire, potentiellement de n'importe quel type, par exemple des images ou des PDF.

Il faut vraiment se demander si des binaires ont leur place dans une base de données relationnelle. Ils sont généralement beaucoup plus gros que les données classiques. La volumétrie peut donc devenir énorme, et encore plus si les binaires sont modifiés, car le mode de fonctionnement de PostgreSQL aura tendance à les dupliquer. Cela aura un impact sur la fragmentation, la quantité de journaux, la taille des sauvegardes, et toutes les opérations de maintenance. Ce qui est intéressant à conserver dans une base sont des données qu'il faudra rechercher, et l'on recherche rarement au sein d'un gros binaire. En général, l'essentiel des données binaires que l'on voudrait confier à une base peut se contenter d'un stockage classique, PostgreSQL ne contenant qu'un chemin ou une URL vers le fichier réel.

PostgreSQL donne le choix entre deux méthodes pour gérer les données binaires :

- `bytea` : un type comme un autre ;
- *Large Object* : des objets séparés, à gérer indépendamment des tables.

5.5.1 bytea

- Un type comme les autres
  - `bytea` : tableau d'octets
  - en texte : `bytea_output` = `hex` ou `escape`
- Récupération intégralement en mémoire !
- Toute modification entraîne la réécriture complète du `bytea`
- Maxi 1 Go (à éviter)
  - en pratique intéressant pour quelques Mo
- Import :

```
SELECT pg_read_binary_file ('/chemin/fichier');
```

Voici un exemple :

```
CREATE TABLE demo_bytea(a bytea);
INSERT INTO demo_bytea VALUES ('bonjour'::bytea);

SELECT * FROM demo_bytea ;

        a
------------------
 \x626f6e6a6f7572
```

Nous avons inséré la chaîne de caractère « bonjour » dans le champ bytea, en fait sa représentation binaire dans l'encodage courant (UTF-8). Si nous interrogeons la table, nous voyons la représentation textuelle du champ bytea. Elle commence par `\x` pour indiquer un encodage de type `hex`. Ensuite, chaque paire de valeurs hexadécimales représente un octet.

Un second format d'affichage est disponible : `escape` :

```
SET bytea_output = escape ;
SELECT * FROM demo_bytea ;

    a
---------
 bonjour

INSERT INTO demo_bytea VALUES ('journée'::bytea);
SELECT * FROM demo_bytea ;

       a
----------------
 bonjour
 journ\303\251e
```

Le format de sortie `escape` ne protège donc que les valeurs qui ne sont pas représentables en ASCII 7 bits. Ce format peut être plus compact pour des données textuelles essentiellement en alphabet latin sans accent, où le plus gros des caractères n'aura pas besoin d'être protégé.

Cependant, le format `hex` est bien plus efficace à convertir, et est le défaut depuis PostgreSQL 9.0.

Avec les vieilles applications, ou celles restées avec cette configuration, il faudra peut-être forcer `bytea_output` à `escape`, sous peine de corruption.)

Pour charger directement un fichier, on peut notamment utiliser la fonction `pg_read_binary_file`, exécutée par le serveur PostreSQL :

```
INSERT INTO demo_bytea (a)
SELECT pg_read_binary_file ('/chemin/fichier');
```

En théorie, un `bytea` peut contenir 1 Go. En pratique, on se limitera à nettement moins, ne serait-ce que parce `pg_dump` tombe en erreur quand il doit exporter des bytea de plus de 500 Mo environ (le décodage double le nombre d'octets et dépasse cette limite de 1 Go).

La documentation officielle[14] liste les fonctions pour encoder, décoder, extraire, hacher… les `bytea`.

5.5.2 Large Object

- Objet indépendant des tables
- Identifié par un OID
  - à stocker dans les tables
- Suppression manuelle
  - trigger
  - batch (extensions) : `lo_unlink` & `vacuumlo`
- Fonction de manipulation, modification
  - `lo_create`, `lo_import`
  - `lo_seek`, `lo_open`, `lo_read`, `lo_write`…
- Maxi 4 To (à éviter aussi…)

[14] https://docs.postgresql.fr/current/functions-binarystring.html

Un *large object* est un objet totalement décorrélé des tables. (il est stocké en fait dans la table système `pg_largeobject`). Le code doit donc gérer cet objet séparément :

- créer le *large object* et stocker ce qu'on souhaite dedans ;
- stocker la référence à ce *large object* dans une table (avec le type `lob`) ;
- interroger l'objet séparément de la table ;
- le supprimer explicitement quand il n'est plus référencé : il ne disparaîtra pas automatiquement !

