Livre Blanc DALIBO #04

Bonnes pratiques de modélisation avec PostgreSQL

18.10

DALIBO
L'expertise PostgreSQL

Dalibo SCOP

https ://wwww.dalibo.com/

Bonnes pratiques de modélisation avec PostgreSQL

Livre Blanc DALIBO #04

TITRE : Bonnes pratiques de modélisation avec PostgreSQL
SOUS-TITRE : Livre Blanc DALIBO #04

REVISION : 18.10

Remerciements : Ce livre blanc est le fruit d’un travail collectif. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Carole Arnaud, Damien Clochard, Léo Cossic, Adrien Nayrat, Thomas Reiss, Maël Rimbault. Nous remercions également la société DSIA qui a contribué au financement de ce livre blanc.

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 livres blancs sur https ://dalibo.com/

Chers lectrices & lecteurs,

Nos livres blancs sont issus de plus de 12 ans 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 livres blancs est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'ils 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 productions 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 contact@dalibo.com !

Table des Matières

Exploiter toute la puissance de PostgreSQL **9**

Modélisation **10**
- Normalisation et atomicité des attributs . . . 10
- Nommage des objets . . . 14
- Bases de données et schémas . . . 15
- Typage des données . . . 16
  - Types standards . . . 16
  - Types avancées spécifiques à PostgreSQL . . . 20
- Contraintes d'intégrité . . . 28
  - Clé primaire . . . 28
  - Clé étrangère . . . 29
  - Unicité . . . 30
  - Contraintes CHECK et NOT NULL . . . 31
  - Contraintes d'exclusion . . . 32
- DDL transactionnel . . . 32
- Droits sur les objets . . . 33
  - Organisation des rôles et des droits . . . 33
  - Sécurisation des fonctions . . . 34
  - Sécuriser une installation par défaut . . . 35
- Indexation . . . 35
  - Indexation des clés étrangères . . . 35
  - Indexation pour les requêtes . . . 36
- Dénormaliser . . . 39
- Pièges . . . 40
  - NULL . . . 40
  - Gestion des objets binaires . . . 40
  - Colonne « fourre-tout » . . . 41

Bibliographie **49**

EXPLOITER TOUTE LA PUISSANCE DE POSTGRESQL

PostgreSQL est un moteur de base de données à la fois robuste et puissant. Depuis plus de 20 ans, il est porté par une communauté internationale, décentralisée et très dynamique. Chaque année, une nouvelle version majeure est publiée avec de nouvelles fonctionnalités, ce qui en fait le SGBD le plus innovant du secteur.

Cette richesse et ce dynamisme font qu'il est parfois difficile de suivre les dernières avancées ou simplement de connaitre toutes les possibilités offertes par le moteur.

Le but premier de ce guide est de faire un tour d'horizon des bonnes pratiques et des conventions pour modéliser des applications qui tirent le meilleur de PostgreSQL. Il décrit comment définir une bonne structuration des données : définir le schéma, choisir les bons types, créer des index utiles, protéger les données, etc.

Ce manuel est construit comme un boite à outils dans laquelle le lecteur pourra piocher des conseils et des idées en fonction du contexte et des constraintes de chaque projet.

MODÉLISATION

NORMALISATION ET ATOMICITÉ DES ATTRIBUTS

Commençons par le … commencement. Avant de créer quelque table que ce soit, il faut avoir les idées claires sur les données à gérer. La formalisation d'un Modèle Conceptuel des Données (MCD) reste un exercice utile.

La première forme normale spécifie qu'un attribut doit être **atomique**. C'est-à-dire qu'il ne doit pas être composé. Prenons un exemple :

Immatriculation	Modèle	Caractéristiques
TT-802-AX	Clio	4 roues motrices, ABS, toit ouvrant, peinture verte
QS-123-DB	AX	jantes aluminium, peinture bleu

Cette modélisation viole la première forme normale (atomicité des attributs). Si on recherche toutes les voitures vertes, on va devoir utiliser une clause `WHERE` de ce type :

```
WHERE caracteristiques LIKE '%peinture verte%'
```

Ce qui sera évidemment très inefficace.

Par ailleurs, on n'a évidemment aucun contrôle sur ce qui est mis dans le champs `caractéristiques`, ce qui est la garantie de données incohérentes au bout de quelques jours (heures ?) d'utilisation.

Il aurait certainement fallu modéliser cela en rajoutant des colonnes `boolean quatre_roues_motrices`, `boolean abs`, `varchar couleur` (ou même, encore mieux, une table des couleurs, ou un enum de la liste des couleurs autorisées, etc.)

```
INSERT INTO voiture
VALUES ('AD-057-GD','Clio','interieur bleu, anti-blocage des roues');
```

Dans ce cas, rien n'empêche d'ajouter une ligne avec des caractéristiques similaires mais définies autrement :

- ABS / antiblocage des roues
- Le moteur retournera le véhicule "AD-057-GD si on veut rechercher un véhicule de couleur"bleu"

Il apparaît donc clairement que ce modèle ne permet pas d'assurer la cohérence des données. En revanche, le modèle de données suivant offrirait des performances de recherche et une cohérence des données nettement améliorées :

Column	Type	Description
immatriculation	text	
modele	text	
couleur	color	Couleur vehicule (bleu,rouge,vert)
toit_ouvrant	boolean	Option toit ouvrant
abs	boolean	Option anti-blocage des roues
type_roue	boolean	tole/aluminium
motricite	boolean	2 roues motrices / 4 roues motrices

Exemples de formes normales

Dans les exemples ci-dessous, la clé est représentée en **gras** les colonnes.

Prenons pour exemple cette très mauvaise table issue de la gestion de stock d'un libraire :

stocks
titre
edition
année
quantité
nom_auteur
prenom_auteur
adresse_auteur
ville_auteur
pays_auteur

Cette table est bien en première forme normale, tous les attributs sont bien atomiques.

En revanche, à l'usage, on se rend compte que si on retire un livre du stock, on peut perdre beaucoup d'information.

On décide donc de diviser la table en deux entités, d'abord l'entité `stocks` :

stocks		
titre	**edition**	quantité

Et l'entité `livres` :

livres
titre
edition
année
nom_auteur
prenom_auteur
adresse_auteur
ville_auteur
pays_auteur

Maintenant on remarque qu'on ne peut pas ajouter un auteur si on n'a pas de livre associé et inversement (on ne peut pas supprimer un livre sans supprimer son auteur).

En réalité, cet exemple est l'illustration d'une violation de la deuxième forme normale : Les attributs `nom_auteur` et `prenom_auteur` dépendent d'une partie de la clé. En effet, en obtenant le titre du livre, on peut déterminer le nom et prénom de l'auteur.

À nouveau, nous encore divisons la table, cette fois en trois entité. D'abord l'entité `stocks` :

stocks		
titre	**edition**	quantité

Une entité `livres` :

livres				
titre	**edition**	année	nom_auteur	prenom_auteur

Et enfin une entité `auteurs` :

auteurs				
nom	**prenom**	adresse	ville	pays

Plus aucun attribut non-clé ne dépend d'une partie de la clé. Notre modèle de données respecte maintenant la deuxième forme normale.

En revanche, on remarque que si on connaît la ville, on peut déterminer le pays. Cela

signifie qu'un attribut n'appartenant pas à la clé détermine un autre attribut non-clé. C'est une violation de la troisième forme normale. Le fait d'avoir la ville et le pays dans la même table entraîne beaucoup de données dupliquées. Si on devait modifier le pays d'une ville, cela impliquerait de modifier toutes les lignes.

À nouveau divisons une dernière fois la table. Nous aurons l'entité `stocks` :

stocks		
titre	**edition**	quantité

L'entité `livres` :

livres				
titre	**edition**	année	nom_auteur	prenom_auteur

L'entité `auteurs` :

auteurs			
nom	**prenom**	adresse	ville

Et enfin l'entité `villes` :

villes	
ville	pays

La table est bien en troisième forme normale (parfois notée 3NF).

Il faut donc commencer par structurer les données en troisième forme normale, ou mieux, sous la forme normale de Boyce-Codd ou FNBC. Un petit rafraîchissement des idées sur la théorie relationnelle peut être salutaire, même si, somme toute, il s'agit de beaucoup de bon sens. Avec PostgreSQL comme avec tout SGBD relationnel, cet effort initial est toujours récompensé sur le long terme.

Il existe une définition mathématique précise de chacune des 7 formes normales. Cependant, on pourra noter que la troisième forme normale peut toujours être atteinte. La forme suivante (forme normale de Boyce-Codd, ou FNBC) ne peut pas toujours être atteinte. Pour reprendre une définition simple par Chris Date : une relation (table) est en

troisième forme normale si tous les attributs (colonnes) dépendent de la clé (primaire), de toute la clé (pas d'un sous-ensemble de ses colonnes), et de rien d'autre que de la clé (une colonne supplémentaire) ; « Chaque attribut dépend de la clé, de TOUTE la clé, et QUE de la clé » ou « The key, the whole key, nothing but the key » dans sa version originale.

> Si vos tables vérifient déjà ces trois points, votre modélisation est probablement assez bonne. Ce n'est qu'ensuite qu'on pourra dénormaliser. Mais seulement lorsque cela sera nécessaire et après une justification argumentée.

L'article de Wikipedia présentant l'ensemble des formes normales permet de mieux appréhender le sujet : https://fr.wikipedia.org/wiki/Forme_normale_(bases_de_donn%C3%A9es_relationnelles).

Il existe aussi le Manga Guide to Databases[1] qui aborde le sujet de manière très ludique.

NOMMAGE DES OBJETS

Il est temps alors de nommer les différents objets de la base de données, les tables et les colonnes en premier lieu.

Il est recommandé d'utiliser un système de nommage cohérent pour les objets de la base de données, avec des noms porteurs d'un minimum de sémantique.

