Module S9

# Types avancés


24.04

# Table des matières

## Sur ce document

| | |
|---|---|
| **Formation** | Module S9 |
| **Titre** | Types avancés |
| **Révision** | 24.04 |
| **PDF** | https://dali.bo/s9_pdf |
| **EPUB** | https://dali.bo/s9_epub |
| **HTML** | https://dali.bo/s9_html |
| **Slides** | https://dali.bo/s9_slides |
| **TP** | https://dali.bo/s9_tp |
| **TP (solutions)** | https://dali.bo/s9_solutions |

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

## Chers lectrices & lecteurs,

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

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

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

## À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

## Remerciements

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

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

## Forme de ce manuel

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

## Licence Creative Commons CC-BY-NC-SA

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

## Marques déposées

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

## Versions de PostgreSQL couvertes

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

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

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

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

# 1/ Types avancés

PostgreSQL offre des types avancés :

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

## 1.1 TYPES COMPOSÉS : GÉNÉRALITÉS

- Un champ = plusieurs attributs
- De loin préférable à une table Entité/Attribut/Valeur
- Uniquement si le modèle relationnel n'est pas assez souple
- 3 types dans PostgreSQL :
  - `hstore` : clé/valeur
  - `json` : JSON, stockage texte, validation syntaxique, fonctions d'extraction
  - `jsonb` : JSON, stockage binaire, accès rapide, fonctions d'extraction, de requêtage, indexation avancée

Ces types sont utilisés quand le modèle relationnel n'est pas assez souple, donc s'il est nécessaire d'ajouter dynamiquement des colonnes à la table suivant les besoins du client, ou si le détail des attributs d'une entité n'est pas connu (modélisation géographique par exemple), etc.

La solution traditionnelle est de créer des tables entité/attribut de ce format :

```
CREATE TABLE attributs_sup (entite int, attribut text, valeur text);
```

On y stocke dans `entite` la clé de l'enregistrement de la table principale, dans `attribut` la colonne supplémentaire, et dans `valeur` la valeur de cet attribut. Ce modèle présente l'avantage évident de résoudre le problème. Les défauts sont par contre nombreux :

- Les attributs d'une ligne peuvent être totalement éparpillés dans la table `attributs_sup` : récupérer n'importe quelle information demandera donc des accès à de nombreux blocs différents.
- Il faudra plusieurs requêtes (au moins deux) pour récupérer le détail d'un enregistrement, avec du code plus lourd côté client pour fusionner le résultat des deux requêtes, ou bien une requête effectuant des jointures (autant que d'attributs, sachant que le nombre de jointures complexifie énormément le travail de l'optimiseur SQL) pour retourner directement l'enregistrement complet.

Toute recherche complexe est très inefficace : une recherche multi-critères sur ce schéma va être extrêmement peu performante. Les statistiques sur les valeurs d'un attribut deviennent nettement moins faciles à estimer pour PostgreSQL. Quant aux contraintes d'intégrité entre valeurs, elles deviennent pour le moins complexes à gérer.

Les types `hstore`, `json` et `jsonb` permettent de résoudre le problème autrement. Ils permettent de stocker les différentes entités dans un seul champ pour chaque ligne de l'entité. L'accès aux attributs se fait par une syntaxe ou des fonctions spécifiques.

Il n'y a même pas besoin de créer une table des attributs séparée : le mécanisme du « TOAST » permet de déporter les champs volumineux (texte, JSON, `hstore`…) dans une table séparée gérée par PostgreSQL, éventuellement en les compressant, et cela de manière totalement transparente. On y gagne donc en simplicité de développement.

## 1.2 HSTORE

Stocker des données non structurées

- Extension
- Stockage Clé/Valeur, uniquement texte
- Binaire
- Indexable
- Plusieurs opérateurs disponibles

**hstore** est une extension, fournie en « contrib ». Elle est donc systématiquement disponible. L'installer permet d'utiliser le type de même nom. On peut ainsi stocker un ensemble de clés/valeurs, exclusivement textes, dans un unique champ.

Ces champs sont indexables et peuvent recevoir des contraintes d'intégrité (unicité, non recouvrement…).

Les `hstore` ne permettent par contre qu'un modèle « plat ». Il s'agit d'un pur stockage clé-valeur. Si vous avez besoin de stocker des informations davantage orientées document, vous devrez vous tourner vers un type JSON.

Ce type perd donc de son intérêt depuis que PostgreSQL 9.4 a apporté le type `jsonb`. Il lui reste sa simplicité d'utilisation.

### 1.2.1 hstore : exemple

```
CREATE EXTENSION hstore ;

CREATE TABLE animaux (nom text, caract hstore);
INSERT INTO animaux VALUES ('canari','pattes=>2,vole=>oui');
INSERT INTO animaux VALUES ('loup','pattes=>4,carnivore=>oui');
INSERT INTO animaux VALUES ('carpe','eau=>douce');

CREATE INDEX idx_animaux_donnees ON animaux
                                 USING gist (caract);

SELECT *, caract->'pattes' AS nb_pattes FROM animaux
WHERE caract@>'carnivore=>oui';

 nom  |              caract                | nb_pattes
------+------------------------------------+-----------
 loup | "pattes"=>"4", "carnivore"=>"oui" | 4
```

Les ordres précédents installent l'extension, créent une table avec un champ de type `hstore`, insèrent trois lignes, avec des attributs variant sur chacune, indexent l'ensemble avec un index GiST, et enfin recherchent les lignes où l'attribut `carnivore` possède la valeur `t`.

```
# SELECT * FROM animaux ;

  nom   |              caract
--------+----------------------------------
 canari | "vole"=>"oui", "pattes"=>"2"
 loup   | "pattes"=>"4", "carnivore"=>"oui"
 carpe  | "eau"=>"douce"
```

Les différentes fonctions disponibles sont bien sûr dans la documentation[1].

