Formation SQL1

SQL conception et mise en œuvre

17.12

Dalibo SCOP

https://dalibo.com/formations

SQL conception et mise en œuvre

Formation SQL1

TITRE : SQL conception et mise en œuvre
SOUS-TITRE : Formation SQL1

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-05-0
COPYRIGHT: © 2005-2017 DALIBO SARL SCOP
LICENCE: Creative Commons BY-NC-SA

Le logo éléphant de PostgreSQL (”Slonik”) est une création sous copyright et
le nom ”PostgreSQL” est marque déposée par PostgreSQL Community Association
of Canada.

Remerciements : Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - **Pas d'Utilisation Commerciale** - **Partage dans les Mêmes Conditions**

Vous êtes autorisé :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale: Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note: Ceci est un résumé de la licence.

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos 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 !

Contents

Licence Creative Commons BY-NC-SA 2.0 FR 5

1 Licence Creative Commons CC-BY-NC-SA 11

2 Introduction et premiers SELECT 12

2.1 Préambule 12
2.2 Principes d'une base de données 12
2.3 Lecture de données 21
2.4 Types de données 37
2.5 Conclusion 51
2.6 Travaux Pratiques 52

3 Création d'objet et mises à jour 60

3.1 Introduction 60
3.2 DDL 60
3.3 DML : mise à jour des données 91
3.4 Transactions 96
3.5 Conclusion 100
3.6 Travaux Pratiques 100

4 Plus loin avec SQL 107

4.1 Préambule 107
4.2 Valeur NULL 107
4.3 Agrégats 113
4.4 Sous-requêtes 118
4.5 Jointures 125
4.6 Expressions CASE 133
4.7 Opérateurs ensemblistes 135
4.8 Conclusion 138
4.9 Travaux Pratiques 139

5 Approfondissement SQL 154

5.1 Fonctions de base 154
5.2 Vues 166
5.3 Requêtes préparées 174
5.4 Indexation 176
5.5 Ce qu'il ne faut pas faire 180
5.6 Conclusion 201
5.7 Travaux Pratiques 202

6 Comprendre EXPLAIN **207**
6.1 Introduction 207
6.2 Exécution globale d'une requête 208
6.3 Quelques définitions 212
6.4 Planificateur 214
6.5 Mécanisme de coûts 221
6.6 Statistiques 222
6.7 Qu'est-ce qu'un plan d'exécution ? 235
6.8 Nœuds d'exécution les plus courants 245
6.9 Problèmes les plus courants 257
6.10 Outils 268
6.11 Conclusion 274
6.12 Annexe : Nœuds d'un plan 275
6.13 Travaux Pratiques 306

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse: http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 INTRODUCTION ET PREMIERS SELECT

2.1 PRÉAMBULE

- Qu'est-ce que le standard SQL ?
- Comment lire des données
- Quel type de données est disponible ?

Ce module a pour but de présenter le standard SQL. Un module ne permet pas de tout voir, aussi ce module se concentrera sur la lecture de données déjà présentes en base. Cela permet d'aborder aussi la question des types de données disponibles.

2.1.1 MENU

- Principes d'une base de données
- Premières requêtes
- Connaître les types de données

2.1.2 OBJECTIFS

- Comprendre les principes
- Écrire quelques requêtes en lecture
- Connaître les différents types de données
  - et quelques fonctions très utiles

2.2 PRINCIPES D'UNE BASE DE DONNÉES

- Base de données
  - ensemble organisé d'informations
- Système de Gestion de Bases de Données
  - acronyme SGBD (DBMS en anglais)
  - programme assurant la gestion et l'accès à une base de données
  - assure la cohérence des données

Si des données sont récoltées, organisées et stockées afin de répondre à un besoin spécifique, alors on parle de base de données. Une base de données peut utiliser différents supports : papier, fichiers informatiques, etc.

Le Système de Gestion de Bases de Données (SGBD), appelé *Database Management System* (DBMS) en anglais, assure la gestion d'une base de données informatisée. Il permet l'accès aux données et assure également la cohérence des données.

2.2.1 TYPE DE BASES DE DONNÉES

- Modèle hiérarchique
- Modèle réseau
- Modèle relationnel
- Modèle objet
- Modèle relationnel-objet
- NoSQL

Au fil des années ont été développés plusieurs modèles de données, que nous allons décrire.

2.2.2 TYPE DE BASES DE DONNÉES (1)

- Modèle hiérarchique
  - structure arborescente
  - redondance des données
- Modèle réseau
  - structure arborescente, mais permettant des associations
  - ex : Bull IDS2 sur GCOS

Les modèles hiérarchiques et réseaux ont été les premiers modèles de données utilisées dans les années 60 sur les mainframes IBM ou Bull. Ils ont été rapidement supplantés par le modèle relationnel car les requêtes étaient dépendantes du modèle de données. Il était nécessaire de connaître les liens entre les différents nœuds de l'arborescence pour concevoir les requêtes. Les programmes sont donc complètement dépendants de la structure de la base de données.

Des recherches souhaitent néanmoins arriver à rendre indépendant la vue logique de l'implémentation physique de la base de données.

2.2.3 TYPE DE BASES DE DONNÉES (2)

- Modèle relationnel
  - basé sur la théorie des ensembles et la logique des prédicats
  - standardisé par la norme SQL
- Modèle objet
  - structure objet
  - pas de standard
- Modèle relationnel-objet
  - le standard SQL ajoute des concepts objets

Le modèle relationnel est issu des travaux du Docteur Edgar F. Codd qu'il a menés dans les laboratoires d'IBM à la fin des années 60. Ses travaux avaient pour but de rendre indépendant le stockage physique de la vue logique de la base de données. Et, mathématicien de formation, il s'est appuyé sur la théorie des ensembles et la logique des prédicats pour établir les fondements des bases de données relationnelles. Pour manipuler les données de façon ensembliste, le Dr Codd a mis au point le point langage SQL. Ce langage est à l'origine du standard SQL qui a émergé dans les années 80 et qui a rendu le modèle relationnel très populaire.

Le modèle objet est, quant à lui, issu de la mouvance autour des langages objets. Du fait de l'absence d'un standard avéré, le modèle objet n'a jamais été populaire et est toujours resté dans l'ombre du modèle relationnel.

Le modèle relationnel a néanmoins était étendu par la norme SQL:1999 pour intégrer des fonctionnalités objets. On parle alors de modèle relationnel-objet. PostgreSQL en est un exemple, c'est un SGBDRO (Système de Gestion de Bases de Données Relationnel-Objet).

2.2.4 TYPE DE BASES DE DONNÉES (3)

- NoSQL : « *Not only SQL* »
  - pas de norme de langage de requête
  - clé-valeur (Redis, Riak)
  - graphe (Neo4J)
  - document (MongoDB, CouchDB)
  - orienté colonne (HBase)
- Rapprochement relationnel/NoSQL

  – PostgreSQL permet de stocker des documents (JSON, XML)

Les bases NoSQL désigne une famille de bases de données qui répondent à d'autres besoins et contraintes que les bases relationnelles. Les bases NoSQL sont souvent des bases « sans schéma », la base ne vérifiant plus l'intégrité des données selon des contraintes définies dans le modèle de données. Chaque base de ce segment dispose d'un langage de requête spécifique, qui n'est pas normé. Une tentative de standardisation, débutée en 2011, n'a d'ailleurs abouti à aucun résultat.

Ce type de base offre souvent la possibilité d'offrir du *sharding* simple à mettre en œuvre. Le sharding consiste à répartir les données physiquement sur plusieurs serveurs. Certaines technologies semblent mieux marcher que d'autres de ce point de vue là. En contrepartie, la durabilité des données n'est pas assurée, au contraire d'une base relationnelle qui assure la durabilité dès la réponse à un `COMMIT`.

Exemple de requête SQL :

```
SELECT person, SUM(score), AVG(score), MIN(score), MAX(score), COUNT(*)
FROM demo
WHERE score > 0 AND person IN('bob','jake')
GROUP BY person;
```

La même requête, pour MongoDB :

```
db.demo.group({
    "key": {
        "person": true
    },
    "initial": {
        "sumscore": 0,
        "sumforaverageaveragescore": 0,
        "countforaverageaveragescore": 0,
        "countstar": 0
    },
    "reduce": function(obj, prev) {
        prev.sumscore = prev.sumscore + obj.score - 0;
        prev.sumforaverageaveragescore += obj.score;
        prev.countforaverageaveragescore++;
        prev.minimumvaluescore = isNaN(prev.minimumvaluescore) ? obj.score :
            Math.min(prev.minimumvaluescore, obj.score);
        prev.maximumvaluescore = isNaN(prev.maximumvaluescore) ? obj.score :
            Math.max(prev.maximumvaluescore, obj.score);
        if (true != null) if (true instanceof Array) prev.countstar +=
            true.length;
        else prev.countstar++;
    },
    "finalize": function(prev) {
```

```
        prev.averagescore = prev.sumforaverageaveragescore /
            prev.countforaverageaveragescore;
        delete prev.sumforaverageaveragescore;
        delete prev.countforaverageaveragescore;
    },
    "cond": {
        "score": {
            "$gt": 0
        },
        "person": {
            "$in": ["bob", "jake"]
        }
    }
});
```

Un des avantages de ces technologies, c'est qu'un modèle clé-valeur permet facilement d'utiliser des algorithmes de type MapReduce : diviser le problème en sous-problèmes traités parallèlement par différents nœuds (phase Map), puis synthétisés de façon centralisée (phase Reduce).

Les bases de données relationnelles ne sont pas incompatibles avec Map Reduce en soit. Simplement, le langage SQL étant déclaratif, il est conceptuellement opposé à la description fine des traitements qu'on doit réaliser avec MapReduce. C'est (encore une fois) le travail de l'optimiseur d'être capable d'effectuer ce genre d'opérations. On peut penser au mode "parallèle" qu'on trouve dans certains SGBD comme Oracle (et que PostgreSQL n'a pas encore), ou à des solutions comme Postgres-XC. On peut aussi utiliser des technologies comme PlProxy, mais on perd l'avantage du côté déclaratif de SQL, puisqu'on utilise alors uniquement des procédures stockées.

2.2.5 MODÈLE RELATIONNEL

- Indépendance entre la vue logique et la vue physique
  - le SGBD gère lui-même le stockage physique
- Table ou *relation*
- Un ensemble de tables représente la vue logique

Le modèle relationnel garantit l'indépendance entre la vue logique et la vue physique. L'utilisateur ne se préoccupe que des objets logiques (pour lire ou écrire des enregistrements), et le SGBD traduit la demande exprimée avec des objets logiques en actions à réaliser sur des objets physiques.

Les objets logiques sont appelés des relations. Ce sont généralement les tables, mais il

existe d'autres objets qui sont aussi des relations (les vues par exemple, mais aussi les index et les séquences).

2.2.6 CARACTÉRISTIQUES DU MODÈLE RELATIONNEL

- Théorie des ensembles
- Logique des prédicats
- Logique 3 états

Le modèle relationnel se base sur la théorie des ensembles. Chaque relation contient un ensemble de données et ces différents ensembles peuvent se joindre suivant certaines conditions.

La logique des prédicats est un sous-ensemble de la théorie des ensembles. Elle sert à exprimer des formules logiques qui permettent de filtrer les ensembles de départ pour créer de nouveaux ensembles (autrement dit, filtrer les enregistrements d'une relation).

Cependant, tout élément d'un enregistrement n'est pas forcément connu à un instant t. Les filtres et les jointures doivent donc gérer trois états lors d'un calcul de prédicat : vrai, faux ou inconnu.

2.2.7 ACID

- **Atomicité** (Atomic)
- **Cohérence** (Consistent)
- **Isolation** (Isolated)
- **Durabilité** (Durable)

Les propriétés ACID (acronyme de "Atomic Consistent Isolated Durable") sont le fondement même de toute base de donnée. Il s'agit de quatre règles fondamentales que toute transaction doit respecter :

- A : Une transaction est entière : « tout ou rien ».
- C : Une transaction amène la base d'un état stable à un autre.
- I : Les transactions n'agissent pas les unes sur les autres.
- D : Une transaction validée provoque des changements permanents.

Les propriétés ACID sont quatre propriétés essentielles d'un sous-système de traitement de transactions d'un système de gestion de base de données. On considère parfois que seuls les SGBD qui respectent ces quatre propriétés sont dignes d'être considérées

comme des bases de données *relationnelles*. Les SGBD de la famille des NoSQL (MongoDB, Cassandra, BigTable…) sont en effet des bases de données, mais ne respectent pas la Cohérence. Elles sont cohérentes à terme, ou en anglais *eventually consistent*, mais la cohérence en fin de transaction n'est pas garantie.

2.2.8 LANGAGE SQL

- Norme ISO 9075
  - dernière version stable : 2016
- Langage déclaratif
  - on décrit le résultat et pas la façon de l'obtenir
  - comme Prolog
- Traitement ensembliste
  - par opposition au traitement procédural
  - « on effectue des opérations sur des relations pour obtenir des relations »

Le langage SQL a été normalisé par l'ANSI en 1986 et est devenu une norme ISO internationale en 1987. Elle a subi plusieurs évolutions dans le but d'ajouter des fonctionnalités correspondant aux attentes de l'industrie logicielle. Parmi ces améliorations, notons l'intégration de quelques fonctionnalités objets pour le modèle relationnel-objet.

2.2.9 SQL EST UN LANGAGE

- Langage
  - règles d'écriture
  - règles de formatage
  - commentaires
- Améliore la lisibilité d'une requête

Il n'y a pas de règles établies concernant l'écriture de requêtes SQL. Il faut néanmoins avoir à l'esprit qu'il s'agit d'un langage à part entière et, au même titre que ce qu'un développeur fait avec n'importe quel code source, il convient de l'écrire de façon lisible.

2.2.10 RECOMMANDATIONS D'ÉCRITURE ET DE FORMATAGE

- Écriture

  - mots clés SQL en MAJUSCULES
  - identifiants de colonnes/tables en minuscule
- Formatage
  - dissocier les éléments d'une requête
  - un prédicat par ligne
  - indentation

Quelle est la requête la plus lisible ?

celle-ci ?

```
select  groupeid,datecreationitem from itemagenda where typeitemagenda = 5  and
groupeid in(12225,12376) and datecreationitem > now() order by groupeid,
datecreationitem ;
```

ou celle-ci ?

```
SELECT groupeid, datecreationitem
  FROM itemagenda
 WHERE typeitemagenda = 5
   AND groupeid IN (12225,12376)
   AND datecreationitem > now()
 ORDER BY groupeid, datecreationitem;
```

Cet exemple est tiré du forum postgresql.fr[1] .

2.2.11 COMMENTAIRES

- Commentaire sur le reste de la ligne

```
-- commentaire
```

- Commentaire dans un bloc

```
/* bloc
 */
```

Une requête SQL peut être commentée au même titre qu'un programme standard.

Le marqueur -- permet de signifier à l'analyseur syntaxique que le reste de la ligne est commenté, il n'en tiendra donc pas compte dans son analyse de la requête.

Un commentaire peut aussi se présenter sous la forme d'un bloc de commentaire, le bloc pouvant occuper plusieurs lignes :

[1] http://forum.postgresql.fr/viewtopic.php?id=2610

```
/* Ceci est un commentaire
   sur plusieurs
   lignes
*/
```

Aucun des éléments compris entre le marqueur de début de bloc `/*` et le marqueur de fin de bloc `*/` ne sera pris en compte. Certains SGBDR propriétaires utilisent ces commentaires pour y placer des informations (appelées `hints` sur Oracle) qui permettent d'influencer le comportement de l'optimiseur, mais PostgreSQL ne possède pas ce genre de mécanisme.

2.2.12 LES 4 TYPES D'ORDRES SQL

- DDL
  - Data Definition Language
  - définit les structures de données
- DML
  - Data Manipulation Language
  - manipule les données
- DCL
  - Data Control Language
  - contrôle l'accès aux données
- TCL
  - Transaction Control Language
  - contrôle les transactions

Le langage SQL est divisé en quatre sous-ensembles qui ont chacun un but différent.

Les ordres DDL (pour `Data Definition Language`) permettent de définir les structures de données. On retrouve les ordres suivants :

- `CREATE` : crée un objet
- `ALTER` : modifie la définition d'un objet
- `DROP` : supprime un objet
- `TRUNCATE` : vide un objet
- `COMMENT` : ajoute un commentaire sur un objet

Les ordres DML (pour `Data Manipulation Language`) permettent l'accès et la modification des données. On retrouve les ordres suivants :

- `SELECT` : lit les données d'une ou plusieurs tables
- `INSERT` : ajoute des données dans une table

- UPDATE : modifie les données d'une table
- DELETE : supprime les données d'une table

Les ordres DCL (pour Data Control Language) permettent de contrôler l'accès aux données. Ils permettent plus précisément de donner ou retirer des droits à des utilisateurs ou des groupes sur les objets de la base de données :

- GRANT : donne un droit d'accès à un rôle sur un objet
- REVOKE : retire un droit d'accès d'un rôle sur un objet

Enfin, les ordres TCL (pour Transaction Control Language) permettent de contrôler les transactions :

- BEGIN : ouvre une transaction
- COMMIT : valide les traitements d'une transaction
- ROLLBACK : annule les traitements d'une transaction
- SAVEPOINT : crée un point de reprise dans une transaction
- SET TRANSACTION : modifie les propriétés d'une transaction en cours

---

2.3 LECTURE DE DONNÉES

- Ordre SELECT
  - lecture d'une ou plusieurs tables
  - ou appel de fonctions

La lecture des données se fait via l'ordre SELECT. Il permet de récupérer des données d'une ou plusieurs tables (il faudra dans ce cas joindre les tables). Il permet aussi de faire appel à des fonctions stockées en base.

2.3.1 SYNTAXE DE SELECT

```
SELECT expressions_colonnes
 [ FROM elements_from ]
 [ WHERE predicats ]
 [ ORDER BY expressions_orderby ]
 [ LIMIT limite ]
 [ OFFSET offset ];
```

L'ordre SELECT est composé de différents éléments dont la plupart sont optionnels. L'exemple de syntaxe donné ici n'est pas complet.

17.12

La syntaxe complète de l'ordre `SELECT` est disponible dans le manuel de PostgreSQL[2] .

2.3.2 LISTE DE SÉLECTION

- Description du résultat de la requête
  - colonnes retournées
  - renommage
  - dédoublonnage

La liste de sélection décrit le format de la table virtuelle qui est retournée par l'ordre `SELECT`. Les types de données des colonnes retournées seront conformes au type des éléments donnés dans la liste de sélection.

2.3.3 COLONNES RETOURNÉES

- Liste des colonnes retournées
  - expression
  - séparées par une virgule
- Expression
  - constante
  - référence de colonne :
    `table.colonne`
- opération sur des colonnes et/ou des constantes

La liste de sélection décrit le format de la table virtuelle qui est retournée par l'ordre `SELECT`. Cette liste est composée d'expressions séparées par une virgule.

Chaque expression peut être une simple constante, peut faire référence à des colonnes d'une table lue par la requête, et peut être un appel à une fonction.

Une expression peut être plus complexe. Par exemple, elle peut combiner plusieurs constantes et/ou colonnes à l'aide d'opérations. Parmi les opérations les plus classiques, les opérateurs arithmétiques classiques sont utilisables pour les données numériques. L'opérateur de concaténation permet de concaténer des chaînes de caractères.

L'expression d'une colonne peut être une constante :

```
SELECT 1;
 ?column?
```

[2] http://docs.postgresql.fr/current/sql-select.html

```
----------
        1
(1 row)
```

Elle peut aussi être une référence à une colonne d'une table :

```
SELECT appellation.libelle
  FROM appellation;
```

Comme il n'y a pas d'ambiguïté avec la colonne `libelle`, la référence de la colonne `appellation.libelle` peut être simplifiée en `libelle` :

```
SELECT libelle
  FROM appellation;
```

Le SGBD saura déduire la table et la colonne mises en œuvre dans cette requête. Il faudra néanmoins utiliser la forme complète `table.colonne` si la requête met en œuvre des tables qui possèdent des colonnes qui portent des noms identiques.

Une requête peut sélectionner plusieurs colonnes. Dans ce cas, les expressions de colonnes sont définies sous la forme d'une liste dont chaque élément est séparé par une virgule :

```
SELECT id, libelle, region_id
  FROM appellation;
```

Le joker `*` permet de sélectionner l'ensemble des colonnes d'une table, elles apparaitront dans leur ordre physique (attention si l'ordre change !) :

```
SELECT *
  FROM appellation;
```

Si une requête met en œuvre plusieurs tables, on peut choisir de retourner toutes les colonnes d'une seule table :

```
SELECT appellation.*
  FROM appellation;
```

Enfin, on peut récupérer un tuple entier de la façon suivante :

```
SELECT appellation
  FROM appellation;
```

Une expression de colonne peut également être une opération, par exemple une addition :

```
SELECT 1 + 1;
 ?column?
----------
        2
(1 row)
```

Ou une soustraction :

17.12

```sql
SELECT annee, nombre - 10
  FROM stock;
```

2.3.4 ALIAS DE COLONNE

- Renommage
  - ou alias
  - AS :

    `expression AS alias`
- le résultat portera le nom de l'alias

Afin de pouvoir nommer de manière adéquate les colonnes du résultat d'une requête SELECT, le mot clé AS permet de définir un alias de colonne. Cet alias sera utilisé dans le résultat pour nommer la colonne en sortie :

```sql
SELECT 1 + 1 AS somme;
 somme
-------
     2
(1 row)
```

Cet alias n'est pas utilisable dans le reste de la requête (par exemple dans la clause WHERE).

2.3.5 DÉDOUBLONNAGE DES RÉSULTATS

- Dédoublonnage des résultats avant de les retourner
  - DISTINCT
  - à ne pas utiliser systématiquement
- `SELECT DISTINCT expressions_colonnes...`

Par défaut, SELECT retourne tous les résultats d'une requête. Parfois, des doublons peuvent se présenter dans le résultat. La clause DISTINCT permet de les éviter en réalisant un dédoublonnage des données avant de retourner le résultat de la requête.

Il faut néanmoins faire attention à l'utilisation systématique de la clause DISTINCT. En effet, elle entraîne un tri systématique des données juste avant de retourner les résultats de la requête, ce qui va consommer de la ressource mémoire, voire de la ressource disque si le volume de données à trier est important. De plus, cela va augmenter le temps de réponse de la requête du fait de cette opération supplémentaire.

En règle générale, la clause DISTINCT devient inutile lorsqu'elle doit trier un ensemble qui contient des colonnes qui sont déjà uniques. Si une requête récupère une clé primaire, les données sont uniques par définition. Le SELECT DISTINCT sera alors transformé en simple SELECT.

2.3.6 DÉRIVATION

- SQL permet de dériver les valeurs des colonnes
  - opérations arithmétiques : +, -, /, *
  - concaténation de chaînes : ||
  - appel de fonction

Les constantes et valeurs des colonnes peuvent être dérivées selon le type des données manipulées.

Les données numériques peuvent être dérivées à l'aide des opérateurs arithmétiques standards : +, -, /, *. Elles peuvent faire l'objet d'autres calculs à l'aide de fonctions internes et de fonctions définies par l'utilisateur.

La requête suivante permet de calculer le volume total en litres de vin disponible dans le stock du caviste :

```
SELECT SUM(c.contenance * s.nombre) AS volume_total
  FROM stock s
  JOIN contenant c
    ON (contenant_id=c.id);
```

Les données de type chaînes de caractères peuvent être concaténées à l'aide de l'opérateur dédié ||. Cet opérateur permet de concaténer deux chaînes de caractères mais également des données numériques avec une chaîne de caractères.

Dans la requête suivante, l'opérateur de concaténation est utilisé pour ajouter l'unité. Le résultat est ainsi implicitement converti en chaîne de caractères.

```
SELECT SUM(c.contenance * s.nombre) || ' litres' AS volume_total
  FROM stock AS s
  JOIN contenant AS c
    ON (contenant_id=c.id);
```

De manière générale, il n'est pas recommandé de réaliser les opérations de formatage des données dans la base de données. La base de données ne doit servir qu'à récupérer les résultats, le formatage étant assuré par l'application.

17.12

Différentes fonctions sont également applicables aux chaînes de caractères, de même qu'aux autres types de données.

2.3.7 FONCTIONS UTILES

- Fonctions sur données temporelles :
  - date et heure courante : `now()`
  - âge : `age(timestamp)`
  - extraire une partie d'une date : `extract( 'year' FROM timestamp)`
  - ou `date_part('Y',timestamp)`
- Fonctions sur données caractères :
  - longueur d'une chaîne de caractère : `char_length(chaine)`
- Compter les lignes : `count(*)`

Parmi les fonctions les plus couramment utilisés, la fonction `now()` permet d'obtenir la date et l'heure courante. Elle ne prend aucun argument. Elle est souvent utilisée, notamment pour affecter automatiquement la valeur de l'heure courante à une colonne.

La fonction `age(timestamp)` permet de connaître l'âge d'une date par rapport à la date courante.

La fonction `char_length(varchar)` permet de connaître la longueur d'une chaîne de caractère.

Enfin, la fonction `count(*)` permet de compter le nombre de lignes. Il s'agit d'une fonction d'agrégat, il n'est donc pas possible d'afficher les valeurs d'autres colonnes sans faire appel aux capacités de regroupement des lignes de SQL.

Exemples

Affichage de l'heure courante :

```
SELECT now();
              now
------------------------------
 2017-08-29 14:45:17.213097+02
```

Affichage de l'âge du 1er janvier 2000 :

```
SELECT age(date '2000-01-01');
          age
------------------------
 17 years 7 mons 28 days
```

Affichage de la longueur de la chaîne "Dalibo" :

```
SELECT char_length('Dalibo');
 char_length
-------------
           6
```

Affichage du nombre de lignes de la table `vin` :

```
SELECT count(*) FROM vin;
 count
-------
  6067
```

2.3.8 CLAUSE FROM

```
FROM expression_table [, expression_table ...]
```

- Description des tables mises en œuvre dans la requête
  - une seule table
  - plusieurs tables jointes
  - sous-requête

La clause `FROM` permet de lister les tables qui sont mises en œuvres dans la requêtes `SELECT`. Il peut s'agir d'une table physique, d'une vue ou d'une sous-requête. Le résultat de leur lecture sera une table du point de vue de la requête qui la met en œuvre.

Plusieurs tables peuvent être mises en œuvre, généralement dans le cadre d'une jointure.

2.3.9 ALIAS DE TABLE

- mot-clé `AS`
  - optionnel :

    ```
    reference_table alias
    ```

- la table sera ensuite référencée par l'alias

  ```
  reference_table [AS] alias
  ```

  ```
  reference_table AS alias (alias_colonne1, ...)
  ```

De la même façon qu'on peut créer des alias de colonnes, on peut créer des alias de tables. La table sera ensuite référencée uniquement par cet alias dans la requête. Elle ne pourra

plus être référencée par son nom réel. L'utilisation du nom réel provoquera d'ailleurs une erreur.

Le mot clé AS permet de définir un alias de table. Le nom réel de la table se trouve à gauche, l'alias se trouve à droite. L'exemple suivant définie un alias reg sur la table region :

```
SELECT id, libelle
  FROM region AS reg;
```

Le mot clé AS est optionnel :

```
SELECT id, libelle
  FROM region reg;
```

La requête suivante montre l'utilisation d'un alias pour les deux tables mises en œuvre dans la requête. La table stock a pour alias s et la table contenant a pour alias c. Les deux tables possèdent toutes les deux une colonnes id, ce qui peut poser une ambiguïté dans la clause de jointure (ON (contenant_id=c.id)). La condition de jointure portant sur la colonne contenant_id de la table stock, son nom est unique et ne porte pas à ambiguïté. La condition de jointure porte également sur la colonne id de table contenant, il faut préciser le nom complet de la colonne en utilisant le préfixe c pour la nommer : c.id.

```
SELECT SUM(c.contenance * s.nombre) AS volume_total
  FROM stock s
  JOIN contenant c
    ON (contenant_id=c.id);
```

Enfin, la forme reference_table AS alias (alias_colonne1, ...) permet de définir un alias de table et définir par la même occasion des alias de colonnes. Cette forme est peu recommandé car les alias de colonnes dépendent de l'ordre physique de ces colonnes. Cet ordre peut changer dans le temps et donc amener à des erreurs :

```
SELECT id_region, nom_region
  FROM region AS reg (id_region, nom_region);
```

2.3.10 NOMMAGE DES OBJETS

- Noms d'objets convertis en minuscules
  - Nom_Objet devient nom_objet
  - certains nécessitent l'emploi de majuscules
- Le guillemet double " conserve la casse
  - "Nom_Objet"

Avec PostgreSQL, les noms des objets sont automatiquement convertis en minuscule, sauf s'ils sont englobés entre des guillemets doubles. Si jamais ils sont créés avec une

casse mixte en utilisant les guillemets doubles, chaque appel à cet objet devra utiliser la bonne casse et les guillemets doubles. Il est donc conseillé d'utiliser une notation des objets ne comprenant que des caractères minuscules.

Il est aussi préférable de ne pas utiliser d'accents ou de caractères exotiques dans les noms des objets.

2.3.11 CLAUSE WHERE

- Permet d'exprimer des conditions de filtrage
  - prédicats
- Un prédicat est une opération logique
  - renvoie vrai ou faux
- La ligne est présente dans le résultat
  - si l'expression logique des prédicats est vraie

La clause `WHERE` permet de définir des conditions de filtrage des données. Ces conditions de filtrage sont appelées des prédicats.

Après le traitement de la clause `FROM`, chaque ligne de la table virtuelle dérivée est vérifiée avec la condition de recherche. Si le résultat de la vérification est positif (true), la ligne est conservée dans la table de sortie, sinon (c'est-à-dire si le résultat est faux ou nul) la ligne est ignorée.

La condition de recherche référence typiquement au moins une colonne de la table générée dans la clause FROM ; ceci n'est pas requis mais, dans le cas contraire, la clause WHERE n'aurait aucune utilité.

2.3.12 EXPRESSION ET OPÉRATEURS DE PRÉDICATS

- Comparaison
  - `=`, `<`, `>`, `=<`, `>=`, `<>`
- Négation
  - `NOT`

```
expression operateur_comparaison expression
```

Un prédicat est composé d'une expression qui est soumise à un opérateur de prédicat pour être éventuellement comparé à une autre expression. L' opérateur de prédicat re-

tourne alors `true` si la condition est vérifiée ou `false` si elle ne l'est pas ou `NULL` si son résultat ne peut être calculé.

Les opérateurs de comparaison sont les opérateurs de prédicats les plus souvent utilisés. L'opérateur d'égalité `=` peut être utilisé pour vérifier l'égalité de l'ensemble des types de données supportés par PostgreSQL. Il faudra faire attention à ce que les données comparées soient de même type.

L'opérateur `NOT` est une négation. Si un prédicat est vrai, l'opérateur `NOT` retournera faux. À l'inverse, si un prédicat est faux, l'opérateur `NOT` retournera vrai. La clause `NOT` se place devant l'expression entière.

Exemples

Sélection de la région dont l'identifiant est égal à 3 (en ensuite différent de 3) :

```
SELECT *
  FROM region
 WHERE id = 3;

SELECT *
  FROM region
 WHERE NOT id = 3;
```

2.3.13 COMBINER DES PRÉDICATS

- OU logique
  - `predicat OR predicat`
- ET logique
  - `predicat AND predicat`

Les opérateurs logiques `OR` et `AND` permettent de combiner plusieurs prédicats dans la clause `WHERE`.

L'opérateur `OR` est un OU logique. Il retourne vrai si au moins un des deux prédicats combinés est vrai. L'opérateur `AND` est un ET logique. Il retourne vrai si et seulement si les deux prédicats combinés sont vrais.

Au même titre qu'une multiplication ou une division sont prioritaires sur une addition ou une soustraction dans un calcul, l'évaluation de l'opérateur `AND` est prioritaire sur celle de l'opérateur `OR`. Et, tout comme dans un calcul, il est possible de protéger les opérations prioritaires en les encadrant de parenthèses.

Exemples

Dans le stock, affiche les vins dont le nombre de bouteilles est inférieur à 2 ou supérieur à 16 :

```
SELECT *
  FROM stock
 WHERE nombre < 2
    OR nombre > 16;
```

2.3.14 CORRESPONDANCE DE MOTIF

- Comparaison de motif

  ```
  chaine LIKE motif ESCAPE 'c'
  ```

- % : toute chaine de 0 à plusieurs caractères
  - _ : un seul caractère
- Expression régulière POSIX

  ```
  chaine ~ motif
  ```

L'opérateur LIKE permet de réaliser une recherche simple sur motif. La chaine exprimant le motif de recherche peut utiliser deux caractères joker : _ et %. Le caractère _ prend la place d'un caractère inconnu, qui doit toujours être présent. Le caractère % est un joker qui permet d'exprimer que PostgreSQL doit trouver entre 0 et plusieurs caractères.

Exploiter la clause LIKE avec un motif sans joker ne présente pas d'intérêt. Il est préférable dans ce cas d'utiliser l'opérateur d'égalité.

Le mot clé ESCAPE 'c' permet de définir un caractère d'échappement pour protéger les caractères _ et % qui font légitimement partie de la chaîne de caractère du motif évalué. Lorsque PostgreSQL rencontre le caractère d'échappement indiqué, les caractères _ et % seront évalués comme étant les caractères _ et % et non comme des jokers.

L'opérateur LIKE dispose d'une déclinaison qui n'est pas sensible à la casse. Il s'agit de l'opérateur ILIKE.

Exemples

Création d'un jeu d'essai :

```
CREATE TABLE motif (chaine varchar(30));
INSERT INTO motif (chaine) VALUES ('Durand'), ('Dupont'), ('Dupond'),
    ('Dupon'), ('Dupuis');
```

Toutes les chaînes commençant par la suite de caractères Dur :

17.12

```
SELECT * FROM motif WHERE chaine LIKE 'Dur%';
 chaine
--------
 Durand
```

Toutes les chaînes terminant par `d` :

```
SELECT * FROM motif WHERE chaine LIKE '%d';
 chaine
--------
 Durand
 Dupond
```

Toutes les chaînes qui commencent par `Dupon` suivi d'un caractère inconnu. La chaîne `Dupon` devrait être ignorée :

```
SELECT * FROM motif WHERE chaine LIKE 'Dupon_';
   chaine
------------
 Dupont
 Dupond
```

2.3.15 LISTES ET INTERVALLES

- Liste de valeurs

  ```
  expression IN (valeur1 [, ...])
  ```

- Chevauchement d'intervalle de valeurs

  ```
  expression BETWEEN expression AND expression
  ```

- Chevauchement d'intervalle de dates

  ```
  (date1, date2) OVERLAPS (date3, date4)
  ```

La clause `IN` permet de vérifier que l'expression de gauche est égale à une valeur présente dans l'expression de droite, qui est une liste d'expressions. La négation peut être utilisée en utilisant la construction `NOT IN`.

L'opérateur `BETWEEN` permet de vérifier que la valeur d'une expression est comprise entre deux bornes. Par exemple, l'expression `valeur BETWEEN 1 AND 10` revient à exprimer la condition suivante : `valeur >= 1 AND valeur<= 10`. La négation peut être utilisée en utilisant la construction `NOT BETWEEN`.

Exemples

Recherche les chaînes qui sont présentes dans la liste `IN` :

```
SELECT * FROM motif WHERE chaine IN ('Dupont', 'Dupond', 'Ducobu');
 chaine
---------
 Dupont
 Dupond
```

2.3.16 TRIS

- SQL ne garantit pas l'ordre des résultats
  - tri explicite requis
- Tris des lignes selon des expressions

  ```
  ORDER BY expression  [ ASC | DESC | USING opérateur ]
                       [ NULLS { FIRST | LAST } ] [, ...]
  ```

- ordre du tri : `ASC` ou `DESC`
  - placement des valeurs `NULL` : `NULLS FIRST` ou `NULLS LAST`
  - ordre de tri des caractères : `COLLATE collation`

La clause `ORDER BY` permet de trier les lignes du résultat d'une requête selon une ou plusieurs expressions combinées.

L'expression la plus simple est le nom d'une colonne. Dans ce cas, les lignes seront triées selon les valeurs de la colonne indiquée, et par défaut dans l'ordre ascendant, c'est-à-dire de la valeur la plus petite à la plus grande pour une donnée numérique ou temporelle, et dans l'ordre alphabétique pour une donnée textuelle.

Les lignes peuvent être triées selon une expression plus complexe, par exemple en dérivant la valeur d'une colonne.

L'ordre de tri peut être modifié à l'aide de la clause `DESC` qui permet un tri dans l'ordre descendant, donc de la valeur la plus grande à la plus petite (ou alphabétique inverse le cas échéant).

La clause `NULLS` permet de contrôler l'ordre d'apparition des valeurs `NULL`. La clause `NULLS FIRST` permet de faire apparaître d'abord les valeurs `NULL` puis les valeurs non `NULL` selon l'ordre de tri. La clause `NULLS LAST` permet de faire apparaître d'abord les valeurs non `NULL` selon l'ordre de tri suivies par les valeurs NULL. Si cette clause n'est pas précisée, alors PostgreSQL utilise implicitement `NULLS LAST` dans le cas d'un tri ascendant (`ASC`, par défaut) ou `NULLS FIRST` dans le cas d'un tri descendant (`DESC`, par défaut).

Exemples

Tri de la table `region` selon le nom de la région :

```
SELECT *
  FROM region
 ORDER BY libelle;
```

Tri de la table `stock` selon le nombre de bouteille, dans l'ordre décroissant :

```
SELECT *
  FROM stock
 ORDER BY nombre DESC;
```

Enfin, la clause `COLLATE` permet d'influencer sur l'ordre de tri des chaînes de caractères.

2.3.17 LIMITER LE RÉSULTAT

- Obtenir des résultats à partir de la ligne n
  - `OFFSET n`
- Limiter le nombre de lignes à n lignes
  - `FETCH {FIRST | NEXT} n ROWS ONLY`
  - `LIMIT n`
- Opérations combinables
  - `OFFSET` doit apparaitre avant `FETCH`
- Peu d'intérêt sur des résultats non triés

La clause `OFFSET` permet d'exclure les `n` premières lignes du résultat. Toutes les autres lignes sont ramenées.

La clause `FETCH` permet de limiter le résultat d'une requête. La requête retournera au maximum `n` lignes de résultats. Elle en retournera moins, voire aucune, si la requête ne peut ramener suffisamment de lignes. La clause `FIRST` ou `NEXT` est obligatoire mais le choix de l'une ou l'autre n'a aucune conséquence sur le résultat.

La clause `FETCH` est synonyme de la clause `LIMIT`. Mais `LIMIT` est une clause propre à PostgreSQL et quelques autres SGBD. Il est recommandé d'utiliser `FETCH` pour se conformer au standard.

Ces deux opérations peuvent être combinées. La norme impose de faire apparaître la clause `OFFSET` avant la clause `FETCH`. PostgreSQL permet néanmoins d'exprimer ces clauses dans un ordre différent, mais la requête ne pourra pas être portée sur un autre SGBD sans transformation.

Il faut faire attention au fait que ces fonctions ne permettent pas d'obtenir des résultats stables si les données ne sont pas triées explicitement. En effet, le standard SQL ne

garantie en aucune façon l'ordre des résultats à moins d'employer la clause `ORDER BY`.

Exemples

La fonction `generate_series` permet de générer une suite de valeurs numériques. Par exemple, une suite comprise entre 1 et 10 :

```
SELECT * FROM generate_series(1, 10);
 generate_series
-----------------
               1
(...)
              10
(10 rows)
```

La clause `FETCH` permet donc de limiter le nombre de lignes du résultats :

```
SELECT * FROM generate_series(1, 10) FETCH FIRST 5 ROWS ONLY;
 generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

La clause `LIMIT` donne un résultat équivalent :

```
SELECT * FROM generate_series(1, 10) LIMIT 5;
 generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

La clause `OFFSET 4` permet d'exclure les quatre premières lignes et de retourner les autres lignes du résultat :

```
SELECT * FROM generate_series(1, 10) OFFSET 4;
 generate_series
-----------------
               5
               6
               7
               8
               9
```

```
17.12
                10
(6 rows)
```

Les clauses `LIMIT` et `OFFSET` peuvent être combinées pour ramener les deux lignes en excluant les quatre premières :

```
SELECT * FROM generate_series(1, 10) OFFSET 4 LIMIT 2;
 generate_series
-----------------
               5
               6
(2 rows)
```

2.3.18 UTILISER PLUSIEURS TABLES

- Clause `FROM`
  - liste de tables séparées par `,`
- Une table est combinée avec une autre
  - jointure
  - produit cartésien

Il est possible d'utiliser plusieurs tables dans une requête `SELECT`. Lorsque c'est le cas, et sauf cas particulier, on fera correspondre les lignes d'une table avec les lignes d'une autre table selon certains critères. Cette mise en correspondance s'appelle une jointure et les critères de correspondances s'appellent une condition de jointure.

Si aucune condition de jointure n'est donnée, chaque ligne de la première table est mise en correspondance avec toutes les lignes de la seconde table. C'est un produit cartésien. En général, un produit cartésien n'est pas souhaitable et est généralement le résultat d'une erreur de conception de la requête.

Exemples

Création d'un jeu de données simple :

```
CREATE TABLE mere (id integer PRIMARY KEY, val_mere text);
CREATE TABLE fille (
    id_fille integer PRIMARY KEY,
    id_mere integer REFERENCES mere(id),
    val_fille text
);

INSERT INTO mere (id, val_mere) VALUES (1, 'mere 1');
INSERT INTO mere (id, val_mere) VALUES (2, 'mere 2');
```

```
INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (1, 1, 'fille 1');
INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (2, 1, 'fille 2');
```

Pour procéder à une jointure entre les tables `mere` et `fille`, les identifiants `id_mere` de la table `fille` doivent correspondre avec les identifiants `id` de la table `mere` :

```
SELECT * FROM mere, fille
 WHERE mere.id = fille.id_mere;
 id | val_mere | id_fille | id_mere | val_fille
----+----------+----------+---------+-----------
  1 | mere 1   |        1 |       1 | fille 1
  1 | mere 1   |        2 |       1 | fille 2
(2 rows)
```

Un produit cartésien est créé en omettant la condition de jointure, le résultat n'a plus de sens :

```
SELECT * FROM mere, fille;
 id | val_mere | id_fille | id_mere | val_fille
----+----------+----------+---------+-----------
  1 | mere 1   |        1 |       1 | fille 1
  1 | mere 1   |        2 |       1 | fille 2
  2 | mere 2   |        1 |       1 | fille 1
  2 | mere 2   |        2 |       1 | fille 2
(4 rows)
```

2.4 TYPES DE DONNÉES

- Type de données
  - du standard SQL
  - certains spécifiques PostgreSQL

PostgreSQL propose l'ensemble des types de données du standard SQL, à l'exception du type `BLOB` qui a toutefois un équivalent. Mais PostgreSQL a été conçu pour être extensible et permet de créer facilement des types de données spécifiques. C'est pourquoi PostgreSQL propose un certain nombre de types de données spécifiques qui peuvent être intéressants.

2.4.1 QU'EST-CE QU'UN TYPE DE DONNÉES ?

- Le système de typage valide les données

17.12

- Un type détermine
  - les valeurs possibles
  - comment les données sont stockées
  - les opérations que l'on peut appliquer

On utilise des types de données pour représenter une information de manière pertinente. Les valeurs possibles d'une donnée vont dépendre de son type. Par exemple, un entier long ne permet par exemple pas de coder des valeurs décimales. De la même façon, un type entier ne permet pas de représenter une chaîne de caractère, mais l'inverse est possible.

L'intérêt du typage des données est qu'il permet également à la base de données de valider les données manipulées. Ainsi un entier `integer` permet de représenter des valeurs comprises entre -2,147,483,648 et 2,147,483,647. Si l'utilisateur tente d'insérer une donnée qui dépasse les capacités de ce type de données, une erreur lui sera retournée. On retrouve ainsi la notion d'intégrité des données. Comme pour les langages de programmation fortement typés, cela permet de détecter davantage d'erreurs, plus tôt : à la compilation dans les langages typés, ou ici des la première exécution d'une requête, plutôt que plus tard, quand une chaîne de caractère ne pourra pas être convertie à la volée en entier par exemple.

Le choix d'un type de données va également influencer la façon dont les données sont représentées. En effet, toute donnée à une représentation textuelle et une représentation en mémoire et sur disque. Ainsi, un `integer` est représenté sous la forme d'une suite de 4 octets, manipulables directement par le processeur, alors que sa représentation textuelle est une suite de caractères. Cela a une implication forte sur les performances de la base de données.

Le type de données choisi permet également de déterminer les opérations que l'on pourra appliquer. Tous les types de données permettent d'utiliser des opérateurs qui leur sont propres. Ainsi il est possible d'additionner des entiers, de concaténer des chaines de caractères, etc. Si une opération ne peut être réalisée nativement sur le type de données, il faudra utiliser des conversions coûteuses. Vaut-il mieux additionner deux entiers issus d'une conversion d'une chaîne de caractère vers un entier ou additionner directement deux entiers ? Vaut-il mieux stocker une adresse IP avec un `varchar` ou avec un type de données dédié ?

Il est à noter que l'utilisateur peut contrôler lui-même certains types de données paramétrés. Le paramètre représente la longueur ou la précision du type de données. Ainsi, un type `varchar(15)` permettra de représenter des chaînes de caractères de 15 caractères maximum.

2.4.2 TYPES DE DONNÉES

- Types standards SQL
- Types dérivés
- Types spécifiques à PostgreSQL
- Types utilisateurs

Les types de données standards permettent de traiter la plupart des situations qui peuvent survenir. Dans certains cas, il peut être nécessaire de faire appel aux types spécifiques à PostgreSQL, par exemple pour stocker des adresses IP avec le type spécifique et bénéficier par la même occasion de toutes les classes d'opérateurs qui permettent de manipuler simplement ce type de données.

Et si cela ne s'avère pas suffisant, PostgreSQL permet à l'utilisateur de créer lui-même ses propres types de données, ainsi que les classes d'opérateurs et fonctions permettant d'indexer ces données.

2.4.3 TYPES STANDARDS (1)

- Caractère
  - `char`, `varchar`
- Numérique
  - `integer`, `smallint`, `bigint`
  - `real`, `double precision`
  - `numeric`, `decimal`
- Booléen
  - `boolean`

Le standard SQL propose des types standards pour stocker des chaînes de caractères (de taille fixe ou variable), des données numériques (entières, à virgule flottante) et des booléens.