Mais il faut éviter que les noms soient sensibles à la casse. On peut mélanger minuscules et majuscules dans un nom d'objet. Mais si les noms d'objets sont entourés de double guillemets, alors, ces guillemets seront requis dans toutes les requêtes où l'objet sera cité. Notons que PostgreSQL stocke les noms d'objets en minuscule dans ses tables internes, sauf quand le nom utilisé à leur création est entre guillemets. Le projet PostgreSQL recommande d'ailleurs d'écrire les requêtes en utilisant des majuscules pour les mots-clés et des minuscules pour les noms[2] , qui est une convention largement utilisée.

Par ailleurs, on peut légitimement se poser la question d'utiliser un singulier ou un pluriel pour nommer une table. Lorsque c'est possible, on préférera l'utilisation d'un invariant pour représenter un ensemble. Par exemple, une table `personnel` représente l'ensemble du personnel d'une société, plutôt qu'une table `employe` ou `employes`.

Le site SQL Style Guide[3] donne un résumé des bonnes pratiques d'écriture et de nommage.

1. https://nostarch.com/mg_databases.htm
2. https://www.postgresql.org/docs/current/static/sql-syntax-lexical.html
3. https://www.sqlstyle.guide/

BASES DE DONNÉES ET SCHÉMAS

Mais avant de créer physiquement les tables, il faut définir l'organisation générale de l'instance PostgreSQL qui va les gérer.

Une instance se compose d'une ou plusieurs *databases* (avec la base postgres créée par défaut). Dans une *database*, on peut trouver un ou plusieurs schémas (avec le schéma public créé par défaut). Ce sont les schémas qui vont contenir tables, vues, fonctions, etc.

Pour définir la façon dont on va organiser les choses, il faut savoir que :

- une requête SQL est soumise par un client connecté à une *database*. De manière standard, cette requête ne pourra accéder qu'à des tables contenues dans la *database*. La connexion ne verra aucun objet contenu dans une autre *database*,
- les schémas ne représentent que des espaces de nommage permettant éventuellement de gérer des objets portant le même nom.

En conséquence, toutes les tables pouvant être accédées depuis une même connexion, voire dans une même requête au travers de jointures, doivent être localisées dans la même **database**.

Notons pour être complet qu'il existe tout de même des moyens d'accéder à des tables se trouvant dans une autre base (utilisation de Foreign Data Wrapper ou dblink). Mais cela nécessite un travail d'administration plus important et surtout est plus pénalisant en terme de performance.

Pour ce qui concerne la **répartition des tables dans les schémas**, tant qu'il n'y a pas d'homonyme, on peut placer toutes les tables dans le même schéma, 'public' par exemple. Mais ce serait dommage de se priver de cette capacité de catégorisation, en particulier lorsque le nombre de tables devient important. Ainsi, créer au minimum un schéma par application est une bonne pratique, en laissant 'public' pour les tables qui ne font pas partie directement de l'application (la table créée temporairement par un testeur par exemple). On peut aussi aller plus loin, en séparant les tables en fonction de leur profil d'accès, ou de toute autre typologie. Les schémas permettent aussi de regrouper les fonctions et ainsi émuler les concepts de packages d'autres SGBD.

Chaque *database* possède un attribut important, **l'encodage**, avec un vaste choix de possibilités. Utiliser **UTF8** de préférence. Il permet d'encoder les caractères de toutes les langues, dont naturellement les caractères accentués français. Au cas où le caractère multi-byte de l'encodage UTF8 poserait problème, on pourra se rabattre sur l'encodage ISO889-15, alias LATIN-9. En revanche, il est fortement déconseillé d'opter pour l'encodage SQL_ASCII, car ce dernier ne sait traiter aucun transcodage entre clients et serveurs.

TYPAGE DES DONNÉES

D'une manière générale, il faut typer les données de la manière la plus précise possible. Ceci permet d'allier intégrité des données, performance et efficacité du stockage.

PostgreSQL supporte l'ensemble des types de bases définis par la norme SQL mais implémente également des extensions particulièrement utiles. Le SGBD propose une grande variété de types, certains sont assez standards (`char`, `int` ...). D'autres sont plus avancés (`polygon`, `jsonb`, `inet`...) et sont hélas moins connus. On peut retrouver l'ensemble des types dans la documentation : Data types[4] .

Parallèlement au choix des types, il est recommandé d'utiliser des unités issues du système international pour représenter des grandeurs physiques.

TYPES STANDARDS

Types alphanumériques

Pour représenter une chaîne de caractère, PostgreSQL offre les types `CHAR` et `VARCHAR` :

- Le type `char(n)` permet de stocker des chaînes de caractères de taille fixe, donnée par l'argument `n`. Si la chaîne que l'on souhaite stocker est plus petite que la taille donnée à la déclaration de la colonne, elle sera complétée par des espaces à droite. Si la chaîne que l'on souhaite stocker est trop grande, une erreur sera générée.
- Le type `varchar(n)` permet de stocker des chaînes de caractères de taille variable. La taille maximale de la chaîne est donnée par l'argument `n`. Toute chaîne qui excédera cette taille ne sera pas prise en compte et générera une erreur. Les chaînes de taille inférieure à la taille limite seront stockées sans altérations. Avec PostgreSQL, la longueur de chaîne est mesurée en nombre de caractères (contrairement à d'autres SGBD qui l'expriment en octets).
- Le type `text` permet de stocker des chaînes de caractères sans contrainte de longueur. La seule contrainte est la limite physique de stockage d'une colonne, qui est de 1 Go.

Il n'y a pas d'écart de performance significatifs entre ces différents types (contrairement à d'autres SGBD). On peut toutefois noter un surcout au niveau du stockage pour le type `char(n)` à cause de sa taille fixe. Les types contraints en longueur induisent un léger surcoût CPU pour contrôler la longueur des chaînes de caractères. Le type `text`, n'effectuant aucun contrôle de taille et n'étant pas à taille fixe, devrait être le plus performant. (Voir : https://www.postgresql.org/docs/current/static/datatype-character.html)

4. https://www.postgresql.org/docs/current/static/datatype.html

Types numériques entiers

Si une donnée numérique ne contient que des valeurs entières, l'utilisation d'un type `INTEGER`, `SMALLINT` ou `BIGINT` sera nettement plus performant qu'un `NUMERIC`.

Les types entiers sont signés, donc le type `INTEGER` par exemple, codé sur 4 octets, permet de stocker des valeurs comprises entre -2147483648 et +2147483647. Le standard SQL ne propose pas de stockage d'entiers non signés.

Types numérique décimaux

Par ailleurs, le standard SQL permet de stocker des valeurs décimales en utilisant les types à virgules flottantes. Avant de les utiliser, il faut avoir à l'esprit que ces types de données ne permettent pas de stocker des valeurs exactes, des différences peuvent donc apparaître entre la donnée insérée et la donnée restituée. Le type `REAL` permet d'exprimer des valeurs à virgules flottantes sur 4 octets, avec une précision relative de six décimales. Le type `DOUBLE PRECISION` permet d'exprimer des valeurs à virgules flottantes sur huit octets, avec une précision relative de 15 décimales.

Ce petit exemple nous permet de voir rapidement que les calculs ne sont pas précis dès qu'un type flottant entre en jeu :

```
SELECT 1.345::real + 10 AS resultat_flottant ;

resultat_flottant
-------------------
11.3450000286102
```

Beaucoup d'applications, notamment les applications financières, ne se satisfont pas de valeurs inexactes. Pour cela, le standard SQL propose le type `NUMERIC`, ou son synonyme `DECIMAL`, qui permet de stocker des valeurs exactes, selon la précision arbitraire donnée. Dans la déclaration `NUMERIC(precision, echelle)`, la partie `precision` indique combien de chiffres significatifs sont stockés, la partie `echelle` exprime le nombre de chiffres après la virgule. Au niveau du stockage, PostgreSQL ne permet pas d'insérer des valeurs qui dépassent les capacités du type déclaré. En revanche, si l'échelle de la valeur à stocker dépasse l'échelle déclarée de la colonne, alors sa valeur est simplement arrondie.

On peut aussi utiliser `NUMERIC` sans aucune contrainte de taille, pour stocker de façon exacte n'importe quel nombre sans l'arrondir. On préférera donc `NUMERIC` aux types flottants, qui sont à proscrire pour toute représentation de données où la précision est de mise.

Type booléen

À noter l'existence d'un type `BOOLEAN` pour gérer directement les … booléens.

```
CREATE TABLE evenements (
    ...
    en_attente BOOLEAN NOT NULL, ...
);

SELECT * FROM evenements WHERE en_attente;
```

Mais attention, un type booléen peut avoir 3 états : `true`, `false` ou `NULL`. C'est une particularité du standard SQL. En conséquence, si la colonne en_attente n'avait pas l'attribut NOT NULL, les deux conditions "WHERE en-attente" et "WHERE NOT (en_attente IS FALSE)" ne seraient **pas** équivalentes !

Types temporels

Pour les données temporelles, le type `DATE` gère … des dates (donc sans notion d'heure). Pour des "points dans le temps", utiliser le type `TIMESTAMP`. En fait, il est préférable dans ce cas d'utiliser le type `TIMESTAMPTZ` qui incorpore la notion de fuseau horaire ("Time Zone", par défaut celui du serveur). Ainsi, on a des références temporelles valables dans le monde entier. Même si toutes les données ne concerne qu'un fuseau horaire, il est préférable d'utiliser `TIMESTAMP WITH TIME ZONE` car les fonctions qui manipulent ce type sont `IMMUTABLE` et offriront de meilleures performances. Les fonctions `IMMUTABLE` permettent à PostgreSQL de mieux ordonner l'exécution d'une fonction.

Enfin, le type `INTERVAL` représente une différence entre deux points dans le temps. Son utilisation permet de faire des calculs sur les dates de manière simple et efficace, sans avoir besoin d'utiliser des fonctions alambiquées.

