Module S65

SQL : Ce qu’il ne faut pas faire

25.09

Table des matières

Sur ce document

Formation	Module S65
Titre	SQL : Ce qu'il ne faut pas faire
Révision	25.09
PDF	https://dali.bo/s65_pdf
EPUB	https://dali.bo/s65_epub
HTML	https://dali.bo/s65_html
Slides	https://dali.bo/s65_slides
TP	https://dali.bo/s65_tp
TP (solutions)	https://dali.bo/s65_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

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

Marques déposées

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

Versions de PostgreSQL couvertes

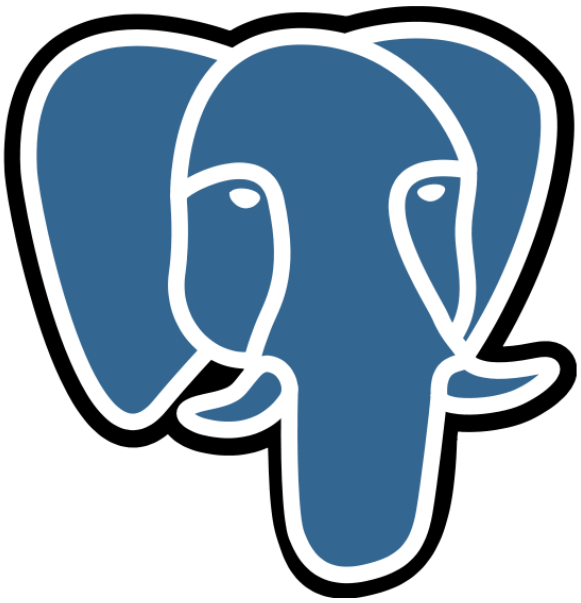
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ SQL : Ce qu’il ne faut pas faire



1.1 DES MAUVAISES PRATIQUES

- Modélisation
- Écriture de requêtes
- Conception de l'application

Cette partie présente différents problèmes fréquemment rencontrés et leurs solutions. Elles ont trait aussi bien à des problèmes courants qu'à des mauvaises pratiques.

1.2 PROBLÈMES DE MODÉLISATION

- Rappels sur le modèle relationnel
- Formes normales
- Atomicité !

1.2.1 Que veut dire « relationnel » ?

- PostgreSQL est un SGBD-R, un système de gestion de bases de données relationnel
- Le schéma est d'une importance capitale
- « Relationnel » n'est pas « relation entre tables »
- Les tables SONT les relations (entre attributs)

Contrairement à une idée assez fréquemment répandue, le terme relationnel ne désigne pas le fait que les tables soient liées entre elles. Les « tables » SONT les relations. On fait référence ici à l'algèbre relationnelle, inventée en 1970 par Edgar Frank Codd.

Les bases de données dites relationnelles n'implémentent habituellement pas exactement cet algèbre, mais en sont très proches. Le langage SQL, entre autres, ne respecte pas l'algèbre relationnelle. Le sujet étant vaste et complexe, il ne sera pas abordé ici. Si vous voulez approfondir le sujet, le livre *Introduction aux bases de données* de Chris J. Date, est un des meilleurs ouvrages sur l'algèbre relationnelle et les déficiences du langage SQL à ce sujet.

1.2.2 Quelques rappels sur le modèle relationnel

- Le but est de modéliser un ensemble de faits
- Le modèle relationnel a été introduit à l'époque des bases de données hiérarchiques
  - pointeur : incohérence à terme
  - formalisme : relations, modélisation évitant les incohérences suite à modification
  - formes normales
- Un modèle n'est qu'un modèle : il ne traduit pas la réalité, simplement ce qu'on souhaite en représenter
- Identifier rapidement les problèmes les plus évidents

Le modèle relationnel est apparu suite à un constat : les bases de données de l'époque (hiérarchiques) reposaient sur la notion de pointeur. Une mise à jour pouvait donc facilement casser le modèle : doublons simples, données pointant sur du « vide », doublons incohérents entre eux, etc.

Le modèle relationnel a donc été proposé pour remédier à tous ces problèmes. Un système relationnel repose sur le concept de relation (table en SQL). Une relation est un ensemble de faits. Chaque fait est identifié par un identifiant (clé naturelle). Le fait lie cet identifiant à un certain nombre d'attributs. Une relation ne peut donc pas avoir de doublon.

La modélisation relationnelle étant un vaste sujet en soi, nous n'allons pas tout détailler ici, mais plutôt rappeler les points les plus importants.

1.2.3 Formes normales

Il existe une définition mathématique précise de chacune des 7 formes normales.
- La troisième forme normale peut toujours être atteinte
- La forme suivante (forme normale de Boyce-Codd, ou FNBC) ne peut pas toujours être atteinte
- La cible est donc habituellement la `3FN`
- Chris Date :
  - « *Chaque attribut dépend de la clé, de TOUTE la clé, et QUE de la clé* »
  - « *The key, the whole key, nothing but the key* »

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).

Si vos tables vérifient déjà ces trois points, votre modélisation est probablement assez bonne.

Voir l'article wikipedia[1] présentant l'ensemble des formes normales.

[1] https://fr.wikipedia.org/wiki/Forme_normale_(bases_de_donn%C3%A9es_relationnelles)

1.3 ATOMICITÉ

- Un attribut (colonne) doit être atomique :
  - Modifier l'attribut sans en toucher un autre
  - Donnée correcte (délicat !)
  - Recherche efficace : accédé en entier dans une clause `WHERE`
- Non respect = violation de la première forme normale

L'exemple suivant utilise une table `voiture`. Les deux tables `voitures` et `voitures_ecv` peuvent être téléchargées installées comme suit :

```
createdb voitures
curl -kL https://dali.bo/tp_voitures -o /tmp/voitures.dmp
pg_restore -d voitures /tmp/voitures.dmp
# un message sur le schéma public préexistant est normal
```

Ne pas oublier d'effectuer un `VACUUM ANALYZE`.

1.3.1 Atomicité - mauvais exemple

Immatriculation	Modèle	Caractéristiques
NH-415-DG	twingo	4 roues motrices,toit ouvrant, climatisation
EO-538-WR	clio	boite automatique,abs,climatisation

```
INSERT INTO voitures
VALUES ('AD-057-GD','clio','toit ouvrant,abs');
```

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

```
SELECT * FROM voitures
WHERE caracteristiques LIKE '%abs%'
```

ce qui sera évidemment très inefficace.

Par ailleurs, on n'a évidemment aucun contrôle sur ce qui est mis dans le champ `caractéristiques`, ce qui est la garantie de données incohérentes au bout de quelques jours (heures ?) d'utilisation. Par exemple, rien n'empêche d'ajouter une ligne avec des caractéristiques similaires légèrement différentes, comme « ABS », « boîte automatique ».

Ce modèle ne permet donc pas d'assurer la cohérence des données.

1.3.2 Atomicité - propositions

— Champs dédiés :

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

— Plusieurs valeurs : contrainte `CHECK` /enum/table de référence
— Beaucoup de champs : clé/valeur (plusieurs formes possibles)

Une alternative plus fiable est de rajouter des colonnes `boolean quatre_roues_motrices`, `boolean abs`, `varchar couleur`. C'est ce qui est à privilégier si le nombre de caractéristiques est fixe et pas trop important.

Dans le cas où un simple booléen ne suffit pas, un champ avec une contrainte est possible. Il y a plusieurs méthodes :

— une contrainte simple :

```
ALTER TABLE voitures ADD COLUMN couleur text
CHECK (couleur IN ('rouge','bleu','vert')) ;
```

— un type « énumération[2] » :

```
CREATE TYPE color AS ENUM ('bleu', 'rouge', 'vert') ;
ALTER TABLE voitures ADD COLUMN couleur color ;
```

(Les énumérations ne sont pas adaptées à des modifications fréquentes et nécessitent parfois un transtypage vers du `text`).

— une table de référence avec contrainte, c'est le plus flexible :

```
CREATE TABLE couleurs (
    couleur_id int  PRIMARY KEY,
    couleur    text
) ;
ALTER TABLE voitures ADD COLUMN couleur_id REFERENCES couleurs ;
```

Ce modèle facilite les recherches et assure la cohérence. L'indexation est facilitée, et les performances ne sont pas dégradées, bien au contraire.

Dans le cas où le nombre de propriétés n'est pas aussi bien défini qu'ici, ou est grand, même un modèle clé-valeur dans une associée vaut mieux que l'accumulation de propriétés dans un champ texte. Même une simple table des caractéristiques est plus flexible (voir le TP).

[2] https://docs.postgresql.fr/current/datatype-enum.html

Un modèle clé/valeur existe sous plusieurs variantes (table associée, champs `hstore` ou JSON…) et a ses propres inconvénients, mais il offre au moins plus de flexibilité et de possibilités d'indexation ou de validation des données. Ce sujet est traité plus loin.

1.4 CONTRAINTES ABSENTE

- Parfois (souvent ?) ignorées pour diverses raisons :
  - faux gains de performance
  - flexibilité du modèle de données
  - compatibilité avec d'autres SGBD (MySQL/MyISAM…)
  - commodité de développement

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, mais surtout une garantie contre de très coûteuses corruption de données logiques.

L'absence de contraintes a souvent des conséquences catastrophiques.

1.4.1 Conséquences de l'absence de contraintes

- Conséquences
  - problèmes d'intégrité des données
  - fonctions de vérification de cohérence des données
- Les contraintes sont utiles à l'optimiseur :
  - déterminent l'unicité des valeurs
  - éradiquent des lectures de tables inutiles sur des `LEFT JOIN`
  - utilisent les contraintes `CHECK` pour exclure une partition

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 *race condition*[3] en l'absence de clé étrangère.

Imaginez le scénario suivant :

- la transaction x1 s'assure que la donnée est présente dans la table t1 ;
- la transaction x2 supprime la donnée précédente dans la table t1 ;
- la transaction x1 insère une ligne dans la table t2 faisant référence à la ligne de t1 qu'elle pense encore présente.

Ce cas est très facilement gérable pour un moteur de base de donnée si une clé étrangère existe. Redévelopper ces mêmes contrôles dans la couche applicative sera toujours plus coûteux en terme de performance, voire impossible à faire dans certains cas sans passer par la base de donnée elle-même (multiples serveurs applicatifs accédant à la même base de donnée).

[3] Situation où deux sessions ou plus modifient des données en tables au même moment.

Il peut s'ensuivre des calculs d'agrégats faux et des problèmes applicatifs de toute sorte. Souvent, plutôt que de corriger le modèle de données, des fonctions de vérification de la cohérence des données seront mises en place, entraînant ainsi un travail supplémentaire pour trouver et corriger les incohérences.

Lorsque ces problèmes d'intégrité seront détectés, il s'en suivra 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, mais sous une forme différente.

De plus, 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.

1.4.2 Suspension des contraintes le temps d'une transaction

— Solution :
  — contraintes `DEFERRABLE` !

Parfois, les clés étrangères sont supprimées simplement parce que des transactions sont en erreur car des données sont insérées dans une table fille sans avoir alimenté la table mère. Des identifiants de clés étrangères de la table fille sont absents de la table mère, entraînant l'arrêt en erreur de la transaction. Il est possible de contourner cela en différant la vérification des contraintes d'intégrité à la fin de la transaction

Une contrainte `DEFERRABLE` associée à un `SET CONSTRAINT … DEFERRED` n'est vérifiée que lors du `COMMIT`. Elle ne gêne donc pas le développeur, qui peut insérer les données dans l'ordre qu'il veut ou insérer temporairement des données incohérentes. Ce qui compte est que la situation soit saine à la fin de la transaction, quand les données seront enregistrées et deviendront visibles par les autres sessions.

L'exemple ci-dessous montre l'utilisation de la vérification des contraintes d'intégrité en fin de transaction.

```
CREATE TABLE mere (id integer, t text);

CREATE TABLE fille (id integer, mere_id integer, t text);

ALTER TABLE mere ADD CONSTRAINT pk_mere PRIMARY KEY (id);

ALTER TABLE fille
```

```
  ADD CONSTRAINT fk_mere_fille
      FOREIGN KEY (mere_id)
      REFERENCES mere (id)
         MATCH FULL
         ON UPDATE NO ACTION
         ON DELETE CASCADE
         DEFERRABLE;
```

La transaction insère d'abord les données dans la table fille, puis ensuite dans la table mère :

```
BEGIN ;
SET CONSTRAINTS ALL DEFERRED ;

INSERT INTO fille (id, mere_id, t) VALUES (1, 1, 'val1');
INSERT INTO fille (id, mere_id, t) VALUES (2, 2, 'val2');

INSERT INTO mere (id, t) VALUES (1, 'val1'), (2, 'val2');

COMMIT;
```

Sans le `SET CONSTRAINTS ALL DEFERRED`, le premier ordre serait tombé en erreur.

