Module S22

Types avancés

25.09

Table des matières

Sur ce document

Formation	Module S22
Titre	Types avancés
Révision	25.09
PDF	https://dali.bo/s22_pdf
EPUB	https://dali.bo/s22_epub
HTML	https://dali.bo/s22_html
Slides	https://dali.bo/s22_slides
TP	https://dali.bo/s22_tp
TP (solutions)	https://dali.bo/s22_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cette licence interdit la réutilisation pour l'apprentissage d'une IA. Elle couvre les diapositives, les manuels eux-mêmes et les travaux pratiques.

Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

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

Versions de PostgreSQL couvertes

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ Types avancés

PostgreSQL offre des types avancés :

- UUID
- Tableaux
- Composés :
  - `hstore`
  - JSON : `json`, `jsonb`
  - XML
- Pour les objets binaires :
  - `bytea`
  - Large Objects

1.1 UUID

1.1.1 UUID : principe

— Ex : `d67572bf-5d8c-47a7-9457-a9ddce259f05`
— Un identifiant universellement unique sur 128 bits
— Type : `uuid`
— Avec des inconvénients…

Les UUID (pour *Universally Unique IDentifier*) sont nés d'un besoin d'avoir des identifiants uniques au niveau mondial pour divers objets, avec un risque de collision théoriquement négligeable. Ce sont des identifiants sur 128 bits.

Le standard propose plusieurs versions à cause d'un historique déjà long depuis les années 1980, et de différents algorithmes de création ou d'utilisation dans des bases de données. Il existe aussi des versions dérivées liées à certains éditeurs.

Dans une base, les clés primaires « techniques » (*surrogate*), servent à identifier de manière unique une ligne, sans posséder de sens propre : les UUID peuvent donc parfaitement remplacer les numéros de séquence traditionnels. Ce n'est pas toujours une bonne idée.

Références :

— Pages Wikipédia francophone[1] et anglophone[2] ;
— RFC 4122[3], ISO/IEC 9834-8 :2005 (juillet 2005), définissant les UUID versions 1 à 5 et leurs utilisations ;
— RFC 9562[4] (mai 2024), définissant les UUID versions 1 à 8 ;
— Un excellent article de Victor Adossi sur le choix de différents types de clés primaires[5], les différents types d'UUID, et des benchmarks de génération d'UUID en masse (septembre 2022).

[1] https://fr.wikipedia.org/wiki/Universally_unique_identifier
[2] https://fr.wikipedia.org/wiki/Universally_unique_identifier
[3] https://datatracker.ietf.org/doc/html/rfc4122
[4] https://datatracker.ietf.org/doc/html/rfc9562
[5] https://supabase.com/blog/choosing-a-postgres-primary-key

1.1.2 UUID : avantages

- Faciles à générer
  - `gen_random_uuid()` et d'autres
- Pouvoir désigner des entités arbitrairement
- Pouvoir exporter des données de sources différentes
  - au contraire des séquences traditionnelles
- Pouvoir fusionner des bases
- Pour toutes les clés primaires ?

Sous PostgreSQL, nous verrons que de simples fonctions comme `gen_random_uuid()`, ou celles de l'extension standard `uuid-ossp`, permettent de générer des UUID aussi facilement que des numéros de séquences. Il est bien sûr possible que ces UUID soient fournis par des applications extérieures.

Généralement, les clés primaires des tables proviennent d'entiers générés successivement (séquences), généralement en partant de 1. L'unicité des identifiants est ainsi facilement garantie. Cela ne pose aucun souci jusqu'au jour où les données sont à rapprocher de données d'une autre base. Il y a de bonnes chances que les deux bases utilisent les mêmes identifiants pour des choses différentes. Souvent, une clé fonctionnelle (unique aussi) permet de faire le lien (commande `DALIBO-CRA-1234`, personne de numéro `25502123123`…) mais ce n'est pas toujours le cas et des erreurs de génération sont possibles. Un UUID arbitraire et unique est une solution facile pour nommer n'importe quelle entité logique ou physique sans risque de collision avec les identifiants d'un autre système.

Les UUID sont parfaits s'il y a des cas où il faut fusionner des bases de données issues de plusieurs bases de même structure. Cela peut arriver dans certains contextes distribués ou multitenants.

Hormis ce cas particulier, ils sont surtout utiles pour identifier un ensemble de données échangés entre deux systèmes (que ce soit en JSON, CSV ou un autre moyen) : l'UUID devient une sorte de clé primaire publique. En interne, les deux bases peuvent continuer à utiliser des séquences classiques pour leurs jointures.

Il est techniquement possible de pousser la logique jusqu'au bout et de décider que chaque clé primaire d'une ligne sera un UUID et non un entier, et de joindre sur ces UUID.

Des numéros de séquence consécutifs peuvent se deviner (dans l'URL d'un site web par exemple), ce qui peut être une faille de sécurité. Des UUID (apparemment) aléatoires ne présentent pas ce problème… si l'on a bien choisi la version d'UUID (voir plus loin).

1.1.3 UUID : inconvénients

- Lisibilité
- Temps de génération (mineur)
- Taille
  - 16 octets…
- Surtout : fragmentation des index
  - mauvais pour le cache
  - sauf UUID v7 (tout récent)

Lisibilité :

Le premier inconvénient n'est pas technique mais humain : il est plus aisé de lire et retenir des valeurs courtes comme un ticket 10023, une commande 2024-67 ou une immatriculation AT-389-RC que « d67572bf-5d8c-47a7-9457-a9ddce259f05 ». Les UUID sont donc à réserver aux clés techniques. De même, un développeur qui consulte une base retiendra et discernera plus facilement des valeurs entre 1 000 et 100 000 que des UUID à première vue aléatoires, et surtout à rallonge.

Pour la base de données, il y a d'autres inconvénients :

Taille :

Le type `uuid` de PostgreSQL prend 128 bits, donc 16 octets. Les types numériques entiers de PostgreSQL[6] utilisent 2 octets pour un `smallint` (`int2`, de -32768 à +32767), 4 pour un `integer` (de -2 à +2 milliards environ), 8 pour un `bigint` (`int8`, de -9.10^{18} à $+9.10^{18}$ environ). Ces types entiers suffisent généralement à combler les besoins, tout en permettant de choisir le type le plus petit possible. On a donc une différence de 8 octets par ligne entre `uuid` et `bigint`, à multiplier par autant de lignes, parfois des milliards.

Cette différence s'amplifie tout le long de l'utilisation de la clé :

- index plus gros (puisque ces champs sont toujours indexés) ;
- clés étrangères plus grosses, avec leurs index ;
- jointures plus gourmandes en CPU, mémoire, voire disque ;
- avec un impact sur le cache.

Ce n'est pas forcément bloquant si votre utilisation le nécessite.

Le pire est le stockage d'UUID dans un champ `varchar` : la taille passe à 36, les jointures sont bien plus lourdes, et la garantie d'avoir un véritable UUID disparaît !

Temps de génération :

Selon l'algorithme de génération utilisé, la création d'un UUID peut être plusieurs fois plus lente que celle d'un numéro de séquence. Mais ce n'est pas vraiment un souci avec les processeurs modernes,

[6] https://docs.postgresql.fr/current/datatype-numeric.html

qui sont capables de générer des dizaines, voire des centaines de milliers d'UUID, aléatoires ou pas, par seconde.

Fragmentation des index :

Le plus gros problème des UUID vient de leur apparence aléatoire. Cela a un impact sur la fragmentation des index et leur utilisation du cache.

Parlons d'abord de l'insertion de nouvelles lignes. Par défaut, les UUID sont générés en utilisant la version 4. Elle repose sur un algorithme générant des nombres aléatoires. Par conséquent, les UUID produits sont imprévisibles. Cela peut entraîner de fréquents *splits* des pages d'index (division d'une page pleine qui doit accueillir de nouvelles entrées). Les conséquences directes sont la fragmentation de l'index, une augmentation de sa taille (avec un effet négatif sur le cache), et l'augmentation du nombre d'accès disques (en lecture et écriture).

De plus, toujours avec des UUID version 4, comme les mises à jour sont réparties sur toute la longueur des index, ceux-ci tendent à rester entièrement dans le cache de PostgreSQL. Si celui-ci est insuffisant, des accès disques aléatoires fréquents peuvent devenir gênants.

À l'inverse, une séquence génère des valeurs strictement croissantes, donc toujours insérées à la fin de l'index. Non seulement la fragmentation reste basse, mais la partie utile de l'index, en cache, reste petite.

Évidemment, tout cela devient plus complexe quand on modifie ensuite les lignes. Mais beaucoup d'applications ont tendance à modifier surtout les lignes récentes, et délaissent les blocs d'index des lignes anciennes.

Pour un index qui reste petit, donc une table statique ou dont les anciennes lignes sont vite supprimées, ce n'est pas vraiment un problème. Mais un modèle où chaque clé de table et chaque clé étrangère est un index a intérêt à pouvoir garder tous ces index en mémoire.

Récemment, une solution standardisée est apparue avec les UUID version 7 (standardisés dans la RFC 9562[7] en 2024) : ces UUID utilisent l'heure de génération et sont donc triés. Le souci de pollution du cache disparaît donc.

1.1.4 UUID : utilisation sous PostgreSQL

- `uuid` :
  - type simple (16 octets)
  - garantit le format
- Génération :
  - `gen_random_uuid` (v4, aléatoire)
  - extension `uuid-ossp` (v1,3,4,5)
  - extension (`pg_idkit`…) ou fonction en SQL (v7)

[7] https://datatracker.ietf.org/doc/html/rfc9562

Le type `uuid` est connu de PostgreSQL, c'est un champ simple de taille fixe. Si l'UUID provient de l'extérieur, le type garantit qu'il s'agit d'un UUID valide.

gen_random_uuid() :

Générer un UUID depuis le SQL est très simple avec la fonction `gen_random_uuid()` [8] :

```
SELECT gen_random_uuid() FROM generate_series (1,4) ;
```

```
           gen_random_uuid
--------------------------------------
 d1ac1da0-4c0c-4e56-9302-72362cc5726c
 c32fa82d-a2c1-4520-8b70-95919c6cb15f
 dd980a9c-05a8-4659-a1e7-ca7836bc7da7
 27de59d3-60bc-43b9-8d03-4779a1a01e47
```

Les UUID générés sont de version 4, c'est-à-dire totalement aléatoires, avec tous les inconvénients vus ci-dessus.

uuid-ossp :

La fonction `gen_random_uuid()` n'est disponible directement que depuis PostgreSQL 13. Auparavant, il fallait forcément utiliser une extension : soit `pgcrypto` [9], qui fournissait cette fonction, soit `uuid-ossp` [10], toutes deux livrées avec PostgreSQL. `uuid-ossp` reste utile car elle fournit plusieurs algorithmes de génération d'UUID avec les fonctions suivantes.

Avec `uuid_generate_v1()`, l'UUID généré est lié à l'adresse MAC de la machine et à l'heure.

Cette propriété peut faciliter la prédiction de futures valeurs d'UUID. Les UUID v1 peuvent donc être considérés comme une faille de sécurité dans certains contextes.

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp" ;

SELECT uuid_generate_v1() from generate_series(1,5) ;
```

```
           uuid_generate_v1
--------------------------------------
 82e94192-45e3-11ef-92e5-04cf4b21f39a
 82e94193-45e3-11ef-92e5-04cf4b21f39a
 82e94194-45e3-11ef-92e5-04cf4b21f39a
 82e94195-45e3-11ef-92e5-04cf4b21f39a
```

Sur une autre machine :

```
SELECT uuid_generate_v1(),pg_sleep(5) from generate_series(1,5) ;
```

```
           uuid_generate_v1           | pg_sleep
--------------------------------------+----------
 ef5078b4-45e3-11ef-a2d4-67bc5acec5f2 |
 f24c2982-45e3-11ef-a2d4-67bc5acec5f2 |
```

[8] https://www.postgresql.org/docs/current/functions-uuid.html
[9] https://www.postgresql.org/docs/current/pgcrypto.html
[10] https://www.postgresql.org/docs/current/uuid-ossp.html

```
 f547b552-45e3-11ef-a2d4-67bc5acec5f2 |
 f84345aa-45e3-11ef-a2d4-67bc5acec5f2 |
 fb3ed120-45e3-11ef-a2d4-67bc5acec5f2 |
```

Noter que le problème de fragmentation des index se pose déjà.

Il existe une version `uuid_generate_v1mc()` un peu plus sécurisée.

`uuid_generate_v3()` et `uuid_generate_v5()` génèrent des valeurs reproductibles en fonction des paramètres. La version 5 utilise un algorithme plus sûr.

`uuid_generate_v4` génère un UUID totalement aléatoire, comme `gen_random_uuid()`.

UUID version 7 :

PostgreSQL ne sait pas encore générer d'UUID en version 7. Il existe cependant plusieurs extensions dédiées, avec les soucis habituels de disponibilité de paquets, maintenance des versions, confiance dans le mainteneur et disponibilité dans un PostgreSQL en SaaS. Par exemple, Supabase propose `pg_idkit` (versions Rust[11], et PL/pgSQL[12]).

Le plus simple est sans doute d'utiliser la fonction SQL suivante, de Kyle Hubert, modifiée par Daniel Vérité[13]. Elle est sans doute suffisamment rapide pour la plupart des besoins.

```
CREATE FUNCTION uuidv7() RETURNS uuid
AS $$
  -- Replace the first 48 bits of a uuidv4 with the current
  -- number of milliseconds since 1970-01-01 UTC
  -- and set the "ver" field to 7 by setting additional bits
  SELECT encode(
    set_bit(
      set_bit(
        overlay(uuid_send(gen_random_uuid()) placing
        substring(int8send((extract(epoch FROM clock_timestamp())*1000)::bigint)
                  FROM 3)
          FROM 1 for 6),
        52, 1),
      53, 1), 'hex')::uuid;
$$ LANGUAGE sql VOLATILE;
```

Il existe une version plus lente avec une précision inférieure à la milliseconde. Le même billet de blog offre une fonction retrouvant l'heure de création d'un UUID v7 :

```
CREATE FUNCTION uuidv7_extract_timestamp(uuid) RETURNS timestamptz
AS $$
 SELECT to_timestamp(
   right(substring(uuid_send($1) FROM 1 for 6)::text, -1)::bit(48)::int8
    /1000.0);
$$ LANGUAGE sql IMMUTABLE STRICT;

-- 10 UUID v 7 espacés de 3 secondes
WITH us AS  (SELECT uuidv7() AS u, pg_sleep(3)
             FROM generate_series (1,10))
SELECT u, uuidv7_extract_timestamp(u)
FROM us ;
```

[11] https://github.com/VADOSWARE/pg_idkit
[12] https://github.com/kiwicopple/pg-extensions/tree/main/pg_idkit
[13] https://postgresql.verite.pro/blog/2024/07/15/uuid-v7-pure-sql.html

```
                  u                   |  uuidv7_extract_timestamp
--------------------------------------+---------------------------
 0190cbaf-7879-7a4c-9ee3-8d383157b5cc | 2024-07-19 17:49:52.889+02
 0190cbaf-8435-7bb8-8417-30376a2e7251 | 2024-07-19 17:49:55.893+02
 0190cbaf-8fef-7535-8fd6-ab7316259338 | 2024-07-19 17:49:58.895+02
 0190cbaf-9baa-74f3-aa9e-bf2d2fa84e68 | 2024-07-19 17:50:01.898+02
 0190cbaf-a766-7ef6-871d-2f25e217a6ea | 2024-07-19 17:50:04.902+02
 0190cbaf-b321-717b-8d42-5969de7e7c1e | 2024-07-19 17:50:07.905+02
 0190cbaf-bedb-79c1-b67d-0034d51ac1ad | 2024-07-19 17:50:10.907+02
 0190cbaf-ca95-7d70-a8c0-f4daa60cbe21 | 2024-07-19 17:50:13.909+02
 0190cbaf-d64f-7ffe-89cd-987377b2cc07 | 2024-07-19 17:50:16.911+02
 0190cbaf-e20a-7260-95d6-32fec0a7e472 | 2024-07-19 17:50:19.914+02
```

Ils sont classés à la suite dans l'index, ce qui est tout l'intérêt de la version 7.

Noter que cette fonction économise les 8 octets par ligne d'un champ `creation_date`, que beaucoup de développeurs ajoutent.

Création de table :

Utilisez une clause `DEFAULT` pour générer l'UUID à la volée :

```
CREATE TABLE test_uuidv4 (id uuid DEFAULT ( gen_random_uuid() ) PRIMARY KEY,
                        <autres champ>…) ;

CREATE TABLE test_uuidv7 (id uuid DEFAULT ( uuidv7() ) PRIMARY KEY,
                        <autres champ>…) ;
```

1.1.5 UUID : une indexation facile

```
CREATE UNIQUE INDEX ON nomtable USING btree (champ_uuid) ;
```

L'index B-tree classique convient parfaitement pour trier des UUID. En général on le veut `UNIQUE` (plus pour parer à des erreurs humaines qu'à de très improbables collisions dans l'algorithme de génération).

1.1.6 UUID : résumé

— Si vous avez besoin des UUID, préférez la version 7.
— Les séquences habituelles restent recommandables en interne.

Si des données doivent être échangées avec d'autres systèmes, les UUID sont un excellent moyen de garantir l'unicité d'identifiants universels.

Si vous les générez vous-mêmes, préférez les UUID version 7. Des UUID v4 (totalement aléatoires) restent sinon recommandables, avec les soucis potentiels de cache et de fragmentation évoqués ci-dessus.

Pour les jointures internes à l'applicatif, conservez les séquences habituelles, (notamment avec `GENERATED ALWAYS AS IDENTITY`), ne serait-ce que pour leur simplicité.

1.2 TYPES TABLEAUX

1.2.1 Tableaux : principe

— Collection ordonnée d'un même type
— Types `integer[]`, `text[]`, etc.