Par exemple :

```
# UPDATE animaux SET caract = caract||'poil=>t'::hstore
  WHERE nom = 'loup' ;

# SELECT * FROM animaux WHERE caract@>'carnivore=>oui';

 nom  |                      caract
------+-------------------------------------------------
 loup | "poil"=>"t", "pattes"=>"4", "carnivore"=>"oui"
```

Il est possible de convertir un `hstore` en tableau :

```
# SELECT hstore_to_matrix(caract) FROM animaux
  WHERE caract->'vole' = 'oui';

    hstore_to_matrix
-------------------------
 { {vole,oui},{pattes,2} }
```

ou en JSON :

```
# SELECT caract::jsonb FROM animaux
  WHERE (caract->'pattes')::int > 2;

                      caract
---------------------------------------------------
 {"pattes": "4", "poil": "t", "carnivore": "oui"}
```

L'indexation de ces champs peut se faire avec divers types d'index. Un index unique n'est possible qu'avec un index B-tree classique. Les index GIN ou GiST sont utiles pour rechercher des valeurs d'un attribut. Les index hash ne sont utiles que pour des recherches d'égalité d'un champ entier ; par contre ils sont très compacts. (Rappelons que les index hash sont inutilisables avant PostgreSQL 10).

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

## 1.3 JSON

```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "postalCode": "75009"
  }
}
```

- `json` : format texte
- `jsonb` : format binaire, à préférer
- `jsonpath` : SQL/JSON paths (PG 12+)

Le format JSON[2] est devenu extrêmement populaire. Au-delà d'un simple stockage clé/valeur, il permet de stocker des tableaux, ou des hiérarchies, de manière plus simple et lisible qu'en XML. Par exemple, pour décrire une personne, on peut utiliser cette structure :

```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "state": "",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    {
      "type": "personnel",
      "number": "06 12 34 56 78"
    },
    {
      "type": "bureau",
      "number": "07 89 10 11 12"
    }
  ],
  "children": [],
  "spouse": null
}
```

[2] 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. Les opérateurs SQL/JSON path[3] ont été ajoutés dans PostgreSQL 12, suite à l'introduction du JSON dans le standard SQL:2016. Ils permettent de spécifier des parties d'un champ JSON.

### 1.3.1 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 », qui compresse au besoin les plus grands champs, 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.

On préférera généralement le type binaire `jsonb`, pour les performances, notamment l'indexation GIN, et ses fonctionnalités supplémentaires. Au final, l'intérêt du type `json` est surtout de conserver un objet JSON sous sa forme originale.

Beaucoup d'exemples suivants avec `jsonb` sont aussi applicables à `json`.

### 1.3.2 Type jsonb

- **JSON** au format **B**inaire
- Indexation GIN
- Gestion du langage JSON Path (v12+)

Le type `jsonb` permet de stocker les données dans un format binaire optimisé. Ainsi, il n'est plus nécessaire de désérialiser l'intégralité du document pour accéder à une propriété.

[3] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

Pour un exemple extrême (document JSON d'une centaine de Mo), voici une comparaison des performances entre les deux types `json` et `jsonb` pour la récupération d'un champ sur 1300 lignes :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_json;

                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on test_json  (cost=0.00..26.38 rows=1310 width=32)
                        (actual time=893.454..912.168 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.021 ms
 Execution time: 912.194 ms

EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_jsonb;

                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on test_jsonb  (cost=0.00..26.38 rows=1310 width=32)
                         (actual time=77.707..84.148 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.026 ms
 Execution time: 84.177 ms
```

### 1.3.3 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
```

### 1.3.4 JSON : Exemple d'utilisation

```
CREATE TABLE personnes (datas jsonb );

INSERT INTO personnes  (datas) VALUES ('
{
  "firstName": "Jean",
  "lastName": "Valjean",
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    { "number": "06 12 34 56 78" },
    {"type": "bureau",
     "number": "07 89 10 11 12"}
  ],
  "children": [],
  "spouse": null
}
'),
('
{
  "firstName": "Georges",
  "lastName": "Durand",
  "address": {
    "streetAddress": "27 rue des Moulins",
    "city": "Châteauneuf",
    "postalCode": "45990"
  },
  "phoneNumbers": [
    { "number": "06 21 34 56 78" },
    {"type": "bureau",
     "number": "07 98 10 11 12"}
  ],
  "children": [],
  "spouse": null
}
');
```

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

### 1.3.5 JSON : Affichage de champs

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

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

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

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

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

Le type `json` dispose de nombreuses fonctions de manipulation et d'extraction. Les opérateurs `->>` et `->` renvoient respectivement une valeur au format texte, et au format JSON.

```
SELECT datas->>'firstName' AS prenom,
       datas->'address'    AS addr
FROM personnes   \gdesc
```

```
 Column | Type
--------+-------
 prenom | text
 addr   | jsonb
```

```
\g
```

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

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

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

```
   villes
-------------
```

```
 Paris
 Châteauneuf
```

Depuis la version 14, une autre syntaxe plus claire est disponible, plus simple, et qui renvoie du JSON :

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

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

`jsonb_array_elements` permet de parcourir des tableaux de JSON :

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

```
     numero
----------------
 06 12 34 56 78
 07 89 10 11 12
 06 21 34 56 87
 07 98 10 11 13
```

### 1.3.6 Conversions jsonb / relationnel

- Construire un objet JSON depuis un ensemble :
  - `json_object_agg()`
- Construire un ensemble de tuples depuis un objet JSON :
  - `jsonb_each()`
  - `jsonb_to_record()`
- Manipuler des tableaux :
  - `jsonb_array_elements()`
  - `jsonb_to_recordset()`

Les fonctions permettant de construire du `jsonb`, ou de le manipuler de manière ensembliste permettent une très forte souplesse. Il est aussi possible de déstructurer des tableaux, mais il est compliqué de requêter sur leur contenu.