À noter que l'utilisation du type `TIMESTAMP WITHOUT TIME ZONE` peut être dangereuse. La donnée stockée ne portera pas d'information relative au fuseau horaire :

```
SELECT TIMESTAMP '08-03-2018 11 :12 :36+05' ;
      timestamp
---------------------
 2018-03-08 11:12 :36
(1 ligne)
```

Il faudra également faire attention, lorsque l'on ne précise pas que le type comporte ou non la composante fuseau horaire, PostgreSQL utilisera le type `TIMESTAMP WITHOUT TIME ZONE` :

```
    CREATE TABLE test_ts (
      ts timestamp
```

```
    );

\d test_ts
                        Table "public.test_ts"
    Column |            Type             | Collation | Nullable | Default
    -------+-----------------------------+-----------+----------+---------
     ts    | timestamp without time zone |           |          |
```

On peut facilement réaliser des calculs à l'aide du type `interval` :

```
SELECT current_timestamp + interval '3 days' AS dans_3_jours;
         dans_3_jours
-------------------------------
 2018-03-25 10:42:07.759239+02
```

De nombreuses fonctions de manipulations de date[5] permettent également de manipuler des dates de façon intéressante. Par exemple la fonction `date_trunc` est particulièrement adaptée pour regrouper des données par trimestre :

```
SELECT date_trunc('quarter', date_evenement) AS date_debut_trimestre,
       extract('quarter', date_evenement) AS trimestre,
       count(*) AS occurences
    FROM donnees_temporelles
 GROUP BY date_debut_trimestre, trimestre
 ORDER BY date_debut_trimestre;
 date_debut_trimestre | trimestre | occurences
----------------------+-----------+------------
 2017-07-01 00:00:00  |         3 |         24
 2017-10-01 00:00:00  |         4 |         10
 2018-01-01 00:00:00  |         1 |          4
```

Par ailleurs, bien que la norme ISO (ISO-8601) impose le format de date "année-mois-jour", la norme SQL est plus permissive. PostgreSQL permet donc de restituer une date au format "jour/mois/année" si DateStyle est égal à 'SQL, DMY' :

```
SET datestyle = 'ISO, DMY';

SELECT current_timestamp;
              now
-------------------------------
 2018-03-22 11:45:36.17611+01

SET datestyle = 'SQL, DMY';

SELECT current_timestamp;
              now
```

5. https://www.postgresql.org/docs/current/static/functions-datetime.html

```
-------------------------------
 22/03/2018 11:45 :52.652619 CET
```

Attention toutefois, le paramètre `datestyle` n'influence que la restitution et n'a aucun impact sur l'entrée. PostgreSQL prend en compte l'entrée de dates sous les deux formes, indépendamment de `datestyle` :

```
SELECT timestamp '31/10/2016 00 :00 :00' AStest ;
            test
    ---------------------
     2016-10-31 00:00 :00
    (1 row)

SELECT timestamp '2016-10-10 00 :00 :00' AStest ;
            test
    ---------------------
     2016-10-10 00:00 :00
    (1 row)
```

La gestion intégrée des fuseaux horaires permet de transposer simplement un point de temps dans un fuseau horaire particulier :

```
SELECT current_timestamp AT TIME ZONE 'Asia/Tokyo' AS heure_tokyo ;
        heure_tokyo
----------------------------
 22/03/2018 19:46 :43.117204
```

TYPES AVANCÉES SPÉCIFIQUES À POSTGRESQL

JSON/JSONB

Parmi les types plus avancés, le format `JSON` est issue du monde Web et est très utilisé avec la tendance NoSQL.

PostgreSQL propose deux types : `JSON` et `JSONB`. Le type `JSON` est un type historique qui consiste à stocker l'objet dans un champ texte. Apparu ultérieurement, le type JSONB permet de stocker l'objet sous forme binaire, ce qui apporte des fonctionnalités de compression et d'indexation. Il convient donc d'utiliser systématiquement le type `JSONB` par rapport au type `JSON`, sauf très peu de recherches seront effectuées sur le document JSON et qu'il y a intérêt de garder la forme initiale de celui-ci.

Voici quelques exemples de fonctions et opérateurs :

Créons une table voiture contenant des objets jsonb :

```
CREATE TABLE demo_json (voiture jsonb) ;
INSERT INTO demo_json
```

```
VALUES ('{"marque" : "renault",
          "modele" : "clio",
          "couleur" : "noire",
          "carburant" : "diesel"}'::jsonb);

SELECT jsonb_pretty(voiture) FROM demo_json ;
       jsonb_pretty
---------------------------
 {                        +
     "marque": "renault", +
     "modele": "clio",    +
     "couleur": "noire",  +
     "carburant": "diesel"+
 }
(1 ligne)
```

On peut ajouter un tableau de passagers avec l'ordre UPDATE et l'opérateur de concaténation `||` :

```
UPDATE demo_json
SET voiture = voiture || '{"passagers" : ["tintin","milou"]}'
WHERE voiture='{"marque" : "renault",
                "modele" : "clio",
                "couleur" : "noire",
                "carburant" : "diesel"}';

SELECT jsonb_pretty(voiture) FROM demo_json ;
        jsonb_pretty
----------------------------
 {                         +
     "marque": "renault",  +
     "modele": "clio",     +
     "couleur": "noire",   +
     "carburant": "diesel",+
     "passagers": [        +
         "tintin",         +
         "milou"           +
     ]                     +
 }
(1 ligne)
```

Voici un autre exemple avec une table beaucoup plus volumineuse (40 Go) : celle-ci comprend tout simplement deux colonnes, `post_id` de type integer et `json` de type jsonb.

On peut utiliser l'opérateur d'inclusion '@>' : Est-ce que l'objet json contient un autre objet JSON ? Une recherche sur cette grosse table entraînerait un scan complet de la table comme en témoigne ce plan d'exécution :

```
explain SELECT * FROM json_stack
    WHERE json @>  '{"displayname" :"anayrat"}'::jsonb;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on json_stack  (cost=0.00..5034687.70 rows=33283 width=1014)
   Filter: (json @> '{"displayname" : "anayrat"}'::jsonb)
(2 lignes)
```

Enfin, toute la puissance de PostgreSQL est d'offrir des méthodes d'indexation pour la plupart des types. Le JSONB peut ainsi être indexé à l'aide d'index de type GIN :

Il est possible de créer un index GIN portant sur la colonne json (notez le `USING gin`) :

```
CREATE INDEX ON json_stack  USING gin ( json ) ;
```

Le moteur peut maintenant utiliser l'index et la recherche devient très rapide :

```
explain (analyze,buffers)  SELECT * FROM json_stack
    WHERE json @>  '{"displayname" :"anayrat"}'::jsonb;
                     QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on json_stack  (cost=286.95..33866.98 rows=33283 width=1014)
                                 (actual time=0.101..0.103 rows=2 loops=1)
   Recheck Cond : (json @> '{"displayname" : "anayrat"}'::jsonb)
   Heap Blocks : exact=2
   Buffers : shared hit=17
   ->  Bitmap Index Scan on json_stack_json_idx
               (cost=0.00..278.62 rows=33283 width=0)
               (actual time=0.095..0.095 rows=2 loops=1)
         Index Cond : (json @> '{"displayname" : "anayrat"}'::jsonb)
         Buffers : shared hit=15
 Planning time: 0.069 ms
 Execution time: 0.122 ms
(9 lignes)
```

Pour les données qui sont par nature peu structurées, on privilégiera un stockage avec JSONB (voire JSON). Sinon, il vaut mieux privilégier une structure classique.

Par exemple, le moteur ne possède pas de statistiques sur le type JSON, ce qui peut le conduire à choisir des plans non optimaux. Un contournement reste toujours possible grâce à des index fonctionnels. Cet article donne un exemple : PostgreSQL - JSONB et Statistiques[6] . Il n'est également pas possible de garantir la cohérence des données (typage, contraintes,...). Ces vérifications doivent être faites côté applicatif.

Pour plus d'informations, vous pouvez consulter la documentation :

- pour les opérateurs : JSON Functions and Operators[7]

6. https://blog.anayrat.info/2017/11/26/postgresql---jsonb-et-statistiques/
7. https://www.postgresql.org/docs/current/static/functions-json.html

- pour les techniques d'indexation : jsonb Indexing[8]
- module de formation : PostgreSQL avancé, partie 2[9]

Types tableaux

Les types `ARRAY` permettent de stocker des tableaux multi-dimensionnels de n'importe quels types supportés par PostgreSQL.

Stocker des ensembles de valeurs viole la première forme normale[10] . On n'utilisera le type tableau lorsqu'il y a un intérêt de dénormaliser les données. Il faudra toujours se questionner sur l'utilisation de types tableaux.

Il existe deux syntaxes :

La plus répandue :

- `type[]` : Par exemple `int[]` permet d'obtenir un tableau d'entier sans restriction de nombre d'éléments.
- `type[n]` : Permet de spécifier une taille de `n` éléments.
- `type[][]` : Permet de créer un tableau à deux dimensions

Celle conforme au standard :

- `type ARRAY[n]` : Crée un tableau d'une dimension avec `n` éléments
- `type ARRAY` : Crée un tableau à une dimension sans restriction de nombre d'élément.

A noter que la syntaxe conforme au standard est plus restrictive. Elle ne permet pas d'avoir des tableaux à plusieurs dimensions.

Pour valoriser un tableau, la syntaxe est la suivante : `{ valeur delimiteur valeur delimiteur }`

Le délimiteur courant est la virgule `,` mais cela peut dépendre du type.