```
SELECT ARRAY [1,2,3] ;

SELECT '{1,2,3}'::integer[] ;

-- lignes vers tableau
SELECT array_agg (i) FROM generate_series (1,3) i ;

-- tableau vers lignes
SELECT unnest ( '{1,2,3}'::integer[] ) ;

CREATE TABLE demotab ( id int, liste integer[] ) ;
```

Une tableau est un ensemble d'objets d'un même type. Ce type de base est généralement un numérique ou une chaîne, mais ce peut être un type structuré (géométrique, JSON, type personnalisé…), voire un type tableau. Les tableaux peuvent être multidimensionnels.

Un tableau se crée par exemple avec le constructeur `ARRAY`, avec la syntaxe `{…}::type[]`, ou en agrégeant des lignes existantes avec `array_agg`. À l'inverse, on peut transformer un tableau en lignes grâce à la fonction `unnest`. Les syntaxes `[numéro]` et `[début:fin]` permettent d'extraire un élément ou une partie d'un tableau. Deux tableaux se concatènent avec `||`.

Les tableaux sont ordonnés, ce ne sont pas des ensembles. Deux tableaux avec les mêmes données dans un ordre différent ne sont pas identiques.

Références :

— Documentation officielle :
  — syntaxe et exemples[14] ;
  — constructeur de tableau ARRAY[15] ;
  — fonctions et opérateurs tableau[16] : `array_to_string`, `string_to_array`, `unnest`, `array_agg`, `array_length`, `array_cat`, `array_append`, `array_prepend`, `cardinality`, `array_position` / `array_positions`, `array_fill`, `array_remove`, `array_shuffle`, `trim_array`…

[14] https://docs.postgresql.fr/current/arrays.html
[15] https://docs.postgresql.fr/current/sql-expressions.html#SQL-SYNTAX-ARRAY-CONSTRUCTORS
[16] https://docs.postgresql.fr/current/arrays.html

1.2.2 Tableaux : recherche de valeurs

```
-- Recherche de valeurs (toutes)
SELECT * FROM demotab
WHERE liste @> ARRAY[15,11] ;

-- Au moins 1 élément commun ?
SELECT * FROM demotab
WHERE liste && ARRAY[11,55] ;

-- 1 tableau exact
SELECT * FROM demotab
WHERE liste = '{11,55}'::int[] ;
```

```
CREATE TABLE demotab ( id int, liste int[] ) ;

INSERT INTO demotab (id, liste)
SELECT  i, array_agg (j)
FROM    generate_series (1,5) i,
LATERAL generate_series (i*10, i*10+5) j
GROUP BY i
;

TABLE demotab ;
```

```
 id |        liste
----+---------------------
  1 | {10,11,12,13,14,15}
  3 | {30,31,32,33,34,35}
  5 | {50,51,52,53,54,55}
  4 | {40,41,42,43,44,45}
  2 | {20,21,22,23,24,25}
```

Recherchons des lignes contenant certaines valeurs :

```
-- Ceci échoue car 11 n'est le PREMIER élément sur aucune ligne
SELECT * FROM demotab
WHERE liste[1] = 11 ;

-- Recherche de la ligne qui contient 11
SELECT * FROM demotab
WHERE liste @> ARRAY[11] ;
```

```
 id |        liste
----+---------------------
  1 | {10,11,12,13,14,15}
```

```
-- Recherche de la ligne qui contient 11 ET 15 (ordre indifférent)
SELECT * FROM demotab
WHERE liste @> ARRAY[15,11] ;
```

```
 id |        liste
----+---------------------
  1 | {10,11,12,13,14,15}
```

```
-- Recherche des deux lignes contenant 11 OU 55 (éléments communs)
SELECT * FROM demotab
WHERE liste && ARRAY[11,55] ;
```

```
 id |        liste
----+---------------------
  1 | {10,11,12,13,14,15}
  5 | {50,51,52,53,54,55}
```

1.2.3 Tableaux : performances

Quel intérêt par rapport à 1 valeur par ligne ?

- Allège certains modèles en réduisant les jointures
  - ex : champ avec des n°s de téléphone
  - si pas de contraintes
- Grosse économie en place (*time series*)
  - 24 octets par ligne
  - et parfois mécanisme TOAST
- Mais…
  - mise à jour compliquée/lente
  - indexation moins simple

Une bonne modélisation aboutit en général à des valeurs uniques, chacune dans son champ sur sa ligne. Les tableaux stockent plusieurs valeurs dans un même champ d'une ligne, et vont donc à l'encontre de cette bonne pratique.

Cependant les tableaux peuvent être très pratiques pour alléger la modélisation sans tomber dans de mauvaises pratiques. Typiquement, on remplacera :

```
-- Mauvaise pratique : champs identiques séparés à nombre fixe
CREATE TABLE personnes ( …
  telephone1  text,
  telephone2  text … ) ;
```

par :

```
CREATE TABLE personnes ( …
  telephones text[]  ) ;
```

Une table des numéros de téléphone serait *stricto censu* plus propre et flexible, mais induirait des jointures supplémentaires. De plus, il est impossible de poser des contraintes de validation (`CHECK`) sur des éléments de tableau sans créer un type intermédiaire. (Dans des cas plus complexes où il faut typer le numéro, on peut utiliser un tableau d'un type structuré, ou basculer vers un type JSON, qui peut lui-même contenir des tableaux, mais a un maniement un peu moins évident. L'intérêt du type structuré sur un champ JSON ou hstore est qu'il est plus compact, mais évidemment sans aucune flexibilité.)

Quand il y a beaucoup de lignes et peu de valeurs sur celles-ci, il faut se rappeler que chaque ligne d'une base de données PostgreSQL a un coût d'au moins 24 octets de données « administratives »,

même si l'on ne stocke qu'un entier par ligne, par exemple. Agréger des valeurs dans un tableau permet de réduire mécaniquement le nombre de lignes et la volumétrie sur le disque et celles des écritures. Par contre, le développeur devra savoir comment utiliser ces tableaux, comment retrouver une valeur donnée à l'intérieur d'un champ multicolonne, et comment le faire efficacement.

De plus, si le champ concaténé est assez gros (typiquement 2 ko), le mécanisme TOAST[17] peut s'activer et procéder à la compression du champ tableau, ou à son déport dans une table système de manière transparente.

Les tableaux peuvent donc permettre un gros gain de volumétrie. De plus, les données d'un même tableau, souvent utilisées ensemble, restent forcément dans le même bloc. L'effet sur le cache est donc extrêmement intéressant.

Par exemple, ces deux tables contiennent 6,3 millions de valeurs réparties sur 366 jours :

```
-- Table avec 1 valeur/ligne
CREATE TABLE serieparligne (d  timestamptz PRIMARY KEY,
                            valeur int );

INSERT INTO serieparligne
SELECT d, extract (hour from d)
FROM generate_series ('2020-01-01'::timestamptz,
                      '2020-12-31'::timestamptz, interval '5 s') d ;

SET default_toast_compression TO lz4 ; -- conseillé pour PG >= 14

-- Table avec les 17280 valeurs du jour sur 366 lignes :
CREATE TABLE serieparjour (d         date  PRIMARY KEY,
                           valeurs   int[]
                           ) ;

INSERT INTO serieparjour
SELECT d::date, array_agg ( extract (hour from d) )
FROM generate_series ('2020-01-01'::timestamptz,
                      '2020-12-31'::timestamptz, interval '5 s') d
GROUP BY d::date ;
```

La différence de taille est d'un facteur 1000 :

```
ANALYZE serieparjour,serieparligne ;

SELECT
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltuples AS nb_lignes_estimees,
    pg_size_pretty(pg_table_size(c.oid)) AS " Table (dont TOAST)",
    pg_size_pretty(pg_relation_size(c.oid)) AS "  Heap",
    pg_size_pretty(pg_relation_size(reltoastrelid)) AS "  Toast",
    pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
    pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname like 'seriepar%' ;

-[ RECORD 1 ]-------+--------------------
```

[17] https://dali.bo/m4_html#m%C3%A9canisme-toast

```
table                | public.serieparjour
nb_lignes_estimees   | 366
 Table (dont TOAST)  | 200 kB
  Heap               | 168 kB
  Toast              | 0 bytes
 Index               | 16 kB
Total                | 216 kB
-[ RECORD 2 ]-------+--------------------
table                | public.serieparligne
nb_lignes_estimees   | 6.3072e+06
 Table (dont TOAST)  | 266 MB
  Heap               | 266 MB
  Toast              | ø
 Index               | 135 MB
Total                | 402 MB
```

Ce cas est certes extrême (beaucoup de valeurs par ligne et peu de valeurs distinctes).

Les cas réels sont plus complexes, avec un horodatage moins régulier. Par exemple, l'outil OPM stocke plutôt des tableaux d'un type composé d'une date et de la valeur relevée[18], et non la valeur seule.

Dans beaucoup de cas, cette technique assez simple évite de recourir à des extensions spécialisées payantes comme TimescaleDB[19], voire à des bases de données spécialisées.

Évidemment, le code devient moins simple. Selon les besoins, il peut y avoir besoin de stockage temporaire, de fonctions de compactage périodique…
Il devient plus compliqué de retrouver une valeur précise. Ce n'est pas trop un souci dans les cas pour une recherche ou pré-sélection à partir d'un autre critère (ici la date, indexée). Pour la recherche dans les tableaux, voir plus bas.
Les mises à jour des données à l'intérieur d'un tableau deviennent moins faciles et peuvent être lourdes en CPU avec de trop gros tableaux.

1.2.4 Tableaux : indexation

- B-tree inutilisable
- GIN plus adapté pour recherche d'une valeur
  - opérateurs `&&` , `@>`
  - mais plus lourd

Indexer un champ tableau est techniquement possible avec un index B-tree, le type d'index par défaut. Mais cet index est en pratique peu performant. De toute façon il ne permet que de chercher un tableau entier (ordonné) comme critère.

[18] https://github.com/OPMDG/opm-core/blob/ae89f025407ab144e1e30abd7d6580f258945d61/pg/opm_core--2.6.sql#L41
[19] https://www.timescale.com/products

Dans les exemples précédents, les index B-tree sont plutôt à placer sur un autre champ (la date, l'ID), qui ramène une ligne entière.

D'autres cas nécessitent de chercher une valeur parmi celles des tableaux (par exemple dans des listes de propriétés). Dans ce cas, un index GIN sera plus adapté, même si cet index est un peu lourd. Les opérateurs `||` et `@>` sont utilisables. La valeur recherchée peut être n'importe où dans les tableaux.

```
TRUNCATE TABLE demotab ;
-- 500 000 lignes
INSERT INTO demotab (id, liste)
SELECT i, array_agg (j)
FROM generate_series (1,500000) i,
LATERAL generate_series (mod(i*10,100000), mod(i*10,100000)+5) j
GROUP BY i ;

CREATE INDEX demotab_gin ON demotab USING gin (liste);

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demotab
WHERE liste  @> ARRAY[45] ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on demotab  (cost=183.17..2377.32 rows=2732 width=49) (actual
↪  time=0.958..1.014 rows=50 loops=1)
   Recheck Cond: (liste @> '{45}'::integer[])
   Heap Blocks: exact=50
   Buffers: shared hit=211
   ->  Bitmap Index Scan on demotab_gin  (cost=0.00..182.49 rows=2732 width=0)
↪  (actual time=0.945..0.945 rows=50 loops=1)
         Index Cond: (liste @> '{45}'::integer[])
         Buffers: shared hit=161
 Planning:
   Buffers: shared hit=4
 Planning Time: 0.078 ms
 Execution Time: 1.042 ms
```

Là encore, on récupère les tableaux entiers qui contiennent la valeur demandée. Selon le besoin, il faudra peut-être reparcourir les éléments récupérés, ce qui coûtera un peu de CPU :

```
EXPLAIN (ANALYZE)
SELECT id,
    (SELECT count(*) FROM unnest (liste) e WHERE e=45) AS nb_occurences_45
FROM demotab
WHERE liste  @> ARRAY[45] ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on demotab  (cost=23.37..2417.62 rows=2500 width=12) (actual
↪  time=0.067..0.325 rows=50 loops=1)
   Recheck Cond: (liste @> '{45}'::integer[])
   Heap Blocks: exact=50
   ->  Bitmap Index Scan on demotab_gin  (cost=0.00..22.75 rows=2500 width=0) (actual
↪  time=0.024..0.024 rows=50 loops=1)
         Index Cond: (liste @> '{45}'::integer[])
   SubPlan 1
     ->  Aggregate  (cost=0.13..0.14 rows=1 width=8) (actual time=0.003..0.003 rows=1
↪  loops=50)
```

```
          ->  Function Scan on unnest e  (cost=0.00..0.13 rows=1 width=0) (actual
↪  time=0.002..0.002 rows=1 loops=50)
                Filter: (e = 45)
                Rows Removed by Filter: 5
Planning Time: 0.240 ms
Execution Time: 0.388 ms
```

Quant aux recherches sur une plage de valeurs dans les tableaux, elles ne sont pas directement indexables par un index GIN.

Pour les détails sur les index GIN, voir le module J5[20].

[20] https://dali.bo/j5_html#gin-et-les-tableaux

1.3 TYPES COMPOSÉS

1.3.1 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 multicritère 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 stockent 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[21] permet

[21] https://dali.bo/m4_html#m%C3%A9canisme-toast

de déporter les champs volumineux (texte, JSON, `hstore`…) dans une table séparée gérée par PostgreSQL, éventuellement en les compressant, et cela de manière totalement transparente. On y gagne donc en simplicité de développement.

1.4 HSTORE

1.4.1 hstore : principe

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.

1.4.2 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[22].

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.

[22] https://docs.postgresql.fr/current/hstore.html

1.5 JSON

1.5.1 JSON & PostgreSQL

```
{
  "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 (requêtage)

Le format JSON[23] 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
}
```

[23] https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

Historiquement, le JSON est apparu dans PostgreSQL 9.2, mais n'est vraiment utilisable qu'avec l'arrivée du type `jsonb` (binaire) dans PostgreSQL 9.4. Ce dernier est le type à utiliser.

Les opérateurs SQL/JSON path ont été ajoutés dans PostgreSQL 12[24], suite à l'introduction du JSON dans le standard SQL :2016.

1.5.2 Type json

- Type texte
  - avec validation du format
- Réservé au stockage à l'identique
- 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[25], qui compresse les grands champs au besoin, de manière transparente pour l'utilisateur. 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.

Mais on préférera généralement le type binaire `jsonb` pour les performances, et ses fonctionnalités supplémentaires. Le seul intérêt du type `json` texte est de conserver un objet JSON sous sa forme originale, y compris l'ordre des clés, les espaces inutiles compris, et les clés dupliquées (la dernière étant celle prise en compte) :

```
SELECT '{"cle2": 0, "cle1": 6,      "cle2": 4, "cle3": 17}'::json ;

                        json
-----------------------------------------------------
 {"cle2": 0, "cle1": 6,      "cle2": 4, "cle3": 17}

SELECT '{"cle2": 0, "cle1": 6,      "cle2": 4, "cle3": 17}'::jsonb ;

               jsonb
-----------------------------------
 {"cle1": 6, "cle2": 4, "cle3": 17}
```

Une partie des exemples suivants avec le type `jsonb` est aussi applicable au `json`. Beaucoup de fonctions existent sous les deux formes (par exemple `json_build_object` et `jsonb_build_object`), mais beaucoup d'autres sont propres au type `jsonb`.

[24] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/
[25] https://dali.bo/m4_html#mécanisme-toast

1.5.3 Type jsonb

— **JSON** au format **B**inaire
— Indexation GIN
— Langage JSONPath

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.

Les gains en performance sont importants. Par exemple une requête simple comme celle-ci :

```
SELECT personne_nom->'id' FROM json.personnes;
```

passe de 5 à 1,5 s sur une machine en convertissant le champ de `json` à `jsonb` (pour ½ million de champs JSON totalisant 900 Mo environ pour chaque version, ici sans TOAST notable et avec une table intégralement en cache).

Encore plus intéressant : `jsonb` supporte les index GIN, et la syntaxe JSONPath pour le requêtage et l'extraction d'attributs. `jsonb` est donc le type le plus intéressant pour stocker du JSON dans PostgreSQL.

1.5.4 Validation du format JSON

```
SELECT  '{"auteur": "JRR Tolkien","titre":"Le Hobbit"}' IS JSON ;
-- true

SELECT  '{"auteur": "JRR Tolkien","titre":"Le Hobbit}' IS JSON ;
-- false
```

— Aussi : `IS JSON WITH/WITHOUT UNIQUE KEYS`, `IS JSON ARRAY`, `IS JSON OBJECT`, `IS JSON SCALAR`
— PostgreSQL 16+

À partir de PostgreSQL 16 existe le prédicat `IS JSON`. Il peut être appliqué sur des champs `text` ou `bytea` et évidemment sur des champs `json` et `jsonb`. Il permet de repérer notamment une faute de syntaxe comme dans le deuxième exemple ci-dessus.

Existent aussi :

— l'opérateur `IS JSON WITH UNIQUE KEYS` pour garantir l'absence de clé en doublon :

```
SELECT '{"auteur": "JRR", "auteur": "Tolkien", "titre": "Le Hobbit"}'
IS JSON WITH UNIQUE KEYS AS valid ;
```

```
 valid
----------
 f
```

```
SELECT '{"prenom": "JRR", "nom": "Tolkien", "titre": "Le Hobbit"}'
IS JSON WITH UNIQUE KEYS  AS valid ;
```

```
 valid
----------
 t
```

- l'opérateur `IS JSON WITHOUT UNIQUE KEYS` pour garantir l'absence de clé unique ;
- l'opérateur `IS JSON ARRAY` pour le bon formatage des tableaux :

```
SELECT
$$[{"auteur": "JRR Tolkien", "titre": "La confrérie de l'anneau"},
   {"auteur": "JRR Tolkien", "titre": "Les deux tours"},
   {"auteur": "JRR Tolkien", "titre": "Le retour du roi"}]$$
IS JSON ARRAY AS valid ;
```

```
 valid
-------
 t
```

- des opérateurs `IS JSON SCALAR` et `IS JSON OBJECT` pour valider par exemple le contenu de fragments d'un objet JSON.

```
-- NB : l'opérateur ->> renvoie un texte
SELECT '{"nom": "production", "version":"1.1"}'::json ->> 'version'
IS JSON SCALAR AS est_nombre ;
```

```
 est_nombre
------------
 t
```

Noter que la RFC impose qu'un JSON soit en UTF-8, qui est l'encodage recommandé, mais pas obligatoire, d'une base PostgreSQL.

1.5.5 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": ["Cosette"]
} '),  ('{
  "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": []
} '),  ('{
  "firstName": "Jacques",
  "lastName": "Dupont",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "state": "",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    {
      "type": "personnel",
      "number": "+33 1 23 45 67 89"
    },
    {
      "type": "bureau",
      "number": "07 00 00 01 23"
    }
  ],
  "children": [],
  "spouse": "Martine Durand"
} ');
```

Un champ de type `jsonb` (ou `json`) accepte tout champ JSON directement.