Le *large object* nécessite donc un plus gros investissement au niveau du code.

En contrepartie, il a les avantages suivant :

- une taille jusqu'à 4 To, ce qui n'est tout de même pas conseillé ;
- la possibilité d'accéder à une partie directement (par exemple les octets de 152 000 à 153 020), ce qui permet de le transférer par parties sans le charger en mémoire (notamment, le driver JDBC de PostgreSQL fournit une classe LargeObject[15]) ;
- de ne modifier que cette partie sans tout réécrire.

Il y a plusieurs méthodes pour nettoyer les *large objets* devenu inutiles :

- appeler la fonction `lo_unlink` dans le code client — au risque d'oublier ;
- utiliser la fonction trigger `lo_manage` fournie par le module contrib `lo` : (voir documentation[16], si les *large objects* ne sont jamais référencés plus d'une fois ;
- appeler régulièrement le programme `vacuumlo` (là encore un contrib[17]) : il liste tous les *large objects* référencés dans la base, puis supprime les autres. Ce traitement est bien sûr un peu lourd.

Voir la documentation[18] pour les détails.

[15] https://jdbc.postgresql.org/documentation/binary-data/
[16] https://docs.postgresql.fr/current/lo.html
[17] https://docs.postgresql.fr/current/vacuumlo.html
[18] https://docs.postgresql.fr/current/largeobjects.html

5.6 QUIZ

https://dali.bo/s9_quiz

5.7 TRAVAUX PRATIQUES

Les TP sur les types hstore et JSON utilisent la base **cave**. La base **cave** peut être téléchargée depuis https://dali.bo/tp_cave (dump de 2,6 Mo, pour 71 Mo sur le disque au final) et importée ainsi :

```
$ psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
$ psql -c "CREATE DATABASE cave OWNER caviste"
$ pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

5.7.1 Hstore (Optionnel)

But : Découvrir hstore

Pour ce TP, il est fortement conseillé d'aller regarder la documentation officielle du type `hstore` sur https://docs.postgresql.fr/current/hstore.html.

But : Obtenir une version dénormalisée de la table `stock` : elle contiendra une colonne de type `hstore` contenant l'année, l'appellation, la région, le récoltant, le type, et le contenant :

```
vin_id      integer
nombre      integer
attributs   hstore
```

Ce genre de table n'est évidemment pas destiné à une application transactionnelle: on n'aurait aucun moyen de garantir l'intégrité des données de cette colonne. Cette colonne va nous permettre d'écrire une recherche multi-critères efficace sur nos stocks.

Afficher les informations à stocker avec la requête suivante :

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
LIMIT 10;
```

(`LIMIT 10` est là juste pour éviter de ramener tous les enregistrements).

> Créer une table `stock_denorm (vin_id int, nombre int, attributs hstore)` et y copier le contenu de la requête. Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de `hstore` sans se soucier de formatage de chaîne de caractères. (Voir la documentation.)

> Créer un index sur le champ `attributs` pour accélérer les recherches.

> Rechercher le nombre de bouteilles (attribut `bouteille`) en stock de vin blanc (attribut `type_vin`) d'Alsace (attribut `region`). Quel est le temps d'exécution de la requête ? Combien de buffers accédés ?

> Refaire la même requête sur la table initiale. Qu'en conclure ?

5.7.2 jsonb

But : Découvrir JSON

Nous allons créer une table dénormalisée contenant uniquement un champ JSON.

Pour chaque vin, le document JSON aura la structure suivante :

```
{
  vin: {
      recoltant: {
        nom: text,
        adressse: text
      },
      appellation: {
        libelle: text,
        region: text
      },
      type_vin: text
  },
  stocks: [{
    contenant: {
      contenance: real,
      libelle: text
    },
    annee: integer,
    nombre: integer
  }]
}
```

Pour écrire une requête permettant de générer ces documents, nous allons procéder par étapes.