Pour imbriquer des tableaux cela donne :

```
CREATE TABLE t4 (c1 text[][]);

INSERT INTO t4
VALUES ('{ { valeur[1][1],valeur[1][2] } , { valeur[2][1],valeur[2][2] } } ');
```

Pour accéder à un certain élément du tableau il suffit d'identifier l'élément entre crochets :

8. https://www.postgresql.org/docs/current/static/datatype-json.html#JSON-INDEXING
9. https://dali.bo/dba2_html#hstore-json-jsonb
10. https://fr.wikipedia.org/wiki/Forme_normale_(bases_de_donn%C3%A9es_relationnelles)#1FN_%E2%80%93_Premi%C3%A8re_forme_normale

```
SELECT c1[2][1] FROM t4;
      c1
--------------
 valeur[2][1]
(1 ligne)
```

Il est aussi possible de faire du *slicing* :

```
SELECT c1[1:2][1] FROM t4;
               c1
---------------------------------
 {{valeur[1][1]},{valeur[2][1]}}
(1 ligne)
```

Pour rechercher un enregistrement contenant tel élément il est possible d'utiliser deux syntaxes :

- `WHERE element = ANY (colonne)`
- `WHERE colonne @> 'tableau'`

Par exemple à partir d'un jeu de donnée :

```
create table t5 (c1 int[]);
insert into t5 (
    select array(select generate_series(1,i ::int))
    from (
        select floor(1000*random()) i
        -- génère un tableau de taille aléatoire
        -- contenant entre 1 et 1000 éléments
        from generate_series(1,500000))
       sub
  );
  --génère 500 000 lignes
```

Combien de lignes contient l'élément 999 ?

```
SELECT count(*) FROM t5 WHERE 999 = ANY(c1);
 count
-------
   527
(1 ligne)
```

Autre syntaxe :

```
SELECT count(*) FROM t5 WHERE c1 @> '{999}';
 count
-------
   527
(1 ligne)
```

Encore une fois, l'intérêt d'utiliser des types spécialisés (et leurs opérateurs) est d'exploiter les possibilités d'indexation du moteur. Ainsi il est possible de créer un index GIN permettant d'indexer tous les éléments du tableau :

```
CREATE INDEX ON t5 USING gin ( c1 ) ;
```

Le moteur utilise bien l'index avec l'opérateur `@>` :

```
explain (analyze,buffers) SELECT count(*) FROM t5 WHERE c1 @> '{999}';
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=2489.87..2489.88 rows=1 width=8)
            (actual time=0.730..0.730 rows=1 loops=1)
   Buffers : shared hit=521
   ->  Bitmap Heap Scan on t5
               (cost=39.37..2483.62 rows=2500 width=0)
               (actual time=0.166..0.680 rows=527 loops=1)
         Recheck Cond : (c1 @> '{999}'::integer[])
         Heap Blocks : exact=518
         Buffers : shared hit=521
         ->  Bitmap Index Scan on t5_c1_idx
                  (cost=0.00..38.75 rows=2500 width=0)
                  (actual time=0.103..0.103 rows=527 loops=1)
               Index Cond : (c1 @> '{999}'::integer[])
               Buffers : shared hit=3
 Planning time: 0.061 ms
 Execution time: 0.755 ms
(11 lignes)
```

Ce qui n'est pas le cas avec l'autre syntaxe :

```
explain (analyze,buffers) SELECT count(*) FROM t5 WHERE 999 = ANY(c1) ;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=45898.07..45898.08 rows=1 width=8)
            (actual time=2840.531..2840.531 rows=1 loops=1)
   Buffers : shared hit=1074121
   ->  Seq Scan on t5  (cost=0.00..45891.82 rows=2500 width=0)
                       (actual time=4.868..2840.246 rows=527 loops=1)
         Filter: (999 = ANY (c1))
         Rows Removed by Filter: 499478
         Buffers : shared hit=1074121
 Planning time: 0.053 ms
 Execution time: 2840.549 ms
(8 lignes)
```

Pour plus d'informations, vous pouvez consulter la documentation :

https ://wwww.dalibo.com/

- pour les opérateurs : Array Functions and Operators[11]
- pour les techniques d'indexation : Index Types[12]

Range

Plus utile, le type `range` correspond à un intervalle de valeur, par exemple l'intervalle `[0, 10]` pour représenter tous les nombres entiers de 0 à 10 inclus. Le type ou plutôt les types `range`[13] permettent de représenter des intervalles de valeurs de différents types : intervalles d'entier (`int4range`), de numeric (`numericrange`) et des intervalles de date, timestamp avec ou sans fuseau (`daterange`, `tsrange`, `tstzrange`).

Un type `daterange` peut, par exemple, avantageusement remplacer des colonnes de type date de début et date de fin de validité. L'exemple ci-dessous montre l'implémentation d'une table de codes associés à une valeur et à une date de validité. L'implémentation de la contrainte d'exclusion (vue dans la partie sur les contraintes d'exclusion) nécessite l'utilisation de l'extension `btree_gist`. À noter l'utilisation systématique de la contrainte `NOT NULL`, la présence d'une clé artificielle et surtout d'une contrainte d'exclusion pour éviter d'avoir le même code qui apparaît sur des intervalles de temps qui se chevauchent (opérateur `&&`), donc pour avoir une seule valeur valide pour un code donné, à une date donnée.

```
CREATE EXTENSION btree_gist;

CREATE TABLE codes (
    code_id serial      PRIMARY KEY,
    code char(5)        NOT NULL,
    validite daterange  NOT NULL,
    valeur numeric(4,2) NOT NULL,
    EXCLUDE USING GIST (code WITH =, validite WITH &&)
);
```

La table pourra être peuplée par exemple avec des valeurs de TVA - bien que le nom de la table soit alors très mal choisi. On notera l'utilisation de la valeur spéciale `infinity` pour représenter une date de fin de validité infinie, bien plus parlante que le code traditionnel en l'an 9999 (`9999-12-31`) :

```
INSERT INTO codes (code, validite, valeur)
VALUES ('TVA', '[1992-01-01, 1994-12-31]', 1.186);

INSERT INTO codes (code, validite, valeur)
VALUES ('TVA', '[1995-01-01, 1999-12-31]', 1.206);
```

11. https://www.postgresql.org/docs/current/static/functions-array.html
12. https://www.postgresql.org/docs/current/static/indexes-types.html
13. https://www.postgresql.org/docs/current/static/rangetypes.html

```
INSERT INTO codes (code, validite, valeur)
VALUES ('TVA', '[2000-01-01, 2013-12-31]', 1.196);

INSERT INTO codes (code, validite, valeur)
VALUES ('TVA', '[2014-01-01, infinity]',   1.20);
```

En termes d'interrogation, on pourra facilement recalculer un prix en fonction de la TVA au jour de la vente, en utilisant l'opérateur de contenance d'une valeur scalaire dans un intervalle ('@>') pour déterminer la bonne valeur de TVA à la date de la vente :

```
SELECT article, prix_ht * tva.valeur AS prix_ttc
    FROM ventes v
    JOIN codes tva ON (tva.validite @> v.date_vente AND code = 'TVA');
```

On pourra également tester si une borne d'un intervalle vaut `infinity` avec les fonctions `lower_inf` et `upper_inf`.

Synthèse

Utiliser les types avancés présente plusieurs avantages :

- Vérification du type (quand c'est possible)
- Fonctions et opérateurs spécialisés
- Possibilité d'utiliser les techniques d'indexation
- Stockage plus efficace

Néanmoins cela peut présenter quelques limites :

- On peut perdre le bénéfice de modélisation. Par exemple une colonne JSON peut vite devenir « fourre-tout ».
- Suivant les types, le moteur ne dispose pas de statistiques avancées (JSONB, types géométriques etc)

Les types tels que JSONB ou hstore sont utilisés quand le modèle relationnel n'est pas assez souple. Les contextes où, lors de la conception, il serait nécessaire d'ajouter dynamiquement des colonnes à la table suivant les besoins du client, où le détail des attributs d'une entité ne sont pas connus (modélisation géographique par exemple), etc

CONTRAINTES D'INTÉGRITÉ

CLÉ PRIMAIRE

La clé primaire[14] est un concept fondamental pour l'accès aux données dans une base de données relationnelle. Une base relationnelle ne doit jamais être accédée via des positions physiques d'enregistrements. Mais on utilisera la clé primaire pour identifier une ligne de façon unique et certaine.

La ou les colonnes qui vont permettre d'identifier une ligne de manière unique seront appelées clés candidates. La clé primaire sera choisie parmi les clés candidates, en général, la plus courte. Il peut s'agir du matricule d'un employé par exemple. Dans certains cas, il faudra utiliser une clé dite artificielle[15] ou technique, qui sera par exemple un identifiant numérique séquentiel ajouté à la table, pour identifier les lignes de façon unique. Le cas se présentera par exemple lorsque l'identifiant est une longue chaîne de caractères, si les données composant la clé sont susceptibles d'être modifiées ou si les données sont susceptibles d'apparaître en doublon. Il faudra toutefois veiller à ne pas en abuser et utiliser le plus souvent possible des clés naturelles lorsqu'elles sont présentes et utilisables.

Comme indiqué plus haut, lorsque qu'il n'a pas été possible d'identifier une clé primaire pour une table, on utilisera une clé technique. Plusieurs choix s'offrent alors au concepteur de la base de données :

- le type entier (`INTEGER` ou `BIGINT`) accompagné d'une séquence mise à jour par l'application,
- le type `SERIAL` ou `BIGSERIAL`, qui équivaut à un entier accompagné d'une séquence et d'une valeur par défaut qui auto-incrémente la séquence,
- le type `IDENTITY` tel qu'il est défini par la norme SQL, équivalent d'un auto-incrément (disponible depuis la version 10 de PostgreSQL),
- le type `UUID` qui génère un Universal Unique Identifier[16] .

Bien que très séduisant à priori, le type `UUID` n'est pas recommandé pour plusieurs raisons. Tout d'abord, une donnée UUID est stockée sous la forme d'un entier de 128 bits. Elle nécessite donc 2 à 4 fois plus de place que les types `BIGINT` ou `INTEGER` habituels,ces derniers types étant respectivement des entiers de 64 et 32 bits.

Par ailleurs, les identifiants de type `UUID` générés ne sont pas monotoniques comme les identifiants issus d'une séquence. En conséquence, le type UUID entraîne une fragmentation élevée des index.

14. https://fr.wikipedia.org/wiki/Cl%C3%A9_primaire
15. https://fr.wikipedia.org/wiki/Cl%C3%A9_artificielle
16. https://fr.wikipedia.org/wiki/Universal_Unique_Identifier

Ce qu'il faut retenir :

- Une clé primaire permet d'identifier une ligne de façon unique, il n'en existe qu'une seule par table.
- Toute table doit avoir une clé primaire.
- Une clé primaire garantit que toutes les valeurs de la ou des colonnes qui composent cette clé sont uniques et non nulles. Elle peut être composée d'une seule colonne ou de plusieurs colonnes, selon le besoin.
- La clé primaire est déterminée au moment de la conception du modèle de données.
- Les clés primaires créent implicitement un index qui permet de renforcer cette contrainte.
- Un index `UNIQUE` portant sur des colonnes non nulles peut faire office de clé primaire. Mais il est nettement préférable d'indiquer explicitement au SGBD qu'il s'agit d'une clé primaire. De nombreux outils tirent profit de cette information.