1.5.6 JSON : construction

```
SELECT '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb ;

SELECT jsonb_build_object ('nom', 'Lagaffe', 'prenom', 'Gaston') ;
```

Pour construire un petit JSON, le transtypage d'une chaîne peut suffire dans les cas simples. `jsonb_build_object` permet de limiter les erreurs de syntaxe.

Dans un JSON, l'ordre n'a pas d'importance.

1.5.7 JSON : Affichage des attributs

```
SELECT datas->>'firstName' AS prenom,    -- text
       datas->'address'    AS addr       -- jsonb
FROM personnes ;

SELECT  datas #>> '{address,city}',      -- text
        datas #> '{address,city}'        -- jsonb
FROM personnes ; -- text

SELECT datas['address']['city'] as villes from personnes ;  -- jsonb, v14

SELECT datas['address']->>'city' as villes from personnes ;  -- text, v14
```

— Attention : pas de contrôle de l'existence de la clé !
— et attention à la casse

Le type `json` dispose de nombreuses fonctions et opérateurs de manipulation et d'extraction.

Attention au type en retour, qui peut être du texte ou du JSON. 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
```

Pour l'affichage, la fonction `jsonb_pretty` améliore la lisibilité :

```
SELECT datas->>'firstName' AS prenom,
       jsonb_pretty (datas->'address')  AS addr
FROM personnes   ;
```

```
 prenom  |                          addr
---------+-------------------------------------------------------
 Jean    | {                                                     +
         |     "city": "Paris",                                  +
         |     "postalCode": "75002",                            +
         |     "streetAddress": "43 rue du Faubourg Montmartre"+
         | }
 Georges | {                                                     +
         |     "city": "Châteauneuf",                            +
         |     "postalCode": "45990",                            +
         |     "streetAddress": "27 rue des Moulins"             +
         | }
 Jacques | {                                                     +
         |     "city": "Paris",                                  +
         |     "state": "",                                      +
         |     "postalCode": "75002",                            +
         |     "streetAddress": "43 rue du Faubourg Montmartre"+
         | }
```

L'équivalent existe avec des chemins, avec `#>` et `#>>` :

```
SELECT datas #>> '{address,city}' AS villes FROM personnes ;
```

```
   villes
-------------
 Paris
 Châteauneuf
 Paris
```

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"
 "Paris"
```

Avec cette syntaxe, une petite astuce permet de convertir en texte sans utiliser `->>['city']` (en toute rigueur, `->>0` renverra le premier élément d'un tableau) :

```
SELECT datas['address']['city']->>0 AS villes FROM personnes ;
```

```
   villes
---------------
 Paris
 Châteauneuf
 paris
```

PostgreSQL ne contrôle absolument pas que les clés JSON (comme ici `firstName` ou `city`) sont valides ou pas. La moindre faute de frappe (ou de casse!) entraînera une valeur `NULL` en retour. C'est la conséquence de l'absence de schéma dans un JSON, contrepartie de sa souplesse.

1.5.8 Modifier un JSON

```
-- Concaténation (attention aux NULL)
SELECT '{"nom": "Durand"}'::jsonb ||
  '{"address" : {"city": "Paris", "postalcode": "75002"}}'::jsonb ;

-- Suppression
SELECT  '{"nom": "Durand",
         "address": {"city": "Paris", "postalcode": "75002"}}'::jsonb
        - 'nom' ;

SELECT  '{"nom": "Durand",
         "address": {"city": "Paris", "postalcode": "75002"}}'::jsonb
        #- '{address,postalcode}' ;

-- Modification
SELECT jsonb_set ('{"nom": "Durand", "address": {"city":
 ↪  "Paris"}}'::jsonb,
       '{address}',
      '{"ville": "Lyon" }'::jsonb) ;
```

L'opérateur `||` concatène deux `jsonb` pour donner un autre `jsonb` :

```
SELECT '{"nom": "Durand"}'::jsonb ||
  '{"address" : {"city": "Paris", "postalcode": "75002"}}'::jsonb ;
```

```
 {"nom": "Durand", "address": {"city": "Paris", "postalcode": "75002"}}
```

Comme d'habitude, le résultat est NULL si l'un des JSON est `NULL`. Dans le doute, on peut utiliser `{}` comme élément neutre :

```
SELECT '{"nom": "Durand"}'::jsonb || coalesce (NULL::jsonb, '{}') ;
```

```
 {"nom": "Durand"}
```

Pour supprimer un attribut d'un `jsonb`, il suffit de l'opérateur `-` et d'un champ texte indiquant l'attribut à supprimer. Il existe une variante avec `text[]` pour supprimer plusieurs attributs :

```
SELECT '{"nom": "Durand", "prenom": "Georges",
         "address": {"city": "Paris"}}'::jsonb
       - '{nom, prenom}'::text[] ;
```

```
 {"address": {"city": "Paris"}}
```

ainsi que l'opérateur pour supprimer un sous-attribut :

```
SELECT  '{"nom": "Durand",
         "address": {"city": "Paris", "postalcode": "75002"}}'::jsonb
        #- '{address,postalcode}' ;

 {"nom": "Durand", "address": {"city": "Paris"}}
```

La fonction `jsonb_set` modifie l'attribut indiqué dans un `jsonb` :

```
SELECT jsonb_set ('{"nom": "Durand", "address": {"city": "Paris"}}'::jsonb,
       '{address}',
      '{"ville": "Lyon" }'::jsonb) ;

 {"nom": "Durand", "address": {"ville": "Lyon"}}
```

Attention, le sous-attribut est intégralement remplacé, et non fusionné. Dans cet exemple, le code postal disparaît :

```
SELECT jsonb_set ('{"nom": "Durand",
                   "address": {"postalcode": 69001, "city": "Paris"}}'::jsonb,
       '{address}',
      '{"ville": "Lyon" }'::jsonb) ;

 {"nom": "Durand", "address": {"ville": "Lyon"}}
```

Il vaut mieux indiquer le chemin complet en second paramètre :

```
SELECT jsonb_set ('{"nom": "Durand",
                   "address": {"postalcode": 69001, "city": "Paris"}}'::jsonb,
       '{address, city}',
      '"Lyon"'::jsonb) ;

  {"nom": "Durand", "address": {"city": "Lyon", "postalcode": 69001}}
```

1.5.9 JSON, tableaux et agrégation

— Manipuler des tableaux :
  — `jsonb_array_elements()`, `jsonb_array_elements_text()`
  — `jsonb_agg()` (de JSON ou de scalaires)
  — `jsonb_agg()_strict` (sans les NULL)

```
SELECT jsonb_array_elements (datas->'phoneNumbers')->>'number'
FROM   personnes ;
```

Un JSON peut contenir un tableau de texte, nombre, ou autres JSON. Il est possible de déstructurer ces tableaux, mais il est compliqué de requêter sur leur contenu.

`jsonb_array_elements` permet de parcourir ces tableaux :

```
SELECT  datas->>'firstName' AS prenom,
        jsonb_array_elements (datas->'phoneNumbers')->>'number' AS numero
FROM   personnes ;
```

```
 prenom  |     numero
---------+----------------
 Jean    | 06 12 34 56 78
 Jean    | 07 89 10 11 12
 Georges | 06 21 34 56 78
 Georges | 07 98 10 11 12
 Jacques | +33 1 23 45 67 89
 Jacques | 07 00 00 01 23
```

Avec la syntaxe JSONPath, le résultat est le même :

```
SELECT datas->>'firstName',
       jsonb_path_query (datas, '$.phoneNumbers[*].number')->>0 AS numero
FROM   personnes ;
```

Si l'on veut retrouver un type tableau, il faut réagréger, car `jsonb_path_query` et `jsonb_array_elements` renvoient un ensemble de lignes. On peut utiliser une clause `LATERAL`, qui sera appelée pour chaque ligne (ce qui est lent) puis réagréger :

```
SELECT  datas->>'firstName' AS prenom,
        jsonb_agg (n) AS numeros_en_json  -- tableau de JSON
FROM   personnes,
LATERAL (SELECT jsonb_path_query (datas, '$.phoneNumbers[*].number') ) AS nums(n)
GROUP BY prenom ;
```

```
 prenom  |           numeros_en_json
---------+--------------------------------------
 Jean    | ["06 12 34 56 78", "07 89 10 11 12"]
 Georges | ["06 21 34 56 78", "07 98 10 11 12"]
 Jacques | ["+33 1 23 45 67 89", "07 00 00 01 23"]
```

On voit que la fonction `jsonb_agg` envoie un tableau de JSON.

Si l'on veut un tableau de textes :

```
SELECT  datas->>'firstName' AS prenom,
        array_agg ( n->>'number' ) AS telephones              -- text[]
FROM   personnes,
LATERAL (SELECT jsonb_path_query (datas, '$.phoneNumbers[*]') ) AS nums(n)
GROUP BY prenom ;
```

```
 prenom  |             telephones
---------+-------------------------------------
 Jean    | {"06 12 34 56 78","07 89 10 11 12"}
 Georges | {"06 21 34 56 78","07 98 10 11 12"}
 Jacques | {"+33 1 23 45 67 89","07 00 00 01 23"}
```

Noter que la clause `LATERAL` supprime les personnes sans téléphone. On peut utiliser `LEFT OUTER JOIN LATERAL`, ou l'exemple suivant. On suppose aussi que le prénom est une clé de regroupement suffisante ; un identifiant quelconque serait plus pertinent.

Cette autre variante avec un sous-`SELECT` sera plus performante avec de nombreuses lignes, car elle évite le regroupement :

```
SELECT  datas->>'firstName' AS prenom,
        (SELECT array_agg (nt) FROM (
          SELECT (jsonb_path_query (datas, '$.phoneNumbers[*]'))->>'number'
        ) AS nums(nt) ) AS telephones              -- text[]
FROM   personnes ;
```

Il existe une autre fonction d'agrégation des JSON plus pratique, nommée `jsonb_agg_strict()`, qui supprime les valeurs à `null` de l'agrégat (mais pas un attribut à `null`). Pour d'autres cas, il existe aussi `jsonb_strip_nulls()` pour nettoyer un JSON de toutes les valeurs `null` si une clé est associée (pas dans un tableau) :

```
SELECT jsonb_agg (usename) AS u1,
       jsonb_strip_nulls (jsonb_agg (usename)) AS u1b,
       jsonb_agg_strict (usename) AS u2
FROM pg_stat_activity \gx
```

```
-[ RECORD 1 ]----------------------------------------
u1  | ["postgres", null, "postgres", null, null, null]
u1b | ["postgres", null, "postgres", null, null, null]
u2  | ["postgres", "postgres"]
```

```
SELECT  jsonb_agg ( to_jsonb(rq) ) AS a1,
        jsonb_agg_strict ( to_jsonb(rq)  ) AS a2,
        jsonb_agg ( jsonb_strip_nulls (to_jsonb(rq) ) ) AS a3,
        jsonb_strip_nulls(jsonb_agg ( to_jsonb(rq) ) ) AS a4
FROM (
  SELECT pid, usename FROM pg_stat_activity
  ) AS rq  ;
```

```
-[ RECORD 1 ]------------------------------------------------------------
a1 | [{"pid": 1583585, "usename": "postgres"}, {"pid": 2425, "usename": null},
 ↪  {"pid": 2426, "usename": "postgres"}, {"pid": 2421, "usename": null}, {"pid":
 ↪  2422, "usename": null}, {"pid": 2424, "usename": null}]
a2 | [{"pid": 1583585, "usename": "postgres"}, {"pid": 2425, "usename": null},
 ↪  {"pid": 2426, "usename": "postgres"}, {"pid": 2421, "usename": null}, {"pid":
 ↪  2422, "usename": null}, {"pid": 2424, "usename": null}]
a3 | [{"pid": 1583585, "usename": "postgres"}, {"pid": 2425}, {"pid": 2426,
 ↪  "usename": "postgres"}, {"pid": 2421}, {"pid": 2422}, {"pid": 2424}]
a4 | [{"pid": 1583585, "usename": "postgres"}, {"pid": 2425}, {"pid": 2426,
 ↪  "usename": "postgres"}, {"pid": 2421}, {"pid": 2422}, {"pid": 2424}]
```

1.5.10 Conversions jsonb / relationnel (1)

- Construire un ensemble de tuples depuis un objet JSON :
  - `jsonb_each()`
- Avec des types préexistants :
  - `jsonb_populate_record()`, `jsonb_populate_recordset`
- Types définis à la volée :
  - `jsonb_to_record()`, `jsonb_to_recordset()`
- Construire un JSON depuis une requête :
  - `to_jsonb (<requête>)`
- Attention aux types
  - `jsonb_typeof()`

Plusieurs fonctions permettant de construire du `jsonb`, ou de le manipuler de manière ensembliste. (En version 17 ou supérieure, le plus simple et pratique sera souvent la fonction `JSON_TABLE` présentée plus bas.)

jsonb_each :

`jsonb_each` décompose les clés et retourne une ligne par clé. Là encore, on multiplie le nombre de lignes.

```
SELECT
  j.key,    -- text
  j.value   -- jsonb
FROM personnes p CROSS JOIN jsonb_each(p.datas) j ;
```

```
     key      |                                          value
--------------+-------------------------------------------------------------------
 address      | {"city": "Paris", "postalCode": "75002", "streetAddress":
              "43 rue du Faubourg Montmartre"}
 children     | ["Cosette"]
 lastName     | "Valjean"
 firstName    | "Jean"
 phoneNumbers | [{"number": "06 12 34 56 78"}, {"type": "bureau", "number":
              "07 89 10 11 12"}]
 address      | {"city": "Châteauneuf", "postalCode": "45990", "streetAddress":
              "27 rue des Moulins"}
 children     | []
 lastName     | "Durand"
 firstName    | "Georges"
 phoneNumbers | [{"number": "06 21 34 56 78"}, {"type": "bureau", "number":
              "07 98 10 11 12"}]
 age          | 27
 spouse       | "Martine Durand"
…
```

jsonb_populate_record/jsonb_populate_recordset :

Si les noms des attributs et champs sont bien identiques (casse comprise !) entre JSON et table cible, `jsonb_populate_record` peut être pratique :

```
CREATE TABLE nom_prenom_age (
"firstName" text,
"lastName" text,
age int,
present boolean) ;
-- Ceci renvoie un RECORD, peu pratique :
SELECT jsonb_populate_record (null::nom_prenom_age, datas) FROM personnes ;
-- Cette version renvoie des lignes
SELECT np.*
FROM personnes,
    LATERAL jsonb_populate_record (null::nom_prenom_age, datas) np ;

 firstName | lastName | age | present
-----------+----------+-----+---------
 Jean      | Valjean  |     |
 Georges   | Durand   |     |
 Jacques   | Dupont   |  27 |
```

Les attributs du JSON non récupérés sont ignorés, les valeurs absentes du JSON sont à NULL. Il existe une possibilité de créer des valeurs par défaut :

```
SELECT np.*
FROM personnes,
    LATERAL jsonb_populate_record (('X','Y',null,true)::nom_prenom_age, datas) np ;

 firstName | lastName | age | present
-----------+----------+-----+---------
 Jean      | Valjean  |     | t
 Georges   | Durand   |     | t
 Jacques   | Dupont   |  27 | t
```

`jsonb_populate_recordset` sert dans le cas des tableaux de JSON.

Définir un type au lieu d'une table fonctionne aussi.

jsonb_to_record/jsonb_to_recordset :

`jsonb_to_record` renvoie un enregistrement avec des champs correspondant à chaque attribut JSON, donc bien typés. Pour fonctionner, la fonction exige une clause `AS` avec les attributs voulus et leur bon type. (Là encore, attention à la casse exacte des noms d'attributs sous peine de se retrouver avec des valeurs à `NULL`.)

```
SELECT p.*
FROM personnes,
    LATERAL jsonb_to_record(datas) AS p ("firstName" text, "lastName" text);

 firstName | lastName
-----------+----------
 Jean      | Valjean
 Georges   | Durand
 Jacques   | Dupont
```

Les autres attributs de notre exemple peuvent être extraits également, ou re-convertis en enregistrements avec une autre clause `LATERAL`. Si ces attributs sont des tableaux, on peut générer une ligne par élément de tableau avec `json_to_recordset` :

```
SELECT p.*, t.*
  FROM personnes,
  LATERAL jsonb_to_record(datas) AS p ("firstName" text, "lastName" text),
  LATERAL jsonb_to_recordset (datas->'phoneNumbers') AS t ("number" json) ;
```

```
 firstName | lastName |       number
-----------+----------+--------------------
 Jean      | Valjean  | "06 12 34 56 78"
 Jean      | Valjean  | "07 89 10 11 12"
 Georges   | Durand   | "06 21 34 56 78"
 Georges   | Durand   | "07 98 10 11 12"
 Jacques   | Dupont   | "+33 1 23 45 67 89"
 Jacques   | Dupont   | "07 00 00 01 23"
```

À l'inverse, transformer le résultat d'une requête en JSON est très facile avec `to_jsonb` :

```
SELECT to_jsonb(rq) FROM (
  SELECT pid, datname, application_name FROM pg_stat_activity
  ) AS rq  ;
```

```
                                to_jsonb
-------------------------------------------------------------------------
 {"pid": 2428, "datname": null, "application_name": ""}
 {"pid": 2433, "datname": null, "application_name": ""}
 {"pid": 1404455, "datname": "postgres", "application_name": "pgbench"}
 {"pid": 1404456, "datname": "postgres", "application_name": "pgbench"}
 {"pid": 1404457, "datname": "postgres", "application_name": "pgbench"}
 {"pid": 1404458, "datname": "postgres", "application_name": "pgbench"}
 {"pid": 1404459, "datname": "postgres", "application_name": "pgbench"}
 {"pid": 1406914, "datname": "postgres", "application_name": "psql"}
 {"pid": 2425, "datname": null, "application_name": ""}
 {"pid": 2424, "datname": null, "application_name": ""}
 {"pid": 2426, "datname": null, "application_name": ""}
```

jsonb_typeof :

Pour connaître le type d'un attribut JSON :

```
SELECT jsonb_typeof (datas->'firstName'),
       jsonb_typeof (datas->'age')
FROM personnes ;
```

```
 jsonb_typeof | jsonb_typeof
--------------+--------------
 string       |
 string       |
 string       | number
```

1.5.11 Conversions jsonb / relationnel (2) : JSON_TABLE

```
SELECT * FROM t,
JSON_TABLE (
          t.champ_json ,
          '$ ? (@.age == 42)'
          COLUMNS (age int PATH '$.age',
                   nom text PATH '$.name')) ;
```

- PostgreSQL 17+
- conversion d'un JSON en vue, avec types
- clause `NESTED PATH` pour les tableaux dans le JSON
- paramétres : clause `PASSING`

À partir de PostgreSQL 17, la fonction `JSON_TABLE()` permet de présenter des données JSON sous forme de vue. C'est sans doute la manière la plus pratique.