2.4.4 TYPES STANDARDS (2)

- Temporel
  - `date`, `time`
  - `timestamp`
  - `interval`

- Chaînes de bit
  - `bit`, `bit varying`
- Formats validés
  - `XML`

Le standard SQL propose également des types standards pour stocker des éléments temporels (date, heure, la combinaison des deux avec ou sans fuseau horaire, intervalle).

D'utilisation plus rare, SQL permet également de stocker des chaînes de bit et des données validées au format XML.

2.4.5 CARACTÈRES

- `char(n)`
  - longueur fixe
  - de *n* caractères
  - complété à droite par des espaces si nécessaire
- `varchar(n)`
  - longueur variable
  - maximum *n* caractères
  - *n* optionnel

Le type `char(n)` permet de stocker des chaînes de caractères de taille fixe, donnée par l'argument `n`. Si la chaîne que l'on souhaite stocker est plus petite que la taille donnée à la déclaration de la colonne, elle sera complétée par des espaces à droite. Si la chaîne que l'on souhaite stocker est trop grande, une erreur sera levée.

Le type `varchar(n)` permet de stocker des chaînes de caractères de taille variable. La taille maximale de la chaîne est donnée par l'argument `n`. Toute chaîne qui excèdera cette taille ne sera pas prise en compte et génèrera une erreur. Les chaînes de taille inférieure à la taille limite seront stockées sans altérations.

La longueur de chaîne est mesurée en nombre de caractères sous PostgreSQL. Ce n'est pas forcément le cas dans d'autres SGBD.

2.4.6 REPRÉSENTATION DONNÉES CARACTÈRES

- Norme SQL
  - chaîne encadrée par `'`

    - `'chaîne de caractères'`
- Chaînes avec échappement du style C
    - chaîne précédée par `E` ou `e`
    - `E'chaîne de caractères'`
- Chaînes avec échappement Unicode
    - chaîne précédée par `U&`
    - `U&'chaîne de caractères'`

La norme SQL définit que les chaînes de caractères sont représentées encadrées de guillemets simples (caractère `'`). Le guillemet double (caractère `"`) ne peut être utilisé car il sert à protéger la casse des noms d'objets. PostgreSQL interprétera alors la chaîne comme un nom d'objet et générera une erreur.

Une représentation correcte d'une chaîne de caractères est donc de la forme suivante :

```
'chaîne de caractères'
```

Les caractères `'` doivent être doublés s'ils apparaissent dans la chaîne :

```
'J''ai acheté des croissants'
```

Une extension de la norme par PostgreSQL permet d'utiliser les méta-caractères des langages tels que le C, par exemple `\n` pour un retour de ligne, `\t` pour une tabulation, etc. :

```
E'chaîne avec un retour \nde ligne et une \ttabulation'
```

2.4.7 NUMÉRIQUES

- Entier
    - `smallint`, `integer`, `bigint`
    - signés
- Virgule flottante
    - `real`, `double precision`
    - valeurs inexactes
- Précision arbitraire
    - `numeric(precision, echelle)`, `decimal(precision, echelle)`
    - valeurs exactes

Le standard SQL propose des types spécifiques pour stocker des entiers signés. Le type `smallint` permet de stocker des valeurs codées sur 2 octets, soit des valeurs comprises entre -32768 et +32767. Le type `integer` ou `int`, codé sur 4 octets, permet de stocker des valeurs comprises entre -2147483648 et +2147483647. Enfin, le type `bigint`, codé

sur 8 octets, permet de stocker des valeurs comprises entre -9223372036854775808 et 9223372036854775807. Le standard SQL ne propose pas de stockage d'entiers non signés.

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

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

On peut aussi utiliser `numeric` sans aucune contrainte de taille, pour stocker de façon exacte n'importe quel nombre.

2.4.8 REPRÉSENTATION DE DONNÉES NUMÉRIQUES

- Chiffres décimaux : 0 à 9
- Séparateur décimal : `.`
- *chiffres*
- *chiffres.[chiffres][*`e`*[+-]chiffres]*
- *[chiffres].chiffres[*`e`*[+-]chiffres]*
- *chiffres*`e`*[+-]chiffres*
- Conversion
  - `TYPE 'chaine'`

Au moins un chiffre doit être placé avant ou après le point décimal, s'il est utilisé. Au moins un chiffre doit suivre l'indicateur d'exponentiel (caractère `e`), s'il est présent. Il peut ne pas y avoir d'espaces ou d'autres caractères imbriqués dans la constante. Notez que tout signe plus ou moins en avant n'est pas forcément considéré comme faisant part de la constante ; il est un opérateur appliqué à la constante.

Les exemples suivants montrent différentes représentations valides de constantes numériques :

```
42
3.5
4.
.001
5e2
1.925e-3
```

Une constante numérique contenant soit un point décimal soit un exposant est tout d'abord présumée du type `integer` si sa valeur est contenue dans le type integer (4 octets). Dans le cas contraire, il est présumé de type `bigint` si sa valeur entre dans un type `bigint` (8 octets). Dans le cas contraire, il est pris pour un type `numeric`. Les constantes contenant des points décimaux et/ou des exposants sont toujours présumées de type `numeric`.

Le type de données affecté initialement à une constante numérique est seulement un point de départ pour les algorithmes de résolution de types. Dans la plupart des cas, la constante sera automatiquement convertie dans le type le plus approprié suivant le contexte. Si nécessaire, vous pouvez forcer l'interprétation d'une valeur numérique sur un type de données spécifiques en la convertissant. Par exemple, vous pouvez forcer une valeur numérique à être traitée comme un type `real` (`float4`) en écrivant :

```
REAL '1.23'
```

2.4.9 BOOLÉENS

- `boolean`
- 3 valeurs possibles
  - `TRUE`
  - `FALSE`
  - `NULL` (ie valeur absente)

Le type `boolean` permet d'exprimer des valeurs booléennes, c'est-à-dire une valeur exprimant vrai ou faux. Comme tous les types de données en SQL, une colonne booléenne peut aussi ne pas avoir de valeur, auquel cas sa valeur sera `NULL`.

Un des intérêts des types booléens est de pouvoir écrire :

```
SELECT * FROM ma_table WHERE valide;
SELECT * FROM ma_table WHERE not consulte;
```

17.12

2.4.10 TEMPOREL

- Date
  - `date`
- Heure
  - `time`
  - avec ou sans fuseau horaire
- Date et heure
  - `timestamp`
  - avec ou sans fuseau horaire
- Intervalle de temps
  - `interval`

Le type `date` exprime une date. Ce type ne connaît pas la notion de fuseau horaire.

Le type `time` exprime une heure. Par défaut, il ne connaît pas la notion de fuseau horaire. En revanche, lorsque le type est déclaré comme `time with time zone`, il prend en compte un fuseau horaire. Mais cet emploi n'est pas recommandé. En effet, une heure convertie d'un fuseau horaire vers un autre pose de nombreux problèmes. En effet, le décalage horaire dépend également de la date : quand il est 6h00, heure d'été, à Paris, il est 21H00 sur la côte Pacifique aux États-Unis mais encore à la date de la veille.

Le type `timestamp` permet d'exprimer une date et une heure. Par défaut, il ne connaît pas la notion de fuseau horaire. Lorsque le type est déclaré `timestamp with time zone`, il est adapté aux conversions d'heure d'un fuseau horaire vers un autre car le changement de date sera répercuté dans la composante date du type de données. Il est précis à la microseconde.

Le format de saisie et de restitution des dates et heures dépend du paramètre `DateStyle`. La documentation de ce paramètre permet de connaître les différentes valeurs possibles. Il reste néanmoins recommandé d'utiliser les fonctions de formatage de date qui permettent de rendre l'application indépendante de la configuration du SGBD.

La norme ISO (ISO-8601) impose le format de date "année-mois-jour". La norme SQL est plus permissive et permet de restituer une date au format "jour/mois/ année" si DateStyle est égal à `'SQL, DMY'`.

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

SELECT current_timestamp;
              now
```

```
-------------------------------
 2017-08-29 16:11:58.290174+02

SET datestyle = 'SQL, DMY';

SELECT current_timestamp;
              now
--------------------------------
 29/08/2017 16:12:25.650716 CEST
```

2.4.11 REPRÉSENTATION DES DONNÉES TEMPORELLES

- Conversion explicite
  - `TYPE 'chaine'`
- Format d'un timestamp
  - `'YYYY-MM-DD HH24:MI:SS.ssssss'`
  - `'YYYY-MM-DD HH24:MI:SS.ssssss+fuseau'`
  - `'YYYY-MM-DD HH24:MI:SS.ssssss' AT TIME ZONE 'fuseau'`
- Format d'un intervalle
  - `INTERVAL 'durée interval'`

Expression d'une date, forcément sans gestion du fuseau horaire :

```
DATE '2017-08-29'
```

Expression d'une heure sans fuseau horaire :

```
TIME '10:20:10'
```

Ou, en spécifiant explicitement l'absence de fuseau horaire :

```
TIME WITHOUT TIME ZONE '10:20:10'
```

Expression d'une heure, avec fuseau horaire invariant. Cette forme est déconseillée :

```
TIME WITH TIME ZONE '10:20:10' AT TIME ZONE 'CEST'
```

Expression d'un timestamp sans fuseau horaire :

```
TIMESTAMP '2017-08-29 10:20:10'
```

Ou, en spécifiant explicitement l'absence de fuseau horaire :

```
TIMESTAMP WITHOUT TIME ZONE '2017-08-29 10:20:10'
```

Expression d'un timestamp avec fuseau horaire, avec microseconde :

```
TIMESTAMP WITH TIME ZONE '2017-08-29 10:20:10.123321'
AT TIME ZONE 'Europe/Paris'
```

17.12

Expression d'un intervalle d'une journée :

```
INTERVAL '1 day'
```

Il est possible de cumuler plusieurs expressions :

```
INTERVAL '1 year 1 day'
```

Les valeurs possibles sont :

- `YEAR` pour une année ;
- `MONTH` pour un mois ;
- `DAY` pour une journée ;
- `HOUR` pour une heure ;
- `MINUTE` pour une minute ;
- `SECOND` pour une seconde.

2.4.12 GESTION DES FUSEAUX HORAIRES

- Paramètre `timezone`
- Session : `SET TIME ZONE`
- Expression d'un fuseau horaire
  - nom complet : `'Europe/Paris'`
  - nom abbrégé : `'CEST'`
  - décalage : `'+02'`

Le paramètre `timezone` du `postgresql.conf` permet de positionner le fuseau horaire de l'instance PostgreSQL. Elle est initialisée par défaut en fonction de l'environnement du système d'exploitation.

Le fuseau horaire de l'instance peut également être défini au cours de la session à l'aide de la commande `SET TIME ZONE`.

La France utilise deux fuseaux horaires normalisés. Le premier, `CET`, correspond à *Central European Time* ou autrement dit à l'heure d'hiver en Europe centrale. Le second, `CEST`, correspond à *Central European Summer Time*, c'est-à-dire l'heure d'été en Europe centrale.

La liste des fuseaux horaires supportés est disponible dans la table système `pg_timezone_names` :

```
SELECT * FROM pg_timezone_names ;
               name               | abbrev | utc_offset | is_dst
----------------------------------+--------+------------+--------
 GB                               | BST    | 01:00:00   | t
 ROK                              | KST    | 09:00:00   | f
```

```
 Greenwich                              | GMT     | 00:00:00    | f
(...)
```

Il est possible de positionner le fuseau horaire au niveau de la session avec l'ordre `SET TIME ZONE` :

```
SET TIME ZONE "Europe/Paris";

SELECT now();
              now
-------------------------------
 2017-08-29 10:19:56.640162+02

SET TIME ZONE "Europe/Kiev";

SELECT now();
              now
-------------------------------
 2017-08-29 11:20:17.199983+03
```

Conversion implicite d'une donnée de type `timestamp` dans le fuseau horaire courant :

```
SET TIME ZONE "Europe/Kiev";

SELECT TIMESTAMP WITH TIME ZONE '2017-08-29 10:20:10 CEST';
      timestamptz
------------------------
 2017-08-29 11:20:10+03
```

Conversion explicite d'une donnée de type `timestamp` dans un autre fuseau horaire :

```
SELECT '2017-08-29 06:00:00' AT TIME ZONE 'US/Pacific';
      timezone
---------------------
 28/08/2017 21:00:00
```

2.4.13 CHAÎNES DE BITS

- Chaînes de bits
  - `bit(n)`, `bit varying(n)`

Les types `bit` et `bit varying` permettent de stocker des masques de bits. Le type `bit(n)` est à longueur fixe alors que le type `bit varying(n)` est à longueur variable mais avec un maximum de `n` bits.

17.12

2.4.14 REPRÉSENTATION DES CHAÎNES DE BITS

- Représentation binaire
  - Chaîne de caractères précédée de la lettre `B`
  - `B'01010101'`
- Représentation hexadécimale
  - Chaîne de caractères précédée de la lettre `X`
  - `X'55'`

2.4.15 XML

- Type validé
  - `xml`
- Chaîne de caractères
  - validation du document XML

Le type `xml` permet de stocker des documents XML. Par rapport à une chaîne de caractères simple, le type `xml` apporte la vérification de la structure du document XML ainsi que des fonctions de manipulations spécifiques (voir la documentation officielle[3]).

2.4.16 TYPES DÉRIVÉS

- Types spécifiques à PostgreSQL
- Sériés
  - principe de l'« autoincrement »
  - `serial`
  - `smallserial`
  - `bigserial`
  - équivalent à un type entier associé à une séquence et avec une valeur par défaut
  - (v 10+) préférer un type entier + la propriété `IDENTITY`
- Caractères
  - `text`

Les types `smallserial`, `serial` et `bigserial` permettent d'obtenir des fonctionnalités similaires aux types `autoincrement` rencontrés dans d'autres SGBD.

[3] http://docs.postgresql.fr/current/functions-xml.html

Néanmoins, ces types restent assez proches de la norme car ils définissent au final une colonne qui utilise un type et des objets standards. Selon le type dérivé utilisé, la colonne sera de type `smallint`, `integer` ou `bigint`. Une séquence sera également créée et la colonne prendra pour valeur par défaut la prochaine valeur de cette séquence.

Il est à noter que la notion d'identité apparaît en version 10 et qu'il est préférable de passer par cette contrainte que par ces types dérivés.

Attention : ces types n'interdisent pas l'insertion manuelle de doublons. Une contrainte de clé primaire explicite reste nécessaire pour les éviter.

Le type `text` est l'équivalent du type `varchar` mais sans limite de taille de la chaîne de caractère.

2.4.17 TYPES ADDITIONNELS NON SQL

- `bytea`
- `array`
- `enum`
- `cidr`, `inet`, `macaddr`
- `uuid`
- `json`
- `jsonb`
- `hstore`
- `range`

Les types standards ne sont pas toujours suffisants pour représenter certaines données. À l'instar d'autres SGBDR, PostgreSQL propose des types de données pour répondre à certains besoins.

On notera le type `bytea` qui permet de stocker des objets binaires dans une table. Le type `array` permet de stocker des tableaux et `enum` des énumérations.

Les types `json` et `hstore` permettent de stocker des documents non structurés dans la base de données. Le premier au format JSON, le second dans un format de type clé/valeur. Le type `hstore` est d'ailleurs particulièrement efficace car il dispose de méthodes d'indexation et de fonctions de manipulations performantes. Le type `json` a été completé par `jsonb` qui permet de stocker un document JSON binaire et optimisé, et d'accéder à une propriété sans désérialiser intégralement le document.

Le type `range` permet de stocker des intervalles de données. Ces données sont ensuite

manipulables par un jeu d'opérateurs dédiés et par le biais de méthodes d'indexation permettant d'accélérer les recherches.

2.4.18 TYPES UTILISATEURS

- Types utilisateurs
  - composites
  - énumérés (`enum`)
  - intervalles (`range`)
  - scalaires
  - tableau

  `CREATE TYPE`

PostgreSQL permet de créer ses propres types de données. Les usages les plus courants consistent à créer des types composites pour permettre à des fonctions de retourner des données sous forme tabulaire (retour de type `SETOF`).

L'utilisation du type énuméré (`enum`) nécessite aussi la création d'un type spécifique. Le type sera alors employé pour déclarer les objets utilisant une énumération.

Enfin, si l'on souhaite étendre les types intervalles (`range`) déjà disponibles, il est nécessaire de créer un type spécifique.

La création d'un type scalaire est bien plus marginale. Elle permet en effet d'étendre les types fournis par PostgreSQL mais nécessite d'avoir des connaissances fines des mécanismes de PostgreSQL. De plus, dans la majeure partie des cas, les types standards suffisent en général à résoudre les problèmes qui peuvent se poser à la conception.

Quant aux types tableaux, ils sont créés implicitement par PostgreSQL quand un utilisateur crée un type personnalisé.

Exemples

Utilisation d'un type `enum` :

```
CREATE TYPE arc_en_ciel AS ENUM (
    'red', 'orange', 'yellow', 'green', 'blue', 'purple'
);

CREATE TABLE test (id integer, couleur arc_en_ciel);

INSERT INTO test (id, couleur) VALUES (1, 'red');

INSERT INTO test (id, couleur) VALUES (2, 'pink');
```

```
ERROR:  invalid input value for enum arc_en_ciel: "pink"
LINE 1: INSERT INTO test (id, couleur) VALUES (2, 'pink');
```

Création d'un type interval `float8_range` :

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

2.5 CONCLUSION

- SQL : traitement d'ensembles d'enregistrements
- Pour les lectures : `SELECT`
- Nom des objets en minuscules
- Des types de données simples et d'autres plus complexes

Le standard SQL permet de traiter des ensembles d'enregistrements. Un enregistrement correspond à une ligne dans une relation. Il est possible de lire ces relations grâce à l'ordre `SELECT`.

2.5.1 BIBLIOGRAPHIE

- *Bases de données - de la modélisation au SQL* (Laurent Audibert)
- *SQL avancé : programmation et techniques avancées* (Joe Celko)
- *SQL : Au coeur des performances* (Markus Winand)
- *The Manga Guide to Databases* (Takahashi, Mana, Azuma, Shoko)
- *The Art of SQL* (Stéphane Faroult)

Bases de données - de la modélisation au SQL

- Auteur : Laurent Audibert,
- Éditeur : chez Ellipses,
- ISBN : 978-2729851200

Ce livre présente les notions essentielles pour modéliser une base de données et utiliser le langage SQL pour utiliser les bases de données créées. L'auteur appuie ses exercices sur PostgreSQL.

SQL avancé : programmation et techniques avancées

- Auteur : Joe Celko
- Editeur : Vuibert
- ISBN : 978-2711786503

17.12

Ce livre est écrit par une personne ayant participé à l'élaboration du standard SQL. Il a souhaité montré les bonnes pratiques pour utiliser le SQL pour résoudre un certain nombre de problèmes de tous les jours. Le livre s'appuie cependant sur la norme SQL-92, voire SQL-89. L'édition anglaise *SQL for Smarties* est bien plus à jour. Pour les anglophones, la lecture de l'ensemble des livres de Joe Celko est particulièrement recommandée.

SQL : Au coeur des performances

- Auteur : Markus Winand
- Éditeur : auto-édité
- ISBN : 978-3950307832
- site Internet[4]

Il s'agit du livre de référence sur les performances en SQL. Il dresse un inventaire des différents cas d'utilisation des index par la base de données, ce qui permettra de mieux prévoir l'indexation dès la conception. Ce livre s'adresse à un public avancé.

The Manga Guide to Databases

- Auteur : Takahashi, Mana, Azuma, Shoko
- Éditeur : No Starch Press
- ASIN : B00BUFN70E

The Art of SQL

- Auteur : Stéphane Faroult
- Éditeur : O'Reilly
- ISBN : 978-0-596-00894-9
- ISBN : 978-0-596-15971-9 (e-book)

Ce livre s'adresse également à un public avancé. Il présence également les bonnes pratiques lorsque l'on utilise une base de données.

2.5.2 QUESTIONS

N'hésitez pas, c'est le moment !

2.6 TRAVAUX PRATIQUES

---

[4] http://use-the-index-luke.com/fr

2.6.1 ÉNONCÉS

Initialisez la base `tpc` et connectez-vous à la base `tpc` à l'aide de pgAdminIII.

Le schéma suivant montre les différentes tables de la base de TP :

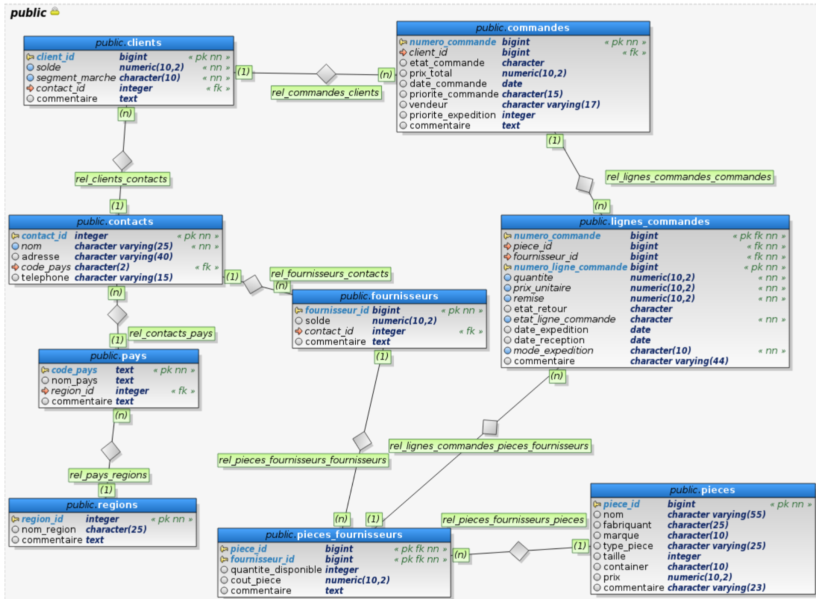


FIGURE 1: SCHÉMA BASE TPC

1. Afficher l'heure courante, au méridien de Greenwich.
2. Afficher la date et l'heure qu'il sera dans 1 mois et 1 jour.
3. Ajouter 1 au nombre de type réel '1.42'. Pourquoi ce résultat ? Quel type de données permet d'obtenir un résultat correct ?
4. Afficher le contenu de la table `pays` en classant les pays dans l'ordre alphabétique.
5. Afficher les pays contenant la lettre `a`, majuscule ou minuscule. Plusieurs solutions sont possibles.
6. Afficher le nombre lignes de commandes dont la quantité commandée est comprise entre 5 et 10.
7. Pour chaque pays, afficher son nom et la région du monde dont il fait partie.

```
nom_pays                |         nom_region
-------------------------------+---------------------------
 ALGÉRIE                       | Afrique
(...)
```

8. Afficher le nombre total de clients français et allemands.

Sortie attendue :

```
count
-------
 12418
```

9. Afficher le numéro de commande et le nom du client ayant passé la commande. Seul un sous-ensemble des résultats sera affiché : les 20 premières lignes du résultat seront exclues et seules les 20 suivantes seront affichées. Il faut penser à ce que le résultat de cette requête soit stable entre plusieurs exécutions.

Sortie attendue :

```
numero_commande |  nom_client
-----------------+--------------
              67 | Client112078
              68 | Client33842
(...)
```

10. Afficher les noms et codes des pays qui font partie de la région « Europe ».

Sortie attendue :

```
nom_pays        | code_pays
-----------------------+-----------
 ALLEMAGNE             | DE
(...)
```

11. Pour chaque pays, afficher une chaîne de caractères composée de son nom, suivi entre parenthèses de son code puis, séparé par une virgule, du nom de la région dont il fait partie.

Sortie attendue :

```
detail_pays
--------------------------------------------------
 ALGÉRIE (DZ), Afrique
(...)
```

12. Pour les clients ayant passé des commandes durant le mois de janvier 2011, affichez les identifiants des clients, leur nom, leur numéro de téléphone et le nom de leur pays.

Sortie attendue :

```
client_id |      nom       |     telephone     |            nom_pays
-----------+--------------+-----------------+-------------------------------
     83279 | Client83279   | 12-835-574-2048 | JAPON
```

13. Pour les dix premières commandes de l'année 2011, afficher le numéro de la commande, la date de la commande ainsi que son âge.

Sortie attendue :

```
numero_commande | date_commande |             age
-----------------+---------------+--------------------------
           11364 | 2011-01-01    | 1392 days 15:25:19.012521
(...)
```

2.6.2 SOLUTIONS

1. Afficher l'heure courante, au méridien de Greenwich :

```
SELECT now() AT TIME ZONE 'GMT';
```

2. Afficher la date et l'heure qu'il sera dans 1 mois et 1 jour

```
SELECT now() + INTERVAL '1 month 1 day';
```

3. Ajouter 1 au nombre de type réel '1.42'. Pourquoi ce résultat ?

```
SELECT REAL '1.42' + 1 AS resultat;
     resultat
------------------
 2.41999995708466
(1 row)
```

Le type de données `real` est un type numérique à virgule flottante, codé sur 4 octets. Il n'offre pas une précision suffisante pour les calculs précis. Son seul avantage est la vitesse de calcul. Pour effectuer des calculs précis, il vaut mieux privilégier le type de données `numeric`.

4. Afficher le contenu de la table `pays` en classant les pays dans l'ordre alphabétique.

```
SELECT * FROM pays ORDER BY nom_pays;
```

5. Afficher les pays contenant la lettre `a`, majuscule ou minuscule :

```
SELECT * FROM pays WHERE lower(nom_pays) LIKE '%a%';
```

17.12

```sql
SELECT * FROM pays WHERE nom_pays ILIKE '%a%';
```

```sql
SELECT * FROM pays WHERE nom_pays LIKE '%a%' OR nom_pays LIKE '%A%';
```

En terme de performances, la seconde variante sera plus rapide sur un volume de données important si l'on dispose du bon index. La taille de la table `pays` ne permet pas d'observer de différence significative sur cette requête.

6. Afficher le nombre lignes de commandes dont la quantité commandée est comprise entre 5 et 10 :

```sql
SELECT count(*)
  FROM lignes_commandes
 WHERE quantite BETWEEN 5 AND 10;
```

Autre écriture possible :

```sql
SELECT count(*)
  FROM lignes_commandes
 WHERE quantite >= 5
   AND quantite <= 10;
```

7. Pour chaque pays, afficher son nom et la région du monde dont il fait partie :

```sql
SELECT nom_pays, nom_region
  FROM pays p, regions r
 WHERE p.region_id = r.region_id;
```

8. Afficher le nombre total de clients français et allemands :

```sql
SELECT count(*)
  FROM clients cl, contacts cn, pays p
 WHERE cl.contact_id = cn.contact_id
   AND cn.code_pays = p.code_pays
   AND p.nom_pays IN ('FRANCE', 'ALLEMAGNE');
```

À noter que cette syntaxe est obsolète, il faut utiliser la clause `JOIN`, plus lisible et plus complète, qui sera vue plus loin :

```sql
SELECT count(*)
  FROM clients cl
  JOIN contacts cn ON (cl.contact_id = cn.contact_id)
  JOIN pays p ON (cn.code_pays = p.code_pays)
 WHERE p.nom_pays IN ('FRANCE', 'ALLEMAGNE');
```

En connaissant les codes de ces pays, il est possible d'éviter la lecture de la table `pays` :

```sql
SELECT count(*)
  FROM clients cl, contacts cn
 WHERE cl.contact_id = cn.contact_id
   AND cn.code_pays IN ('FR', 'DE');
```

L'équivalent avec la syntaxe JOIN serait :

```
SELECT count(*)
  FROM clients cl
  JOIN contacts cn ON (cl.contact_id = cn.contact_id)
 WHERE cn.code_pays IN ('FR', 'DE');
```

9. Afficher le numéro de commande et le nom du client ayant passé la commande. Seul un sous-ensemble des résultats sera affiché : les 20 premières lignes du résultat seront exclues et seules les 20 suivantes seront affichées. Il faut penser à ce que le résultat de cette requête soit stable entre plusieurs exécutions.

La syntaxe normalisée SQL impose d'écrire la requête de la façon suivante. La stabilité du résultat de la requête est garantie par un tri explicite, s'il n'est pas précisé, la base de données va retourner les lignes dans l'ordre physique qui est susceptible de changer entre deux exécutions :

```
SELECT numero_commande, nom AS nom_client
  FROM commandes cm, clients cl, contacts cn
 WHERE cm.client_id = cl.client_id
   AND cl.contact_id = cn.contact_id
 ORDER BY numero_commande
 FETCH FIRST 20 ROWS ONLY
 OFFSET 20;
```

Mais PostgreSQL supporte également la clause LIMIT :

```
SELECT numero_commande, nom AS nom_client
  FROM commandes cm, clients cl, contacts cn
 WHERE cm.client_id = cl.client_id
   AND cl.contact_id = cn.contact_id
 ORDER BY numero_commande
 LIMIT 20
 OFFSET 20;
```

Et l'équivalent avec la syntaxe JOIN serait :

```
SELECT numero_commande, nom AS nom_client
  FROM commandes cm
  JOIN clients cl ON (cm.client_id = cl.client_id)
  JOIN contacts cn ON (cl.contact_id = cn.contact_id)
 ORDER BY numero_commande
 LIMIT 20
 OFFSET 20;
```

10. Afficher les noms et codes des pays qui font partie de la région « Europe ».

```
SELECT nom_pays, code_pays
  FROM regions r, pays p
```

```sql
 WHERE r.region_id = p.region_id
   AND r.nom_region = 'Europe';
```

Et l'équivalent avec la syntaxe `JOIN` serait :

```sql
SELECT nom_pays, code_pays
  FROM regions r
  JOIN pays p ON (r.region_id = p.region_id)
 WHERE r.nom_region = 'Europe';
```

11. Pour chaque pays, afficher une chaîne de caractères composée de son nom, suivi entre parenthèses de son code puis, séparé par une virgule, du nom de la région dont il fait partie.

```sql
SELECT nom_pays || ' (' || code_pays || '), ' || nom_region
  FROM regions r, pays p
 WHERE r.region_id = p.region_id;
```

Et l'équivalent avec la syntaxe `JOIN` serait :

```sql
SELECT nom_pays || ' (' || code_pays || '), ' || nom_region
  FROM regions r
  JOIN pays p ON (r.region_id = p.region_id);
```

12. Pour les clients ayant passé des commandes durant le mois de janvier 2011, affichez les identifiants des clients, leur nom, leur numéro de téléphone et le nom de leur pays.

```sql
SELECT cl.client_id, nom, telephone, nom_pays
  FROM clients cl, commandes cm, contacts cn, pays p
 WHERE cl.client_id = cm.client_id
   AND cl.contact_id = cn.contact_id
   AND cn.code_pays = p.code_pays
   AND date_commande BETWEEN '2011-01-01' AND '2011-01-31';
```

Le troisième module de la formation abordera les jointures et leurs syntaxes. À l'issue de ce prochain module, la requête de cet exercice pourrait être écrite de la façon suivante :

```sql
SELECT cl.client_id, nom, telephone, nom_pays
  FROM clients cl
  JOIN commandes cm
       USING (client_id)
  JOIN contacts co
       USING (contact_id)
  JOIN pays p
       USING (code_pays)
 WHERE date_commande BETWEEN '2011-01-01' AND '2011-01-31';
```

13. Pour les dix premières commandes de l'année 2011, afficher le numéro de la commande, la date de la commande ainsi que son âge.

```
SELECT numero_commande, date_commande, now() - date_commande AS age
  FROM commandes
 WHERE date_commande BETWEEN '2011-01-01' AND '2011-12-31'
 ORDER BY date_commande
 LIMIT 10;
```

3 CRÉATION D'OBJET ET MISES À JOUR

3.1 INTRODUCTION

- DDL, gérer les objets
- DML, écrire des données
- Gérer les transactions

Le module précédent nous a permis de voir comment lire des données à partir de requêtes SQL. Ce module a pour but de présenter la création et la gestion des objets dans la base de données (par exemple les tables), ainsi que l'ajout, la suppression et la modification de données.

Une dernière partie sera consacrée aux transactions.

3.1.1 MENU

- DDL (`Data Definition Language`)
- DML (`Data Manipulation Language`)
- TCL (`Transaction Control Language`)

3.1.2 OBJECTIFS

- Savoir créer, modifier et supprimer des objets
- Savoir utiliser les contraintes d'intégrité
- Savoir mettre à jour les données
- Savoir utiliser les transactions

3.2 DDL

- DDL
  - `Data Definition Language`
  - langage de définition de données
- Permet de créer des objets dans la base de données

Les ordres DDL (acronyme de `Data Definition Language`) permettent de créer des objets dans la base de données et notamment la structure de base du standard SQL : les tables.

3.2.1 OBJETS D'UNE BASE DE DONNÉES

- Objets définis par la norme SQL :
  - schémas
  - séquences
  - tables
  - contraintes
  - domaines
  - vues
  - fonctions
  - triggers

La norme SQL définit un certain nombre d'objets standards qu'il est possible de créer en utilisant les ordres DDL. D'autres types d'objets existent bien entendu, comme les domaines. Les ordres DDL permettent également de créer des index, bien qu'ils ne soient pas définis dans la norme SQL.

La seule structure de données possible dans une base de données relationnelle est la table.

3.2.2 CRÉER DES OBJETS

- Ordre `CREATE`
- Syntaxe spécifique au type d'objet
- Exemple :

```sql
CREATE SCHEMA s1;
```

La création d'objet passe généralement par l'ordre `CREATE`. La syntaxe dépend fortement du type d'objet. Voici trois exemples :

```sql
CREATE SCHEMA s1;
CREATE TABLE t1 (c1 integer, c2 text);
CREATE SEQUENCE s1 INCREMENT BY 5 START 10;
```

17.12

Pour créer un objet, il faut être propriétaire du schéma ou de la base auquel appartiendra l'objet ou avoir le droit `CREATE` sur le schéma ou la base.

3.2.3 MODIFIER DES OBJETS

- Ordre `ALTER`
- Syntaxe spécifique pour modifier la définition d'un objet, exemple:
- renommage

```
ALTER type_objet ancien_nom RENAME TO nouveau_nom ;
```

- changement de propriétaire

```
ALTER type_objet nom_objet OWNER TO proprietaire ;
```

- changement de schéma

```
ALTER type_objet nom_objet SET SCHEMA nom_schema ;
```

Modifier un objet veut dire modifier ses propriétés. On utilise dans ce cas l'ordre `ALTER`. Il faut être propriétaire de l'objet pour pouvoir le faire.

Deux propriétés sont communes à tous les objets : le nom de l'objet et son propriétaire. Deux autres sont fréquentes et dépendent du type de l'objet : le schéma et le tablespace. Les autres propriétés dépendent directement du type de l'objet.

3.2.4 SUPPRIMER DES OBJETS

- Ordre `DROP`
- Exemples :
  - supprimer un objet :

    ```
    DROP type_objet nom_objet ;
    ```

- supprimer un objet et ses dépendances :

```
DROP type_objet nom_objet CASCADE ;
```

Seul un propriétaire peut supprimer un objet. Il utilise pour cela l'ordre `DROP`. Pour les objets ayant des dépendances, l'option `CASCADE` permet de tout supprimer d'un coup. C'est très pratique, et c'est en même temps très dangereux : il faut donc utiliser cette option à bon escient.

3.2.5 SCHÉMA

- Identique à un espace de nommage
- Permet d'organiser les tables de façon logique
- Possibilité d'avoir des objets de même nom dans des schémas différents
- Pas d'imbrication (contrairement à des répertoires par exemple)
- Schéma `public`
  - créé par défaut dans une base de données PostgreSQL

La notion de schéma dans PostgreSQL est à rapprocher de la notion d'espace de nommage (ou *namespace*) de certains langages de programmation. Le catalogue système qui contient la définition des schémas dans PostgreSQL s'appelle d'ailleurs `pg_namespace`.

Les schémas sont utilisés pour répartir les objets de façon logique, suivant un schéma interne à l'entreprise. Ils servent aussi à faciliter la gestion des droits (il suffit de révoquer le droit d'utilisation d'un schéma à un utilisateur pour que les objets contenus dans ce schéma ne soient plus accessibles à cet utilisateur).

Un schéma `public` est créé par défaut dans toute nouvelle base de données. Tout le monde a le droit d'y créer des objets. Il est cependant possible de révoquer ce droit ou supprimer ce schéma.

3.2.6 GESTION D'UN SCHÉMA

- `CREATE SCHEMA nom_schéma`
- `ALTER SCHEMA nom_schéma`
  - renommage
  - changement de propriétaire
- `DROP SCHEMA [ IF EXISTS ] nom_schéma [ CASCADE ]`

L'ordre `CREATE SCHEMA` permet de créer un schéma. Il suffit de lui spécifier le nom du schéma. `CREATE SCHEMA` offre d'autres possibilités qui sont rarement utilisées.

L'ordre `ALTER SCHEMA nom_schema RENAME TO nouveau_nom_schema` permet de renommer un schéma. L'ordre `ALTER SCHEMA nom_schema OWNER TO proprietaire` permet de donner un nouveau propriétaire au schéma.

Enfin, l'ordre `DROP SCHEMA` permet de supprimer un schéma. La clause `IF EXISTS` permet d'éviter la levée d'une erreur si le schéma n'existe pas (très utile dans les scripts SQL). La

clause CASCADE permet de supprimer le schéma ainsi que tous les objets qui sont positionnés dans le schéma.

Exemples

Création d'un schéma reference :

```
CREATE SCHEMA reference;
```

Une table peut être créée dans ce schéma :

```
CREATE TABLE reference.communes (
  commune      text,
  codepostal   char(5),
  departement  text,
  codeinsee    integer
);
```

La suppression directe du schéma ne fonctionne pas car il porte encore la table communes :

```
DROP SCHEMA reference;
ERROR:  cannot drop schema reference because other objects depend on it
DETAIL:  table reference.communes depends on schema reference
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

L'option CASCADE permet de supprimer le schéma et ses objets dépendants :

```
DROP SCHEMA reference CASCADE;
NOTICE:  drop cascades to table reference.communes
```

3.2.7 ACCÈS AUX OBJETS

- Nommage explicite
  - nom_schema.nom_objet
- Chemin de recherche de schéma
  - paramètre search_path
  - SET search_path = schema1,schema2,public;
  - par défaut : $user, public

Le paramètre search_path permet de définir un chemin de recherche pour pouvoir retrouver les tables dont le nom n'est pas qualifié par le nom de son schéma. PostgreSQL procèdera de la même façon que le système avec la variable $PATH : il recherche la table dans le premier schéma listé. S'il trouve une table portant ce nom dans le schéma, il préfixe le nom de table avec celui du schéma. S'il ne trouve pas de table de ce nom dans le schéma, il effectue la même opération sur le prochain schéma de la liste

du search_path. S'il n'a trouvé aucune table de ce nom dans les schémas listés par search_path, PostgreSQL lève une erreur.

Comme beaucoup d'autres paramètres, le search_path peut être positionné à différents endroits. Par défaut, il est assigné à $user, public, c'est-à-dire que le premier schéma de recherche portera le nom de l'utilisateur courant, et le second schéma de recherche est public.

On peut vérifier la variable search_path à l'aide de la commande SHOW :

```
SHOW search_path;
  search_path
----------------
 "$user",public
(1 row)
```

Pour obtenir une configuration particulière, la variable search_path peut être positionnée dans le fichier postgresql.conf :

```
search_path = '"$user",public'
```

Cette variable peut aussi être positionnée au niveau d'un utilisateur. Chaque fois que l'utilisateur se connectera, il prendra le search_path de sa configuration spécifique :

```
ALTER ROLE nom_role SET search_path = "$user", public;
```

Cela peut aussi se faire au niveau d'une base de données. Chaque fois qu'un utilisateur se connectera à la base, il prendra le search_path de cette base, sauf si l'utilisateur a déjà une configuration spécifique :

```
ALTER DATABASE nom_base SET search_path = "$user", public;
```

La variable search_path peut également être positionnée pour un utilisateur particulier, dans une base particulière :

```
ALTER ROLE nom_role IN DATABASE nom_base SET search_path = "$user", public;
```

Enfin, la variable search_path peut être modifiée dynamiquement dans la session avec la commande SET :

```
SET search_path = "$user", public;
```

Avant la version 9.3, les requêtes préparées et les fonctions conservaient en mémoire le plan d'exécution des requêtes. Ce plan ne faisait plus référence aux noms des objets mais à leurs identifiants. Du coup, un search_path changeant entre deux exécutions d'une requête préparée ou d'une fonction ne permettait pas de cibler une table différente. Voici un exemple le montrant :

```
-- création des objets
CREATE SCHEMA s1;
```

```
CREATE SCHEMA s2;
CREATE TABLE s1.t1 (c1 text);
CREATE TABLE s2.t1 (c1 text);
INSERT INTO s1.t1 VALUES('schéma s1');
INSERT INTO s2.t1 VALUES('schéma s2');

SELECT * FROM s1.t1;
    c1
-----------
 schéma s1
(1 row)

SELECT * FROM s2.t1;
    c1
-----------
 schéma s2
(1 row)

-- il y a bien des données différentes dans chaque table

SET search_path TO s1;
PREPARE req AS SELECT * FROM t1;

EXECUTE req;
    c1
-----------
 schéma s1
(1 row)

SET search_path TO s2;
EXECUTE req;
    c1
-----------
 schéma s1
(1 row)

-- malgré le changement de search_path, nous en sommes toujours
-- aux données de l'autre table

b1=# SELECT * FROM t1;
    c1
-----------
 schéma s2
(1 row)
```

Dans ce cas, il est préférable de configurer le paramètre `search_path` directement au

niveau de la fonction.

À partir de la version 9.3, dès que le `search_path` change, les plans en cache sont supprimés (dans le cas de la fonction) ou recréés (dans le cas des requêtes préparées).

3.2.8 SÉQUENCES

- Séquence
  - génère une séquence de nombres
- Paramètres
  - valeur minimale `MINVALUE`
  - valeur maximale `MAXVALUE`
  - valeur de départ `START`
  - incrément `INCREMENT`
  - cache `CACHE`
  - cycle autorisé `CYCLE`

Les séquences sont des objets standards qui permettent de générer des séquences de valeur. Elles sont utilisées notamment pour générer un numéro unique pour un identifiant ou, plus rarement, pour disposer d'un compteur informatif, mis à jour au besoin.

Le cache de la séquence a pour effet de générer un certain nombre de valeurs en mémoire afin de les mettre à disposition de la session qui a utilisé la séquence. Même si les valeurs pré-calculées ne sont pas consommées dans la session, elles seront consommées au niveau de la séquence. Cela peut avoir pour effet de créer des trous dans les séquences d'identifiants et de consommer très rapidement les numéros de séquence possibles. Le cache de séquence n'a pas besoin d'être ajusté sur des applications réalisant de petites transactions. Il permet en revanche d'améliorer les performances sur des applications qui utilisent massivement des numéros de séquences, notamment pour réaliser des insertions massives.

3.2.9 CRÉATION D'UNE SÉQUENCE

```
CREATE SEQUENCE nom [ INCREMENT incrément ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ CACHE cache ]
```

```
    [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
```

La syntaxe complète est donnée dans le slide.

Le mot clé `TEMPORARY` ou `TEMP` permet de définir si la séquence est temporaire. Si tel est le cas, elle sera détruite à la déconnexion de l'utilisateur.

Le mot clé `INCREMENT` définit l'incrément de la séquence, `MINVALUE`, la valeur minimale de la séquence et `MAXVALUE`, la valeur maximale. `START` détermine la valeur de départ initiale de la séquence, c'est-à-dire juste après sa création. La clause `CACHE` détermine le cache de séquence. `CYCLE` permet d'indiquer au SGBD que la séquence peut reprendre son compte à `MINVALUE` lorsqu'elle aura atteint `MAXVALUE`. La clause `NO CYCLE` indique que le rebouclage de la séquence est interdit, PostgreSQL lèvera alors une erreur lorsque la séquence aura atteint son `MAXVALUE`. Enfin, la clause `OWNED BY` détermine l'appartenance d'une séquence à une colonne d'une table. Ainsi, si la colonne est supprimée, la séquence sera implicitement supprimée.

Exemple de séquence avec rebouclage :

```
CREATE SEQUENCE testseq INCREMENT BY 1 MINVALUE 3 MAXVALUE 5 CYCLE START WITH 4;

SELECT nextval('testseq');
 nextval
---------
       4

SELECT nextval('testseq');
 nextval
---------
       5

SELECT nextval('testseq');
 nextval
---------
       3
```

3.2.10 MODIFICATION D'UNE SÉQUENCE

```
ALTER SEQUENCE nom [ INCREMENT increment ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ RESTART [ [ WITH ] nouveau_début ] ]
```

```
[ CACHE cache ] [ [ NO ] CYCLE ]
[ OWNED BY { nom_table.nom_colonne | NONE } ]
```

- Il est aussi possible de modifier
  - le propriétaire
  - le schéma

Les propriétés de la séquence peuvent être modifiés avec l'ordre `ALTER SEQUENCE`.