1.5 STOCKAGE ENTITÉ-CLÉ-VALEUR

- Entité-Attribut-Valeur (ou Entité-Clé-Valeur)
- Quel but ?
  - flexibilité du modèle de données
  - adapter sans délai ni surcoût le modèle de données
- Conséquences :
  - création d'une table : `identifiant / nom_attribut / valeur`
  - requêtes abominables et coûteuses

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 caractère, 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.

1.5.1 Stockage Entité-Clé-Valeur : exemple

Comment lister tous les DBA ?

id_pers	nom_attr	val_attr
1	nom	Prunelle
1	prenom	Léon
1	telephone	0123456789
1	fonction	dba

1.5.2 Stockage Entité-Clé-Valeur : requête associée

```
SELECT id, att_nom.val_attr       AS nom,
           att_prenom.val_attr    AS prenom,
           att_telephone.val_attr AS tel
FROM personnes p
JOIN personne_attributs AS att_nom
 ON (p.id=att_nom.id_pers AND att_nom.nom_attr='nom')
JOIN personne_attributs AS att_prenom
 ON (p.id=att_prenom.id_pers AND att_prenom.nom_attr='prenom')
JOIN personne_attributs AS att_telephone
 ON (p.id=att_telephone.id_pers AND att_telephone.nom_attr='telephone')
JOIN personne_attributs AS att_fonction
 ON (p.id=att_fonction.id_pers AND att_fonction.nom_attr='fonction')
WHERE att_fonction.val_attr='dba';
```

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.

1.5.3 Stockage Entité-Clé-Valeur, hstore, JSON

- Solutions :
  - revenir sur la conception du modèle de données
  - utiliser un type de données plus adapté : `hstore`, `jsonb`
- On économise jointures et place disque.

Lorsque de telles solutions sont déployées 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 où le nombre de champs est *vraiment* élevé et variable, 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`. On conserve ainsi l'intégrité des données (on n'a qu'une ligne par personne), on évite de très nombreuses jointures source d'erreurs et de ralentissements, et même de la place disque.

De plus, 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"

SELECT id, attributs->'prenom' AS prenom FROM personnes;

 id |  prenom
----+----------
  1 | Léon
  2 | Gaston
  3 |

SELECT id, attributs->'nom' AS nom FROM personnes;

 id |     nom
----+-------------
  1 | Prunelle
  2 | Lagaffe
  3 | DeMaesmaker
```

Le principe du JSON est similaire.

1.6 ATTRIBUTS MULTICOLONNES

- Pourquoi
  - stocker plusieurs attributs pour une même ligne
  - exemple : les différents numéros de téléphone d'une personne
- Pratique courante
  - ex : `telephone_1`, `telephone_2`
- Conséquences
  - et s'il faut rajouter encore une colonne ?
  - maîtrise de l'unicité des valeurs ?
  - requêtes complexes à maintenir
- Solutions
  - créer une table dépendante
  - ou un type tableau

Dans certains cas, le modèle de données doit être étendu pour pouvoir stocker des données complémentaires. Un exemple typique est une table qui stocke les informations pour contacter une personne. Une table `personnes` ou `contacts` possède une colonne `telephone` qui permet de stocker le numéro de téléphone d'une personne. Or, une personne peut disposer de plusieurs numéros. Le premier réflexe est souvent de créer une seconde colonne `telephone_2` pour stocker un numéro de téléphone complémentaire. S'en suit une colonne `telephone_3` voire `telephone_4` en fonction des besoins.

Dans de tels cas, les requêtes deviennent plus complexes à maintenir et il est difficile de garantir l'unicité des valeurs stockées pour une personne car l'écriture des contraintes d'intégrité devient de plus en plus complexe au fur et à mesure que l'on ajoute une colonne pour stocker un numéro.

La solution la plus pérenne pour gérer ce cas de figure est de créer une table de dépendance qui est dédiée au stockage des numéros de téléphone. Ainsi, la table `personnes` ne portera plus de colonnes `telephone`, mais une table `telephones` portera un identifiant référençant une personne et un numéro de téléphone. Ainsi, si une personne dispose de trois, quatre… numéros de téléphone, la table `telephones` comportera autant de lignes qu'il y a de numéros pour une personne.

Les différents numéros de téléphone seront obtenus par jointure entre la table `personnes` et la table `telephones`. L'application se chargera de l'affichage.

Ci-dessous, un exemple d'implémentation du problème où une table `telephones` dans laquelle plusieurs numéros seront stockés sur plusieurs lignes plutôt que dans plusieurs colonnes.

```
CREATE TABLE personnes (
  per_id SERIAL PRIMARY KEY,
  nom VARCHAR(50) NOT NULL,
  pnom VARCHAR(50) NOT NULL,
  ...
);
```

```
CREATE TABLE telephones (
  per_id INTEGER NOT NULL,
  numero VARCHAR(20),
  PRIMARY KEY (per_id, numero),
  FOREIGN KEY (per_id) REFERENCES personnes (per_id)
);
```

L'unicité des valeurs sera garantie à l'aide d'une contrainte d'unicité posée sur l'identifiant `per_id` et le numéro de téléphone.

Une autre solution consiste à utiliser un tableau pour représenter cette information. D'un point de vue conceptuel, le lien entre une personne et son ou ses numéros de téléphone est plus une « composition » qu'une réelle « relation » : le numéro de téléphone ne nous intéresse pas en tant que tel, mais uniquement en tant que détail d'une personne. On n'accédera jamais à un numéro de téléphone séparément : la table `telephones` donnée plus haut n'a pas de clé « naturelle », un simple rattachement à la table personnes par l'identifiant de la personne. Sans même parler de partitionnement, on gagnerait donc en performances en stockant directement les numéros de téléphone dans la table `personnes`, ce qui est parfaitement faisable sous PostgreSQL :

```
CREATE TABLE personnes (
  per_id SERIAL PRIMARY KEY,
  nom VARCHAR(50) NOT NULL,
  pnom VARCHAR(50) NOT NULL,
  numero VARCHAR(20)[]
);

-- Ajout d'une personne
INSERT INTO personnes (nom, pnom, numero)
     VALUES ('Simpson', 'Omer', '{0607080910}');

SELECT *
  FROM personnes;

 per_id |   nom   | pnom |    numero
--------+---------+------+--------------
      1 | Simpson | Omer | {0607080910}

-- Ajout d'un numéro de téléphone pour une personne donnée :
UPDATE personnes
   SET numero = numero || '{0102030420}'
 WHERE per_id = 1;

-- Vérification de l'ajout :
SELECT * FROM personnes;

 per_id |   nom   | pnom |          numero
--------+---------+------+--------------------------
      1 | Simpson | Omer | {0607080910,0102030420}

-- Séparation des éléments du tableau :
SELECT per_id, nom, pnom, unnest(numero) AS numero
  FROM personnes;

 per_id |   nom   | pnom |   numero
--------+---------+------+------------
```

```
1 | Simpson | Omer | 0607080910
1 | Simpson | Omer | 0102030420
```

1.7 NOMBREUSES LIGNES DE PEU DE COLONNES

- Énormément de lignes, peu de colonnes
  - Cas typique : séries temporelles
- Volumétrie augmentée par les entêtes
- Regrouper les valeurs dans un `ARRAY` ou un type composite
- Partitionner

Certaines applications, typiquement celles récupérant des données temporelles, stockent peu de colonnes (parfois juste date, capteur, valeur…) mais énormément de lignes.

Dans le modèle MVCC de PostgreSQL, chaque ligne utilise au bas mot 23 octets pour stocker `xmin`, `xmax` et les autres informations de maintenance de la ligne. On peut donc se retrouver avec un *overhead* représentant la majorité de la table. Cela peut avoir un fort impact sur la volumétrie :

```
CREATE TABLE valeurs_capteur (d timestamp, v smallint);
-- soit 8 + 2 = 10 octets de données utiles par ligne

--  100 valeurs chaque seconde pendant 100 000 s = 10 millions de lignes
INSERT INTO valeurs_capteur (d, v)
SELECT current_timestamp + (i%100000) * interval '1 s',
       (random()*200)::smallint
FROM   generate_series (1,10000000) i ;

SELECT pg_size_pretty(pg_relation_size ('valeurs_capteur')) ;

 pg_size_pretty
----------------
  422 MB
-- dont seulement 10 octets * 10 Mlignes = 100 Mo de données utiles
```

Il est parfois possible de regrouper les valeurs sur une même ligne au sein d'un `ARRAY`, ici pour chaque seconde :

```
CREATE TABLE valeurs_capteur_2 (d timestamp, tv smallint[]);

INSERT INTO valeurs_capteur_2
SELECT current_timestamp+  (i%100000) * interval '1 s' ,
       array_agg((random()*200)::smallint)
FROM   generate_series (1,10000000) i
GROUP BY 1 ;

SELECT pg_size_pretty(pg_relation_size ('valeurs_capteur_2'));

 pg_size_pretty
----------------
  25 MB
-- soit par ligne :
-- 23 octets d'entête + 8 pour la date + 100 * 2 octets de valeurs smallint
```

Dans cet exemple, on économise la plupart des entêtes de ligne, mais aussi les données redondantes (la date), et le coût de l'alignement des champs. Avec suffisamment de valeurs à stocker, une partie des données peut même se retrouver compressée dans la partie TOAST de la table.

La récupération des données se fait de manière à peine moins simple :

```
SELECT unnest(tv) FROM valeurs_capteur_2
WHERE  d = '2018-06-15 22:07:47.651295' ;
```

L'indexation des valeurs à l'intérieur du tableau nécessite un index GIN :

```
CREATE INDEX tvx ON valeurs_capteur_2 USING gin(tv);
```

```
EXPLAIN (ANALYZE) SELECT * FROM valeurs_capteur_2 WHERE '{199}' && tv ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on valeurs_capteur_2 (cost=311.60..1134.20 rows=40000 width=232)
                                    (actual time=8.299..20.460 rows=39792 loops=1)
   Recheck Cond: ('{199}'::smallint[] && tv)
   Heap Blocks: exact=3226
   ->  Bitmap Index Scan on tvx  (cost=0.00..301.60 rows=40000 width=0)
                           (actual time=7.723..7.723 rows=39792 loops=1)
         Index Cond: ('{199}'::smallint[] && tv)
 Planning time: 0.214 ms
 Execution time: 22.386 ms
```

Évidemment cette technique est à réserver aux cas où les données mises en tableau sont insérées et mises à jour ensemble.

Le maniement des tableaux est détaillé dans la documentation officielle[4].

Tout cela est détaillé et mesuré dans ce billet de Julien Rouhaud[5]. Il évoque aussi le cas de structures plus complexes : au lieu d'un `hstore` ou d'un `ARRAY`, on peut utiliser un type qui regroupe les différentes valeurs.

Une autre option, complémentaire, est le partitionnement. Il peut être géré manuellement (tables générées par l'applicatif, par date et/ou par source de données…) ou profiter des deux modes de partitionnement de PostgreSQL. Il n'affectera pas la volumétrie totale mais permet de gérer des partitions plus maniables. Il a aussi l'intérêt de ne pas nécessiter de modification du code pour lire les données.

[4] https://www.postgresql.org/docs/current/static/arrays.html
[5] https://rjuju.github.io/postgresql/2016/09/16/minimizing-tuple-overhead.html

1.8 TABLES AUX TRÈS NOMBREUSES COLONNES

Tables à plusieurs dizaines, voire centaines de colonnes :
- Les entités sont certainement trop grosses dans la modélisation
- Il y a probablement dépendance entre certaines colonnes (*Only the key*)
- On accède à beaucoup d'attributs inutiles (tout est stocké au même endroit)

Il arrive régulièrement de rencontrer des tables ayant énormément de colonnes (souvent à `NULL` d'ailleurs). Cela signifie qu'on modélise une entité ayant tous ces attributs (centaines d'attributs). Il est très possible que cette entité soit en fait composée de « sous-entités », qu'on pourrait modéliser séparément. On peut évidemment trouver des cas particuliers contraires, mais une table de ce type reste un bon indice.

Surtout si vous trouvez dans les dernières colonnes des attributs comme `attribut_supplementaire_1`…

1.9 CHOIX D'UN TYPE NUMÉRIQUE

- Pour : représenter des valeurs décimales
- Pratique courante :
  - `real` ou `double` (`float`)
  - `money`
  - … erreurs d'arrondis !
- Solution :
  - `numeric` pour les calculs précis (financiers notamment)

Certaines applications scientifiques se contentent de types flottants standards, car ils permettent d'encoder des valeurs plus importantes que les types entiers standards. En pratique, les types `float(x)` correspondent aux types `real` ou `double precision` de PostgreSQL.

Néanmoins, les types flottants sont peu précis, notamment pour les applications financières où une erreur d'arrondi n'est pas tolérable. Par exemple :

```
test=# CREATE TABLE comptes (compte_id serial PRIMARY KEY, solde float);
CREATE TABLE

test=# INSERT INTO comptes (solde) VALUES (100000000.1), (10.1), (10000.2),
(100000000000000.1);
INSERT 0 4

test=# SELECT SUM(solde) FROM comptes;
       sum
-----------------
 100000100010010
```

Le type `numeric` est alors généralement conseillé. Sa valeur est exacte et les calculs sont justes.