La requête suivante permet de générer les parties `vin` du document, avec `recoltant` et `appellation`. Créer un document JSON pour chaque ligne de `vin` grâce à la fonction `jsonb_build_object`.

```
SELECT
  recoltant.nom,
  recoltant.adresse,
  appellation.libelle,
  region.libelle,
  type_vin.libelle
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;
```

Écrire une requête permettant de générer la partie `stocks` du document, soit un document JSON pour chaque ligne de la table `stock`, incluant le `contenant`.

Fusionner les requêtes précédentes pour générer un document complet pour chaque ligne de la table `vin`. Créer une table `stock_jsonb` avec un unique champ JSONB rassemblant ces documents.

Calculer la taille de la table, et la comparer à la taille du reste de la base.

Depuis cette nouvelle table, renvoyer l'ensemble des récoltants de la région Beaujolais.

Renvoyer l'ensemble des vins pour lesquels au moins une bouteille entre 1992 et 1995 existe. (la difficulté est d'extraire les différents stocks d'une même ligne de la table)

Indexer le document jsonb en utilisant un index de type GIN.

Peut-on réécrire les deux requêtes précédentes pour utiliser l'index ?

5.7.3 Large Objects

But : Utilisation de Large Objects

- Créer une table `fichiers` avec un texte et une colonne permettant de référencer des *Large Objects*.

- Importer un fichier local à l'aide de psql dans un large object.
- Noter l'`oid` retourné.

- Importer un fichier du serveur à l'aide de psql dans un large object.

- Afficher le contenu de ces différents fichiers à l'aide de psql.

- Les sauvegarder dans des fichiers locaux.

5.8 TRAVAUX PRATIQUES (SOLUTIONS)

5.8.1 Hstore (Optionnel)

> Afficher les informations à stocker avec la requête suivante :

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
LIMIT 10;
```

(`LIMIT 10` est là juste pour éviter de ramener tous les enregistrements).

> Créer une table `stock_denorm (vin_id int, nombre  int, attributs hstore)` et y copier le contenu de la requête. Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de `hstore` sans se soucier de formatage de chaîne de caractères. (Voir la documentation.)

Une remarque toutefois : les éléments du tableau doivent tous être de même type, d'où la conversion en text des quelques éléments entiers. C'est aussi une limitation du type `hstore` : il ne supporte que les attributs texte.

Cela donne :

```
CREATE EXTENSION hstore;

CREATE TABLE stock_denorm AS
SELECT stock.vin_id,
       stock.nombre,
       hstore(ARRAY['annee', stock.annee::text,
                    'recoltant', recoltant.nom,
                    'appellation',appellation.libelle,
                    'region',region.libelle,
                    'type_vin',type_vin.libelle,
                    'contenance',contenant.contenance::text,
                    'contenant',contenant.libelle]) AS attributs
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
```

```
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id);
```

Et l'on n'oublie pas les statistiques :

```
ANALYZE stock_denorm;
```

> Créer un index sur le champ `attributs` pour accélérer les recherches.

```
CREATE INDEX idx_stock_denorm on stock_denorm USING gin (attributs );
```

> Rechercher le nombre de bouteilles (attribut `bouteille`) en stock de vin blanc (attribut `type_vin`) d'Alsace (attribut `region`). Quel est le temps d'exécution de la requête ? Combien de buffers accédés ?

Attention au A majuscule de Alsace, les hstore sont sensibles à la casse !

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT *
FROM stock_denorm
WHERE attributs @>
'type_vin=>blanc, region=>Alsace, contenant=>bouteille';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on stock_denorm (cost=64.70..374.93 rows=91 width=193)
                                  (actual time=64.370..68.526 rows=1680 loops=1)
   Recheck Cond: (attributs @> '"region"=>"Alsace", "type_vin"=>"blanc",
                               "contenant"=>"bouteille"'::hstore)
   Heap Blocks: exact=1256
   Buffers: shared hit=1353
   ->  Bitmap Index Scan on idx_stock_denorm
            (cost=0.00..64.68 rows=91 width=0)
            (actual time=63.912..63.912 rows=1680 loops=1)
         Index Cond: (attributs @> '"region"=>"Alsace", "type_vin"=>"blanc",
                                    "contenant"=>"bouteille"'::hstore)
         Buffers: shared hit=97
 Planning time: 0.210 ms
 Execution time: 68.927 ms
```

> Refaire la même requête sur la table initiale. Qu'en conclure ?