Par exemple, si l'on souhaite filtrer des documents de la sorte pour ne ramener que ceux dont une catégorie est `categorie` :

```
{
  "id": 3,
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
```

```
CREATE TABLE json_table (id serial, document jsonb);
INSERT INTO json_table (document) VALUES ('
{
  "id": 3,
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
');
```

```
SELECT document->'id'
FROM json_table j,
LATERAL jsonb_array_elements(document #> '{sous_document, mon_tableau}')
    AS elements_tableau
WHERE elements_tableau->>'categorie' = 'categorie';
```

Ce type de requête devient rapidement compliqué à écrire, et n'est pas indexable.

### 1.3.7 JSON : performances

Inconvénients par rapport à un modèle normalisé :

- Perte d'intégrité (types, contraintes, FK…)
- Complexité du code
- Pas de statistiques sur les clés JSON !
- Pas forcément plus léger en disque
  - clés répétées
- Lire 1 champ = lire tout le JSON
  - voire accès table TOAST
- Mise à jour : tout ou rien

Les champs JSON sont très pratiques quand le schéma est peu structuré. Mais la complexité supplémentaire de code nuit à la lisibilité des requêtes. En termes de performances, ils sont coûteux, pour les raisons que nous allons voir.

Les contraintes d'intégrité sur les types, les tailles, les clés étrangères… ne sont pas disponibles. Les contraintes protègent de nombreux bugs, mais elles sont aussi une aide précieuse pour l'optimiseur.

Chaque JSON récupéré l'est en bloc. Si un seul champ est récupéré, PostgreSQL devra charger tout le JSON et le décomposer. Cela peut même coûter un accès supplémentaire à une table TOAST pour les gros JSON. Rappelons que le mécanisme du TOAST permet à PostgreSQL de compresser à la volée un grand champ texte, binaire, JSON… et/ou de le déporter dans une table annexe interne, le tout étant totalement transparent pour l'utilisateur. Pour les détails, voir cet extrait de la formation DBA2[4].

Il n'y a pas de mise à jour partielle : changer un champ implique de décomposer tout le JSON pour le réécrire entièrement (et parfois en le *détoastant*/*retoastant*).

Un gros point noir est l'absence de statistiques propres aux clés du JSON. Le planificateur va avoir beaucoup de mal à estimer les cardinalités des critères.

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

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

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

### 1.3.8 jsonb : indexation (1/2)

- Index fonctionnel sur un champ précis
  - bonus : statistiques

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

- Champ dénormalisé:
  - champ normal, indexable, facile à utiliser
  - statistiques

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

[4] https://dali.bo/m4_html#mécanisme-toast

**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 champ fréquemment utilisé pour des recherches, c'est le plus efficace.

On n'oubliera pas de lancer un `ANALYZE` pour calculer les statistiques après création de l'index fonctionnel. Même si l'index est peu discriminant, on obtient ainsi de bonnes statistiques sur son critère. Un `VACUUM` permet aussi les *Index Only Scan* quand c'est possible.

**Champ dénormalisé** :

Une autre possibilité est de dénormaliser le champ JSON intéressant dans un champ séparé de la table, géré automatiquement, et indexable :

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

ANALYZE personnes ;

CREATE INDEX ON personnes (lastname) ;
```

Ce champ coûte un peu d'espace disque supplémentaire, mais il peut être intéressant pour la lisibilité du code, la facilité d'utilisation avec certains outils ou pour certains utilisateurs. Dans le cas des gros JSON, il peut aussi éviter quelques allers-retours vers la table TOAST.

### 1.3.9 jsonb : indexation (2/2)

- Indexation « schemaless » grâce au GIN :

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

**Index GIN** :

Les champs `jsonb` peuvent tirer parti de fonctionnalités avancées de PostgreSQL, notamment les index GIN, et ce via deux classes d'opérateurs.

Même si l'opérateur par défaut GIN pour `jsonb` supporte plus d'opérations, il est souvent suffisant, et surtout plus efficace, de choisir l'opérateur `jsonb_path_ops` (voir les détails[5]) :

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

`jsonb_path_ops` supporte notamment l'opérateur « contient » ( `@>` ) :

```
# EXPLAIN (ANALYZE) SELECT datas->>'firstName' FROM personnes
  WHERE datas @> '{"lastName": "Dupont"}'::jsonb ;
```

```
                            QUERY PLAN
------------------------------------------------------------------
Bitmap Heap Scan on personnes  (cost=2.01..3.02 rows=1 width=32)
                      (actual time=0.018..0.019 rows=1 loops=1)
  Recheck Cond: (datas @> '{"lastName": "Dupont"}'::jsonb)
  Heap Blocks: exact=1
  ->  Bitmap Index Scan on idx_prs  (cost=0.00..2.01 rows=1 width=0)
                           (actual time=0.010..0.010 rows=1 loops=1)
        Index Cond: (datas @> '{"lastName": "Dupont"}'::jsonb)
Planning Time: 0.052 ms
Execution Time: 0.104 ms
```

Ce type d'index est moins efficace qu'un index fonctionnel B-tree classique, mais il est idéal quand la clé de recherche n'est pas connue, et que n'importe quel champ du JSON est un critère. De plus il est compressé.

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

Surtout, un index GIN ne permet pas de recherches sur des opérateurs B-tree classiques ( `<` , `<=` , `>` , `>=` ), ou sur le contenu de tableaux. On est obligé pour cela de revenir au monde relationnel, ou se rabattre sur les index fonctionnels vus plus haut. Il est donc préférable d'utiliser les opérateurs spécifiques, comme « contient » ( `@>` ).

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

### 1.3.10 SQL/JSON & JSONpath

- SQL:2016 introduit SQL/JSON et le langage JSON Path
- JSON Path :
  - langage de recherche pour JSON
  - concis, flexible, plus rapide
  - inclus dans PostreSQL 12 pour l'essentiel
  - exemple :

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

JSON path facilite la recherche et le parcours dans les documents JSON complexes. Il évite de parcourir manuellement les nœuds du JSON.

Par exemple, une recherche peut se faire ainsi :

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

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

Mais l'intérêt est d'extraire facilement des parties d'un tableau :

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

               jsonb_path_query
-----------------------------------------------
 {"type": "bureau", "number": "07 89 10 11 12"}
 {"type": "bureau", "number": "07 98 10 11 13"}
```

On trouvera d'autres exemples dans la présentation de Postgres Pro dédié à la fonctionnalité lors la parution de PostgreSQL 12[6], ou dans un billet de Michael Paquier[7].