```
test=# CREATE TABLE comptes (compte_id serial PRIMARY KEY, solde numeric);
CREATE TABLE

test=# INSERT INTO comptes (solde) VALUES (100000000.1), (10.1), (10000.2),
(100000000000000.1);
INSERT 0 4

test=# SELECT SUM(solde) FROM comptes;
        sum
-------------------
 100000100010010.5
```

`numeric` (sans autre indication de précision) autorise même un calcul exact sans arrondi avec des ordres de grandeur très différents ; comme `SELECT 1e9999 + 1e-9999 ;`.

Paradoxalement, le type `money` n'est pas adapté aux montants financiers : sa manipulation implique de convertir en `numeric` pour éviter des erreurs d'arrondis. Autant utiliser directement `numeric` : si l'on ne mentionne pas la précision, elle est exacte.

Le type `numeric` paye sa précision par un stockage parfois plus important et par des calculs plus lents que ceux des types natifs comme les `intX` et les `floatX`.

Pour plus de détails, voir la documentation officielle :

— types à virgule flottante[6] ;
— type monétaire[7] ;
— type à prévision arbitraire[8].

[6] https://docs.postgresql.fr/current/datatype-numeric.html#DATATYPE-FLOAT
[7] https://docs.postgresql.fr/current/datatype-money.html
[8] https://docs.postgresql.fr/current/datatype-numeric.html#DATATYPE-NUMERIC-DECIMAL

1.10 COLONNE DE TYPE VARIABLE

Plus rarement, on rencontre aussi :
- Une colonne de type `varchar` contenant
  - quelquefois un entier
  - quelquefois une date
  - un `NULL`
  - une chaîne autre
  - etc.
- À éviter comme la peste !
- Plusieurs sens = plusieurs champs

On rencontre parfois ce genre de choses :

Immatriculation Camion	Numero de tournee
TP-108-AX	12
TF-112-IR	ANNULÉE

avec bien sûr une table `tournée` décrivant la tournée elle-même, avec une clé technique numérique.

Cela pose un gros problème de modélisation : la colonne a un type de contenu qui dépend de l'information qu'elle contient. On va aussi avoir un problème de performance en joignant cette chaîne à la clé numérique de la table `tournée`. Le moteur n'aura que deux choix : convertir la chaîne 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'identifiant de tournée par index impossible. D'où un parcours complet (*Seq Scan*) de la table `tournée` à chaque accès et des performances qui décroissent au fur et à mesure que la table grossit.

La solution est une supplémentaire (un booléen `tournee_ok` par exemple).

Un autre classique est le champ date stocké au format texte. Le format correct de cette date ne peut être garanti par la base, ce qui mène systématiquement à des erreurs de conversion si un humain est impliqué. Dans un environnement international où l'on mélange DD-MM-YYYY et MM-DD-YYYY, un rattrapage manuel est même illusoire. Les calculs de date sont évidemment impossibles.

1.11 PROBLÈMES COURANTS D'ÉCRITURE DE REQUÊTES

- Utilisation de `NULL`
- Ordre implicite des colonnes
- Requêtes spaghetti
- Moteur de recherche avec `LIKE`

Le langage SQL est généralement méconnu, ce qui amène à l'écriture de requêtes peu performantes, voire peu pérennes.

1.12 NULL

- `NULL` signifie habituellement :
  - Valeur non renseignée
  - Valeur inconnue
- Absence d'information
- Une table remplie de `NULL` est habituellement signe d'un problème de modélisation.
- `NOT NULL` recommandé

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 `NULL` 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. »

Il ne ne s'agit pas de remplacer ce `NULL` par des valeurs « magiques » (par exemple -1 pour « Non renseigné » , cela ne ferait que complexifier le code) mais de se demander si `NULL` a une vraie signification.

1.13 ORDRE IMPLICITE DES COLONNES

- Objectif
  - s'économiser d'écrire la liste des colonnes dans une requête
- Problèmes
  - si l'ordre des colonnes change, les résultats changent
  - résultats faux
  - données corrompues
- Solutions
  - nommer les colonnes impliquées

Le langage SQL permet de s'appuyer sur l'ordre physique des colonnes d'une table. Or, faire confiance à la base de données pour conserver cet ordre physique peut entraîner de graves problèmes applicatifs en cas de changements. Dans le meilleur des cas, l'application ne fonctionnera plus, ce qui permet d'éviter les corruptions de données silencieuses, où une colonne prend des valeurs destinées normalement à être stockées dans une autre colonne. Si l'application continue de fonctionner, elle va générer des résultats faux et des incohérences d'affichage.

Par exemple, l'ordre des colonnes peut changer notamment lorsque certains ETL sont utilisés pour modifier le type d'une colonne `varchar(10)` en `varchar(11)`. Par exemple, pour la colonne `username`, l'ETL Kettle génère les ordres suivants :

```
ALTER TABLE utilisateurs ADD COLUMN username_KTL VARCHAR(11);
UPDATE utilisateurs SET username_KTL=username;
ALTER TABLE utilisateurs DROP COLUMN username;
ALTER TABLE utilisateurs RENAME username_KTL TO username
```

Il génère des ordres SQL inutiles et consommateurs d'entrées/sorties disques car il doit générer des ordres SQL compris par tous les SGBD du marché. Or, tous les SGBD ne permettent pas de changer le type d'une colonne aussi simplement que dans PostgreSQL. PostgreSQL, lui, ne permet pas de changer l'ordre d'apparition des colonnes.

C'est pourquoi il est préférable de lister explicitement les colonnes dans les ordres `INSERT` et `SELECT`, afin de garder un ordre d'insertion déterministe.

Exemples

Exemple de modification du schéma pouvant entraîner des problèmes d'insertion si les colonnes ne sont pas listées explicitement :

```
CREATE TABLE insere (id integer PRIMARY KEY, col1 varchar(5), col2 integer);

INSERT INTO insere VALUES (1, 'XX', 10);

SELECT * FROM insere ;

 id | col1 | col2
----+------+------
  1 | XX   |   10
```

```
ALTER TABLE insere ADD COLUMN col1_tmp varchar(6);
UPDATE insere SET col1_tmp = col1;
ALTER TABLE insere DROP COLUMN col1;
ALTER TABLE insere RENAME COLUMN col1_tmp TO col1;

INSERT INTO insere VALUES (2, 'XXX', 10);

ERROR:  invalid input syntax for integer: "XXX"
LINE 1: INSERT INTO insere VALUES (2, 'XXX', 10);
                                      ^

INSERT INTO insere (id, col1, col2) VALUES (2, 'XXX', 10);

SELECT * FROM insere ;

 id | col2 | col1
----+------+------
  1 |   10 | XX
  2 |   10 | XXX
```

L'utilisation de `SELECT *` à la place d'une liste explicite est une erreur similaire. Le nombre de colonnes peut brutalement varier. De plus, toutes les colonnes sont rarement utilisées dans un tel cas, ce qui provoque un gaspillage de ressources.

1.14 CODE SPAGHETTI

Le problème est similaire à tout autre langage :

- Code spaghetti pour le SQL
  - Écriture d'une requête à partir d'une autre requête
  - Ou évolution d'une requête au fil du temps avec des ajouts
- Non optimisable
- Vite ingérable
  - Ne pas la patcher !
  - Ne pas hésiter à reprendre la requête à zéro, en repensant sa sémantique
  - Souvent, un changement de spécification est un changement de sens, au niveau relationnel, de la requête

Un exemple (sous Oracle) :

```
SELECT  Article.datem                                AS Article_1_9,
    Article.degre_alcool                            AS Article_1_10,
    Article.id                                      AS Article_1_19,
    Article.iddf_categor                            AS Article_1_20,
    Article.iddp_clsvtel                            AS Article_1_21,
    Article.iddp_cdelist                            AS Article_1_22,
    Article.iddf_cd_prix                            AS Article_1_23,
    Article.iddp_agreage                            AS Article_1_24,
    Article.iddp_codelec                            AS Article_1_25,
    Article.idda_compo                              AS Article_1_26,
    Article.iddp_comptex                            AS Article_1_27,
    Article.iddp_cmptmat                            AS Article_1_28,
    Article.idda_articleparent                      AS Article_1_29,
    Article.iddp_danger                             AS Article_1_30,
    Article.iddf_fabric                             AS Article_1_33,
    Article.iddp_marqcom                            AS Article_1_34,
    Article.iddp_nomdoua                            AS Article_1_35,
    Article.iddp_pays                               AS Article_1_37,
    Article.iddp_recept                             AS Article_1_40,
    Article.idda_unalvte                            AS Article_1_42,
    Article.iddb_sitecl                            AS Article_1_43,
    Article.lib_caisse                              AS Article_1_49,
    Article.lib_com                                 AS Article_1_50,
    Article.maj_en_attente                          AS Article_1_61,
    Article.qte_stk                                 AS Article_1_63,
    Article.ref_tech                                AS Article_1_64,
    1                                               AS Article_1_70,
    CASE
      WHEN (SELECT COUNT(MA.id)
            FROM   da_majart MA
                   join da_majmas MM
                     ON MM.id = MA.idda_majmas
                   join gt_tmtprg TMT
                     ON TMT.id = MM.idgt_tmtprg
                   join gt_prog PROG
                     ON PROG.id = TMT.idgt_prog
```

```
        WHERE  idda_article = Article.id
           AND TO_DATE(TO_CHAR(PROG.date_lancement, 'DDMMYYYY')
                       || TO_CHAR(PROG.heure_lancement, ' HH24:MI:SS'),
                       'DDMMYYYY HH24:MI:SS') >= SYSDATE) >= 1 THEN 1
  ELSE 0
END                                          AS Article_1_74,
Article.iddp_compnat                         AS Article_2_0,
Article.iddp_modven                          AS Article_2_1,
Article.iddp_nature                          AS Article_2_2,
Article.iddp_preclin                         AS Article_2_3,
Article.iddp_raybala                         AS Article_2_4,
Article.iddp_sensgrt                         AS Article_2_5,
Article.iddp_tcdtfl                          AS Article_2_6,
Article.iddp_unite                           AS Article_2_8,
Article.idda_untgrat                         AS Article_2_9,
Article.idda_unpoids                         AS Article_2_10,
Article.iddp_unilogi                         AS Article_2_11,
ArticleComplement.datem                      AS ArticleComplement_5_6,
ArticleComplement.extgar_depl                AS ArticleComplement_5_9,
ArticleComplement.extgar_mo                  AS ArticleComplement_5_10,
ArticleComplement.extgar_piece               AS ArticleComplement_5_11,
ArticleComplement.id                         AS ArticleComplement_5_20,
ArticleComplement.iddf_collect               AS ArticleComplement_5_22,
ArticleComplement.iddp_gpdtcul               AS ArticleComplement_5_23,
ArticleComplement.iddp_support               AS ArticleComplement_5_25,
ArticleComplement.iddp_typcarb               AS ArticleComplement_5_27,
ArticleComplement.mt_ext_gar                 AS ArticleComplement_5_36,
ArticleComplement.pres_cpt                   AS ArticleComplement_5_44,
GenreProduitCulturel.code                    AS GenreProduitCulturel_6_0,
Collection.libelle                           AS Collection_8_1,
Gtin.date_dern_vte                           AS Gtin_10_0,
Gtin.gtin                                    AS Gtin_10_1,
Gtin.id                                      AS Gtin_10_3,
Fabricant.code                               AS Fabricant_14_0,
Fabricant.nom                                AS Fabricant_14_2,
ClassificationVenteLocale.niveau1            AS ClassificationVenteL_16_2,
ClassificationVenteLocale.niveau2            AS ClassificationVenteL_16_3,
ClassificationVenteLocale.niveau3            AS ClassificationVenteL_16_4,
ClassificationVenteLocale.niveau4            AS ClassificationVenteL_16_5,
MarqueCommerciale.code                       AS MarqueCommerciale_18_0,
MarqueCommerciale.libellelong                AS MarqueCommerciale_18_4,
Composition.code                             AS Composition_20_0,
CompositionTextile.code                      AS CompositionTextile_21_0,
AssoArticleInterfaceBalance.datem            AS AssoArticleInterface_23_0,
AssoArticleInterfaceBalance.lib_envoi        AS AssoArticleInterface_23_3,
AssoArticleInterfaceCaisse.datem             AS AssoArticleInterface_24_0,
AssoArticleInterfaceCaisse.lib_envoi         AS AssoArticleInterface_24_3,
NULL                                         AS TypeTraitement_25_0,
NULL                                         AS TypeTraitement_25_1,
RayonBalance.code                            AS RayonBalance_31_0,
RayonBalance.max_cde_article                 AS RayonBalance_31_5,
RayonBalance.min_cde_article                 AS RayonBalance_31_6,
TypeTare.code                                AS TypeTare_32_0,
GrilleDePrix.datem                           AS GrilleDePrix_34_1,
GrilleDePrix.libelle                         AS GrilleDePrix_34_3,
FicheAgreage.code                            AS FicheAgreage_38_0,
Codelec.iddp_periact                         AS Codelec_40_1,
```