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
```

```
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
WHERE type_vin.libelle='blanc' AND region.libelle='Alsace'
AND contenant.libelle = 'bouteille';

                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=11.64..873.33 rows=531 width=75)
              (actual time=0.416..24.779 rows=1680 loops=1)
   Join Filter: (stock.contenant_id = contenant.id)
   Rows Removed by Join Filter: 3360
   Buffers: shared hit=6292
   ->  Seq Scan on contenant  (cost=0.00..1.04 rows=1 width=16)
                              (actual time=0.014..0.018 rows=1 loops=1)
         Filter: (libelle = 'bouteille'::text)
         Rows Removed by Filter: 2
         Buffers: shared hit=1
   ->  Nested Loop  (cost=11.64..852.38 rows=1593 width=67)
                    (actual time=0.392..22.162 rows=5040 loops=1)
         Buffers: shared hit=6291
         ->  Hash Join  (cost=11.23..138.40 rows=106 width=55)
                        (actual time=0.366..5.717 rows=336 loops=1)
               Hash Cond: (vin.recoltant_id = recoltant.id)
               Buffers: shared hit=43
               ->  Hash Join  (cost=10.07..135.78 rows=106 width=40)
                              (actual time=0.337..5.289 rows=336 loops=1)
                     Hash Cond: (vin.type_vin_id = type_vin.id)
                     Buffers: shared hit=42
                     ->  Hash Join  (cost=9.02..132.48 rows=319 width=39)
                                    (actual time=0.322..4.714 rows=1006 loops=1)
                           Hash Cond: (vin.appellation_id = appellation.id)
                           Buffers: shared hit=41
                           ->  Seq Scan on vin
                                 (cost=0.00..97.53 rows=6053 width=16)
                                 (actual time=0.011..1.384 rows=6053 loops=1)
                               Buffers: shared hit=37
                           ->  Hash  (cost=8.81..8.81 rows=17 width=31)
                                     (actual time=0.299..0.299 rows=53 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 4kB
                                 Buffers: shared hit=4
                                 ->  Hash Join
                                      (cost=1.25..8.81 rows=17 width=31)
                                      (actual time=0.033..0.257 rows=53 loops=1)
                                       Hash Cond:
                                            (appellation.region_id = region.id)
                                       Buffers: shared hit=4
                                       ->  Seq Scan on appellation
                                            (cost=0.00..6.19 rows=319 width=24)
                                            (actual time=0.010..0.074 rows=319
                                             loops=1)
                                             Buffers: shared hit=3
                                       ->  Hash
```

```
                                          (cost=1.24..1.24 rows=1 width=15)
                                          (actual time=0.013..0.013 rows=1
                                           loops=1)
                                        Buckets: 1024  Batches: 1
                                                 Memory Usage: 1kB
                                        Buffers: shared hit=1
                                        ->  Seq Scan on region
                                          (cost=0.00..1.24 rows=1 width=15)
                                          (actual time=0.005..0.012 rows=1
                                                  loops=1)
                                              Filter: (libelle =
                                                       'Alsace'::text)
                                              Rows Removed by Filter: 18
                                              Buffers: shared hit=1
                    ->  Hash  (cost=1.04..1.04 rows=1 width=9)
                              (actual time=0.008..0.008 rows=1 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 1kB
                          Buffers: shared hit=1
                          ->  Seq Scan on type_vin
                                    (cost=0.00..1.04 rows=1 width=9)
                                    (actual time=0.005..0.007 rows=1 loops=1)
                                Filter: (libelle = 'blanc'::text)
                                Rows Removed by Filter: 2
                                Buffers: shared hit=1
              ->  Hash  (cost=1.07..1.07 rows=7 width=23)
                        (actual time=0.017..0.017 rows=7 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 1kB
                    Buffers: shared hit=1
                    ->  Seq Scan on recoltant
                              (cost=0.00..1.07 rows=7 width=23)
                              (actual time=0.004..0.009 rows=7 loops=1)
                          Buffers: shared hit=1
        ->  Index Scan using idx_stock_vin_annee on stock
                    (cost=0.42..6.59 rows=15 width=16)
                    (actual time=0.013..0.038 rows=15 loops=336)
              Index Cond: (vin_id = vin.id)
              Buffers: shared hit=6248
 Planning time: 4.341 ms
 Execution time: 25.232 ms
(53 lignes)
```

La requête sur le schéma normalisé est ici plus rapide. On constate tout de même qu'elle accède à 6300 buffers, contre 1300 à la requête dénormalisée, soit 4 fois plus de données. Un test identique exécuté sur des données hors du cache donne environ 80 ms pour la requête sur la table dénormalisée, contre près d'une seconde pour les tables normalisées. Ce genre de transformation est très utile lorsque le schéma ne se prête pas à une normalisation, et lorsque le volume de données à manipuler est trop important pour tenir en mémoire. Les tables dénormalisées avec hstore se prêtent aussi bien mieux aux recherches multi-critères.

5.8.2 jsonb

Pour chaque vin, le document JSON aura la structure suivante :