La séquence peut être affectée à un nouveau propriétaire :

```
ALTER SEQUENCE [ IF EXISTS ] nom OWNER TO nouveau_propriétaire
```

Elle peut être renommée :

```
ALTER SEQUENCE [ IF EXISTS ] nom RENAME TO nouveau_nom
```

Enfin, elle peut être positionnée dans un nouveau schéma :

```
ALTER SEQUENCE [ IF EXISTS ] nom SET SCHEMA nouveau_schema
```

3.2.11 SUPPRESSION D'UNE SÉQUENCE

```
DROP SEQUENCE nom [, ...]
```

Voici la syntaxe complète de `DROP SEQUENCE` :

```
DROP SEQUENCE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Le mot clé `CASCADE` permet de supprimer la séquence ainsi que tous les objets dépendants (par exemple la valeur par défaut d'une colonne).

3.2.12 SÉQUENCES, UTILISATION

- Obtenir la valeur suivante
  - `nextval('nom_sequence')`
- Obtenir la valeur courante
  - `currval('nom_sequence')`
  - mais `nextval()` doit être appelé avant dans la même session

La fonction `nextval()` permet d'obtenir le numéro de séquence suivant. Son comportement n'est pas transactionnel. Une fois qu'un numéro est consommé, il n'est pas possible de revenir dessus, malgré un `ROLLBACK` de la transaction. La séquence est le seul objet à avoir un comportement de ce type.

La fonction `currval()` permet d'obtenir le numéro de séquence courant, mais son usage nécessite d'avoir utilisé `nextval()` dans la session.

Il est possible d'interroger une séquence avec une requête `SELECT`. Cela permet d'obtenir des informations sur la séquence, dont la dernière valeur utilisée dans la colonne `last_value`. Cet usage n'est pas recommandé en production et doit plutôt être utilisé à titre informatif.

Exemples

Utilisation d'une séquence simple :

```
CREATE SEQUENCE testseq
INCREMENT BY 1 MINVALUE 10 MAXVALUE 20 START WITH 15 CACHE 1;


SELECT currval('testseq');
ERROR:  currval of sequence "testseq" is not yet defined in this session


SELECT * FROM testseq ;
- [ RECORD 1 ]-+--------
sequence_name  | testseq
last_value     | 15
start_value    | 15
increment_by   | 1
max_value      | 20
min_value      | 10
cache_value    | 5
log_cnt        | 0
is_cycled      | f
is_called      | f


SELECT nextval('testseq');
 nextval
---------
      15
(1 row)


SELECT currval('testseq');
 currval
---------
      15

SELECT nextval('testseq');
```

```
 nextval
---------
      16
(1 row)

ALTER SEQUENCE testseq RESTART WITH 5;
ERROR:  RESTART value (5) cannot be less than MINVALUE (10)

DROP SEQUENCE testseq;
```

Utilisation d'une séquence simple avec cache :

```
CREATE SEQUENCE testseq INCREMENT BY 1 CACHE 10;

SELECT nextval('testseq');
 nextval
---------
       1
```

Déconnexion et reconnexion de l'utilisateur :

```
SELECT nextval('testseq');
 nextval
---------
      11
```

Suppression en cascade d'une séquence :

```
CREATE TABLE t2 (id serial);

\d t2
                            Table "s2.t2"
 Column |  Type   |                      Modifiers
--------+---------+-----------------------------------------------
 id     | integer | not null default nextval('t2_id_seq'::regclass)

DROP SEQUENCE t2_id_seq;
ERROR:  cannot drop sequence t2_id_seq because other objects depend on it
DETAIL:  default for table t2 column id depends on sequence t2_id_seq
HINT:  Use DROP ... CASCADE to drop the dependent objects too.

DROP SEQUENCE t2_id_seq CASCADE;
NOTICE:  drop cascades to default for table t2 column id

\d t2
        Table "s2.t2"
 Column |  Type   | Modifiers
--------+---------+-----------
```

```
id      | integer | not null
```

3.2.13 TYPE SERIAL

- Type `serial`/`bigserial`/`smallserial`
  - séquence générée automatiquement
  - valeur par défaut `nextval(...)`
- (v 10+) Préférer un entier avec `IDENTITY`

Certaines bases de données offre des colonnes auto-incrémentées (`autoincrement` de MySQL ou `identity` de SQL Server).

PostgreSQL ne possède `identity` qu'à partir de la v 10. Jusqu'en 9.6 on pourra utiliser `serial` un équivalent qui s'appuie sur les séquences et la possibilité d'appliquer une valeur par défaut à une colonne.

Par exemple, si l'on crée la table suivante :

```
CREATE TABLE exemple_serial (
  id SERIAL PRIMARY KEY,
  valeur INTEGER NOT NULL
);
```

On s'aperçoit que table a été créée telle que demandé, mais qu'une séquence a aussi été également créée. Elle porte un nom dérivé de la table associé à la colonne correspondant au type `serial`, terminé par `seq` :

```
postgres=# \d
                 List of relations
 Schema |          Name          |   Type   | Owner
--------+------------------------+----------+--------
 public | exemple_serial         | table    | thomas
 public | exemple_serial_id_seq  | sequence | thomas
```

En examinant plus précisément la définition de la table, on s'aperçoit que la colonne `id` porte une valeur par défaut qui correspond à l'appel de la fonction `nextval()` sur la séquence qui a été créée implicitement :

```
postgres=# \d exemple_serial
                        Table "public.exemple_serial"
 Column |  Type   |                          Modifiers
--------+---------+------------------------------------------------------------
 id     | integer | not null default nextval('exemple_serial_id_seq'::regclass)
 valeur | integer | not null
```

```
Indexes:
    "exemple_serial_pkey" PRIMARY KEY, btree (id)
```

smallserial et bigserial sont des variantes de serial s'appuyant sur des types d'entiers plus courts ou plus longs.

3.2.14 DOMAINES

- Permet d'associer
  - un type standard
  - et une contrainte (optionnelle)

Un domaine est utilisé pour définir un type utilisateur qui est en fait un

Un domaine est utilisé pour définir un type utilisateur qui est en fait un type utilisateur standard accompagné de la définition de contraintes particulières.

Les domaines sont utiles pour ramener la définition de contraintes communes à plusieurs colonnes sur un seul objet. La maintenance en est ainsi facilitée.

L'ordre CREATE DOMAIN permet de créer un domaine, ALTER DOMAIN permet de modifier sa définition, et enfin, DROP DOMAIN permet de supprimer un domaine.

Exemples

Gestion d'un domaine salaire :

```
-- ajoutons le domaine et la table
CREATE DOMAIN salaire AS integer CHECK (VALUE > 0);
CREATE TABLE employes (id serial, nom text, paye salaire);

\d employes
                        Table « public.employes »
 Colonne |  Type   | NULL-able |              Par défaut
---------+---------+-----------+--------------------------------------
 id      | integer | not null  | nextval('employes_id_seq'::regclass)
 nom     | text    |           |
 paye    | salaire |           |


-- insérons des données dans la nouvelle table
INSERT INTO employes (nom, paye) VALUES ('Albert', 1500);
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 0);
ERROR:  value for domain salaire violates check constraint "salaire_check"
-- erreur logique vu qu'on ne peut avoir qu'un entier strictement positif
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 1000);
```

```
INSERT 0 1
INSERT INTO employes (nom, paye) VALUES ('Bertrand', NULL);
INSERT 0 1
-- tous les employés doivent avoir un salaire
-- il faut donc modifier la contrainte, pour s'assurer
-- qu'aucune valeur NULL ne soit saisi
ALTER DOMAIN salaire SET NOT NULL;
ERROR:  column "paye" of table "employes" contains null values
-- la ligne est déjà présente, il faut la modifier
UPDATE employes SET paye=1500 WHERE nom='Bertrand';
-- maintenant, on peut ajouter la contrainte au domaine
ALTER DOMAIN salaire SET NOT NULL;
INSERT INTO employes (nom, paye) VALUES ('Delphine', NULL);
ERROR:  domain salaire does not allow null values
-- la contrainte est bien vérifiée
-- supprimons maintenant la contrainte
DROP DOMAIN salaire;
ERROR:  cannot drop type salaire because other objects depend on it
DETAIL:  table employes column paye depends on type salaire
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
-- il n'est pas possible de supprimer le domaine car il est référencé dans une
-- table.  Il faut donc utiliser l'option CASCADE
b1=# DROP DOMAIN salaire CASCADE;
NOTICE:  drop cascades to table employes column paye
DROP DOMAIN
-- le domaine a été supprimée ainsi que toutes les colonnes ayant ce type
\d employes
                              Table « public.employes »
 Colonne |  Type   | NULL-able |               Par défaut
---------+---------+-----------+---------------------------------------
 id      | integer | not null  | nextval('employes_id_seq'::regclass)
 nom     | text    |           |
```

Création et utilisation d'un domaine `code_postal_us` :

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE courrier_us (
  id_adresse SERIAL PRIMARY KEY,
  rue1 TEXT NOT NULL,
  rue2 TEXT,
  rue3 TEXT,
  ville TEXT NOT NULL,
```

```
  code_postal code_postal_us NOT NULL
);

INSERT INTO courrier_us (rue1,ville,code_postal)
VALUES ('51 Franklin Street', 'Boston, MA', '02110-1335' );

INSERT 0 1

INSERT INTO courrier_us (rue1,ville,code_postal)
VALUES ('10 rue d''Uzès','Paris','F-75002') ;

ERREUR:  la valeur pour le domaine code_postal_us viole la contrainte de
vérification « code_postal_us_check »
```

3.2.15 TABLES

- Équivalent ensembliste d'une relation
- Composé principalement de
  - colonnes ordonnées
  - contraintes

La table est l'élément de base d'une base de données. Elle est composée de colonnes (à sa création) et est remplie avec des enregistrements (lignes de la table). Sa définition peut aussi faire intervenir des contraintes, qui sont au niveau table ou colonne.

3.2.16 CRÉATION D'UNE TABLE

- Définition de son nom
- Définition de ses colonnes
  - nom, type, contraintes éventuelles
- Clauses de stockage
- `CREATE TABLE`

Pour créer une table, il faut donner son nom et la liste des colonnes. Une colonne est définie par son nom et son type, mais aussi des contraintes optionnelles.

Des options sont possibles pour les tables, comme les clauses de stockage. Dans ce cas, on sort du contexte logique pour se placer au niveau physique.

3.2.17 CREATE TABLE

```
CREATE TABLE nom_table (
  definition_colonnes
  definition_contraintes
) clause_stockage;
```

La création d'une table passe par l'ordre `CREATE TABLE`. La définition des colonnes et des contraintes sont entre parenthèse après le nom de la table.

3.2.18 DÉFINITION DES COLONNES

```
nom_colonne type [ COLLATE collation ] [ contrainte ]
[, ...]
```

Les colonnes sont indiquées l'une après l'autre, en les séparant par des virgules.

Deux informations sont obligatoires pour chaque colonne : le nom et le type de la colonne. Dans le cas d'une colonne contenant du texte, il est possible de fournir le collationnement de la colonne. Quelque soit la colonne, il est ensuite possible d'ajouter des contraintes.

3.2.19 VALEUR PAR DÉFAUT

- `DEFAULT`
  - affectation implicite
- Utiliser directement par les types sériés

La clause `DEFAULT` permet d'affecter une valeur par défaut lorsqu'une colonne n'est pas référencée dans l'ordre d'insertion ou si une mise à jour réinitialise la valeur de la colonne à sa valeur par défaut.

Les types sériés définissent une valeur par défaut sur les colonnes de ce type. Cette valeur est le retour de la fonction `nextval()` sur la séquence affectée automatiquement à cette colonne.

Exemples

Assignation d'une valeur par défaut :

```
CREATE TABLE valdefaut (
  id integer,
  i integer DEFAULT 0,
  j integer DEFAULT 0
```

```
);

INSERT INTO valdefaut (id, i) VALUES (1, 10);

SELECT * FROM valdefaut ;
 id | i  | j
----+----+---
  1 | 10 | 0
(1 row)
```

3.2.20 COPIE DE LA DÉFINITION D'UNE TABLE

- Création d'une table à partir d'une autre table
  - `CREATE TABLE ... (LIKE table clause_inclusion)`
- Avec les valeurs par défaut des colonnes :
  - `INCLUDING DEFAULTS`
- Avec ses autres contraintes :
  - `INCLUDING CONSTRAINTS`
- Avec ses index :
  - `INCLUDING INDEXES`

L'ordre `CREATE TABLE` permet également de créer une table à partir de la définition d'une table déjà existante en utilisant la clause `LIKE` en lieu et place de la définition habituelles des colonnes. Par défaut, seule la définition des colonnes avec leur typage est repris.

Les clauses `INCLUDING` permettent de récupérer d'autres éléments de la définition de la table, comme les valeurs par défaut (`INCLUDING DEFAULTS`), les contraintes d'intégrité (`INCLUDING CONSTRAINTS`), les index (`INCLUDING INDEXES`), les clauses de stockage (`INCLUDING STORAGE`) ainsi que les commentaires (`INCLUDING COMMENTS`). Si l'ensemble de ces éléments sont repris, il est possible de résumer la clause `INCLUDING` à `INCLUDING ALL`.

La clause `CREATE TABLE` suivante permet de créer une table `archive_evenements_2010` à partir de la définition de la table `evenements` :

```
CREATE TABLE archive_evenements_2010
 (LIKE evenements
  INCLUDING DEFAULTS
  INCLUDING CONSTRAINTS
  INCLUDING INDEXES
  INCLUDING STORAGE
  INCLUDING COMMENTS
);
```

17.12

Elle est équivalente à :

```
CREATE TABLE archive_evenements_2010
 (LIKE evenements
  INCLUDING ALL
);
```

3.2.21 MODIFICATION D'UNE TABLE

- ALTER TABLE
- Définition de la table
  - renommage de la table
  - ajout/modification/suppression d'une colonne
  - déplacement dans un schéma différent
  - changement du propriétaire
- Définition des colonnes
  - renommage d'une colonne
  - changement de type d'une colonne
- Définition des contraintes
  - ajout/suppression d'une contrainte

Pour modifier la définition d'une table (et non pas son contenu), il convient d'utiliser l'ordre ALTER TABLE. Il permet de traiter la définition de la table (nom, propriétaire, schéma, liste des colonnes), la définition des colonnes (ajout, modification de nom et de type, suppression... mais pas de changement au niveau de leur ordre), et la définition des contraintes (ajout et suppression).

3.2.22 SUPPRESSION D'UNE TABLE

- Supprimer une table :

  ```
  DROP TABLE nom_table;
  ```

- Supprimer une table et tous les objets dépendants :

  ```
  DROP TABLE nom_table CASCADE;
  ```

L'ordre DROP TABLE permet de supprimer une table. L'ordre DROP TABLE ... CASCADE permet de supprimer une table ainsi que tous ses objets dépendants. Il peut s'agir de séquences rattachées à une colonne d'une table, à des colonnes référençant la table à supprimer, etc.

3.2.23 CONTRAINTES D'INTÉGRITÉ

- ACID
  - **C**ohérence
  - une transaction amène la base d'un état stable à un autre
- Assurent la cohérence des données
  - unicité des enregistrements
  - intégrité référentielle
  - vérification des valeurs
  - identité des enregistrements
  - règles sémantiques

Les données dans les différentes tables ne sont pas indépendantes mais obéissent à des règles sémantiques mises en place au moment de la conception du modèle conceptuel des données. Les contraintes d'intégrité ont pour principal objectif de garantir la cohérence des données entre elles, et donc de veiller à ce qu'elles respectent ces règles sémantiques. Si une insertion, une mise à jour ou une suppression viole ces règles, l'opération est purement et simplement annulée.

3.2.24 CLÉS PRIMAIRES

- Identifie une ligne de manière unique
- Une seule clé primaire par table
- Une ou plusieurs colonnes
- À choisir parmi les clés candidates
  - parfois, utiliser une clé artificielle

Une clé primaire permet d'identifier une ligne de façon unique, il n'en existe qu'une seule par table.

Une clé primaire garantit que toutes les valeurs de la ou des colonnes qui composent cette clé sont uniques et non nulles. Elle peut être composée d'une seule colonne ou de plusieurs colonnes, selon le besoin.

La clé primaire est déterminée au moment de la conception du modèle de données.

Les clés primaires créent implicitement un index qui permet de renforcer cette contrainte.

3.2.25 DÉCLARATION D'UNE CLÉ PRIMAIRE

Construction :

```
[CONSTRAINT nom_contrainte]
PRIMARY KEY ( nom_colonne [, ... ] )
```

Exemples

Définition de la table `region` :

```
CREATE TABLE region
  (
    id        serial   UNIQUE NOT NULL,
    libelle   text     NOT NULL,

    PRIMARY KEY(id)
  );

INSERT INTO region VALUES (1, 'un');
INSERT INTO region VALUES (2, 'deux');

INSERT INTO region VALUES (NULL, 'trois');
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (null, trois).

INSERT INTO region VALUES (1, 'trois');
ERROR:  duplicate key value violates unique constraint "region_pkey"
DETAIL:  Key (id)=(1) already exists.

INSERT INTO region VALUES (3, 'trois');

SELECT * FROM region;
 id | libelle
----+---------
  1 | un
  2 | deux
  3 | trois
(3 rows)
```

3.2.26 CONTRAINTE D'UNICITÉ

- Garantie l'unicité des valeurs d'une ou plusieurs colonnes
- Permet les valeurs `NULL`
- Clause `UNIQUE`

- Contrainte UNIQUE != index UNIQUE

Une contrainte d'unicité permet de garantir que les valeurs de la ou des colonnes sur lesquelles porte la contrainte sont uniques. Elle autorise néanmoins d'avoir plusieurs valeurs NULL car elles ne sont pas considérées comme égales mais de valeur inconnue (UNKNOWN).

Une contrainte d'unicité peut être créée simplement en créant un index UNIQUE approprié. Ceci est fortement déconseillé du fait que la contrainte ne sera pas référencée comme telle dans le schéma de la base de données. Il sera donc très facile de ne pas la remarquer au moment d'une reprise du schéma pour une évolution majeure de l'application. Une colonne possédant un index UNIQUE peut malgré tout être référencée par une clé étrangère.

Les contraintes d'unicité créent implicitement un index qui permet de renforcer cette unicité.

3.2.27 DÉCLARATION D'UNE CONTRAINTE D'UNICITÉ

Construction :

```
[ CONSTRAINT nom_contrainte]
{ UNIQUE ( nom_colonne [, ... ] )
```

3.2.28 INTÉGRITÉ RÉFÉRENTIELLE

- **Contrainte d'intégrité référentielle**
  - ou **Clé étrangère**
- Référence une **clé primaire** ou un groupe de colonnes UNIQUE et NOT NULL
- Garantie l'intégrité des données
- FOREIGN KEY

Une clé étrangère sur une table fait référence à une clé primaire ou une contrainte d'unicité d'une autre table. La clé étrangère garantit que les valeurs des colonnes de cette clé existent également dans la table portant la clé primaire ou la contrainte d'unicité. On parle de contrainte référentielle d'intégrité : la contrainte interdit les valeurs qui n'existent pas dans la table référencée.

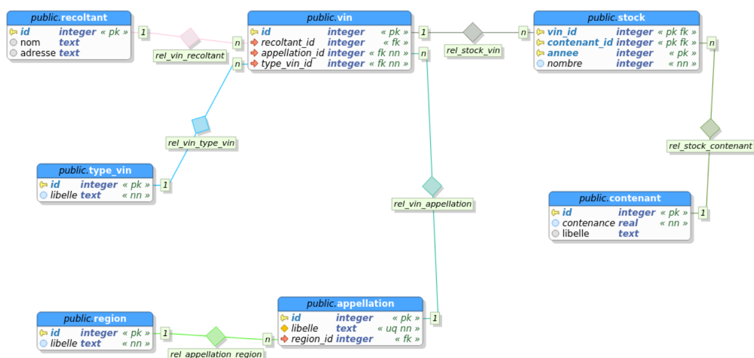
Ainsi, la base cave définit une table region et une table appellation. Une appellation d'origine est liée au terroir, et par extension à son origine géographique. La table

17.12

appellation est donc liée par une clé étrangère à la table region : la colonne region_id de la table appellation référence la colonne id de la table region.

Cette contrainte permet d'empêcher les utilisateurs d'entrer dans la table appellation des identifiants de région (region_id) qui n'existent pas dans la table region.

3.2.29 EXEMPLE



3.2.30 DÉCLARATION D'UNE CLÉ ÉTRANGÈRE

```
[ CONSTRAINT nom_contrainte ] FOREIGN KEY ( nom_colonne [, ...] )
    REFERENCES table_reference [ (colonne_reference [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
```

Exemples

Définition de la table stock :

```
CREATE TABLE stock
  (
    vin_id          int     not null,
    contenant_id    int     not null,
    annee           int4    not null,
    nombre          int4    not null,

    PRIMARY KEY(vin_id,contenant_id,annee),
```

```
    FOREIGN KEY(vin_id) REFERENCES vin(id) ON DELETE CASCADE,
    FOREIGN KEY(contenant_id) REFERENCES contenant(id) ON DELETE CASCADE
  );
```

Création d'une table mère et d'une table fille. La table fille possède une clé étrangère qui référence la table mère :

```
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;

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

-- l'ajout de données dans la table fille qui font bien référence
-- à la table mere fonctionne
INSERT INTO fille (id, mere_id, t) VALUES (1, 1, 'val1');
INSERT INTO fille (id, mere_id, t) VALUES (2, 2, 'val2');

-- l'ajout de données dans la table fille qui ne font pas référence
-- à la table mere est annulé
INSERT INTO fille (id, mere_id, t) VALUES (3, 3, 'val3');
ERROR:  insert or update on table "fille" violates foreign key constraint
        "fk_mere_fille"
DETAIL:  Key (mere_id)=(3) is not present in table "mere".

b1=# SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
(2 rows)

-- mettre à jour la référence dans la table mere ne fonctionnera pas
-- car la contrainte a été définie pour refuser les mises à jour
-- (ON UPDATE NO ACTION)
```

17.12

```
b1=# UPDATE mere SET id=3 WHERE id=2;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(2) is still referenced from table "fille".

-- par contre, la suppression d'une ligne de la table mere référencée dans la
-- table fille va propager la suppression jusqu'à la table fille
-- (ON DELETE CASCADE)

b1=# DELETE FROM mere WHERE id=2;
DELETE 1

b1=# SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
(1 row)

b1=# SELECT * FROM mere;
 id |  t
----+------
  1 | val1
(1 row)
```

3.2.31 VÉRIFICATION SIMPLE OU COMPLÈTE

- Vérification complète ou partielle d'une clé étrangère
- `MATCH`
  - `MATCH FULL` (complète)
  - `MATCH SIMPLE` (partielle)

La directive `MATCH` permet d'indiquer si la contrainte doit être entièrement vérifiée (`MATCH FULL`) ou si la clé étrangère autorise des valeurs `NULL` (`MATCH SIMPLE`). `MATCH SIMPLE` est la valeur par défaut.

Avec `MATCH FULL`, toutes les valeurs des colonnes qui composent la clé étrangère de la table référençant doivent avoir une correspondance dans la table référencée.

Avec `MATCH SIMPLE`, les valeurs des colonnes qui composent la clé étrangère de la table référençant peuvent comporter des valeurs `NULL`. Dans le cas des clés étrangères multi-colonnes, toutes les colonnes peuvent ne pas être renseignées. Dans le cas des clés étrangères sur une seule colonne, la contrainte autorise les valeurs `NULL`.

Exemples

Les exemples reprennent les tables `mere` et `fille` créées plus haut.

```
INSERT INTO fille VALUES (4, NULL, 'test');

SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
  4 |         | test
(2 rows)
```

3.2.32 COLONNES D'IDENTITÉ

- Identité d'un enregistrement
- `GENERATED ... AS IDENTITY`
  - `ALWAYS`
  - `BY DEFAULT`
- Préférer à `serial`
- Unicité non garantie sans contrainte explicite !

Cette contrainte permet d'avoir une colonne dont la valeur est incrémentée automatiquement, soit en permanence (clause `ALWAYS`), soit quand aucune valeur n'est saisie (clause `BY DEFAULT`). Cette solution au besoin d'auto-incrémentation des valeurs des colonnes satisfait le standard SQL, contrairement au pseudo-type `serial` qui était utilisé jusqu'à la version 10.

De plus, elle corrige certains défauts de ce pseudo-type. Avec le type `serial`, l'utilisation de `CREATE TABLE .. LIKE` copiait la contrainte de valeur par défaut sans changer le nom de la séquence. Il n'est pas possible d'ajouter ou de supprimer un pseudo-type `serial` avec l'instruction `ALTER TABLE`. La suppression de la contrainte `DEFAULT` d'un type `serial` ne supprime pas la séquence associée. Tout ceci fait que la définition d'une colonne d'identité est préférable à l'utilisation du pseudo-type `serial`.

Il reste obligatoire de définir une clé primaire ou unique si l'on tient à l'unicité des valeurs car même une clause `GENERATED ALWAYS AS IDENTITY` peut être contournée avec une mise à jour portant la mention `OVERRIDING SYSTEM VALUE`.

Exemple :

```
CREATE table personnes (id int GENERATED ALWAYS AS IDENTITY, nom TEXT);
```

```
17.12
CREATE TABLE

INSERT INTO personnes (nom) VALUES ('Dupont') ;
INSERT 0 1
INSERT INTO personnes (nom) VALUES ('Durand') ;
INSERT 0 1

SELECT * FROM personnes ;
 id |  nom
----+--------
  1 | Dupont
  2 | Durand
(2 lignes)

INSERT INTO personnes (id,nom) VALUES (3,'Martin') ;

ERROR:  cannot insert into column "id"
DÉTAIL : Column "id" is an identity column defined as GENERATED ALWAYS.
ASTUCE : Use OVERRIDING SYSTEM VALUE to override.

INSERT INTO personnes (id,nom) OVERRIDING SYSTEM VALUE VALUES (3,'Martin') ;
INSERT 0 1

INSERT INTO personnes (id,nom) OVERRIDING SYSTEM VALUE VALUES (3,'Dupond') ;
INSERT 0 1

SELECT * FROM personnes ;
 id |  nom
----+--------
  1 | Dupont
  2 | Durand
  3 | Martin
  3 | Dupond
```

3.2.33 MISE À JOUR DE LA CLÉ PRIMAIRE

- Que faire en cas de mise à jour d'une clé primaire ?
  - les clés étrangères seront fausses
- `ON UPDATE`
- `ON DELETE`
- Définition d'une action au niveau de la clé étrangère

DALIBO

- interdiction
- propagation de la mise à jour
- NULL
- valeur par défaut

Si des valeurs d'une clé primaire sont mises à jour ou supprimées, cela peut entrainer des incohérences dans la base de données si des valeurs de clés étrangères font référence aux valeurs de la clé primaire touchées par le changement.

Afin de pouvoir gérer cela, la norme SQL prévoit plusieurs comportements possibles. La clause `ON UPDATE` permet de définir comment le SGBD va réagir si la clé primaire référencée est mise à jour. La clause `ON DELETE` fait de même pour les suppressions.

Les actions possibles sont :

- `NO ACTION` (ou `RESTRICT`), qui produit une erreur si une ligne référence encore le ou les lignes touchées par le changement ;
- `CASCADE`, pour laquelle la mise à jour ou la suppression est propagée aux valeurs référençant le ou les lignes touchées par le changement ;
- `SET NULL`, la valeur de la colonne devient `NULL` ;
- `SET DEFAULT`, pour lequel la valeur de la colonne prend la valeur par défaut de la colonne.

Le comportement par défaut est `NO ACTION`, ce qui est habituellement recommandé pour éviter les suppressions en chaîne mal maîtrisées.

Exemples

Les exemples reprennent les tables `mere` et `fille` créées plus haut.

Tentative d'insertion d'une ligne dont la valeur de `mere_id` n'existe pas dans la table `mere` :

```
INSERT INTO fille (id, mere_id, t) VALUES (1, 3, 'val3');
ERROR:  insert or update on table "fille" violates foreign key constraint
        "fk_mere_fille"
DETAIL:  Key (mere_id)=(3) is not present in table "mere".
```

Mise à jour d'une ligne de la table `mere` pour modifier son `id`. La clé étrangère est déclarée `ON UPDATE NO ACTION`, donc la mise à jour devrait être interdite :

```
UPDATE mere SET id = 3 WHERE id = 1;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

Suppression d'une ligne de la table `mere`. La clé étrangère sur `fille` est déclarée `ON DELETE CASCADE`, la suppression sera donc propagée aux tables qui référencent la table

17.12

`mere` :

```
DELETE FROM mere WHERE id = 1;

SELECT * FROM fille ;
 id | mere_id |  t
----+---------+------
  2 |       2 | val2
(1 row)
```

3.2.34 VÉRIFICATIONS

- Présence d'une valeur
  - `NOT NULL`
- Vérification de la valeur d'une colonne
  - `CHECK`

La clause `NOT NULL` permet de s'assurer que la valeur de la colonne portant cette contrainte est renseignée. Dis autrement, elle doit obligatoirement être renseignée. Par défaut, la colonne peut avoir une valeur `NULL`, donc n'est pas obligatoirement renseignée.

La clause `CHECK` spécifie une expression de résultat booléen que les nouvelles lignes ou celles mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Les expressions de résultat `TRUE` ou `UNKNOWN` réussissent. Si une des lignes de l'opération d'insertion ou de mise à jour produit un résultat `FALSE`, une exception est levée et la base de données n'est pas modifiée. Une contrainte de vérification sur une colonne ne fait référence qu'à la valeur de la colonne tandis qu'une contrainte sur la table fait référence à plusieurs colonnes.

Actuellement, les expressions `CHECK` ne peuvent ni contenir des sous-requêtes ni faire référence à des variables autres que les colonnes de la ligne courante. C'est techniquement réalisable, mais non supporté.

3.2.35 VÉRIFICATIONS DIFFÉRÉS

- Vérifications après chaque ordre SQL
  - problèmes de cohérence
- Différer les vérifications de contraintes
  - clause `DEFERRABLE`, `NOT DEFERRABLE`
  - `INITIALLY DEFERED`, `INITIALLY IMMEDIATE`

Par défaut, toutes les contraintes d'intégrité sont vérifiées lors de l'exécution de chaque ordre SQL de modification, y compris dans une transaction. Cela peut poser des problèmes de cohérences de données : insérer dans une table fille alors qu'on n'a pas encore inséré les données dans la table mère, la clé étrangère de la table fille va rejeter l'insertion et annuler la transaction.

Le moment où les contraintes sont vérifiées est modifiable dynamiquement par l'ordre `SET CONSTRAINTS` :

```
SET CONSTRAINTS { ALL | nom [, ...] } { DEFERRED | IMMEDIATE }
```

mais ce n'est utilisable que pour les contraintes déclarées comme déferrables.

Voici quelques exemples :

- avec la définition précédente des tables `mere` et `fille`

```
b1=# BEGIN;
UPDATE mere SET id=3 where id=1;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

- cette erreur survient aussi dans le cas où on demande que la vérification des contraintes soit différée pour cette transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED;
UPDATE mere SET id=3 WHERE id=1;
ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

- il faut que la contrainte soit déclarée comme étant différable :

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

17.12

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

BEGIN;
SET CONSTRAINTS all deferred;
UPDATE mere SET id=3 WHERE id=1;
SELECT * FROM mere;
 id |  t
----+------
  2 | val2
  3 | val1
(2 rows)

SELECT * FROM fille;
 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
(2 rows)

UPDATE fille SET mere_id=3 WHERE mere_id=1;
COMMIT;
```

3.2.36 VÉRIFICATIONS PLUS COMPLEXES

- Un trigger
  - si une contrainte porte sur plusieurs tables
  - si sa vérification nécessite une sous-requête
- Préférer les contraintes déclaratives

Les contraintes d'intégrités du SGBD ne permettent pas d'exprimer une contrainte qui porte sur plusieurs tables ou simplement si sa vérification nécessite une sous-requête. Dans ce cas là, il est nécessaire d'écrire un trigger spécifique qui sera déclenché après chaque modification pour valider la contrainte.

Il ne faut toutefois pas systématiser l'utilisation de triggers pour valider des contraintes d'intégrité. Cela aurait un impact fort sur les performances et sur la maintenabilité de la base de données. Il vaut mieux privilégier les contraintes déclaratives et n'envisager l'emploi de triggers que dans les cas où ils sont réellement nécessaires.

3.3 DML : MISE À JOUR DES DONNÉES

- SELECT peut lire les données d'une table ou plusieurs tables
  - mais ne peut pas les mettre à jour
- Ajout de données dans une table
  - INSERT
- Modification des données d'une table
  - UPDATE
- Suppression des données d'une table
  - DELETE

L'ordre SELECT permet de lire une ou plusieurs tables. Les mises à jours utilisent des ordres distincts.

L'ordre INSERT permet d'ajouter ou insérer des données dans une table. L'ordre UPDATE permet de modifier des lignes déjà existantes. Enfin, l'ordre DELETE permet de supprimer des lignes. Ces ordres ne peuvent travailler que sur une seule table à la fois. Si on souhaite par exemple insérer des données dans deux tables, il est nécessaire de réaliser deux INSERT distincts.

3.3.1 AJOUT DE DONNÉES : INSERT

- Ajoute des lignes à partir des données de la requête
- Ajoute des lignes à partir d'une requête SELECT
- Syntaxe :

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    { liste_valeurs | requete }
```

L'ordre INSERT insère de nouvelles lignes dans une table. Il permet d'insérer une ou plusieurs lignes spécifiées par les expressions de valeur, ou zéro ou plusieurs lignes provenant d'une requête.

La liste des noms de colonnes est optionnelle. Si elle n'est pas spécifiée, alors PostgreSQL utilisera implicitement la liste de toutes les colonnes de la table dans l'ordre de leur déclaration, ou les N premiers noms de colonnes si seules N valeurs de colonnes sont fournies dans la clause VALUES ou dans la requête. L'ordre des noms des colonnes dans la liste n'a pas d'importance particulière, il suffit de nommer les colonnes mises à jour.

Chaque colonne absente de la liste, implicite ou explicite, se voit attribuer sa valeur par défaut, s'il y en a une ou NULL dans le cas contraire. Les expressions de colonnes qui

ne correspondent pas au type de données déclarées sont transtypées automatiquement, dans la mesure du possible.

3.3.2 INSERT AVEC LISTE D'EXPRESSIONS

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    VALUES ( { expression | DEFAULT } [, ...] ) [, ...]
```

La clause `VALUES` permet de définir une liste d'expressions qui va constituer la ligne à insérer dans la base de données. Les éléments de cette liste d'expression sont séparés par une virgule. Cette liste d'expression est composée de constantes ou d'appels à des fonctions retournant une valeur, pour obtenir par exemple la date courante ou la prochaine valeur d'une séquence. Les valeurs fournies par la clause `VALUES` ou par la requête sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Exemples

Insertion d'une ligne dans la table `stock` :

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre)
 VALUES (12, 1, 1935, 1);
```

Insertion d'une ligne dans la table `vin` :

```
INSERT INTO vin (id, recoltant_id, appellation_id, type_vin_id)
 VALUES (nextval('vin_id_seq'), 3, 6, 1);
```

3.3.3 INSERT À PARTIR D'UN SELECT

```
INSERT INTO nom_table [ ( nom_colonne [, ...] ) ]
    requête
```

L'ordre `INSERT` peut aussi prendre une requête SQL en entrée. Dans ce cas, `INSERT` va insérer autant de lignes dans la table d'arrivée qu'il y a de lignes retournées par la requête `SELECT`. L'ordre des colonnes retournées par `SELECT` doit correspondre à l'ordre des colonnes de la liste des colonnes. Leur type de données doit également correspondre.

Exemples

Insertion dans une table `stock2` à partir d'une requête SELECT sur la table `stock1` :

```
INSERT INTO stock2 (vin_id, contenant_id, annee, nombre)
SELECT vin_id, contenant_id, annee, nombre FROM stock;
```

3.3.4 INSERT ET COLONNES IMPLICITES

- L'ordre physique peut changer dans le temps
  - résultats incohérents
  - requêtes en erreurs

Il est préférable de lister explicitement les colonnes touchées par l'ordre `INSERT` afin de garder un ordre d'insertion déterministe. En effet, 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.

Exemples

Exemple de modification du schéma pouvant entrainer 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);

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);
```

3.3.5 MISE À JOUR DE DONNÉES : UPDATE

- Ordre `UPDATE`

17.12

- Met à jour une ou plusieurs colonnes d'une même ligne
  - à partir des valeurs de la requête
  - à partir des anciennes valeurs
  - à partir d'une requête SELECT
  - à partir de valeurs d'une autre table

L'ordre de mise à jour de lignes s'appelle `UPDATE`.

3.3.6 CONSTRUCTION D'UPDATE

```
UPDATE nom_table
    SET
    {
     nom_colonne = { expression | DEFAULT }
    |
     ( nom_colonne [, ...] ) = ( { expression | DEFAULT } [, ...] )
    } [, ...]
    [ FROM liste_from ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
```

L'ordre `UPDATE` permet de mettre à jour les lignes d'une table.

L'ordre `UPDATE` ne met à jour que les lignes qui satisfont les conditions de la clause `WHERE`. La clause `SET` permet de définir les colonnes à mettre à jour. Le nom des colonne mises à jour doivent faire partie de la table mise à jour.

Les valeurs mises à jour peuvent faire référence aux valeurs avant mise à jour de la colonne, dans ce cas on utilise la forme `nom_colonne = nom_colonne`. La partie de gauche référence la colonne à mettre à jour, la partie de droite est une expression qui permet de déterminer la valeur à appliquer à la colonne. La valeur à appliquer peut bien entendu être une référence à une ou plusieurs colonnes et elles peuvent être dérivées par une opération arithmétique.

La clause `FROM` ne fait pas partie de la norme SQL mais certains SGBDR la supportent, notamment SQL Server et PostgreSQL. Elle permet de réaliser facilement la mise à jour d'une table à partir des valeurs d'une ou plusieurs tables annexes.

La norme SQL permet néanmoins de réaliser des mises à jour en utilisant une sous-requête, permettant d'éviter l'usage de la clause `FROM`.

Exemples

Mise à jour du prix d'un livre particulier :

```
UPDATE livres SET prix = 10 WHERE isbn = '978-3-8365-3872-5';
```

Augmentation de 5% du prix des livres :

```
UPDATE livres SET prix = prix * 1.05;
```

Mise à jour d'une table `employees` à partir des données d'une table `bonus_plan` :

```
UPDATE employees e
   SET commission_rate = bp.commission_rate
  FROM bonus_plan bp
    ON (e.bonus_plan = bp.planid)
```

La même requête avec une sous-requête, conforme à la norme SQL :

```
UPDATE employees
   SET commission_rate = (SELECT commission_rate
                            FROM bonus_plan bp
                           WHERE bp.planid = employees.bonus_plan);
```

Lorsque plusieurs colonnes doivent être mises à jour à partir d'une jointure, il est possible d'utiliser ces deux écritures :

```
UPDATE employees e
   SET commission_rate = bp.commission_rate,
       commission_rate2 = bp.commission_rate2
  FROM bonus_plan bp
    ON (e.bonus_plan = bp.planid);
```

et

```
UPDATE employees e
   SET (commission_rate, commission_rate2) = (
      SELECT bp.commission_rate, bp.commission_rate2
      FROM bonus_plan bp ON (e.bonus_plan = bp.planid)
   );
```

3.3.7 SUPPRESSION DE DONNÉES : DELETE

- Supprime les lignes répondant au prédicat
- Syntaxe :

```
DELETE FROM nom_table [ [ AS ] alias ]
    [ WHERE condition
```

L'ordre `DELETE` supprime l'ensemble des lignes qui répondent au prédicat de la clause `WHERE`.

```
DELETE FROM nom_table [ [ AS ] alias ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
```

17.12

Exemples

Suppression d'un livre épuisé du catalogue :

```
DELETE FROM livres WHERE isbn = '978-0-8707-0635-6';
```

3.3.8 CLAUSE RETURNING

- Spécifique à PostgreSQL
- Permet de retourner les lignes complètes ou partielles résultants de `INSERT`, `UPDATE` ou `DELETE`
- Syntaxe :

```
requete_sql RETURNING ( * | expression )
```

La clause `RETURNING` est une extension de PostgreSQL. Elle permet de retourner les lignes insérées, mises à jour ou supprimées par un ordre DML de modification. Il est également possible de dériver une valeur retournée.

L'emploi de la clause `RETURNING` peut nécessiter des droits complémentaires sur les objets de la base.

Exemples

Mise à jour du nombre de bouteilles en stock :

```
SELECT annee, nombre FROM stock
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967;
 annee | nombre
-------+--------
  1967 |     17
(1 row)

UPDATE stock SET nombre = nombre - 1
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967 RETURNING nombre;
 nombre
--------
     16
(1 row)
```

3.4 TRANSACTIONS

- ACID

    - atomicité
    - un traitement se fait en entier ou pas du tout
- TCL pour Transaction Control Language
    - valide une transaction
    - annule une transaction
    - points de sauvegarde

Les transactions sont une partie essentielle du langage SQL. Elles permettent de rendre atomique un certain nombre de requêtes. Le résultat de toutes les requêtes d'une transaction est validée ou pas, mais on ne peut pas avoir d'état intermédiaire.

Le langage SQL définit qu'une transaction peut être validée ou annulée. Ce sont respectivement les ordres `COMMIT` et `ROLLBACK`. Il est aussi possible de faire des points de reprise ou de sauvegarde dans une transaction. Ils se font en utilisant l'ordre `SAVEPOINT`.

3.4.1 AUTO-COMMIT ET TRANSACTIONS

- Par défaut, PostgreSQL fonctionne en auto-commit
    - à moins d'ouvrir explicitement une transaction
- Ouvrir une transaction
    - `BEGIN TRANSACTION`

PostgreSQL fonctionne en auto-commit. Autrement dit, sans `BEGIN`, une requête est considérée comme une transaction complète et n'a donc pas besoin de `COMMIT`.

Une transaction débute toujours par un `START` ou un `BEGIN`.

3.4.2 VALIDATION OU ANNULATION D'UNE TRANSACTION

- Valider une transaction
    - `COMMIT`
- Annuler une transaction
    - `ROLLBACK`
- Sans validation, une transaction est forcément annulée

Une transaction est toujours terminée par une `COMMIT` ou un `END` quand on veut que les modifications soient définitivement enregistrées, et par un `ROLLBACK` dans le cas contraire.

17.12

La transaction en cours d'une session qui se termine, quelle que soit la raison, sans `COMMIT` et sans `ROLLBACK` est considérée comme annulée.

Exemples

Avant de retirer une bouteille du stock, on vérifie tout d'abord qu'il reste suffisamment de bouteilles en stock :

```
BEGIN TRANSACTION;

SELECT annee, nombre FROM stock WHERE vin_id = 7 AND contenant_id = 1
AND annee = 1967;
 annee | nombre
-------+--------
  1967 |     17
(1 row)

UPDATE stock SET nombre = nombre - 1
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967 RETURNING nombre;
 nombre
--------
     16
(1 row)

COMMIT;
```

3.4.3 PROGRAMMATION

- Certains langages implémentent des méthodes de gestion des transactions
  - PHP, Java, etc.
- Utiliser ces méthodes prioritairement

La plupart des langages permettent de gérer les transactions à l'aide de méthodes ou fonctions particulières. Il est recommandé de les utiliser.

En Java, ouvrir une transaction revient à désactiver l'auto-commit :

```
String url =
    "jdbc:postgresql://localhost/test?user=fred&password=secret&ssl=true";
Connection conn = DriverManager.getConnection(url);
conn.setAutoCommit(false);
```

La transaction est confirmée (`COMMIT`) avec la méthode suivante :

```
conn.commit();
```

À l'inverse, elle est annulée (`ROLLBACK`) avec la méthode suivante :

```
conn.rollback();
```

3.4.4 POINTS DE SAUVEGARDE

- Certains traitements dans une transaction peuvent être annulés
  - mais la transaction est atomique
- Définir un point de sauvegarde
  - `SAVEPOINT nom_savepoint`
- Valider le traitement depuis le dernier point de sauvegarde
  - `RELEASE SAVEPOINT nom_savepoint`
- Annuler le traitement depuis le dernier point de sauvegarde
  - `ROLLBACK TO SAVEPOINT nom_savepoint`

déroule jusqu'au bout, le point de sauvegarde pourra être relâché (`RELEASE SAVEPOINT`), confirmant ainsi les traitements. Si le traitement tombe en erreur, il suffira de revenir au point de sauvegarde (`ROLLBACK TO SAVEPOINT` pour annuler uniquement cette partie du traitement sans affecter le reste de la transaction.

Les points de sauvegarde sont des éléments nommés, il convient donc de leur affecter un nom particulier. Leur nom doit être unique dans la transaction courante.

Les langages de programmation permettent également de gérer les points de sauvegarde en utilisant des méthodes dédiées. Par exemple, en Java :

```
Savepoint save1 = connection.setSavepoint();
```

En cas d'erreurs, la transaction peut être ramener à l'état du point de sauvegarde avec :

```
connection.rollback(save1);
```

À l'inverse, un point de sauvegarde est relâché de la façon suivante :

```
connection.releaseSavepoint(save1);
```

Exemples

Transaction avec un point de sauvegarde et la gestion de l'erreur :

```
BEGIN;

INSERT INTO mere (id, val_mere) VALUES (10, 'essai');

SAVEPOINT insert_fille;

INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (1, 10, 'essai 2');
ERROR:  duplicate key value violates unique constraint "fille_pkey"
```

17.12

```
DETAIL:  Key (id_fille)=(1) already exists.

ROLLBACK TO SAVEPOINT insert_fille;

COMMIT;

SELECT * FROM mere;
 id | val_mere
----+----------
  1 | mere 1
  2 | mere 2
 10 | essai
```

3.5 CONCLUSION

- SQL : toujours un traitement d'ensembles d'enregistrements
  - c'est le côté relationnel
- Pour les définitions d'objets
  - `CREATE`, `ALTER`, `DROP`
- Pour les données
  - `INSERT`, `UPDATE`, `DELETE`

Le standard SQL permet de traiter des ensembles d'enregistrements, que ce soit en lecture, en insertion, en modification et en suppression. Les ensembles d'enregistrements sont généralement des tables qui, comme tous les autres objets, sont créées (`CREATE`), modifier (`ALTER`) et/ou supprimer (`DROP`).

3.5.1 QUESTIONS

N'hésitez pas, c'est le moment !

3.6 TRAVAUX PRATIQUES

3.6.1 ÉNONCÉS

Pour cet exercice, les modifications de schéma doivent être effectuées par un rôle ayant suffisamment de droits pour modifier son schéma. Le rôle `tpc_admin` a les droits suffisants.

1. Ajouter une colonne `email` de type `text` à la table `contacts`. Cette colonne va permettre de stocker l'adresse e-mail des clients et des fournisseurs. Ajouter également un commentaire décrivant cette colonne dans le catalogue de PostgreSQL (utiliser la commande `COMMENT`).

2. Mettre à jour la table des contacts pour indiquer l'adresse e-mail de *Client6657* qui est `client6657@dalibo.com`.

3. Ajouter une contrainte d'intégrité qui valide que la valeur de la colonne `email` créée est bien formée (vérifier que la chaîne de caractère contient au moins le caractère `@`).

4. Valider la contrainte dans une transaction de test.

5. Déterminer quels sont les contacts qui disposent d'une adresse e-mail et affichez leur nom ainsi que le code de leur pays.

6. La génération des numéros de commande est actuellement réalisée à l'aide de la séquence `commandes_commande_id_seq`. Cette méthode ne permet pas de garantir que tous les numéros de commande se suivent. Proposer une solution pour sérialiser la génération des numéros de commande. Autrement dit, proposer une méthode pour obtenir un numéro de commande sans avoir de « trou » dans la séquence en cas d'échec d'une transaction.

7. Noter le nombre de lignes de la table `pieces`. Dans une transaction, majorer de 5% le prix des pièces de moins de 1500 € et minorer de 5 % le prix des pièces dont le prix actuel est égal ou supérieur à 1500 €. Vérifier que le nombre de lignes mises à jour au total correspond au nombre total de lignes de la table `pieces`.

8. Dans une même transaction, créer un nouveau client en incluant l'ajout de l'ensemble des informations requises pour pouvoir le contacter. Un nouveau client a un solde égal à 0.

3.6.2 SOLUTIONS

1. Ajouter une colonne `email` de type `text` à la table `contacts`. Cette colonne va permettre de stocker l'adresse e-mail des clients et des fournisseurs. Ajouter égale-

ment un commentaire décrivant cette colonne dans le catalogue de PostgreSQL (utiliser la commande COMMENT).