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

## 1.4 XML

- Type `xml`
  - stocke un document XML
  - valide sa structure
- Quelques fonctions et opérateurs disponibles :
  - `XMLPARSE`, `XMLSERIALIZE`, `query_to_xml`, `xmlagg`
  - `xpath` (XPath 1.0 uniquement)

Le type `xml`, inclus de base, vérifie que le XML inséré est un document « bien formé », ou constitue des fragments de contenu (« content »). L'encodage UTF-8 est impératif. Il y a quelques limitations par rapport aux dernières versions du standard, XPath et XQuery[8]. 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[9] ou celle des fonctions[10]. 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>
```

[8] https://docs.postgresql.fr/current/xml-limits-conformance.html
[9] https://docs.postgresql.fr/current/datatype-xml.html
[10] https://docs.postgresql.fr/current/functions-xml.html

```
    <TITLE>Olympia 1961</TITLE>
    <ARTIST>Jacques Brel</ARTIST>
    <COUNTRY>France</COUNTRY>
    <YEAR>1962</YEAR>
  </CD>
</CATALOG> $$ ) ;
--- Noter le $$ pour délimiter une chaîne contenant une apostrophe

SELECT XMLSERIALIZE (DOCUMENT catalogue AS text) FROM liste_cd;

                   xmlserialize
-------------------------------------------------
 <?xml version="1.0" encoding="UTF-8"?>          +
 <CATALOG>                                       +
   <CD>                                          +
     <TITLE>The Times They Are a-Changin'</TITLE>+
     <ARTIST>Bob Dylan</ARTIST>                  +
     <COUNTRY>USA</COUNTRY>                      +
     <YEAR>1964</YEAR>                           +
   </CD>                                         +
   <CD>                                          +
     <TITLE>Olympia 1961</TITLE>                 +
     <ARTIST>Jacques Brel</ARTIST>               +
     <COUNTRY>France</COUNTRY>                   +
     <YEAR>1962</YEAR>                           +
   </CD>                                         +
 </CATALOG>
(1 ligne)
```

Il existe aussi `query_to_xml` pour convertir un résultat de requête en XML, `xmlagg` pour agréger des champs XML, ou `xpath` pour extraire des nœuds suivant une expression XPath 1.0.

NB : l'extension xml2[11] est dépréciée et ne doit pas être utilisée dans les nouveaux projets.

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

## 1.5 OBJETS BINAIRES

- Souvent une mauvaise idée…
- 2 méthodes
  - `bytea` : type binaire
  - *Large Objects* : manipulation comme un fichier

PostgreSQL permet de stocker des données au format binaire, potentiellement de n'importe quel type, par exemple des images ou des PDF.

Il faut vraiment se demander si des binaires ont leur place dans une base de données relationnelle. Ils sont généralement beaucoup plus gros que les données classiques. La volumétrie peut donc devenir énorme, et encore plus si les binaires sont modifiés, car le mode de fonctionnement de PostgreSQL aura tendance à les dupliquer. Cela aura un impact sur la fragmentation, la quantité de journaux, la taille des sauvegardes, et toutes les opérations de maintenance. Ce qui est intéressant à conserver dans une base sont des données qu'il faudra rechercher, et l'on recherche rarement au sein d'un gros binaire. En général, l'essentiel des données binaires que l'on voudrait confier à une base peut se contenter d'un stockage classique, PostgreSQL ne contenant qu'un chemin ou une URL vers le fichier réel.

PostgreSQL donne le choix entre deux méthodes pour gérer les données binaires :

- `bytea` : un type comme un autre ;
- *Large Object* : des objets séparés, à gérer indépendamment des tables.

### 1.5.1 bytea

- Un type comme les autres
  - `bytea` : tableau d'octets
  - en texte : `bytea_output` = `hex` ou `escape`
- Récupération intégralement en mémoire !
- Toute modification entraîne la réécriture complète du `bytea`
- Maxi 1 Go (à éviter)
  - en pratique intéressant pour quelques Mo
- Import :

```
SELECT pg_read_binary_file ('/chemin/fichier');
```

Voici un exemple :

```
CREATE TABLE demo_bytea(a bytea);
INSERT INTO demo_bytea VALUES ('bonjour'::bytea);

SELECT * FROM demo_bytea ;

        a
------------------
 \x626f6e6a6f7572
```

Nous avons inséré la chaîne de caractère « bonjour » dans le champ bytea, en fait sa représentation binaire dans l'encodage courant (UTF-8). Si nous interrogeons la table, nous voyons la représentation textuelle du champ bytea. Elle commence par `\x` pour indiquer un encodage de type `hex`. Ensuite, chaque paire de valeurs hexadécimales représente un octet.

Un second format d'affichage est disponible : `escape` :

```
SET bytea_output = escape ;
SELECT * FROM demo_bytea ;

    a
---------
 bonjour

INSERT INTO demo_bytea VALUES ('journée'::bytea);
SELECT * FROM demo_bytea ;

       a
----------------
 bonjour
 journ\303\251e
```

Le format de sortie `escape` ne protège donc que les valeurs qui ne sont pas représentables en ASCII 7 bits. Ce format peut être plus compact pour des données textuelles essentiellement en alphabet latin sans accent, où le plus gros des caractères n'aura pas besoin d'être protégé.

Cependant, le format `hex` est bien plus efficace à convertir, et est le défaut depuis PostgreSQL 9.0.

Avec les vieilles applications, ou celles restées avec cette configuration, il faudra peut-être forcer `bytea_output` à `escape`, sous peine de corruption.)

Pour charger directement un fichier, on peut notamment utiliser la fonction `pg_read_binary_file`, exécutée par le serveur PostreSQL :