CLÉ ÉTRANGÈRE

Une clé étrangère permet d'assurer la cohérence du modèle de donnée. Elle référence une colonne ou un ensemble de colonne d'une autre table qui forme soit une clé primaire, soit une contrainte unique.

L'exemple suivant montre comment créer une clé étrangère sur la table `commandes` qui référence la table `produits` :

```
CREATE TABLE produits (
  no_produit   integer PRIMARY KEY,
  nom_produit  text    NOT NULL,
  prix         numeric NOT NULL
);

CREATE TABLE commandes (
  id_commande serial  PRIMARY KEY,
  -- la colonne no_produit référence la colonne du même nom de la table produits
  no_produit  integer NOT NULL REFERENCES produits (no_produit),
  quantite    integer NOT NULL
);

INSERT INTO produits (no_produit, nom_produit, prix)
VALUES (1, 'Processeur 68000', 40.00),
       (2, '8375 Adress Unit Generator', 30.00),
       (3, '8373R4 Display Encoder Chip', 20.00),
       (4, '8364R7 Ports Audio Uart and Logic', 20.00);
```

Ainsi, le SGBD nous empêche d'insérer des commandes de produits qui n'existent pas :

```
INSERT INTO commandes (no_produit,quantite)
       VALUES (5, 100)
       RETURNING id_commande;

ERROR :  insert or update on table "commandes" violates
        foreign key constraint "commandes_no_produit_fkey"
DETAIL :  Key (no_produit)=(5) is not present in table "produits".
```

Il nous interdit également de supprimer un produit qui a été commandé :

```
DELETE FROM produits WHERE no_produit3;
ERROR :  update or delete on table "produits" violates
        foreign key constraint "commandes_no_produit_fkey"
        on table "commandes"
DETAIL :  Key (no_produit)=(3) is still referenced from table "commandes".
```

Pour approfondir le sujet :

- Intégrité référentielle[17]
- Clé étrangère[18]

UNICITÉ

Une contrainte d'unicité permet d'assurer l'unicité des valeurs d'une colonne, sans recourir à une clé primaire. Une clé étrangère peut aussi référencer une contrainte d'unicité.

Par exemple, on a une table utilisateurs qui utilise une clé technique par souci de performance, mais qui ajoute une contrainte d'unicité sur l'email de l'utilisateur :

```
CREATE TABLE utilisateurs (
  id    serial PRIMARY KEY,
  ...
  email text   UNIQUE NOT NULL,
  ...
);
```

Cet exemple permet de s'assurer qu'un utilisateur ne crée pas deux comptes avec la même adresse email.

17. https://fr.wikipedia.org/wiki/Int%C3%A9grit%C3%A9_r%C3%A9f%C3%A9rentielle
18. https://fr.wikipedia.org/wiki/Cl%C3%A9_%C3%A9trang%C3%A8re

CONTRAINTES CHECK ET NOT NULL

Pour maintenir une qualité des données optimum, il faut contrôler les données dès leur arrivée. Le typage fin des colonnes abordé plus tard apporte un premier niveau de contrôle. Mais il ne faut pas hésiter à ajouter des contraintes sur les tables et les colonnes.

Ainsi, une clause `NOT NULL` doit être ajoutée quand la colonne ne peut contenir de valeur `NULL` (même si elle ne fait pas partie de la clé primaire).

En principe, chaque table doit également avoir une clé primaire explicite. Cela oblige au moins une fois à se poser la question : "Qu'est-ce qui identifie une ligne de cette table de manière unique ?". Il est préférable de déclarer une `PRIMARY KEY` plutôt que d'avoir des colonnes déclarées `NOT NULL` et `UNIQUE`, car de nombreux outils, comme Slony voire PostgreSQL lui-même, ont besoin de connaître cette clé primaire.

La clause `CHECK` permet d'ajouter des contrôles sur le contenu des données (`CHECK > 0` permet de vérifier qu'une donnée est strictement positive par exemple). Pour des types de contrôles rencontrés fréquemment, il est utile de créer des domaines et de les utiliser ensuite comme type de colonne. Par exemple :

```
CREATE DOMAIN salaire AS integer CHECK (VALUE > 0);
CREATE TABLE employes (id serial, nom text, paye salaire);
```

De la même manière, il est préférable de déclarer des clés étrangères (`FOREIGN KEY`) entre les tables, plutôt que de vouloir assurer l'intégrité référentielle par les applications. Les contrôles sont plus systématiques et la gestion par le SGBD est moins coûteuse. Il faut également penser à créer les index sur ces clés étrangères.

Un dernier type de contrainte peut être très efficace : les contraintes d'exclusion, par exemple pour vérifier très simplement que des intervalles de date ne se chevauchent pas (voir exemple dans la partie sur les contraintes avancées).

Il faut néanmoins garder en tête que toute contrainte à un coût (très faible pour un `NOT NULL`, limité pour un `CHECK`, plus lourd pour une clé étrangère). On évitera donc d'aller dans un extrême où chaque colonne se verrait dotée d'un grand nombre de contraintes.

En fonction de leur type, certaines contraintes, dites `DEFFERABLE`, peuvent être contrôlées à la fin des transactions. Dans ce cas, même si on souhaite que le contrôle soit immédiat, il peut être intéressant pour la souplesse de déclarer ces contraintes `DEFFERABLE INITIALLY IMMEDIATE`. Si on en a besoin, on pourra plus facilement basculer à un contrôle différé. Pour plus d'information consulter cet article dans la base de connaissances Retarder la vérification des contraintes[19]

19. https://blog.anayrat.info/2016/08/13/postgresql--retarder-la-v%C3%A9rification-des-contraintes/

Par ailleurs, il faut avoir à l'esprit que le typage que l'on définit sur la base de données modélisée amène également une contrainte : il ne sera pas possible de stocker un entier dans une colonne de type `date`, ou de stocker une chaîne de caractère trop longue dans un type `varchar` de taille définie.

CONTRAINTES D'EXCLUSION

Les contraintes d'exclusion sont spécifiques à PostgreSQL. Elles permettent d'exprimer simplement certaines contraintes complexes, par exemple pour éviter de réserver une même salle sur deux créneaux horaires qui se chevauchent. Le codage d'une telle contrainte au niveau des applications est particulièrement complexe à réaliser, car elle doit non seulement tenir compte des aspects métiers mais également des aspects techniques comme la garantie de la concurrence d'accès.

Dans l'exemple ci-dessous, il est possible de créer une contrainte d'exclusion pour éviter d'avoir le même code qui apparaît sur des intervalles de temps qui se chevauchent (opérateur `&&`, voir le types "range" plus haut dans le document), donc pour avoir une seule valeur valide pour un code donné, à une date donnée. Cette contrainte s'appuie sur un index GiST, l'extension `btree_gist` permettra d'utiliser l'opérateur d'égalité pour montrer que la contrainte de recouvrement ne s'applique que sur un même code.

```
CREATE EXTENSION btree_gist;

CREATE TABLE codes (
    code_id serial      PRIMARY KEY,
    code char(5)        NOT NULL,
    validite daterange  NOT NULL,
    valeur numeric(4,2) NOT NULL,
    EXCLUDE USING GIST (code WITH =, validite WITH &&)
);
```

DDL TRANSACTIONNEL

Avec PostgreSQL, tous les ordres de types DDL sur des objets d'une base de données sont transactionnels. On peut donc insérer ses `CREATE TABLE`, `ALTER TABLE` et autres `CREATE INDEX` dans des transactions. Il ne faut donc pas hésiter à encadrer ses scripts de DDL par des ordres BEGIN et COMMIT.

Il y a quelques exceptions à cela, entre autres les ordres `CREATE INDEX CONCURRENTLY`, `DROP DATABASE` et `VACUUM` ne peuvent être exécutés dans une transaction.

DROITS SUR LES OBJETS

La sécurité est l'affaire de tous, et pas seulement des DBA de production ou des RSSI ! Mais pour ce qui nous concerne, il s'agit pour l'essentiel de gérer les droits d'accès aux objets relationnels donnés aux ROLEs (utilisateurs) de connexion.

ORGANISATION DES RÔLES ET DES DROITS

Il est recommandé de se bâtir un "schéma de sécurité" en définissant comment on va limiter et contrôler les accès aux tables, fonctions, etc.

L'utilisation des rôles de niveau groupe (option NOLOGIN) permet de simplifier cette gestion d'accès. On a ainsi intérêt à définir des profils d'accès que l'on attribuera ensuite aux rôles de niveau utilisateur. Dans cet esprit, on distinguera probablement un rôle ayant le droit de modifier la structure de la base, des rôles ayant des droits d'écriture dans les tables et des rôles qui n'ont que des droits en lecture.

Dans PostgreSQL, seul un super-utilisateur ou le propriétaire d'un objet peut en modifier la définition, ou le supprimer. Il est donc recommandé de distinguer les rôles applicatifs du propriétaire des objets.

Ainsi, on pourra entrevoir les ordres suivants pour différencier les accès aux tables :

```
ALTER TABLE nomtable OWNER TO proprietaire_schema ;

GRANT SELECT, INSERT, UPDATE, DELETE
ON nomtable TO utilisateur_applicatif_rw ;

GRANT SELECT
ON nomtable TO utilisateur_applicatif_ro ;
```

Lorsque l'on ajoute un nouvel utilisateur à une base existante, on pourra octroyer des privilèges particuliers sur l'ensemble des objets d'un schéma :

```
GRANT SELECT
ON ALL TABLES IN SCHEMA public TO utilisateur_applicatif_ro ;

GRANT SELECT, INSERT, UPDATE, DELETE
ON ALL TABLES IN SCHEMA public TO utilisateur_applicatif_rw ;

GRANT USAGE
ON ALL SEQUENCES IN SCHEMA public TO utilisateur_applicatif_rw ;
```

La page de manuel de l'ordre GRANT[20] permet de lister l'ensemble des privilèges que l'on peut octroyer en fonction du type d'objet défini.