```
ALTER TABLE contacts
   ADD COLUMN email text
;

COMMENT ON COLUMN contacts.email IS
   'Adresse e-mail du contact'
;
```

2. Mettre à jour la table des contacts pour indiquer l'adresse e-mail de *Client6657* qui est client6657@dalibo.com.

```
UPDATE contacts
   SET email = 'client6657@dalibo.com'
WHERE nom = 'Client6657'
;
```

Vérifier les résultats :

```
SELECT *
FROM contacts
WHERE nom = 'Client6657'
;
```

3. Ajouter une contrainte d'intégrité qui valide que la valeur de la colonne email créée est bien formée (vérifier que la chaîne de caractère contient au moins le caractère @).

```
ALTER TABLE contacts
ADD CONSTRAINT chk_contacts_email_valid
   CHECK (email LIKE '%@%')
;
```

Cette expression régulière est simplifiée et simpliste pour les besoins de l'exercice. Des expressions régulières plus complexes permettent de valider réellement une adresse e-mail.

Voici un exemple un tout petit peu plus évolué en utilisant une expression rationnelle simple, ici pour vérifier que la chaîne précédent le caractère @ contient au moins un caractère, et que la chaîne le suivant est une chaîne de caractères contenant un point :

```
ALTER TABLE contacts
ADD CONSTRAINT chk_contacts_email_valid
   CHECK (email ~ '.+@.+\..+')
;
```

4. Valider la contrainte dans une transaction de test.

Démarrer la transaction :

```
BEGIN ;
```

Tenter de mettre à jour la table `contacts` avec une adresse e-mail ne répondant pas à la contrainte :

```
UPDATE contacts
   SET email = 'test'
;
```

L'ordre `UPDATE` retourne l'erreur suivante, indiquant que l'expression régulière est fonctionnelle :

```
ERROR:  new row for relation "contacts" violates check constraint
        "chk_contacts_email_valid"
DETAIL: Failing row contains
        (300001, Client1737, nkD, SA, 20-999-929-1440, test).
```

La transaction est ensuite annulée :

```
ROLLBACK ;
```

5. Déterminer quels sont les contacts qui disposent d'une adresse e-mail et afficher leur nom ainsi que le code de leur pays.

```
SELECT nom, code_pays
FROM contacts
WHERE email IS NOT NULL
;
```

6. La génération des numéros de commande est actuellement réalisée à l'aide de la séquence `commandes_commande_id_seq`. Cette méthode ne permet pas de garantir que tous les numéros de commande se suivent. Proposer une solution pour sérialiser la génération des numéros de commande. Autrement dit, proposer une méthode transactionnelle pour obtenir un numéro de commande, sans avoir de « trou » dans la séquence en cas d'échec d'une transaction.

La solution la plus simple pour imposer la sérialisation des numéros de commandes est d'utiliser une table de séquences. Une ligne de cette table correspondra au compteur des numéros de commande.

```
-- création de la table qui va contenir la séquence :
CREATE TABLE numeros_sequences (
   nom text NOT NULL PRIMARY KEY,
   sequence integer NOT NULL
)
;
```

```
-- initialisation de la séquence :
INSERT INTO numeros_sequences (nom, sequence)
SELECT 'sequence_numero_commande', max(numero_commande)
FROM commandes
;
```

L'obtention d'un nouveau numéro de commande sera réalisé dans la transaction de création de la commande de la façon suivante :

```
BEGIN ;

UPDATE numeros_sequences
   SET sequence = sequence + 1
WHERE nom = 'numero_commande'
RETURNING sequence
;

/* insertion d'une nouvelle commande en utilisant le numéro de commande
   retourné par la commande précédente :
   INSERT INTO commandes (numero_commande, ...)
   VALUES (<la nouvelle valeur de la séquence>, ...) ;
*/

COMMIT ;
```

L'ordre `UPDATE` pose un verrou exclusif sur la ligne mise à jour. Tant que la mise à jour n'aura pas été validée ou annulée par `COMMIT` ou `ROLLBACK`, le verrou posé va bloquer toutes les autres transactions qui tenteraient de mettre à jour cette ligne. De cette façon, toutes les transactions seront sérialisées.

Concernant la génération des numéros de séquence, si la transaction est annulée, alors le compteur `sequence` retrouvera sa valeur précédente et la transaction suivante obtiendra le même numéro de séquence. Si la transaction est validée, alors le compteur `sequence` est incrémenté. La transaction suivante verra alors cette nouvelle valeur et non plus l'ancienne. Cette méthode garantit qu'il n'y ait pas de rupture de séquence.

Il va de soi que les transactions de création de commandes doivent être extrêmement courtes. Si une telle transaction est bloquée, toutes les transactions suivantes seront également bloquées, paralysant ainsi tous les utilisateurs de l'application.

7. Noter le nombre de lignes de la table `pieces`. Dans une transaction, majorer de 5 % le prix des pièces de moins de 1500 € et minorer de 5 % le prix des pièces dont le prix actuel est égal ou supérieur à 1500 €. Vérifier que le nombre de lignes mises à jour au total correspond au nombre total de lignes de la table `pieces`.

```
BEGIN ;
```

```
SELECT count(*)
FROM pieces
;

UPDATE pieces
   SET prix = prix * 1.05
WHERE prix < 1500
;

UPDATE pieces
   SET prix = prix * 0.95
WHERE prix >= 1500
;
```

Au total, la transaction a mis à jour 214200 (99922+114278) lignes, soit 14200 lignes de trop mises à jour.

Annuler la mise à jour :

```
ROLLBACK ;
```

Explication : Le premier `UPDATE` a majoré de 5 % les pièces dont le prix est inférieur à 1500 €. Or, tous les prix supérieurs à 1428,58 € passent la barre des 1500 € après le premier `UPDATE`. Le second `UPDATE` minore les pièces dont le prix est égal ou supérieur à 1500 €, ce qui inclue une partie des prix majorés par le précédent `UPDATE`. Certaines lignes ont donc subies *deux* modifications au lieu d'une. L'instruction `CASE` du langage SQL, qui sera abordée dans le prochain module, propose une solution à ce genre de problématique :

```
UPDATE pieces
   SET prix = (
      CASE
         WHEN prix < 1500 THEN prix * 1.05
         WHEN prix >= 1500 THEN prix * 0.95
      END
   )
;
```

8. Dans une même transaction, créer un nouveau client en incluant l'ajout de l'ensemble des informations requises pour pouvoir le contacter. Un nouveau client a un solde égal à 0.

```
-- démarrer la transaction
BEGIN ;

-- créer le contact et récupérer le contact_id généré
INSERT INTO contacts (nom, adresse, telephone, code_pays)
   VALUES ('M. Xyz', '3, Rue du Champignon, 96000 Champiville',
```

```
            '+33554325432', 'FR')
RETURNING contact_id
;

-- réaliser l'insertion en utilisant le numéro de contact récupéré précédemment
INSERT INTO clients (solde, segment_marche, contact_id, commentaire)
   -- par exemple ici avec le numéro 350002
   VALUES (0, 'AUTOMOBILE', 350002, 'Client très important')
;

-- valider la transaction
COMMIT ;
```

4 PLUS LOIN AVEC SQL

4.1 PRÉAMBULE

- Après la définition des objets, leur lecture et leur écriture
- Aller plus loin dans l'écriture de requêtes
  - avec les jointures
  - avec les requêtes intégrées

Maintenant que nous avons vu comment définir des objets, comment lire des données provenant de relation et comment écrire des données, nous allons pousser vers les perfectionnements du langage SQL. Nous allons notamment aborder la lecture de plusieurs tables en même temps, que ce soit par des jointures ou par des sous-requêtes.

4.1.1 MENU

- Valeur `NULL`
- Agrégats, `GROUP BY`, `HAVING`
- Sous-requêtes
- Jointures
- Expression conditionnelle `CASE`
- Opérateurs ensemblistes : `UNION`, `EXCEPT`, `INTERSECT`

4.1.2 OBJECTIFS

- Comprendre l'intérêt du NULL
- Savoir écrire des requêtes complexes

4.2 VALEUR NULL

- Comment représenter une valeur que l'on ne connait pas ?
  - Valeur `NULL`
- Trois sens possibles pour `NULL` :
  - valeur inconnue
  - valeur inapplicable

    - absence de valeur
- Logique 3 états

Le standard SQL définit très précisément la valeur que doit avoir une colonne dont on ne connaît pas la valeur. Il faut utiliser le mot clé NULL. En fait, ce mot clé est utilisé dans trois cas : pour les valeurs inconnues, pour les valeurs inapplicables et pour une absence de valeurs.

4.2.1 AVERTISSEMENT

- Chris J. Date a écrit :
    - *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.*
- Éviter d'utiliser `NULL` le plus possible
    - utiliser `NULL` correctement lorsqu'il le faut

Il ne faut utiliser `NULL` que lorsque cela est réellement nécessaire. La gestion des valeurs NULL est souvent source de confusions et d'erreurs, ce qui explique qu'il est préférable de l'éviter tant qu'on n'entre pas dans les trois cas vu ci-dessus (valeur inconnue, valeur inapplicable, absence de valeur).

4.2.2 ASSIGNATION DE NULL

- Assignation de `NULL` pour `INSERT` et `UPDATE`
- Explicitement :
    - NULL est indiqué explicitement dans les assignations
- Implicitement :
    - la colonne n'est pas affectée par `INSERT`
    - et n'a pas de valeur par défaut
- Empêcher la valeur `NULL`
    - contrainte `NOT NULL`

Il est possible de donner le mot-clé NULL pour certaines colonnes dans les `INSERT` et les `UPDATE`. Si jamais une colonne n'est pas indiqué dans un `INSERT`, elle aura comme valeur sa valeur par défaut (très souvent, il s'agit de `NULL`). Si jamais on veut toujours avoir une valeur dans une colonne particulière, il faut utiliser la clause `NOT NULL` lors de l'ajout de la colonne. C'est le cas pour les clés primaires par exemple.

Voici quelques exemples d'insertion et de mise à jour :

```
CREATE TABLE public.personnes
(
   id serial,
   nom character varying(60) NOT NULL,
   prenom character varying(60),
   date_naissance date,
   CONSTRAINT pk_personnes PRIMARY KEY (id)
);

INSERT INTO personnes(
            nom, prenom, date_naissance)
    VALUES ('Lagaffe', 'Gaston', date '1957-02-28');

-- assignation explicite

INSERT INTO personnes(
            nom, prenom, date_naissance)
    VALUES ('Fantasio', NULL, date '1938-01-01');

-- assignation implicite

INSERT INTO personnes(
            nom, prenom)
    VALUES ('Prunelle', 'Léon');

-- observation des résultats

 id |    nom     | prenom | date_naissance
----+-----------+--------+----------------
  1 | Lagaffe   | Gaston | 1957-02-28
  2 | Fantasio  | (null) | 1938-01-01
  3 | Prunelle  | Léon   | (null)
(3 rows)
```

L'affichage `(null)` dans `psql` est obtenu avec la méta-commande `\pset null (null)`.

4.2.3 CALCULS AVEC NULL

- Utilisation dans un calcul
  - propagation de `NULL`
- `NULL` est inapplicable
  - le résultat vaut `NULL`

17.12

La valeur NULL est définie comme inapplicable. Ainsi, si elle présente dans un calcul, elle est propagée sur l'ensemble du calcul : le résultat vaudra NULL.

Exemples de calcul

Calculs simples :

```
SELECT 1 + 2 AS resultat;
 resultat
----------
        3
(1 row)

SELECT 1 + 2 + NULL AS resultat;
 resultat
----------
   (null)
(1 row)
```

Calcul à partir de l'âge :

```
SELECT nom, prenom,
    1 + extract('year' from age(date_naissance)) AS calcul_age FROM personnes;
   nom    | prenom | calcul_age
----------+--------+------------
 Lagaffe  | Gaston |         60
 Fantasio | (null) |         79
 Prunelle | Léon   |     (null)
(3 rows)
```

Exemple d'utilisation de NULL dans une concaténation :

```
SELECT nom || ' ' || prenom AS nom_complet FROM personnes;
  nom_complet
----------------
 Lagaffe Gaston
 (null)
 Prunelle Léon
(3 rows)
```

L'affichage (null) est obtenu avec la méta-commande \pset null (null) du shell psql.

4.2.4 NULL ET LES PRÉDICATS

- Dans un prédicat du WHERE :
  - opérateur IS NULL ou IS NOT NULL
- AND :

  - vaut `false` si `NULL AND false`
  - vaut `NULL` si `NULL AND true` ou `NULL AND NULL`
- `OR` :
  - vaut `true` si `NULL OR true`
  - vaut `NULL` si `NULL OR false` ou `NULL OR NULL`

Les opérateurs de comparaisons classiques ne sont pas fonctionnels avec une valeur `NULL`. Du fait de la logique à trois états de PostgreSQL, une comparaison avec `NULL` vaut toujours `NULL`, ainsi `expression = NULL` vaudra toujours `NULL` et de même pour `expression <> NULL` vaudra toujours `NULL`. Cette comparaison ne vaudra jamais ni vrai, ni faux.

De ce fait, il existe les opérateurs de prédicats `IS NULL` et `IS NOT NULL` qui permettent de vérifier qu'une expression est `NULL` ou n'est pas `NULL`.

Pour en savoir plus sur la logique ternaire qui régit les règles de calcul des prédicats, se conformer à la page Wikipedia sur la logique ternaire[5] .

Exemples

Comparaison directe avec `NULL`, qui est invalide :

```
SELECT * FROM personnes WHERE date_naissance = NULL;
 id | nom | prenom | date_naissance
----+-----+--------+----------------
(0 rows)
```

L'opérateur `IS NULL` permet de retourner les lignes dont la date de naissance n'est pas renseignée :

```
SELECT * FROM personnes WHERE date_naissance IS NULL;
 id |    nom    | prenom | date_naissance
----+-----------+--------+-----------------
  3 | Prunelle  | Léon   | (null)
(1 row)
```

4.2.5 NULL ET LES AGRÉGATS

- Opérateurs d'agrégats
  - ignorent `NULL`
  - sauf `count(*)`

Les fonctions d'agrégats ne tiennent pas compte des valeurs `NULL` :

[5] http://fr.wikipedia.org/wiki/Logique_ternaire

17.12

```
SELECT SUM(extract('year' from age(date_naissance))) AS age_cumule
FROM personnes;
 age_cumule
------------
        139
(1 row)
```

Sauf `count(*)` et uniquement `count(*)`, la fonction `count(_expression_)` tient compte des valeurs `NULL` :

```
SELECT count(*) AS compte_lignes, count(date_naissance) AS compte_valeurs
  FROM (SELECT date_naissance
         FROM personnes) date_naissance;
 compte_lignes | compte_valeurs
---------------+----------------
             3 |              2
(1 row)
```

4.2.6 COALESCE

- Remplacer `NULL` par une autre valeur
  - `COALESCE(attribut, ...);`

Cette fonction permet de tester une colonne et de récupérer sa valeur si elle n'est pas NULL et une autre valeur dans le cas contraire. Elle peut avoir plus de deux arguments. Dans ce cas, la première expression de la liste qui ne vaut pas `NULL` sera retournée par la fonction.

Voici quelques exemples :

Remplace les prénoms non-renseignés par la valeur `X` dans le résultat :

```
SELECT nom, COALESCE(prenom, 'X') FROM personnes;
   nom    | coalesce
----------+----------
 Lagaffe  | Gaston
 Fantasio | X
 Prunelle | Léon
(3 rows)
```

Cette fonction est efficace également pour la concaténation précédente :

```
SELECT nom || ' ' || COALESCE(prenom, '') AS nom_complet FROM personnes;
  nom_complet
----------------
 Lagaffe Gaston
```

```
 Fantasio
 Prunelle Léon
(3 rows)
```

4.3 AGRÉGATS

- Regroupement de données
- Calculs d'agrégats

Comme son nom l'indique, l'agrégation permet de regrouper des données, qu'elles viennent d'une ou de plusieurs colonnes. Le but est principalement de réaliser des calculs sur les données des lignes regroupées.

4.3.1 REGROUPEMENT DE DONNÉES

- Regroupement de données :

  ```
  GROUP BY expression [, ...]
  ```

- Chaque groupe de données est ensuite représenté sur une seule ligne
- Permet d'appliquer des calculs sur les ensembles regroupés
  - comptage, somme, moyenne, etc.

La clause `GROUP BY` permet de réaliser des regroupements de données. Les données regroupées sont alors représentées sur une seule ligne. Le principal intérêt de ces regroupements est de permettre de réaliser des calculs sur ces données.

4.3.2 CALCULS D'AGRÉGATS

- Effectue un calcul sur un ensemble de valeurs
  - somme, moyenne, etc.
- Retourne `NULL` si l'ensemble est vide
  - sauf `count()`

Nous allons voir les différentes fonctions d'agrégats disponibles.

4.3.3 AGRÉGATS SIMPLES

- Comptage :

  ```
  count(expression)
  ```

- compte les lignes : `count(*)`
  - compte les valeurs renseignées : `count(colonne)`
- Valeur minimale :

  ```
  min(expression)
  ```

- Valeur maximale :

  ```
  max(expression)
  ```

La fonction `count()` permet de compter les éléments. La fonction est appelée de deux façons.

La première forme consiste à utiliser `count(*)` qui revient à transmettre la ligne complète à la fonction d'agrégat. Ainsi, toute ligne transmise à la fonction sera comptée, même si elle n'est composée que de valeurs `NULL`. On rencontre parfois une forme du type `count(1)`, qui transmet une valeur arbitraire à la fonction, et qui permettait d'accélérer le temps de traitement sur certains SGBD mais qui reste sans intérêt avec PostgreSQL.

La seconde forme consiste à utiliser une expression, par exemple le nom d'une colonne : `count(nom_colonne)`. Dans ce cas-là, seules les valeurs renseignées, donc non NULL, seront prises en compte. Les valeurs `NULL` seront exclues du comptage.

La fonction `min()` permet de déterminer la valeur la plus petite d'un ensemble de valeurs données. La fonction `max()` permet à l'inverse de déterminer la valeur la plus grande d'un ensemble de valeurs données. Les valeurs `NULL` sont bien ignorées. Ces deux fonctions permettent de travailler sur des données numériques, mais fonctionnent également sur les autres types de données comme les chaînes de caractères.

La documentation de PostgreSQL permet d'obtenir la liste des fonctions d'agrégats disponibles[6] .

Exemples

Différences entre count(*) et count(colonne) :

```
CREATE TABLE test (x INTEGER);
-- insertion de cinq lignes dans la table test
INSERT INTO test (x) VALUES (1), (2), (2), (NULL), (NULL);
```

[6] http://docs.postgresql.fr/9.2/functions-aggregate.html

```
SELECT x, count(*) AS count_etoile, count(x) AS count_x FROM test GROUP BY x;
   x    | count_etoile | count_x
--------+--------------+---------
 (null) |            2 |       0
      1 |            1 |       1
      2 |            2 |       2
(3 rows)
```

Déterminer la date de naissance de la personne la plus jeune :

```
SELECT MAX(date_naissance) FROM personnes;
    max
------------
 1957-02-28
(1 row)
```

4.3.4 CALCULS D'AGRÉGATS

- Moyenne :

  ```
  avg(expression)
  ```

- Somme :

  ```
  sum(expression)
  ```

- Écart-type :

  ```
  stddev(expression)
  ```

- Variance :

  ```
  variance(expression)
  ```

La fonction `avg()` permet d'obtenir la moyenne d'un ensemble de valeurs données. La fonction `sum()` permet, quant à elle, d'obtenir la somme d'un ensemble de valeurs données. Enfin, les fonctions `stddev()` et `variance()` permettent d'obtenir respectivement l'écart-type et la variance d'un ensemble de valeurs données.

Ces fonctions retournent `NULL` si aucune donnée n'est applicable. Elles ne prennent en compte que des valeurs numériques.

La documentation de PostgreSQL permet d'obtenir la liste des fonctions d'agrégats disponibles[7] .

Exemples

[7] http://docs.postgresql.fr/9.2/functions-aggregate.html

17.12

Quel est le nombre total de bouteilles en stock par millésime ?

```
SELECT annee, sum(nombre) FROM stock GROUP BY annee ORDER BY annee;
 annee |  sum
-------+--------
  1950 | 210967
  1951 | 201977
  1952 | 202183
...
```

Calcul de moyenne avec des valeurs NULL :

```
CREATE TABLE test (a int, b int);
INSERT INTO test VALUES (10,10);
INSERT INTO test VALUES (20,20);
INSERT INTO test VALUES (30,30);
INSERT INTO test VALUES (null,0);

SELECT avg(a), avg(b) FROM test;
         avg          |         avg
----------------------+---------------------
 20.0000000000000000 | 15.0000000000000000
(1 row)
```

4.3.5 AGRÉGATS SUR PLUSIEURS COLONNES

- Possible d'avoir plusieurs paramètres sur la même fonction d'agrégat
- Quelques exemples
  - pente, regr_slope(Y,X)
  - intersection avec l'axe des ordonnées, regr_intercept(Y,X)
  - indice de corrélation, corr (Y,X)

Une fonction d'aggrégat peut aussi prendre plusieurs variables.

Par exemple concernant la méthode des «moindres carrés» :

- pente : `regr_slope(Y,X)`
- intersection avec l'axe des ordonnées : `regr_intercept(Y,X)`
- indice de corrélation : `corr (Y,X)`

Voici un exemple avec un nuage de points proches d'une fonction y=2x+5 :

```
CREATE TABLE test (x real, y real);
INSERT INTO test VALUES (0,5.01), (1,6.99), (2,9.03);

SELECT regr_slope(y,x) FROM test;
```

```
    regr_slope
------------------
 2.00999975204468
(1 ligne)

SELECT regr_intercept(y,x) FROM test;
  regr_intercept
------------------
 5.00000015894572
(1 ligne)

SELECT corr(y,x) FROM test;
       corr
-------------------
 0.999962873745297
```

4.3.6 CLAUSE HAVING

- Filtrer sur des regroupements
  - `HAVING`
- `WHERE` s'applique sur les lignes lues
- `HAVING` s'applique sur les lignes groupées

La clause `HAVING` permet de filtrer les résultats sur les regroupements réalisés par la clause `GROUP BY`. Il est possible d'utiliser une fonction d'agrégat dans la clause `HAVING`.

Il faudra néanmoins faire attention à ne pas utiliser la clause `HAVING` comme clause de filtrage des données lues par la requête. La clause `HAVING` ne doit permettre de filtrer que les données traitées par la requête.

Ainsi, si l'on souhaite le nombre de vins rouge référencés dans le catalogue. La requête va donc exclure toutes les données de la table vin qui ne correspondent pas au filtre `type_vin = 3`. Pour réaliser cela, on utilisera la clause `WHERE`.

En revanche, si l'on souhaite connaître le nombre de vins par type de cépage si ce nombre est supérieur à 2030, on utilisera la clause `HAVING`.

Exemples

```
SELECT type_vin_id, count(*)
  FROM vin
 GROUP BY type_vin_id
HAVING count(*) > 2030;
 type_vin_id | count
```

```
17.12
-------------+-------
           1 |   2031
```

Si la colonne correspondant à la fonction d'agrégat est renommée avec la clause `AS`, il n'est pas possible d'utiliser le nouveau nom au sein de la clause `HAVING`. Par exemple :

```
SELECT type_vin_id, count(*) AS nombre
  FROM vin
 GROUP BY type_vin_id
HAVING nombre > 2030;

ERROR:  column "nombre" does not exist
```

4.4 SOUS-REQUÊTES

- Corrélation requête/sous-requête
- Sous-requêtes retournant une seule ligne
- Sous-requêtes retournant une liste de valeur
- Sous-requêtes retournant un ensemble
- Sous-requêtes retournant un ensemble vide ou non-vide

4.4.1 CORRÉLATION REQUÊTE/SOUS-REQUÊTE

- Fait référence à la requête principale
- Peut utiliser une valeur issue de la requête principale

Une sous-requête peut faire référence à des variables de la requête principale. Ces variables seront ainsi transformées en constante à chaque évaluation de la sous-requête.

La corrélation requête/sous-requête permet notamment de créer des clauses de filtrage dans la sous-requête en utilisant des éléments de la requête principale.

4.4.2 QU'EST-CE QU'UNE SOUS-REQUÊTE ?

- Une requête imbriquée dans une autre requête
- Le résultat de la requête principale dépend du résultat de la sous-requête
- Encadrée par des parenthèses : `(` et `)`

Une sous-requête consiste à exécuter une requête à l'intérieur d'une autre requête. La requête principale peut être une requête de sélection (SELECT) ou une requête de modification (INSERT, UPDATE, DELETE). La sous- requête est obligatoirement un SELECT.

Le résultat de la requête principale dépend du résultat de la sous-requête. La requête suivante effectue la sélection des colonnes d'une autre requête, qui est une sous-requête. La sous-requête effectue une lecture de la table appellation. Son résultat est transformé en un ensemble qui est nommé requete_appellation :

```
SELECT * FROM (SELECT libelle, region_id FROM appellation) requete_appellation;
                  libelle                    | region_id
-------------------------------------------+-----------
 Ajaccio                                   |         1
 Aloxe-Corton                              |         2
...
```

4.4.3 UTILISER UNE SEULE LIGNE

- La sous-requête ne retourne qu'une seule ligne
  - sinon une erreur est levée
- Positionnée
  - au niveau de la liste des expressions retournées par SELECT
  - au niveau de la clause WHERE
  - au niveau d'une clause HAVING

La sous-requête peut être positionnée au niveau de la liste des expressions retournées par SELECT. La sous-requête est alors généralement un calcul d'agrégat qui ne donne en résultat qu'une seule colonne sur une seule ligne. Ce type de sous-requête est peu performant. Elle est en effet appelée pour chaque ligne retournée par la requête principale.

La requête suivante permet d'obtenir le cumul du nombre de bouteilles année par année.

```
SELECT annee,
       sum(nombre) AS stock,
       (SELECT sum(nombre)
          FROM stock s
         WHERE s.annee <= stock.annee) AS stock_cumule
  FROM stock
 GROUP BY annee
 ORDER BY annee;
 annee | stock  | stock_cumule
-------+--------+--------------
  1950 | 210967 |       210967
```

```
  1951 | 201977 |        412944
  1952 | 202183 |        615127
  1953 | 202489 |        817616
  1954 | 202041 |       1019657
...
```

Une telle sous-requête peut également être positionnée au niveau de la clause `WHERE` ou de la clause `HAVING`.

Par exemple, pour retourner la liste des vins rouge :

```
SELECT *
  FROM vin
 WHERE type_vin_id = (SELECT id
                        FROM type_vin
                       WHERE libelle = 'rouge');
```

4.4.4 UTILISER UNE LISTE DE VALEURS

- La sous-requête retourne
  - plusieurs lignes
  - sur une seule colonne
- Positionnée
  - avec une clause `IN`
  - avec une clause `ANY`
  - avec une clause `ALL`

Les sous-requêtes retournant une liste de valeur sont plus fréquemment utilisées. Ce type de sous-requête permet de filtrer les résultats de la requête principale à partir des résultats de la sous-requête.

4.4.5 CLAUSE IN

```
expression IN (sous-requete)
```

- L'expression de gauche est évaluée et vérifiée avec la liste de valeurs de droite
- `IN` vaut `true`
  - si l'expression de gauche correspond à un élément de la liste de droite
- `IN` vaut `false`
  - si aucune correspondance n'est trouvée et la liste ne contient pas `NULL`
- `IN` vaut `NULL`

  - si l'expression de gauche vaut NULL
  - si aucune valeur ne correspond et la liste contient NULL

La clause IN dans la requête principale permet alors d'exploiter le résultat de la sous-requête pour sélectionner les lignes dont une colonne correspond à une valeur retournée par la sous-requête.

L'opérateur IN retourne true si la valeur de l'expression de gauche est trouvée au moins une fois dans la liste de droite. La liste de droite peut contenir la valeur NULL dans ce cas :

```
SELECT 1 IN (1, 2, NULL) AS in;
 in
----
 t
```

Si aucune correspondance n'est trouvée entre l'expression de gauche et la liste de droite, alors IN vaut false :

```
SELECT 1 IN (2, 4) AS in;
 in
----
 f
```

Mais IN vaut NULL si aucune correspondance n'est trouvée et que la liste de droite contient au moins une valeur NULL :

```
SELECT 1 IN (2, 4, NULL) AS in;
   in
--------
 (null)
```

IN vaut également NULL si l'expression de gauche vaut NULL :

```
SELECT NULL IN (2, 4) AS in;
   in
--------
 (null)
```

Exemples

La requête suivante permet de sélectionner les bouteilles du stock de la cave dont la contenance est comprise entre 0,3 litre et 1 litre. Pour répondre à la question, la sous-requête retourne les identifiants de contenant qui correspondent à la condition. La requête principale ne retient alors que les lignes dont la colonne contenant_id correspond à une valeur d'identifiant retournée par la sous-requête.

```
SELECT *
  FROM stock
```

```
WHERE contenant_id IN (SELECT id
                         FROM contenant
                        WHERE contenance
                      BETWEEN 0.3 AND 1.0);
```

4.4.6 CLAUSE NOT IN

```
expression NOT IN (sous-requete)
```

- L'expression de droite est évaluée et vérifiée avec la liste de valeurs de gauche
- `NOT IN` vaut `true`
  - si aucune correspondance n'est trouvée et la liste ne contient pas `NULL`
- `NOT IN` vaut `false`
  - si l'expression de gauche correspond à un élément de la liste de droite
- `NOT IN` vaut `NULL`
  - si l'expression de gauche vaut `NULL`
  - si aucune valeur ne correspond et la liste contient `NULL`

À l'inverse, la clause `NOT IN` permet dans la requête principale de sélectionner les lignes dont la colonne impliquée dans la condition ne correspond pas aux valeurs retournées par la sous-requête.

La requête suivante permet de sélectionner les bouteilles du stock dont la contenance n'est pas inférieure à 2 litres.

```
SELECT *
  FROM stock
 WHERE contenant_id NOT IN (SELECT id
                              FROM contenant
                             WHERE contenance < 2.0);
```

Il est à noter que les requêtes impliquant les clauses `IN` ou `NOT IN` peuvent généralement être réécrites sous la forme d'une jointure.

De plus, les optimiseurs SQL parviennent difficilement à optimiser une requête impliquant `NOT IN`. Il est préférable d'essayer de réécrire ces requêtes en utilisant une jointure.

Avec `NOT IN`, la gestion des valeurs `NULL` est à l'inverse de celle de la clause `IN` :

Si une correspondance est trouvée, `NOT IN` vaut `false` :

```
SELECT 1 NOT IN (1, 2, NULL) AS notin;
 notin
-------
 f
```

Si aucune correspondance n'est trouvée, NOT IN vaut true :

```
SELECT 1 NOT IN (2, 4) AS notin;
 notin
-------
 t
```

Si aucune correspondance n'est trouvée mais que la liste de valeurs de droite contient au moins un NULL, NOT IN vaut NULL :

```
SELECT 1 NOT IN (2, 4, NULL) AS notin;
 notin
--------
 (null)
```

Si l'expression de gauche vaut NULL, alors NOT IN vaut NULL également :

```
SELECT NULL IN (2, 4) AS notin;
 notin
--------
 (null)
```

Les sous-requêtes retournant des valeurs NULL posent souvent des problèmes avec NOT IN. Il est préférable d'utiliser EXISTS ou NOT EXISTS pour ne pas avoir à se soucier des valeurs NULL.

4.4.7 CLAUSE ANY

```
expression operateur ANY (sous-requete)
```

- L'expression de gauche est comparée au résultat de la sous-requête avec l'opérateur donné
- La ligne de gauche est retournée
  - si le résultat d'au moins une comparaison est vraie
- La ligne de gauche n'est pas retournée
  - si aucun résultat de la comparaison n'est vrai
  - si l'expression de gauche vaut NULL
  - si la sous-requête ramène un ensemble vide

La clause ANY, ou son synonyme SOME, permet de comparer l'expression de gauche à chaque ligne du résultat de la sous-requête en utilisant l'opérateur indiqué. Ainsi, la requête de l'exemple avec la clause IN aurait pu être écrite avec = ANY de la façon suivante :

```
SELECT *
  FROM stock
 WHERE contenant_id = ANY (SELECT id
```

```
FROM contenant
WHERE contenance
BETWEEN 0.3 AND 1.0);
```

4.4.8 CLAUSE ALL

```
expression operateur ALL (sous-requete)
```

- L'expression de gauche est comparée à tous les résultats de la sous-requête avec l'opérateur donné
- La ligne de gauche est retournée
  - si tous les résultats des comparaisons sont vrais
  - si la sous-requête retourne un ensemble vide
- La ligne de gauche n'est pas retournée
  - si au moins une comparaison est fausse
  - si au moins une comparaison est `NULL`

La clause `ALL` permet de comparer l'expression de gauche à chaque ligne du résultat de la sous-requête en utilisant l'opérateur de comparaison indiqué.

La ligne de la table de gauche sera retournée si toutes les comparaisons sont vraies ou si la sous-requête retourne un ensemble vide. En revanche, la ligne de la table de gauche sera exclue si au moins une comparaison est fausse ou si au moins une comparaison est `NULL`.

La requête d'exemple de la clause `NOT IN` aurait pu être écrite avec `<> ALL` de la façon suivante :

```
SELECT *
  FROM stock
 WHERE contenant_id <> ALL (SELECT id
                              FROM contenant
                             WHERE contenance < 2.0);
```

4.4.9 UTILISER UN ENSEMBLE

- La sous-requête retourne
  - plusieurs lignes
  - sur plusieurs colonnes
- Positionnée au niveau de la clause `FROM`

- Nommée avec un alias de table

La sous-requête peut être utilisée dans la clause `FROM` afin d'être utilisée comme une table dans la requête principale. La sous-requête devra obligatoirement être nommée avec un alias de table. Lorsqu'elles sont issues d'un calcul, les colonnes résultantes doivent également être nommées avec un alias de colonne afin d'éviter toute confusion ou comportement incohérent.

La requête suivante permet de déterminer le nombre moyen de bouteilles par années :

```
SELECT AVG(nombre_total_annee) AS moyenne
  FROM (SELECT annee, sum(nombre) AS nombre_total_annee
          FROM stock
         GROUP BY annee) stock_total_par_annee;
```

4.4.10 CLAUSE EXISTS

```
EXISTS (sous-requete)
```

- Intéressant avec une corrélation
- La clause `EXISTS` vérifie la présence ou l'absence de résultats
  - vrai si l'ensemble est non vide
  - faux si l'ensemble est vide

`EXISTS` présente peu d'intérêt sans corrélation entre la sous-requête et la requête principale.

Le prédicat `EXISTS` est en général plus performant que `IN`. Lorsqu'une requête utilisant `IN` ne peut pas être réécrite sous la forme d'une jointure, il est recommandé d'utiliser `EXISTS` en lieu et place de `IN`. Et à l'inverse, une clause `NOT IN` sera réécrite avec `NOT EXISTS`.

La requête suivante permet d'identifier les vins pour lesquels il y a au moins une bouteille en stock :

```
SELECT *
  FROM vin
 WHERE EXISTS (SELECT *
                 FROM stock
                WHERE vin_id = vin.id);
```

4.5 JOINTURES

- Produit cartésien

17.12

- Jointure interne
- Jointures externes
- Jointure ou sous-requête ?

Les jointures permettent d'écrire des requêtes qui impliquent plusieurs tables. Elles permettent de combiner les colonnes de plusieurs tables selon des critères particuliers, appelés conditions de jointures.

Les jointures permettent de tirer parti du modèle de données dans lequel les tables sont associées à l'aide de clés étrangères.

Bien qu'il soit possible de décrire une jointure interne sous la forme d'une requête `SELECT` portant sur deux tables dont la condition de jointure est décrite dans la clause `WHERE`, cette forme d'écriture n'est pas recommandée. En effet, les conditions de jointures se trouveront mélangées avec les clauses de filtrage, rendant ainsi la compréhension et la maintenance difficiles. Il arrive aussi que, noyé dans les autres conditions de filtrage, l'utilisateur oublie la configuration de jointure, ce qui aboutie à un produit cartésien, n'ayant rien à voir avec le résultat attendu, sans même parler de la lenteur de la requête.

Il est recommandé d'utiliser la syntaxe SQL:92 et d'exprimer les jointures à l'aide de la clause `JOIN`. D'ailleurs, cette syntaxe est la seule qui soit utilisable pour exprimer simplement et efficacement une jointure externe. Cette syntaxe facilite la compréhension de la requête mais facilite également le travail de l'optimiseur SQL qui peut déduire beaucoup plus rapidement les jointures qu'en analysant la clause `WHERE` pour déterminer les conditions de jointure et les tables auxquelles elles s'appliquent le cas échéant.

4.5.1 PRODUIT CARTÉSIEN

- Clause `CROSS JOIN`
- Réalise toutes les combinaisons entre les lignes d'une table et les lignes d'une autre
- À éviter dans la mesure du possible
  - peu de cas d'utilisation
  - peu performant

Le produit cartésien peut être exprimé avec la clause de jointure `CROSS JOIN` :

```sql
-- préparation du jeu de données
CREATE TABLE t1 (i1 integer, v1 integer);
CREATE TABLE t2 (i2 integer, v2 integer);
INSERT INTO t1 (i1, v1) VALUES (0, 0), (1, 1);
INSERT INTO t2 (i2, v2) VALUES (2, 2), (3, 3);
```

```
-- requête CROSS JOIN
SELECT * FROM t1 CROSS JOIN t2;
 i1 | v1 | i2 | v2
----+----+----+----
  0 |  0 |  2 |  2
  0 |  0 |  3 |  3
  1 |  1 |  2 |  2
  1 |  1 |  3 |  3
(4 rows)
```

Ou plus simplement, en listant les deux tables dans la clause `FROM` sans indiquer de condition de jointure :

```
SELECT * FROM t1, t2;
 i1 | v1 | i2 | v2
----+----+----+----
  0 |  0 |  2 |  2
  0 |  0 |  3 |  3
  1 |  1 |  2 |  2
  1 |  1 |  3 |  3
(4 rows)
```

Voici un autre exemple utilisant aussi un `NOT EXISTS` :

```
CREATE TABLE sondes (id_sonde int, nom_sonde text);
CREATE TABLE releves_horaires (
    id_sonde int,
    heure_releve timestamptz check
        (date_trunc('hour',heure_releve)=heure_releve),
    valeur numeric);

INSERT INTO sondes VALUES (1,'sonde 1'),
                          (2, 'sonde 2'),
                          (3, 'sonde 3');

INSERT INTO releves_horaires VALUES
  (1,'2013-01-01 12:00:00',10),
  (1,'2013-01-01 13:00:00',11),
  (1,'2013-01-01 14:00:00',12),
  (2,'2013-01-01 12:00:00',10),
  (2,'2013-01-01 13:00:00',12),
  (2,'2013-01-01 14:00:00',12),
  (3,'2013-01-01 12:00:00',10),
  (3,'2013-01-01 14:00:00',10);

-- quels sont les relevés manquants entre 12h et 14h ?

SELECT id_sonde,
```

```
       heures_releves
FROM sondes
CROSS JOIN generate_series('2013-01-01 12:00:00','2013-01-01 14:00:00',
    interval '1 hour') series(heures_releves)
WHERE NOT EXISTS
    (SELECT 1
     FROM releves_horaires
     WHERE releves_horaires.id_sonde=sondes.id_sonde
       AND releves_horaires.heure_releve=series.heures_releves);

 id_sonde |      heures_releves
----------+------------------------
        3 | 2013-01-01 13:00:00+01
(1 ligne)
```

4.5.2 JOINTURE INTERNE

- Clause `INNER JOIN`
  - meilleure lisibilité
  - facilite le travail de l'optimiseur
- Joint deux tables entre elles
  - Selon une condition de jointure

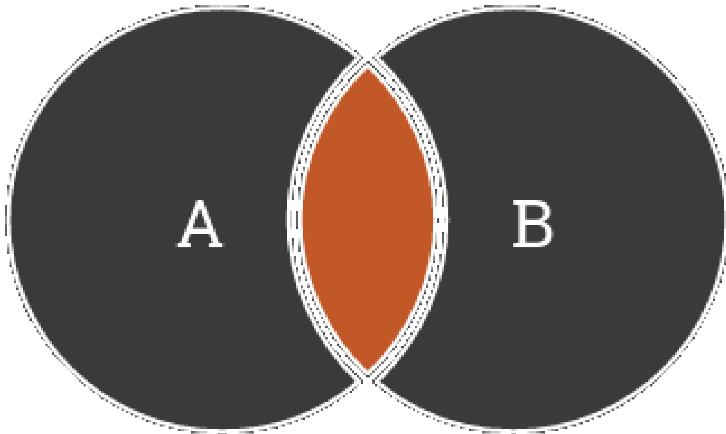


FIGURE 2: SCHÉMA DE JOINTURE INTERNE

Une jointure interne est considérée comme un produit cartésien accompagné d'une clause

de jointure pour ne conserver que les lignes qui répondent à la condition de jointure. Les SGBD réalisent néanmoins l'opération plus simplement.

La condition de jointure est généralement une égalité, ce qui permet d'associer entre elles les lignes de la table à gauche et de la table à droite dont les colonnes de condition de jointure sont égales.

La jointure interne est exprimée à travers la clause `INNER JOIN` ou plus simplement `JOIN`. En effet, si le type de jointure n'est pas spécifié, l'optimiseur considère la jointure comme étant une jointure interne.

4.5.3 SYNTAXE D'UNE JOINTURE INTERNE

- Condition de jointure par prédicats :

  ```
  table1 [INNER] JOIN table2 ON prédicat [...]
  ```

- Condition de jointure implicite par liste des colonnes impliquées :

  ```
  table1 [INNER] JOIN table2 USING (colonne [, ...])
  ```

- Liste des colonnes implicites :

  ```
  table1 NATURAL [INNER] JOIN table2
  ```

La clause `ON` permet d'écrire les conditions de jointures sous la forme de prédicats tels qu'on les retrouve dans une clause `WHERE`.

La clause `USING` permet de spécifier les colonnes sur lesquelles porte la jointure. Les tables jointes devront posséder toutes les colonnes sur lesquelles portent la jointure. La jointure sera réalisée en vérifiant l'égalité entre chaque colonne portant le même nom.

La clause `NATURAL` permet de réaliser la jointure entre deux tables en utilisant les colonnes qui portent le même nom sur les deux tables comme condition de jointure. La forme `NATURAL JOIN` est déconseillée car elle entraîne des comportements inattendus.

La requête suivante permet de joindre la table appellation avec la table `region` pour déterminer l'origine d'une appellation :

```
SELECT apl.libelle AS appellation, reg.libelle AS region
  FROM appellation apl
  JOIN region reg
    ON (apl.region_id = reg.id);
```

4.5.4 JOINTURE EXTERNE

- Jointure externe à gauche
  - ramène le résultat de la jointure interne
  - ramène l'ensemble de la table de gauche qui ne peut être joint avec la table de droite
  - les attributs de la table de droite sont alors `NULL`

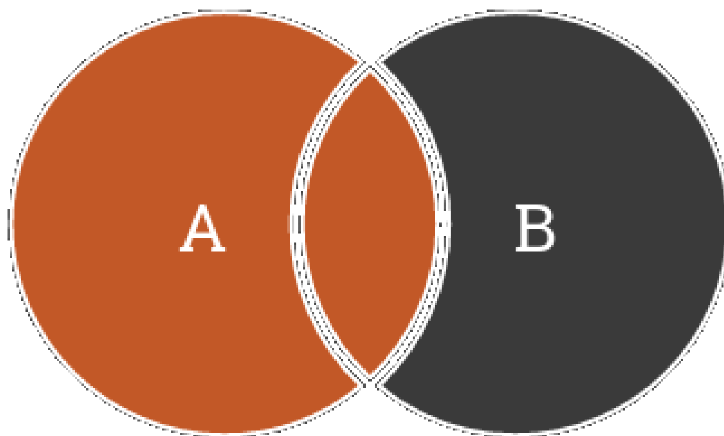


FIGURE 3: SCHÉMA DE JOINTURE EXTERNE GAUCHE

Il existe deux types de jointure externe : la jointure à gauche et la jointure à droite. Cela ne concerne que l'ordre de la jointure, le traitement en lui- même est identique.

4.5.5 JOINTURE EXTERNE - 2

- Jointure externe à droite
  - ramène le résultat de la jointure interne
  - ramène l'ensemble de la table de droite qui ne peut être joint avec la table de gauche
  - les attributs de la table de gauche sont alors `NULL`

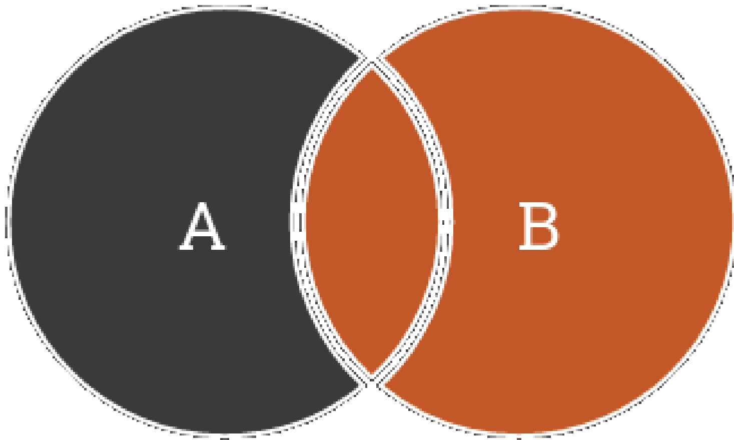


FIGURE 4: SCHÉMA DE JOINTURE EXTERNE DROITE

4.5.6 JOINTURE EXTERNE COMPLÈTE

- Ramène le résultat de la jointure interne
- Ramène l'ensemble de la table de gauche qui ne peut être joint avec la table de droite
  - les attributs de la table de droite sont alors `NULL`
- Ramène l'ensemble de la table de droite qui ne peut être joint avec la table de gauche
  - les attributs de la table de gauche sont alors `NULL`

4.5.7 SYNTAXE D'UNE JOINTURE EXTERNE À GAUCHE

- Condition de jointure par prédicats :

```
table1 LEFT [OUTER] JOIN table2 ON prédicat [...]
```

- Condition de jointure implicite par liste des colonnes impliquées :

```
table1 LEFT [OUTER] JOIN table2 USING (colonne [, ...])
```

- Liste des colonnes implicites :

```
table1 NATURAL LEFT [OUTER] JOIN table2
```

Il existe trois écritures différentes d'une jointure externe à gauche. La clause `NATURAL` permet de réaliser la jointure entre deux tables en utilisant les colonnes qui portent le

même nom sur les deux tables comme condition de jointure.

Les voici en exemple :

- par prédicat :

```
SELECT article.art_titre, auteur.aut_nom
  FROM article
  LEFT JOIN auteur
    ON (article.aut_id=auteur.aut_id);
```

- par liste de colonnes :

```
SELECT article.art_titre, auteur.aut_nom
  FROM article
  LEFT JOIN auteur
    USING (aut_id);
```

4.5.8 SYNTAXE D'UNE JOINTURE EXTERNE À DROITE

- Condition de jointure par prédicats :

```
table1 RIGHT [OUTER] JOIN table2 ON prédicat [...]
```

- Condition de jointure implicite par liste des colonnes impliquées :

```
table1 RIGHT [OUTER] JOIN table2 USING (colonne [, ...])
```

- Liste des colonnes implicites :

```
table1 NATURAL RIGHT [OUTER] JOIN table2
```

Les jointures à droite sont moins fréquentes mais elles restent utilisées.

4.5.9 SYNTAXE D'UNE JOINTURE EXTERNE COMPLÈTE

- Condition de jointure par prédicats :

```
table1 FULL OUTER JOIN table2 ON prédicat [...]
```

- Condition de jointure implicite par liste des colonnes impliquées :

```
table1 FULL OUTER JOIN table2 USING (colonne [, ...])
```

- Liste des colonnes implicites :

```
table1 NATURAL FULL OUTER JOIN table2
```

4.5.10 JOINTURE OU SOUS-REQUÊTE ?

- Jointures
  - algorithmes très efficaces
  - ne gèrent pas tous les cas
- Sous-requêtes
  - parfois peu performantes
  - répondent à des besoins non couverts par les jointures

Les sous-requêtes sont fréquemment utilisées mais elles sont moins performantes que les jointures. Ces dernières permettent d'utiliser des optimisations très efficaces.

4.6 EXPRESSIONS CASE

- Équivalent à l'instruction `switch` en C ou Java
- Emprunté au langage Ada
- Retourne une valeur en fonction du résultat de tests

`CASE` permet de tester différents cas. Il s'utilise de la façon suivante :

```
SELECT
  CASE WHEN col1=10 THEN 'dix'
       WHEN col1>10 THEN 'supérieur à 10'
       ELSE 'inférieur à 10'
  END AS test
FROM t1;
```

4.6.1 CASE SIMPLE

```
CASE expression
     WHEN valeur THEN expression
     WHEN valeur THEN expression
     (...)
     ELSE expression
 END
```

Il est possible de tester le résultat d'une expression avec `CASE`. Dans ce cas, chaque clause `WHEN` reprendra la valeur à laquelle on souhaite associé une expression particulière :

```
CASE nom_region
  WHEN 'Afrique' THEN 1
  WHEN 'Amérique' THEN 2
```

```
  WHEN 'Asie' THEN 3
  WHEN 'Europe' THEN 4
  ELSE 0
END
```

4.6.2 CASE SUR EXPRESSIONS

```
CASE WHEN expression THEN expression
     WHEN expression THEN expression
     (...)
     ELSE expression
 END
```

Une expression peut être évaluée pour chaque clause `WHEN`. Dans ce cas, l'expression `CASE` retourne la première expression qui est vraie. Si une autre peut satisfaire la suivante, elle ne sera pas évaluée.

Par exemple :

```
CASE WHEN salaire * prime < 1300 THEN salaire * prime
     WHEN salaire * prime < 3000 THEN salaire
     WHEN salaire * prime > 5000 THEN salaire * prime
END
```

4.6.3 SPÉCIFICITÉS DE CASE

- Comportement procédural
  - les expressions sont évaluées dans l'ordre d'apparition
- Transtypage
  - le type du retour de l'expression dépend du type de rang le plus élevé de toute l'expression
- Imbrication
  - des expressions `CASE` à l'intérieur d'autres expressions `CASE`
- Clause `ELSE`
  - recommandé

Il est possible de placer plusieurs clauses `WHEN`. Elles sont évaluées dans leur ordre d'apparition.