Le premier paramètre que prend la fonction, appelé `context_item`, est une source JSON sur laquelle sera évaluée l'expression JSON. Il est possible de récupérer des données présentes dans une structure imbriquée, avec la clause `NESTED PATH`.

Exemple :

Prenons l'exemple suivant d'une table créée à partir d'un fichier JSON. Le fichier d'exemple généré à l'aide de mockaroo.com peut être récupéré sur https://dali.bo/999people.

```
# Récupération du fichier d'exemple
curl -L https://dali.bo/999people -o /tmp/999people.json

-- Récupération du JSON, par exemple ainsi :
-- Table de travail
CREATE TEMP TABLE t0 (coll jsonb) ;
-- Chargement du JSON en modifiant les délimiteurs du CSV
\copy t0 (coll) FROM '/tmp/999people.json' CSV quote e'\x01' delimiter e'\x02' ;
-- Les 999 lignes sont converties en autant de lignes
CREATE TABLE t1 (uid int GENERATED ALWAYS AS IDENTITY, utilisateur jsonb);
INSERT INTO t1 (utilisateur)
SELECT jsonb_array_elements (coll) AS utilisateur from t0 ;
-- Insertion d'une nouvelle ligne
INSERT INTO t1 (utilisateur)
SELECT '{
  "guid": "c39c493b-6759-47b8-9c3f-10b1b59ba8ef",
  "isActive": false,
  "balance": "$3,363.44",
  "age": 23,
  "name": "Claudine Howell",
  "gender": "female",
  "email": "claudinehowell@aquasseur.com",
  "address": "482 Fiske Place, Ogema, Pennsylvania, 1753",
  "registered": "2023-12-22T05:16:27 -01:00",
```

```
  "latitude": -88.430942,
  "longitude": 34.196733
}'::jsonb ;
```

Voici un exemple d'une ligne contenue dans la table `t1` :

```
SELECT utilisateur FROM t1 WHERE uid=1000 ;
```

```
 {"age": 23, "guid": "c39c493b-6759-47b8-9c3f-10b1b59ba8ef", "name": "Claudine
  ↪  Howell", "email": "claudinehowell@aquasseur.com", "gender": "female",
  ↪  "address": "482 Fiske Place, Ogema, Pennsylvania, 1753", "balance":
  ↪  "$3,363.44", "isActive": false, "latitude": -88.430942, "longitude": 34.196733,
  ↪  "registered": "2023-12-22T05:16:27 -01:00"}
```

Si l'on souhaite récupérer une vue avec uniquement les informations de l'âge et du nom présents dans les lignes de la table `t1`, la commande suivante peut être utilisée :

```
SELECT resultats.*
FROM t1,
  JSON_TABLE(utilisateur,
             '$' COLUMNS (age int PATH '$.age', nom text PATH '$.name'))
  AS resultats ;
```

```
 age |        nom
-----+--------------------
  40 | Selma Tippell
  39 | Leonardo Coupland
  83 | Jessica Riggott
  81 | Smith Caroll
  59 | Marthe MacCaig
…
  66 | Hurley Cosgra
  23 | Claudine Howell
(1000 lignes)
```

- `JSON_TABLE(utilisateur, '$'` indique que toute la source JSON (le champ de nom `utilisateur`) doit être utilisée lors de l'évaluation. Cette fonction sera appelée une fois pour chaque ligne de `t1`.
- `COLUMNS (age int PATH '$.age', name text PATH '$.name'))` est la clause qui permet d'indiquer quelles colonnes seront présentes dans la vue résultante et où retrouver les données associées.
- La clause `PATH` indique l'emplacement de la donnée dans la donnée JSON.

Le résultat peut être manipulé comme tout résultat de requête : consultation, export, vue…

Les champs sont correctement typés (suivant ceux indiqués dans `COLUMNS`). En cas d'incohérence, il n'y aura pas d'erreur, mais une valeur `NULL`.

Filtrage des valeurs :

La clause `'$'` peut être plus complexe pour ne renvoyer que certaines lignes :

```
SELECT resultats.*
FROM t1,
  JSON_TABLE(t1.utilisateur,
             '$ ? (@.age == 23)'
             COLUMNS (age int PATH '$.age', nom text PATH '$.name'))
  AS resultats ;
```

```
 age |        nom
-----+--------------------
  23 | Cleo Folkard
  23 | Olivie Schenfisch
…
  23 | Ernestine Pugsley
  23 | Tobit Herrema
  23 | Ainsley Wootton
  23 | Claudine Howell
(17 lignes)
```

La syntaxe utilisée pour les filtrages et sélections de champs est le JSON path, qui sera détaillée plus loin.

Nested path :

La clause `NESTED PATH` permet d'exploiter des tableaux à l'intérieur d'un champ JSON tel que celui-ci, ce qui multiplie encore les lignes. Ils peuvent être numérotés avec la clause `FOR ORDINALITY`.

```
INSERT INTO t1 (utilisateur)
SELECT '{"name":"Roach Crosby",
        "friends":[{"id":0,"name":"Deloris Wilkerson"},
                   {"id":1,"name":"Pitts Lambert"},
                   {"id":2,"name":"Alicia Myers"}]}'::jsonb ;
```

L'expression d'évaluation est légèrement plus complexe :

```
SELECT t1.uid, resultats.*
FROM t1,
      JSON_TABLE(t1.utilisateur,
            '$' COLUMNS (
            nom text PATH '$.name',
            NESTED PATH '$.friends[*]'
              COLUMNS (
                  numero FOR ORDINALITY,
                  ami text PATH '$.name' ))
                ) AS resultats
WHERE t1.uid >= 1000 ;
```

```
 uid  |       nom       | numero |        ami
------+-----------------+--------+-------------------
 1000 | Claudine Howell |        |
 1001 | Roach Crosby    |      1 | Deloris Wilkerson
 1001 | Roach Crosby    |      2 | Pitts Lambert
 1001 | Roach Crosby    |      3 | Alicia Myers
(4 lignes)
```

Il existe une clause `PASSING` pour transmettre des critères de recherche proprement. Par exemple, pour connaître les amis de Roach Crosby (paramètre `filter1`), qui ne commencent *pas* par « Pitts » (paramètre `filter2`), et pour calculer un critère booléen sur un critère (`filter2`) :

```
SELECT t1.uid, resultats.*
FROM t1,
     JSON_TABLE(utilisateur,
            '$ ? (@.name == $filter1)'
            PASSING 'Roach Crosby' AS filter1,
                    'Pitts' AS filter2,
                    0 AS filter3
            COLUMNS (
                nom text PATH '$.name',
                NESTED PATH '$.friends[*] ? (!(@.name starts with $filter2))'
                  COLUMNS (
                      numero FOR ORDINALITY,
                      ami text PATH '$.name',
                      meilleurami bool EXISTS PATH '$.id ? (@ == $filter3)'))
                     ) AS resultats
;
```

```
 uid  |     nom      | numero |        ami        | meilleurami
------+--------------+--------+-------------------+-------------
 1001 | Roach Crosby |      1 | Deloris Wilkerson | t
 1001 | Roach Crosby |      2 | Alicia Myers      | f
```

Le plan d'exécution d'une telle requête (ici avec `VERBOSE`) indique un appel de fonction pour chaque ligne (`loops=1001`), sans détail sur l'exécution de cette fonction :

```
                            QUERY PLAN
------------------------------------------------------------------------
 Nested Loop (actual time=1.795..1.799 rows=2 loops=1)
   Output: t1.uid, resultats.nom, resultats.numero, resultats.ami,
↪  resultats.meilleurami
   Buffers: shared hit=48
   ->  Seq Scan on public.t1 (actual time=0.013..0.165 rows=1001 loops=1)
         Output: t1.uid, t1.utilisateur
         Buffers: shared hit=48
   ->  Table Function Scan on "json_table" resultats (actual time=0.001..0.001 rows=0
↪  loops=1001)
         Output: resultats.nom, resultats.numero, resultats.ami,
↪  resultats.meilleurami
         Table Function Call: JSON_TABLE(t1.utilisateur, '$?(@."name" == $"filter1")'
↪  AS json_table_path_0 PASSING 'Roach Crosby'::text AS filter1, 'Pitts'::text AS
↪  filter2, 0 AS filter3 COLUMNS (nom text PATH '$."name"', NESTED PATH
↪  '$."friends"[*]?(!(@."name" starts with $"filter2"))' AS json_table_path_1
↪  COLUMNS (numero FOR ORDINALITY, ami text PATH '$."name"', meilleurami boolean
↪  EXISTS PATH '$."id"?(@ == $"filter3")')))
 Planning Time: 0.114 ms
 Execution Time: 1.835 ms
```

Documentation :

— Documentation officielle : JSON_TABLE[26]

[26] https://www.postgresql.org/docs/current/functions-json.html#FUNCTIONS-SQLJSON-TABLE

1.5.12 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 attribut = lire tout le JSON
  - voire accès table TOAST
- Mise à jour : tout ou rien
- Indexation délicate

Les attributs 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. Rien ne vous interdit d'utiliser un attribut `country` au lieu de `pays`, avec une valeur `FR` au lieu de `France`. 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 attribut 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[27].

Il n'y a pas de mise à jour partielle : modifier un attribut implique de décomposer tout le JSON pour le réécrire entièrement (et parfois en le *détoastant*/*retoastant*). Si le JSON est trop gros, modifier ses sous-attributs par plusieurs processus différents peut poser des problèmes de verrouillage. Pour citer la documentation[28] :

« Les données JSON sont sujettes aux mêmes considérations de contrôle de concurrence que pour n'importe quel autre type de données quand elles sont stockées en table. Même si stocker de gros documents est prévisible, il faut garder à l'esprit que chaque mise à jour acquiert un verrou de niveau ligne sur toute la ligne. Il faut envisager de limiter les documents JSON à une taille gérable pour réduire les contentions sur verrou lors des transactions en mise à jour. Idéalement, les documents JSON devraient chacun représenter une donnée atomique, que les règles métiers imposent de ne pas pouvoir subdiviser en données plus petites qui pourraient être modifiées séparément. »

[27] https://dali.bo/m4_html#mécanisme-toast
[28] https://docs.postgresql.fr/current/datatype-json.html#JSON-INDEXING

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. Nous allons voir des contournements possibles.

Suivant le modèle, il peut y avoir une perte de place, puisque les clés sont répétées entre chaque attribut JSON, et non normalisées dans des tables séparées.

Enfin, nous allons voir que l'indexation est possible, mais moins triviale qu'à l'habitude.

Ces inconvénients sont à mettre en balance avec les intérêts du JSON (surtout : éviter des lignes avec trop d'attributs toujours à NULL, si même on les connaît), les fréquences de lecture et mises à jour des JSON, et les modalités d'utilisation des attributs.

Certaines de ces limites peuvent être réduites par les techniques ci-dessous.

1.5.13 Recherche dans un champ JSON (1)

Sans JSONPath :

```
SELECT * FROM personnes …
 WHERE datas->>'lastName' = 'Durand';  -- textes
 WHERE datas->'lastName'  = '"Durand"'::jsonb ; -- JSON
 WHERE datas @> '{"nom": "Durand"}'::jsonb ; -- contient
 WHERE datas ? 'spouse' ; -- attibut existe ?
 WHERE datas ?| '{spouse,children}'::text[] ; -- un des champs ?
 WHERE datas ?& '{spouse,children}'::text[] ; -- tous les champs ?
```

Mais comment indexer ?

Pour chercher les lignes avec un champ JSON possédant un attribut d'une valeur donnée, il existe plusieurs opérateurs (au sens syntaxique). Les comparaisons directes de textes ou de JSON sont possibles, mais nous verrons qu'elles ne sont pas simplement indexables.

L'opérateur `@>` (« contient ») est généralement plus adapté, mais il faut fournir un JSON avec le critère de recherche.

L'opérateur `?` permet de tester l'existence d'un attribut dans le JSON (même vide). Plusieurs attributs peuvent être testés avec `?|` (« ou » logique) ou `?&` (« et » logique).

1.5.14 Recherche dans un champ JSON (2) : SQL/JSON et JSONPath

- SQL :2016 introduit SQL/JSON et le langage JSONPath
- JSONPath :
  - langage de recherche pour JSON
  - concis, flexible, plus rapide
  - depuis PostgreSQL 12, étendu à chaque version

JSONPath[29] est un langage de requêtage permettant de spécifier des parties d'un champ JSON, même complexe. Il a été implémenté dans de nombreux langages, et a donc une syntaxe différente de celle du SQL, mais souvent déjà familière aux développeurs. Il évite de parcourir manuellement les nœuds et tableaux du JSON, ce qui est vite fastidieux en SQL.

Le standard SQL :2016 intègre le SQL/JSON[30]. PostgreSQL 12 contient déjà l'essentiel des fonctionnalités SQL/JSON, y compris JSONPath, mais elles sont complétées dans les versions suivantes.

1.5.15 Recherche dans un champ JSON (3) : Exemples SQL/JSON & JSONPath

```
SELECT … FROM …

 -- recherche
 WHERE datas @? '$.lastName ? (@ == "Valjean")' ;
 WHERE datas @@ '$.lastName  == "Valjean"' ;

SELECT jsonb_path_query(datas,'$.phoneNumbers[*] ? (@.type == "bureau")')
FROM   personnes
WHERE  datas @@ '$.lastName == "Durand"' ;

-- Affichage + filtrage
SELECT datas->>'lastName',
       jsonb_path_query(datas,'$.address ? (@.city == "Paris")')
FROM   personnes ;
```

Par exemple, une recherche peut se faire ainsi, et elle profitera d'un index GIN :

```
SELECT datas->>'firstName' AS prenom
FROM personnes
WHERE  datas @@ '$.lastName == "Durand"' ;

 prenom
----------
 Georges
```

[29] https://en.wikipedia.org/wiki/JSONPath
[30] https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016#json-path

Les opérateurs `@@` et `@?` sont liés à la recherche et au filtrage. La différence entre les deux est liée à la syntaxe à utiliser. Ces deux exemples renvoient la même ligne :

```
SELECT * FROM personnes
WHERE  datas @? '$.lastName ? (@ == "Valjean")' ;

SELECT * FROM personnes
WHERE  datas @@ '$.lastName  == "Valjean"' ;
```

Il existe des fonctions équivalentes, `jsonb_path_exists` et `jsonb_path_match` :

```
SELECT  datas->>'lastName' AS nom,
  jsonb_path_exists (datas, '$.lastName ? (@ == "Valjean")'),
  jsonb_path_match  (datas, '$.lastName == "Valjean"')
FROM personnes ;
```

```
   nom    | jsonb_path_exists | jsonb_path_match
---------+-------------------+------------------
 Valjean | t                 | t
 Durand  | f                 | f
 Dupont  | f                 | f
```

(Pour les détails sur ces opérateurs et fonctions, et des exemples sur des filtres plus complexes (inégalités par exemple), voir par exemple : https://justatheory.com/2023/10/sql-jsonpath-operators/.)

Un autre intérêt est la fonction `jsonb_path_query`, qui permet 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"}
```

Ici, `jsonb_path_query` génère une ligne par élément du tableau `phoneNumbers` inclus dans le JSON.

L'appel suivant effectue un filtrage sur la ville :

```
SELECT jsonb_path_query (datas, '$.address ? (@.city == "Paris")')
FROM   personnes ;
```

Cependant, pour que l'indexation GIN fonctionne, il faudra l'opérateur `@?` :

```
SELECT datas->>'lastName',
FROM   personnes
WHERE  personne @? '$.address ? (@.city == "Paris")' ;
```

Au final, le code JSONPath est souvent plus lisible que celui utilisant de nombreuses fonctions `jsonb` spécifiques à PostgreSQL. Un développeur le manipule déjà souvent dans un autre langage.

On trouvera d'autres exemples dans la présentation de Postgres Pro dédié à la fonctionnalité lors la parution de PostgreSQL 12[31], ou dans un billet de Michael Paquier[32].

[31] https://www.postgresql.eu/events/pgconfeu2019/sessions/session/2555/slides/221/jsonpath-pgconfeu-2019.pdf
[32] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

1.5.16 jsonb : indexation (1/2)

- Index fonctionnel sur un attribut précis
  - bonus : statistiques

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
ANALYZE personnes ;
```

- Colonne générée (dénormalisée) :
  - champ normal, indexable, facile à utiliser, rapide à lire
  - statistiques

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

- Préférer `@>` et `@?` que des fonctions

Index fonctionnel :

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 attribut 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.

Colonne générée :

Une autre possibilité est de dénormaliser l'attribut JSON intéressant dans un champ séparé de la table, et indexable :

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

ANALYZE personnes ;

CREATE INDEX ON personnes (lastname) ;
```

Cette colonne générée est mise à jour quand le JSON est modifié, et n'est pas modifiable autrement. C'est à part cela un champ simple, indexable avec un B-tree, et avec ses statistiques propres.

Ce champ coûte certes un peu d'espace disque supplémentaire, mais il améliore la lisibilité du code, et facilite l'usage avec certains outils ou pour certains utilisateurs. Dans le cas des gros JSON, il peut

aussi éviter quelques allers-retours vers la table TOAST. Même sans utilisation d'un index, un champ normal est beaucoup plus rapide à lire dans la ligne qu'un attribut extrait d'un JSON.

1.5.17 jsonb : indexation (2/2)

— Indexation « schemaless » grâce au GIN :
  — attention au choix de l'opérateur

```
-- opérateur par défaut
--  pour critères : ?, ?|, ?& , @> , @?, @@)
CREATE INDEX idx_prs ON personnes USING gin(datas) ;
-- index plus performant
--  mais que @> , @?, @@
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.