20. https://www.postgresql.org/docs/current/static/sql-grant.html

Une autre technique, spécifique à PostgreSQL, peut simplifier la maintenance dans le temps du schéma de sécurité. Il s'agit des privilèges par défaut (DEFAULT PRIVILEGES). Ils permettent d'affecter des droits par défaut lors de la création d'un objet (typiquement une table) en fonction des rôles et/ou du schéma hébergeant le nouvel objet.

L'ordre ALTER DEFAULT PRIVILEGES[21] permet de réaliser cela, par exemple pour accorder à `utilisateur_applicatif_ro` le droit de lire toutes les tables qui seront créées par le rôle `proprietaire_schema` dans les schémas `referentiel` et `archives` :

```
ALTER DEFAULT PRIVILEGES
     FOR ROLE proprietaire_schema
     IN SCHEMA referentiel, archives
GRANT SELECT
    ON TABLES
    TO utilisateur_applicatif_ro ;
```

Pour approfondir le sujet vous pouvez consulter ce module de formation Droits sur les objets[22]

SÉCURISATION DES FONCTIONS

Concernant les droits d'exécution des fonctions, il est important de noter que, par défaut, une fonction qui vient d'être créée est exécutable par tous. Une bonne pratique consiste donc à retirer le droit d'exécution pour tous les utilisateurs sur les fonctions après leur création et à n'ouvrir des droits d'exécution que de manière explicite. L'ordre pour cette opération est de la forme : `REVOKE EXECUTE … FROM PUBLIC` .

Le code d'une fonction peut parfois nécessiter des droits plus élevés que ce que possède l'utilisateur de la fonction. Dans ce cas, plutôt que d'étendre les droits de l'utilisateur en question, il est nettement préférable de simplement déclarer la fonction `SECURITY DEFINER`. Seule l'exécution de la fonction s'effectuera avec les droits plus étendus du créateur de la fonction.

21. https://www.postgresql.org/docs/current/static/sql-alterdefaultprivileges.html
22. https://cloud.dalibo.com/p/exports/formation/manuels/modules/f/f.handout.html#droits-sur-les-objets

SÉCURISER UNE INSTALLATION PAR DÉFAUT

En tout premier lieu, on donnera un mot de passe au rôle postgres : `ALTER ROLE postgres WITH PASSWORD 'new_password';`

Le fichier pg_hba.conf sera configuré pour utiliser a minima la méthode d'authentification `md5`, la méthode `scram-sha-256` étant préférable depuis la version 10, car plus sûre (si le client le permet).

Donner la propriété des bases à un rôle non applicatif : `ALTER DATABASE nombase OWNER TO role ;`

Enfin, pour éviter les effets de bords de la faille CVE-2018-1058, supprimer les privilèges du rôle PUBLIC. On retirera a minima le privilège `CREATE` :

```
REVOKE ALL ON DATABASE db_name FROM PUBLIC;
REVOKE ALL ON SCHEMA public FROM PUBLIC;
```

Pour de plus amples détails concernant cette faille, l'article suivant de Daniel Vérité à ce sujet est particulièrement intéressant : Schéma et CVE 2018-1058[23] .

INDEXATION

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

INDEXATION DES CLÉS ÉTRANGÈRES

Il est recommandé de créer systématiquement des index sur les colonnes portant des clés étrangères. Cela facilite souvent les jointures entre la table portant la contrainte et la table référencée. De plus, lorsque la contrainte permet de mettre à jour ou supprimer les données en cascade, ces mêmes mises à jour ou suppressions peuvent bénéficier de l'index sur la clé étrangère.

23. http://blog-postgresql.verite.pro/2018/03/14/schemas-et-cve-2018-1058.html

INDEXATION POUR LES REQUÊTES

PostgreSQL offre de très nombreuses possibilités d'indexation. Non seulement les index Btree offrent des fonctionnalités parfois inédites, mais PostgreSQL offrent également des structures d'index (GiST, GIN, etc) qui permettent de répondre à bon nombre de besoins.

Pour explorer les possibilités d'utilisation des index Btree, nous recommandons la lecture du site Use the index, Luke[24] qui présente les cas d'utilisations les plus courants sur les SGBD les plus courants eux aussi. Le site, maintenu par Markus Winand, propose une version en ligne de son livre *SQL Performance Explained*. Une version française est par ailleurs disponible sous le titre *SQL : au cœur des performances*.

Index pour une clause WHERE

Un index peut satisfaire une clause `WHERE`, par exemple pour la requête simple `SELECT * FROM produits WHERE nom = 'perceuse'` :

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

On peut voir que l'index est alors utilisé lorsque l'on effectue une recherche :

```
EXPLAIN SELECT * FROM produits WHERE nom = 'perceuse';
                        QUERY PLAN
-----------------------------------------------
 Index Scan using produits_nom_idx on produits
         (cost=0.28..8.29 rows=1 width=184)
   Index Cond : ((nom) ::text = 'perceuse'::text)
(2 rows)
```

Index pour un tri ou un regroupement

Un index btree étant par construction trié, le moteur peut exploiter cette caractéristiques lorqu'un résultat trié est requis.

Sans index pertinent, un tri est effectué par une opération `Sort` dans le plan d'exécution, à la suite d'une lecture. Ici, la lecture est un parcours séquentiel de la table t2 (`Seq Scan`) :

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=121342.45..123997.45 rows=1062000 width=4)
       (actual time=308.129..354.035 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: quicksort  Memory : 71452kB
   ->  Seq Scan on t2
```

24. https://use-the-index-luke.com/fr

```
            (cost=0.00..15045.00 rows=1062000 width=4)
            (actual time=0.088..142.787 rows=1000000 loops=1)
 Total runtime : 425.160 ms
```

Mais un index permet de récupérer les données triées directement :

```
b1=# CREATE INDEX ON t2(id);
CREATE INDEX
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
              QUERY PLAN
-----------------------------------------------------
 Index Scan using t2_id_idx on t2
    (cost=0.00..30408.36 rows=1000000 width=4)
    (actual time=0.145..308.651 rows=1000000 loops=1)
 Total runtime : 355.175 ms
(2 rows)
```

Si le critère de regroupement porte sur la clé primaire d'une table, PostgreSQL tire parti de cet index pour récupérer les données triées et réaliser le regroupement ainsi. Peu utile sur une requête portant sur une seule table, cela devient intéressant lorsque plusieurs tables sont jointes :

```
EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
    FROM commandes
    JOIN lignes_commandes
       USING (numero_commande)
 GROUP BY commandes.numero_commande;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Group (actual time=0.067..580.198 rows=168749 loops=1)
   ->  Merge Join (actual time=0.061..435.154 rows=675543 loops=1)
         Merge Cond : (commandes.numero_commande =
                       lignes_commandes.numero_commande)
         ->  Index Scan using orders_pkey on commandes
                       (actual time=0.027..49.784 rows=168750 loops=1)
         ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                       (actual time=0.025..131.606 rows=675543 loops=1)
               Heap Fetches : 0
 Total runtime : 584.624 ms
```

Index pour une jointure

Un index peut également accélérer une jointure, comme on peut le voir ici où l'index `orders_pkey` permet d'appliquer le filtre `numero_commande < 1000` et l'index `lignes_commandes_pkey` qui est utilisé pour réaliser la jointure entre les deux tables. On s'aperçoit que cela vérifie le conseil donné plus haut quant à l'indexation des colonnes portant une clé étrangère :

```
EXPLAIN SELECT * FROM commandes
    JOIN lignes_commandes USING (numero_commande)
 WHERE numero_commande < 1000;
                          QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.84..4161.14 rows=1121 width=154)
   ->  Index Scan using orders_pkey on commandes
           (cost=0.42..29.64 rows=280 width=80)
         Index Cond : (numero_commande < 1000)
   ->  Index Scan using lignes_commandes_pkey on lignes_commandes
           (cost=0.42..14.71 rows=5 width=82)
         Index Cond : (numero_commande = commandes.numero_commande)
```

Combiner plusieurs utilisations de l'index

On peut combiner deux colonnes d'index pour filtrer et récupérer les données triées. Par exemple, pour répondre à la requête suivante :

```
SELECT *
    FROM lignes_commandes
 WHERE numero_commande = 1000
 ORDER BY quantite;
```

Dans un cas comme celui-là, on crée un index qui indexe tout d'abord la colonne filtrante, puis la colonne sur laquelle on réalise le tri : `CREATE INDEX index_combine ON lignes_commandes (numero_commande, quantite);`

Cela permet de minimiser les accès et d'avoir un temps d'exécution extrêmement faible :

```
explain (ANALYZE)
SELECT *
    FROM lignes_commandes
 WHERE numero_commande = 1000
 ORDER BY quantite;
            QUERY PLAN
-----------------------------------------------
 Index Scan using index_combine
     (cost=0.43..34.97 rows=16 width=74)
     (actual time=0.036..0.036 rows=1 loops=1)
```

```
   Index Cond : (numero_commande = 1000)
 Planning time: 3.238 ms
 Execution time: 0.062 ms
```

Coût de maintenance d'un index

La présence d'un index ralentit les écritures sur une table. En effet, il faut non seulement ajouter ou modifier les données dans la table, mais il faut également maintenir le ou les index de cette table. Il convient donc de créer seulement les index **utiles** en fonction du modèle de donnée et des requêtes effectuées. L'exemple suivant l'illustre très bien le surcoût :

```
INSERT INTO t1 SELECT i FROM generate_series(1, 10000000)i;
Temps : 39674,079 ms