```
CASE nom_region
  WHEN 'Afrique' THEN 1
  WHEN 'Amérique' THEN 2
```

```
  /* l'expression suivante ne sera jamais évaluée */
  WHEN 'Afrique' THEN 5
  WHEN 'Asie' THEN 1
  WHEN 'Europe' THEN 3
  ELSE 0
END
```

Le type de données renvoyé par l'instruction `CASE` correspond au type indiqué par l'expression au niveau des `THEN` et du `ELSE`. Ce doit être le même type. Si les types de données ne correspondent pas, alors PostgreSQL retournera une erreur :

```
SELECT *,
  CASE nom_region
    WHEN 'Afrique' THEN 1
    WHEN 'Amérique' THEN 2
    WHEN 'Asie' THEN 1
    WHEN 'Europe' THEN 3
    ELSE 'inconnu'
  END
 FROM regions;
ERROR:  invalid input syntax for integer: "inconnu"
LIGNE 7 :      ELSE 'inconnu'
```

La clause `ELSE` n'est pas obligatoire mais fortement recommandé. En effet, si une expression `CASE` ne comporte pas de clause `ELSE`, alors la base de données ajoutera une clause `ELSE NULL` à l'expression.

Ainsi l'expression suivante :

```
CASE
  WHEN salaire < 1000 THEN 'bas'
  WHEN salaire > 3000 THEN 'haut'
END
```

Sera implicitement transformée de la façon suivante :

```
CASE
  WHEN salaire < 1000 THEN 'bas'
  WHEN salaire > 3000 THEN 'haut'
  ELSE NULL
END
```

4.7 OPÉRATEURS ENSEMBLISTES

- `UNION`
- `INTERSECT`

- `EXCEPT`

4.7.1 REGROUPEMENT DE DEUX ENSEMBLES

- Regroupement avec dédoublonnage :

  `requete_select1 UNION requete_select2`

- Regroupement sans dédoublonnage :

  `requete_select1 UNION ALL requete_select2`

L'opérateur ensembliste `UNION` permet de regrouper deux ensembles dans un même résultat.

Le dédoublonnage peut être particulièrement coûteux car il implique un tri des données.

Exemples

La requête suivante assemble les résultats de deux requêtes pour produire le résultat :

```
SELECT *
  FROM appellation
 WHERE region_id = 1
UNION ALL
SELECT *
  FROM appellation
 WHERE region_id = 3;
```

4.7.2 INTERSECTION DE DEUX ENSEMBLES

- Intersection de deux ensembles avec dédoublonnage :

  `requete_select1 INTERSECT requete_select2`

- Intersection de deux ensembles sans dédoublonnage :

  `requete_select1 INTERSECT ALL requete_select2`

L'opérateur ensembliste `INTERSECT` permet d'obtenir l'intersection du résultat de deux requêtes.

Le dédoublonnage peut être particulièrement coûteux car il implique un tri des données.

Exemples

L'exemple suivant n'a pas d'autre intérêt que de montrer le résultat de l'opérateur `INTERSECT` sur deux ensembles simples :

```
SELECT *
  FROM region
INTERSECT
SELECT *
  FROM region
 WHERE id = 3;

 id | libelle
----+---------
  3 | Alsace
```

4.7.3 DIFFÉRENCE ENTRE DEUX ENSEMBLES

- Différence entre deux ensembles avec dédoublonnage :

  ```
  requete_select1 EXCEPT requete_select2
  ```

- Différence entre deux ensembles sans dédoublonnage :

  ```
  requete_select1 EXCEPT ALL requete_select2
  ```

L'opérateur ensembliste `EXCEPT` est l'équivalent de l'opérateur `MINUS` d'Oracle. Il permet d'obtenir la différence entre deux ensembles : toutes les lignes présentes dans les deux ensembles sont exclues du résultat.

Le dédoublonnage peut être particulièrement coûteux car il implique un tri des données.

Exemples

L'exemple suivant n'a pas d'autre intérêt que de montrer le résultat de l'opérateur `EXCEPT` sur deux ensembles simples. La première requête retourne l'ensemble des lignes de la table `region` alors que la seconde requête retourne la ligne qui correspond au prédicat `id = 3`. Cette ligne est ensuite retirée du résultat car elle est présente dans les deux ensembles de gauche et de droite :

```
SELECT *
  FROM region
EXCEPT
SELECT *
  FROM region
 WHERE id = 3;

 id |          libelle
```

17.12

```
----+----------------------------
 11 | Cotes du Rhone
 12 | Provence produit a Cassis.
 10 | Beaujolais
 19 | Savoie
  7 | Languedoc-Roussillon
  4 | Loire
  6 | Provence
 16 | Est
  8 | Bordeaux
 14 | Lyonnais
 15 | Auvergne
  2 | Bourgogne
 17 | Forez
  9 | Vignoble du Sud-Ouest
 18 | Charente
 13 | Champagne
  5 | Jura
  1 | Provence et Corse
(18 rows)
```

4.8 CONCLUSION

- Possibilité d'écrire des requêtes complexes
- C'est là où PostgreSQL est le plus performant

Le standard SQL va bien plus loin que ce que les requêtes simplistes laissent penser. Utiliser des requêtes complexes permet de décharger l'application d'un travail conséquent et le développeur de coder quelque chose qui existe déjà. Cela aide aussi la base de données car il est plus simple d'optimiser une requête complexe qu'un grand nombre de requêtes simplistes.

4.8.1 QUESTIONS

N'hésitez pas, c'est le moment !

4.9 TRAVAUX PRATIQUES

4.9.1 ÉNONCÉS

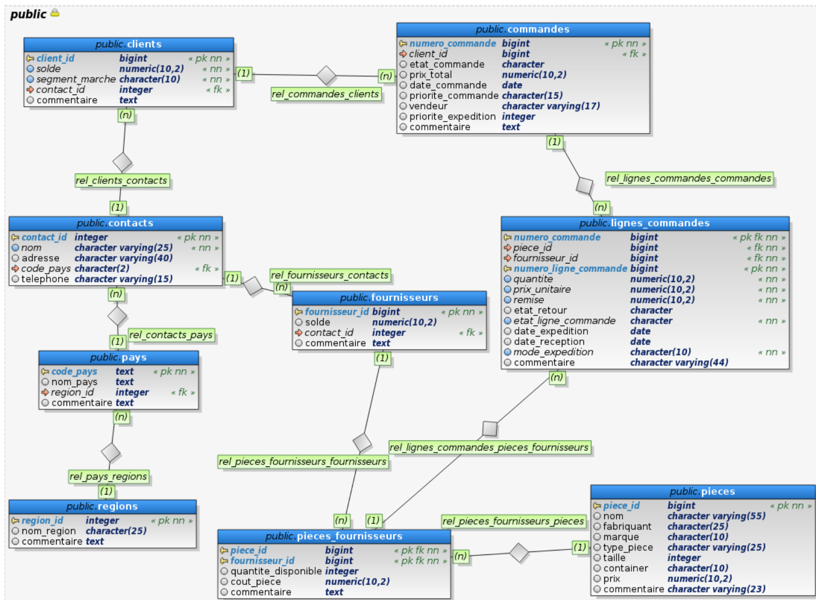


FIGURE 5: SCHÉMA BASE TPC

1. Affichez, par pays, le nombre de fournisseurs.

Sortie attendue :

```
          nom_pays             | nombre
-------------------------------+--------
 ARABIE SAOUDITE               |    425
 ARGENTINE                     |    416
 (...)
```

2. Affichez, par continent (`regions`), le nombre de fournisseurs.

Sortie attendue :

```
         nom_region           | nombre
---------------------------+--------
 Afrique                   |   1906
 Moyen-Orient              |   2113
 Europe                    |   2094
 Asie                      |   2002
 Amérique                  |   1885
```

3. Affichez le nombre de commandes trié selon le nombre de lignes de commandes au sein de chaque commande.

Sortie attendue :

```
 num | count
-----+-------
   1 | 13733
   2 | 27816
   3 | 27750
   4 | 27967
   5 | 27687
   6 | 27876
   7 | 13895
```

4. Pour les 30 premières commandes (selon la date de commande), affichez le prix total de la commande, en appliquant la remise accordée sur chaque article commandé. La sortie sera triée de la commande la plus chère à la commande la moins chère.

Sortie attendue :

```
 numero_commande | prix_total
-----------------+------------
               3 |  259600.00
              40 |  258959.00
               6 |  249072.00
              69 |  211330.00
              70 |  202101.00
               4 |  196132.00
 (...)
```

5. Affichez, par année, le total des ventes. La date de commande fait foi. La sortie sera triée par année.

Sortie attendue :

```
 annee |  total_vente
```

```
-------+---------------
  2005 | 3627568010.00
  2006 | 3630975501.00
  2007 | 3627112891.00
(...)
```

6. Pour toutes les commandes, calculez le temps moyen de livraison, depuis la date d'expédition. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

Sortie attendue :

```
temps_moyen_livraison
-----------------------
 8 jour(s)
```

7. Pour les 30 commandes les plus récentes (selon la date de commande), calculez le temps moyen de livraison de chaque commande, depuis la date de commande. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

Sortie attendue :

```
temps_moyen_livraison
-----------------------
 38 jour(s)
```

8. Déterminez le taux de retour des marchandises (l'état à `R` indiquant qu'une marchandise est retournée).

Sortie attendue :

```
taux_retour
-------------
       24.29
```

9. Déterminez le mode d'expédition qui est le plus rapide, en moyenne.

Sortie attendue :

```
 mode_expedition |       delai
-----------------+--------------------
 AIR             | 7.4711070230494535
```

10. Un bug applicatif est soupçonné, déterminez s'il existe des commandes dont la date de commande est postérieure à la date de livraison des articles.

Sortie attendue :

```
17.12
```

```
count
-------
      2
```

11. Écrivez une requête qui corrige les données erronés en positionnant la date de commande à la date de livraison la plus ancienne des marchandises. Vérifiez qu'elle soit correcte. Cette requête permet de corriger des calculs de statistiques sur les délais de livraison.

12. Écrivez une requête qui calcule le délai total maximal de livraison de la totalité d'une commande donnée, depuis la date de la commande.

Sortie attendue pour la commande n°1 :

```
delai_max
-----------
        102
```

13. Écrivez une requête pour déterminer les 10 commandes dont le délai de livraison, entre la date de commande et la date de réception, est le plus important, pour l'année 2011 uniquement.

Sortie attendue :

```
 numero_commande | delai
-----------------+-------
          413510 |   146
          123587 |   143
          224453 |   143
(...)
```

14. Un autre bug applicatif est détecté. Certaines commandes n'ont pas de lignes de commandes. Écrivez une requête pour les retrouver.

```
-[ RECORD 1 ]------------------------
numero_commande     | 91495
client_id           | 93528
etat_commande       | P
prix_total          |
date_commande       | 2007-07-07
priorite_commande   | 5-NOT SPECIFIED
vendeur             | Vendeur 000006761
priorite_expedition | 0
commentaire         | xxxxxxxxxxxxx
```

15. Écrivez une requête pour supprimer ces commandes. Vérifiez le travail avant de valider.

16. Écrivez une requête pour déterminer les 20 pièces qui ont eu le plus gros volume de commande.

Sortie attendue :

```
                    nom                    |  sum
-------------------------------------------+--------
 lemon black goldenrod seashell plum       | 461.00
 brown lavender dim white indian           | 408.00
 burlywood white chiffon blanched lemon    | 398.00
(...)
```

17. Affichez les fournisseurs des 20 pièces qui ont été le plus commandées sur l'année 2011.

Sortie attendue :

```
     nom      | piece_id
--------------+----------
 Supplier4395 |   191875
 Supplier4397 |   191875
 Supplier6916 |   191875
 Supplier9434 |   191875
 Supplier4164 |    11662
 Supplier6665 |    11662
(...)
```

18. Affichez le pays qui a connu, en nombre, le plus de commandes sur l'année 2011.

Sortie attendue :

```
    nom_pays     | count
-----------------+-------
 ARABIE SAOUDITE |  1074
```

19. Affichez pour les commandes passées en 2011, la liste des continents et la marge brute d'exploitation réalisée par continents, triés dans l'ordre décroissant.

Sortie attendue :

```
        nom_region         |   benefice
---------------------------+---------------
 Moyen-Orient              | 2008595508.00
```

17.12
(...)

20. Affichez le nom, le numéro de téléphone et le pays des fournisseurs qui ont un commentaire contenant le mot clé `Complaints` :

Sortie attendue :

```
 nom_fournisseur |    telephone    |            nom_pays
-----------------+-----------------+-------------------------------
 Supplier3873    | 10-741-199-8614 | IRAN, RÉPUBLIQUE ISLAMIQUE D'
(...)
```

21. Déterminez le top 10 des fournisseurs ayant eu le plus long délai de livraison, entre la date de commande et la date de réception, pour l'année 2011 uniquement.

Sortie attendue :

```
 fournisseur_id | nom_fournisseur | delai
----------------+-----------------+-------
           9414 | Supplier9414    |   146
(...)
```

4.9.2 SOLUTIONS

1. Affichez, par pays, le nombre de fournisseurs.

```sql
SELECT p.nom_pays, count(*)
FROM fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
GROUP BY p.nom_pays
;
```

2. Affichez, par continent, le nombre de fournisseurs.

```sql
SELECT r.nom_region, count(*)
FROM fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
   JOIN regions r ON p.region_id = r.region_id
GROUP BY r.nom_region
;
```

3. Affichez le nombre de commandes trié selon le nombre de lignes de commandes au sein de chaque commande.

```sql
SELECT
   nombre_lignes_commandes,
   count(*) AS nombre_total_commandes
FROM (
   /* cette sous-requête permet de compter le nombre de lignes de commande de
      chaque commande, et remonte cette information à la requête principale */
   SELECT count(numero_ligne_commande) AS nombre_lignes_commandes
   FROM lignes_commandes
   GROUP BY numero_commande
   ) comm_agg
/* la requête principale aggrège et trie les données sur ce nombre de lignes
   de commandes pour compter le nombre de commandes distinctes ayant le même
   nombre de lignes de commandes */
GROUP BY nombre_lignes_commandes
ORDER BY nombre_lignes_commandes DESC
;
```

4. Pour les 30 premières commandes (selon la date de commande), affichez le prix total de la commande, en appliquant la remise accordée sur chaque article commandé. La sortie sera triée de la commande la plus chère à la commande la moins chère.

```sql
SELECT c.numero_commande, sum(quantite * prix_unitaire - remise) prix_total
FROM (
   SELECT numero_commande, date_commande
   FROM commandes
   ORDER BY date_commande
   LIMIT 30
   ) c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
GROUP BY c.numero_commande
ORDER BY sum(quantite * prix_unitaire - remise) DESC
;
```

5. Affichez, par année, le total des ventes. La date de commande fait foi. La sortie sera triée par année.

```sql
SELECT
   extract ('year' FROM date_commande),
   sum(quantite * prix - remise) AS prix_total
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
   JOIN pieces p ON lc.piece_id = p.piece_id
GROUP BY extract ('year' FROM date_commande)
ORDER BY extract ('year' FROM date_commande)
;
```

6. Pour toutes les commandes, calculez le temps moyen de livraison, depuis la date d'expédition. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier

supérieur (fonction `ceil()`).

```
SELECT ceil(avg(date_reception - date_expedition))::text || ' jour(s)'
FROM lignes_commandes lc
;
```

7. Pour les 30 commandes les plus récentes (selon la date de commande), calculez le temps moyen de livraison de chaque commande, depuis la date de commande. Le temps de livraison moyen sera exprimé en jours, arrondi à l'entier supérieur (fonction `ceil()`).

```
SELECT count(*), ceil(avg(date_reception - date_commande))::text || ' jour(s)'
FROM (
   SELECT numero_commande, date_commande
   FROM commandes
   ORDER BY date_commande DESC
   LIMIT 30
   ) c
JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande ;
```

Note : la colonne `date_commande` de la table `commandes` n'a pas de contrainte `NOT NULL`, il est donc possible d'avoir des commandes sans date de commande renseignée. Dans ce cas, ces commandes vont remonter par défaut en haut de la liste, puisque la clause `ORDER BY` renvoie les `NULL` après les valeurs les plus grandes, et que l'on inverse le tri. Pour éviter que ces commandes ne faussent les résultats, il faut donc les exclure de la sous-requête, de la façon suivante :

```
   SELECT numero_commande, date_commande
   FROM commandes
   WHERE date_commande IS NOT NULL
   ORDER BY date_commande DESC
   LIMIT 30
```

8. Déterminez le taux de retour des marchandises (l'état à `R` indiquant qu'une marchandise est retournée).

```
SELECT
   round(
      sum(
         CASE etat_retour
            WHEN 'R' THEN 1.0
            ELSE 0.0
         END
      ) / count(*)::numeric * 100,
      2
   )::text || ' %' AS taux_retour
FROM lignes_commandes
;
```

À partir de la version 9.4 de PostgreSQL, la clause `FILTER` des fonctions d'aggrégation permet d'écrire une telle requête plus facilement :

```
SELECT
   round(
      count(*) FILTER (WHERE etat_retour = 'R') / count(*)::numeric * 100,
      2
   )::text || ' %' AS taux_retour
FROM lignes_commandes
;
```

9. Déterminez le mode d'expédition qui est le plus rapide, en moyenne.

```
SELECT mode_expedition, avg(date_reception - date_expedition)
FROM lignes_commandes lc
GROUP BY mode_expedition
ORDER BY avg(date_reception - date_expedition) ASC
LIMIT 1
;
```

10. Un bug applicatif est soupçonné, déterminez s'il existe des commandes dont la date de commande est postérieure à la date d'expédition des articles.

```
SELECT count(*)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
      AND c.date_commande > lc.date_expedition
;
```

11. Écrivez une requête qui corrige les données erronés en positionnant la date de commande à la date d'expédition la plus ancienne des marchandises. Vérifiez qu'elle soit correcte. Cette requête permet de corriger des calculs de statistiques sur les délais de livraison.

Afin de se protéger d'une erreur de manipulation, on ouvre une transaction :

```
BEGIN;

UPDATE commandes c_up
SET date_commande = (
   SELECT min(date_expedition)
   FROM commandes c
      JOIN lignes_commandes lc ON lc.numero_commande = c.numero_commande
         AND c.date_commande > lc.date_expedition
   WHERE c.numero_commande = c_up.numero_commande
   )
WHERE EXISTS (
   SELECT 1
   FROM commandes c2
```

```
      JOIN lignes_commandes lc ON lc.numero_commande = c2.numero_commande
         AND c2.date_commande > lc.date_expedition
      WHERE c_up.numero_commande  = c2.numero_commande
   GROUP BY 1
   )
;
```

La requête réalisée précédemment doit à présent retourner 0 :

```
SELECT count(*)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
      AND c.date_commande > lc.date_expedition
;
```

Si c'est le cas, on valide la transaction :

```
COMMIT;
```

Si ce n'est pas le cas, il doit y avoir une erreur dans la transaction, on l'annule :

```
ROLLBACK;
```

12. Écrivez une requête qui calcule le délai total maximal de livraison de la totalité d'une commande donnée, depuis la date de la commande.

Par exemple pour la commande dont le numéro de commande est le 1 :

```
SELECT max(date_reception - date_commande)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
WHERE c.numero_commande = 1
;
```

13. Écrivez une requête pour déterminer les 10 commandes dont le délai de livraison, entre la date de commande et la date de réception, est le plus important, pour l'année 2011 uniquement.

```
SELECT
   c.numero_commande,
   max(date_reception - date_commande)
FROM commandes c
   JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
    AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY c.numero_commande
ORDER BY max(date_reception - date_commande) DESC
LIMIT 10
;
```

14. Un autre bug applicatif est détecté. Certaines commandes n'ont pas de lignes de commandes. Écrivez une requête pour les retrouver.

Pour réaliser cette requête, il faut effectuer une jointure spéciale, nommée « Anti-jointure ». Il y a plusieurs façons d'écrire ce type de jointure. Les différentes méthodes sont données de la moins efficace à la plus efficace.

La version la moins performante est la suivante, avec `NOT IN` :

```
SELECT c.numero_commande
FROM commandes
WHERE numero_commande NOT IN (
   SELECT numero_commande
   FROM lignes_commandes
   )
;
```

Il n'y a aucune corrélation entre la requête principale et la sous-requête. PostgreSQL doit donc vérifier pour chaque ligne de `commandes` que `numero_commande` n'est pas présent dans l'ensemble retourné par la sous-requête. Il est préférable d'éviter cette syntaxe.

Autre écriture possible, avec `LEFT JOIN` :

```
SELECT c.numero_commande
FROM commandes c
   LEFT JOIN lignes_commandes lc ON c.numero_commande = lc.numero_commande
/* c'est le filtre suivant qui permet de ne conserver que les lignes de la
   table commandes qui n'ont PAS de correspondance avec la table
   numero_commandes */
WHERE lc.numero_commande IS NULL
;
```

Enfin, l'écriture généralement préférée, tant pour la lisibilité que pour les performances, avec `NOT EXISTS` :

```
SELECT c.numero_commande
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   )
;
```

15. Écrivez une requête pour supprimer ces commandes. Vérifiez le travail avant de valider.

Afin de se protéger d'une erreur de manipulation, on ouvre une transaction :

```
BEGIN;
```

17.12

La requête permettant de supprimer ces commandes est dérivée de la version NOT EXISTS de la requête ayant permis de trouver le problème :

```sql
DELETE
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   )
-- on peut renvoyer directement les numeros de commande qui ont été supprimés :
-- RETURNING numero_commande
;
```

Pour vérifier que le problème est corrigé :

```sql
SELECT count(*)
FROM commandes c
WHERE NOT EXISTS (
   SELECT 1
   FROM lignes_commandes lc
   WHERE lc.numero_commande = c.numero_commande
   )
;
```

Si la requête ci-dessus remonte 0, alors la transaction peut être validée :

```sql
COMMIT;
```

16. Écrivez une requête pour déterminer les 20 pièces qui ont eu le plus gros volume de commande.

```sql
SELECT p.nom,
   sum(quantite)
FROM pieces p
   JOIN lignes_commandes lc ON p.piece_id = lc.piece_id
GROUP BY p.nom
ORDER BY sum(quantite) DESC
LIMIT 20
;
```

17. Affichez les fournisseurs des 20 pièces qui ont été le plus commandées sur l'année 2011.

```sql
SELECT co.nom, max_p.piece_id, total_pieces
FROM (
   /* cette sous-requête est sensiblement la même que celle de l'exercice
      précédent, sauf que l'on remonte cette fois l'id de la piece plutôt
      que son nom pour pouvoir faire la jointure avec pieces_fournisseurs, et
      que l'on ajoute une jointure avec commandes pour pouvoir filtrer sur
```

```
    l'année 2011 */
  SELECT
    p.piece_id,
    sum(quantite) AS total_pieces
  FROM pieces p
    JOIN lignes_commandes lc ON p.piece_id = lc.piece_id
    JOIN commandes c ON c.numero_commande = lc.numero_commande
  WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
    AND to_date('31/12/2011', 'DD/MM/YYYY')
  GROUP BY p.piece_id
  ORDER BY sum(quantite) DESC
  LIMIT 20
  ) max_p
  /* il faut passer par la table de liens pieces_fournisseurs pour récupérer
    la liste des fournisseurs d'une piece */
  JOIN pieces_fournisseurs pf ON max_p.piece_id = pf.piece_id
  JOIN fournisseurs f ON f.fournisseur_id = pf.fournisseur_id
  -- la jointure avec la table contact permet d'afficher le nom du fournisseur
  JOIN contacts co ON f.contact_id = co.contact_id
;
```

18. Affichez le pays qui a connu, en nombre, le plus de commandes sur l'année 2011.

```
SELECT nom_pays,
  count(c.numero_commande)
FROM commandes c
  JOIN clients cl ON (c.client_id = cl.client_id)
  JOIN contacts co ON (cl.contact_id = co.contact_id)
  JOIN pays p ON (co.code_pays = p.code_pays)
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
  AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY p.nom_pays
ORDER BY count(c.numero_commande) DESC
LIMIT 1;
```

19. Affichez pour les commandes passées en 2011, la liste des régions et la marge brute d'exploitation réalisée par régions, triés dans l'ordre décroissant.

```
SELECT
  nom_region,
  round(sum(quantite * prix - remise) - sum(quantite * cout_piece), 2)
   AS marge_brute
FROM
  commandes c
  JOIN lignes_commandes lc ON lc.numero_commande = c.numero_commande
  /* il faut passer par la table de liens pieces_fournisseurs pour récupérer
    la liste des fournisseurs d'une piece - attention, la condition de
    jointure entre lignes_commandes et pieces_fournisseurs porte sur deux
```

```
      colonnes ! */
   JOIN pieces_fournisseurs pf ON lc.piece_id = pf.piece_id
      AND lc.fournisseur_id = pf.fournisseur_id
   JOIN pieces p ON p.piece_id = pf.piece_id
   JOIN fournisseurs f ON f.fournisseur_id = pf.fournisseur_id
   JOIN clients cl ON c.client_id = cl.client_id
   JOIN contacts co ON cl.contact_id = co.contact_id
   JOIN pays pa ON co.code_pays = pa.code_pays
   JOIN regions r ON r.region_id = pa.region_id
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY nom_region
ORDER BY sum(quantite * prix - remise) - sum(quantite * cout_piece) DESC
;
```

20. Affichez le nom, le numéro de téléphone et le pays des fournisseurs qui ont un commentaire contenant le mot clé `Complaints` :

```
SELECT
   nom,
   telephone,
   nom_pays
FROM
   fournisseurs f
   JOIN contacts c ON f.contact_id = c.contact_id
   JOIN pays p ON c.code_pays = p.code_pays
WHERE f.commentaire LIKE '%Complaints%'
;
```

21. Déterminez le top 10 des fournisseurs ayant eu le plus long délai de livraison, entre la date de commande et la date de réception, pour l'année 2011 uniquement.

```
SELECT
   f.fournisseur_id,
   co.nom,
   max(date_reception - date_commande)
FROM
   lignes_commandes lc
   JOIN commandes c ON c.numero_commande = lc.numero_commande
   JOIN pieces_fournisseurs pf ON lc.piece_id = pf.piece_id
      AND lc.fournisseur_id = pf.fournisseur_id
   JOIN fournisseurs f ON pf.fournisseur_id = f.fournisseur_id
   JOIN contacts co ON f.contact_id = co.contact_id
WHERE date_commande BETWEEN to_date('01/01/2011', 'DD/MM/YYYY')
   AND to_date('31/12/2011', 'DD/MM/YYYY')
GROUP BY f.fournisseur_id, co.nom
ORDER BY max(date_reception - date_commande) DESC
LIMIT 10
```

;

5 APPROFONDISSEMENT SQL

5.0.1 PRÉAMBULE

- Aller encore plus loin
  - tout en évitant le pire
- Appréhender de nouveaux objets
  - fonctions utiles
  - vues
- Utiliser des requêtes préparées

Notre tour du SQL va se terminer. Il nous reste encore à voir des objets spéciaux, souvent utilisés, comme les vues et les procédures stockées. Nous allons aussi nous intéresser aux requêtes préparées. Nous terminerons avec une liste de recommendations pour éviter une mauvaise utilisation du SQL.

5.0.2 MENU

- Fonctions de base
- Vues
- Requêtes préparées
- Index
- Ce qu'il ne faut pas faire

5.0.3 OBJECTIFS

- Utiliser des objets avancées
- Gagner en performance
- Éviter les pièges les plus fréquents

5.1 FONCTIONS DE BASE

- Transtypage

- Manipulation de chaines
- Manipulation de types numériques
- Manipulation de dates
- Génération de jeu de données

PostgreSQL propose un nombre conséquent de fonctions permettant de manipuler les différents types de données disponibles. Les étudier de façon exhaustive n'est pas l'objet de ce module. Néanmoins, le manuel de PostgreSQL établie une liste complète des fonctions disponibles dans le SGBD[8] .

5.1.1 TRANSTYPAGE

- Conversion d'un type de données vers un autre type de données
- `CAST (expression AS type)`
- `expression::type`

Les opérateurs de transtypages permettent de convertir une donnée d'un type particulier vers un autre type. La conversion échoue si les types de données sont incompatibles.

Exemples

Transtypage incorrect d'une chaîne de caractères vers un entier :

```
SELECT 'toto'::integer;
ERROR:  invalid input syntax for integer: "toto"
LINE 1: SELECT 'toto'::integer;
```

5.1.2 OPÉRATIONS SIMPLES SUR LES CHAÎNES

- Concaténation : `chaîne1 || chaîne2`
- Longueur de la chaîne : `char_length(chaîne)`
- Conversion en minuscules : `lower(chaîne)`
- Conversion en majuscules : `upper(chaîne)`

L'opérateur de concaténation permet de concaténer deux chaînes de caractères :

```
SELECT 'Bonjour' || ', Monde!';
-----------------
 Bonjour, Monde!
```

Il permet aussi de concaténer une chaîne de caractères avec d'autres type de données :

[8] http://docs.postgresql.fr/9.2/functions.html

17.12

```
SELECT 'Texte ' || 1::integer;
---------
 Texte 1
```

La fonction `char_length()` permet de connaître la longueur d'une chaîne de caractères :

```
SELECT char_length('Texte' || 1::integer);
-------------
           6
```

Les fonctions `lower` et `upper` permettent de convertir une chaîne respectivement en minuscule et en majuscule :

```
SELECT lower('Bonjour, Monde!');
-----------------
 bonjour, monde!

SELECT upper('Bonjour, Monde!');
-----------------
 BONJOUR, MONDE!
```

5.1.3 MANIPULATIONS DE CHAÎNES

- Extrait une chaîne à partir d'une autre : `substring(chaîne [from int] [for int])`
- Emplacement d'une sous-chaîne : `position(sous-chaîne in chaîne)`

La fonction `substring` permet d'extraire une chaîne de caractère à partir d'une chaîne en entrée. Il faut lui indiquer, en plus de la chaîne source, la position de départ, et la longeur de la sous-chaîne. Par exemple :

```
SELECT substring('Bonjour, Monde' from 5 for 4);
 substring
-----------
 our,
(1 row)
```

Notez que vous pouvez aussi utiliser un appel de fonction plus standard :

```
SELECT substring('Bonjour, Monde', 5, 4);
 substring
-----------
 our,
(1 row)
```

La fonction `position` indique la position d'une chaîne de caractère dans la chaîne indiquée. Par exemple :

```
SELECT position (',' in 'Bonjour, Monde');
 position
----------
        8
(1 row)
```

La combinaison des deux est intéressante :

```
SELECT version();
                            version
-------------------------------------------------------------------
 PostgreSQL 10.0 on x86_64-pc-linux-gnu, compiled by gcc (GCC) ...
(1 row)

SELECT substring(version() from 1 for position(' on' in version()));
    substring
-----------------
 PostgreSQL 10.0
```

5.1.4 MANIPULATION DE TYPES NUMÉRIQUES

- Opérations arithmétiques
- Manipulation de types numériques
- Génération de données

5.1.5 OPÉRATIONS ARITHMÉTIQUES

- Addition : `+`
- Soustraction : `-`
- Multiplication : `*`
- Division : `/`
- Reste (modulo) : `%`

L'ensemble des opérations arithmétiques disponibles sont documentées dans le manuel[9]

.

---

[9] http://docs.postgresql.fr/9.2/functions-math.html

5.1.6 FONCTIONS NUMÉRIQUES COURANTES

- Arrondi : `round(numeric)`
- Troncature : `trunc(numeric [, precision])`
- Entier le plus petit : `floor(numeric)`
- Entier le plus grand : `ceil(numeric)`

Ces fonctions sont décrites dans le manuel[10] .

5.1.7 GÉNÉRATION DE DONNÉES

- Générer une suite d'entiers : `generate_series(borne_debut, borne_fin, intervalle)`
- Générer un nombre aléatoire : `random()`

La fonction `generate_series(n, m)` est spécifique à PostgreSQL et permet de générer une suite d'entiers compris entre une borne de départ et une borne de fin, en suivant un certain intervalle :

```
SELECT generate_series(1, 4);
 generate_series
-----------------
               1
               2
               3
               4
(4 rows)
```

La déclinaison `generate_series(n, m, interval)` permet de spécifier un incrément pour chaque itération :

```
SELECT generate_series(1, 10, 4);
 generate_series
-----------------
               1
               5
               9
(3 rows)
```

Quant à la fonction `random()`, elle génère un nombre aléatoire, de type `numeric`, compris entre 0 et 1.

[10] http://docs.postgresql.fr/current/functions-math.html

```
SELECT random();
      random
-------------------
 0.381810061167926
(1 row)
```

Pour générer un entier compris entre 0 et 100, il suffit de réaliser la requête suivante :

```
SELECT round(100*random())::integer;
 round
-------
    74
(1 row)
```

Il est possible de contrôler la graine du générateur de nombres aléatoires en positionnant le paramètre de session `SEED` :

```
SET SEED = 0.123;
```

ou à l'aide de la fonction `setseed()` :

```
SELECT setseed(0.123);
```

La graine est un flottant compris entre -1 et 1.

Ces fonctions sont décrites dans le manuel de PostgreSQL[11] .

5.1.8 MANIPULATION DE DATES

- Obtenir la date et l'heure courante
- Manipuler des dates
- Opérations arithmétiques
- Formatage de données

5.1.9 DATE ET HEURE COURANTE

- Retourne la date courante : `current_date`
- Retourne l'heure courante : `current_time`
- Retourne la date et l'heure courante : `current_timestamp`

[11] http://docs.postgresql.fr/9.2/functions-math.html

17.12

Les fonctions current_date et current_time permettent d'obtenir respectivement la date courante et l'heure courante. La première fonction retourne le résultat sous la forme d'un type date et la seconde sous la forme d'un type time with time zone.

La fonction current_timestamp et son synonyme now() permettent d'obtenir la date et l'heure courante, le résultat étant de type timestamp with time zone.

Les fonctions current_date, current_time et current_timestamp n'ont pas besoin d'être invoquée avec les parenthèses ouvrantes et fermantes typiques de l'appel d'une fonction. En revanche, l'appel de la fonction now() requiert ces parenthèses.

```
SELECT current_date;
 current_date
--------------
 2017-10-04
(1 row)

SELECT current_time;
    current_time
--------------------
 16:32:47.386689+02
(1 row)

SELECT current_timestamp;
       current_timestamp
-------------------------------
 2017-10-04 16:32:50.314897+02
(1 row)

SELECT now();
              now
-------------------------------
 2017-10-04 16:32:53.684813+02
(1 row)
```

Il est possible d'utiliser ces variables comme valeur par défaut d'une colonne :

```
CREATE TABLE test (
  id SERIAL PRIMARY KEY,
  dateheure_creation TIMESTAMP DEFAULT current_timestamp,
  valeur VARCHAR);

INSERT INTO test (valeur) VALUES ('Bonjour, monde!');

SELECT * FROM test;
 id |      dateheure_creation      |      valeur
----+-----------------------------+-----------------
```

```
  1 | 2017-10-04 16:33:46.961088 | Bonjour, monde!
(1 row)
```

5.1.10 MANIPULATION DES DONNÉES

- Âge
  - Par rapport à la date courante : `age(timestamp)`
  - Par rapport à une date de référence : `age(timestamp, timestamp)`

La fonction `age(timestamp)` permet de déterminer l'âge de la date donnée en paramètre par rapport à la date courante. L'âge sera donné sous la forme d'un type `interval`.

La forme `age(timestamp, timestamp)` permet d'obtenir l'âge d'une date par rapport à une autre date, par exemple pour connaître l'âge de Gaston Lagaffe au 5 janvier 1997 :

```
SELECT age(date '1997-01-05', date '1957-02-28');
           age
-------------------------
 39 years 10 mons 5 days
```

5.1.11 TRONQUER ET EXTRAIRE

- Troncature d'une date : `date_trunc(text, timestamp)`
- Exemple : `date_trunc('month' from date_naissance)`
- Extrait une composante de la date : `extract(text, timestamp)`
- Exemple : `extract('year' from date_naissance)`

La fonction `date_trunc(text, timestamp)` permet de tronquer la date à une précision donnée. La précision est exprimée en anglais, et autorise les valeurs suivantes :

- microseconds
- milliseconds
- second
- minute
- hour
- day
- week
- month
- quarter
- year

- decade
- century
- millennium

La fonction `date_trunc()` peut agir sur une donnée de type `timestamp`, `date` ou `interval`. Par exemple, pour *arrondir* l'âge de Gaston Lagaffe de manière à ne représenter que le nombre d'année :

```
SELECT date_trunc('year',
    age(date '1997-01-05', date '1957-02-28')) AS age_lagaffe;
 age_lagaffe
-------------
 39 years
```

La fonction `extract(text from timestamp)` permet d'extraire uniquement une composante donnée d'une date, par exemple l'année. Elle retourne un type de données flottant `double precision`.

```
SELECT extract('year' from
    age(date '1997-01-05', date '1957-02-28')) AS age_lagaffe;
 age_lagaffe
-------------
          39
```

5.1.12 ARITHMÉTIQUE SUR LES DATES

- Opérations arithmétiques sur `timestamp`, `time` ou `date`
  - `date/time - date/time = interval`
  - `date/time + time = date/time`
  - `date/time + interval = date/time`
- Opérations arithmétiques sur `interval`
  - `interval * numeric  = interval`
  - `interval / numeric  = interval`
  - `interval + interval = interval`

La soustraction de deux types de données représentant des dates permet d'obtenir un intervalle qui représente le délai écoulé entre ces deux dates :

```
SELECT timestamp '2012-01-01 10:23:10' - date '0001-01-01' AS soustraction;
     soustraction
----------------------
 734502 days 10:23:10
```

L'addition entre deux types de données est plus restreinte. En effet, l'expression de gauche est obligatoirement de type `timestamp` ou `date` et l'expression de droite doit être obligatoirement de type `time`. Le résultat de l'addition permet d'obtenir une donnée de type `timestamp`, avec ou sans information sur le fuseau horaire selon que cette information soit présente ou non sur l'expression de gauche.

```
SELECT timestamp '2001-01-01 10:34:12' + time '23:56:13' AS addition;
      addition
---------------------
 2001-01-02 10:30:25

SELECT date '2001-01-01' + time '23:56:13' AS addition;
      addition
---------------------
 2001-01-01 23:56:13
```

L'addition d'une donnée datée avec une donnée de type `interval` permet d'obtenir un résultat du même type que l'expression de gauche :

```
SELECT timestamp with time zone '2001-01-01 10:34:12' +
    interval '1 day 1 hour' AS addition;
        addition
------------------------
 2001-01-02 11:34:12+01

SELECT date '2001-01-01'  + interval '1 day 1 hour' AS addition;
      addition
---------------------
 2001-01-02 01:00:00

SELECT time '10:34:24' + interval '1 day 1 hour' AS addition;
 addition
----------
 11:34:24
```

Une donnée de type `interval` peut subir des opérations arithmétiques. Le résultat sera de type `interval` :

```
SELECT interval '1 day 1 hour' * 2 AS multiplication;
 multiplication
-----------------
 2 days 02:00:00

SELECT interval '1 day 1 hour' / 2 AS division;
 division
----------
 12:30:00
```

17.12

```
SELECT interval '1 day 1 hour' + interval '2 hour' AS addition;
    addition
----------------
 1 day 03:00:00

SELECT interval '1 day 1 hour' - interval '2 hour' AS soustraction;
  soustraction
-----------------
 1 day -01:00:00
```

5.1.13 DATE VERS CHAÎNE

- Conversion d'une date en chaîne de caractères : `to_char(timestamp, text)`
- Exemple : `to_char(current_timestamp, 'DD/MM/YYYY HH24:MI:SS')`

La fonction `to_char()` permet de restituer une date selon un format donné :

```
SELECT current_timestamp;
       current_timestamp
-------------------------------
 2017-10-04 16:35:39.321341+02

SELECT to_char(current_timestamp, 'DD/MM/YYYY HH24:MI:SS');
       to_char
---------------------
 04/10/2017 16:35:43
```

5.1.14 CHAÎNE VERS DATE

- Conversion d'une chaîne de caractères en date : `to_date(text, text)` `to_date('05/12/2000', 'DD/MM/YYYY')`
- Conversion d'une chaîne de caractères en timestamp : `to_timestamp(text, text)` `to_timestamp('05/12/2000 12:00:00', 'DD/MM/YYYY HH24:MI:SS')`
- Paramètre `datestyle`

Quant à la fonction `to_date()`, elle permet de convertir une chaîne de caractères dans une donnée de type `date`. La fonction `to_timestamp()` permet de réaliser la même mais en donnée de type `timestamp`.

```
SELECT to_timestamp('04/12/2000 12:00:00', 'DD/MM/YYYY HH24:MI:SS');
to_timestamp
```

```
------------------------
 2000-12-04 12:00:00+01
```

Ces fonctions sont détaillées dans la section concernant les fonctions de formatage de données du manuel[12] .

Le paramètre `DateStyle` contrôle le format de saisie et de restitution des dates et heures. La documentation de ce paramètre permet de connaître les différentes valeurs possibles. Il reste néanmoins recommandé d'utiliser les fonctions de formatage de date qui permettent de rendre l'application indépendante de la configuration du SGBD.

La norme ISO impose le format de date "année/mois/jour". La norme SQL est plus permissive et permet de restituer une date au format "jour/mois/année" si DateStyle est égal à `'SQL, DMY'`.

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

SELECT current_timestamp;
              now
-------------------------------
 2017-10-04 16:36:38.189973+02

SET datestyle = 'SQL, DMY';

SELECT current_timestamp;
              now
---------------------------------
 04/10/2017 16:37:04.307034 CEST
```

5.1.15 GÉNÉRATION DE DONNÉES

- Générer une suite de timestamp : `generate_series(timestamp_debut, timestamp_fin, intervalle)`

La fonction `generate_series(date_debut, date_fin, interval)` permet de générer des séries de dates :

```
SELECT generate_series(date '2012-01-01',date '2012-12-31',interval '1 month');
    generate_series
------------------------
 2012-01-01 00:00:00+01
 2012-02-01 00:00:00+01
 2012-03-01 00:00:00+01
```

[12] http://docs.postgresql.fr/9.2/functions-formatting.html

17.12

```
 2012-04-01 00:00:00+02
 2012-05-01 00:00:00+02
 2012-06-01 00:00:00+02
 2012-07-01 00:00:00+02
 2012-08-01 00:00:00+02
 2012-09-01 00:00:00+02
 2012-10-01 00:00:00+02
 2012-11-01 00:00:00+01
 2012-12-01 00:00:00+01
(12 rows)
```

5.2 VUES

- Tables virtuelles
  - définies par une requête SELECT
  - définition stockée dans le catalogue de la base de données
- Objectifs
  - masquer la complexité d'une requête
  - masquer certaines données à l'utilisateur

Les vues sont des tables virtuelles qui permettent d'obtenir le résultat d'une requête `SELECT`. Sa définition est stockée dans le catalogue système de la base de données.