L'opérateur par défaut de GIN pour `jsonb` est `jsonb_ops`. Mais il est souvent plus efficace de choisir l'opérateur `jsonb_path_ops`. Ce dernier donne des index plus petits et performants sur des clés fréquentes, bien qu'il ne supporte que certains opérateurs de recherche (`@>` , `@?` et `@@`) (voir les détails[33]), ce qui suffit généralement.

```
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
```

Un index GIN 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 attribut du JSON peut être un critère.

Un index GIN ne permet cependant pas d'*Index Only Scan*.

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

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 ou colonnes générées vus plus haut. Attention, des fonctions comme `jsonb_path_query` ne savent pas utiliser les index. Il est donc préférable d'utiliser les opérateurs spécifiques, comme « contient » (`@>`) ou « existe » en JSONPath (`@?`).

1.5.18 Pour aller plus loin…

Lire la documentation
- 8.14 Types JSON :
  - https://docs.postgresql.fr/current/datatype-json.html
- 9.16 Fonctions & opérateurs JSON :
  - https://docs.postgresql.fr/current/functions-json.html

Les fonctions et opérateurs indiqués ici ne représentent qu'une partie de ce qui existe. Certaines fonctions sont très spécialisées, ou existent en plusieurs variantes voisines. Il est conseillé de lire ces deux chapitres de documentation lors de tout travail avec les JSON. Attention à la version de la page : des fonctionnalités sont ajoutées à chaque version de PostgreSQL.

1.6 XML

1.6.1 XML : présentation

- 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[34]. 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[35] ou celle des fonctions[36]. 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>
    <ARTIST>Jacques Brel</ARTIST>
    <COUNTRY>France</COUNTRY>
```

[34] https://docs.postgresql.fr/current/xml-limits-conformance.html
[35] https://docs.postgresql.fr/current/datatype-xml.html
[36] https://docs.postgresql.fr/current/functions-xml.html

```
    <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[37] est dépréciée et ne doit pas être utilisée dans les nouveaux projets.

[37] https://docs.postgresql.fr/current/xml2.html

1.7 OBJETS BINAIRES

1.7.1 Objets binaires : les types

- 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.

1.7.2 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
  - compressé/déporté (TOAST)
- 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).

Les `bytea` profitent du mécanisme TOAST[38] et sont donc généralement compressés et/ou dépla-

[38] https://dali.bo/m4_html#m%C3%A9canisme-toast

cés dans une table associé à la table principale. Encore plus que d'habitude, il faut proscrire les `SELECT *` dans votre code pour éviter de lire cette table quand ce n'est pas nécessaire.

La documentation officielle[39] liste les fonctions pour encoder, décoder, extraire, hacher… les `bytea`.

1.7.3 Large Object

- À éviter…
  - préférer `bytea`
- Maxi 4 To (éviter…)
- Objet indépendant des tables
  - OID à stocker dans les tables
  - se compresse mal
- Suppression manuelle !
  - trigger
  - `lo_unlink` & `vacuumlo`
- Fonction de manipulation, modification
  - `lo_create`, `lo_import`
  - `lo_seek`, `lo_open`, `lo_read`, `lo_write`…

Un *large object* est un objet totalement décorrélé des tables. 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[40]) ;
- de ne modifier que cette partie sans tout réécrire.

Cependant, nous déconseillons son utilisation autant que possible :

- le stockage des *large objects* ne prévoit pas vraiment de compression : des `bytea` prendront moins de place ;

[39] https://docs.postgresql.fr/current/functions-binarystring.html
[40] https://jdbc.postgresql.org/documentation/binary-data/

— il est facile et fréquent d'oublier de purger des *large objects* supprimés ;
— une sauvegarde logique partielle les oublie facilement (voir l'option `--large-objects` de `pg_dump`[41]) ;
— `pg_dump` n'est pas optimisé pour sauver de nombreux *large objects* : la sauvegarde de la table `pg_largeobject` ne peut être parallélisée et peut consommer transitoirement énormément de mémoire s'il y a trop d'objets.

Il y a plusieurs méthodes pour nettoyer les *large objects* 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[42], si les *large objects* ne sont jamais référencés plus d'une fois ;
— appeler régulièrement le programme `vacuumlo` (là encore un contrib[43]) : il liste tous les *large objects* référencés dans la base, puis supprime les autres. Ce traitement est bien sûr un peu lourd.

Techniquement, un *large object* est stocké dans la table système `pg_largeobject` sous forme de pages de 2 ko. Voir la documentation[44] pour les détails.

[41] https://docs.postgresql.fr/current/app-pgdump.html
[42] https://docs.postgresql.fr/current/lo.html
[43] https://docs.postgresql.fr/current/vacuumlo.html
[44] https://docs.postgresql.fr/current/largeobjects.html

1.8 QUIZ

https://dali.bo/s22_quiz

1.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s22_solutions.

1.9.1 UUID

But : Constater l'impact en cache des types d'UUID avec pg_buffercache.

Ce TP étant purement descriptif, allez voir directement la solution.

1.9.2 jsonb

But : Découvrir JSON

La base **personnes_et_dossiers** pèse en version complète 613 Mo, pour 2 Go sur disque au final. Elle peut être installée comme suit :

```
# Dump complet
curl -kL https://dali.bo/tp_personnes -o /tmp/personnes.dump
# Taille 40%
# curl -kL https://dali.bo/tp_personnes_200k -o /tmp/personnes.dump
# Taille 16%
# curl -kL https://dali.bo/tp_personnes_fr -o /tmp/personnes.dump


createdb --echo personnes
# L'erreur sur un schéma 'public' existant est normale
pg_restore -v -d personnes /tmp/personnes.dump
rm -- /tmp/personnes.dump
```

La base `personnes` contient alors deux schémas `json` et `eav` avec les mêmes données sous deux formes différentes.

Chercher la ville et le numéro de téléphones (sous-attribut `ville` de l'attribut `adresse` du champ JSON `personne`) de Gaston Lagaffe, grâce aux attributs `prenom` et `nom`. Effectuer de préférence la recherche en cherchant un JSON avec `@>` (« contient ») (Ne pas chercher encore à utiliser JSONPath).

Créer une requête qui renvoie les attributs `nom`, `prenom`, `date_naissance` (comme type date) de toutes les personnes avec le nom « Lagaffe ». Utiliser la fonction `jsonb_to_record()` et `LATERAL`. Rajouter `ville` et `pays` ensuite de la même manière.

En supprimant le filtre, comparer le temps d'exécution de la requête précédente avec cette requête plus simple qui récupère les champs plus manuellement :

```
SELECT personne->>'nom',
    personne->>'prenom',
    (personne->>'date_naissance')::date,
    personne#>>'{adresse,ville}',
    personne#>>'{adresse,pays}'
FROM json.personnes;
```

Créer un index GIN ainsi :

```
CREATE INDEX personnes_gin ON json.personnes
USING gin(personne jsonb_path_ops);
```

Quelle taille fait-il ?

Retenter les requêtes précédentes. Lesquelles utilisent l'index ?

Récupérer les numéros de téléphone de Léon Prunelle.

Afficher les noms et prénoms de Prunelles, et un tableau de champs texte contenant ses téléphones (utiliser `jsonb_array_elements_text`).

Comparer le résultat et les performances de ces deux requêtes, qui récupèrent aussi les numéros de téléphone de Prunelle :

Chercher qui possède le numéro de téléphone `0650041821` avec la syntaxe JSONPath.

Compter le nombre de personnes habitant à Paris ou Bruxelles avec :

— la syntaxe `@>` et un `OR` ;
— une syntaxe JSONPath `@?` et un « ou logique (`||`) ;
— une syntaxe JSONPath `@?` et une regex `@.ville like_regex "^(Paris|Bruxelles)$"`.

Le compte du nombre de personne par pays doit être optimisé au maximum. Ajouter un index fonctionnel sur l'attribut `pays`. Tester l'efficacité sur une recherche, et un décompte de toute les personnes par pays.

Ajouter un champ généré dans `json.personne`, correspondant à l'attribut `pays`.

Comparer les temps d'exécution du décompte des pays par l'attribut, et par cette colonne générée.

Créer un index B-tree sur la colonne générée `pays`. Consulter les statistiques dans `pg_stats`. Cet index est-il utilisable pour des filtres et le décompte par `pays` ?

(Optionnel) Créer des colonnes générées sur `nom`, `prenom`, `date_naissance`, et `ville` (en un seul ordre). Reprendre la requête plus haut qui les affiche tous et comparer les performances.

Ajouter l'attribut `animaux` à Gaston Lagaffe, avec la valeur 18. Vérifier en relisant la ligne.

Ajouter l'attribut `animaux` à 2% des individus au hasard, avec une valeur 1 ou 2.

Compter le nombre de personnes avec des animaux (avec ou sans JSONPath). Proposer un index qui pourait convenir à d'autres futurs nouveaux attributs peu fréquents.

1.9.3 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.

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

1.10.1 UUID

Tout ce qui suit doit se dérouler dans la même base, par exemple :

```
CREATE DATABASE capteurs ;
```

Ce TP est prévu pour un `shared_buffers` de 128 Mo (celui par défaut). Si le vôtre est plus gros, le TP devra peut-être durer plus longtemps :

```
SHOW shared_buffers ;
```

Utilisez au moins une fenêtre pour les ordres shell et une pour les ordres SQL.

> Créer avec le script suivants les deux versions d'un petit modèle avec des capteurs, et les données horodatées qu'ils renvoient; ainsi que les deux procédures pour remplir ces tables ligne à ligne :

```
\c capteurs

-- Modèle : une table 'capteurs' et ses 'donnees' horodatées
-- liées par une contrainte
-- Deux versions : avec ID et une séquence, et avec UUID

DROP TABLE IF EXISTS donnees1, donnees2, capteurs1, capteurs2 ;

-- Avec identifiants bigint
CREATE TABLE capteurs1 (id_capteur  bigint  PRIMARY KEY,
                        nom         char (50) UNIQUE,
                        filler      char (50) default ''
                       ) ;
CREATE TABLE donnees1 (id_donnee   bigserial PRIMARY KEY,
                       id_capteur  int NOT NULL REFERENCES capteurs1,
                       horodatage  timestamp with time zone,
                       valeur1     int,
                       valeur2     int,
                       valeur3     float
                      ) ;
CREATE INDEX ON donnees1 (horodatage) ;

-- Version avec les UUID
CREATE TABLE capteurs2 (id_capteur  uuid PRIMARY KEY,
                        nom         char (50) UNIQUE,
                        filler      char (50) default ''
                       ) ;
CREATE TABLE donnees2 (id_donnee   uuid PRIMARY KEY,
                       id_capteur  uuid NOT NULL REFERENCES capteurs2,
                       horodatage  timestamp with time zone,
                       valeur1     int,
                       valeur2     int,
                       valeur3     float
                      ) ;
CREATE INDEX ON donnees2 (horodatage) ;

-- 1000 capteurs identiques
```

```
INSERT INTO capteurs1 (id_capteur, nom)
SELECT  i,
        'M-'||md5(i::text)
FROM generate_series (1,1000) i
ORDER BY random() ;

INSERT INTO capteurs2 (id_capteur, nom)
SELECT gen_random_uuid(), nom FROM capteurs1 ;

-- 2 procédures d'insertion de données identiques sur quelques capteurs au hasard
-- insertion dans donnees1 avec une séquence
CREATE OR REPLACE PROCEDURE insere_donnees_1 ()
AS $$
    SET synchronous_commit TO off ; -- accélère
    INSERT INTO donnees1 (id_donnee, id_capteur, horodatage, valeur1, valeur2,
 ↪  valeur3)
    SELECT  nextval('donnees1_id_donnee_seq'::regclass), -- clé primaire des données
            m.id_capteur,                                -- clé étrangère
            now(), (random()*1000)::int,(random()*1000)::int,random()
    FROM capteurs1 m TABLESAMPLE BERNOULLI (1) ; -- 1% des lignes
$$ LANGUAGE sql;
-- insertion dans donnees2 avec un UUID v7
CREATE OR REPLACE PROCEDURE insere_donnees_2 ()
AS $$
    SET synchronous_commit TO off ; -- accélère
    INSERT INTO donnees2 (id_donnee, id_capteur, horodatage, valeur1, valeur2,
 ↪  valeur3)
    SELECT  gen_random_uuid(),   -- clé primaire des données, UUID v4
            m.id_capteur,        -- clé étrangère
            now(), (random()*1000)::int,(random()*1000)::int,random()
    FROM capteurs2 m TABLESAMPLE BERNOULLI (1) ; -- 1% des lignes
$$ LANGUAGE sql;
```

Vous devez obtenir ces tables et une séquence :

```
capteurs=# \d+
                                                  Liste des relations
 Schéma |          Nom           |   Type   | … | … | … |   Taille   | Description
--------+------------------------+----------+---+---+---+------------+-------------
 public | capteurs1              | table    | … | … | … | 168 kB     |
 public | capteurs2              | table    | … | … | … | 176 kB     |
 public | donnees1               | table    | … | … | … | 0 bytes    |
 public | donnees1_id_donnee_seq | séquence | … | … |   | 8192 bytes |
 public | donnees2               | table    | … | … | … | 0 bytes    |
(5 lignes)
```

et ces index :

```
capteurs=# \di
                        Liste des relations
 Schéma |           Nom           | Type  | Propriétaire |   Table
--------+-------------------------+-------+--------------+-----------
 public | capteurs1_nom_key       | index | postgres     | capteurs1
 public | capteurs1_pkey          | index | postgres     | capteurs1
 public | capteurs2_nom_key       | index | postgres     | capteurs2
 public | capteurs2_pkey          | index | postgres     | capteurs2
 public | donnees1_horodatage_idx | index | postgres     | donnees1
```

```
 public | donnees1_pkey           | index | postgres     | donnees1
 public | donnees2_horodatage_idx | index | postgres     | donnees2
 public | donnees2_pkey           | index | postgres     | donnees2
```

> Créer deux fichiers SQL contenants juste les appels de fonctions, qui serviront pour pgbench :

```
echo "CALL insere_donnees_1 ()" > /tmp/insere1.sql
echo "CALL insere_donnees_2 ()" > /tmp/insere2.sql
```

> Dans la même base que les table ci-dessus, installer l'extension pg_buffercache[a] qui va nous permettre de voir ce qu'il y a dans le cache de PostgreSQL :
>
> [a] https://dali.bo/x2_html

```
CREATE EXTENSION IF NOT EXISTS pg_buffercache ;
```

La vue du même nom contient une ligne par bloc. La requête suivante permet de voir lesquelles de nos tables utilisent le cache :

```
SELECT CASE WHEN datname = current_database()
        AND relname NOT LIKE 'pg%'
        THEN relname ELSE '*AUTRES*' END AS objet,
       count(*),
       pg_size_pretty(count(bufferid)*8192) as Taille_Mo
FROM pg_buffercache b
LEFT OUTER JOIN pg_class c ON c.relfilenode = b.relfilenode
LEFT OUTER JOIN pg_database d ON (d.oid = b.reldatabase)
GROUP BY objet
ORDER BY count(bufferid) DESC ;
```

Cette version semi-graphique est peut-être plus parlante :

```
SELECT CASE WHEN datname = current_database()
        AND relname NOT LIKE 'pg%'
        THEN relname ELSE '*AUTRES*' END AS objet,
       pg_size_pretty(count(bufferid)*8192) as Taille_Mo,
       lpad('',(count(bufferid)/200)::int, '#') AS Taille
FROM pg_buffercache b
LEFT OUTER JOIN pg_class c ON c.relfilenode = b.relfilenode
LEFT OUTER JOIN pg_database d ON (d.oid = b.reldatabase)
GROUP BY objet
ORDER BY objet DESC ;
```

> Dans une fenêtre, lancer l'une de ces requêtes (dans la bonne base !), puis la répéter toutes les secondes ainsi :

```
-- sous psql
\watch 1
```

> Dans une autre fenêtre, lancer `pgbench` avec deux clients, et le script pour remplir la table `donnees1` :

```
# Sous Rocky Linux/Almalinux…
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql
```

Le nombre de transactions dépend fortement de la machine, mais peut atteindre plusieurs milliers à la seconde.

Les tables peuvent rapidement atteindre plusieurs gigaoctets. N'hésitez pas à les vider ensemble de temps à autre.

```
TRUNCATE donnees1, donnees2 ;
```

Quelle est la répartition des données dans le cache ?

Après peu de temps, la répartition doit ressembler à peu près à ceci :

```
          objet           | taille_mo  |             taille
--------------------------+------------+--------------------------------
 donnees1_pkey            | 28 MB      | #########
 donnees1_id_donnee_seq   | 8192 bytes |
 donnees1_horodatage_idx  | 12 MB      | ####
 donnees1                 | 86 MB      | ############################
 capteurs1_pkey           | 48 kB      |
 capteurs1                | 144 kB     |
 *AUTRES*                 | 2296 kB    | #
```

Et ce, même si la table et ses index ne tiennent plus intégralement dans le cache.

La table `donnees1` représente la majorité du cache.

Interrompre `pgbench` et le relancer pour remplir `donnees2` :

```
# Sous Rocky Linux/Almalinux…
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere2.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere2.sql
```

Noter que le débit en transaction est du même ordre de grandeur : les UUID ne sont pas spécialement lourds à générer.

Que devient la répartition des données dans le cache ?

`donnees1` et ses index est chassé du cache par les nouvelles données, ce qui est logique.