CREATE INDEX idx_t1_i ON t1(i);
INSERT INTO t1 SELECT i FROM generate_series(1, 10000000)i;
Temps : 94925,140 ms
```

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

DÉNORMALISER

Les techniques de dénormalisation classique sont utilisables avec PostgreSQL (quand c'est nécessaire, bien sûr) : découpage ou regroupement de tables, redondance de données. Les triggers et fonctions associées peuvent permettre d'éviter les inconvénients induits par la distorsion du modèle.

Dans ce domaine, une autre technique peut être utile : déclarer une colonne comme un tableau d'éléments de n'importe quel type. Par exemple une colonne déclarée `INTEGER[]` stocke des tableaux d'entiers. Ce qui peut permettre de facilement rompre avec la première forme normale ! Mais à consommer là aussi avec modération, car cela complique les accès aux données.

PIÈGES

NULL

La valeur `NULL` signifie habituellement :

- valeur non renseignée,
- valeur inconnue,
- valeur non applicable.

Dans tous les cas, c'est une absence d'information. Ou plus précisément l'information selon laquelle on n'a pas d'information, … ce qui est déjà une information !

Une table qui contient majoritairement des valeurs `NULL` contient bien peu de faits utilisables. La plupart du temps, c'est une table dans laquelle on stocke beaucoup de choses n'ayant que peu de rapport entre elles, les champs étant renseignés suivant le type de chaque « chose ». C'est donc le plus souvent un signe de mauvaise modélisation. Cette table aurait certainement dû être éclatée en plusieurs tables, chacune représentant une des relations qu'on veut modéliser.

Il est donc recommandé que tous les attributs d'une table portent une contrainte `NOT NULL`. Quelques colonnes peuvent ne pas porter ce type de contraintes, mais elles doivent être une exception. En effet, le comportement de la base de données est souvent source de problèmes lorsqu'une valeur `NULL` entre en jeu, par exemple la concaténation d'une chaîne de caractères avec une valeur retourne une valeur `NULL`, car elle est propagée dans les calculs. D'autres types de problèmes apparaissent également pour les prédicats.

Il faut avoir à l'esprit cette citation de Chris Date : « La valeur `NULL` telle qu'elle est implémentée dans SQL peut poser plus de problèmes qu'elle n'en résout. Son comportement est parfois étrange et est source de nombreuses erreurs et de confusions. »

GESTION DES OBJETS BINAIRES

Pour les objets binaires de taille "modeste", les colonnes de type `BYTEA` sont la réponse au besoin. Elles peuvent contenir des données jusqu'à 1 Go.

Pour les objets plus gros, il existe un ensemble de fonctions permettant de stocker puis retrouver de grands objets binaires. Mais leur gestion est particulière et nécessite un codage adapté. En fait, avant de vouloir étudier cette piste, il faut se demander si le stockage de tels objets à l'intérieur d'une base de données est pertinent. Bien souvent, le stockage d'un emplacement de fichier en base est suffisant.

COLONNE « FOURRE-TOUT »

Parmi les erreurs à éviter, il y a les colonnes de type variable, c'est-à-dire une colonne de type varchar qui contient :

- quelquefois un entier,
- quelquefois une date,
- un NULL,
- une chaîne autre,
- etc.

On a ici un gros problème de modélisation : la colonne a un type de contenu qui dépend de l'information qu'elle contient.

Si on prend l'exemple d'une colonne de type variable pour stocker le statut ou la date d'une tournée postale. On pourra trouver dans cette colonne 'NULL' si la tournée n'a pas eu lieu, la date de la tournée si elle a eu lieu et la chaîne « ANNULÉE » en cas d'annulation.

On aurait dû avoir une colonne supplémentaire (un booléen tournee_ok par exemple). On va aussi avoir un problème de performance en joignant ce varchar à la clé numérique de la table tournée. Le moteur n'aura que deux choix : convertir le varchar en numérique, avec une exception à la clé en essayant de convertir « ANNULÉE », ou bien (ce qu'il fera) convertir le numérique de la table tournee en chaîne. Cette dernière méthode rendra l'accès à l'id de tournée par index impossible. D'où un parcours complet (opération Seq Scan dans un plan d'exécution) de la table tournée à chaque accès.

Stockage entité-clé-valeur

Le modèle relationnel a été critiqué depuis sa création pour son manque de souplesse pour ajouter de nouveaux attributs ou pour proposer plusieurs attributs sans pour autant nécessiter de redévelopper l'application.

La solution souvent retenue est d'utiliser une table « à tout faire » entité-attribut-valeur qui est associée à une autre table de la base de données. Techniquement, une telle table comporte trois colonnes. La première est un identifiant généré qui permet de référencer la table mère. Les deux autres colonnes stockent le nom de l'attribut représenté et la valeur représentée.

Ainsi, pour reprendre l'exemple des informations de contacts pour un individu, une table personnes permet de stocker un identifiant de personne. Une table personne_attributs permet d'associer des données à un identifiant de personne. Le type de données de la colonne est souvent prévu largement pour faire tenir tout type d'informations, mais sous forme textuelle. Les données ne peuvent donc pas être validées.

```
CREATE TABLE personnes (id SERIAL PRIMARY KEY);

CREATE TABLE personne_attributs (
    id_pers INTEGER NOT NULL,
    nom_attr varchar(20) NOT NULL,
    val_attr varchar(100) NOT NULL
);

INSERT INTO personnes (id) VALUES (nextval('personnes_id_seq')) RETURNING id;
 id
----
    1

INSERT INTO personne_attributs (id_pers, nom_attr, val_attr)
    VALUES (1, 'nom', 'Prunelle'),
         (1, 'prenom', 'Léon');
(...)
```

Un tel modèle peut sembler souple mais pose plusieurs problèmes. Le premier concerne l'intégrité des données. Il n'est pas possible de garantir la présence d'un attribut comme on le ferait avec une contrainte `NOT NULL`. Si l'on souhaite stocker des données dans un autre format qu'une chaîne de caractères, pour bénéficier des contrôles de la base de données sur ce type, la seule solution est de créer autant de colonnes d'attributs qu'il y a de types de données à représenter. Ces colonnes ne permettront pas d'utiliser des contraintes CHECK pour garantir la cohérence des valeurs stockées avec ce qui est attendu, car les attributs peuvent stocker n'importe quelle donnée.

Les requêtes SQL qui permettent de récupérer les données requises dans l'application sont également particulièrement lourdes à écrire et à maintenir, à moins de récupérer les données attribut par attribut.

Des problèmes de performances vont donc très rapidement se poser. Cette représentation des données entraîne souvent l'effondrement des performances d'une base de données relationnelle. Les requêtes sont difficilement optimisables et nécessitent de réaliser beaucoup d'entrées-sorties disque, car les données sont éparpillées un peu partout dans la table.

Lorsque de telles solutions sont envisagées pour stocker des données transactionnelles, il vaut mieux revenir à un modèle de données traditionnel qui permet de typer correctement les données, de mettre en place les contraintes d'intégrité adéquates et d'écrire des requêtes SQL efficaces.

Dans d'autres cas, il vaut mieux utiliser un type de données de PostgreSQL qui est approprié, comme hstore qui permet de stocker des données sous la forme clé->valeur. Ce type de données peut être indexé pour garantir de bons temps de réponses des requêtes qui

nécessitent des recherches sur certaines clés ou certaines valeurs.

Voici l'exemple précédent revu avec l'extension `hstore` :

```
CREATE EXTENSION hstore;
CREATE TABLE personnes (id SERIAL PRIMARY KEY, attributs hstore);

INSERT INTO personnes (attributs) VALUES ('nom=>Prunelle, prenom=>Léon');
INSERT INTO personnes (attributs) VALUES ('prenom=>Gaston,nom=>Lagaffe');
INSERT INTO personnes (attributs) VALUES ('nom=>DeMaesmaker');

SELECT * FROM personnes;
 id |              attributs
----+--------------------------------------
  1 | "nom"=>"Prunelle", "prenom"=>"Léon"
  2 | "nom"=>"Lagaffe", "prenom"=>"Gaston"
  3 | "nom"=>"DeMaesmaker"
(3 rows)

SELECT id, attributs->'prenom' FROM personnes;
 id | ?column?
----+----------
  1 | Léon
  2 | Gaston
  3 |
(3 rows)

SELECT id, attributs->'nom' FROM personnes;
 id |  ?column?
----+-------------
  1 | Prunelle
  2 | Lagaffe
  3 | DeMaesmaker
(3 rows)
```

Trop de colonnes

Une table comportant un très grand nombre de colonnes (plusieurs dizaines voire centaines) est souvent le signe d'un problème de conception, surtout si bon nombre d'entre elles contiennent beaucoup de valeur NULL. Il est très possible que cette entité puisse être décomposée de « sous-entités », qu'on pourrait modéliser séparément. Il peut naturellement y avoir des cas où ce grand nombre de colonne est justifié. Mais cette caractéristique est souvent l'indice du faiblesse dans la modélisation, surtout si on trouve en fin de structure des noms de colonne du genre `attribut_supplementaire_1`...

En dehors du problème de modélisation, il y a également des incidences sur les perfor-

mances. Souvent, les requêtes n'ont besoin que d'un nombre limité de colonnes. Mais comme toutes les colonnes d'une ligne sont stockées physiquement au même endroit, beaucoup de données est accédé inutilement, entraînant une sensible perte de performances.

Absence de contraintes

Les contraintes d'intégrité et notamment les clés étrangères sont parfois absentes des modèles de données. Les problématiques de performance et de flexibilité sont souvent mises en avant, alors que les contraintes sont justement une aide pour l'optimisation de requêtes par le planificateur. La compatibilité avec d'autres SGBD ou la commodité de développement sont également des arguments parfois mis en avant.

De plus, l'absence de contraintes va également entraîner des problèmes d'intégrité des données. Il est par exemple très compliqué de se prémunir efficacement contre une situation où deux sessions ou plus modifient des données en tables au même moment. en l'absence de clé étrangère. Lorsque ces problèmes d'intégrité seront détectés, il s'ensuivra également la création de procédures de vérification de cohérence des données qui vont aussi alourdir les développements, entraînant ainsi un travail supplémentaire pour trouver et corriger les incohérences. Ce qui a été gagné d'un côté est perdu de l'autre.

Les contraintes d'intégrité sont des informations qui garantissent non seulement la cohérence des données mais qui vont également influencer l'optimiseur dans ses choix de plans d'exécution.

Parmi les informations utilisées par l'optimiseur, les contraintes d'unicité permettent de déterminer sans difficulté la répartition des valeurs stockées dans une colonne : chaque valeur est simplement unique. L'utilisation des index sur ces colonnes sera donc probablement favorisée. Les contraintes d'intégrité permettent également à l'optimiseur de pouvoir éliminer des jointures inutiles avec un LEFT JOIN. Enfin, les contraintes CHECK sur des tables partitionnées permettent de cibler les lectures sur certaines partitions seulement, et donc d'exclure les partitions inutiles.

Clé technique

Il est à noter que l'emploi d'une clé technique peut poser des problèmes de performance, notamment sur des jointures. L'exemple suivant permet d'illustrer le problème que l'on peut rencontrer avec une clé technique :