```
    Codelec.libelle                                AS Codelec_40_2,
    Codelec.niveau1                                AS Codelec_40_3,
    Codelec.niveau2                                AS Codelec_40_4,
    Codelec.niveau3                                AS Codelec_40_5,
    Codelec.niveau4                                AS Codelec_40_6,
    PerimetreActivite.code                         AS PerimetreActivite_41_0,
    DonneesPersonnalisablesCodelec.gestionreftech AS DonneesPersonnalisab_42_0,
    ClassificationArticleInterne.id                AS ClassificationArticl_43_0,
    ClassificationArticleInterne.niveau1           AS ClassificationArticl_43_2,
    DossierCommercial.id                           AS DossierCommercial_52_0,
    DossierCommercial.codefourndc                  AS DossierCommercial_52_1,
    DossierCommercial.anneedc                      AS DossierCommercial_52_3,
    DossierCommercial.codeclassdc                  AS DossierCommercial_52_4,
    DossierCommercial.numversiondc                 AS DossierCommercial_52_5,
    DossierCommercial.indice                       AS DossierCommercial_52_6,
    DossierCommercial.code_ss_classement           AS DossierCommercial_52_7,
    OrigineNegociation.code                        AS OrigineNegociation_53_0,
    MotifBlocageInformation.libellelong            AS MotifBlocageInformat_54_3,
    ArbreLogistique.id                             AS ArbreLogistique_63_1,
    ArbreLogistique.codesap                        AS ArbreLogistique_63_5,
    Fournisseur.code                               AS Fournisseur_66_0,
    Fournisseur.nom                                AS Fournisseur_66_2,
    Filiere.code                                   AS Filiere_67_0,
    Filiere.nom                                    AS Filiere_67_2,
    ValorisationAchat.val_ach_patc                 AS Valorisation_74_3,
    LienPrixVente.code                             AS LienPrixVente_76_0,
    LienPrixVente.datem                            AS LienPrixVente_76_1,
    LienGratuite.code                              AS LienGratuite_78_0,
    LienGratuite.datem                             AS LienGratuite_78_1,
    LienCoordonnable.code                          AS LienCoordonnable_79_0,
    LienCoordonnable.datem                         AS LienCoordonnable_79_1,
    LienStatistique.code                           AS LienStatistique_81_0,
    LienStatistique.datem                          AS LienStatistique_81_1
FROM   da_article Article
       join (SELECT idarticle,
                    poids,
                    ROW_NUMBER()
                      over (
                        PARTITION BY RNA.id
                        ORDER BY INNERSEARCH.poids) RN,
                    titre,
                    nom,
                    prenom
             FROM   da_article RNA
                    join (SELECT idarticle,
                        pkg_db_indexation.CALCULPOIDSMOTS(chaine,
                            'foire vins%') AS POIDS,
                        DECODE(index_clerecherche, 'Piste.titre', chaine,
                                                   '')               AS TITRE,
                        DECODE(index_clerecherche, 'Artiste.nom_prenom',
                            SUBSTR(chaine, 0, INSTR(chaine, '_') - 1),
                                                   '')               AS NOM,
                        DECODE(index_clerecherche, 'Artiste.nom_prenom',
                            SUBSTR(chaine, INSTR(chaine, '_') + 1),
                                                   '')               AS PRENOM
                      FROM   ((SELECT index_idenreg AS IDARTICLE,
                                      C.cde_art     AS CHAINE,
```

```
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Article.codeArticle'
        join da_article C
          ON id = index_idenreg
 WHERE  mots_mot = 'foire'
 INTERSECT
 SELECT index_idenreg AS IDARTICLE,
        C.cde_art     AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Article.codeArticle'
        join da_article C
          ON id = index_idenreg
 WHERE  mots_mot LIKE 'vins%'
    AND 1 = 1)
UNION ALL
(SELECT index_idenreg AS IDARTICLE,
        C.cde_art_bal AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Article.codeArticleBalance'
        join da_article C
          ON id = index_idenreg
 WHERE  mots_mot = 'foire'
 INTERSECT
 SELECT index_idenreg AS IDARTICLE,
        C.cde_art_bal AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Article.codeArticleBalance'
        join da_article C
          ON id = index_idenreg
 WHERE  mots_mot LIKE 'vins%'
    AND 1 = 1)
UNION ALL
(SELECT index_idenreg AS IDARTICLE,
        C.lib_com     AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Article.libelleCommercial'
        join da_article C
```

```
          ON id = index_idenreg
 WHERE  mots_mot = 'foire'
 INTERSECT
 SELECT index_idenreg AS IDARTICLE,
        C.lib_com     AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Article.libelleCommercial'
        join da_article C
          ON id = index_idenreg
 WHERE  mots_mot LIKE 'vins%'
    AND 1 = 1)
UNION ALL
(SELECT idda_article AS IDARTICLE,
        C.gtin       AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
               'Gtin.gtin'
        join da_gtin C
          ON id = index_idenreg
 WHERE  mots_mot = 'foire'
 INTERSECT
 SELECT idda_article AS IDARTICLE,
        C.gtin       AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
              'Gtin.gtin'
        join da_gtin C
          ON id = index_idenreg
 WHERE  mots_mot LIKE 'vins%'
    AND 1 = 1)
UNION ALL
(SELECT idda_article AS IDARTICLE,
        C.ref_frn    AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
        join cstd_index I
          ON I.mots_id = M.mots_id
             AND index_clerecherche =
      'ArbreLogistique.referenceFournisseur'
        join da_arblogi C
          ON id = index_idenreg
 WHERE  mots_mot = 'foire'
 INTERSECT
 SELECT idda_article AS IDARTICLE,
        C.ref_frn    AS CHAINE,
        index_clerecherche
 FROM   cstd_mots M
```

```
                                    join cstd_index I
                                      ON I.mots_id = M.mots_id
                                         AND index_clerecherche =
                                  'ArbreLogistique.referenceFournisseur'
                                    join da_arblogi C
                                      ON id = index_idenreg
                               WHERE  mots_mot LIKE 'vins%'
                                  AND 1 = 1))) INNERSEARCH
                  ON INNERSEARCH.idarticle = RNA.id) SEARCHMC
    ON SEARCHMC.idarticle = Article.id
       AND 1 = 1
  left join da_artcmpl ArticleComplement
         ON Article.id = ArticleComplement.idda_article
  left join dp_gpdtcul GenreProduitCulturel
         ON ArticleComplement.iddp_gpdtcul = GenreProduitCulturel.id
  left join df_collect Collection
         ON ArticleComplement.iddf_collect = Collection.id
  left join da_gtin Gtin
         ON Article.id = Gtin.idda_article
            AND Gtin.principal = 1
            AND Gtin.db_suplog = 0
  left join df_fabric Fabricant
         ON Article.iddf_fabric = Fabricant.id
  left join dp_clsvtel ClassificationVenteLocale
         ON Article.iddp_clsvtel = ClassificationVenteLocale.id
  left join dp_marqcom MarqueCommerciale
         ON Article.iddp_marqcom = MarqueCommerciale.id
  left join da_compo Composition
         ON Composition.id = Article.idda_compo
  left join dp_comptex CompositionTextile
         ON CompositionTextile.id = Article.iddp_comptex
  left join da_arttrai AssoArticleInterfaceBalance
         ON AssoArticleInterfaceBalance.idda_article = Article.id
            AND AssoArticleInterfaceBalance.iddp_tinterf = 1
  left join da_arttrai AssoArticleInterfaceCaisse
         ON AssoArticleInterfaceCaisse.idda_article = Article.id
            AND AssoArticleInterfaceCaisse.iddp_tinterf = 4
  left join dp_raybala RayonBalance
         ON Article.iddp_raybala = RayonBalance.id
  left join dp_valdico TypeTare
         ON TypeTare.id = RayonBalance.iddp_typtare
  left join df_categor Categorie
         ON Categorie.id = Article.iddf_categor
  left join df_grille GrilleDePrix
         ON GrilleDePrix.id = Categorie.iddf_grille
  left join dp_agreage FicheAgreage
         ON FicheAgreage.id = Article.iddp_agreage
  join dp_codelec Codelec
    ON Article.iddp_codelec = Codelec.id
  left join dp_periact PerimetreActivite
         ON PerimetreActivite.id = Codelec.iddp_periact
  left join dp_perscod DonneesPersonnalisablesCodelec
         ON Codelec.id = DonneesPersonnalisablesCodelec.iddp_codelec
            AND DonneesPersonnalisablesCodelec.db_suplog = 0
            AND DonneesPersonnalisablesCodelec.iddb_sitecl = 1012124
  left join dp_clsart ClassificationArticleInterne
         ON DonneesPersonnalisablesCodelec.iddp_clsart =
```

```
          ClassificationArticleInterne.id
left join da_artdeno ArticleDenormalise
       ON Article.id = ArticleDenormalise.idda_article
left join df_clasmnt ClassementFournisseur
       ON ArticleDenormalise.iddf_clasmnt = ClassementFournisseur.id
left join tr_dosclas DossierDeClassement
       ON ClassementFournisseur.id = DossierDeClassement.iddf_clasmnt
          AND DossierDeClassement.date_deb <= '2013-09-27'
          AND COALESCE(DossierDeClassement.date_fin,
             TO_DATE('31129999', 'DDMMYYYY')) >= '2013-09-27'
left join tr_doscomm DossierCommercial
       ON DossierDeClassement.idtr_doscomm = DossierCommercial.id
left join dp_valdico OrigineNegociation
       ON DossierCommercial.iddp_dossref = OrigineNegociation.id
left join dp_motbloc MotifBlocageInformation
       ON MotifBlocageInformation.id = ArticleDenormalise.idda_motinf
left join da_arblogi ArbreLogistique
       ON Article.id = ArbreLogistique.idda_article
          AND ArbreLogistique.princ = 1
          AND ArbreLogistique.db_suplog = 0
left join df_filiere Filiere
       ON ArbreLogistique.iddf_filiere = Filiere.id
left join df_fourn Fournisseur
       ON Filiere.iddf_fourn = Fournisseur.id
left join od_dosal dossierALValo
       ON dossierALValo.idda_arblogi = ArbreLogistique.id
          AND dossierALValo.idod_dossier IS NULL
left join tt_val_dal valoDossier
       ON valoDossier.idod_dosal = dossierALValo.id
          AND valoDossier.estarecalculer = 0
left join tt_valo ValorisationAchat
       ON ValorisationAchat.idtt_val_dal = valoDossier.id
          AND ValorisationAchat.date_modif_retro IS NULL
          AND ValorisationAchat.date_debut_achat <= '2013-09-27'
          AND COALESCE(ValorisationAchat.date_fin_achat,
             TO_DATE('31129999', 'DDMMYYYY')) >= '2013-09-27'
          AND ValorisationAchat.val_ach_pab IS NOT NULL
left join da_lienart assoALPXVT
       ON assoALPXVT.idda_article = Article.id
          AND assoALPXVT.iddp_typlien = 14893
left join da_lien LienPrixVente
       ON LienPrixVente.id = assoALPXVT.idda_lien
left join da_lienart assoALGRAT
       ON assoALGRAT.idda_article = Article.id
          AND assoALGRAT.iddp_typlien = 14894
left join da_lien LienGratuite
       ON LienGratuite.id = assoALGRAT.idda_lien
left join da_lienart assoALCOOR
       ON assoALCOOR.idda_article = Article.id
          AND assoALCOOR.iddp_typlien = 14899
left join da_lien LienCoordonnable
       ON LienCoordonnable.id = assoALCOOR.idda_lien
left join da_lienal assoALSTAT
       ON assoALSTAT.idda_arblogi = ArbreLogistique.id
          AND assoALSTAT.iddp_typlien = 14897
left join da_lien LienStatistique
       ON LienStatistique.id = assoALSTAT.idda_lien WHERE
```