```
{
  vin: {
      recoltant: {
        nom: text,
        adressse: text
      },
      appellation: {
        libelle: text,
        region: text
      },
      type_vin: text
  },
  stocks: [{
    contenant: {
      contenance: real,
      libelle: text
    },
    annee: integer,
    nombre: integer
  }]
}
```

> La requête suivante permet de générer les parties `vin` du document, avec `recoltant` et `appellation`. Créer un document JSON pour chaque ligne de `vin` grâce à la fonction `jsonb_build_object`.

```
SELECT
  recoltant.nom,
  recoltant.adresse,
  appellation.libelle,
  region.libelle,
  type_vin.libelle
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;

SELECT
  jsonb_build_object(
    'recoltant',
    json_build_object('nom', recoltant.nom, 'adresse',
                      recoltant.adresse
    ),
    'appellation',
    jsonb_build_object('libelle', appellation.libelle, 'region', region.libelle),
    'type_vin', type_vin.libelle
)
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id ;
```

> Écrire une requête permettant de générer la partie `stocks` du document, soit un document JSON pour chaque ligne de la table `stock`, incluant le `contenant`.

La partie `stocks` du document est un peu plus compliquée, et nécessite l'utilisation de fonctions d'agrégations.

```
SELECT json_build_object(
  'contenant',
  json_build_object('contenance', contenant.contenance, 'libelle',
                    contenant.libelle),
  'annee', stock.annee,
  'nombre', stock.nombre)
FROM stock join contenant on stock.contenant_id = contenant.id;
```

Pour un vin donné, le tableau `stock` ressemble à cela :

```
SELECT json_agg(json_build_object(
  'contenant',
  json_build_object('contenance', contenant.contenance, 'libelle',
                    contenant.libelle),
  'annee', stock.annee,
  'nombre', stock.nombre))
FROM stock
INNER JOIN contenant on stock.contenant_id = contenant.id
WHERE vin_id = 1
GROUP BY vin_id;
```

> Fusionner les requêtes précédentes pour générer un document complet pour chaque ligne de la table `vin`. Créer une table `stock_jsonb` avec un unique champ JSONB rassemblant ces documents.

On assemble les deux parties précédentes :

```
CREATE TABLE stock_jsonb  AS (
  SELECT
    json_build_object(
    'vin',
      json_build_object(
        'recoltant',
        json_build_object('nom', recoltant.nom, 'adresse', recoltant.adresse),
        'appellation',
        json_build_object('libelle', appellation.libelle, 'region',
                          region.libelle),
        'type_vin', type_vin.libelle),
    'stocks',
      json_agg(json_build_object(
        'contenant',
        json_build_object('contenance', contenant.contenance, 'libelle',
                          contenant.libelle),
        'annee', stock.annee,
        'nombre', stock.nombre)))::jsonb as document
  FROM vin
  INNER JOIN recoltant on vin.recoltant_id = recoltant.id
```

```
  INNER JOIN appellation on vin.appellation_id = appellation.id
  INNER JOIN region on region.id = appellation.region_id
  INNER JOIN type_vin on vin.type_vin_id = type_vin.id
  INNER JOIN stock on stock.vin_id = vin.id
  INNER JOIN contenant on stock.contenant_id = contenant.id
  GROUP BY vin_id, recoltant.id, region.id, appellation.id, type_vin.id
);
```

> Calculer la taille de la table, et la comparer à la taille du reste de la base.

La table avec JSON contient toutes les mêmes informations que l'ensemble des tables normalisées de la base cave (à l'exception des `id`). Elle occupe en revanche une place beaucoup moins importante, puisque les documents individuels vont pouvoir être compressés en utilisant le mécanisme TOAST. De plus, on économise les 26 octets par ligne de toutes les autres tables.

Elle est même plus petite que la seule table `stock` :