De cette façon, il est possible de créer une vue à destination de certains utilisateurs pour combler différents besoins :

- permettre d'interroger facilement une vue qui exécute une requête complexe, lourde à écrire et utilisée fréquemment,
- masquer certaines lignes ou certaines colonnes aux utilisateurs, pour amener un niveau de sécurité complémentaire,
- rendre les données plus intelligibles, en nommant mieux les colonnes d'une vue et/ou en simplifiant la structure de données.

En plus de cela, les vues permettent d'obtenir facilement des valeurs dérivées d'autres colonnes. Ces valeurs dérivées pourront alors être utilisées simplement en appelant la vue plutôt qu'en réécrivant systématiquement le calcul de dérivation à chaque requête qui le nécessite.

5.2.1 CRÉATION D'UNE VUE

- Une vue porte un nom au même titre qu'une table
  - elle sera nommée avec les mêmes règles
- Ordre de création d'une vue : `CREATE VIEW vue (colonne ...) AS SELECT ...`

Bien qu'une vue n'ait pas de représentation physique directe, elle est accédée au même titre qu'une table avec `SELECT` et dans certains cas avec `INSERT`, `UPDATE` et `DELETE`. La vue logique ne distingue pas les accès à une vue des accès à une table. De cette façon, une vue doit utiliser les mêmes conventions de nommage qu'une table.

Une vue est créée avec l'ordre SQL `CREATE VIEW` :

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW nom
    [ ( nom_colonne [, ...] ) ]
    [ WITH ( nom_option_vue [= valeur_option_vue] [, ... ] ) ]
    AS requete
```

Le mot clé `CREATE VIEW` permet de créer une vue. Si elle existe déjà, il est possible d'utiliser `CREATE OR REPLACE VIEW` qui aura pour effet de créer la vue si elle n'existe pas ou de remplacer la définition de la vue si elle existe déjà. Attention, les colonnes et les types de données retournés par la vue ne doivent pas changer.

La clause `nom` permet de nommer la vue. La clause `nom_colonne, ...`' permet lister explicitement les colonnes retournées par une vue, cette clause est optionnelle mais recommandée pour mieux documenter la vue.

La clause `requete` correspond simplement à la requête `SELECT` exécutée lorsqu'on accède à la vue.

Exemples

```
CREATE TABLE phone_data (person text, phone text, private boolean);

CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;

GRANT SELECT ON phone_number TO secretary;
```

5.2.2 LECTURE D'UNE VUE

- Une vue est lue comme une table
  - `SELECT * FROM vue;`

17.12

Une vue est lue de la même façon qu'une table. On utilisera donc l'ordre `SELECT` pour le faire. L'optimiseur de PostgreSQL remplacera l'appel à la vue par la définition de la vue pendant la phase de réécriture de la requête. Le plan d'exécution prendra alors compte des particularités de la vue pour optimiser les accès aux données.

Exemples

```
CREATE TABLE phone_data (person text, phone text, private boolean);

CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;

INSERT INTO phone_data (person, phone, private)
  VALUES ('Titi', '0123456789', true);

INSERT INTO phone_data (person, phone, private)
  VALUES ('Rominet', '0123456788', false);

SELECT person, phone FROM phone_number;
 person  |   phone
---------+------------
 Titi    |
 Rominet | 0123456788
(2 rows)
```

5.2.3 SÉCURISATION D'UNE VUE

- Sécuriser une vue
  - droits avec `GRANT` et `REVOKE`
- Utiliser les vues comme moyen de filtrer les lignes est dangereux
  - option `security_barrier`

Il est possible d'accorder (ou de révoquer) à un utilisateur les mêmes droits sur une vue que sur une table :

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

Le droit `SELECT` autorise un utilisateur à lire une table. Les droits `INSERT`, `UPDATE` et `DELETE` permettent de contrôler les accès en mise à jour à une vue.

Les droits TRUNCATE et REFERENCES n'ont pas d'utilité avec une vue. Ils ne sont tout simplement pas supportés car TRUNCATE n'agit que sur une table et une clé étrangère ne peut être liée d'une table qu'à une autre table.

Les vues sont parfois utilisées pour filtrer les lignes pouvant être lues par l'utilisateur. Cette protection peut être contournée si l'utilisateur a la possibilité de créer une fonction. À partir de PostgreSQL 9.2, l'option security_barrier permet d'éviter ce problème.

Exemples

```
postgres=# CREATE TABLE elements (id serial, contenu text, prive boolean);
CREATE TABLE
postgres=# INSERT INTO elements (contenu, prive)
VALUES ('a', false), ('b', false), ('c super prive', true),
       ('d', false), ('e prive aussi', true);
INSERT 0 5
postgres=# SELECT * FROM elements;
 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
(5 rows)
```

La table elements contient cinq lignes, trois considérés comme privés. Nous allons donc créer une vue ne permettant de ne voir que les lignes publiques.

```
postgres=# CREATE OR REPLACE VIEW elements_public AS
SELECT * FROM elements
WHERE CASE WHEN current_user='postgres' THEN TRUE ELSE NOT prive END;
CREATE VIEW
postgres=# SELECT * FROM elements_public;
 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
(5 rows)

postgres=# CREATE USER u1;
CREATE ROLE
postgres=# GRANT SELECT ON elements_public TO u1;
GRANT
```

17.12

```
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT * FROM elements;
ERROR:  permission denied for relation elements
postgres=> SELECT * FROM elements_public ;
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
(3 rows)
```

L'utilisateur `u1` n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Avec une simple fonction, cela peut changer :

```
postgres=> CREATE OR REPLACE FUNCTION abracadabra(integer, text, boolean)
RETURNS bool AS $$
BEGIN
RAISE NOTICE '% - % - %', $ 1, $ 2, $ 3;
RETURN true;
END$$
LANGUAGE plpgsql
COST 0.0000000000000000000001;
CREATE FUNCTION
postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);
NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  3 - c super prive - t
NOTICE:  4 - d - f
NOTICE:  5 - e prive aussi - t
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
(3 rows)
```

Que s'est-il passé ? pour comprendre, il suffit de regarder l'EXPLAIN de cette requête :

```
postgres=> EXPLAIN SELECT * FROM elements_public
WHERE abracadabra(id, contenu, prive);
                                                  QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on elements  (cost=0.00..28.15 rows=202 width=37)
   Filter: (abracadabra(id, contenu, prive) AND
```

```
            CASE WHEN ("current_user"() = 'u1'::name)
                 THEN (NOT prive) ELSE true END)
(2 rows)
```

La fonction `abracadrabra` a un coût si faible que PostgreSQL l'exécute avant le filtre de la vue. Du coup, la fonction voit toutes les lignes de la table.

Seul moyen d'échapper à cette optimisation du planificateur, utiliser l'option `security_barrier` en 9.2 :

```
postgres=> \c - postgres
You are now connected to database "postgres" as user "postgres".
postgres=# CREATE OR REPLACE VIEW elements_public WITH (security_barrier) AS
SELECT * FROM elements
WHERE CASE WHEN current_user='postgres' THEN true ELSE NOT prive END;

CREATE VIEW
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);
NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  4 - d - f
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
(3 rows)

postgres=> EXPLAIN SELECT * FROM elements_public WHERE
abracadabra(id, contenu, prive);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..34.20 rows=202 width=37)
   Filter: abracadabra(elements_public.id, elements_public.contenu,
           elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.15 rows=605 width=37)
         Filter: CASE WHEN ("current_user"() = 'u1'::name)
                 THEN (NOT prive) ELSE true END
(4 rows)
```

Voir aussi cet article de blog[13] .

---

[13] http://rhaas.blogspot.com/2012/03/security-barrier-views.html

5.2.4 MISE À JOUR DES VUES

- Trigger `INSTEAD OF`
- *Updatable view* (PostgreSQL 9.3)

La mise à jour des vues était impossible auparavant sans programmation complémentaire.

Depuis PostgreSQL 9.3, le moteur gère lui-même la possibilité de mettre à jour des vues simples. Les critères permettant de déterminer si une vue peut être mise à jour ou non sont assez simples à résumer : la vue doit reprendre la définition de la table avec éventuellement une clause `WHERE` pour restreindre les résultats. Attention néanmoins, la clause `WITH CHECK OPTION` n'est pas encore supportée par PostgreSQL. L'absence de support de cette clause permet à l'utilisateur d'insérer des données qui ne satisfont pas les critères de filtrage de la vue. Par exemple, il est possible d'insérer un numéro de téléphone privé alors que la vue ne permet pas d'afficher les numéros privés.

Pour gérer les cas plus complexes, PostgreSQL permet de créer des triggers `INSTEAD OF` sur des vues. Cette fonctionnalité est disponible depuis la version 9.1. Une alternative est d'utiliser le système de règles (`RULES`) mais cette pratique est peu recommandé du fait de la difficulté de débuggage et de la maintenance que cela entraine.

Un trigger `INSTEAD OF` permet de déclencher une fonction utilisateur lorsqu'une opération de mise à jour est déclenchée sur une vue. Le code de la fonction sera exécuté en lieu et place de la mise à jour.

Exemples

```
CREATE TABLE phone_data (person text, phone text, private boolean);


CREATE VIEW maj_phone_number (person, phone, private) AS
    SELECT person, phone, private
      FROM phone_data
     WHERE private = false;


CREATE VIEW phone_number (person, phone) AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
      FROM phone_data;


INSERT INTO phone_number VALUES ('Titi', '0123456789');
ERROR:  cannot insert into column "phone" of view "phone_number"
DETAIL:  View columns that are not columns of their base relation are not updatable.
HINT:  To make the view insertable, provide an unconditional ON INSERT
       DO INSTEAD rule or an INSTEAD OF INSERT trigger.
```

```
INSERT INTO maj_phone_number VALUES ('Titi', '0123456789', false);


CREATE OR REPLACE FUNCTION phone_number_insert_row()
  RETURNS TRIGGER
  LANGUAGE plpgsql
AS $function$
BEGIN
  INSERT INTO phone_data (person, phone, private)
    VALUES (NEW.person, NEW.phone, false);
  RETURN NEW;
END;
$function$;


CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON phone_number
    FOR EACH ROW
    EXECUTE PROCEDURE phone_number_insert_row();


INSERT INTO phone_number VALUES ('Rominet', '0123456788');


SELECT * FROM phone_number;
 person  |   phone
---------+------------
 Titi    | 0123456789
 Rominet | 0123456788
(2 rows)
```

5.2.5 MAUVAISES UTILISATIONS

- Prolifération des vues
  - créer une vue doit se justifier
  - ne pas créer une vue par table

La création d'une vue doit être pensée préalablement et doit se justifier du point de vue de l'application ou d'une règle métier. Toute vue créée doit être documentée, au moins en plaçant un commentaire sur la vue pour la documenter.

Bien qu'une vue n'ait pas de représentation physique, elle occupe malgré tout de l'espace

disque. En effet, le catalogue système comporte une entrée pour chaque vue créée, autant d'entrées qu'il y a de colonnes à la vue, etc. Trop de vues entraine donc malgré tout l'augmentation de la taille du catalogue système, donc une empreinte mémoire plus importante car ce catalogue reste en général systématiquement présent en cache.

5.3 REQUÊTES PRÉPARÉES

- Exécution en deux temps
  - préparation du plan d'exécution de la requête
  - exécution de la requête en utilisant le plan préparé
- Objectif :
  - éviter simplement les injections SQL
  - améliorer les performances

Les *requêtes préparées*, aussi appelées *requêtes paramétrées*, permettent de séparer la phase de préparation du plan d'exécution de la phase d'exécution. Le plan d'exécution qui est alors généré est générique car les paramètres de la requêtes sont inconnus à ce moment là.

L'exécution est ensuite commandée par l'application, en passant l'ensemble des valeurs des paramètres de la requête. De plus, ces paramètres sont passés de façon à éviter les injections SQL.

L'exécution peut être ensuite commandée plusieurs fois, sans avoir à préparer le plan d'exécution. Cela permet un gain important en terme de performances car l'étape d'analyse syntaxique et de recherche du plan d'exécution optimal n'est plus à faire.

L'utilisation de requêtes préparées peut toutefois être contre-performant si les sessions ne sont pas maintenues et les requêtes exécutées qu'une seule fois. En effet, l'étape de préparation oblige à un premier aller-retour entre l'application et la base de données et l'exécution oblige à un second aller- retour, ajoutant ainsi une surcharge qui peut devenir significative.

5.3.1 UTILISATION

- `PREPARE`, préparation du plan d'exécution d'une requête
- `EXECUTE`, passage des paramètres de la requête et exécution réelle
- L'implémentation dépend beaucoup du langage de programmation utilisé

 - le connecteur JDBC supporte les requêtes préparées
 - le connecteur PHP/PDO également

L'ordre PREPARE permet de préparer le plan d'exécution d'une requête. Le plan d'exécution prendra en compte le contexte courant de l'utilisateur au moment où la requête est préparée, et notamment le search_path. Tout changement ultérieur de ces variables ne sera pas pris en compte à l'exécution.

L'ordre EXECUTE permet de passer les paramètres de la requête et de l'exécuter.

La plupart des langages de programmation mettent à disposition des méthodes qui permettent d'employer les mécanismes de préparation de plans d'exécution directement. Les paramètres des requêtes seront alors transmis un à un à l'aide d'une méthode particulière.

Voici comment on prépare une requête :

```
PREPARE req1 (text) AS
    SELECT person, phone FROM phone_number WHERE person = $1;
```

Le test suivant montre le gain en performance qu'on peut attendre d'une requête préparée :

- préparation de la table :

```
CREATE TABLE t1 (c1 integer primary key, c2 text);
INSERT INTO t1 select i, md5(random()::text)
FROM generate_series(1, 1000000) AS i;
```

- préparation de deux scripts SQL, une pour les requêtes standards, l'autre pour les requêtes préparées :

```
$ for i in $(seq 1 100000); do
    echo "SELECT * FROM t1 WHERE c1=$i;";
done > requetes_std.sql
echo "PREPARE req AS SELECT * FROM t1 WHERE c1=\$1;" > requetes_prep.sql
for i in $(seq 1 100000); do echo "EXECUTE req($i);"; done >> requetes_prep.sql
```

- exécution du test (deux fois pour s'assurer que les temps d'exécution sont réalistes) :

```
$ time psql -f requetes_std.sql postgres >/dev/null

real    0m12.742s
user    0m2.633s
sys 0m0.771s
$ time psql -f requetes_std.sql postgres >/dev/null

real    0m12.781s
user    0m2.573s
sys 0m0.852s
```

17.12

```
$ time psql -f requetes_prep.sql postgres >/dev/null

real  0m10.186s
user  0m2.500s
sys 0m0.814s
$ time psql -f requetes_prep.sql postgres >/dev/null

real  0m10.131s
user  0m2.521s
sys 0m0.808s
```

Le gain est de 16% dans cet exemple. Il peut être bien plus important. En lisant 500000 lignes (et non pas 100000), on arrive à 25% de gain.

5.4 INDEXATION

- Comment gagner en performance
- Index
  - représentation valeur / pointeur
  - arbre de valeurs

Quand on cherche à récupérer une ou plusieurs valeurs sur une très grosse table, il serait dommage de devoir lire toutes les lignes de la table. Un peu comme lorsqu'on cherche un nom dans un annuaire, avoir une table des matières indiquant où à quelle page se trouvent les noms commençant par telle ou telle lettre permet de trouver plus rapide le nom recherché.

Un index est donc tout simplement ça : un moyen rapide de trouver l'information recherché. Il est construit généralement sous la forme d'un arbre car c'est la forme la plus aboutie pour trouver rapidement une valeur. Lorsque la valeur est trouvée, le pointeur vers la table est disponible et permet de ne lire que le ou les bloc(s) concernés de la table.

5.4.1 CRÉER UN INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ]
  ON table_name
  ( { column_name | ( expression ) }
    [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
  [ TABLESPACE tablespace_name ]
  [ WHERE predicate ]
```

Le plus simple moyen de créer un index est d'indiquer la ou les colonnes à indexer et le nom de la table. Voici un exemple :

```
CREATE INDEX i_c1_t1 on t1(c1);
```

Cependant, il existe beaucoup d'autres options. Il est possible d'indiquer l'ordre des valeurs (ascendant ou descendant), ainsi que l'emplacement des valeurs NULL.

5.4.2 LECTURE AVEC INDEX

- L'index permet une recherche plus rapide
- Il n'a besoin de lire que quelques blocs
  - et non pas la table entière

L'index contient des informations redondantes. Il contient uniquement les données disponibles dans la tête. Son intérêt n'est donc pas ce qu'il stocke mais la façon dont il le fait. Le stockage réalisé par l'index est optimisé pour la recherche.

L'exemple suivant montre la rapidité pour trouver un élément dans une table en contenant un million, avec un index et sans index :

```
CREATE TABLE t1 (c1 integer, c2 text);
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;

\timing
Timing is on.

select * from t1 where c1=500000;
   c1   |      c2
--------+--------------
 500000 | Ligne 500000
(1 row)

Time: 109.569 ms

select * from t1 where c1=500000;
   c1   |      c2
--------+--------------
 500000 | Ligne 500000
(1 row)

Time: 109.386 ms

CREATE INDEX ON t1(c1);
```

```
SELECT * FROM t1 WHERE c1=500000;
   c1    |       c2
---------+---------------
 500000 | Ligne 500000
(1 row)

Time: 0.675 ms
```

Donc il faut 100 millisecondes pour trouver une ligne parmi un million sans index, et 0,7 millisecondes avec un index. Le gain est appréciable.

5.4.3 INSERTION AVEC INDEX

- Accélère la recherche
  - mais ralentit les écritures
- Une écriture dans la table doit mettre à jour l'index
- Faire attention à ne pas créer des index inutiles

Voici un exemple montrant la différence de durée d'exécution d'une insertion d'un million de lignes sur une table sans index, puis avec un index :

```
DROP TABLE t1;
CREATE TABLE t1 (c1 integer, c2 text);
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
Time: 2761.236 ms

DROP TABLE t1;
CREATE TABLE t1 (c1 integer, c2 text);
CREATE INDEX ON t1(c1);
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
Time: 5401.505 ms
```

On passe ici du simple au double. Il est donc essentiel pour les performances de faire attention aux index créés. Oui, ils permettent de gagner en performances lors des lectures mais les écritures sont forcément ralenties. La création d'un index doit se faire en ayant pris ceci en considération.

5.4.4 TRI AVEC UN INDEX

- Un index ne sert pas qu'à la recherche
- Les données étant triées, il sert aussi à accélérer les tris

Voici un exemple montrant la différence de durée d'exécution d'un tri sur un million de lignes sur une table sans index, puis avec un index :

```
\o /dev/null
DROP TABLE t1;
CREATE TABLE t1 (c1 integer, c2 text);
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
\timing

SELECT * FROM t1 ORDER BY c1;
Time: 618.203 ms

CREATE INDEX ON t1(c1);
SELECT * FROM t1 ORDER BY c1;
Time: 359.369 ms
```

Dans cet exemple, le tri est deux fois plus rapide avec un index. Il est donc tout à fait concevable d'ajouter un index uniquement pour satisfaire un tri.

5.4.5 INDEX PARTIEL

- N'indexer qu'une partie des données
- Requête typique : `SELECT * FROM taches WHERE fait IS false ORDER BY date_rendu;`
- Index typique : `CREATE INDEX ON taches(date_rendu) WHERE fait IS false;`

L'index indiqué ici permet de réaliser le filtre et le tri en un seul parcours d'index. Il aidera fortement cette requête. En revanche, il y a peu de chances que cet index soit utile pour d'autres requêtes.

L'autre intérêt de cet index concerne sa volumétrie sur disque. Sur cette requête typique, il clair que, plus le temps passe, et plus il y aura de tâches réalisées. Du coup, même si la table grossit rapidement, l'index devrait rester à la même volumétrie. En tout cas, il grossira beaucoup moins que la table.

5.4.6 INDEX FONCTIONNEL

- Indexer le résultat d'une expression
- Fonctions immuables seulement
- Requête typique : `SELECT * FROM employes WHERE upper(nom)='DUPONT';`

17.12

- Index typique : `CREATE INDEX ON employes(upper(nom));`

L'idée de ce type d'index est de satisfaire les requêtes qui font une recherche sur le résultat d'un calcul ou d'une fonction.

Dans la table suivante :

```
CREATE TABLE employe(id serial, nom text, prenom text);
```

si on exécute la requête de recherche sur le nom, comme :

```
SELECT * FROM employes WHERE nom ILIKE 'DUPONT';
```

ou comme :

```
SELECT * FROM employes WHERE upper(nom)='DUPONT';
```

PostgreSQL doit faire un parcours séquentiel car il ne peut pas utiliser un index sur nom. Il faut utiliser la deuxième variante de la recherche et créer l'index, non pas sur la colonne nom, mais sur le résultat de la fonction `upper` sur la colonne nom.

5.5 CE QU'IL NE FAUT PAS FAIRE

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

Cette partie présente différents problèmes fréquemment rencontrés et leurs solutions.

5.5.1 ABSENCE DE CONTRAINTES

- Concerne surtout les clés étrangères
- Parfois (souvent?) ignorées pour diverses raisons :
  - performances
  - flexibilité du modèle de données
  - compatibilité avec d'autres SGBD
  - héritage de MySQL/MyISAM
  - commodité de développement
- Conséquences
  - problèmes d'intégrité des données
  - procédures de vérification de cohérence des données

Les contraintes d'intégrités et notamment les clés étrangères sont parfois absentes des modèles de données. Les problématiques de performances et de flexibilité sont souvent mises en avant alors que les contraintes sont justement une aide pour l'optimisation de requêtes par le planificateur et empêche des incohérences dans les données.

Bien que cela peut sembler être une bonne idée en début de projet, l'absence de ces contraintes va poser des problèmes d'intégrité des données au fur et à mesure de l'évolution des besoins et du code. Les données seront très certainement manipulées de façon différentes par la suite, par différentes applications, entraînant ainsi de réels problèmes d'intégrité des données. Mais surtout, du point de vue applicatif, il est très difficile, si ce n'est impossible, d'empêcher deux écritures concurrentes qui mèneraient à une incohérence (cas d'une *Race Condition*). 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, voir impossible à faire dans certains cas sans passer par la base de donnée elle même (multiple serveurs applicatifs accédant à la même base de donnée).

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

L'absence de clés étrangères et plus généralement de contraintes d'intégrité entraîne dans le temps une charge de travail supplémentaire. Il est bien préférable de mieux concevoir le modèle de données dès le début avec les bonnes contraintes d'intégrité.

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 alimenter la table mère. Des identifiants de clés étrangères de la table fille sont absents de la table mère, entrainant 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

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);
```

```sql
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 :

```sql
BEGIN TRANSACTION;
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;
```

5.5.2 STOCKAGE EAV

- Entité-Attribut-Valeur
- Pourquoi
  - 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
- Solutions
  - revenir sur la conception du modèle de données
  - utiliser un type de données plus adapté (`hstore`, `jsonb`)

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 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 n'y a de types de données à représentés. Ces colonnes ne permettront pas d'utiliser des contraintes `CHECK` pour garantir la cohérence des valeurs stockées avec ce qui est attendu, car les attributs peuvent stocker n'importe quelle donnée.

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

Des problèmes de performances vont 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, au contraire d'une table, jointe éventuellement à d'autres

tables, où les accès par index permettent de miniser les entrées-sorties.

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, il vaut mieux utiliser un type de données de PostgreSQL qui est approprié, comme `hstore` ou `jsonb` qui permet de stocker des données sous la forme `clé->valeur`. Le type de données `hstore` existe depuis longtemps dans PostgreSQL et a pour principal défaut de ne pouvoir représenter que des chaînes de caractères. Le type `jsonb` est une nouveautés de PostgreSQL 9.4 et permet de représenter des documents JSON et en les stockant dans un format interne optimisé et indexable. Ces types de données peuvent être indexés pour garantir de bons temps de réponses pour les requêtes qui nécessitent des recherches sur certaines clés ou certaines valeurs.

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

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

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

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

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

SELECT id, attributs->'nom' AS nom FROM personnes;
 id |     nom
----+-------------
  1 | Prunelle
  2 | Lagaffe
```

```
  3 | DeMaesmaker
(3 rows)
```

5.5.3 ATTRIBUTS MULTI-COLONNES

- Pourquoi
  - stocker plusieurs attributs pour une même ligne
  - exemple : les différents numéros de téléphone d'une personne
- Pratique courante
  - ajouter plusieurs colonnes portant le même nom
- Conséquences
  - et s'il faut rajouter encore une colonne ?
  - maîtrise de l'unicité des valeurs
  - requêtes complexes à maintenir
- Solutions
  - créer une table de dépendance
  - utiliser 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 plusieurs 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.

17.12

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'accèdera 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'id 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}
(1 ligne)

-- 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
(2 lignes)
```

5.5.4 CHOIX DES TYPES DE DONNÉES

- Objectif
  - représenter des valeurs décimales
- Pratique courante
  - utiliser le type `float` ou `double`
- Problèmes :
  - les types flottants ne stockent qu'une approximation de la valeur représentée
  - les erreurs d'arrondis se multiplient
  - les requêtes produisent des résultats faux
- Solutions
  - privilégier le type `numeric(x, y)` 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. Néanmoins, les types flottants sont peu précis, notamment pour les applications financières où une erreur d'arrondi n'est pas envisageable.

17.12

Exemple montrant une perte de précision dans les calculs :

```
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
(1 row)
```

L'utilisation du type `numeric` permet d'éviter la perte de précision :

```
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
(1 row)
```

5.5.5 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.

5.5.6 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.

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. 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.

Exemples

Exemple de modification du schéma pouvant entrainer 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);
                                      ^

SELECT * FROM insere ;
 id | col2 | col1
----+------+------
  1 |   10 | XX
```

5.5.7 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
- Vite ingérable
  - 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
  - Ne pas la patcher !

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,
```


DALIBO

```
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
```

```sql
 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
```

```sql
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
```

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

5.5.8 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
- Solution
  - utiliser 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, depuis la version 9.2, le module pg_trgm permet d'optimiser ces recherches à l'aide d'un index GiST ou GIN.

Mais cette solution n'offre pas la même souplesse que la recherche plein texte, en anglais Full Text Search, de PostgreSQL.

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)
```

5.6 CONCLUSION

- Des objets de plus en plus complexes
  - mais performants
  - et remplissant un besoin

- Des conseils
  - pour améliorer les performances
  - et pour éviter les pièges

Nous avons vu dans ce module de nouveaux objets SQL comme les procédures stockées livrées par défaut avec PostgreSQL ainsi que les vues. Nous avons vu aussi les requêtes préparées, mécanisme sensé fournir un surplus de performances si vous avez de nombreuses requêtes à exécuter qui suivent un même motif.

Nous avons fini avec des conseils, principalement pour éviter les pièges habituels mais aussi pour améliorer les performances.

5.6.1 QUESTIONS

N'hésitez pas, c'est le moment !

5.7 TRAVAUX PRATIQUES

5.7.1 ÉNONCÉS

1. Ajouter une adresse mail à chaque contact avec la concaténation du nom avec le texte '@dalibo.com'.
2. Concaténer nom et adresse mail des contacts français sous la forme 'nom '.
3. Même demande mais avec le nom en majuscule et l'adresse mail en minuscule.
4. Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`. Mettre à jour la colonne prix_total des commandes avec les informations des lignes de la table lignes_commandes.
5. Récupérer le montant total des commandes par mois pour l'année 2010. Les montants seront arrondis à deux décimales.
6. Supprimer les commandes de mai 2010.
7. Ré-exécuter la requête trouvée au point 5.
8. Qu'observez-vous ?

9. Corriger le problème rencontré.
10. Créer une vue calculant le prix total de chaque commande.
11. Réécrire la requête du point 5 pour utiliser la vue créée au point 10.

5.7.2 SOLUTIONS

1. Ajouter une adresse mail à chaque contact avec la concaténation du nom avec le texte '@dalibo.com'.

```
BEGIN ;

UPDATE contacts
SET email = nom||'@dalibo.com'
;

COMMIT ;
```

Note : pour éviter de mettre à jour les contacts ayant déjà une adresse mail, il suffit d'ajouter une clause WHERE :

```
UPDATE contacts
SET email = nom||'@dalibo.com'
WHERE email IS NULL
;
```

2. Concaténer nom et adresse mail des contacts français sous la forme 'nom '.

```
SELECT nom||' <'||email||'>'
FROM contacts
;
```

3. Même demande mais avec le nom en majuscule et l'adresse mail en minuscule.

```
SELECT upper(nom)||' <'||lower(email)||'>'
FROM contacts
;
```

4. 4. Ajouter la colonne `prix_total` de type `numeric(10,2)` à la table `commandes`. Mettre à jour la colonne prix_total des commandes avec les informations des lignes de la table lignes_commandes.

```
ALTER TABLE commandes ADD COLUMN prix_total numeric(10,2);
BEGIN ;

UPDATE commandes c
SET prix_total= (
   /* cette sous-requête fait une jointure entre lignes_commandes et la table
      commandes à mettre à jour pour calculer le prix par commande */
```

```
    SELECT SUM(quantite * prix_unitaire - remise)
    FROM lignes_commandes lc
    WHERE lc.numero_commande=c.numero_commande
)
-- on peut récupérer le détail de la mise à jour directement dans la requête :
-- RETURNING numero_commande, prix_total
;

COMMIT ;
```

Une autre variante de cette requête serait :

```
UPDATE commandes c SET prix_total=prix_calc
FROM (
    SELECT numero_commande, SUM(quantite * prix_unitaire - remise) AS prix_calc
        FROM lignes_commandes
       GROUP BY numero_commande
) as prix_detail
WHERE prix_detail.numero_commande = c.numero_commande
```

Bien que cette dernière variante soit moins lisible, elle est bien plus rapide sur un gros volume de données.

5. Récupérer le montant total des commandes par mois pour l'année 2010. Les montants seront arrondis à deux décimales.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(prix_total),2) AS montant_total
FROM commandes
WHERE date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1
;
```

Attention, il n'y a pas de contrainte `NOT NULL` sur le champ `date_commande`, donc s'il existe des commandes sans date de commande, celles-ci seront agrégées à part des autres, puisque `extract()` renverra `NULL` pour ces lignes.

6. Supprimer les commandes de mai 2010.

```
BEGIN;

/* en raison de la présence de clés étrangères, il faut en premier leur
   supprimer les lignes de la table lignes_commandes correspondant aux
   commandes à supprimer */
DELETE
FROM lignes_commandes
WHERE numero_commande IN (
```

```
    SELECT numero_commande
    FROM commandes
    WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
       AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
    )
;

-- ensuite seulement on peut supprimer les commandes
DELETE
FROM commandes
WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
;

COMMIT ;
```

Le problème de l'approche précédente est d'effectuer l'opération en deux temps. Il est possible de réaliser la totalité des suppressions dans les deux tables `lignes_commandes` et `commandes` en une seule requête en utilisant une *CTE* :

```
WITH del_lc AS (
   /* ici on déclare la CTE qui va se charger de supprimer les lignes dans la
      table lignes_commandes et retourner les numeros de commande supprimés */
   DELETE
   FROM lignes_commandes
   WHERE numero_commande IN (
      SELECT numero_commande
      FROM commandes
      WHERE date_commande >= to_date('01/05/2010', 'DD/MM/YYYY')
         AND date_commande < to_date('01/06/2010', 'DD/MM/YYYY')
      )
   RETURNING numero_commande
)
/* requête principale, qui supprime les commandes dont les numéros
   correspondent aux numéros de commandes remontés par la CTE */
DELETE
FROM commandes c
WHERE EXISTS (
   SELECT 1
   FROM del_lc
   WHERE del_lc.numero_commande = c.numero_commande
   )
;
```

7. Ré-exécuter la requête trouvée au point 5.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(prix_total),2) AS montant_total
```

```
FROM commandes
GROUP BY 1
ORDER BY 1
;
```

8. Qu'observez-vous ?

La ligne correspondant au mois de mai a disparu.

9. Corriger le problème rencontré.

```
SELECT numero_mois, round(coalesce(sum(prix_total), 0.0),2) AS montant_total
  /* la fonction generate_series permet de générer une pseudo-table d'une
     colonne contenant les chiffres de 1 à 12 */
FROM generate_series(1, 12) AS numero_mois
  /* le LEFT JOIN entre cette pseudo-table et la table commandes permet de
     s'assurer que même si aucune commande n'a eu lieu sur un mois, la ligne
     correspondante sera tout de même présente */
LEFT JOIN commandes ON extract('month' from date_commande) = numero_mois
      AND date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
      AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1
;
```

Notez l'utilisation de la fonction `coalesce()` dans le `SELECT`, afin d'affecter la valeur `0.0` aux lignes « ajoutées » par le `LEFT JOIN` qui n'ont par défaut aucune valeur (`NULL`).

10. Créer une vue calculant le prix total de chaque commande.

```
CREATE VIEW commande_montant AS
SELECT
   numero_commande,
   sum(quantite * prix_unitaire - remise) AS total_commande
FROM lignes_commandes
GROUP BY numero_commande
;
```

11. Réécrire la requête du point 5 pour utiliser la vue créée au point 10.

```
SELECT extract('month' from date_commande) AS numero_mois,
    round(sum(total_commande),2) AS montant_total
FROM commandes c
JOIN commande_montant cm ON cm.numero_commande = c.numero_commande
WHERE date_commande >= to_date('01/01/2010', 'DD/MM/YYYY')
   AND date_commande < to_date('01/01/2011', 'DD/MM/YYYY')
GROUP BY 1
ORDER BY 1
;
```

6 COMPRENDRE EXPLAIN

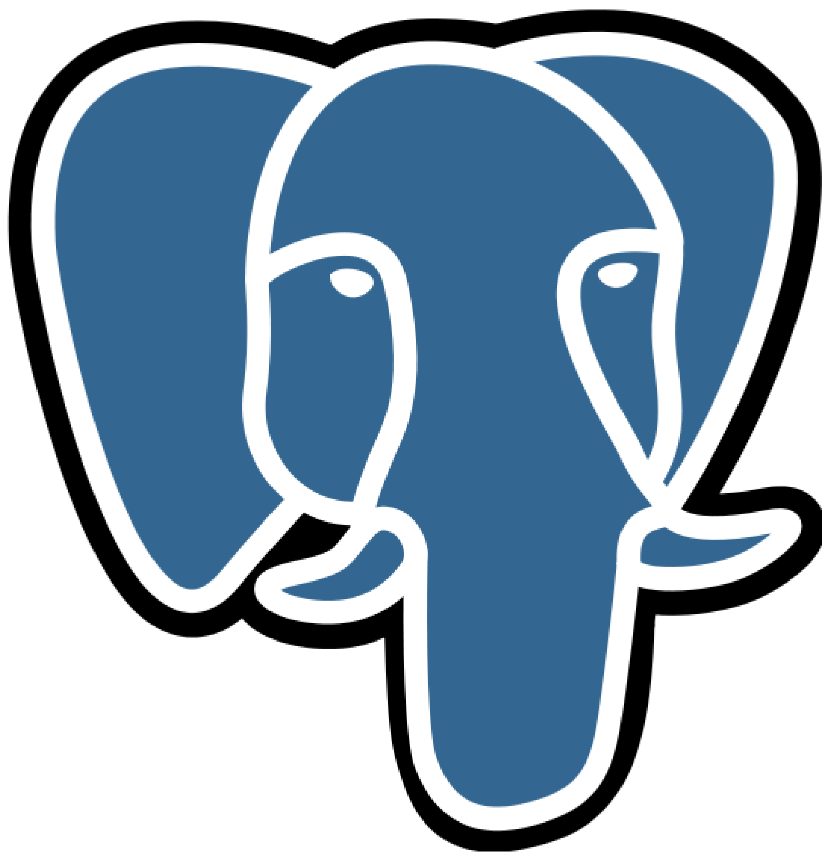


FIGURE 6: POSTGRESQL

6.1 INTRODUCTION

- Le matériel, le système et la configuration sont importants pour les performances
- Mais il est aussi essentiel de se préoccuper des requêtes et de leurs performances

Face à un problème de performances, l'administrateur se retrouve assez rapidement face à une (ou plusieurs) requête(s). Une requête en soi représente très peu d'informations.

Suivant la requête, des dizaines de plans peuvent être sélectionnés pour l'exécuter. Il est donc nécessaire de pouvoir trouver le plan d'exécution et de comprendre ce plan. Cela permet de mieux appréhender la requête et de mieux comprendre les pistes envisageables pour la corriger.

6.1.1 AU MENU

- Exécution globale d'une requête
- Planificateur : utilité, statistiques et configuration
- EXPLAIN
- Nœuds d'un plan
- Outils

Avant de détailler le fonctionnement du planificateur, nous allons regarder la façon dont une requête s'exécute globalement. Ensuite, nous aborderons le planificateur : en quoi est-il utile, comment fonctionne-t-il, et comment le configurer. Nous verrons aussi l'ensemble des opérations utilisables par le planificateur. Enfin, nous expliquerons comment utiliser `EXPLAIN` ainsi que les outils essentiels pour faciliter la compréhension d'un plan de requête.

Tous les exemples proposés ici viennent d'une version 9.1.

6.2 EXÉCUTION GLOBALE D'UNE REQUÊTE

- L'exécution peut se voir sur deux niveaux
  - Niveau système
  - Niveau SGBD
- De toute façon, composée de plusieurs étapes

L'exécution d'une requête peut se voir sur deux niveaux :

- ce que le système perçoit ;
- ce que le SGBD fait.

Dans les deux cas, cela va nous permettre de trouver les possibilités de lenteurs dans l'exécution d'une requête par un utilisateur.

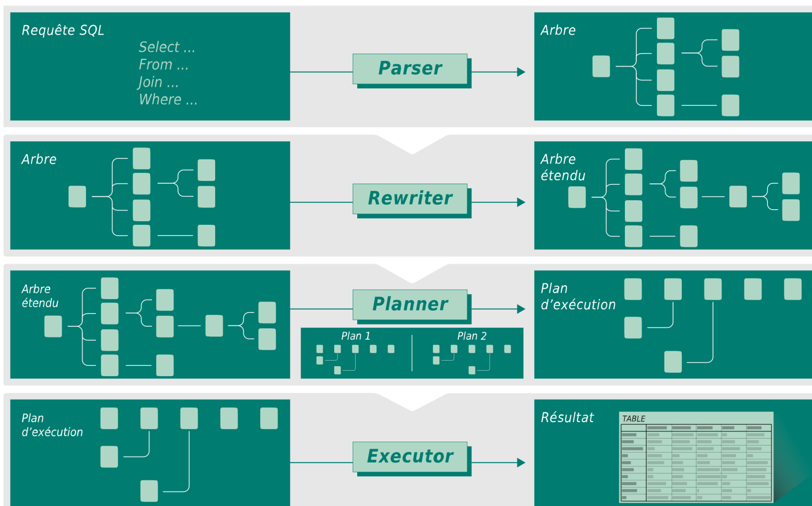
6.2.1 NIVEAU SYSTÈME

- Le client envoie une requête au serveur de bases de données
- Le serveur l'exécute
- Puis il renvoie le résultat au client

PostgreSQL est un système client-serveur. L'utilisateur se connecte via un outil (le client) à une base d'une instance PostgreSQL (le serveur). L'outil peut envoyer une requête au serveur, celui-ci l'exécute et finit par renvoyer les données résultant de la requête ou le statut de la requête.

Généralement, l'envoi de la requête est rapide. Par contre, la récupération des données peut poser problème si une grosse volumétrie est demandée sur un réseau à faible débit.

6.2.2 NIVEAU SGBD



Lorsque le serveur récupère la requête, un ensemble de traitements est réalisé :

- le `parser` va réaliser une analyse syntaxique de la requête
- le `rewriter` va réécrire, si nécessaire la requête

- pour cela, il prend en compte les règles et vues
- si une règle demande de changer la requête, la requête envoyée est remplacée par la nouvelle
- si une vue est utilisée, la requête qu'elle contient est intégrée dans la requête envoyée
- le `planner` va générer l'ensemble des plans d'exécutions
- il calcule le coût de chaque plan
- puis il choisit le plan le moins coûteux, donc le plus intéressant
- l' `executer` exécute la requête
- pour cela, il doit commencer par récupérer les verrous nécessaires sur les objets ciblés
- une fois les verrous récupérés, il exécute la requête
- une fois la requête exécutée, il envoie les résultats à l'utilisateur

Plusieurs goulets d'étranglement sont visibles ici. Les plus importants sont :

- la planification (à tel point qu'il est parfois préférable de ne générer qu'un sous-ensemble de plans, pour passer plus rapidement à la phase d'exécution) ;
- la récupération des verrous (une requête peut attendre plusieurs secondes, minutes, voire heures avant de récupérer les verrous et exécuter réellement la requête) ;
- l'exécution de la requête ;
- l'envoi des résultats à l'utilisateur.

Il est possible de tracer l'exécution des différentes étapes grâce aux options `log_parser_stats`, `log_planner_stats` et `log_executor_stats`. Voici un exemple complet :

- Mise en place de la configuration sur la session :

```
b1=# SET log_parser_stats TO on;
b1=# SET log_planner_stats TO on;
b1=# SET log_executor_stats TO on;
b1=# SET client_min_messages TO log;
```

- Exécution de la requête :

```
b1=# SELECT * FROM t1 WHERE id=10;
```

- Trace du `parser`

```
LOG:  PARSER STATISTICS
DETAIL:  ! system usage stats:
! 0.000051 elapsed 0.000000 user 0.000000 system sec
! [0.017997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/0 [40/1589] page faults/reclaims, 0 [0] swaps
```


DALIBO

```
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
LOG:  PARSE ANALYSIS STATISTICS
DETAIL:  ! system usage stats:
! 0.000197 elapsed 0.001000 user 0.000000 system sec
! [0.018997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/1 [40/1590] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
```

- Trace du `rewriter`

```
LOG:  REWRITER STATISTICS
DETAIL:  ! system usage stats:
! 0.000007 elapsed 0.000000 user 0.000000 system sec
! [0.018997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/0 [40/1590] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
```

- Trace du `planner`

```
LOG:  PLANNER STATISTICS
DETAIL:  ! system usage stats:
! 0.000703 elapsed 0.000000 user 0.000000 system sec
! [0.018997 user 0.021996 sys total]
! 0/0 [13040/248] filesystem blocks in/out
! 0/6 [40/1596] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [167/6] voluntary/involuntary context switches
```

- Trace du `executer`

```
LOG:  EXECUTOR STATISTICS
DETAIL:  ! system usage stats:
! 0.078548 elapsed 0.000000 user 0.000000 system sec
! [0.019996 user 0.021996 sys total]
! 16/0 [13056/248] filesystem blocks in/out
! 0/2 [40/1599] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 1/0 [168/6] voluntary/involuntary context switches
```

6.2.3 EXCEPTIONS

- Requêtes DDL
- Instructions TRUNCATE et COPY
- Pas de réécriture, pas de plans d'exécution... une exécution directe

Il existe quelques requêtes qui échappent à la séquence d'opérations présentées précédemment. Toutes les opérations DDL (modification de la structure de la base), les instructions `TRUNCATE` et `COPY` (en partie) sont vérifiées syntaxiquement, puis directement exécutées. Les étapes de réécriture et de planification ne sont pas réalisées.

Le principal souci pour les performances sur ce type d'instructions est donc l'obtention des verrous et l'exécution réelle.

6.3 QUELQUES DÉFINITIONS

- Prédicat
  - filtre de la clause `WHERE`
- Sélectivité
  - pourcentage de lignes retournées après application d'un prédicat
- Cardinalité
  - nombre de lignes d'une table
  - nombre de lignes retournées après filtrage

Un prédicat est une condition de filtrage présente dans la clause `WHERE` d'une requête. Par exemple `colonne = valeur`.

La sélectivité est liée à l'application d'un prédicat sur une table. Elle détermine le nombre de lignes remontées par la lecture d'une relation suite à l'application d'une clause de filtrage, ou prédicat. Elle peut être vue comme un coefficient de filtrage d'un prédicat. La sélectivité est exprimée sous la forme d'un pourcentage. Pour une table de 1000 lignes, si la sélectivité d'un prédicat est de 10%, la lecture de la table en appliquant le prédicat devrait retourner 100 lignes.

La cardinalité représente le nombre de lignes d'une relation. En d'autres termes, la cardinalité représente le nombre de lignes d'une table ou du résultat d'une fonction. Elle représente aussi le nombre de lignes retourné par la lecture d'une table après application d'un ou plusieurs prédicats.

6.3.1 REQUÊTE ÉTUDIÉE

Cette requête d'exemple :

```sql
SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM employes emp
  JOIN services ser ON (emp.num_service = ser.num_service)
 WHERE ser.localisation = 'Nantes';
```

Cette requête permet de déterminer quels sont les employés basés à Nantes.

Le script suivant permet de recréer le jeu d'essai :

```sql
CREATE TABLE services (
    num_service integer primary key,
    nom_service character varying(20),
    localisation character varying(20)
);

CREATE TABLE employes (
    matricule     integer primary key,
    nom           varchar(15) not null,
    prenom        varchar(15) not null,
    fonction      varchar(20) not null,
    manager       integer,
    date_embauche date,
    num_service   integer not null references services (num_service)
);

INSERT INTO services VALUES (1, 'Comptabilité', 'Paris');
INSERT INTO services VALUES (2, 'R&D', 'Rennes');
INSERT INTO services VALUES (3, 'Commerciaux', 'Limoges');
INSERT INTO services VALUES (4, 'Consultants', 'Nantes');