Surtout, on constate que la clé primaire de `donnnes2` finit par remplir presque tout le cache. Dans ce petit cache, il n'y a plus de place même pour les données de `donnees2` !

```
          objet          | taille_mo |                    taille
-------------------------+-----------+-----------------------------------------
 donnees2_pkey           | 120 MB    | ########################################
 donnees2_horodatage_idx | 728 kB    |
 donnees2                | 6464 kB   | ##
 capteurs2_pkey          | 48 kB     |
 capteurs2               | 152 kB    |
 *AUTRES*                | 408 kB    |
```

> Interrompre `pgbench`, purger les tables et lancer les deux scripts d'alimentation en même temps.

```
TRUNCATE donnees1, donnees2 ;

# Sous Rocky Linux/Almalinux…
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
```

On constate le même phénomène de monopolisation du cache par `donnees2`, bien que les deux tables de données aient le même nombre de lignes :

```
          objet          | taille_mo  |                   taille
-------------------------+------------+-------------------------------------
 donnees2_pkey           | 115 MB     | ####################################
 donnees2_horodatage_idx | 624 kB     |
 donnees2                | 5568 kB    | ##
 donnees1_pkey           | 1504 kB    |
 donnees1_id_donnee_seq  | 8192 bytes |
 donnees1_horodatage_idx | 632 kB     |
 donnees1                | 4544 kB    | #
 capteurs2_pkey          | 48 kB      |
 capteurs2               | 152 kB     |
 capteurs1_pkey          | 48 kB      |
 capteurs1               | 144 kB     |
 *AUTRES*                | 408 kB     |
(12 lignes)
```

> Avez-vous remarqué une différence de vitesse entre les deux traitements ?

Ce ne peut être rigoureusement établi ici. Les volumétries sont trop faibles par rapport à la taille des mémoires et il faut tester sur la durée. Le nombre de clients doit être étudié pour utiliser au mieux les capacités de la machine sans monter jusqu'à ce que la contention devienne un problème. Les checkpoints font également varier les débits.

Cependant, si vous laissez le test tourner très longtemps avec des tailles de tables de plusieurs Go, les effets de cache seront très différents :

— dans `donnees1`, le débit en insertion devrait rester correct, car seuls les derniers blocs en cache sont utiles ;

— dans `donnees2`, le débit en insertion doit progressivement baisser : chaque insertion a besoin d'un bloc de l'index de clé primaire différent, qui a de moins en moins de chance de se trouver dans le cache de PostgreSQL, puis dans le cache de Linux.

L'impact sur les I/O augmente donc, et pas seulement à cause de la volumétrie supérieure des tables avec UUID. À titre d'exemple, sur une petite base de formation avec 3 Go de RAM :

```
# avec ID numériques, pendant des insertions dans donnees1 uniquement,
# qui atteint 1,5 Go
# débit des requêtes : environ 3000 tps
$ iostat -h 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          88,6%    0,0%   10,7%    0,0%    0,0%    0,7%

      tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn Device
    86,00         0,0k        17,0M       0,0k      17,0M vda
     0,00         0,0k         0,0k       0,0k       0,0k scd0

# avec UUID v4, pendant des insertions dans donnees2 uniquement,
# qui atteint 1,5 Go
# débit des requêtes : environ 700 tps
$ iostat -h 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          41,2%    0,0%   17,3%   25,9%    0,7%   15,0%

      tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn Device
  2379,00         0,0k        63,0M       0,0k      63,0M vda
     0,00         0,0k         0,0k       0,0k       0,0k scd0
```

> Comparer les tailles des tables et index avant et après un `VACUUM FULL`. Où était la fragmentation ?

`VACUUM FULL` reconstruit complètement les tables et aussi les index.

Tables avant le `VACUUM FULL` :

```
capteurs=# \d+
                              Liste des relations
 Schéma |          Nom           |   Type   | … | … | … |   Taille   | …
--------+------------------------+----------+---+---+---+------------+--
 public | capteurs1              | table    | … | … | … | 168 kB     |
 public | capteurs2              | table    | … | … | … | 176 kB     |
 public | donnees1               | table    | … | … | … | 2180 MB    |
 public | donnees1_id_donnee_seq | séquence | … | … |   | 8192 bytes |
 public | donnees2               | table    | … | … | … | 2227 MB    |
 public | pg_buffercache         | vue      | … | … |   | 0 bytes    |
(6 lignes)
```

Après :

```
capteurs=# \d+
                              Liste des relations
 Schéma |          Nom           |   Type   | … | … | … |   Taille   | …
--------+------------------------+----------+---+---+---+------------+--
 public | capteurs1              | table    | … | … | … | 144 kB     |
```

```
 public | capteurs2               | table    | … | … | … | 152 kB     |
 public | donnees1                | table    | … | … | … | 2180 MB    |
 public | donnees1_id_donnee_seq  | séquence | … | … |   | 8192 bytes |
 public | donnees2                | table    | … | … | … | 2227 MB    |
 public | pg_buffercache          | vue      | … | … |   | 0 bytes    |
(6 lignes)
```

Les tailles des tables `donnees1` et `donnees2` ne bougent pas. C'est normal, il n'y a eu que des insertions à chaque fois en fin de table, et ni modification ni suppression de données.

Index avant le `VACUUM FULL` :

```
capteurs=# \di+
                            Liste des relations
 Schéma |            Nom            | Type  | … |   Table    | … | Méth. | Taille  | …
--------+---------------------------+-------+---+------------+---+-------+---------+--
 public | capteurs1_nom_key         | index | … | capteurs1  | … | btree | 120 kB  |
 public | capteurs1_pkey            | index | … | capteurs1  | … | btree | 56 kB   |
 public | capteurs2_nom_key         | index | … | capteurs2  | … | btree | 120 kB  |
 public | capteurs2_pkey            | index | … | capteurs2  | … | btree | 56 kB   |
 public | donnees1_horodatage_idx   | index | … | donnees1   | … | btree | 298 MB  |
 public | donnees1_pkey             | index | … | donnees1   | … | btree | 717 MB  |
 public | donnees2_horodatage_idx   | index | … | donnees2   | … | btree | 245 MB  |
 public | donnees2_pkey             | index | … | donnees2   | … | btree | 1166 MB |
(8 lignes)
```

Index après :

```
capteurs=# \di+
                            Liste des relations
 Schéma |            Nom            | Type  | … |   Table    | … | Méth. | Taille | …
--------+---------------------------+-------+---+------------+---+-------+--------+--
 public | capteurs1_nom_key         | index | … | capteurs1  | … | btree | 96 kB  |
 public | capteurs1_pkey            | index | … | capteurs1  | … | btree | 40 kB  |
 public | capteurs2_nom_key         | index | … | capteurs2  | … | btree | 96 kB  |
 public | capteurs2_pkey            | index | … | capteurs2  | … | btree | 48 kB  |
 public | donnees1_horodatage_idx   | index | … | donnees1   | … | btree | 296 MB |
 public | donnees1_pkey             | index | … | donnees1   | … | btree | 717 MB |
 public | donnees2_horodatage_idx   | index | … | donnees2   | … | btree | 245 MB |
 public | donnees2_pkey             | index | … | donnees2   | … | btree | 832 MB |
(8 lignes)
```

Les index d'horodatage gardent la même taille qu'avant (la différence entre eux est dû à des nombres de lignes différents dans cet exemple). L'index sur la clé primaire de `donnees1` (`bigint`) n'était pas fragmenté. Par contre, `donnees2_pkey` se réduit de 29% ! Les index UUID (v4) ont effectivement tendance à se fragmenter.

> Les UUID générés avec `gen_random_uuid` sont de version 4. Créer la fonction suivante pour générer des UUID version 7, l'utiliser dans la fonction d'alimentation de `donnees2`, et relancer les deux alimentations :

```
-- Source : https://postgresql.verite.pro/blog/2024/07/15/uuid-v7-pure-sql.html
-- Daniel Vérité d'après Kyle Hubert
CREATE OR REPLACE FUNCTION uuidv7() RETURNS uuid
```

```
AS $$
  -- Replace the first 48 bits of a uuidv4 with the current
  -- number of milliseconds since 1970-01-01 UTC
  -- and set the "ver" field to 7 by setting additional bits
  select encode(
    set_bit(
      set_bit(
        overlay(uuid_send(gen_random_uuid()) placing
        substring(int8send((extract(epoch from clock_timestamp())*1000)::bigint)
            from 3)
        from 1 for 6),
        52, 1),
      53, 1), 'hex')::uuid;
$$ LANGUAGE sql volatile ;

-- insertion dans donnees2 avec un UUID v7
CREATE OR REPLACE PROCEDURE insere_donnees_2 ()
AS $$
    SET synchronous_commit TO off ; -- accélère
    INSERT INTO donnees2 (id_donnee, id_capteur, horodatage, valeur1, valeur2,
 ↪  valeur3)
    SELECT  uuidv7(),   -- clé primaire des données, UUID v7
            m.id_capteur,        -- clé étrangère
            now(), (random()*1000)::int,(random()*1000)::int,random()
    FROM capteurs2 m TABLESAMPLE BERNOULLI (1) ; -- 1% des capteurs
$$ LANGUAGE sql;

# Sous Rocky Linux/Almalinux…
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
```

Après quelques dizaines de secondes :

- les deux tables doivent être présentes dans le cache de manière similaire ;
- les index primaires doivent être présents de manière anecdotique ;
- `donnees2` occupe une taille un peu supérieure à cause de la taille double des UUID par rapport aux `bigint` de `donnees1` :

```
          objet          | taille_mo  |                 taille
-------------------------+------------+----------------------------------------
 donnees2_pkey           | 18 MB      | ######
 donnees2_horodatage_idx | 5392 kB    | ##
 donnees2                | 48 MB      | ###############
 donnees1_pkey           | 13 MB      | ####
 donnees1_id_donnee_seq  | 8192 bytes |
 donnees1_horodatage_idx | 5376 kB    | ##
 donnees1                | 38 MB      | ############
 capteurs2_pkey          | 48 kB      |
 capteurs2               | 152 kB     |
 capteurs1_pkey          | 48 kB      |
 capteurs1               | 144 kB     |
 *AUTRES*                | 648 kB     |
(12 lignes)
```

> Relancez `pgbench` pour charger `donnees2`, alternez entre les deux versions de la fonction `insere_donnees_2`.

```
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 -f /tmp/insere2.sql

…    [fonction avec gen_random_uuid (UUID v4) ]
progress: 202.0 s, 781.2 tps, lat 2.546 ms stddev 6.631, 0 failed
progress: 203.0 s, 597.6 tps, lat 3.229 ms stddev 10.497, 0 failed
progress: 204.0 s, 521.7 tps, lat 3.995 ms stddev 20.001, 0 failed
progress: 205.0 s, 837.0 tps, lat 2.307 ms stddev 7.743, 0 failed
progress: 206.0 s, 1112.1 tps, lat 1.856 ms stddev 7.602, 0 failed
progress: 207.0 s, 1722.8 tps, lat 1.097 ms stddev 0.469, 0 failed
progress: 208.0 s, 894.4 tps, lat 2.352 ms stddev 12.725, 0 failed
progress: 209.0 s, 1045.6 tps, lat 1.911 ms stddev 5.631, 0 failed
progress: 210.0 s, 1040.0 tps, lat 1.921 ms stddev 8.009, 0 failed
progress: 211.0 s, 734.6 tps, lat 2.259 ms stddev 9.833, 0 failed
progress: 212.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 213.0 s, 266.3 tps, lat 16.299 ms stddev 165.541, 0 failed
progress: 214.0 s, 1548.9 tps, lat 1.290 ms stddev 1.970, 0 failed
progress: 215.0 s, 896.0 tps, lat 2.163 ms stddev 5.404, 0 failed
progress: 216.0 s, 1113.0 tps, lat 1.798 ms stddev 4.115, 0 failed
progress: 217.0 s, 886.9 tps, lat 1.990 ms stddev 4.609, 0 failed
progress: 218.0 s, 771.1 tps, lat 2.965 ms stddev 9.767, 0 failed
…    [modification avec uuidv7 (UUID v7) ]
progress: 219.0 s, 1952.1 tps, lat 1.022 ms stddev 2.513, 0 failed
progress: 220.0 s, 2241.1 tps, lat 0.890 ms stddev 0.431, 0 failed
progress: 221.0 s, 2184.0 tps, lat 0.914 ms stddev 0.853, 0 failed
progress: 222.0 s, 2191.1 tps, lat 0.911 ms stddev 0.373, 0 failed
progress: 223.0 s, 2355.8 tps, lat 0.847 ms stddev 0.332, 0 failed
progress: 224.0 s, 2267.0 tps, lat 0.880 ms stddev 0.857, 0 failed
progress: 225.0 s, 2308.0 tps, lat 0.864 ms stddev 0.396, 0 failed
progress: 226.0 s, 2230.9 tps, lat 0.894 ms stddev 0.441, 0 failed
progress: 227.0 s, 2225.1 tps, lat 0.897 ms stddev 1.284, 0 failed
progress: 228.0 s, 2250.2 tps, lat 0.886 ms stddev 0.408, 0 failed
progress: 229.0 s, 2325.1 tps, lat 0.858 ms stddev 0.327, 0 failed
progress: 230.0 s, 2172.1 tps, lat 0.919 ms stddev 0.442, 0 failed
progress: 231.0 s, 2209.8 tps, lat 0.903 ms stddev 0.373, 0 failed
progress: 232.0 s, 2379.0 tps, lat 0.839 ms stddev 0.342, 0 failed
progress: 233.0 s, 2349.1 tps, lat 0.849 ms stddev 0.506, 0 failed
progress: 234.0 s, 2274.9 tps, lat 0.877 ms stddev 0.350, 0 failed
progress: 235.0 s, 2245.0 tps, lat 0.889 ms stddev 0.351, 0 failed
progress: 236.0 s, 2155.9 tps, lat 0.925 ms stddev 0.344, 0 failed
progress: 237.0 s, 2299.2 tps, lat 0.869 ms stddev 0.343, 0 failed
…    [nouvelle modification, retour à gen_random_uuid ]
progress: 238.0 s, 1296.9 tps, lat 1.540 ms stddev 2.092, 0 failed
progress: 239.0 s, 1370.1 tps, lat 1.457 ms stddev 2.794, 0 failed
progress: 240.0 s, 1089.9 tps, lat 1.832 ms stddev 4.234, 0 failed
progress: 241.0 s, 770.0 tps, lat 2.594 ms stddev 13.761, 0 failed
progress: 242.0 s, 412.0 tps, lat 4.736 ms stddev 28.332, 0 failed
progress: 243.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 244.0 s, 632.6 tps, lat 6.403 ms stddev 65.839, 0 failed
progress: 245.0 s, 1183.0 tps, lat 1.655 ms stddev 3.732, 0 failed
progress: 246.0 s, 869.0 tps, lat 2.287 ms stddev 5.968, 0 failed
progress: 247.0 s, 967.0 tps, lat 2.118 ms stddev 4.860, 0 failed
progress: 248.0 s, 954.5 tps, lat 2.088 ms stddev 3.967, 0 failed
progress: 249.0 s, 759.3 tps, lat 2.635 ms stddev 10.382, 0 failed
```

```
progress: 250.0 s, 787.0 tps, lat 2.395 ms stddev 9.791, 0 failed
progress: 251.0 s, 744.0 tps, lat 2.518 ms stddev 10.636, 0 failed
progress: 252.0 s, 815.1 tps, lat 2.744 ms stddev 11.983, 0 failed
progress: 253.0 s, 931.2 tps, lat 1.998 ms stddev 7.886, 0 failed
progress: 254.0 s, 665.0 tps, lat 2.946 ms stddev 13.315, 0 failed
progress: 255.0 s, 537.1 tps, lat 3.970 ms stddev 19.232, 0 failed
progress: 256.0 s, 683.9 tps, lat 2.757 ms stddev 10.356, 0 failed
```

Le débit en transactions varie ici d'un facteur 2. Noter que la durée des transactions est aussi beaucoup plus stable (`stddev`).

1.10.2 jsonb : lecture de champs

La base **personnes_et_dossiers** pèse en version complète 613 Mo, pour 2 Go sur disque au final. Elle peut être installée comme suit :

```
# Dump complet
curl -kL https://dali.bo/tp_personnes -o /tmp/personnes.dump
# Taille 40%
# curl -kL https://dali.bo/tp_personnes_200k -o /tmp/personnes.dump
# Taille 16%
# curl -kL https://dali.bo/tp_personnes_fr -o /tmp/personnes.dump


createdb --echo personnes
# L'erreur sur un schéma 'public' existant est normale
pg_restore -v -d personnes /tmp/personnes.dump
rm -- /tmp/personnes.dump
```

La base `personnes` contient alors deux schémas `json` et `eav` avec les mêmes données sous deux formes différentes.

La table `json.personnes` contient une ligne par personne, un identifiant et un champ JSON avec de nombreux attributs. Elle n'est pas encore indexée :

```
 \d json.personnes
                    Table « json.personnes »
   Colonne    |  Type   | Collationnement | NULL-able | Par défaut
--------------+---------+-----------------+-----------+------------
 id_personne | integer |                 |           |
 personne    | jsonb   |                 |           |
```

> Chercher la ville et le numéro de téléphone (sous-attribut `ville` de l'attribut `adresse` du champ JSON `personne`) de Gaston Lagaffe, grâce aux attributs `prenom` et `nom`. Effectuer de préférence la recherche en cherchant un JSON avec `@>` (« contient ») (Ne pas chercher encore à utiliser JSONPath).

La recherche peut s'effectuer en convertissant tous les attributs en texte :

```
SELECT personne->'adresse'->>'ville'
FROM   json.personnes p
WHERE  personne->>'nom' = 'Lagaffe'
  AND  personne->>'prenom' = 'Gaston' ;
```

On obtient « Bruxelles ».

Avec la syntaxe en version 14 :

```
SELECT personne['adresse']['ville']->>0 AS ville
FROM   json.personnes p
WHERE  personne['nom'] = '"Lagaffe"'::jsonb
  AND  personne['prenom'] = '"Gaston"'::jsonb ;
```

Il est plus propre de rechercher grâce à une de ces syntaxes, notamment parce qu'elles seront indexables plus tard :

```
SELECT personne->'adresse'->>'ville'
FROM json.personnes p
WHERE personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb ;
```

ou :

```
SELECT personne->'adresse'->>'ville'
FROM json.personnes p
WHERE personne @> jsonb_build_object ('nom', 'Lagaffe', 'prenom', 'Gaston') ;
```

> Créer une requête qui renvoie les attributs `nom`, `prenom`, `date_naissance` (comme type date) de toutes les personnes avec le nom « Lagaffe ». Utiliser la fonction `jsonb_to_record()` et `LATERAL`. Rajouter `ville` et `pays` ensuite de la même manière.