```
 \d+
                                        Liste des relations
 Schéma |        Nom         |   Type   | Propriétaire | Persistence | Taille | …
--------+--------------------+----------+--------------+-------------+--------+-
…
 public | stock              | table    | caviste      | permanent   | 36 MB  |
 public | stock_jsonb        | table    | postgres     | permanent   | 12 MB  |
…
```

> Depuis cette nouvelle table, renvoyer l'ensemble des récoltants de la région Beaujolais.

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document #>> '{vin, appellation, region}' = 'Beaujolais';
```

> Renvoyer l'ensemble des vins pour lesquels au moins une bouteille entre 1992 et 1995 existe. (la difficulté est d'extraire les différents stocks d'une même ligne de la table)

La fonction `jsonb_array_elements` permet de convertir les différents éléments du document `stocks` en autant de lignes. La clause `LATERAL` permet de l'appeler une fois pour chaque ligne :

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb,
LATERAL jsonb_array_elements(document #> '{stocks}') as stock
WHERE (stock->'annee')::text::integer BETWEEN 1992 AND 1995;
```

> Indexer le document jsonb en utilisant un index de type GIN.

```
CREATE INDEX ON stock_jsonb USING gin (document jsonb_path_ops);
```

> Peut-on réécrire les deux requêtes précédentes pour utiliser l'index ?

Pour la première requête, on peut utiliser l'opérateur « contient » pour passer par l'index :

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document @> '{"vin": {"appellation": {"region": "Beaujolais"}}}';
```

La seconde ne peut malheureusement pas être réécrite pour tirer partie de l'index.

La dénormalisation vers un champ externe n'est pas vraiment possible, puisqu'il y a plusieurs `stocks` par ligne.

5.8.3 Large Objects

- Créer une table `fichiers` avec un texte et une colonne permettant de référencer des *Large Objects*.

```
CREATE TABLE fichiers (nom text PRIMARY KEY, data OID);
```

- Importer un fichier local à l'aide de psql dans un large object.
- Noter l'`oid` retourné.

```
psql -c "\lo_import '/etc/passwd'"
```

```
lo_import 6821285
```

```
INSERT INTO fichiers VALUES ('/etc/passwd',6821285) ;
```

- Importer un fichier du serveur à l'aide de psql dans un large object.

```
INSERT INTO fichiers SELECT 'postgresql.conf',
lo_import('/var/lib/pgsql/15/data/postgresql.conf') ;
```

- Afficher le contenu de ces différents fichiers à l'aide de psql.

```
psql -c "SELECT nom,encode(l.data,'escape') \
         FROM fichiers f JOIN pg_largeobject l ON f.data = l.loid;"
```

- Les sauvegarder dans des fichiers locaux.

```
psql -c "\lo_export loid_retourné '/home/dalibo/passwd_serveur';"
```

6/ Fonctionnalités avancées pour la performance

6.1 PRÉAMBULE

Comme tous les SGBD, PostgreSQL fournit des fonctionnalités avancées.
Ce module présente des fonctionnalités internes au moteur généralement liées aux performances.

6.1.1 Au menu

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

6.2 TABLES TEMPORAIRES

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

- N'existent que pendant la session
- Non journalisées
- Ne pas en abuser !
- Ignorées par autovacuum : ANALYZE et VACUUM manuels !
- Paramétrage :
  - `temp_buffers` : cache disque pour les objets temporaires, par session, à augmenter ?

Principe :

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

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

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

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

Aspect technique :

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

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

Exemple :

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

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

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

SELECT pg_relation_filepath ('travail') ;
 pg_relation_filepath
----------------------
 base/13746/t7_5148873

\d pg_temp_7.travail

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

Cache :

Dans les plans d'exécution avec BUFFERS, l'habituelle mention `shared` est remplacée par `local` pour les tables temporaires. En effet, leur cache disque dédié est au niveau de la session, non des *shared buffers*. Ce cache est défini par le paramètre `temp_buffers` (exprimé par session, et à 8 Mo par défaut). Ce paramètre peut être augmenté, avant la création de la table. Bien sûr, on risque de saturer la RAM en cas d'abus ou s'il y a trop de sessions, comme avec `work_mem`. Ce cache n'empêche pas l'écriture des petites tables temporaires sur le disque.