```
SEARCHMC.rn = 1
   AND ( ValorisationAchat.id IS NULL
          OR ValorisationAchat.date_debut_achat = (
                 SELECT MAX(VALMAX.date_debut_achat)
                 FROM   tt_valo VALMAX
                 WHERE  VALMAX.idtt_val_dal = ValorisationAchat.idtt_val_dal
                    AND VALMAX.date_modif_retro IS NULL
                    AND VALMAX.val_ach_pab IS NOT NULL
                    AND VALMAX.date_debut_achat <= '2013-09-27') )
   AND ( Article.id IN (SELECT A.id
                        FROM   da_article A
                               join du_ucutiar AssoUcUtiAr
                                 ON AssoUcUtiAr.idda_article = A.id
                               join du_asucuti AssoUcUti
                                 ON AssoUcUti.id = AssoUcUtiAr.iddu_asucuti
                        WHERE  ( AssoUcUti.iddu_uti IN ( 90000000000022 ) )
                           AND a.iddb_sitecl = 1012124) )
   AND Article.db_suplog = 0
ORDER  BY SEARCHMC.poids ASC
```

Comprendre un tel monstre implique souvent de l'imprimer pour acquérir une vision globale et prendre des notes :

Figure 1/ .1 – Un exemple à ne pas suivre

Ce code a été généré initialement par Hibernate, puis édité plusieurs fois à la main.

1.15 RECHERCHE TEXTUELLE

- Objectif
  - ajouter un moteur de recherche à l'application
- Pratique courante
  - utiliser l'opérateur `LIKE`
- Problèmes
  - requiert des index spécialisés
  - recherche uniquement le terme exact
- Solutions
  - `pg_trgm`
  - *Full Text Search*

Les bases de données qui stockent des données textuelles ont souvent pour but de permettre des recherches sur ces données textuelles.

La première solution envisagée lorsque le besoin se fait sentir est d'utiliser l'opérateur `LIKE`. Il permet en effet de réaliser des recherches de motif sur une colonne stockant des données textuelles. C'est une solution simple et qui peut s'avérer simpliste dans de nombreux cas.

Tout d'abord, les recherches de type `LIKE '%motif%'` ne peuvent généralement pas tirer partie d'un index btree normal. Cela étant dit, l'extension `pg_trgm` permet d'optimiser ces recherches à l'aide d'un index GiST ou GIN. Elle fait partie des extensions standard et ne nécessite pas d'adaptation du code.

Exemples

L'exemple ci-dessous montre l'utilisation du module `pg_trgm` pour accélérer une recherche avec `LIKE '%motif%'` :

```
CREATE INDEX idx_appellation_libelle ON appellation
USING btree (libelle varchar_pattern_ops);

EXPLAIN SELECT * FROM appellation WHERE libelle LIKE '%wur%';

                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=3 width=24)
   Filter: (libelle ~~ '%wur%'::text)

CREATE EXTENSION pg_trgm;

CREATE INDEX idx_appellation_libelle_trgm ON appellation
USING gist (libelle gist_trgm_ops);

EXPLAIN SELECT * FROM appellation WHERE libelle LIKE '%wur%';

                                QUERY PLAN
```

```
------------------------------------------------------------------------------
 Bitmap Heap Scan on appellation  (cost=4.27..7.41 rows=3 width=24)
   Recheck Cond: (libelle ~~ '%wur%'::text)
   ->  Bitmap Index Scan on idx_appellation_libelle_trgm  (cost=0.00..4.27...)
         Index Cond: (libelle ~~ '%wur%'::text)
```

Mais cette solution n'offre pas la même souplesse que la recherche plein texte, en anglais *Full Text Search*, de PostgreSQL. Elle est cependant plus complexe à mettre en œuvre et possède une syntaxe spécifique.

1.16 CONCLUSION

— La base est là pour vous aider
— Le modèle relationnel doit être compris et appliqué
— Avant de contourner un problème, chercher s'il n'existe pas une fonctionnalité dédiée

1.17 QUIZ

https://dali.bo/s65_quiz

1.18 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s65_solutions.

Ce TP utilise les tables `voitures` et `voitures_ecv`.

Les deux tables `voitures` et `voitures_ecv` peuvent être téléchargées installées comme suit :

```
createdb voitures
curl -kL https://dali.bo/tp_voitures -o /tmp/voitures.dmp
pg_restore -d voitures /tmp/voitures.dmp
# un message sur le schéma public préexistant est normal
```

Ne pas oublier d'effectuer un `VACUUM ANALYZE`.

1.18.1 Normalisation de schéma

But : Normaliser un schéma de données.

La table `voitures` viole la première forme normale (attribut répétitif, non atomique). De plus elle n'a pas de clé primaire.

Renommer la table en `voitures_orig`. \ Ne pas la supprimer (nous en aurons besoin plus tard).

Écrire des requêtes permettant d'éclater cette table en trois tables : \ `voitures`, `caracteristiques` et `caracteristiques_voitures`. \ (La fonction `regexp_split_to_table` permettra de séparer les champs de caractéristiques.)

Mettre en place les contraintes d'intégrité : clé primaire sur chaque table, et clés étrangères. \ Ne pas prévoir encore d'index supplémentaire. \ **Attention** : la table de départ contient des immatriculations en doublon !

Tenter d'insérer une Clio avec les caractéristiques « ABS » (majusucules) et « phares LED ».

Comparer les performances entre les deux modèles pour une recherche des voitures ayant un toit ouvrant.

Les plans sont-ils les mêmes si la caractéristique recherchée n'existe pas ?

> Indexer la colonne de clé étrangère `caracteristiques_voitures.carateristique` et voir ce que devient le plan de la dernière requête.

> Rechercher une voitures possédant les 3 options ABS, toit ouvrant et 4 roues motrices, et voir le plan.

1.18.2 Entité-clé-valeur

> **But** : Manipuler des données au format entité/clé/valeur.

Une autre version de la table `voiture` existe aussi dans cette base au format « entité/clé/valeur » c'est la table `voitures_ecv`. Sa clé primaire est `entite` (immatriculation) / `cle` (caractéristique). En pratique il n'y a que des booléens.

> Afficher toutes les caractéristiques d'une voiture au hasard (par exemple ZY-745-KT).

> Trouver toutes les caractéristiques de toutes les voitures ayant un toit ouvrant dans `voitures_ecv`. Trier par immatriculation. Quel est le plan d'exécution ?

`hstore` est une extension qui permet de stocker des clés/valeur dans un champ. Sa documentation est sur le site du projet[9].

> Installer l'extension `hstore`. \ Convertir cette table pour qu'elle utilise une ligne par immatriculation, avec les caractéristiques dans un champ `hstore`. \ Une méthode simple est de récupérer les lignes d'une même immatriculation avec la fonction `array_agg` puis de convertir simplement en champ `hstore`.

> Rechercher la voiture précédente.

> Insérer une voiture avec les caractéristiques `couleur=>vert` et `phares=>LED`.

> Définir un index de type GiST sur ce champ `hstore`. \ Retrouver la voiture insérée par ses caractéristiques.

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

1.18.3 Indexation de champs tableau

But : Indexer un champ tableau pour améliorer les performances.

Il est possible, si on peut réécrire la requête, d'obtenir de bonnes performances avec la première table `voitures_orig`. En effet, PostgreSQL sait indexer des tableaux et des fonctions. Il saurait donc indexer un tableau résultat d'une fonction sur le champ `caracteristiques`.

> Trouver cette fonction dans la documentation de PostgreSQL (chercher dans les fonctions de découpage de chaîne de caractères).

> Définir un index fonctionnel sur le résultat de cette fonction, de type GIN.

> Rechercher toutes les voitures avec toit ouvrant et voir le plan.

1.18.4 Pagination et index

But : Effectuer une requête avec pagination

La pagination est une fonctionnalité que l'on retrouve de plus en plus souvent, surtout depuis que les applications web ont pris une place prépondérante.

Nous allons utiliser une version simplifiée d'une table de forum.

La table `posts` (dump de 358 Mo, 758 Mo sur disque) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_posts -o /tmp/posts.dump
createdb posts
pg_restore -d posts /tmp/posts.dump
# le message sur le schéma public préexistant est normal
rm -- /tmp/posts.dump
```

Ne pas oublier d'effectuer ensuite un `VACUUM ANALYZE`.

Nous voulons afficher le plus rapidement possible les messages (*posts*) associés à un article : les 10 premiers, puis du 11 au 20, etc. Nous allons examiner les différentes stratégies possibles.

La table contient 5 000 articles de 1000 posts, d'au plus 200 signes.

La description de la table est :

```
# \d posts
                              Table « public.posts »
  Colonne   |           Type           | Collationnement | NULL-able | Par défaut
------------+--------------------------+-----------------+-----------+------------
 id_article | integer                  |                 |           |
 id_post    | integer                  |                 |           |
 ts         | timestamp with time zone |                 |           |
 message    | text                     |                 |           |
Index :
    "posts_ts_idx" btree (ts)
```

Pour la clarté des plans, désactiver le JIT et le parallélisme dans votre session :

```
SET jit to off ;
SET max_parallel_workers_per_gather TO 0 ;
```

> Écrire une requête permettant de récupérer les 10 premiers posts de l'article d'`id_article` = `12`, triés dans l'ordre de `id_post`. Il n'y a pas d'index, la requête va être très lente.

> Créer un index permettant d'améliorer cette requête.

> Utiliser les clauses `LIMIT` et `OFFSET` pour récupérer les 10 posts suivants. Puis du post 901 au 921. Que constate-t-on sur le plan d'exécution ?

> Trouver une réécriture de la requête pour trouver directement les posts 901 à 911 une fois connu le post 900 récupéré au travers de la pagination.

1.18.5 Clauses WHERE et pièges

> **But** : Mise en évidence de cas piégeux dans les clauses WHERE.

Nous utilisons toujours la table `posts`. Nous allons maintenant manipuler le champ `ts`, de type `timestamp`. Ce champ est indexé.

> La requête \ `SELECT * FROM posts WHERE to_char(ts,'YYYYMM')='201302'` \ retourne tous les enregistrements de février 2013. Examiner son plan d'exécution. Où est le problème ?

> Réécrire la clause `WHERE` avec une inégalité de dates pour utiliser l'index sur `ts`.

> Plus compliqué : retourner tous les posts ayant eu lieu un dimanche, en 2013, en passant par un index et en une seule requête. \ (Indice : il est possible de générer la liste de tous les dimanches de l'année 2013 avec `generate_series('2013-01-06 00:00:00','2014-01-01 00:00:00', INTERVAL '7 days')`

> On cherche un article à peu près au tiers de la liste avec la requête suivante. Pourquoi est-elle si lente ?

```
SELECT * FROM posts
WHERE id_article =
    (SELECT max(id_article) * 0.333
     FROM posts
     ) ;
```

1.19 TRAVAUX PRATIQUES (SOLUTIONS)

1.19.1 Normalisation de schéma

> Renommer la table en `voitures_orig`. \ Ne pas la supprimer (nous en aurons besoin plus tard).

```
ALTER TABLE voitures rename TO voitures_orig;
```

> Écrire des requêtes permettant d'éclater cette table en trois tables : \ `voitures`, `caracteristiques` et `caracteristiques_voitures`. \ (La fonction `regexp_split_to_table` permettra de séparer les champs de caractéristiques.)

```
CREATE TABLE voitures AS
  SELECT DISTINCT ON (immatriculation) immatriculation, modele
  FROM voitures_orig ;

ALTER TABLE voitures ADD PRIMARY KEY (immatriculation);

CREATE TABLE caracteristiques
  AS SELECT *
     FROM (
             SELECT DISTINCT
              regexp_split_to_table(caracteristiques,',') caracteristique
             FROM voitures_orig)
          AS tmp
     WHERE caracteristique <> '' ;

ALTER TABLE caracteristiques ADD PRIMARY KEY (caracteristique);

CREATE TABLE caracteristiques_voitures
  AS SELECT DISTINCT *
  FROM (
       SELECT
         immatriculation,
         regexp_split_to_table(caracteristiques,',') caracteristique
       FROM voitures_orig
      )
    AS tmp
   WHERE caracteristique <> '';

VACUUM ANALYZE ;
```

```
\d+
                                    Liste des relations
 Schéma |            Nom             | Type  | Propriétaire | ... | Taille  | ...
--------+----------------------------+-------+--------------+-----+---------+-
 public | caracteristiques           | table | postgres     |     | 48 kB   |
 public | caracteristiques_voitures  | table | postgres     |     | 3208 kB |
 public | voitures                   | table | postgres     |     | 4952 kB |
 public | voitures_ecv               | table | postgres     |     | 3336 kB |
 public | voitures_orig              | table | postgres     |     | 5736 kB |
```

> Mettre en place les contraintes d'intégrité : clé primaire sur chaque table, et clés étrangères. \ Ne pas prévoir encore d'index supplémentaire. \ **Attention** : la table de départ contient des immatriculations en doublon !