```
INSERT INTO demo_bytea (a)
SELECT pg_read_binary_file ('/chemin/fichier');
```

En théorie, un `bytea` peut contenir 1 Go. En pratique, on se limitera à nettement moins, ne serait-ce que parce `pg_dump` tombe en erreur quand il doit exporter des bytea de plus de 500 Mo environ (le décodage double le nombre d'octets et dépasse cette limite de 1 Go).

La documentation officielle[12] liste les fonctions pour encoder, décoder, extraire, hacher… les `bytea`.

[12] https://docs.postgresql.fr/current/functions-binarystring.html

### 1.5.2 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[13]) ;
- de ne modifier que cette partie sans tout réécrire.

Cependant, nous déconseillons son utilisation autant que possible :

[13] https://jdbc.postgresql.org/documentation/binary-data/

- le stockage des *large objects* ne prévoit pas vraiment de compression : des `bytea` prendront moins de place (penser à changer `default_toast_compression` à `lz4` sur les versions 14 et supérieures) ;
- 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`[14]) ;
- `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[15], si les *large objects* ne sont jamais référencés plus d'une fois ;
- appeler régulièrement le programme `vacuumlo` (là encore un contrib[16]) : 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[17] pour les détails.

[14] https://docs.postgresql.fr/current/app-pgdump.html
[15] https://docs.postgresql.fr/current/lo.html
[16] https://docs.postgresql.fr/current/vacuumlo.html
[17] https://docs.postgresql.fr/current/largeobjects.html

## 1.6 QUIZ

**https://dali.bo/s9_quiz**

## 1.7 TRAVAUX PRATIQUES

Les TP sur les types hstore et JSON utilisent la base **cave**. La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

### 1.7.1 Hstore (Optionnel)

**But** : Découvrir hstore

Pour ce TP, il est fortement conseillé d'aller regarder la documentation officielle du type `hstore` sur https://docs.postgresql.fr/current/hstore.html.

**But** : Obtenir une version dénormalisée de la table `stock` : elle contiendra une colonne de type `hstore` contenant l'année, l'appellation, la région, le récoltant, le type, et le contenant :

```
vin_id      integer
nombre      integer
attributs   hstore
```

Ce genre de table n'est évidemment pas destiné à une application transactionnelle: on n'aurait aucun moyen de garantir l'intégrité des données de cette colonne. Cette colonne va nous permettre d'écrire une recherche multi-critères efficace sur nos stocks.

Afficher les informations à stocker avec la requête suivante :

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
LIMIT 10;
```

(`LIMIT 10` est là juste pour éviter de ramener tous les enregistrements).

> Créer une table `stock_denorm (vin_id int, nombre  int, attributs hstore)` et y copier le contenu de la requête. Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de `hstore` sans se soucier de formatage de chaîne de caractères. (Voir la documentation.)

> Créer un index sur le champ `attributs` pour accélérer les recherches.

> Rechercher le nombre de bouteilles (attribut `bouteille`) en stock de vin blanc (attribut `type_vin`) d'Alsace (attribut `region`). Quel est le temps d'exécution de la requête ? Combien de buffers accédés ?

> Refaire la même requête sur la table initiale. Qu'en conclure ?

### 1.7.2 jsonb

**But** : Découvrir JSON

Nous allons créer une table dénormalisée contenant uniquement un champ JSON.

Pour chaque vin, le document JSON aura la structure suivante :

```
{
  vin: {
      recoltant: {
        nom: text,
        adressse: text
      },
      appellation: {
        libelle: text,
        region: text
      },
      type_vin: text
  },
  stocks: [{
    contenant: {
      contenance: real,
      libelle: text
    },
    annee: integer,
    nombre: integer
  }]
}
```

Pour écrire une requête permettant de générer ces documents, nous allons procéder par étapes.

> La requête suivante permet de générer les parties `vin` du document, avec `recoltant` et `appellation`. Créer un document JSON pour chaque ligne de `vin` grâce à la fonction `jsonb_build_object`.

```
SELECT
  recoltant.nom,
  recoltant.adresse,
  appellation.libelle,
  region.libelle,
  type_vin.libelle
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;
```

> Écrire une requête permettant de générer la partie `stocks` du document, soit un document JSON pour chaque ligne de la table `stock`, incluant le `contenant`.

> Fusionner les requêtes précédentes pour générer un document complet pour chaque ligne de la table `vin`. Créer une table `stock_jsonb` avec un unique champ JSONB rassemblant ces documents.

> Calculer la taille de la table, et la comparer à la taille du reste de la base.

> Depuis cette nouvelle table, renvoyer l'ensemble des récoltants de la région Beaujolais.

> Renvoyer l'ensemble des vins pour lesquels au moins une bouteille entre 1992 et 1995 existe. (la difficulté est d'extraire les différents stocks d'une même ligne de la table)

> Indexer le document jsonb en utilisant un index de type GIN.

> Peut-on réécrire les deux requêtes précédentes pour utiliser l'index ?

### 1.7.3 Large Objects

> **But** : Utilisation de Large Objects

- Créer une table `fichiers` avec un texte et une colonne permettant de référencer des *Large Objects*.

- Importer un fichier local à l'aide de psql dans un large object.
- Noter l'`oid` retourné.

- Importer un fichier du serveur à l'aide de psql dans un large object.

- Afficher le contenu de ces différents fichiers à l'aide de psql.

- Les sauvegarder dans des fichiers locaux.