`jsonb_to_record` exige que l'on fournisse le nom de l'attribut et son type :

```
SELECT r.*
FROM json.personnes,
LATERAL jsonb_to_record (personne) AS r (nom text, prenom text, date_naissance date)
WHERE personne @> '{"nom": "Lagaffe"}'::jsonb;
```

```
   nom    | prenom | date_naissance
----------+--------+----------------
 Lagaffe | Gaston | 1938-09-22
 Lagaffe | Jeanne | 1940-02-14
```

Avec la ville, qui est dans un sous-attribut, il faut rajouter une clause `LATERAL` :

```
SELECT r1.*, r2.*
FROM json.personnes,
LATERAL jsonb_to_record (personne)
        AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
        AS r2 (ville text, pays text)
WHERE personne @> '{"nom": "Lagaffe"}'::jsonb;
```

```
   nom    | prenom | date_naissance |   ville   |   pays
----------+--------+----------------+-----------+----------
 Lagaffe | Gaston | 1938-09-22     | Bruxelles | Belgique
 Lagaffe | Jeanne | 1940-02-14     | Bruxelles | Belgique
```

> En supprimant le filtre, comparer le temps d'exécution de la requête précédente avec cette requête plus simple qui récupère les champs plus manuellement :
>
> ```
> SELECT personne->>'nom',
>     personne->>'prenom',
>     (personne->>'date_naissance')::date,
>     personne#>>'{adresse,ville}',
>     personne#>>'{adresse,pays}'
> FROM json.personnes;
> ```

Cette dernière requête est nettement plus lente que l'utilisation de `jsonb_to_record`, même si les I/O sont plus réduites :

```
EXPLAIN (COSTS OFF,ANALYZE,BUFFERS)
SELECT personne->>'nom',
     personne->>'prenom',
     (personne->>'date')::date,
     personne#>>'{adresse,ville}',
     personne#>>'{adresse,pays}'
FROM json.personnes;

                          QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on personnes  (cost=0.00..71383.74 rows=532645 width=132) (actual
↪  time=0.079..6009.601 rows=532645 loops=1)
   Buffers: shared hit=3825357 read=122949
 Planning Time: 0.078 ms
 Execution Time: 6022.738 ms

EXPLAIN (ANALYZE,BUFFERS)
SELECT r1.*, r2.* FROM json.personnes,
LATERAL jsonb_to_record (personne)
       AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
       AS r2 (ville text, pays text) ;

                          QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=0.01..83368.26 rows=532645 width=132) (actual
↪  time=0.064..3820.847 rows=532645 loops=1)
   Buffers: shared hit=1490408 read=122956
   ->  Nested Loop  (cost=0.00..72715.35 rows=532645 width=832) (actual
↪  time=0.059..2247.303 rows=532645 loops=1)
         Buffers: shared hit=712094 read=122956
         ->  Seq Scan on personnes  (cost=0.00..62062.45 rows=532645 width=764)
↪  (actual time=0.037..98.138 rows=532645 loops=1)
               Buffers: shared read=56736
         ->  Function Scan on jsonb_to_record r1  (cost=0.00..0.01 rows=1 width=68)
↪  (actual time=0.004..0.004 rows=1 loops=532645)
               Buffers: shared hit=712094 read=66220
   ->  Function Scan on jsonb_to_record r2  (cost=0.01..0.01 rows=1 width=64) (actual
↪  time=0.003..0.003 rows=1 loops=532645)
         Buffers: shared hit=778314
 Planning Time: 0.103 ms
 Execution Time: 3953.137 ms
```

La maintenabilité plaide pour la seconde version. Quant à la lisibilité entre les deux versions de la requête, c'est un choix personnel.

1.10.3 jsonb : index GIN jsonb_path_ops

> Créer un index GIN ainsi :
>
> ```
> CREATE INDEX personnes_gin ON json.personnes
> USING gin(personne jsonb_path_ops);
> ```
>
> Quelle taille fait-il ?

L'index peut être un peu long à construire (plusieurs dizaines de secondes) et est assez gros :

```
\di+ json.personnes_gin
                          Liste des relations
 Schéma |      Nom      | … |   Table   | … | Méthode d'accès | Taille | …
--------+---------------+---+-----------+---+-----------------+--------+-
 json   | personnes_gin | … | personnes | … | gin             | 230 MB |
```

> Retenter les requêtes précédentes. Lesquelles utilisent l'index ?

Les requêtes utilisant les égalités (que ce soit sur du texte ou en JSON) n'utilisent pas l'index :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT personne->'adresse'->>'ville'
FROM   json.personnes p
WHERE  personne->>'nom' = 'Lagaffe'
  AND  personne->>'prenom' = 'Gaston' ;

                          QUERY PLAN
-----------------------------------------------------------------------
 Gather (actual time=0.427..566.202 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=712208 read=122962
   ->  Parallel Seq Scan on personnes p (actual time=372.989..561.152 rows=0 loops=3)
         Filter: (((personne ->> 'nom'::text) = 'Lagaffe'::text) AND ((personne ->>
↪  'prenom'::text) = 'Gaston'::text))
         Rows Removed by Filter: 177548
         Buffers: shared hit=712208 read=122962
 Planning Time: 0.110 ms
 Execution Time: 566.228 ms
```

Par contre, la syntaxe `@>` (« contient ») utilise l'index, quelle que soit la manière dont on construit le JSON critère. Le gain en temps et en I/O (et en CPU) grâce à l'index est assez foudroyant. Et ceci, quelle que soit la manière dont on récupère les champs, puisqu'il n'y a plus qu'une poignée de lignes à analyser :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT personne->'adresse'->>'ville'
FROM   json.personnes p
WHERE  personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb ;

                             QUERY PLAN
---------------------------------------------------------------------
 Bitmap Heap Scan on personnes p (actual time=0.047..0.049 rows=1 loops=1)
   Recheck Cond: (personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb)
   Heap Blocks: exact=1
   Buffers: shared hit=8
   ->  Bitmap Index Scan on personnes_gin (actual time=0.026..0.027 rows=1 loops=1)
         Index Cond: (personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb)
         Buffers: shared hit=7
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.408 ms
 Execution Time: 0.081 ms
```

```
EXPLAIN (ANALYZE, VERBOSE)
SELECT r1.*, r2.*
FROM json.personnes,
LATERAL jsonb_to_record (personne)
        AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
        AS r2 (ville text, pays text)
WHERE personne @> '{"nom": "Lagaffe"}'::jsonb;

                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=25.90..235.79 rows=53 width=132) (actual time=0.051..0.063
↪  rows=2 loops=1)
   Output: r1.nom, r1.prenom, r1.date_naissance, r2.ville, r2.pays
   ->  Nested Loop  (cost=25.90..234.73 rows=53 width=821) (actual time=0.047..0.056
↪  rows=2 loops=1)
         Output: personnes.personne, r1.nom, r1.prenom, r1.date_naissance
         ->  Bitmap Heap Scan on json.personnes  (cost=25.90..233.66 rows=53
↪  width=753) (actual time=0.029..0.034 rows=2 loops=1)
               Output: personnes.id_personne, personnes.personne
               Recheck Cond: (personnes.personne @> '{"nom": "Lagaffe"}'::jsonb)
               Heap Blocks: exact=2
               ->  Bitmap Index Scan on personnes_gin  (cost=0.00..25.88 rows=53
↪  width=0) (actual time=0.017..0.018 rows=2 loops=1)
                     Index Cond: (personnes.personne @> '{"nom": "Lagaffe"}'::jsonb)
         ->  Function Scan on pg_catalog.jsonb_to_record r1  (cost=0.00..0.01 rows=1
↪  width=68) (actual time=0.009..0.009 rows=1 loops=2)
               Output: r1.nom, r1.prenom, r1.date_naissance
               Function Call: jsonb_to_record(personnes.personne)
   ->  Function Scan on pg_catalog.jsonb_to_record r2  (cost=0.01..0.01 rows=1
↪  width=64) (actual time=0.002..0.003 rows=1 loops=2)
         Output: r2.ville, r2.pays
         Function Call: jsonb_to_record((personnes.personne -> 'adresse'::text))
 Planning Time: 0.259 ms
 Execution Time: 0.098 ms
```

Les requêtes sans filtre n'utilisent pas l'index, bien sûr.

1.10.4 jsonb et tableaux

> Récupérer les numéros de téléphone de Léon Prunelle.

```
--(Syntaxe pour PostgreSQL 14 minimum)
SELECT personne['adresse']['telephones'],
       personne['adresse']->'telephones',
       personne['adresse']['telephones']#>'{}',
       personne['adresse']['telephones']->0,
       personne['adresse']->>'telephones',
       personne['adresse']['telephones']#>>'{}',
       personne['adresse']['telephones']->>0
FROM   json.personnes p
WHERE  personne @> '{"nom": "Prunelle", "prenom": "Léon"}'::jsonb ;
```

Le sous-attribut `telephones` est un tableau. La syntaxe `->0` ne renvoie que le premier élément :

```
-[ RECORD 1 ]-------------------------
personne | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | "0129951489"
?column? | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | 0129951489
```

Les 4 premières lignes renvoient un `jsonb`, les trois dernières sa conversion en texte :

```
 \gdesc
  Column   | Type
-----------+-------
 personne  | jsonb
 ?column?  | jsonb
 ?column?  | jsonb
 ?column?  | jsonb
 ?column?  | text
 ?column?  | text
 ?column?  | text
```

> Afficher les noms et prénoms de Prunelles, et un tableau de champs texte contenant ses numéros de téléphone (utiliser `jsonb_array_elements_text`).

Il vaut mieux ne pas « bricoler » avec des conversions manuelles du JSON en texte puis en tableau. La fonction dédiée est `jsonb_array_elements_text`.

```
SELECT personne->>'prenom' AS  prenom, personne->>'nom' AS nom,
       jsonb_array_elements_text (personne->'adresse'->'telephones') AS tel
FROM   json.personnes p
WHERE  personne @> '{"nom": "Prunelle", "prenom": "Léon"}'::jsonb ;
```

```
 prenom |   nom    |    tel
--------+----------+------------
 Léon   | Prunelle | 0129951489
 Léon   | Prunelle | 0678327400
```

Cependant on multiplie les lignes par le nombre de numéros de téléphone, et il faut réagréger :

```
SELECT personne->>'prenom' AS  prenom, personne->>'nom' AS nom,
       (SELECT array_agg (t) FROM
        jsonb_array_elements_text (personne->'adresse'->'telephones') tels(t)
       ) AS tels
FROM   json.personnes p
WHERE  personne @> '{"nom": "Prunelle", "prenom": "Léon"}'::jsonb ;
```

```
 prenom |   nom    |          tels
--------+----------+-------------------------
 Léon   | Prunelle | {0129951489,0678327400}
```

```
\gdesc
 Column |  Type
--------+--------
 prenom | text
 nom    | text
 tels   | text[]
```

La version suivante fonctionnerait aussi dans ce cas précis (cependant elle sera moins performante s'il y a beaucoup de lignes, car PostgreSQL voudra faire un agrégat global au lieu d'un simple parcours ; il faudra aussi vérifier que la clé d'agrégation tient compte d'homonymes).

```
SELECT personne->>'prenom' AS  prenom, personne->>'nom' AS nom,
       array_agg (t) AS tels
FROM   json.personnes p
       LEFT OUTER JOIN LATERAL jsonb_array_elements_text (
                         personne->'adresse'->'telephones') AS tel(t) ON (true)
WHERE  personne @> '{"nom": "Prunelle", "prenom": "Léon"}'::jsonb
GROUP BY 1,2 ;
```

(Noter que la fonction sœur `jsonb_array_elements()` renverrait, elle, des JSON.)

1.10.5 Accès JSONPath

> Comparer le résultat et les performances de ces deux requêtes, qui récupèrent aussi les numéros de téléphone de Prunelle :

```
SELECT jsonb_path_query (personne,
'$.adresse.telephones[*] ? ($.nom == "Prunelle" && $.prenom == "Léon")' ) #>>'{}' AS
↪  tel
FROM json.personnes ;
```

```
SELECT jsonb_path_query (personne, '$.adresse.telephones[*]')#>>'{}'
  AS tel
FROM json.personnes
WHERE  personne @@ '$.nom == "Prunelle" && $.prenom == "Léon"' ;
```

Le résultat est le même dans les deux cas :

```
    tel
------------
 0129951489
 0678327400
```

Par contre, le plan et les temps d'exécutions sont totalement différents. La clause `jsonb_path_query` unique parcourt complètement la table :

```
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS)
SELECT jsonb_path_query (personne,
'$.adresse.telephones[*] ? ($.nom == "Prunelle" && $.prenom == "Léon")'
) #>>'{}' AS tel
FROM   json.personnes ;
```

```
                              QUERY PLAN
---------------------------------------------------------------------
 Gather (actual time=1290.193..1293.496 rows=2 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=729122 read=123788
   ->  Result (actual time=1113.807..1269.568 rows=1 loops=3)
         Buffers: shared hit=729122 read=123788
         ->  ProjectSet (actual time=1113.803..1269.564 rows=1 loops=3)
               Buffers: shared hit=729122 read=123788
```

```
               ->  Parallel Seq Scan on personnes (actual time=0.240..304.804
↪  rows=177548 loops=3)
                     Buffers: shared read=55888
 Planning Time: 0.134 ms
 Execution Time: 1293.548 ms
```

Tandis que la séparation du filtrage et de l'affichage permet à PostgreSQL de sélectionner les lignes, et donc de passer par un index avant de procéder à l'affichage.

```
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS)
SELECT jsonb_path_query (personne, '$.adresse.telephones[*]')#>>'{}' AS tel
FROM   json.personnes
WHERE  personne @@ '$.nom == "Prunelle" && $.prenom == "Léon"' ;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Result (actual time=2.196..2.207 rows=2 loops=1)
   Buffers: shared hit=2 read=6
   ->  ProjectSet (actual time=2.186..2.194 rows=2 loops=1)
         Buffers: shared hit=2 read=6
         ->  Bitmap Heap Scan on personnes (actual time=2.167..2.170 rows=1 loops=1)
               Recheck Cond: (personne @@ '($."nom" == "Prunelle" && $."prenom" ==
↪  "Léon")'::jsonpath)
               Heap Blocks: exact=1
               Buffers: shared hit=2 read=6
               ->  Bitmap Index Scan on personnes_gin (actual time=2.113..2.114
↪  rows=1 loops=1)
                     Index Cond: (personne @@ '($."nom" == "Prunelle" && $."prenom" ==
↪  "Léon")'::jsonpath)
                     Buffers: shared hit=2 read=5
 Planning:
   Buffers: shared read=4
 Planning Time: 2.316 ms
 Execution Time: 2.269 ms
```

(À la place de `@@`, la syntaxe classique `@>` avec un JSON comme critère, est aussi performante dans ce cas simple.)

> Chercher qui possède le numéro de téléphone `0650041821` avec la syntaxe JSONPath.

Ces deux syntaxes sont équivalentes :

```
SELECT personne->>'nom', personne->>'prenom'
FROM   json.personnes
WHERE  personne @@ '$.adresse.telephones[*] == "0650041821" ' ;

SELECT personne->>'nom', personne->>'prenom'
FROM   json.personnes
WHERE  personne @? '$.adresse.telephones[*] ? (@ == "0650041821")' ;
```

```
 ?column?  | ?column?
-----------+----------
 Delacroix | Justine
```

Dans les deux cas, `EXPLAIN` montre que l'index GIN est bien utilisé.

> Compter le nombre de personnes habitant à Paris ou Bruxelles avec :
>
> — la syntaxe `@>` et un `OR` ;
> — une syntaxe JSONPath `@?` et un « ou » logique (`||`);
> — une syntaxe JSONPath `@?` et une regex `@.ville like_regex "^(Paris|Bruxelles)$"`.

Vous devez trouver 63 personnes avec la version complète de la base.

Cet appel va utiliser l'index GIN :

```
EXPLAIN SELECT count(*) FROM json.personnes
WHERE personne @> '{"adresse": {"ville": "Paris"}}'::jsonb
OR    personne @> '{"adresse": {"ville": "Bruxelles"}}'::jsonb ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=467.65..467.66 rows=1 width=8)
   ->  Bitmap Heap Scan on personnes  (cost=51.82..467.38 rows=107 width=0)
         Recheck Cond: ((personne @> '{"adresse": {"ville": "Paris"}}'::jsonb) OR
↪  (personne @> '{"adresse": {"ville": "Bruxelles"}}'::jsonb))
         ->  BitmapOr  (cost=51.82..51.82 rows=107 width=0)
               ->  Bitmap Index Scan on personnes_gin  (cost=0.00..25.88 rows=53
↪  width=0)
                     Index Cond: (personne @> '{"adresse": {"ville": "Paris"}}'::jsonb)
               ->  Bitmap Index Scan on personnes_gin  (cost=0.00..25.88 rows=53
↪  width=0)
                     Index Cond: (personne @> '{"adresse": {"ville":
↪  "Bruxelles"}}'::jsonb)
```

Cet appel aussi :

```
EXPLAIN SELECT count(*) FROM json.personnes
WHERE personne @? '$.adresse ? ( @.ville == "Paris" || @.ville == "Bruxelles") ' ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=2020.86..2020.87 rows=1 width=8)
   ->  Bitmap Heap Scan on personnes  (cost=48.13..2019.53 rows=533 width=0)
         Recheck Cond: (personne @? '$."adresse"?(@."ville" == "Paris" || @."ville"
↪  == "Bruxelles")'::jsonpath)
         ->  Bitmap Index Scan on personnes_gin  (cost=0.00..47.99 rows=533 width=0)
               Index Cond: (personne @? '$."adresse"?(@."ville" == "Paris" ||
↪  @."ville" == "Bruxelles")'::jsonpath)
```

Par contre, l'index GIN est inutilisable si l'on demande une expression régulière (aussi simple soit-elle) :

```
EXPLAIN SELECT count(*) FROM json.personnes
WHERE personne @? '$.adresse ? ( @.ville like_regex "^(Paris|Bruxelles)$" ) ' ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Finalize Aggregate  (cost=56899.96..56899.97 rows=1 width=8)
   ->  Gather  (cost=56899.75..56899.96 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=55899.75..55899.76 rows=1 width=8)
```

```
               ->  Parallel Seq Scan on personnes  (cost=0.00..55899.19 rows=222
↪  width=0)
                     Filter: (personne @? '$."adresse"?(@."ville" like_regex
↪  "^(Paris|Bruxelles)$")'::jsonpath)
```

1.10.6 Index fonctionnel, colonne générée et JSON

> Le compte du nombre de personne par pays doit être optimisé au maximum. Ajouter un index fonctionnel sur l'attribut `pays`. Tester l'efficacité sur une recherche, et un décompte de toutes les personnes par pays.