Sur `caracteristiques_voitures`, la clé primaire comprend les deux colonnes, et donc interdit qu'une même caractéristique soit présente deux fois sur la même voiture :

```
ALTER TABLE caracteristiques_voitures
ADD PRIMARY KEY (immatriculation,caracteristique);
```

Clé étrangère de cette table vers les deux autres tables :

```
ALTER TABLE caracteristiques_voitures
ADD FOREIGN KEY (immatriculation)
REFERENCES voitures(immatriculation);

ALTER TABLE caracteristiques_voitures
ADD FOREIGN KEY (caracteristique)
REFERENCES caracteristiques(caracteristique);
```

> Tenter d'insérer une Clio avec les caractéristiques « ABS » (majusucules) et « phares LED ».

En toute rigueur il faut le faire dans une transaction :

```
BEGIN ;
INSERT INTO voitures VALUES ('AA-007-JB','clio') ;
INSERT INTO caracteristiques_voitures (immatriculation, caracteristique)
VALUES ('AA-007-JB','ABS') ;
INSERT INTO caracteristiques_voitures (immatriculation, caracteristique)
VALUES ('AA-007-JB','phares LED') ;
COMMIT ;
```

Évidemment, cela échoue :

```
ERROR:  insert or update on table "caracteristiques_voitures" violates foreign key
        constraint "caracteristiques_voitures_caracteristique_fkey"
DÉTAIL : Key (caracteristique)=(ABS) is not present in table "caracteristiques".

ERROR:  insert or update on table "caracteristiques_voitures" violates foreign key
        constraint "caracteristiques_voitures_immatriculation_fkey"
DÉTAIL : Key (immatriculation)=(AA-007-JB) is not present in table "voitures".
```

En cas d'erreur, c'est exactement ce que l'on veut.

Pour que l'insertion fonctionne, il faut corriger la casse de « ABS » et déclarer la nouvelle propriété :

```
BEGIN ;
INSERT INTO voitures VALUES ('AA-007-JB','clio') ;
INSERT INTO caracteristiques VALUES ('phares LED') ;
INSERT INTO caracteristiques_voitures (immatriculation, caracteristique)
VALUES ('AA-007-JB','abs') ;
INSERT INTO caracteristiques_voitures (immatriculation, caracteristique)
VALUES ('AA-007-JB','phares LED') ;
COMMIT ;
```

> Comparer les performances entre les deux modèles pour une recherche des voitures ayant un toit ouvrant.

La version la plus simple est :

```
SELECT * FROM voitures_orig
WHERE caracteristiques like '%toit ouvrant%';
```

Plus rigoureusement (`[[:>:]]` et `[[:<:]]` indiquent des frontières de mots.), on préférera :

```
  EXPLAIN ANALYZE
  SELECT * FROM voitures_orig
  WHERE caracteristiques ~ E'[[:<:]]toit ouvrant[[:>:]]' ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on voitures_orig  (cost=0.00..1962.00 rows=8419 width=25)
                            (actual time=0.030..92.226 rows=8358 loops=1)
   Filter: (caracteristiques ~ '[[:<:]]toit ouvrant[[:>:]]'::text)
   Rows Removed by Filter: 91642
 Planning Time: 0.658 ms
 Execution Time: 92.512 ms
```

Toute la table a été parcourue, 91 642 lignes ont été rejetées, 8358 retenues (~8 %). Les estimations statistiques sont correctes.

NB : pour la lisibilité, les plans n'utilisent pas l'option `BUFFERS` d'`EXPLAIN`. Si on l'active, on pourra vérifier que tous les accès se font bien dans le cache de PostgreSQL (`shared hits`).

Avec le nouveau schéma on peut écrire la requête simplement avec une simple jointure :

```
SELECT *
FROM  voitures
INNER JOIN  caracteristiques_voitures
       ON ( caracteristiques_voitures.immatriculation=voitures.immatriculation)
WHERE caracteristique = 'toit ouvrant' ;
```

Il n'y a pas doublement de lignes si une caractéristique est en double car la clé primaire l'interdit. Sans cette contrainte, une autre écriture serait nécessaire :

```
SELECT *
FROM voitures
WHERE EXISTS (
    SELECT 1 FROM caracteristiques_voitures
    WHERE caracteristiques_voitures.immatriculation=voitures.immatriculation
    AND caracteristique = 'toit ouvrant'
) ;
```

Dans les deux cas, on obtient ce plan[10] :

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=1225.80..3102.17 rows=8329 width=16)
            (actual time=6.307..31.811 rows=8358 loops=1)
```

[10] https://explain.dalibo.com/plan/lz

```
   Hash Cond: (voitures.immatriculation = caracteristiques_voitures.immatriculation)
   ->  Seq Scan on voitures  (cost=0.00..1613.89 rows=99989 width=16)
                             (actual time=0.019..10.432 rows=99989 loops=1)
   ->  Hash  (cost=1121.69..1121.69 rows=8329 width=10)
             (actual time=6.278..6.279 rows=8358 loops=1)
         Buckets: 16384  Batches: 1  Memory Usage: 577kB
         ->  Seq Scan on caracteristiques_voitures
                                   (cost=0.00..1121.69 rows=8329 width=10)
                                   (actual time=0.004..5.077 rows=8358 loops=1)
               Filter: (caracteristique = 'toit ouvrant'::text)
               Rows Removed by Filter: 49697
 Planning Time: 0.351 ms
 Execution Time: 32.155 ms
```

Le temps d'exécution est ici plus court malgré un parcours complet de `voitures`. PostgreSQL prévoit correctement qu'il ramènera 10 % de cette table, ce qui n'est pas si discriminant et justifie fréquemment un `Seq Scan`, surtout que `voitures` est petite. `caracteristiques_voitures` est aussi parcourue entièrement : faute d'index, il n'y a pas d'autre moyen.

> Les plans sont-ils les mêmes si la caractéristique recherchée n'existe pas ?

Si on cherche une option rare ou n'existant pas, le plan change[11] :

```
EXPLAIN ANALYZE
SELECT *
FROM  voitures
INNER JOIN  caracteristiques_voitures
       ON ( caracteristiques_voitures.immatriculation=voitures.immatriculation)
WHERE caracteristique = 'ordinateur de bord' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.42..1130.12 rows=1 width=16)
              (actual time=4.849..4.850 rows=0 loops=1)
   ->  Seq Scan on caracteristiques_voitures  (cost=0.00..1121.69 rows=1 width=10)
                                         (actual time=4.848..4.848 rows=0 loops=1)
         Filter: (caracteristique = 'ordinateur de bord'::text)
         Rows Removed by Filter: 58055
   ->  Index Scan using voitures_pkey on voitures  (cost=0.42..8.44 rows=1 width=16)
                                                   (never executed)
         Index Cond: (immatriculation = caracteristiques_voitures.immatriculation)
 Planning Time: 0.337 ms
 Execution Time: 4.872 ms
```

Avec un seul résultat attendu, ce qui est beaucoup plus discriminant, l'utilisation de l'index sur `voitures` devient pertinente.

Avec l'ancien schéma, on doit toujours lire la table `voitures_orig` en entier.

> Indexer la colonne de clé étrangère `caracteristiques_voitures.carateristique` et voir ce que devient le plan de la dernière requête.

[11] https://explain.dalibo.com/plan/Hij

```
CREATE INDEX ON caracteristiques_voitures (caracteristique) ;
```

Le plan d'exécution[12] devient foudroyant, puisque la table `caracteristiques_voitures` n'est plus intégralement lue :

```
EXPLAIN ANALYZE
SELECT *
FROM  voitures
INNER JOIN  caracteristiques_voitures
       ON ( caracteristiques_voitures.immatriculation=voitures.immatriculation)
WHERE caracteristique = 'ordinateur de bord' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.83..16.78 rows=1 width=16)
              (actual time=0.010..0.011 rows=0 loops=1)
   ->  Index Scan using caracteristiques_voitures_caracteristique_idx
                     on caracteristiques_voitures
              (cost=0.41..8.35 rows=1 width=10)
              (actual time=0.010..0.010 rows=0 loops=1)
         Index Cond: (caracteristique = 'ordinateur de bord'::text)
   ->  Index Scan using voitures_pkey on voitures  (cost=0.42..8.44 rows=1 width=16)
                                                   (never executed)
         Index Cond: (immatriculation = caracteristiques_voitures.immatriculation)
 Planning Time: 0.268 ms
 Execution Time: 0.035 ms
```

Avec `voitures_orig`, il existerait aussi des méthodes d'indexation mais elles sont plus lourdes (index GIN…).

> Rechercher une voitures possédant les 3 options ABS, toit ouvrant et 4 roues motrices, et voir le plan.

Si on recherche plusieurs options en même temps, l'optimiseur peut améliorer les choses en prenant en compte la fréquence de chaque option pour restreindre plus efficacement les recherches. Le plan devient[13] :

```
  EXPLAIN (ANALYZE, COSTS OFF)
  SELECT *
  FROM voitures
  JOIN caracteristiques_voitures AS cr1 USING (immatriculation)
  JOIN caracteristiques_voitures AS cr2 USING (immatriculation)
  JOIN caracteristiques_voitures AS cr3 USING (immatriculation)
  WHERE cr1.caracteristique = 'toit ouvrant'
  AND cr2.caracteristique = 'abs'
  AND cr3.caracteristique='4 roues motrices' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
Nested Loop
->  Hash Join
        Hash Cond: (cr2.immatriculation = cr1.immatriculation)
```

[12] https://explain.dalibo.com/plan/577
[13] https://explain.dalibo.com/plan/O0H

```
        ->  Bitmap Heap Scan on caracteristiques_voitures cr2
            Recheck Cond: (caracteristique = 'abs'::text)
            ->  Bitmap Index Scan on caracteristiques_voitures_caracteristique_idx
                    Index Cond: (caracteristique = 'abs'::text)
        ->  Hash
            ->  Hash Join
                    Hash Cond: (cr1.immatriculation = cr3.immatriculation)
                    ->  Bitmap Heap Scan on caracteristiques_voitures cr1
                        Recheck Cond: (caracteristique = 'toit ouvrant'::text)
                        ->  Bitmap Index Scan
                            on caracteristiques_voitures_caracteristique_idx
                                Index Cond: (caracteristique = 'toit ouvrant'::text)
                    ->  Hash
                        ->  Bitmap Heap Scan on caracteristiques_voitures cr3
                                Recheck Cond: (caracteristique =
                                                    '4 roues motrices'::text)
                                ->  Bitmap Index Scan
                                    on caracteristiques_voitures_caracteristique_idx
                                    Index Cond: (caracteristique =
                                                    '4 roues motrices'::text)
->  Index Scan using voitures_pkey on voitures
        Index Cond: (immatriculation = cr1.immatriculation)
```

Ce plan parcoure deux index, joins leurs résultats, fait de même avec le résultat de l'index pour la 3è caractéristique, puis opère la jointure finale avec la table principale par l'index sur `immatriculation` (un plan complet indiquerait une estimation de 56 lignes de résultat, même si le résultat final est de 461 lignes).

Mais les problématiques de performances ne sont pas le plus important dans ce cas. Ce qu'on gagne réellement, c'est la garantie que les caractéristiques ne seront que celles existant dans la table `caractéristique`, ce qui évite d'avoir à réparer la base plus tard.

1.19.2 Entité-clé-valeur

> Afficher toutes les caractéristiques d'une voiture au hasard (par exemple ZY-745-KT).