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

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

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

6.3 TABLES NON JOURNALISÉES (UNLOGGED)

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

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

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

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

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

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

- table de *spooling*/*staging* ;
- table de cache/session applicative ;
- table de travail partagée entre sessions ;
- table de travail systématiquement reconstruite avant utilisation dans le flux applicatif ;
- et de manière générale toute table contenant des données dont on peut accepter la perte sans impact opérationnel ou dont on peut regénérer aisément les données.

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

6.3.1 Tables non journalisées : mise en place

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

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

6.3.2 Bascule d'une table en/depuis unlogged

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

- réécriture

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

- passage du contenu dans les WAL !

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

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

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

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

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

6.4 JIT : LA COMPILATION À LA VOLÉE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4.2 JIT : algorithme « naïf »

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

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

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

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

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

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

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

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

                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=3403866.94..3403866.95 rows=1 width=16) (…)
   ->  Gather        (cost=3403866.19..3403866.90 rows=7 width=16)
                (actual time=11778.983..11784.235 rows=8 loops=1)
         Workers Planned: 7
         Workers Launched: 7
         ->  Partial Aggregate  (cost=3402866.19..3402866.20 rows=1 width=16)(…)
               ->  Parallel Seq Scan on bigtable  (…)
                     Filter: ((id + 2) > 500000)
                     Rows Removed by Filter: 62500
```

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

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

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

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

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

6.4.3 Quand le JIT est-il utile ?

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

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

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

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

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

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

6.5 RECHERCHE PLEIN TEXTE

Full Text Search : Recherche Plein Texte

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

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

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

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

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

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

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

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

6.5.1 Full Text Search : exemple

- Décomposition :

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

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

- Recherche sur 2 mots :

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

- Recherche sur une phrase : `phrase_totsquery`

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

Autre exemple de décomposition d'une phrase :

```
SHOW default_text_search_config ;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5.2 Full Text Search : dictionnaires

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

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

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

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

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

En demandant le détail de la configuration `french`, on peut voir qu'elle se base sur des « dictionnaires » pour chaque type d'élément qui peut être rencontré : mots, phrases mais aussi URL, entiers…

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

On peut lister ces dictionnaires :

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

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

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

[7] https://snowballstem.org/

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

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

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

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

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

CREATE TEXT SEARCH CONFIGURATION french2 (copy=french);

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

À l'usage :

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

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

Les trois versions de « PostgreSQL » ont été reconnues.

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

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

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

6.5.3 Full Text Search : stockage & indexation

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

Principe :

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

```
ALTER TABLE textes
ADD COLUMN vecteur tsvector
GENERATED ALWAYS AS (to_tsvector ('french', contenu)) STORED ;
```

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

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

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

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

Alternative : index fonctionnel

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

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

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

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

Exemple complet de mise en place de FTS :

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

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

CREATE EXTENSION unaccent ;

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

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

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

ALTER TABLE depeche ADD vect_depeche tsvector;

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

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

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

- Création de l'index associé au vecteur :

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

- Collecte des statistiques sur la table :

```
ANALYZE depeche ;
```

- Utilisation basique :

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

- Tri par pertinenence :

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

- Cette requête peut s'écrire aussi ainsi :

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

6.5.4 Full Text Search sur du JSON

- Vectorisation possible des JSON

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

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

Depuis la version 10 de PostgreSQL, une recherche FTS est directement possible sur des champs JSON. Voici un exemple :

```
CREATE TABLE commandes (info jsonb);

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

La décomposition par FTS donne :

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

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

Une recherche sur « papier » donne :

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

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

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

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

6.6 QUIZ

https://dali.bo/t1_quiz

6.7 TRAVAUX PRATIQUES

6.7.1 Tables non journalisées

But : Tester les tables non journalisées

Afficher le nom du journal de transaction courant.

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

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

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

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

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

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

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

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

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

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

Simuler un crash de l'instance PostgreSQL.

Redémarrer l'instance PostgreSQL.

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

6.7.2 Indexation Full Text

But : Tester l'indexation *Full Text*

Vous aurez besoin de la base **textes**. 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 charger dans une base préexistante avec `pg_restore` et créera juste une table nommée `textes`.

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

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

Quelle est la taille de cet index ?

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

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

6.8 TRAVAUX PRATIQUES (SOLUTIONS)

6.8.1 Tables non journalisées

> Afficher le nom du journal de transaction courant.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> Simuler un crash de l'instance PostgreSQL.

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

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

```
$ kill -9 697
```

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

> Redémarrer l'instance PostgreSQL.

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

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

```
SELECT count(*) FROM pgbench_accounts ;

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

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

6.8.2 Indexation Full Text

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

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

> Quelle est la taille de cet index ?