Suivant la syntaxe préférée, l'index peut être par exemple ceci :

```
CREATE INDEX personnes_pays_idx ON json.personnes
USING btree ( (personne->'adresse'->>'pays'));
VACUUM ANALYZE json.personnes ;
```

L'index contient peu de valeurs et fait au plus 3 Mo (beaucoup plus sur une version antérieure à PostgreSQL 13).

Cet index est utilisable pour une recherche **à condition que la syntaxe de l'expression soit rigoureusement identique**, ce qui limite les cas d'usage.

```
EXPLAIN (ANALYZE,BUFFERS) SELECT count(*) FROM json.personnes
WHERE personne->'adresse'->>'pays' ='Belgique' ;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Aggregate  (cost=8.38..8.39 rows=1 width=8) (actual time=0.045..0.046 rows=1
↪  loops=1)
   Buffers: shared hit=6
   ->  Index Scan using personnes_pays_idx on personnes  (cost=0.42..8.38 rows=1
↪  width=0) (actual time=0.032..0.037 rows=3 loops=1)
         Index Cond: (((personne -> 'adresse'::text) ->> 'pays'::text) =
↪  'Belgique'::text)
         Buffers: shared hit=6
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.154 ms
 Execution Time: 0.078 ms
```

Par contre, pour le décompte complet, il n'a aucun intérêt :

```
EXPLAIN SELECT personne->'adresse'->>'pays', count(*)
FROM json.personnes GROUP BY 1 ;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Finalize GroupAggregate  (cost=61309.88..61312.72 rows=11 width=40)
   Group Key: (((personne -> 'adresse'::text) ->> 'pays'::text))
   ->  Gather Merge  (cost=61309.88..61312.45 rows=22 width=40)
         Workers Planned: 2
         ->  Sort  (cost=60309.86..60309.88 rows=11 width=40)
               Sort Key: (((personne -> 'adresse'::text) ->> 'pays'::text))
```

```
               ->  Partial HashAggregate  (cost=60309.50..60309.67 rows=11 width=40)
                     Group Key: ((personne -> 'adresse'::text) ->> 'pays'::text)
                     ->  Parallel Seq Scan on personnes  (cost=0.00..59199.38
↪  rows=222025 width=32)
```

En effet, un index fonctionnel ne permet pas un *Index Only Scan*. Pourtant, il pourrait être très intéressant ici.

> Ajouter un champ généré dans `json.personne`, correspondant à l'attribut `pays`.

Attention, l'ordre va réécrire la table, ce qui peut être long (de l'ordre de la minute, suivant le matériel) :

```
ALTER TABLE json.personnes ADD COLUMN pays text
GENERATED ALWAYS AS ( personne->'adresse'->>'pays' ) STORED ;
VACUUM ANALYZE json.personnes ;
```

> Comparer les temps d'exécution du décompte des pays par l'attribut, et par cette colonne générée.

```
\timing on

SELECT personne->'adresse'->>'pays', count(*) FROM json.personnes GROUP BY 1 ;

         ?column?         | count
--------------------------+-------
 België                   | 39597
 Belgique                 |     3
 Denmark                  | 21818
 España                   | 79899
 France                   | 82936
 Italia                   | 33997
 Lietuva                  |  6606
 Poland                   | 91099
 Portugal                 | 17850
 United Kingdom           | 64926
 United States of America | 93914

Temps : 601,815 ms
```

Par contre, la lecture directe du champ est nettement plus rapide :

```
SELECT pays, count(*) FROM json.personnes GROUP BY 1 ;

…
Temps : 58,811 ms
```

Le plan est pourtant le même : un *Seq Scan*, faute de clause de filtrage et d'index, suivi d'un agrégat parallélisé n'utilisant que quelques kilooctets de mémoire.

```
                              QUERY PLAN
---------------------------------------------------------------------
 Finalize GroupAggregate  (cost=59529.88..59532.67 rows=11 width=19) (actual
↪  time=61.211..64.244 rows=11 loops=1)
   Group Key: pays
```

```
   Buffers: shared hit=55219
   ->  Gather Merge  (cost=59529.88..59532.45 rows=22 width=19) (actual
↪  time=61.204..64.235 rows=33 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=55219
         ->  Sort  (cost=58529.85..58529.88 rows=11 width=19) (actual
↪  time=45.186..45.188 rows=11 loops=3)
               Sort Key: pays
               Sort Method: quicksort  Memory: 25kB
               Worker 0:  Sort Method: quicksort  Memory: 25kB
               Worker 1:  Sort Method: quicksort  Memory: 25kB
               Buffers: shared hit=55219
               ->  Partial HashAggregate  (cost=58529.55..58529.66 rows=11 width=19)
↪  (actual time=45.159..45.161 rows=11 loops=3)
                     Group Key: pays
                     Buffers: shared hit=55203
                     ->  Parallel Seq Scan on personnes  (cost=0.00..57420.70
↪  rows=221770 width=11) (actual time=0.005..13.678 rows=177548 loops=3)
                           Buffers: shared hit=55203
 Planning Time: 0.105 ms
 Execution Time: 64.297 ms
```

Le champ généré a donc un premier intérêt en terme de rapidité de lecture des champs, surtout avec des JSON importants comme ici.

> Créer un index B-tree sur la colonne générée `pays`. Consulter les statistiques dans `pg_stats`. Cet index est-il utilisable pour des filtres et le décompte par `pays` ?

```
CREATE INDEX personnes_g_pays_btree ON json.personnes (pays);
VACUUM ANALYZE json.personnes ;
```

Ces deux ordres ne durent qu'1 ou 2 secondes.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT p.pays, count(*)
FROM json.personnes p
GROUP BY 1 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize GroupAggregate  (cost=1000.45..8885.35 rows=10 width=19) (actual
↪  time=7.629..49.349 rows=11 loops=1)
   Group Key: pays
   Buffers: shared hit=477
   ->  Gather Merge  (cost=1000.45..8885.15 rows=20 width=19) (actual
↪  time=7.625..49.340 rows=11 loops=1)
         Workers Planned: 2
         Workers Launched: 0
         Buffers: shared hit=477
         ->  Partial GroupAggregate  (cost=0.42..7882.82 rows=10 width=19) (actual
↪  time=7.371..49.034 rows=11 loops=1)
               Group Key: pays
               Buffers: shared hit=477
               ->  Parallel Index Only Scan using personnes_g_pays_btree on personnes
↪  p  (cost=0.42..6771.79 rows=222186 width=11) (actual time=0.023..22.578
↪  rows=532645 loops=1)
```

```
                    Heap Fetches: 0
                    Buffers: shared hit=477
 Planning Time: 0.114 ms
 Execution Time: 49.391 ms
```

Le gain en temps est appréciable. Mais l'intérêt principal réside ici dans le nombre de blocs lus divisé par 100 ! Le nouvel index ne fait que 3 Mo.

```
\di+ json.personnes*
                              Liste des relations
 Schéma |          Nom           | Type  | … | Méthode d'accès | Taille  | …
--------+------------------------+-------+---+-----------------+---------+---
 json   | personnes_g_pays_btree | index | … | btree           | 3664 kB |
 json   | personnes_gin          | index | … | gin             | 230 MB  |
 json   | personnes_pays_idx     | index | … | btree           | 3664 kB |
```

> (Optionnel) Créer des colonnes générées sur `nom`, `prenom`, `date_naissance`, et `ville` (en un seul ordre). Reprendre la requête plus haut qui les affiche tous et comparer les performances.

Un champ va poser problème : la date de naissance. En effet, la date est stockée au format texte, il faudra soi-même faire la conversion. De plus, un simple opérateur `::date` ne peut être utilisé dans une expression de `GENERATED` car il n'est pas « immutable » (pour des raisons techniques[45]).

Un contournement pas très performant est celui-ci :

```
ALTER TABLE json.personnes
ADD COLUMN nom text GENERATED ALWAYS AS (personne->>'prenom') STORED,
ADD COLUMN prenom text GENERATED ALWAYS AS (personne->>'nom') STORED,
ADD COLUMN date_naissance date
  GENERATED ALWAYS AS (
    make_date (left(personne->>'date_naissance',4)::int,
               substring(personne->>'date_naissance',6,2)::int,
               left(personne->>'date_naissance',2)::int))
  STORED,
ADD COLUMN ville text GENERATED ALWAYS AS ( personne->'adresse'->>'ville') STORED ;

VACUUM ANALYZE json.personnes ;
```

Une autre possibilité plus performante est d'enrober `to_date()` dans une fonction immutable, puisqu'il n'y a, dans ce cas précis, pas d'ambiguïté sur le format ISO :

```
CREATE OR REPLACE FUNCTION to_date_immutable (text)
RETURNS date
-- Cette fonction requiert que les dates soient bien
-- stockées au format ci-dessous
-- et ne fait aucune gestion d'erreur sinon
LANGUAGE sql
IMMUTABLE PARALLEL SAFE
AS $body$
  SELECT to_date($1, 'YYYY-MM-DD');
$body$ ;
```

et l'ordre devient :

[45] https://www.postgresql.org/message-id/flat/201111290311.pAT3Bf318279%40momjian.us

```
ALTER TABLE json.personnes
…
ADD COLUMN date_naissance  date
 GENERATED ALWAYS AS (to_date_immutable (personne->>'date_naissance')) STORED,
…;
```

Les conversions de texte vers des dates sont des sources fréquentes de problèmes. Le conseil habituel est de toujours stocker une date dans un champ de type `date` ou `timestamp` / `timestamptz`. Mais si elle provient d'un JSON, il faudra gérer soi-même la conversion.

Quelle que soit la méthode, la requête suivante :

```
SELECT nom, prenom, date_naissance, ville, pays FROM json.personnes ;
```

est **beaucoup** plus rapide que :

```
SELECT r1.nom, r1.prenom, r1.date_naissance, r2.ville, r2.pays
FROM json.personnes,
LATERAL jsonb_to_record (personne)
        AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
        AS r2 (ville text, pays text) ;
```

elle-même plus rapide que les extractions manuelles des attributs un à un, comme vu plus haut.

Certes, la table est un peu plus grosse, mais le coût d'insertion des colonnes générées est donc souvent rentable pour les champs fréquemment utilisés.

1.10.7 jsonb et mise à jour

> Ajouter l'attribut `animaux` à Gaston Lagaffe, avec la valeur 18. Vérifier en relisant la ligne.

```
UPDATE json.personnes
SET personne = personne || '{"animaux": 18}'
WHERE personne @>  '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb;

SELECT personne->>'animaux'
FROM json.personnes WHERE personne @>  '{"nom": "Lagaffe", "prenom":
↪  "Gaston"}'::jsonb ;

 ?column?
----------
 18
```

> Ajouter l'attribut `animaux` à 2% des individus au hasard, avec une valeur 1 ou 2.

On utilise ici la fonction `jsonb_build_object()`, plus adaptée à la construction d'un JSON qui n'est pas une constante. Le choix des individus peut se faire de plusieurs manières, par exemple avec `random()`, `mod()`…

```
UPDATE json.personnes
SET personne = personne ||
    jsonb_build_object ('animaux', 1+mod ((personne->>'numgen')::int, 50))
WHERE mod((personne->>'numgen')::int,50) = 0 ;
```

```
UPDATE 10653
```

```
-- Conseillé après chaque mise à jour importante
VACUUM ANALYZE json.personnes ;
```

> Compter le nombre de personnes avec des animaux (avec ou sans JSONPath). Proposer un index qui pourrait convenir à d'autres futurs nouveaux attributs peu fréquents.

Ces requêtes renvoient 10654, mais effectuent toutes un *Seq Scan* avec une durée d'exécution aux alentours de la seconde :

```
SELECT count(*) FROM json.personnes
WHERE (personne->>'animaux')::int > 0 ;
```

```
SELECT count(*) FROM json.personnes
WHERE  personne ? 'animaux' ;
```

```
SELECT count(*) FROM json.personnes
WHERE  personne @@ '$.animaux > 0' ;
```

```
SELECT count(*) FROM json.personnes
WHERE  personne @? '$.animaux ? (@ > 0) ' ;
```

(Remarquer que les deux dernières requêtes utiliseraient l'index GIN pour des égalités comme `(@ == 0)` ou `(@ == 18)`, et seraient presque instantanées. Là encore, c'est une limite des index GIN.)

On pourrait indexer `(personne->>'animaux')::int`, ce qui serait excellent pour la première requête, mais ne conviendrait pas à d'autres critères.

L'opérateur `?` ne sait pas utiliser l'index GIN `jsonb_path_ops` existant. Par contre, il peut profiter de l'opérateur GIN par défaut :

```
CREATE INDEX personnes_gin_df ON json.personnes USING gin (personne) ;
```

```
EXPLAIN
SELECT count(*) FROM json.personnes
WHERE personne ? 'animaux' ;
```

```
                              QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=42263.34..42263.35 rows=1 width=8)
   ->  Bitmap Heap Scan on personnes  (cost=167.47..42209.54 rows=21521 width=0)
         Recheck Cond: (personne ? 'animaux'::text)
         ->  Bitmap Index Scan on personnes_gin_df  (cost=0.00..162.09 rows=21521
 ↪  width=0)
               Index Cond: (personne ? 'animaux'::text)
```

Il est directement utilisable par tout autre attribut :

```
SELECT count(*) FROM json.personnes
WHERE personne ? 'voitures' ;
```

```
                              QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=233.83..233.84 rows=1 width=8)
   ->  Bitmap Heap Scan on personnes  (cost=25.89..233.70 rows=53 width=0)
         Recheck Cond: (personne ? 'voitures'::text)
        ->  Bitmap Index Scan on personnes_gin_df  (cost=0.00..25.88 rows=53 width=0)
              Index Cond: (personne ? 'voitures'::text)
```

Cet index avec l'opérateur `jsonb_ops` a par contre le gros inconvénient d'être encore plus gros que l'index GIN avec `jsonb_path_ops` (303 Mo contre 235 Mo), et d'alourdir encore les mises à jour. Il peut cependant remplacer ce dernier, de manière un peu moins performante. Il faut aviser selon les requêtes, la place, les écritures…

1.10.8 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.

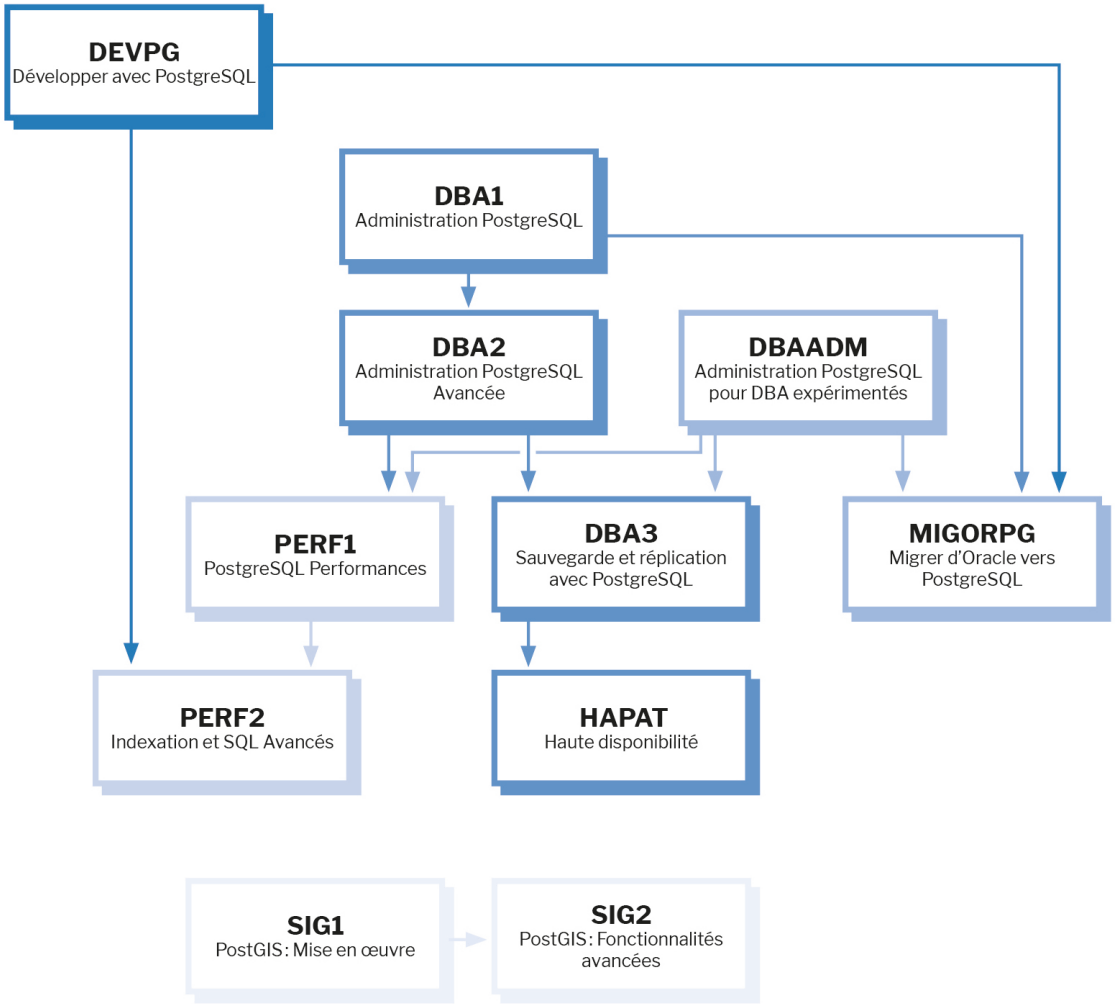
```
psql -c "\lo_export loid_retourné '/home/dalibo/passwd_serveur';"
```

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
— DEV1 : Introduction à SQL
https://dali.bo/dev1
— 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.