## 1.8 TRAVAUX PRATIQUES (SOLUTIONS)

### 1.8.1 Hstore (Optionnel)

> Afficher les informations à stocker avec la requête suivante :

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
LIMIT 10;
```

(`LIMIT 10` est là juste pour éviter de ramener tous les enregistrements).

> Créer une table `stock_denorm (vin_id int, nombre  int, attributs hstore)` et y copier le contenu de la requête. Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de `hstore` sans se soucier de formatage de chaîne de caractères. (Voir la documentation.)

Une remarque toutefois : les éléments du tableau doivent tous être de même type, d'où la conversion en text des quelques éléments entiers. C'est aussi une limitation du type `hstore` : il ne supporte que les attributs texte.

Cela donne :

```
CREATE EXTENSION hstore;

CREATE TABLE stock_denorm AS
SELECT stock.vin_id,
       stock.nombre,
       hstore(ARRAY['annee', stock.annee::text,
                    'recoltant', recoltant.nom,
                    'appellation',appellation.libelle,
                    'region',region.libelle,
                    'type_vin',type_vin.libelle,
                    'contenance',contenant.contenance::text,
                    'contenant',contenant.libelle]) AS attributs
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
```

```
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id);
```

Et l'on n'oublie pas les statistiques :

```
ANALYZE stock_denorm;
```

> Créer un index sur le champ `attributs` pour accélérer les recherches.

```
CREATE INDEX idx_stock_denorm on stock_denorm USING gin (attributs );
```

> Rechercher le nombre de bouteilles (attribut `bouteille`) en stock de vin blanc (attribut `type_vin`) d'Alsace (attribut `region`). Quel est le temps d'exécution de la requête ? Combien de buffers accédés ?

Attention au A majuscule de Alsace, les hstore sont sensibles à la casse !

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT *
FROM stock_denorm
WHERE attributs @>
'type_vin=>blanc, region=>Alsace, contenant=>bouteille';
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on stock_denorm (cost=64.70..374.93 rows=91 width=193)
                                  (actual time=64.370..68.526 rows=1680 loops=1)
   Recheck Cond: (attributs @> '"region"=>"Alsace", "type_vin"=>"blanc",
                               "contenant"=>"bouteille"'::hstore)
   Heap Blocks: exact=1256
   Buffers: shared hit=1353
   ->  Bitmap Index Scan on idx_stock_denorm
            (cost=0.00..64.68 rows=91 width=0)
            (actual time=63.912..63.912 rows=1680 loops=1)
         Index Cond: (attributs @> '"region"=>"Alsace", "type_vin"=>"blanc",
                                   "contenant"=>"bouteille"'::hstore)
         Buffers: shared hit=97
 Planning time: 0.210 ms
 Execution time: 68.927 ms
```

> Refaire la même requête sur la table initiale. Qu'en conclure ?

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
```

```
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
WHERE type_vin.libelle='blanc' AND region.libelle='Alsace'
AND contenant.libelle = 'bouteille';

                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=11.64..873.33 rows=531 width=75)
              (actual time=0.416..24.779 rows=1680 loops=1)
   Join Filter: (stock.contenant_id = contenant.id)
   Rows Removed by Join Filter: 3360
   Buffers: shared hit=6292
   ->  Seq Scan on contenant  (cost=0.00..1.04 rows=1 width=16)
                              (actual time=0.014..0.018 rows=1 loops=1)
         Filter: (libelle = 'bouteille'::text)
         Rows Removed by Filter: 2
         Buffers: shared hit=1
   ->  Nested Loop  (cost=11.64..852.38 rows=1593 width=67)
                    (actual time=0.392..22.162 rows=5040 loops=1)
         Buffers: shared hit=6291
         ->  Hash Join  (cost=11.23..138.40 rows=106 width=55)
                        (actual time=0.366..5.717 rows=336 loops=1)
               Hash Cond: (vin.recoltant_id = recoltant.id)
               Buffers: shared hit=43
               ->  Hash Join  (cost=10.07..135.78 rows=106 width=40)
                              (actual time=0.337..5.289 rows=336 loops=1)
                     Hash Cond: (vin.type_vin_id = type_vin.id)
                     Buffers: shared hit=42
                     ->  Hash Join  (cost=9.02..132.48 rows=319 width=39)
                                    (actual time=0.322..4.714 rows=1006 loops=1)
                           Hash Cond: (vin.appellation_id = appellation.id)
                           Buffers: shared hit=41
                           ->  Seq Scan on vin
                                       (cost=0.00..97.53 rows=6053 width=16)
                                       (actual time=0.011..1.384 rows=6053 loops=1)
                                 Buffers: shared hit=37
                           ->  Hash  (cost=8.81..8.81 rows=17 width=31)
                                     (actual time=0.299..0.299 rows=53 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 4kB
                                 Buffers: shared hit=4
                                 ->  Hash Join
                                      (cost=1.25..8.81 rows=17 width=31)
                                      (actual time=0.033..0.257 rows=53 loops=1)
                                       Hash Cond:
                                            (appellation.region_id = region.id)
                                       Buffers: shared hit=4
                                       ->  Seq Scan on appellation
                                            (cost=0.00..6.19 rows=319 width=24)
                                            (actual time=0.010..0.074 rows=319
                                             loops=1)
```