```
SELECT * FROM voitures_ecv
WHERE entite = 'ZY-745-KT' ;
```

```
  entite   |          cle          | valeur
-----------+-----------------------+--------
 ZY-745-KT | climatisation         | t
 ZY-745-KT | jantes aluminium      | t
 ZY-745-KT | regulateur de vitesse | t
 ZY-745-KT | toit ouvrant          | t
```

> Trouver toutes les caractéristiques de toutes les voitures ayant un toit ouvrant dans `voitures_ecv`. Trier par immatriculation. Quel est le plan d'exécution ?

Autrement dit : on sélectionne toutes les voitures avec un toit ouvrant, et l'on veut toutes les caractéristiques de ces voitures. Cela nécessite d'appeler deux fois la table.

Là encore une jointure de la table avec elle-même sur `entite` serait possible, mais serait dangereuse dans les cas où il y a énormément de propriétés. On préférera encore la version avec `EXISTS`, et PostgreSQL en fera spontanément une jointure[14] :

```
EXPLAIN ANALYZE
SELECT * FROM voitures_ecv
WHERE EXISTS (
  SELECT 1 FROM voitures_ecv test
  WHERE test.entite=voitures_ecv.entite
  AND cle = 'toit ouvrant' AND valeur = true
)
ORDER BY entite ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Sort  (cost=3468.93..3507.74 rows=15527 width=25)
       (actual time=29.854..30.692 rows=17782 loops=1)
   Sort Key: voitures_ecv.entite
   Sort Method: quicksort  Memory: 2109kB
   ->  Hash Join  (cost=1243.09..2388.05 rows=15527 width=25)
                  (actual time=6.915..23.964 rows=17782 loops=1)
         Hash Cond: (voitures_ecv.entite = test.entite)
         ->  Seq Scan on voitures_ecv  (cost=0.00..992.55 rows=58055 width=25)
                                       (actual time=0.006..4.242 rows=58055 loops=1)
         ->  Hash  (cost=1137.69..1137.69 rows=8432 width=10)
                   (actual time=6.899..6.899 rows=8358 loops=1)
               Buckets: 16384  Batches: 1  Memory Usage: 471kB
               ->  Seq Scan on voitures_ecv test
                                          (cost=0.00..1137.69 rows=8432 width=10)
                                    (actual time=0.005..5.615 rows=8358 loops=1)
                     Filter: (valeur AND (cle = 'toit ouvrant'::text))
                     Rows Removed by Filter: 49697
 Planning Time: 0.239 ms
 Execution Time: 31.321 ms
```

> Installer l'extension `hstore`. \ Convertir cette table pour qu'elle utilise une ligne par immatriculation, avec les caractéristiques dans un champ `hstore`. \ Une méthode simple est de récupérer les lignes d'une même immatriculation avec la fonction `array_agg` puis de convertir simplement en champ `hstore`.

`hstore` est normalement présente sur toutes les installations (ou alors l'administrateur a négligé d'installer le paquet `contrib`). Il suffit donc d'une déclaration.

```
CREATE EXTENSION hstore;

CREATE TABLE voitures_hstore
AS
    SELECT entite AS immatriculation,
           hstore(array_agg(cle),array_agg(valeur)::text[]) AS caracteristiques
    FROM voitures_ecv group by entite;

ALTER TABLE voitures_hstore ADD PRIMARY KEY (immatriculation);
```

[14] https://explain.dalibo.com/plan/nn2

> Rechercher la voiture précédente.

```
SELECT * FROM voitures_hstore
WHERE immatriculation = 'ZY-745-KT' \gx

-[ RECORD 1 ]----+-----------------------------------------------------------------
immatriculation  | ZY-745-KT
caracteristiques | "toit ouvrant"=>"true", "climatisation"=>"true",
                 | "jantes aluminium"=>"true", "regulateur de vitesse"=>"true"
```

L'accès à une caractéristique se fait ainsi (attention aux espaces) :

```
SELECT immatriculation, caracteristiques -> 'climatisation'
FROM voitures_hstore
WHERE immatriculation = 'ZY-745-KT' ;
```

> Insérer une voiture avec les caractéristiques `couleur=>vert` et `phares=>LED`.

```
INSERT INTO voitures_hstore
VALUES ('XX-4456-ZT', 'couleur=>vert, phares=>LED'::hstore) ;
```

> Définir un index de type GiST sur ce champ `hstore`. \ Retrouver la voiture insérée par ses caractéristiques.

Les index B-tree classiques sont inadaptés aux types complexes, on préfère donc un index GiST :

```
CREATE INDEX voitures_hstore_caracteristiques
ON voitures_hstore
USING gist (caracteristiques);
```

L'opérateur `@>` signifie « contient » :

```
SELECT *
FROM voitures_hstore
WHERE caracteristiques @> 'couleur=>vert' AND caracteristiques @> 'phares=>LED' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using voitures_hstore_caracteristiques on voitures_hstore
   (cost=0.28..2.30 rows=1 width=55) (actual time=0.033..0.033 rows=1 loops=1)
   Index Cond: ((caracteristiques @> '"couleur"=>"vert"'::hstore)
            AND (caracteristiques @> '"phares"=>"LED"'::hstore))
   Buffers: shared hit=4
 Planning Time: 0.055 ms
 Execution Time: 0.047 ms
```

1.19.3 Indexation de champs tableau

> Trouver cette fonction dans la documentation de PostgreSQL (chercher dans les fonctions de découpage de chaîne de caractères).

La fonction est `regexp_split_to_array` (sa documentation est sur https://docs.postgresql.fr/15/functions-matching.html) :

```
SELECT immatriculation, modele,
       regexp_split_to_array(caracteristiques,',')
FROM voitures_orig
LIMIT 10;
```

```
 immatriculation | modele |          regexp_split_to_array
-----------------+--------+-----------------------------------------
 WW-649-AI       | twingo | {"regulateur de vitesse"}
 QZ-533-JD       | clio   | {"4 roues motrices","jantes aluminium"}
 YY-854-LE       | megane | {climatisation}
 QD-761-QV       | twingo | {""}
 LV-277-QC       | megane | {abs,"jantes aluminium"}
 ZI-003-BQ       | kangoo | {"boite automatique",climatisation}
 WT-817-IK       | megane | {""}
 JK-791-XB       | megane | {""}
 WW-019-EK       | megane | {""}
 BZ-544-OS       | twingo | {""}
```

La syntaxe `{}` est la représentation texte d'un tableau.

> Définir un index fonctionnel sur le résultat de cette fonction, de type GIN.

```
CREATE INDEX idx_voitures_array ON voitures_orig
    USING gin (regexp_split_to_array(caracteristiques,','));
```

> Rechercher toutes les voitures avec toit ouvrant et voir le plan.

```
EXPLAIN ANALYZE
SELECT * FROM voitures_orig
WHERE regexp_split_to_array(caracteristiques,',') @> '{"toit ouvrant"}';
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on voitures_orig  (cost=8.87..387.37 rows=500 width=25)
                                    (actual time=0.707..2.756 rows=8358 loops=1)
   Recheck Cond: (regexp_split_to_array(caracteristiques, ','::text)
                  @> '{"toit ouvrant"}'::text[])
   Heap Blocks: exact=712
   ->  Bitmap Index Scan on idx_voitures_array  (cost=0.00..8.75 rows=500 width=0)
                                      (actual time=0.631..0.631 rows=8358 loops=1)
         Index Cond: (regexp_split_to_array(caracteristiques, ','::text)
                                                  @> '{"toit ouvrant"}'::text[])
 Planning Time: 0.129 ms
 Execution Time: 3.028 ms
```

Noter que les estimations de statistiques sont plus délicates sur un résultat de fonction.

1.19.4 Pagination et index

> Écrire une requête permettant de récupérer les 10 premiers posts de l'article d'`id_article` = `12`, triés dans l'ordre de `id_post`. Il n'y a pas d'index, la requête va être très lente.

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article =12
ORDER BY id_post
LIMIT   10 ;
```

Le plan[15] est un parcours complet de la table, rejetant 4 999 000 lignes et en gardant 1000 lignes, suivi d'un tri :

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=153694.51..153694.53 rows=10 width=115)
        (actual time=500.525..500.528 rows=10 loops=1)
   ->  Sort  (cost=153694.51..153696.95 rows=979 width=115)
             (actual time=500.524..500.525 rows=10 loops=1)
         Sort Key: id_post
         Sort Method: top-N heapsort  Memory: 27kB
         ->  Seq Scan on posts  (cost=0.00..153673.35 rows=979 width=115)
                                (actual time=1.300..500.442 rows=1000 loops=1)
               Filter: (id_article = 12)
               Rows Removed by Filter: 4999000
 Planning Time: 0.089 ms
 Execution Time: 500.549 ms
```

> Créer un index permettant d'améliorer cette requête.

Un index sur `id_article` améliorerait déjà les choses. Mais comme on trie sur `id_post`, il est intéressant de rajouter aussi cette colonne dans l'index :

```
CREATE INDEX posts_id_article_id_post ON posts (id_article, id_post);
```

Testons cet index :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article =12
ORDER BY id_post
LIMIT   10 ;
```

Le plan[16] devient :

```
                      QUERY PLAN
------------------------------------------------------------
 Limit  (cost=0.43..18.26 rows=10 width=115)
        (actual time=0.043..0.053 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
```

[15] https://explain.dalibo.com/plan/xEs
[16] https://explain.dalibo.com/plan/Fgy

```
                  (cost=0.43..1745.88 rows=979 width=115)
                (actual time=0.042..0.051 rows=10 loops=1)
          Index Cond: (id_article = 12)
 Planning Time: 0.204 ms
 Execution Time: 0.066 ms
```

C'est beaucoup plus rapide : l'index trouve tout de suite les lignes de l'article cherché, et retourne les enregistrements directement triés par `id_post`. On évite de parcourir toute la table, et il n'y a même pas d'étape de tri (qui serait certes très rapide sur 10 lignes).

> Utiliser les clauses `LIMIT` et `OFFSET` pour récupérer les 10 posts suivants. Puis du post 901 au 921. Que constate-t-on sur le plan d'exécution ?

Les posts 11 à 20 se trouvent rapidement :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article = 12
ORDER BY id_post
LIMIT   10
OFFSET  10;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=18.26..36.09 rows=10 width=115)
        (actual time=0.020..0.023 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                  (cost=0.43..1745.88 rows=979 width=115)
                  (actual time=0.017..0.021 rows=20 loops=1)
         Index Cond: (id_article = 12)
 Planning Time: 0.061 ms
 Execution Time: 0.036 ms
```

Tout va bien. La requête est à peine plus coûteuse. Noter que l'index a ramené 20 lignes et non 10.

À partir du post 900 :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article = 12
ORDER BY id_post
LIMIT   10
OFFSET  900 ;
```

Le plan[17] reste similaire :

```
                           QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=1605.04..1622.86 rows=10 width=115)
        (actual time=0.216..0.221 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                  (cost=0.43..1745.88 rows=979 width=115)
```

[17] https://explain.dalibo.com/plan/V05

```
                  (actual time=0.018..0.194 rows=910 loops=1)
        Index Cond: (id_article = 12)
 Planning Time: 0.062 ms
 Execution Time: 0.243 ms
```

Cette requête est 4 fois plus lente. Si une exécution unitaire ne pose pas encore problème, des demandes très répétées poseraient problème. Noter que l'index ramène 910 lignes ! Dans notre exemple idéalisée, les posts sont bien rangés ensemble, et souvent présents dans les mêmes blocs. C'est très différent dans une table qui beaucoup vécu.

> Trouver une réécriture de la requête pour trouver directement les posts 901 à 911 une fois connu le post 900 récupéré au travers de la pagination.

Pour se mettre dans la condition du test, récupérons l'enregistrement 900 :

```
SELECT  id_article, id_post
FROM    posts
WHERE   id_article = 12
ORDER BY id_post
LIMIT   1
OFFSET  899 ;
```

```
 id_article | id_post
------------+---------
         12 | 12900
```

(La valeur retournée peut différer sur une autre base.)