INSERT INTO employes VALUES
    (33, 'Roy', 'Arthur', 'Consultant', 105, '2000-06-01', 4);
INSERT INTO employes VALUES
    (81, 'Prunelle', 'Léon', 'Commercial', 97, '2000-06-01', 3);
INSERT INTO employes VALUES
    (97, 'Lebowski', 'Dude', 'Responsable', 104, '2003-01-01', 3);
INSERT INTO employes VALUES
    (104, 'Cruchot', 'Ludovic', 'Directeur Général', NULL, '2005-03-06', 3);
INSERT INTO employes VALUES
    (105, 'Vacuum', 'Anne-Lise', 'Responsable', 104, '2005-03-06', 4);
INSERT INTO employes VALUES
    (119, 'Thierrie', 'Armand', 'Consultant', 105, '2006-01-01', 4);
INSERT INTO employes VALUES
    (120, 'Tricard', 'Gaston', 'Développeur', 125, '2006-01-01', 2);
```

```sql
INSERT INTO employes VALUES
    (125, 'Berlicot', 'Jules', 'Responsable', 104, '2006-03-01', 2);
INSERT INTO employes VALUES
    (126, 'Fougasse', 'Lucien', 'Comptable', 128, '2006-03-01', 1);
INSERT INTO employes VALUES
    (128, 'Cruchot', 'Josépha', 'Responsable', 105, '2006-03-01', 1);
INSERT INTO employes VALUES
    (131, 'Lareine-Leroy', 'Émilie', 'Développeur', 125, '2006-06-01', 2);
INSERT INTO employes VALUES
    (135, 'Brisebard', 'Sylvie', 'Commercial', 97, '2006-09-01', 3);
INSERT INTO employes VALUES
    (136, 'Barnier', 'Germaine', 'Consultant', 105, '2006-09-01', 4);
INSERT INTO employes VALUES
    (137, 'Pivert', 'Victor', 'Consultant', 105, '2006-09-01', 4);
```

6.3.2 PLAN DE LA REQUÊTE ÉTUDIÉE

L'objet de ce module est de comprendre son plan d'exécution :

```
Hash Join  (cost=1.06..2.29 rows=4 width=48)
  Hash Cond: (emp.num_service = ser.num_service)
  ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
  ->  Hash  (cost=1.05..1.05 rows=1 width=21)
        ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
              Filter: ((localisation)::text = 'Nantes'::text)
```

La directive `EXPLAIN` permet de connaître le plan d'exécution d'une requête. Elle permet de savoir par quelles étapes va passer le SGBD pour répondre à la requête.

6.4 PLANIFICATEUR

- Chargé de sélectionner le meilleur plan d'exécution
- Énumère tous les plans d'exécution
  - Tous ou presque…
- Calcule leur coût suivant des statistiques, un peu de configuration et beaucoup de règles
- Sélectionne le meilleur (le moins coûteux)

Le but du planificateur est assez simple. Pour une requête, il existe de nombreux plans d'exécution possibles. Il va donc énumérer tous les plans d'exécution possibles (sauf

si cela représente vraiment trop de plans auquel cas, il ne prendra en compte qu'une partie des plans possibles). Il calcule ensuite le coût de chaque plan. Pour cela, il dispose d'informations sur les données (des statistiques), d'une configuration (réalisée par l'administrateur de bases de données) et d'un ensemble de règles inscrites en dur. Une fois tous les coûts calculés, il ne lui reste plus qu'à sélectionner le plan qui a le plus petit coût.

6.4.1 UTILITÉ

- SQL est un langage déclaratif
- Une requête décrit le résultat à obtenir
  - Mais pas la façon de l'obtenir
- C'est au planificateur de déduire le moyen de parvenir au résultat demandé

Le planificateur est un composant essentiel d'un moteur de bases de données. Les moteurs utilisent un langage SQL qui permet à l'utilisateur de décrire le résultat qu'il souhaite obtenir. Par exemple, s'il veut récupérer des informations sur tous les clients dont le nom commence par la lettre A en triant les clients par leur département, il pourrait utiliser une requête du type :

```
SELECT * FROM clients WHERE nom LIKE 'A%' ORDER BY departement;
```

Un moteur de bases de données peut récupérer les données de plusieurs façons :

- faire un parcours séquentiel de la table `clients` en filtrant les enregistrements d'après leur nom, puis trier les données grâce à un algorithme ;
- faire un parcours d'index sur la colonne nom pour trouver plus rapidement les enregistrements de la table `clients` satisfaisant le filtre 'A%', puis trier les données grâce à un algorithme ;
- faire un parcours d'index sur la colonne département pour récupérer les enregistrements déjà triés, et ne retourner que ceux vérifiant nom like 'A%'

Et ce ne sont que quelques exemples car il serait possible d'avoir un index utilisable pour le tri et le filtre par exemple.

Donc la requête décrit le résultat à obtenir, et le planificateur va chercher le meilleur moyen pour parvenir à ce résultat.

Pour ce travail, il dispose d'un certain nombre d'opérateurs. Ces opérateurs travaillent sur des ensembles de lignes, généralement un ou deux. Chaque opérateur renvoie un seul ensemble de lignes. Le planificateur peut combiner ces opérations suivant certaines règles. Un opérateur peut renvoyer l'ensemble de résultats de deux façons : d'un coup

(par exemple le tri) ou petit à petit (par exemple un parcours séquentiel). Le premier cas utilise plus de mémoire, et peut nécessiter d'écrire des données temporaires sur disque. Le deuxième cas aide à accélérer des opérations comme les curseurs, les sous-requêtes `IN` et `EXISTS`, la clause `LIMIT`, etc.

6.4.2 RÈGLES

- 1ère règle : Récupérer le bon résultat
- 2è règle : Le plus rapidement possible
  - En minimisant les opérations disques
  - En préférant les lectures séquentielles
  - En minimisant la charge CPU
  - En minimisant l'utilisation de la mémoire

Le planificateur suit deux règles :

- il doit récupérer le bon résultat ;
- il doit le récupérer le plus rapidement possible.

Cette deuxième règle lui impose de minimiser l'utilisation des ressources : en tout premier lieu les opérations disques vu qu'elles sont les plus coûteuses, mais aussi la charge CPU et l'utilisation de la mémoire. Dans le cas des opérations disques, s'il doit en faire, il doit absolument privilégier les opérations séquentielles aux opérations aléatoires (qui demandent un déplacement de la tête de disque, ce qui est l'opération la plus coûteuse sur les disques magnétiques).

6.4.3 OUTILS DE L'OPTIMISEUR

- L'optimiseur s'appuie sur :
  - un mécanisme de calcul de coûts
  - des statistiques sur les données
  - le schéma de la base de données

Pour déterminer le chemin d'exécution le moins coûteux, l'optimiseur devrait connaître précisément les données mises en œuvre dans la requête, les particularités du matériel et la charge en cours sur ce matériel. Cela est impossible. Ce problème est contourné en utilisant deux mécanismes liés l'un à l'autre :

- un mécanisme de calcul de coût de chaque opération,

- des statistiques sur les données.

Pour quantifier la charge nécessaire pour répondre à une requête, PostgreSQL utilise un mécanisme de coût. Il part du principe que chaque opération a un coût plus ou moins important. Les statistiques sur les données permettent à l'optimiseur de requêtes de déterminer assez précisément la répartition des valeurs d'une colonne d'une table, sous la forme d'histogramme. Il dispose encore d'autres informations comme la répartition des valeurs les plus fréquentes, le pourcentage de `NULL`, le nombre de valeurs distinctes, etc… Toutes ces informations aideront l'optimiseur à déterminer la sélectivité d'un filtre (prédicat de la clause `WHERE`, condition de jointure) et donc quelle est la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué. Enfin, l'optimiseur s'appuie sur le schéma de la base de données afin de déterminer différents paramètres qui entrent dans le calcul du plan d'exécution : contrainte d'unicité sur une colonne, présence d'une contrainte `NOT NULL`, etc.

6.4.4 OPTIMISATIONS

- À partir du modèle de données
  - suppression de jointures externes inutiles
- Transformation des sous-requêtes
  - certaines sous-requêtes transformées en jointures
- Appliquer les prédicats le plus tôt possible
  - réduit le jeu de données manipulé
- Intègre le code des fonctions SQL simples (inline)
  - évite un appel de fonction coûteux

À partir du modèle de données et de la requête soumise, l'optimiseur de PostgreSQL va pouvoir déterminer si une jointure externe n'est pas utile à la production du résultat.

Suppression des jointures externes inutiles

Sous certaines conditions, PostgreSQL peut supprimer des jointures externes, à condition que le résultat ne soit pas modifié :

```
EXPLAIN SELECT e.matricule, e.nom, e.prenom
  FROM employes e
  LEFT JOIN services s
    ON (e.num_service = s.num_service)
 WHERE e.num_service = 4;
                        QUERY PLAN
-----------------------------------------------------------
```

```
 Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=23)
   Filter: (num_service = 4)
```

Toutefois, si le prédicat de la requête est modifié pour s'appliquer sur la table `services`, la jointure est tout de même réalisée, puisqu'on réalise un test d'existence sur cette table `services` :

```
EXPLAIN SELECT e.matricule, e.nom, e.prenom
  FROM employes e
  LEFT JOIN services s
    ON (e.num_service = s.num_service)
 WHERE s.num_service = 4;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.15..9.39 rows=5 width=19)
   ->  Index Only Scan using services_pkey on services s  (cost=0.15..8.17...)
         Index Cond: (num_service = 4)
   ->  Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=23)
         Filter: (num_service = 4)
```

Transformation des sous-requêtes

Certaines sous-requêtes sont transformées en jointure :

```
EXPLAIN SELECT *
  FROM employes emp
  JOIN (SELECT * FROM services WHERE num_service = 1) ser
    ON (emp.num_service = ser.num_service);
                               QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.15..9.36 rows=2 width=163)
   ->  Index Scan using services_pkey on services  (cost=0.15..8.17...)
         Index Cond: (num_service = 1)
   ->  Seq Scan on employes emp  (cost=0.00..1.18 rows=2 width=43)
         Filter: (num_service = 1)
(5 lignes)
```

La sous-requête `ser` a été remonté dans l'arbre de requête pour être intégré en jointure.

Application des prédicats au plus tôt

Lorsque cela est possible, PostgreSQL essaye d'appliquer les prédicats au plus tôt :

```
EXPLAIN SELECT MAX(date_embauche)
  FROM (SELECT * FROM employes WHERE num_service = 4) e
 WHERE e.date_embauche < '2006-01-01';
                                  QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=1.21..1.22 rows=1 width=4)
   ->  Seq Scan on employes  (cost=0.00..1.21 rows=2 width=4)
```

```
         Filter: ((date_embauche < '2006-01-01'::date) AND (num_service = 4))
(3 lignes)
```

Les deux prédicats `num_service = 4` et `date_embauche < '2006-01-01'` ont été appliqués en même temps, réduisant ainsi le jeu de données à considéré dès le départ.

En cas de problème, il est possible d'utiliser une CTE (clause *WITH*) pour bloquer cette optimisation :

```
EXPLAIN WITH e AS (SELECT * FROM employes WHERE num_service = 4)
SELECT MAX(date_embauche)
 FROM e
WHERE e.date_embauche < '2006-01-01';
                           QUERY PLAN
-----------------------------------------------------------------
 Aggregate  (cost=1.29..1.30 rows=1 width=4)
   CTE e
     ->  Seq Scan on employes  (cost=0.00..1.18 rows=5 width=43)
           Filter: (num_service = 4)
   ->  CTE Scan on e  (cost=0.00..0.11 rows=2 width=4)
         Filter: (date_embauche < '2006-01-01'::date)
```

Function inlining

```
CREATE TABLE inline (id serial, tdate date);
INSERT INTO inline (tdate)
 SELECT generate_series('1800-01-01', '2015-12-01', interval '15 days');

CREATE OR REPLACE FUNCTION add_months_sql(mydate date, nbrmonth integer)
  RETURNS date AS
$BODY$
SELECT ( mydate + interval '1 month' * nbrmonth )::date;
$BODY$
  LANGUAGE SQL;

CREATE OR REPLACE FUNCTION add_months_plpgsql(mydate date, nbrmonth integer)
  RETURNS date AS
$BODY$
 BEGIN RETURN ( mydate + interval '1 month' * nbrmonth ); END;
$BODY$
  LANGUAGE plpgsql;
```

Si l'on utilise la fonction écrite en PL/pgsql, on retrouve l'appel de la fonction dans la clause `Filter` du plan d'exécution de la requête :

```
mabase=#EXPLAIN (ANALYZE, BUFFERS) SELECT *
FROM inline WHERE tdate = add_months_plpgsql(now()::date, -1);
                                            QUERY PLAN
-----------------------------------------------------------------------------
```

```
 Seq Scan on inline  (cost=0.00..1430.52...) (actual time=42.102..42.102...)
   Filter: (tdate = add_months_plpgsql((now())::date, (-1)))
   Rows Removed by Filter: 5258
   Buffers: shared hit=24
 Total runtime: 42.139 ms
(5 lignes)
```

PostgreSQL ne sait pas intégrer le code des fonctions PL/pgsql dans ses plans d'exécution.

En revanche, en utilisant la fonction écrite en langage SQL, la définition de la fonction a été intégrée dans la clause de filtrage de la requête :

```
mabase=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM inline
WHERE tdate = add_months_sql(now()::date, -1);
                                          QUERY PLAN
----------------------------------------------------------------------------
 Seq Scan on inline  (cost=0.00..142.31...) (actual time=6.647..6.647...)
   Filter: (tdate = (((now())::date + '-1 mons'::interval))::date)
   Rows Removed by Filter: 5258
   Buffers: shared hit=24
 Total runtime: 6.699 ms
(5 lignes)
```

Le code de la fonction SQL a été correctement intégré dans le plan d'exécution. Le temps d'exécution a été divisé par 6 sur le jeu de donnée réduit, montrant l'impact de l'appel d'une fonction dans une clause de filtrage.

6.4.5 DÉCISIONS

- Stratégie d'accès aux lignes
  - Par parcours d'une table, d'un index, de TID, etc
- Stratégie d'utilisation des jointures
  - Ordre des jointures
  - Type de jointure (Nested Loop, Merge/Sort Join, Hash Join)
  - Ordre des tables jointes dans une même jointure
- Stratégie d'agrégation
  - Brut, trié, haché

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut utiliser un parcours de table, un parcours d'index ou encore d'autres types de parcours. Ce sont généralement les premières opérations utilisées. Ensuite, d'autres opérations permettent différentes actions :

- joindre deux ensembles de lignes avec des opérations de jointures (trois au total) ;

- agréger un ensemble de lignes avec une opération d'agrégation (trois là- aussi) ;
- trier un ensemble de lignes ;
- etc.

6.5 MÉCANISME DE COÛTS

- Modèle basé sur les coûts
  - quantifier la charge pour répondre à une requête
- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne issue d'une lecture de table ou d'index
  - appliquer un opérateur

L'optimiseur statistique de PostgreSQL utilise un modèle de calcul de coût. Les coûts calculés sont des indications arbitraires sur la charge nécessaire pour répondre à une requête. Chaque facteur de coût représente une unité de travail : lecture d'un bloc, manipulation des lignes en mémoire, application d'un opérateur sur des données.

6.5.1 COÛTS UNITAIRES

- L'optimiseur a besoin de connaître :
  - le coût relatif d'un accès séquentiel au disque.
  - le coût relatif d'un accès aléatoire au disque.
  - le coût relatif de la manipulation d'une ligne en mémoire.
  - le coût de traitement d'une donnée issue d'un index.
  - le coût d'application d'un opérateur.
  - le coût de la manipulation d'une ligne en mémoire pour un parcours parallèle parallélisé.
  - le coût de mise en place d'un parcours parallélisé.

Pour quantifier la charge nécessaire pour répondre à une requête, PostgreSQL utilise un mécanisme de coût. Il part du principe que chaque opération a un coût plus ou moins important.

Sept paramètres permettent d'ajuster les coûts relatifs :

- `seq_page_cost` représente le coût relatif d'un accès séquentiel au disque. Ce paramètre vaut 1 et ne devrait pas être modifié.

- `random_page_cost` représente le coût relatif d'un accès aléatoire au disque. Ce paramètre vaut 4 par défaut, cela signifie que le temps de déplacement de la tête de lecture de façon aléatoire est estimé quatre fois plus important que le temps d'accès d'un bloc à un autre.
- `cpu_tuple_cost` représente le coût relatif de la manipulation d'une ligne en mémoire. Ce paramètre vaut par défaut 0,01.
- `cpu_index_tuple_cost` répercute le coût de traitement d'une donnée issue d'un index. Ce paramètre vaut par défaut 0,005.
- `cpu_operator_cost` indique le coût d'application d'un opérateur sur une donnée. Ce paramètre vaut par défaut 0,0025.
- `parallel_tuple_cost` indique le coût de traitement d'une ligne lors d'un parcours parallélisé. Ce paramètre vaut par défaut 0.1.
- `parallel_setup_cost` indique le coût de mise en place d'un parcours parallélisé. Ce paramètre vaut par défaut 1000.0.

En général, on ne modifie pas ces paramètres sans justification sérieuse. On peut être amené à diminuer `random_page_cost` si le serveur dispose de disques rapides et d'une carte RAID équipée d'un cache important. Mais en faisant cela, il faut veiller à ne pas déstabiliser des plans optimaux qui obtiennent des temps de réponse constant. À trop diminuer `random_page_cost`, on peut obtenir de meilleurs temps de réponse si les données sont en cache, mais aussi des temps de réponse dégradés si les données ne sont pas en cache. Il n'est pas recommandé de modifier les paramètres `cpu_tuple_cost`, `cpu_index_tuple_cost` et `cpu_operator_cost` sans réelle justification.

Pour des besoins particuliers, ces paramètres sont des paramètres de sessions. Ils peuvent être modifiés dynamiquement avec l'ordre `SET` au niveau de l'application en vue d'exécuter des requêtes bien particulières.

6.6 STATISTIQUES

- Toutes les décisions du planificateur se basent sur les statistiques
  - Le choix du parcours
  - Comme le choix des jointures
- Statistiques mises à jour avec ANALYZE
- Sans bonnes statistiques, pas de bons plans

Le planificateur se base principalement sur les statistiques pour ses décisions. Le choix du parcours, le choix des jointures, le choix de l'ordre des jointures, tout cela dépend des statistiques (et un peu de la configuration). Sans statistiques à jour, le choix du planifi-

cateur a un fort risque d'être mauvais. Il est donc important que les statistiques soient mises à jour fréquemment. La mise à jour se fait avec l'instruction `ANALYZE` qui peut être exécuté manuellement ou automatiquement (via un cron ou l'autovacuum par exemple).

6.6.1 UTILISATION DES STATISTIQUES

- L'optimiseur utilise les statistiques pour déterminer :
  - la cardinalité d'un filtre -> quelle stratégie d'accès
  - la cardinalité d'une jointure -> quel algorithme de jointure
  - la cardinalité d'un regroupement -> quel algorithme de regroupement

Les statistiques sur les données permettent à l'optimiseur de requêtes de déterminer assez précisément la répartition des valeurs d'une colonne d'une table, sous la forme d'un histogramme de répartition des valeurs. Il dispose encore d'autres informations comme la répartition des valeurs les plus fréquentes, le pourcentage de `NULL`, le nombre de valeurs distinctes, etc... Toutes ces informations aideront l'optimiseur à déterminer la sélectivité d'un filtre (prédicat de la clause `WHERE`, condition de jointure) et donc quelle sera la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué.

Par exemple, pour une table simple, nommée `test`, de 1 million de lignes dont 250000 lignes ont des valeurs uniques et les autres portent la même valeur :

```
CREATE TABLE test (i integer not null, t text);
INSERT INTO test SELECT CASE WHEN i > 250000 THEN 250000 ELSE i END,
md5(i::text) FROM generate_series(1, 1000000) i;
CREATE INDEX ON test (i);
```

Après un chargement massif de données, il est nécessaire de collecter les statistiques :

```
ANALYZE test;
```

Ensuite, grâce aux statistiques connues par PostgreSQL (voir la vue `pg_stats`), l'optimiseur est capable de déterminer le chemin le plus intéressant selon les valeurs recherchées.

Ainsi, avec un filtre peu sélectif, `i = 250000`, la requête va ramener les 3/ 4 de la table. PostgreSQL choisira donc une lecture séquentielle de la table, ou `Seq Scan` :

```
base=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM test WHERE i = 250000;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on test  (cost=[...] rows=752400) (actual [...] rows=750001 loops=1)
   Filter: (i = 250000)
   Rows Removed by Filter: 249999
```

```
   Buffers: shared hit=8334
 Total runtime: 244.605 ms
(5 lignes)
```

La partie `cost` montre que l'optimiseur estime que la lecture va ramener 752400 lignes. En réalité, ce sont 750001 lignes qui sont ramenées. L'optimiseur se base donc sur une estimation obtenue selon la répartition des données.

Avec un filtre plus sélectif, la requête ne ramènera qu'une seule ligne. L'optimiseur préférera donc passer par l'index que l'on a créé :

```
base=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM test WHERE i = 250;
                                       QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on test  ([...] rows=25 width=37) ([...] rows=1 loops=1)
   Recheck Cond: (i = 250)
   Buffers: shared hit=4
   ->  Bitmap Index Scan on test_i_idx  ([...] rows=25) ([...] rows=1 loops=1)
         Index Cond: (i = 250)
         Buffers: shared hit=3
 Total runtime: 0.134 ms
(7 lignes)
```

Dans ce deuxième essai, l'optimiseur estime ramener 25 lignes. En réalité, il n'en ramène qu'une seule. L'estimation reste relativement précise étant donné le volume de données.

Dans le premier cas, l'optimiseur estime qu'il est moins coûteux de passer par une lecture séquentielle de la table plutôt qu'une lecture d'index. Dans le second cas, où le filtre est très sélectif, une lecture par index est plus appropriée.

6.6.2 STATISTIQUES : TABLE ET INDEX

- Taille
- Cardinalité
- Stocké dans `pg_class`
  - `relpages` et `reltuples`

L'optimiseur a besoin de deux données statistiques pour une table ou un index : sa taille physique et le nombre de lignes portées par l'objet.

Ces deux données statistiques sont stockées dans la table `pg_class`. La taille de la table ou de l'index est exprimée en nombre de blocs de 8 Ko et stockée dans la colonne `relpages`. La cardinalité de la table ou de l'index, c'est-à-dire le nombre de lignes, est stockée dans la colonne `reltuples`.

L'optimiseur utilisera ces deux informations pour apprécier la cardinalité de la table en fonction de sa volumétrie courante en calculant sa densité estimée puis en utilisant cette densité multipliée par le nombre de blocs actuel de la table pour estimer le nombre de lignes réel de la table :

```
density = reltuples / relpages;
tuples = density * curpages;
```

6.6.3 STATISTIQUES : MONO-COLONNE

- Nombre de valeurs distinctes
- Nombre d'éléments qui n'ont pas de valeur (`NULL`)
- Largeur d'une colonne
- Distribution des données
  - tableau des valeurs les plus fréquentes
  - histogramme de répartition des valeurs

Au niveau d'une colonne, plusieurs données statistiques sont stockées :

- le nombre de valeurs distinctes,
- le nombre d'éléments qui n'ont pas de valeur (`NULL`),
- la largeur moyenne des données portées par la colonne,
- le facteur de corrélation entre l'ordre des données triées et la répartition physique des valeurs dans la table,
- la distribution des données.

La distribution des données est représentée sous deux formes qui peuvent être complémentaires. Tout d'abord, un tableau de répartition permet de connaître les valeurs les plus fréquemment rencontrées et la fréquence d'apparition de ces valeurs. Un histogramme de distribution des valeurs rencontrées permet également de connaître la répartition des valeurs pour la colonne considérée.

6.6.4 STOCKAGE DES STATISTIQUES MONO-COLONNE

- Les informations statistiques vont dans la table `pg_statistic`
  - mais elle est difficile à comprendre
  - mieux vaut utiliser la vue `pg_stats`
  - une table vide n'a pas de statistiques
- Taille et cardinalité dans `pg_class`

  - colonnes relpages et reltuples

Le stockage des statistiques se fait dans le catalogue système pg_statistic mais les colonnes de cette table sont difficiles à interpréter. Il est préférable de passer par la vue pg_stats qui est plus facilement compréhensible par un être humain.

La collecte des statistiques va également mettre à jour la table pg_class avec deux informations importantes pour l'optimiseur. Il s'agit de la taille d'une table, exprimée en nombre de blocs de 8 Ko et stockée dans la colonne relpages. La cardinalité de la table, c'est-à-dire le nombre de lignes de la table, est stockée dans la colonne reltuples. L'optimiseur utilisera ces deux informations pour apprécier la cardinalité de la table en fonction de sa volumétrie courante.

6.6.5 VUE PG_STATS

- Une ligne par colonne de chaque table
- 3 colonnes d'identification
  - schemaname, tablename, attname
- 8 colonnes d'informations statistiques
  - inherited, null_frac, avg_width, n_distinct
  - most_common_vals, most_common_freqs, histogram_bounds
  - most_common_elems, most_common_elem_freqs, elem_count_histogram
  - correlation

La vue pg_stats a été créée pour faciliter la compréhension des statistiques récupérées par la commande ANALYZE.

Elle est composée de trois colonnes qui permettent d'identifier la colonne :

- schemaname : nom du schéma (jointure possible avec pg_namespace)
- tablename : nom de la table (jointure possible avec pg_class, intéressant pour récupérer reltuples et relpages)
- attname : nom de la colonne (jointure possible avec pg_attribute, intéressant pour récupérer attstatstarget, valeur d'échantillon)

Suivent ensuite les colonnes de statistiques.

inherited

Si true, les statistiques incluent les valeurs de cette colonne dans les tables filles.

Exemple

```
b1=# SELECT count(*) FROM ONLY parent;
-[ RECORD 1 ]
count | 0
b1=# SELECT * FROM pg_stats WHERE tablename='parent';
-[ RECORD 1 ]-----+-----------------------------------------------------
schemaname        | public
tablename         | parent
attname           | id
inherited         | t
null_frac         | 0
avg_width         | 4
n_distinct        | -0.285714
most_common_vals  | {1,2,17,18,19,20,3,4,5,15,16,6,7,8,9,10}
[...]
histogram_bounds  | {11,12,13,14}
correlation       | 0.762715
```

null_frac

Cette statistique correspond au pourcentage de valeurs NULL dans l'échantillon considéré. Elle est toujours calculée.

avg_width

Il s'agit de la largeur moyenne en octets des éléments de cette colonne. Elle est constante pour les colonnes dont le type est à taille fixe (integer, booléen, char, etc.). Dans le cas du type `char(n)`, il s'agit du nombre de caractères saisissables + 1. Il est variable pour les autres (principalement text, varchar, bytea).

n_distinct

Si cette colonne contient un nombre positif, il s'agit du nombre de valeurs distinctes dans l'échantillon. Cela arrive uniquement quand le nombre de valeurs distinctes possibles semble fixe.

Si cette colonne contient un nombre négatif, il s'agit du nombre de valeurs distinctes dans l'échantillon divisé par le nombre de lignes. Cela survient uniquement quand le nombre de valeurs distinctes possibles semble variable. -1 indique donc que toutes les valeurs sont distinctes, -0,5 que chaque valeur apparaît deux fois.

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur =.

Il est possible de forcer cette colonne a une valeur constante en utilisant l'ordre `ALTER TABLE nom_table ALTER COLUMN nom_colonne SET (parametre = valeur);` où parametre vaut soit `n_distinct` (pour une table standard) soit `n_distinct_inherited` (pour une table comprenant des partitions). Pour les grosses tables contenant des valeurs distinctes, indiquer une grosse valeur ou la valeur -1 permet de favoriser l'utilisation de

parcours d'index à la place de parcours de bitmap. C'est aussi utile pour des tables où les données ne sont pas réparties de façon homogène, et où la collecte de cette statistique est alors faussée.

most_common_vals

Cette colonne contient une liste triée des valeurs les plus communes. Elle peut être `NULL` si les valeurs semblent toujours aussi communes ou si le type de données n'a pas d'opérateur =.

most_common_freqs

Cette colonne contient une liste triée des fréquences pour les valeurs les plus communes. Cette fréquence est en fait le nombre d'occurrences de la valeur divisé par le nombre de lignes. Elle est `NULL` si `most_common_vals` est `NULL`.

histogram_bounds

PostgreSQL prend l'échantillon récupéré par `ANALYZE`. Il trie ces valeurs. Ces données triées sont partagées en x tranches, appelées classes, égales, où x dépend de la valeur du paramètre `default_statistics_target` ou de la configuration spécifique de la colonne. Il construit ensuite un tableau dont chaque valeur correspond à la valeur de début d'une tranche.

most_common_elems, most_common_elem_freqs, elem_count_histogram

Ces trois colonnes sont équivalentes aux trois précédentes, mais uniquement pour les données de type tableau.

correlation

Cette colonne est la corrélation statistique entre l'ordre physique et l'ordre logique des valeurs de la colonne. Si sa valeur est proche de -1 ou 1, un parcours d'index est privilégié. Si elle est proche de 0, un parcours séquentiel est mieux considéré.

Cette colonne peut être `NULL` si le type de données n'a pas d'opérateur `<`.

6.6.6 STATISTIQUES : MULTI-COLONNES

- Pas par défaut
- `CREATE STATISTICS`
- Deux types de statistique
  - nombre de valeurs distinctes
  - dépendances fonctionnelles

- À partir de la version 10

Par défaut, la commande `ANALYZE` de PostgreSQL calcule des statistiques mono-colonnes uniquement. Depuis la version 10, elle peut aussi calculer certaines statistiques multi-colonnes.

Pour cela, il est nécessaire de créer un objet statistique avec l'ordre SQL `CREATE STATISTICS`. Cet objet indique les colonnes concernées ainsi que le type de statistique souhaité.

Actuellement, PostgreSQL supporte deux types de statistiques pour ces objets :

- `ndistinct` pour le nombre de valeurs distinctes sur ces colonnes ;
- `dependencies` pour les dépendances fonctionnelles.

Dans les deux cas, cela peut permettre d'améliorer fortement les estimations de nombre de lignes, ce qui ne peut qu'amener de meilleurs plans d'exécution.

6.6.7 CATALOGUE PG_STATISTIC_EXT

- Une ligne par objet statistique
- 4 colonnes d'identification
  - stxrelid, stxname, stxnamespace, stxkeys
- 1 colonne pour connaître le type de statistiques géré
  - stxkind
- 2 colonnes d'informations statistiques
  - stxndistinct
  - stxdependencies

`stxname` est le nom de l'objet statistique, et `stxnamespace` l'OID de son schéma.

`stxrelid` précise l'OID de la table concernée par cette statistique. `stxkeys` est un tableau d'entiers correspondant aux numéros des colonnes.

`stxkind` peut avoir une ou plusieurs valeurs parmi `d` pour le nombre de valeurs distinctes et `f` pour les dépendances fonctionnelles.

Créons une table avec deux colonnes et peuplons-la avec les mêmes données :

```
postgres=# CREATE TABLE t (a INT, b INT);
CREATE TABLE
postgres=# INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
INSERT 0 10000
postgres=# ANALYZE t;
```

17.12

ANALYZE

Après une analyse des données de la table, les statistiques sont à jour comme le montrent ces deux requêtes :

```
postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE a = 1;
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8)
         (actual time=0.037..1.704 rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
 Planning time: 0.097 ms
 Execution time: 1.731 ms
(5 rows)

postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE b = 1;
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8)
         (actual time=0.010..1.590 rows=100 loops=1)
   Filter: (b = 1)
   Rows Removed by Filter: 9900
 Planning time: 0.029 ms
 Execution time: 1.609 ms
(5 rows)
```

Cela fonctionne bien (*i.e.* l'estimation du nombre de lignes est très proche de la réalité) dans le cas spécifique où le filtre se fait sur une seule colonne. Par contre, si le filtre se fait sur les deux colonnes, l'estimation diffère d'un facteur d'échelle :

```
postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE a = 1 AND b = 1;
                    QUERY PLAN
---------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8)
       (actual time=0.009..1.554 rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
 Planning time: 0.044 ms
 Execution time: 1.573 ms
(5 rows)
```

En fait, il y a une dépendance fonctionnelle entre ces deux colonnes mais PostgreSQL ne le sait pas car ses statistiques sont mono-colonnes par défaut. Pour avoir des statistiques sur les deux colonnes, il faut créer un objet statistique pour ces deux colonnes :

```
postgres=# CREATE STATISTICS stts (dependencies) ON a, b FROM t;
CREATE STATISTICS
postgres=# ANALYZE t;
ANALYZE
postgres=# EXPLAIN (ANALYZE) SELECT * FROM t WHERE a = 1 AND b = 1;
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8)
         (actual time=0.007..0.668 rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
 Planning time: 0.093 ms
 Execution time: 0.683 ms
(5 rows)
```

Cette fois, l'estimation est beaucoup plus proche de la réalité.

Maintenant, prenons le cas d'un regroupement :

```
postgres=# EXPLAIN (ANALYZE) SELECT COUNT(*) FROM t GROUP BY a;
                          QUERY PLAN
-------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12)
            (actual time=2.346..2.358 rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4)
               (actual time=0.006..0.640 rows=10000 loops=1)
 Planning time: 0.024 ms
 Execution time: 2.381 ms
(5 rows)
```

L'estimation du nombre de lignes pour un regroupement sur une colonne est très bonne. Par contre, sur deux colonnes :

```
postgres=# EXPLAIN (ANALYZE) SELECT COUNT(*) FROM t GROUP BY a, b;
                           QUERY PLAN
-------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16)
             (actual time=2.321..2.339 rows=100 loops=1)
```

```
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8)
               (actual time=0.004..0.596 rows=10000 loops=1)
 Planning time: 0.025 ms
 Execution time: 2.359 ms
(5 rows)
```

Là-aussi, on constate un facteur d'échelle important entre l'estimation et la réalité. Et là-aussi, c'est un cas où un objet statistique peut fortement aider :

```
postgres=# DROP STATISTICS stts;
DROP STATISTICS
postgres=# CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
CREATE STATISTICS
postgres=# ANALYZE t;
ANALYZE
postgres=# EXPLAIN (ANALYZE) SELECT COUNT(*) FROM t GROUP BY a, b;
                      QUERY PLAN
-------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16)
            (actual time=3.310..3.324 rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8)
               (actual time=0.007..0.807 rows=10000 loops=1)
 Planning time: 0.087 ms
 Execution time: 3.356 ms
(5 rows)
```

L'estimation est bien meilleure grâce aux statistiques spécifiques aux deux colonnes.

6.6.8 ANALYZE

- Ordre SQL de calcul de statistiques
  - ANALYZE [VERBOSE] [table [(colonne [, ...])]]
- Sans argument : base entière
- Avec argument : la table complète ou certaines colonnes seulement
- Prend un échantillon de chaque table
- Et calcule des statistiques sur cet échantillon
- Si table vide, conservation des anciennes statistiques

ANALYZE est l'ordre SQL permettant de mettre à jour les statistiques sur les données. Sans argument, l'analyse se fait sur la base complète. Si un argument est donné, il doit correspondre au nom de la table à analyser. Il est même possible d'indiquer les colonnes à traiter.

En fait, cette instruction va exécuter un calcul d'un certain nombre de statistiques. Elle ne va pas lire la table entière, mais seulement un échantillon. Sur cet échantillon, chaque colonne sera traitée pour récupérer quelques informations comme le pourcentage de valeurs NULL, les valeurs les plus fréquentes et leur fréquence, sans parler d'un histogramme des valeurs. Toutes ces informations sont stockées dans un catalogue système nommé pg_statistics.

Dans le cas d'une table vide, les anciennes statistiques sont conservées. S'il s'agit d'une nouvelle table, les statistiques sont initialement vides. La table n'est jamais considérée vide par l'optimiseur, qui utilise alors des valeurs par défaut.

6.6.9 FRÉQUENCE D'ANALYSE

- Dépend principalement de la fréquence des requêtes DML
- Cron
  - Avec psql
  - Avec vacuumdb (option --analyze-only en 9.0)
- Autovacuum fait du ANALYZE
  - Pas sur les tables temporaires
  - Pas assez rapidement dans certains cas

Les statistiques doivent être mises à jour fréquemment. La fréquence exacte dépend surtout de la fréquence des requêtes d'insertion/modification/ suppression des lignes des tables. Néanmoins, un ANALYZE tous les jours semble un minimum, sauf cas spécifique.

L'exécution périodique peut se faire avec cron (ou les tâches planifiées sous Windows). Il n'existe pas d'outils PostgreSQL pour lancer un seul ANALYZE . L'outil vaccumdb se voit doté d'une option --analyze-only pour combler ce manque. Avant, il était nécessaire de passer par psql et son option -c qui permet de préciser la requête à exécuter. En voici un exemple :

```
psql -c "ANALYZE" b1
```

Cet exemple exécute la commande ANALYZE sur la base b1 locale.

Le démon autovacuum fait aussi des ANALYZE. La fréquence dépend de sa configuration. Cependant, il faut connaître deux particularités de cet outil :

- Ce démon a sa propre connexion à la base. Il ne peut donc pas voir les tables temporaires appartenant aux autres sessions. Il ne sera donc pas capable de mettre à jour leurs statistiques.
- Après une insertion ou une mise à jour massive, autovacuum ne va pas forcément lancer un `ANALYZE` immédiat. En effet, `autovacuum` ne cherche les tables à traiter que toutes les minutes (par défaut). Si, après la mise à jour massive, une requête est immédiatement exécutée, il y a de fortes chances qu'elle s'exécute avec des statistiques obsolètes. Il est préférable dans ce cas de lancer un `ANALYZE` manuel sur la ou les tables ayant subi l'insertion ou la mise à jour massive.

6.6.10 ÉCHANTILLON STATISTIQUE

- Se configure dans postgresql.conf
  - `default_statistics_target = 100`
- Configurable par colonne

```
ALTER TABLE nom ALTER [ COLUMN ] colonne SET STATISTICS valeur;
```

- Par défaut, récupère 30000 lignes au hasard
  - `300 * default_statistics_target`
- Va conserver les 100 valeurs les plus fréquentes avec leur fréquence

Par défaut, un `ANALYZE` récupère 30000 lignes d'une table. Les statistiques générées à partir de cet échantillon sont bonnes si la table ne contient pas des millions de lignes. Si c'est le cas, il faudra augmenter la taille de l'échantillon. Pour cela, il faut augmenter la valeur du paramètre `default_statistics_target`. Ce dernier vaut 100 par défaut. La taille de l'échantillon est de `300 x default_statistics_target`. Augmenter ce paramètre va avoir plusieurs répercussions. Les statistiques seront plus précises grâce à un échantillon plus important. Mais du coup, les statistiques seront plus longues à calculer, prendront plus de place sur le disque, et demanderont plus de travail au planificateur pour générer le plan optimal. Augmenter cette valeur n'a donc pas que des avantages.

Du coup, les développeurs de PostgreSQL ont fait en sorte qu'il soit possible de le configurer colonne par colonne avec l'instruction suivante :

```
ALTER TABLE nom_table ALTER [ COLUMN ] nom_colonne SET STATISTICS valeur;
```

6.7 QU'EST-CE QU'UN PLAN D'EXÉCUTION ?

- Plan d'exécution
  - représente les différentes opérations pour répondre à la requête
  - sous forme arborescente
  - composé des nœuds d'exécution
  - plusieurs opérations simples mises bout à bout

6.7.1 NŒUD D'EXÉCUTION

- Nœud
  - opération simple : lectures, jointures, tris, etc.
  - unité de traitement
  - produit et consomme des données
- Enchaînement des opérations
  - chaque nœud produit les données consommées par le nœud parent
  - nœud final retourne les données à l'utilisateur

Les nœuds correspondent à des unités de traitement qui réalisent des opérations simples sur un ou deux ensembles de données : lecture d'une table, jointures entre deux tables, tri d'un ensemble, etc. Si le plan d'exécution était une recette, chaque nœud serait une étape de la recette.

Les nœuds peuvent produire et consommer des données.

6.7.2 LECTURE D'UN PLAN

```
                          QUERY PLAN
-----------------------------------------------------------------
 Hash Join  (cost=1.06..2.29 rows=4 width=48)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
   ->  Hash  (cost=1.05..1.05 rows=1 width=21)
         ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
               Filter: ((localisation)::text = 'Nantes'::text)
```

Un plan d'exécution est lu en partant du nœud se trouvant le plus à droite et en remontant jusqu'au nœud final. Quand le plan contient plusieurs nœuds, le premier nœud exécuté est celui qui se trouve le plus à droite. Celui qui est le plus à gauche (la première ligne) est le dernier nœud exécuté. Tous les nœuds sont exécutés simultanément, et traitent les données dès qu'elles sont transmises par le nœud parent (le ou les nœuds juste en dessous, à droite).

Chaque nœud montre les coûts estimés dans le premier groupe de parenthèses :

- `cost` est un couple de deux coûts
- la première valeur correspond au coût pour récupérer la première ligne (souvent nul dans le cas d'un parcours séquentiel) ;
- la deuxième valeur correspond au coût pour récupérer toutes les lignes (cette valeur dépend essentiellement de la taille de la table lue, mais aussi de l'opération de filtre ici présente) ;
- `rows` correspond au nombre de lignes que le planificateur pense récupérer à la sortie de ce nœud ;
- `width` est la largeur en octets de la ligne.

Cet exemple simple permet de voir le travail de l'optimiseur :

```
=> EXPLAIN SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM employes emp
  JOIN services ser ON (emp.num_service = ser.num_service)
 WHERE ser.localisation = 'Nantes';

                              QUERY PLAN
-------------------------------------------------------------------------

 Hash Join  (cost=1.06..2.29 rows=4 width=48)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes emp  (cost=0.00..1.14 rows=14 width=35)
   ->  Hash  (cost=1.05..1.05 rows=1 width=21)
         ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
               Filter: ((localisation)::text = 'Nantes'::text)
```

Ce plan débute par la lecture de la table `services`. L'optimiseur estime que cette lecture ramènera une seule ligne (`rows=1`), que cette ligne occupera 21 octets en mémoire (`width=21`). Il s'agit de la sélectivité du filtre `WHERE localisation = 'Nantes'`. Le coût de départ de cette lecture est de 0 (`cost=0.00`). Le coût total de cette lecture est de `1.05`, qui correspond à la lecture séquentielle d'un seul bloc (donc `seq_page_cost`) et à la manipulation des 4 lignes de la tables `services` (donc 4 * `cpu_tuple_cost` + 4 * `cpu_operator_cost`). Le résultat de cette lecture est ensuite haché par le nœud `Hash`, qui précède la jointure de type `Hash Join`.

La jointure peut maintenant commencer, avec le nœud `Hash Join`. Il est particulier, car il prend 2 entrées : la donnée hachée initialement, et les données issues de la lecture d'une seconde table (peu importe le type d'accès). Le nœud a un coût de démarrage de `1.06`, soit le coût du hachage additionné au coût de manipulation du tuple de départ. Il s'agit du coût de production du premier tuple de résultat. Le coût total de production du résultat est de `2.29`. La jointure par hachage démarre réellement lorsque la lecture de la table `employes` commence. Cette lecture remontera 14 lignes, sans application de filtre. La totalité de la table est donc remontée et elle est très petite donc tient sur un seul bloc de 8 Ko. Le coût d'accès total est donc facilement déduit à partir de cette information. À partir des sélectivités précédentes, l'optimiseur estime que la jointure ramènera 4 lignes au total.

6.7.3 OPTIONS DE L'EXPLAIN

- Des options supplémentaires
  - ANALYZE
  - BUFFERS
  - COSTS
  - TIMING
  - VERBOSE
  - SUMMARY
  - FORMAT
- Donnant des informations supplémentaires très utiles

Au fil des versions, `EXPLAIN` a gagné en options. L'une d'entre elles permet de sélectionner le format en sortie. Toutes les autres permettent d'obtenir des informations supplémentaires.

Option ANALYZE

Le but de cette option est d'obtenir les informations sur l'exécution réelle de la requête.

Avec cette option, la requête est réellement exécutée. Attention aux INSERT/ UPDATE/DELETE. Pensez à les englober dans une transaction que vous annulerez après coup.

Voici un exemple utilisant cette option :

```
b1=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE c1 <1000;
                              QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
```

```
                (actual time=0.015..0.504 rows=999 loops=1)
   Filter: (c1 < 1000)
 Total runtime: 0.766 ms
(3 rows)
```

Quatre nouvelles informations apparaissent, toutes liées à l'exécution réelle de la requête :

- `actual time`
- la première valeur correspond à la durée en milliseconde pour récupérer la première ligne ;
- la deuxième valeur est la durée en milliseconde pour récupérer toutes les lignes ;
- `rows` est le nombre de lignes réellement récupérées ;
- `loops` est le nombre d'exécution de ce nœud.

Multiplier la durée par le nombre de boucles pour obtenir la durée réelle d'exécution du nœud.

L'intérêt de cette option est donc de trouver l'opération qui prend du temps dans l'exécution de la requête, mais aussi de voir les différences entre les estimations et la réalité (notamment au niveau du nombre de lignes).

Option BUFFERS

Cette option apparaît en version 9.1. Elle n'est utilisable qu'avec l'option `ANALYZE`. Elle est désactivée par défaut.

Elle indique le nombre de blocs impactés par chaque nœud du plan d'exécution, en lecture comme en écriture.

Voici un exemple de son utilisation :

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c1 <1000;
                                QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
                 (actual time=0.015..0.493 rows=999 loops=1)
   Filter: (c1 < 1000)
   Buffers: shared hit=5
 Total runtime: 0.821 ms
(4 rows)
```

La nouvelle ligne est la ligne `Buffers`. Elle peut contenir un grand nombre d'informations :

Informations	Type d'objet concerné	Explications
Shared hit	Table ou index standard	Lecture d'un bloc dans le cache

Informations	Type d'objet concerné	Explications
Shared read	Table ou index standard	Lecture d'un bloc hors du cache
Shared written	Table ou index standard	Écriture d'un bloc
Local hit	Table ou index temporaire	Lecture d'un bloc dans le cache
Local read	Table ou index temporaire	Lecture d'un bloc hors du cache
Local written	Table ou index temporaire	Écriture d'un bloc
Temp read	Tris et hachages	Lecture d'un bloc
Temp written	Tris et hachages	Écriture d'un bloc

Option COSTS

L'option `COSTS` apparaît avec la version 9.0. Une fois activée, elle indique les estimations du planificateur.

```
b1=# EXPLAIN (COSTS OFF) SELECT * FROM t1 WHERE c1 <1000;
           QUERY PLAN
--------------------------------
 Seq Scan on t1
   Filter: (c1 < 1000)
(2 rows)

b1=# EXPLAIN (COSTS ON) SELECT * FROM t1 WHERE c1 <1000;
                          QUERY PLAN
--------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
   Filter: (c1 < 1000)
(2 rows)
```

Option TIMING

Cette option n'est disponible que depuis la version 9.2. Elle n'est utilisable qu'avec l'option `ANALYZE`.