```
                                        Buffers: shared hit=3
                                  ->  Hash
                                        (cost=1.24..1.24 rows=1 width=15)
                                        (actual time=0.013..0.013 rows=1
                                         loops=1)
                                        Buckets: 1024  Batches: 1
                                                 Memory Usage: 1kB
                                        Buffers: shared hit=1
                                        ->  Seq Scan on region
                                          (cost=0.00..1.24 rows=1 width=15)
                                          (actual time=0.005..0.012 rows=1
                                                  loops=1)
                                              Filter: (libelle =
                                                       'Alsace'::text)
                                              Rows Removed by Filter: 18
                                              Buffers: shared hit=1
                     ->  Hash  (cost=1.04..1.04 rows=1 width=9)
                               (actual time=0.008..0.008 rows=1 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 1kB
                           Buffers: shared hit=1
                           ->  Seq Scan on type_vin
                                    (cost=0.00..1.04 rows=1 width=9)
                                    (actual time=0.005..0.007 rows=1 loops=1)
                                 Filter: (libelle = 'blanc'::text)
                                 Rows Removed by Filter: 2
                                 Buffers: shared hit=1
               ->  Hash  (cost=1.07..1.07 rows=7 width=23)
                         (actual time=0.017..0.017 rows=7 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 1kB
                     Buffers: shared hit=1
                     ->  Seq Scan on recoltant
                               (cost=0.00..1.07 rows=7 width=23)
                               (actual time=0.004..0.009 rows=7 loops=1)
                           Buffers: shared hit=1
          ->  Index Scan using idx_stock_vin_annee on stock
                        (cost=0.42..6.59 rows=15 width=16)
                        (actual time=0.013..0.038 rows=15 loops=336)
                Index Cond: (vin_id = vin.id)
                Buffers: shared hit=6248
 Planning time: 4.341 ms
 Execution time: 25.232 ms
(53 lignes)
```

La requête sur le schéma normalisé est ici plus rapide. On constate tout de même qu'elle accède à 6300 buffers, contre 1300 à la requête dénormalisée, soit 4 fois plus de données. Un test identique exécuté sur des données hors du cache donne environ 80 ms pour la requête sur la table dénormalisée, contre près d'une seconde pour les tables normalisées. Ce genre de transformation est très utile lorsque le schéma ne se prête pas à une normalisation, et lorsque le volume de données à manipuler est trop important pour tenir en mémoire. Les tables dénormalisées avec hstore se prêtent aussi bien mieux aux recherches multi-critères.

### 1.8.2 jsonb

Pour chaque vin, le document JSON aura la structure suivante :

```
{
  vin: {
      recoltant: {
        nom: text,
        adressse: text
      },
      appellation: {
        libelle: text,
        region: text
      },
      type_vin: text
  },
  stocks: [{
    contenant: {
      contenance: real,
      libelle: text
    },
    annee: integer,
    nombre: integer
  }]
}
```

> La requête suivante permet de générer les parties `vin` du document, avec `recoltant` et `appellation`. Créer un document JSON pour chaque ligne de `vin` grâce à la fonction `jsonb_build_object`.

```
SELECT
  recoltant.nom,
  recoltant.adresse,
  appellation.libelle,
  region.libelle,
  type_vin.libelle
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;
```

```
SELECT
  jsonb_build_object(
    'recoltant',
    json_build_object('nom', recoltant.nom, 'adresse',
                      recoltant.adresse
    ),
    'appellation',
    jsonb_build_object('libelle', appellation.libelle, 'region', region.libelle),
    'type_vin', type_vin.libelle
)
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
```

```
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id ;
```

> Écrire une requête permettant de générer la partie `stocks` du document, soit un document JSON pour chaque ligne de la table `stock`, incluant le `contenant`.

La partie `stocks` du document est un peu plus compliquée, et nécessite l'utilisation de fonctions d'agrégations.

```
SELECT json_build_object(
  'contenant',
  json_build_object('contenance', contenant.contenance, 'libelle',
                    contenant.libelle),
  'annee', stock.annee,
  'nombre', stock.nombre)
FROM stock join contenant on stock.contenant_id = contenant.id;
```

Pour un vin donné, le tableau `stock` ressemble à cela :

```
SELECT json_agg(json_build_object(
  'contenant',
  json_build_object('contenance', contenant.contenance, 'libelle',
                    contenant.libelle),
  'annee', stock.annee,
  'nombre', stock.nombre))
FROM stock
INNER JOIN contenant on stock.contenant_id = contenant.id
WHERE vin_id = 1
GROUP BY vin_id;
```

> Fusionner les requêtes précédentes pour générer un document complet pour chaque ligne de la table `vin`. Créer une table `stock_jsonb` avec un unique champ JSONB rassemblant ces documents.

On assemble les deux parties précédentes :

```
CREATE TABLE stock_jsonb  AS (
  SELECT
    json_build_object(
    'vin',
      json_build_object(
        'recoltant',
        json_build_object('nom', recoltant.nom, 'adresse', recoltant.adresse),
        'appellation',
        json_build_object('libelle', appellation.libelle, 'region',
                          region.libelle),
        'type_vin', type_vin.libelle),
    'stocks',
      json_agg(json_build_object(
        'contenant',
        json_build_object('contenance', contenant.contenance, 'libelle',
```

```
                          contenant.libelle),
        'annee', stock.annee,
        'nombre', stock.nombre)))::jsonb as document
  FROM vin
  INNER JOIN recoltant on vin.recoltant_id = recoltant.id
  INNER JOIN appellation on vin.appellation_id = appellation.id
  INNER JOIN region on region.id = appellation.region_id
  INNER JOIN type_vin on vin.type_vin_id = type_vin.id
  INNER JOIN stock on stock.vin_id = vin.id
  INNER JOIN contenant on stock.contenant_id = contenant.id
  GROUP BY vin_id, recoltant.id, region.id, appellation.id, type_vin.id
);
```

> Calculer la taille de la table, et la comparer à la taille du reste de la base.

La table avec JSON contient toutes les mêmes informations que l'ensemble des tables normalisées de la base cave (à l'exception des `id`). Elle occupe en revanche une place beaucoup moins importante, puisque les documents individuels vont pouvoir être compressés en utilisant le mécanisme TOAST. De plus, on économise les 26 octets par ligne de toutes les autres tables.

Elle est même plus petite que la seule table `stock` :

```
 \d+
                                      Liste des relations
 Schéma |        Nom          |   Type   | Propriétaire | Persistence | Taille | …
--------+---------------------+----------+--------------+-------------+--------+-
…
 public | stock               | table    | caviste      | permanent   | 36 MB  |
 public | stock_jsonb         | table    | postgres     | permanent   | 12 MB  |
…
```

> Depuis cette nouvelle table, renvoyer l'ensemble des récoltants de la région Beaujolais.

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document #>> '{vin, appellation, region}' = 'Beaujolais';
```