```
CREATE TABLE etats (
 id     integer PRIMARY KEY,
 etat   varchar(10) NOT NULL UNIQUE
);

INSERT INTO etats
 VALUES (1, 'Ouvert'),
        (2, 'Résolu'),
        (3, 'Fermé');

CREATE TABLE tickets (
 id       serial PRIMARY KEY,
 id_etat  integer NOT NULL REFERENCES etats (id),
 titre    varchar(250) NOT NULL
);

-- génération d'un jeu de données où la grande majorité des tickets sont fermés
INSERT INTO tickets (id_etat, titre)
SELECT CASE WHEN i < 195000 THEN 3
            WHEN i < 198000 THEN 2
            ELSE 1
        END AS id_etat,
        md5(i ::text) AS titre
    FROM generate_series(1, 200000) i;

CREATE INDEX ON tickets (id_etat);

ANALYZE tickets;
ANALYZE etats;
```

Pour récupérer les tickets fermés, on utilisera la requête suivante :

```
SELECT *
    FROM tickets t
    JOIN etats e ON (t.id_etat = e.id)
 WHERE etat = 'Fermé';
```

Le plan de la requête ci-dessous montre essentiellement deux choses :

- le passage par l'index sur la table `tickets` nécessite la lecture de 2359 blocs, soit 18 Mo

- une mauvaise estimation de la cardinalité lors de la jointure entre la table `tickets` et `etats`

```
EXPLAIN (ANALYZE, BUFFERS, TIMING OFF)
SELECT *
FROM tickets t
JOIN etats e ON (t.id_etat = e.id)
WHERE etat = 'Fermé';
                    QUERY PLAN
-------------------------------------------------------------
 Nested Loop
       (cost=1253.09..4624.13 rows=66667 width=52)
       (actual rows=194999 loops=1)
   Buffers : shared hit=2360
   ->  Seq Scan on etats e
            (cost=0.00..1.04 rows=1 width=11)
            (actual rows=1 loops=1)
         Filter: ((etat) ::text = 'Fermé'::text)
         Rows Removed by Filter: 2
         Buffers : shared hit=1
   ->  Bitmap Heap Scan on tickets t
               (cost=1253.09..3956.43 rows=66667 width=41)
               (actual rows=194999 loops=1)
         Recheck Cond : (id_etat = e.id)
         Heap Blocks : exact=1823
         Buffers : shared hit=2359
         ->  Bitmap Index Scan on tickets_id_etat_idx
               (cost=0.00..1236.42 rows=66667 width=0)
               (actual rows=194999 loops=1)
             Index Cond : (id_etat = e.id)
             Buffers : shared hit=536
 Planning time: 0.127 ms
 Execution time: 31.423 ms
```

Si l'on procède différemment, en utilisant la valeur de `id_etat` correspondant aux tickets fermés, nous pouvons éviter d'utiliser une jointure. L'estimation est plus juste et nous voyons que le plan change pour utiliser une lecture séquentielle de la table tickets (opération `Seq Scan`) qui est plus efficace en termes de temps d'accès et de volume de données lu :

```
SELECT id FROM etats WHERE etat = 'Fermé';
 id
----
  3
(1 row)

EXPLAIN (ANALYZE, BUFFERS, TIMING OFF)
```

```
SELECT *
    FROM tickets t
    JOIN etats e ON (t.id_etat = e.id)
 WHERE id_etat = 3;
                     QUERY PLAN
-----------------------------------------------------------
 Nested Loop
       (cost=0.00..6324.71 rows=195367 width=52)
       (actual rows=194999 loops=1)
   Buffers : shared hit=1871
   ->  Seq Scan on etats e
             (cost=0.00..1.04 rows=1 width=11)
             (actual rows=1 loops=1)
         Filter: (id = 3)
         Rows Removed by Filter: 2
         Buffers : shared hit=1
   ->  Seq Scan on tickets t
              (cost=0.00..4370.00 rows=195367 width=41)
              (actual rows=194999 loops=1)
         Filter: (id_etat = 3)
         Rows Removed by Filter: 5001
         Buffers : shared hit=1870
 Planning time: 0.071 ms
 Execution time: 25.993 ms
(12 rows)
```

Le phénomène présenté est exacerbé sur une table de plus forte volumétrie et peut entraîner une différence très nette de performance entre les deux approches.

Si l'on utilise la colonne `etat` de la table `etats` comme clé primaire :

```
CREATE TABLE etats (
 etat   varchar(10) PRIMARY KEY
);

INSERT INTO etats
 VALUES ('Ouvert'),
        ('Résolu'),
        ('Fermé');

CREATE TABLE tickets (
 id       serial PRIMARY KEY,
 etat     varchar(10) NOT NULL REFERENCES etats (etat),
 titre    varchar(250) NOT NULL
);

-- génération d'un jeu de données où la grande majorité des tickets sont fermés
```

```
INSERT INTO tickets (etat, titre)
SELECT CASE WHEN i < 195000 THEN 'Fermé'
            WHEN i < 198000 THEN 'Résolu'
            ELSE 'Ouvert'
        END AS id_etat,
        md5(i ::text) AS titre
    FROM generate_series(1, 200000) i;

CREATE INDEX ON tickets (etat);

ANALYZE tickets;
ANALYZE etats;
```

Nous pourrons naturellement accéder aux tickets ouverts comme ceci (sans jointure) :

```
EXPLAIN (ANALYZE, BUFFERS, TIMING OFF)
SELECT * FROM tickets WHERE etat = 'Ouvert';
                QUERY PLAN
-----------------------------------------------
 Index Scan using tickets_etat_idx on tickets
       (cost=0.42..83.36 rows=2040 width=44)
       (actual rows=2001 loops=1)
   Index Cond : ((etat) ::text = 'Ouvert'::text)
   Buffers : shared hit=20 read=9
 Planning time: 0.059 ms
 Execution time: 0.519 ms
(5 rows)
```

Si l'on accède aux tickets fermés, le plan utilise la lecture séquentielle attendue :

```
EXPLAIN (ANALYZE, BUFFERS, TIMING OFF)
SELECT * FROM tickets WHERE etat = 'Fermé';
                    QUERY PLAN
-------------------------------------------------
 Seq Scan on tickets
       (cost=0.00..4370.00 rows=194980 width=44)
       (actual rows=194999 loops=1)
   Filter: ((etat) ::text = 'Fermé'::text)
   Rows Removed by Filter: 5001
   Buffers : shared hit=1870
 Planning time: 0.050 ms
 Execution time: 24.827 ms
(6 rows)
```

BIBLIOGRAPHIE

Documentation officielle de PostgreSQL :
https://www.postgresql.org/docs/

Version française
https://docs.postgresql.fr/

Manuels de formation de Dalibo
https://www.dalibo.com/formations

***The World and the Machine*, Michael Jackson**
version en ligne http://users.mct.open.ac.uk/mj665/icse17kn.pdf

***The Art of SQL*, Stéphane Faroult,**
ISBN-13 : 978-0596008949

***Refactoring SQL Applications*, Stéphane Faroult,**
ISBN-13 : 978-0596514976

***SQL Performance Explained*, Markus Winand**
https://use-the-index-luke.com/fr
FR : ISBN-13 : 978-3950307832
EN : ISBN-13 : 978-3950307825

***Introduction aux bases de données*, Chris Date**
FR : ISBN-13 : 978-2711748389 (8e édition)
EN : ISBN-13 : 978-0321197849

Vidéos de Stéphane Faroult,
Chaine https://www.youtube.com/channel/UCW6zsYGFckfczPKUUVdvYjg
Vidéo 1 https://www.youtube.com/watch?v=40Lnoyv-sXg&list=PL767434BC92D459A7
Vidéo 2 https://www.youtube.com/watch?v=GbZgnAINjUw&list=PL767434BC92D459A7
Vidéo 3 https://www.youtube.com/watch?v=y70FmugnhPU&list=PL767434BC92D459A7

Conférence sur les contraintes au PGDay Paris 2018, *Constraints : a Developer's Secret Weapon*
https://www.postgresql.eu/events/pgdayparis2018/sessions/session/1835/slides/70/2018-03-15%20constraints%20a%20developers%20secret%20weapon%20pgday%20paris.pdf
Article de blog associé, *Database constraints in Postgres : The last line of defense* :
https://www.citusdata.com/blog/2018/03/19/postgres-database-constraints/

***Mastering PostgreSQL in Application Development*, Dimitri Fontaine**
https://masteringpostgresql.com/

***Database Normalization and Primary Keys*, Dimitri Fontaine**
https://tapoueh.org/blog/2018/03/database-normalization-and-primary-keys/

Database Modelization Anti-Patterns
https://tapoueh.org/blog/2018/03/database-modelization-anti-patterns

***SQL avancé*, Joe Celko**
ISBN-13 : 978-2711786503
***Bases de données de la modélisation au SQL*, Laurent Audibert,**
ISBN-13 : 978-2729851200
Wikipédia *Forme normale*
https://fr.wikipedia.org/wiki/Forme_normale_(bases_de_donn%C3%A9es_relationnelles)
Wikipedia *Algèbre relationnelle*
https://fr.wikipedia.org/wiki/Alg%C3%A8bre_relationnelle