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

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

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

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

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

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

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

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

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

10 lignes sont ramenées en quelques millisecondes :

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

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

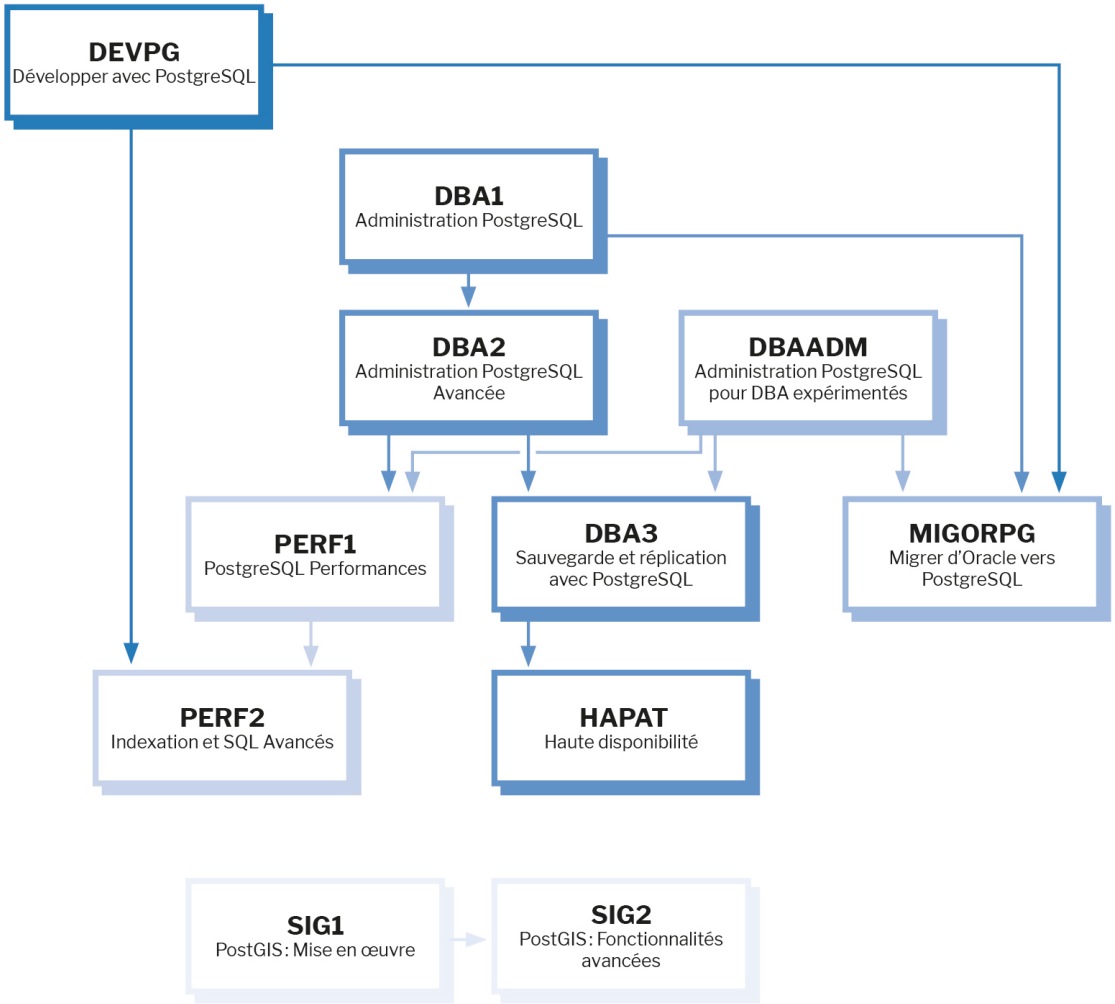
Noter que les pluriels « couteaux » et « affaires » figurent parmi les résultats puisque la recherche porte sur les lexèmes `'affair'' & ''couteau'`.

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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