Il suffit donc de récupérer les 10 articles pour lesquels `id_article = 12` et `id_post > 12900` :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article = 12
AND     id_post> 12900
ORDER BY id_post
LIMIT   10;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=0.43..18.29 rows=10 width=115)
        (actual time=0.018..0.024 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                        (cost=0.43..1743.02 rows=976 width=115)
                        (actual time=0.016..0.020 rows=10 loops=1)
         Index Cond: ((id_article = 12) AND (id_post > 12900))
 Planning Time: 0.111 ms
 Execution Time: 0.039 ms
```

Nous sommes de retour à des temps d'exécution très faibles. Ajouter la condition sur le `id_post` permet de limiter à la source le nombre de lignes à récupérer. L'index n'en renvoie bien que 10.

L'avantage de cette technique par rapport à l'offset est que le temps d'une requête ne variera que l'on chercher la première ou la millième page.

L'inconvénient est qu'il faut mémoriser l'`id_post` où l'on s'est arrêté sur la page précédente.

1.19.5 Clauses WHERE et pièges

Nous allons maintenant manipuler le champ `ts` (de type `timestamp`) de la table `posts`.

> La requête \ `SELECT * FROM posts WHERE to_char(ts,'YYYYMM')='201302'` \ retourne tous les enregistrements de février 2013. Examiner son plan d'exécution. Où est le problème ?

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   to_char(ts,'YYYYMM')='201302' ;
```

Le plan[18] est un parcours complet de la table :

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on posts  (cost=0.00..187728.49 rows=50000 width=269)
                    (actual time=0.380..14163.371 rows=18234 loops=1)
   Filter: (to_char(ts, 'YYYYMM'::text) = '201302'::text)
   Rows Removed by Filter: 9981766
 Total runtime: 14166.265 ms
```

C'est normal : PostgreSQL ne peut pas deviner que `to_char(ts,'YYYYMM')='201302'` veut dire « toutes les dates du mois de février 2013 ». Une fonction est pour lui une boîte noire, et il ne voit pas le lien entre le résultat attendu et les données qu'il va lire.

Ceci est une des causes les plus habituelles de ralentissement de requêtes : une fonction est appliquée à une colonne, ce qui rend le filtre incompatible avec l'utilisation d'un index.

> Réécrire la clause `WHERE` avec une inégalité de dates pour utiliser l'index sur `ts`.

C'est à nous d'indiquer une clause `WHERE` au moteur qu'il puisse directement appliquer sur notre date :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   ts >= '2013-02-01'
AND     ts <  '2013-03-01' ;
```

Le plan[19] montre que l'index est maintenant utilisable :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using posts_ts_idx on posts  (cost=0.43..998.95 rows=20165 width=115)
                                         (actual time=0.050..5.907 rows=20160 loops=1)
   Index Cond: ((ts >= '2013-02-01 00:00:00+01'::timestamp with time zone)
            AND (ts < '2013-03-01 00:00:00+01'::timestamp with time zone))
 Planning Time: 0.095 ms
 Execution Time: 6.526 ms
```

[18] https://explain.dalibo.com/plan/ATT
[19] https://explain.dalibo.com/plan/GDY

Noter la conversion automatique du critère en `timestamp with time zone`.

> Plus compliqué : retourner tous les posts ayant eu lieu un dimanche, en 2013, en passant par un index et en une seule requête. \ (Indice : il est possible de générer la liste de tous les dimanches de l'année 2013 avec `generate_series('2013-01-06 00:00:00','2014-01-01 00:00:00', INTERVAL '7 days')`

Construisons cette requête morceau par morceau. Listons tous les dimanches de 2013 (le premier dimanche est le 6 janvier) :

```
SELECT generate_series(
    '2013-01-06 00:00:00',
    '2013-12-31 00:00:00',
    INTERVAL '7 days'
) ;
```

S'il faut calculer le premier dimanche de l'année, cela peut se faire ainsi :

```
WITH premiersjours AS (
  SELECT '2000-01-01'::timestamp + i * interval '1 year' AS jan1
  FROM   generate_series(1, 30) i
),
dimanches AS (
 SELECT jan1,
   jan1
    + mod(13-extract(dow FROM (jan1 - interval '1 day'))::int, 7)
      + interval '1 day'
   AS dim1
 FROM   premiersjours
)
SELECT jan1, dim1
FROM dimanches ;
```

On n'a encore que des dates à minuit. Il faut calculer les heures de début et de fin de chaque dimanche :

```
SELECT i AS debut,
       i + INTERVAL '1 day' AS fin
FROM generate_series(
    '2013-01-06 00:00:00',
    '2013-12-31 00:00:00',
    INTERVAL '7 days'
) g(i) ;
```

```
         debut          |          fin
------------------------+------------------------
 2013-01-06 00:00:00+01 | 2013-01-07 00:00:00+01
 2013-01-13 00:00:00+01 | 2013-01-14 00:00:00+01
...
 2013-12-29 00:00:00+01 | 2013-12-30 00:00:00+01
```

Il ne nous reste plus qu'à joindre ces deux ensembles. Comme clause de jointure, on teste la présence de la date du post dans un des intervalles des dimanches :

```
EXPLAIN ANALYZE
WITH dimanches AS (
```

```
    SELECT i AS debut,
        i + INTERVAL '1 day' AS fin
    FROM generate_series(
        '2013-01-06 00:00:00',
        '2013-12-31 00:00:00',
        INTERVAL '7 days'
    ) g(i)
)
SELECT posts.*
FROM posts
JOIN dimanches
ON (posts.ts >= dimanches.debut AND posts.ts < dimanches.fin) ;
```

Le plan[20] devient :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.44..17086517.00 rows=555555556 width=115)
              (actual time=0.038..12.978 rows=37440 loops=1)
   ->  Function Scan on generate_series g  (cost=0.00..10.00 rows=1000 width=8)
                                           (actual time=0.016..0.031 rows=52 loops=1)
   ->  Index Scan using posts_ts_idx on posts
                                     (cost=0.43..11530.95 rows=555556 width=115)
                                     (actual time=0.009..0.168 rows=720 loops=52)
         Index Cond: ((ts >= g.i) AND (ts < (g.i + '1 day'::interval)))
 Planning Time: 0.131 ms
 Execution Time: 14.178 ms
```

PostgreSQL génère les 52 lignes d'intervalles (noter qu'il ne sait pas estimer le résultat de cette fonction), puis fait 52 appels à l'index (noter le `loops=52`). C'est efficace.

Attention : des inéqui-jointures entraînent forcément des *nested loops* (pour chaque ligne d'une table, on va chercher les lignes d'une autre table). Sur de grands volumes, ce ne peut pas être efficace. Ici, tout va bien parce que la liste des dimanches est raisonnablement courte.

> On cherche un article à peu près au tiers de la liste avec la requête suivante. Pourquoi est-elle si lente ?

```
SELECT * FROM posts
WHERE id_article =
    (SELECT max(id_article) * 0.333
     FROM posts
     ) ;
```

Le plan[21] est :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on posts  (cost=0.48..166135.48 rows=25000 width=115)
                    (actual time=333.363..1000.696 rows=1000 loops=1)
   Filter: ((id_article)::numeric = $1)
   Rows Removed by Filter: 4999000
   InitPlan 2 (returns $1)
```

[20] https://explain.dalibo.com/plan/nN5
[21] https://explain.dalibo.com/plan/6GI

```
     ->  Result  (cost=0.46..0.48 rows=1 width=32)
                 (actual time=0.016..0.017 rows=1 loops=1)
           InitPlan 1 (returns $0)
             ->  Limit  (cost=0.43..0.46 rows=1 width=4)
                        (actual time=0.012..0.014 rows=1 loops=1)
                   ->  Index Only Scan Backward using posts_id_article_id_post
                                       on posts posts_1
                                       (cost=0.43..142352.43 rows=5000000 width=4)
                                       (actual time=0.012..0.012 rows=1 loops=1)
                         Index Cond: (id_article IS NOT NULL)
                         Heap Fetches: 0
 Planning Time: 0.097 ms
 Execution Time: 1000.753 ms
```

Ce plan indique une recherche du numéro d'article maximal (il est dans l'index ; noter que PostgreSQL restreint à une valeur non vide), puis il calcule la valeur correspondant au tiers et la met dans `$1`. Tout ceci est rapide. La partie lente est le `Seq Scan` pour retrouver cette valeur, avec un filtre et non par l'index.

Le problème est visible sur le filtre même :

```
Filter: ((id_article)::numeric = $1)
```

`(id_article)::numeric` signifie que tous les `id_article` (des entiers) sont convertis en `numeric` pour ensuite être comparés au `$1`. Or une conversion est une fonction, ce qui rend l'index inutilisable. En fait, notre problème est que `$1` n'est pas un entier !

```
SELECT max(id_article) * 0.333
     FROM posts
\gdesc
```

```
  Column   |  Type
-----------+---------
 ?column? | numeric
```

La conversion du critère en `int` peut se faire à plusieurs endroits. Par exemple :

```
SELECT * FROM posts
WHERE id_article =
    (SELECT max(id_article) * 0.333
     FROM posts
     )::int ;
```

Et l'index est donc utilisable immédiatement :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
  Index Scan using posts_id_article_id_post on posts
                           (cost=0.91..1796.42 rows=1007 width=115)
                           (actual time=0.031..0.198 rows=1000 loops=1)
   Index Cond: (id_article = ($1)::integer)
   InitPlan 2 (returns $1)
     ->  Result  (cost=0.46..0.48 rows=1 width=32) (...)
           InitPlan 1 (returns $0)
             ->  Limit  (cost=0.43..0.46 rows=1 width=4) (...)
```

```
                  ->  Index Only Scan Backward using posts_id_article_id_post
                                      on posts posts_1  (...)
                        Index Cond: (id_article IS NOT NULL)
                        Heap Fetches: 0
Planning Time: 0.105 ms
Execution Time: 0.245 ms
```

Si l'on avait fait le calcul avec `/ 3` au lieu de `* 0.333`, on n'aurait pas eu le problème, car la division de deux entiers donne un entier.

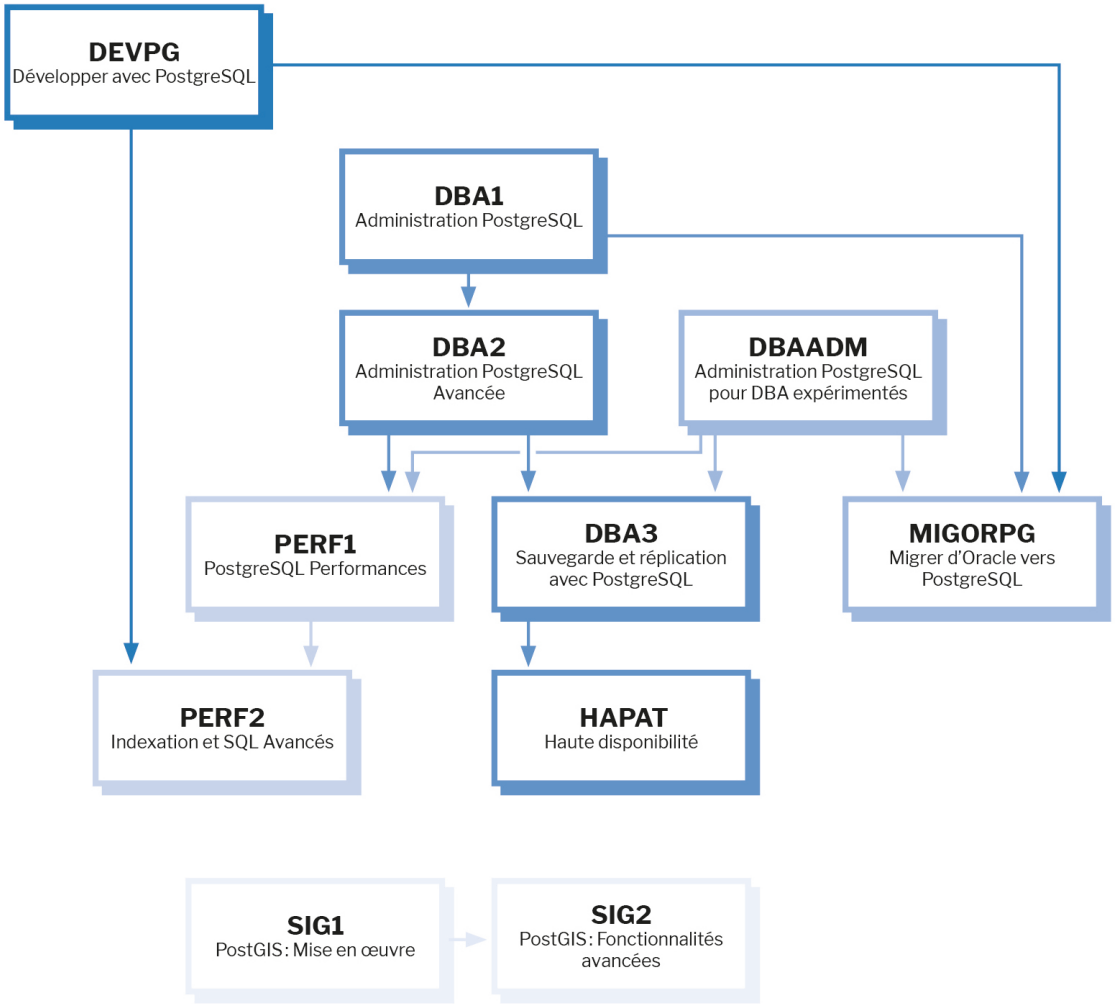
Attention donc à la cohérence des types dans vos critères. Le problème peut se rencontrer même en joignant des `int` et des `bigint` !

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

— DBA1 : Administration PostgreSQL
https://dali.bo/dba1
— DBA2 : Administration PostgreSQL avancé
https://dali.bo/dba2
— DBA3 : Sauvegarde et réplication avec PostgreSQL
https://dali.bo/dba3
— DEV1 : Introduction à SQL
https://dali.bo/dev1
— DEVPG : Développer avec PostgreSQL
https://dali.bo/devpg
— PERF1 : PostgreSQL Performances
https://dali.bo/perf1
— PERF2 : Indexation et SQL avancés
https://dali.bo/perf2
— MIGORPG : Migrer d'Oracle à PostgreSQL
https://dali.bo/migorpg
— HAPAT : Haute disponibilité avec PostgreSQL
https://dali.bo/hapat

Les livres blancs

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

Téléchargement gratuit

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