> Renvoyer l'ensemble des vins pour lesquels au moins une bouteille entre 1992 et 1995 existe. (la difficulté est d'extraire les différents stocks d'une même ligne de la table)

La fonction `jsonb_array_elements` permet de convertir les différents éléments du document `stocks` en autant de lignes. La clause `LATERAL` permet de l'appeler une fois pour chaque ligne :

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb,
LATERAL jsonb_array_elements(document #> '{stocks}') as stock
WHERE (stock->'annee')::text::integer BETWEEN 1992 AND 1995;
```

> Indexer le document jsonb en utilisant un index de type GIN.

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

> Peut-on réécrire les deux requêtes précédentes pour utiliser l'index ?

Pour la première requête, on peut utiliser l'opérateur « contient » pour passer par l'index :

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document @> '{"vin": {"appellation": {"region": "Beaujolais"}}}';
```

La seconde ne peut malheureusement pas être réécrite pour tirer partie de l'index.

La dénormalisation vers un champ externe n'est pas vraiment possible, puisqu'il y a plusieurs `stocks` par ligne.

### 1.8.3 Large Objects

> – Créer une table `fichiers` avec un texte et une colonne permettant de référencer des *Large Objects*.

```
CREATE TABLE fichiers (nom text PRIMARY KEY, data OID);
```

> – Importer un fichier local à l'aide de psql dans un large object.
> – Noter l'`oid` retourné.

```
psql -c "\lo_import '/etc/passwd'"
```

```
lo_import 6821285
```

```
INSERT INTO fichiers VALUES ('/etc/passwd',6821285) ;
```

> – Importer un fichier du serveur à l'aide de psql dans un large object.

```
INSERT INTO fichiers SELECT 'postgresql.conf',
lo_import('/var/lib/pgsql/15/data/postgresql.conf') ;
```

> – Afficher le contenu de ces différents fichiers à l'aide de psql.

```
psql -c "SELECT nom,encode(l.data,'escape') \
         FROM fichiers f JOIN pg_largeobject l ON f.data = l.loid;"
```

> – Les sauvegarder dans des fichiers locaux.

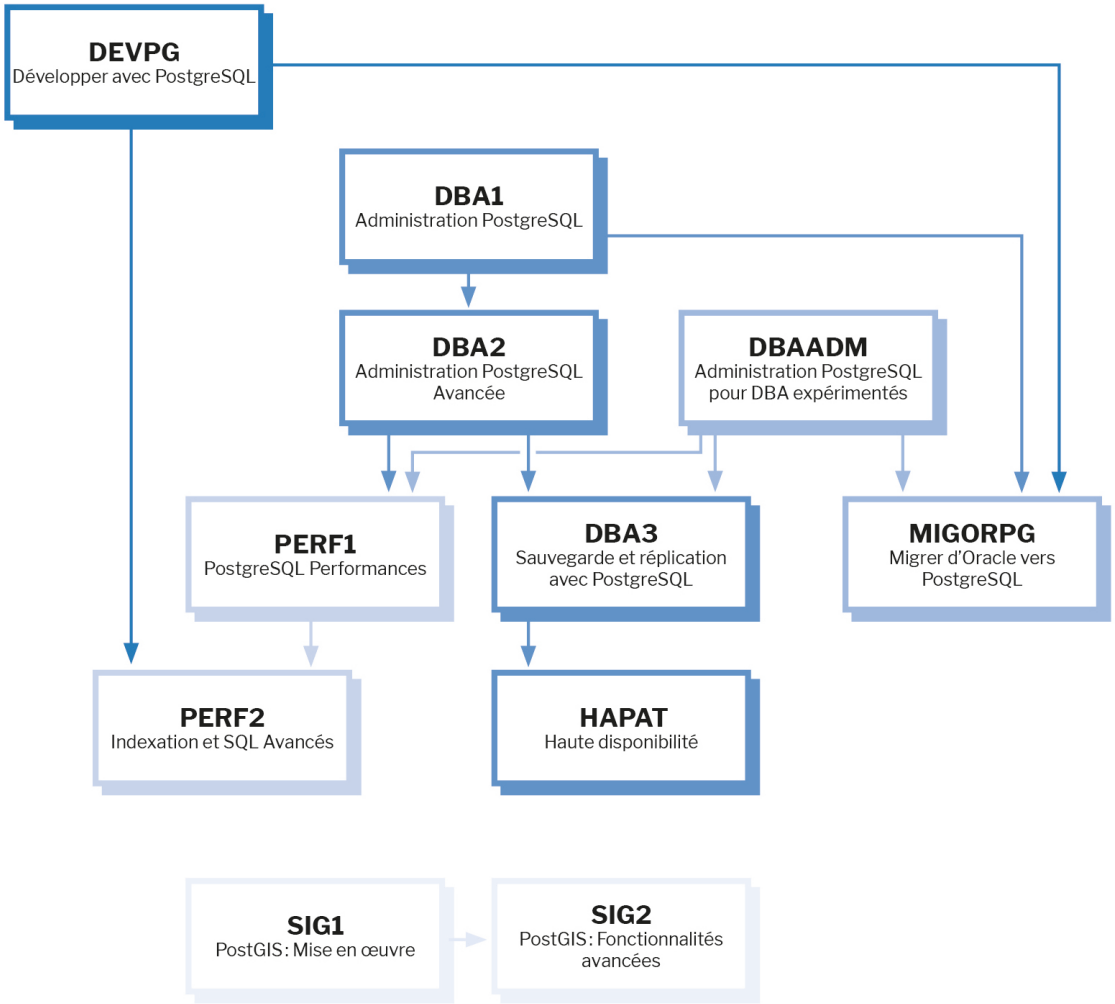
```
psql -c "\lo_export loid_retourné '/home/dalibo/passwd_serveur';"
```

# Les formations Dalibo

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

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

## Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

## Les livres blancs

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

## Téléchargement gratuit

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