Elle ajoute les informations sur les durées en milliseconde. Elle est activée par défaut. Sa désactivation peut être utile sur certains systèmes où le chronométrage prend beaucoup de temps et allonge inutilement la durée d'exécution de la requête.

Voici un exemple de son utilisation :

```
b1=# EXPLAIN (ANALYZE,TIMING ON) SELECT * FROM t1 WHERE c1 <1000;
                                 QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8)
                 (actual time=0.017..0.520 rows=999 loops=1)
   Filter: (c1 < 1000)
```

```
   Rows Removed by Filter: 1
 Total runtime: 0.783 ms
(4 rows)

b1=# EXPLAIN (ANALYZE,TIMING OFF) SELECT * FROM t1 WHERE c1 <1000;
                                 QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..17.50 rows=1000 width=8) (actual rows=999 loops=1)
   Filter: (c1 < 1000)
   Rows Removed by Filter: 1
 Total runtime: 0.418 ms
(4 rows)
```

Option VERBOSE

L'option `VERBOSE` permet d'afficher des informations supplémentaires comme la liste des colonnes en sortie, le nom de la table qualifié du schéma, le nom de la fonction qualifié du schéma, le nom du trigger, etc. Elle est désactivée par défaut.

```
b1=# EXPLAIN (VERBOSE) SELECT * FROM t1 WHERE c1 <1000;
                               QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on public.t1  (cost=0.00..17.50 rows=1000 width=8)
   Output: c1, c2
   Filter: (t1.c1 < 1000)
(3 rows)
```

On voit dans cet exemple que le nom du schéma est ajouté au nom de la table. La nouvelle section `Output` indique la liste des colonnes de l'ensemble de données en sortie du nœud.

Option SUMMARY

Cette option apparaît en version 10. Elle permet d'afficher ou non le résumé final indiquant la durée de la planification et de l'exécution. Un `EXPLAIN` simple n'affiche pas le résumé par défaut. Par contre, un `EXPLAIN ANALYZE` l'affiche par défaut.

```
b1=# EXPLAIN SELECT * FROM t1;
                      QUERY PLAN
------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
(1 row)

b1=# EXPLAIN (SUMMARY on) SELECT * FROM t1;
                      QUERY PLAN
------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
```

```
 Planning time: 0.080 ms
(2 rows)

b1=# EXPLAIN (ANALYZE) SELECT * FROM t1;
                                            QUERY PLAN
------------------------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
            (actual time=0.004..0.004 rows=0 loops=1)
 Planning time: 0.069 ms
 Execution time: 0.037 ms
(3 rows)

b1=# EXPLAIN (ANALYZE, SUMMARY off) SELECT * FROM t1;
                                            QUERY PLAN
------------------------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
            (actual time=0.002..0.002 rows=0 loops=1)
(1 row)
```

Option FORMAT

L'option `FORMAT` apparaît en version 9.0. Elle permet de préciser le format du texte en sortie. Par défaut, il s'agit du texte habituel, mais il est possible de choisir un format balisé parmi XML, JSON et YAML. Voici ce que donne la commande `EXPLAIN` avec le format XML :

```
b1=# EXPLAIN (FORMAT XML) SELECT * FROM t1 WHERE c1 <1000;
                        QUERY PLAN
----------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">+
   <Query>                                               +
     <Plan>                                              +
       <Node-Type>Seq Scan</Node-Type>                   +
       <Relation-Name>t1</Relation-Name>                 +
       <Alias>t1</Alias>                                 +
       <Startup-Cost>0.00</Startup-Cost>                 +
       <Total-Cost>17.50</Total-Cost>                    +
       <Plan-Rows>1000</Plan-Rows>                       +
       <Plan-Width>8</Plan-Width>                        +
       <Filter>(c1 &lt; 1000)</Filter>                   +
     </Plan>                                             +
   </Query>                                              +
 </explain>
(1 row)
```

6.7.4 DÉTECTER LES PROBLÈMES

- Différence importante entre l'estimation du nombre de lignes et la réalité
- Boucles
  - appels très nombreux dans une boucle (nested loop)
  - opération lente sur lesquels PostgreSQL boucle
- Temps d'exécution conséquent sur une opération
- Opérations utilisant beaucoup de blocs (option BUFFERS)

Lorsqu'une requête s'exécute lentement, cela peut être un problème dans le plan. La sortie de `EXPLAIN` peut apporter quelques informations qu'il faut savoir décoder. Une différence importante entre le nombre de lignes estimé et le nombre de lignes réel laisse un doute sur les statistiques présentes. Soit elles n'ont pas été réactualisées récemment, soit l'échantillon n'est pas suffisamment important pour que les statistiques donnent une vue proche du réel du contenu de la table.

L'option `BUFFERS` d'`EXPLAIN` permet également de mettre en valeur les opérations d'entrées/sorties lourdes. Cette option affiche notamment le nombre de blocs lus en/hors cache de PostgreSQL, sachant qu'un bloc fait généralement 8 Ko, il est aisé de déterminer le volume de données manipulé par une requête.

6.7.5 STATISTIQUES ET COÛTS

- Détermine à partir des statistiques
  - cardinalité des prédicats
  - cardinalité des jointures
- Coût d'accès déterminé selon
  - des cardinalités
  - volumétrie des tables

Afin de comparer les différents plans d'exécution possibles pour une requête et choisir le meilleur, l'optimiseur a besoin d'estimer un coût pour chaque nœud du plan.

L'estimation la plus cruciale est celle liée aux nœuds de parcours de données, car c'est d'eux que découlera la suite du plan. Pour estimer le coût de ces nœuds, l'optimiseur s'appuie sur les informations statistiques collectées, ainsi que sur la valeur de paramètres de configuration.

Les deux notions principales de ce calcul sont la cardinalité (nombre de lignes estimées en sortie d'un nœud) et la sélectivité (fraction des lignes conservées après l'application d'un filtre).

Voici ci-dessous un exemple de calcul de cardinalité et de détermination du coût associé.

Calcul de cardinalité

Pour chaque prédicat et chaque jointure, PostgreSQL va calculer sa sélectivité et sa cardinalité. Pour un prédicat, cela permet de déterminer le nombre de lignes retournées par le prédicat par rapport au nombre total de lignes de la table. Pour une jointure, cela permet de déterminer le nombre de lignes retournées par la jointure entre deux tables.

L'optimiseur dispose de plusieurs façons de calculer la cardinalité d'un filtre ou d'une jointure selon que la valeur recherchée est une valeur unique, que la valeur se trouve dans le tableau des valeurs les plus fréquentes ou dans l'histogramme. L'exemple ci-dessous montre comment calculer la cardinalité d'un filtre simple sur une table `pays` de 25 lignes. La valeur recherchée se trouve dans le tableau des valeurs les plus fréquentes, la cardinalité peut être calculée directement. Si ce n'était pas le cas, il aurait fallu passer par l'histogramme des valeurs pour calculer d'abord la sélectivité du filtre pour en déduire ensuite la cardinalité.

Dans l'exemple qui suit, une table `pays` contient 25 entrées

La requête suivante permet de récupérer la fréquence d'apparition de la valeur recherchée dans le prédicat `WHERE region_id = 1` :

```
SELECT tablename, attname, value, freq
  FROM (SELECT tablename, attname, mcv.value, mcv.freq FROM pg_stats,
              LATERAL ROWS FROM (unnest(most_common_vals::text::int[]),
                                 unnest(most_common_freqs)) AS mcv(value, freq)
         WHERE tablename = 'pays'
           AND attname = 'region_id') get_mcv
 WHERE value = 1;
 tablename |  attname   | value | freq
-----------+------------+-------+------
 pays      | region_id |     1 |  0.2
(1 row)
```

L'optimiseur calcule la cardinalité du prédicat `WHERE region_id = 1` en multipliant cette fréquence de la valeur recherchée avec le nombre total de lignes de la table :

```
SELECT 0.2 * reltuples AS cardinalite_predicat
  FROM pg_class
 WHERE relname = 'pays';
 cardinalite_predicat
----------------------
```

```
                    5
(1 row)
```

On peut vérifier que le calcul est bon en obtenant le plan d'exécution de la requête impliquant la lecture de `pays` sur laquelle on applique le prédicat évoqué plus haut :

```
EXPLAIN SELECT * FROM pays WHERE region_id = 1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.31 rows=5 width=49)
   Filter: (region_id = 1)
(2 rows)
```

Calcul de coût

Une table `pays` peuplée de 25 lignes va permettre de montrer le calcul des coûts réalisés par l'optimiseur. L'exemple présenté ci-dessous est simplifié. En réalité, les calculs sont plus complexes car ils tiennent également compte de la volumétrie réelle de la table.

Le coût de la lecture séquentielle de la table `pays` est calculé à partir de deux composantes. Toute d'abord, le nombre de pages (ou blocs) de la table permet de déduire le nombre de blocs à accéder pour lire la table intégralement. Le paramètre `seq_page_cost` sera appliqué ensuite pour indiquer le coût de l'opération :

```
SELECT relname, relpages * current_setting('seq_page_cost')::float AS cout_acces
  FROM pg_class
 WHERE relname = 'pays';
 relname | cout_acces
---------+------------
 pays    |          1
```

Cependant, le coût d'accès seul ne représente pas le coût de la lecture des données. Une fois que le bloc est monté en mémoire, PostgreSQL doit décoder chaque ligne individuellement. L'optimiseur utilise `cpu_tuple_cost` pour estimer le coût de manipulation des lignes :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float AS cout
  FROM pg_class
 WHERE relname = 'pays';
 relname | cout
---------+------
 pays    | 1.25
```

On peut vérifier que le calcul est bon :

```
EXPLAIN SELECT * FROM pays;
                      QUERY PLAN
```

```
------------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.25 rows=25 width=53)
(1 ligne)
```

Si l'on applique un filtre à la requête, les traitements seront plus lourds. Par exemple, en ajoutant le prédicat `WHERE pays = 'FR'`.

Il faut non seulement extraire les lignes les unes après les autres, mais il faut également appliquer l'opérateur de comparaison utilisé. L'optimiseur utilise le paramètre `cpu_operator_cost` pour déterminer le coût d'application d'un filtre :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float
       + reltuples * current_setting('cpu_operator_cost')::float AS cost
  FROM pg_class
 WHERE relname = 'pays';
 relname |  cost
---------+--------
 pays    | 1.3125
```

En récupérant le plan d'exécution de la requête à laquelle est appliqué le filtre `WHERE pays = 'FR'`, on s'aperçoit que le calcul est juste, à l'arrondi près :

```
EXPLAIN SELECT * FROM pays WHERE code_pays = 'FR';
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.31 rows=1 width=53)
   Filter: (code_pays = 'FR'::text)
(2 lignes)
```

Pour aller plus loin dans le calcul de sélectivité, de cardinalité et de coût, la documentation de PostgreSQL montre un exemple complet de calcul de sélectivité et indique les références des fichiers sources dans lesquels fouiller pour en savoir plus :Comment le planificateur utilise les statistiques[14] .

6.8 NŒUDS D'EXÉCUTION LES PLUS COURANTS

- Un plan est composé de nœuds
  - certains produisent des données
  - d'autres consomment des données et les retournent
  - le nœud final retourne les données à l'utilisateur

[14] http://docs.postgresql.fr/current/planner-stats-details.html

- chaque nœud consomme au fur et à mesure les données produites par les nœuds parents

---

6.8.1 NOEUDS DE TYPE PARCOURS

- Seq Scan
- Parallel Seq Scan
- Function Scan
- et des parcours d'index

Les parcours sont les seules opérations qui lisent les données des tables (normales, temporaires ou non journalisées). Elles ne prennent donc rien en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

Il existe plusieurs types de parcours possibles. Parmi les plus fréquents, on retrouve :

- le parcours de table ;
- le parcours de fonction ;
- les parcours d'index.

Depuis la version 9.6, les parcours de table sont parallélisables.

Les parcours d'index sont documentés par la suite.

L'opération `Seq Scan` correspond à une lecture séquentielle d'une table, aussi appelée `Full Table Scan` sur d'autres SGBD. Il consiste à lire l'intégralité de la table, du premier bloc au dernier bloc. Une clause de filtrage peut être appliquée.

On retrouve ce noeud lorsque la requête nécessite de lire l'intégralité de la table :

```
cave=# EXPLAIN SELECT * FROM region;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on region  (cost=0.00..1.19 rows=19 width=15)
```

Ce noeud peut également filtrer directement les données, la présence de la clause `Filter` montre le filtre appliqué à la lecture des données :

```
cave=# EXPLAIN SELECT * FROM region WHERE id=5;
                      QUERY PLAN
------------------------------------------------------
 Seq Scan on region  (cost=0.00..1.24 rows=1 width=15)
   Filter: (id = 5)
```

Le coût d'accès pour ce type de noeud sera dépendant du nombre de blocs à parcourir et du paramètre `seq_page_cost`.

Il est possible d'avoir un parcours parallélisé d'une table sous certaines conditions (la première étant qu'il faut avoir au minimum une version 9.6). Pour que ce type de parcours soit valable, il faut que l'optimiseur soit persuadé que le problème sera le temps CPU et non la bande passante disque. Autrement dit, dans la majorité des cas, il faut un filtre pour que la parallélisation se déclenche et il faut que la table soit suffisamment volumineuse.

```
postgres=# CREATE TABLE t20 AS SELECT id FROM generate_series(1, 1000000) g(id);
postgres=# SET max_parallel_workers_per_gather TO 6;
postgres=# EXPLAIN SELECT * FROM t20 WHERE id<10000;
                              QUERY PLAN
------------------------------------------------------------------------
 Gather  (cost=1000.00..11676.13 rows=10428 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t20  (cost=0.00..9633.33 rows=4345 width=4)
         Filter: (id < 10000)
(4 rows)
```

Ici, deux processus supplémentaires seront exécutés pour réaliser la requête. Dans le cas de ce type de parcours, chaque processus traite toutes les lignes d'un bloc. Enfin quand un processus a terminé de traiter son bloc, il regarde quel est le prochain bloc à traiter et le traite.

On retrouve le noeud `Function Scan` lorsqu'une requête utilise directement le résultat d'une fonction. C'est un noeud que l'on rencontre lorsqu'on utilise les fonctions d'informations systèmes de PostgreSQL :

```
postgres=# EXPLAIN SELECT * from pg_get_keywords();
                              QUERY PLAN
-----------------------------------------------------------------------
 Function Scan on pg_get_keywords  (cost=0.03..4.03 rows=400 width=65)
(1 ligne)
```

En dehors des différents parcours d'index, on retrouve également d'autres types de parcours, mais PostgreSQL les utilise rarement. Ils sont néanmoins détaillés en annexe.

6.8.2 PARCOURS D'INDEX

- Index Scan
- Index Only Scan
- Bitmap Index Scan
- Et leurs versions parallélisées

PostgreSQL dispose de trois moyens d'accéder aux données à travers les index.

Le noeud `Index Scan` est le premier qui a été disponible. Il consiste à parcourir les blocs d'index jusqu'à trouver les pointeurs vers les blocs contenant les données. PostgreSQL lit ensuite les données de la table qui sont pointées par l'index.

```
tpc=# EXPLAIN SELECT * FROM clients WHERE client_id = 10000;
                                      QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using clients_pkey on clients  (cost=0.42..8.44 rows=1 width=52)
   Index Cond: (client_id = 10000)
(2 lignes)
```

Ce type de noeud ne permet pas d'extraire directement les données à retourner depuis l'index, sans passer par la lecture des blocs correspondants de la table. Le noeud `Index Only Scan` permet cette optimisation, à condition que les colonnes retournées fassent partie de l'index :

```
tpc=# EXPLAIN SELECT client_id FROM clients WHERE client_id = 10000;
                                        QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using clients_pkey on clients  (cost=0.42..8.44 rows=1 width=8)
   Index Cond: (client_id = 10000)
(2 lignes)
```

Enfin, on retrouve le dernier parcours sur des opérations de type *range scan*, c'est-à-dire où PostgreSQL doit retourner une plage de valeurs. On le retrouve également lorsque PostgreSQL doit combiner le résultat de la lecture de plusieurs index.

Contrairement à d'autres SGBD, un index bitmap n'a aucune existence sur disque. Il est créé en mémoire lorsque son utilisation a un intérêt. Le but est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table :

- Lecture en un bloc de l'index ;
- Lecture en un bloc de la partie intéressante de la table (dans l'ordre physique de la table, pas dans l'ordre logique de l'index).

```
tpc=# EXPLAIN SELECT * FROM clients WHERE client_id BETWEEN 10000 AND 12000;
                                       QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=44.99..1201.32 rows=2007 width=52)
   Recheck Cond: ((client_id >= 10000) AND (client_id <= 12000))
   ->  Bitmap Index Scan on clients_pkey  (cost=0.00..44.49 rows=2007 width=0)
         Index Cond: ((client_id >= 10000) AND (client_id <= 12000))
(4 lignes)
```

On retrouve aussi des Bitmap Index Scan lorsqu'il s'agit de combiner le résultat de la lecture de plusieurs index :

```
tpc=# EXPLAIN SELECT * FROM clients WHERE client_id
tpc=# BETWEEN 10000 AND 12000 AND segment_marche = 'AUTOMOBILE';
                                       QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=478.25..1079.74 rows=251 width=8)
   Recheck Cond: ((client_id >= 10000) AND (client_id <= 12000)
                 AND (segment_marche = 'AUTOMOBILE'::bpchar))
   ->  BitmapAnd  (cost=478.25..478.25 rows=251 width=0)
     ->  Bitmap Index Scan on clients_pkey  (cost=0.00..44.49 rows=2007 width=0)
           Index Cond: ((client_id >= 10000) AND (client_id <= 12000))
     ->  Bitmap Index Scan on idx_clients_segmarche
                                       (cost=0.00..433.38 rows=18795 width=0)
           Index Cond: (segment_marche = 'AUTOMOBILE'::bpchar)
(7 lignes)
```

À partir de la version 10, une infrastructure a été mise en place pour permettre un parcours parallélisé d'un index. Cela donne donc les noeuds `Parallel Index Scan`, `Parallel Index Only Scan` et `Parallel Bitmap Heap Scan`. Cette infrastructure est actuellement uniquement utilisé pour les index Btree. Par contre, pour le bitmap scan, seul le parcours de la table est parallélisé, ce qui fait que tous les types d'index sont concernés.

6.8.3 NOEUDS DE JOINTURE

- PostgreSQL implémente les 3 algorithmes de jointures habituels :
  - Nested Loop (boucle imbriquée)
  - Hash Join (hachage de la table interne)
  - Merge Join (tri-fusion)
- Parallélisation possible
  - version 9.6 pour Nested Loop et Hash Join
  - version 10 pour Merge Join
- Et pour `EXISTS`, `IN` et certaines jointures externes :
  - Semi Join et Anti Join

Le choix du type de jointure dépend non seulement des données mises en oeuvre, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment des paramètres `work_mem`, `seq_page_cost` et `random_page_cost`.

La `Nested Loop` se retrouve principalement quand on joint de petits ensembles de don-

nées :

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande)
sql1=# WHERE numero_commande < 1000;
                                        QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.84..4161.14 rows=1121 width=154)
   ->  Index Scan using orders_pkey on commandes
                                          (cost=0.42..29.64 rows=280 width=80)
         Index Cond: (numero_commande < 1000)
   ->  Index Scan using lignes_commandes_pkey on lignes_commandes
                                          (cost=0.42..14.71 rows=5 width=82)
         Index Cond: (numero_commande = commandes.numero_commande)
```

Le `Hash Join` se retrouve également lorsque l'ensemble de la table interne est très petit. L'optimiseur réalise alors un hachage des valeurs de la colonne de jointure sur la table externe et réalise ensuite une lecture de la table externe et compare les hachages de la clé de jointure avec le/les hachage(s) obtenus à la lecture de la table interne.

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande);
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=10690.31..59899.18 rows=667439 width=154)
   Hash Cond: (lignes_commandes.numero_commande = commandes.numero_commande)
   ->  Seq Scan on lignes_commandes  (cost=0.00..16325.39 rows=667439 width=82)
   ->  Hash  (cost=6489.25..6489.25 rows=166725 width=80)
         ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
```

La jointure par tri- fusion, ou `Merge Join` prend deux ensembles de données triés en entrée et restitue l'ensemble de données après jointure. Cette jointure est assez lourde à initialiser si PostgreSQL ne peut pas utiliser d'index, mais elle a l'avantage de retourner les données triées directement :

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande)
sql1=# ORDER BY numero_commande DESC;
                                        QUERY PLAN
------------------------------------------------------------------------------
 Merge Join  (cost=1.40..64405.98 rows=667439 width=154)
   Merge Cond: (commandes.numero_commande = lignes_commandes.numero_commande)
   ->  Index Scan Backward using orders_pkey on commandes
                                  (cost=0.42..12898.63 rows=166725 width=80)
   ->  Index Scan Backward using lignes_commandes_pkey on lignes_commandes
                                  (cost=0.42..42747.64 rows=667439 width=82)
```

Il s'agit d'un algorithme de jointure particulièrement efficace pour traiter les volumes de

données importants.

Les clauses EXISTS et NOT EXISTS mettent également en oeuvre des algorithmes dérivés de semi et anti jointures. Par exemple avec la clause EXISTS :

```
sql1=# EXPLAIN
 SELECT count(*)
   FROM commandes c
  WHERE EXISTS (SELECT 1
                  FROM lignes_commandes l
                 WHERE c.date_commande > l.date_expedition
                   AND c.numero_commande = l.numero_commande);
                              QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=42439.18..42439.19 rows=1 width=0)
   ->  Hash Semi Join  (cost=27927.38..42321.76 rows=46967 width=0)
         Hash Cond: (c.numero_commande = l.numero_commande)
         Join Filter: (c.date_commande > l.date_expedition)
         ->  Seq Scan on commandes c (cost=0.00..6489.25 rows=166725 width=12)
         ->  Hash  (cost=16325.39..16325.39 rows=667439 width=12)
               ->  Seq Scan on lignes_commandes l
                                 (cost=0.00..16325.39 rows=667439 width=12)
```

On obtient un plan sensiblement identique, avec NOT EXISTS. Le noeud Hash Semi Join est remplacé par Hash Anti Join :

```
sql1=# EXPLAIN
SELECT *
  FROM commandes
 WHERE NOT EXISTS (SELECT 1
                     FROM lignes_commandes l
                    WHERE l.numero_commande = commandes.numero_commande);
                                    QUERY PLAN
------------------------------------------------------------------------------
 Hash Anti Join  (cost=27276.38..47110.99 rows=25824 width=80)
   Hash Cond: (commandes.numero_commande = l.numero_commande)
   ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
   ->  Hash  (cost=16325.39..16325.39 rows=667439 width=8)
         ->  Seq Scan on lignes_commandes l
                          (cost=0.00..16325.39 rows=667439 width=8)
```

PostgreSQL dispose de la parallélisation depuis la version 9.6. Cela ne concernait que les jointures de type Nested Loop et Hash Join. Quant au Merge Join, il a fallu attendre la version 10 pour que la parallélisation soit supportée.

6.8.4 NOEUDS DE TRIS ET DE REGROUPEMENTS

- Un seul noeud de tri :
  - Sort
- Regroupement/Agrégation :
  - Aggregate
  - HashAggregate
  - GroupAggregate
  - Partial Aggregate/Finalize Aggregate

Pour réaliser un tri, PostgreSQL ne dispose que d'un seul noeud pour réaliser cela : `Sort`. Son efficacité va dépendre du paramètre `work_mem` qui va définir la quantité de mémoire que PostgreSQL pourra utiliser pour un tri.

```
sql1=# explain (ANALYZE) SELECT * FROM lignes_commandes
sql1=# WHERE numero_commande = 1000 ORDER BY quantite;
                                    QUERY PLAN
------------------------------------------------------------------------------
 Sort  (cost=15.57..15.58 rows=5 width=82)
       (actual time=0.096..0.097 rows=4 loops=1)
   Sort Key: quantite
   Sort Method: quicksort  Memory: 25kB
   ->  Index Scan using lignes_commandes_pkey on lignes_commande
                    (cost=0.42..15.51 rows=5 width=82)
                    (actual time=0.017..0.021 rows=4 loops=1)
         Index Cond: (numero_commande = 1000)
```

Si le tri ne tient pas en mémoire, l'algorithme de tri gère automatiquement le débordement sur disque :

```
sql1=# EXPLAIN (ANALYZE) SELECT * FROM commandes ORDER BY prix_total ;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Sort  (cost=28359.74..28776.55 rows=166725 width=80)
       (actual time=993.441..1157.935 rows=166725 loops=1)
   Sort Key: prix_total
   Sort Method: external merge  Disk: 15608kB
   ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
                              (actual time=173.615..236.712 rows=166725 loops=1)
```

Cependant, si un index existe, PostgreSQL peut également utiliser un index pour récupérer les données triées directement :

```
sql1=# EXPLAIN SELECT * FROM commandes ORDER BY date_commande;
                                    QUERY PLAN
-------------------------------------------------------------------------------
```

```
 Index Scan using idx_commandes_date_commande on commandes
                             (cost=0.42..23628.15 rows=166725 width=80)
```

Dans n'importe quel ordre de tri :

```
sql1=# EXPLAIN SELECT * FROM commandes ORDER BY date_commande DESC;
                                        QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan Backward using idx_commandes_date_commande on commandes
                                     (cost=0.42..23628.15 rows=166725 width=80)
```

Le choix du type d'opération de regroupement dépend non seulement des données mises en oeuvres, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment du paramètre `work_mem`.

Concernant les opérations d'agrégations, on retrouve un noeud de type `Aggregate` lorsque la requête réalise une opération d'agrégation simple, sans regroupement :

```
sql1=# EXPLAIN SELECT count(*) FROM commandes;
                                 QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=4758.11..4758.12 rows=1 width=0)
   ->  Index Only Scan using commandes_client_id_idx on commandes
                                     (cost=0.42..4341.30 rows=166725 width=0)
```

Si l'optimiseur estime que l'opération d'agrégation tient en mémoire (paramètre `work_mem`), il va utiliser un noeud de type HashAggregate :

```
sql1=# EXPLAIN SELECT code_pays, count(*) FROM contacts GROUP BY code_pays;
                              QUERY PLAN
----------------------------------------------------------------------
 HashAggregate  (cost=3982.02..3982.27 rows=25 width=3)
   ->  Seq Scan on contacts  (cost=0.00..3182.01 rows=160001 width=3)
```

L'inconvénient de ce noeud est que sa consommation mémoire n'est pas limitée par `work_mem`, il continuera malgré tout à allouer de la mémoire. Dans certains cas, heureusement très rares, l'optimiseur peut se tromper suffisamment pour qu'un noeud `HashAggregate` consomme plusieurs giga-octets de mémoire et ne sature la mémoire du serveur.

Lorsque l'optimiseur estime que le volume de données à traiter ne tient pas dans `work_mem`, il utilise plutôt l'algorithme `GroupAggregate` :

```
sql1=# explain select numero_commande, count(*)
sql1=# FROM lignes_commandes group by numero_commande;
                                    QUERY PLAN
------------------------------------------------------------------------------
 GroupAggregate  (cost=0.42..47493.84 rows=140901 width=8)
```

```
   ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                                                  (cost=0.42..42747.64 rows=667439 width=8)
```

Le calcul d' un agrégat peut être parallélisé à partir de la version 9.6. Dans ce cas, deux noeuds sont utilisés : un pour le calcul partiel de chaque processus (Partial Aggregate), et un pour le calcul final (Finalize Aggregate). Voici un exemple de plan :

```
SELECT count(*), min(C1), max(C1) FROM t1;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate (actual time=1766.820..1766.820 rows=1 loops=1)
   ->  Gather (actual time=1766.767..1766.799 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (actual time=1765.236..1765.236 rows=1 loops=3)
               ->  Parallel Seq Scan on t1
                            (actual time=0.021..862.430 rows=6666667 loops=3)
 Planning time: 0.072 ms
 Execution time: 1769.164 ms
(8 rows)
```

6.8.5 LES AUTRES NOEUDS

- Limit
- Unique
- Append (UNION ALL), Except, Intersect
- Gather
- InitPlan, Subplan, etc.

On rencontre le noeud `Limit` lorsqu'on limite le résultat avec l'ordre `LIMIT` :

```
sql1=# EXPLAIN SELECT * FROM commandes LIMIT 1;
                               QUERY PLAN
------------------------------------------------------------------------
 Limit  (cost=0.00..0.04 rows=1 width=80)
   ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
```

À noter, que le noeud `Sort` utilisera une méthode de tri appelée `top-N heapsort` qui permet d'optimiser le tri pour retourner les *n* premières lignes :

```
sql1=# EXPLAIN ANALYZE SELECT * FROM commandes ORDER BY prix_total LIMIT 5;
                                       QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=9258.49..9258.50 rows=5 width=80)
```

```
        (actual time=86.332..86.333 rows=5 loops=1)
   ->  Sort  (cost=9258.49..9675.30 rows=166725 width=80)
             (actual time=86.330..86.331 rows=5 loops=1)
         Sort Key: prix_total
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on commandes  (cost=0.00..6489.25 rows=166725 width=80)
                                (actual time=3.683..22.687 rows=166725 loops=1)
```

On retrouve le noeud `Unique` lorsque l'on utilise `DISTINCT` pour dédoublonner le résultat d'une requête :

```
sql1=# EXPLAIN SELECT DISTINCT numero_commande FROM lignes_commandes;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Unique  (cost=0.42..44416.23 rows=140901 width=8)
   ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                                    (cost=0.42..42747.64 rows=667439 width=8)
```

À noter qu'il est souvent plus efficace d'utiliser `GROUP BY` pour dédoublonner les résultats d'une requête :

```
sql1=# EXPLAIN (ANALYZE) SELECT DISTINCT numero_commande
sql1=# FROM lignes_commandes GROUP BY numero_commande;
                                      QUERY PLAN
-------------------------------------------------------------------------------
 Unique  (cost=0.42..44768.49 rows=140901 width=8)
         (actual time=0.047..357.745 rows=166724 loops=1)
   ->  Group  (cost=0.42..44416.23 rows=140901 width=8)
              (actual time=0.045..306.550 rows=166724 loops=1)
         ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                    (cost=0.42..42747.64 rows=667439 width=8)
                    (actual time=0.040..197.817 rows=667439 loops=1)
               Heap Fetches: 667439
 Total runtime: 365.315 ms

sql1=# EXPLAIN (ANALYZE) SELECT numero_commande
sql1=# FROM lignes_commandes GROUP BY numero_commande;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Group  (cost=0.42..44416.23 rows=140901 width=8)
        (actual time=0.053..302.875 rows=166724 loops=1)
   ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                (cost=0.42..42747.64 rows=667439 width=8)
                (actual time=0.046..194.495 rows=667439 loops=1)
         Heap Fetches: 667439
 Total runtime: 310.506 ms
```

Le gain est infime, 50 millisecondes environ sur cette requête, mais laisse présager des

gains sur une volumétrie plus importante.

Les noeuds `Append`, `Except` et `Intersect` se rencontrent lorsqu'on utilise les opérateurs ensemblistes `UNION`, `EXCEPT` et `INTERSECT`. Par exemple, avec `UNION ALL` :

```
sql1=# EXPLAIN
SELECT * FROM pays
 WHERE region_id = 1
 UNION ALL
SELECT * FROM pays
 WHERE region_id = 2;
                              QUERY PLAN
-------------------------------------------------------------------
 Append  (cost=0.00..2.73 rows=10 width=53)
   ->  Seq Scan on pays  (cost=0.00..1.31 rows=5 width=53)
         Filter: (region_id = 1)
   ->  Seq Scan on pays pays_1  (cost=0.00..1.31 rows=5 width=53)
         Filter: (region_id = 2)
```

Le noeud Gather a été introduit en version 9.6 et est utilisé comme noeud de rassemblement des données pour les plans parallélisés.

Le noeud InitPlan apparaît lorsque PostgreSQL a besoin d'exécuter une première sous-requête pour ensuite exécuter le reste de la requête. Il est assez rare :

```
sql1=# EXPLAIN SELECT *,
sql1=# (SELECT nom_region FROM regions WHERE region_id=1)
sql1=# FROM pays WHERE region_id = 1;
                             QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on pays  (cost=1.06..2.38 rows=5 width=53)
   Filter: (region_id = 1)
   InitPlan 1 (returns $0)
     ->  Seq Scan on regions  (cost=0.00..1.06 rows=1 width=26)
           Filter: (region_id = 1)
```

Le noeud SubPlan est utilisé lorsque PostgreSQL a besoin d'exécuter une sous-requête pour filtrer les données :

```
sql1=# EXPLAIN
SELECT * FROM pays
 WHERE region_id NOT IN (SELECT region_id FROM regions
                          WHERE nom_region = 'Europe');
                          QUERY PLAN
----------------------------------------------------------------
 Seq Scan on pays  (cost=1.06..2.38 rows=12 width=53)
   Filter: (NOT (hashed SubPlan 1))
   SubPlan 1
```

```
->  Seq Scan on regions  (cost=0.00..1.06 rows=1 width=4)
      Filter: (nom_region = 'Europe'::bpchar)
```

D'autres types de noeud peuvent également être trouvés dans les plans d'exécution. L'annexe décrit tous ces noeuds en détail.

6.9 PROBLÈMES LES PLUS COURANTS

- L'optimiseur se trompe parfois
  - mauvaises statistiques
  - écriture particulière de la requête
  - problèmes connus de l'optimiseur

L'optimiseur de PostgreSQL est sans doute la partie la plus complexe de PostgreSQL. Il se trompe rarement, mais certains facteurs peuvent entraîner des temps d'exécution très lents, voire catastrophiques de certaines requêtes.

6.9.1 COLONNES CORRÉLÉES

```
SELECT * FROM t1 WHERE c1=1 AND c2=1
```

- `c1=1` est vrai pour 20% des lignes
- `c2=1` est vrai pour 10% des lignes
- Le planificateur va penser que le résultat complet ne récupérera que 20% * 10% (soit 2%) des lignes
  - En réalité, ça peut aller de 0 à 10% des lignes
- Problème corrigé en version 10
  - `CREATE STATISTICS` pour des statistiques multi-colonnes

PostgreSQL conserve des statistiques par colonne simple. Dans l'exemple ci- dessus, le planificateur sait que l'estimation pour `c1=1` est de 20% et que l'estimation pour `c2=1` est de 10%. Par contre, il n'a aucune idée de l'estimation pour `c1=1 AND c2=1`. En réalité, l'estimation pour cette formule va de 0 à 10% mais le planificateur doit statuer sur une seule valeur. Ce sera le résultat de la multiplication des deux estimations, soit 2% (20% * 10%).

La version 10 de PostgreSQL corrige cela en ajoutant la possibilité d'ajouter des statistiques sur plusieurs colonnes spécifiques. Ce n'est pas automatique, il faut créer un objet statistique avec l'ordre `CREATE STATISTICS`.

6.9.2 MAUVAISE ÉCRITURE DE PRÉDICATS

```
SELECT *
FROM commandes
WHERE extract('year' from date_commande) = 2014;
```

- L'optimiseur n'a pas de statistiques sur le résultat de la fonction `extract`
  - il estime la sélectivité du prédicat à 0.5%.

Dans un prédicat, lorsque les valeurs des colonnes sont transformées par un calcul ou par une fonction, l'optimiseur n'a aucun moyen pour connaître la sélectivité d'un prédicat. Il utilise donc une estimation codée en dur dans le code de l'optimiseur : 0,5% du nombre de lignes de la table.

Dans la requête suivante, l'optimiseur estime que la requête va ramener 834 lignes :

```
sql1=# EXPLAIN SELECT * FROM commandes
sql1=# WHERE extract('year' from date_commande) = 2014;
                                            QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..7739.69 rows=834 width=80)
   Filter:
    (date_part('year'::text, (date_commande)::timestamp without time zone) =
        2014::double precision)
(2 lignes)
```

Ces 834 lignes correspondent à 0,5% de la table `commandes` :

```
sql1=# SELECT relname, reltuples, round(reltuples*0.005) AS estimé
  FROM pg_class
 WHERE relname = 'commandes';
  relname   | reltuples | estimé
-----------+-----------+--------
 commandes |    166725 |    834
(1 ligne)
```

6.9.3 PROBLÈME AVEC LIKE

```
SELECT * FROM t1 WHERE c2 LIKE 'x%';
```

- PostgreSQL peut utiliser un index dans ce cas
- Si l'encodage n'est pas C, il faut déclarer l'index avec une classe d'opérateur
  - varchar_pattern_ops, text_pattern_ops, etc

- En 9.1, il faut aussi faire attention au collationnement
- Ne pas oublier pg_trgm (surtout en 9.1) et FTS

Dans le cas d'une recherche avec préfixe, PostgreSQL peut utiliser un index sur la colonne. Il existe cependant une spécificité à PostgreSQL. Si l'encodage est autre chose que C, il faut utiliser une classe d'opérateur lors de la création de l'index. Cela donnera par exemple :

```
CREATE INDEX i1 ON t1 (c2 varchar_pattern_ops);
```

De plus, à partir de la version 9.1, il est important de faire attention au collationnement. Si le collationnement de la requête diffère du collationnement de la colonne de l'index, l'index ne pourra pas être utilisé.

6.9.4 PROBLÈMES AVEC LIMIT

- Exemple
  - EXPLAIN avec LIMIT 199
  - EXPLAIN avec LIMIT 200
- Corrigé en 9.2

Le contexte :

```
CREATE TABLE t1 (
    c1 integer PRIMARY KEY
);
INSERT INTO t1 SELECT generate_series(1, 1000);

CREATE TABLE t2 (
    c2 integer
);
INSERT INTO t2 SELECT generate_series(1, 1000);

ANALYZE;
```

Voici un problème survenant dans les versions antérieures à la 9.2.

```
EXPLAIN SELECT * FROM t1
WHERE c1 IN (SELECT c2 FROM t2 LIMIT 199);
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Semi Join (cost=7.46..27.30 rows=199 width=4)
   Hash Cond: (t1.c1 = t2.c2)
   -> Seq Scan on t1 (cost=0.00..15.00 rows=1000 width=4)
   -> Hash (cost=4.97..4.97 rows=199 width=4)
```

```
          -> Limit (cost=0.00..2.98 rows=199 width=4)
                -> Seq Scan on t2 (cost=0.00..15.00 rows=1000 width=4)
(6 rows)
```

Tout se passe bien. PostgreSQL fait tout d'abord un parcours séquentiel sur la table `t2` et ne récupère que les 199 premières lignes grâce à la clause `LIMIT`. Le hachage se fait sur les 199 lignes et une comparaison est faite pour chaque ligne de `t1`.

Maintenant, cherchons à récupérer une ligne de plus avec un `LIMIT` à 200 :

```
EXPLAIN SELECT * FROM t1
WHERE c1 IN (SELECT c2 FROM t2 LIMIT 200);
                                  QUERY PLAN
----------------------------------------------------------------------
 Hash Join (cost=10.00..30.75 rows=500 width=4)
   Hash Cond: (t1.c1 = t2.c2)
   -> Seq Scan on t1 (cost=0.00..15.00 rows=1000 width=4)
   -> Hash (cost=7.50..7.50 rows=200 width=4)
         -> HashAggregate (cost=5.50..7.50 rows=200 width=4)
               -> Limit (cost=0.00..3.00 rows=200 width=4)
                     -> Seq Scan on t2 (cost=0.00..15.00 rows=1000
                                   width=4)
(7 rows)
```

La requête a légèrement changé : on passe d'un `LIMIT 199` à un `LIMIT 200`. L'estimation explose, elle passe de 199 lignes (estimation exacte) à 500 lignes (estimation plus que doublée). En fait, le nombre de lignes est calculé très simplement : nombre de lignes de la table t1 multiplié par 0,5. C'est codé en dur. La raison, jusqu'à PostgreSQL 9.1, est que par défaut une table sans statistiques est estimée posséder 200 valeurs distinctes. Quand l'optimiseur rencontre donc 200 enregistrements distincts en estimation, il pense que la fonction d'estimation de valeurs distinctes n'a pas de statistiques et lui a retourné une valeur par défaut, et applique donc un algorithme de sélectivité par défaut, au lieu de l'algorithme plus fin utilisé en temps normal.

Sur cet exemple, cela n'a pas un gros impact vu la quantité de données impliquées et le schéma choisi. Par contre, ça fait passer une requête de 9ms à 527ms si le `LIMIT 199` est passé à un `LIMIT 200` pour la même requête sur une table plus conséquente.

Ce problème est réglé en version 9.2 :

```
EXPLAIN SELECT * FROM t1
WHERE c1 IN (SELECT c2 FROM t2 LIMIT 200);
                               QUERY PLAN
------------------------------------------------------------------------
 Hash Semi Join (cost=7.46..27.30 rows=200 width=4)
   Hash Cond: (t1.c1 = t2.c2)
   -> Seq Scan on t1 (cost=0.00..15.00 rows=1000 width=4)
```

```
   -> Hash (cost=4.97..4.97 rows=200 width=4)
         -> Limit (cost=0.00..2.98 rows=200 width=4)
               -> Seq Scan on t2 (cost=0.00..15.00 rows=1000 width=4)
(6 rows)
```

6.9.5 DELETE LENT

- DELETE lent
- Généralement un problème de clé étrangère

```
Delete  (actual time=111.251..111.251 rows=0 loops=1)
  ->  Hash Join  (actual time=1.094..21.402 rows=9347 loops=1)
        ->  Seq Scan on lot_a30_descr_lot
            (actual time=0.007..11.248 rows=34934 loops=1)
        ->  Hash  (actual time=0.501..0.501 rows=561 loops=1)
              ->  Bitmap Heap Scan on lot_a10_pdl
                  (actual time=0.121..0.326 rows=561 loops=1)
                    Recheck Cond: (id_fantoir_commune = 320013)
                    ->  Bitmap Index Scan on...
                        (actual time=0.101..0.101 rows=561 loops=1)
                          Index Cond: (id_fantoir_commune = 320013)
Trigger for constraint fk_lotlocal_lota30descrlot:
  time=1010.358 calls=9347
Trigger for constraint fk_nonbatia21descrsuf_lota30descrlot:
  time=2311695.025 calls=9347
Total runtime: 2312835.032 ms
```

Parfois, un DELETE peut prendre beaucoup de temps à s'exécuter. Cela peut être dû à un grand nombre de lignes à supprimer. Cela peut aussi être dû à la vérification des contraintes étrangères.

Dans l'exemple ci-dessus, le DELETE met 38 minutes à s'exécuter (2312835 ms), pour ne supprimer aucune ligne. En fait, c'est la vérification de la contrainte fk_nonbatia21descrsuf_lota30descrlot qui prend pratiquement tout le temps. C'est d'ailleurs pour cette raison qu'il est recommandé de positionner des index sur les clés étrangères, car cet index permet d'accélérer la recherche liée à la contrainte.

Attention donc aux contraintes de clés étrangères pour les instructions DML.

6.9.6 DÉDOUBLONNAGE

```
SELECT DISTINCT t1.* FROM t1 JOIN t2 ON (t1.id=t2.t1_id);
```

- DISTINCT est souvent utilisé pour dédoublonner les lignes de t1
  - mais génère un tri qui pénalise les performances
- GROUP BY est plus rapide
- Une clé primaire permet de dédoublonner efficacement des lignes
  - à utiliser avec GROUP BY

L'exemple ci-dessous montre une requête qui récupère les commandes qui ont des lignes de commandes et réalise le dédoublonnage avec DISTINCT. Le plan d'exécution montre une opération de tri qui a nécessité un fichier temporaire de 60Mo. Toutes ces opérations sont assez gourmandes, la requête répond en 5,9s :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
tpc-# SELECT DISTINCT commandes.* FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande);
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Unique (actual time=5146.904..5833.600 rows=168749 loops=1)
   -> Sort (actual time=5146.902..5307.633 rows=675543 loops=1)
        Sort Key: commandes.numero_commande, commandes.client_id,
                  commandes.etat_commande, commandes.prix_total,
                  commandes.date_commande, commandes.priorite_commande,
                  commandes.vendeur, commandes.priorite_expedition,
                  commandes.commentaire
        Sort Method: external sort  Disk: 60760kB
        -> Merge Join (actual time=0.061..601.674 rows=675543 loops=1)
              Merge Cond: (commandes.numero_commande =
                           lignes_commandes.numero_commande)
              -> Index Scan using orders_pkey on commandes
                     (actual time=0.026..71.544 rows=168750 loops=1)
              -> Index Only Scan using lignes_com_pkey on lignes_commandes
                     (actual time=0.025..175.321 rows=675543 loops=1)
                    Heap Fetches: 0
 Total runtime: 5849.996 ms
```

En restreignant les colonnes récupérées à celle réellement intéressante et en utilisant GROUP BY au lieu du DISTINCT, le temps d'exécution tombe à 4,5s :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
 FROM commandes
 JOIN lignes_commandes
      USING (numero_commande)
 GROUP BY commandes.numero_commande, commandes.etat_commande,
```

```
          commandes.prix_total, commandes.date_commande,
          commandes.priorite_commande, commandes.vendeur,
          commandes.priorite_expedition;
                                        QUERY PLAN
-----------------------------------------------------------------------------
 Group (actual time=4025.069..4663.992 rows=168749 loops=1)
   -> Sort (actual time=4025.065..4191.820 rows=675543 loops=1)
         Sort Key: commandes.numero_commande, commandes.etat_commande,
                   commandes.prix_total, commandes.date_commande,
                   commandes.priorite_commande, commandes.vendeur,
                   commandes.priorite_expedition
         Sort Method: external sort  Disk: 46232kB
         -> Merge Join (actual time=0.062..579.852 rows=675543 loops=1)
               Merge Cond: (commandes.numero_commande =
                            lignes_commandes.numero_commande)
               -> Index Scan using orders_pkey on commandes
                            (actual time=0.027..70.212 rows=168750 loops=1)
               -> Index Only Scan using lignes_com_pkey on lignes_commandes
                            (actual time=0.026..170.555 rows=675543 loops=1)
                     Heap Fetches: 0
 Total runtime: 4676.829 ms
```

Mais, à partir de PostgreSQL 9.1, il est possible d'améliorer encore les temps d'exécution de cette requête. Dans le plan d'exécution précédent, on voit que l'opération `Sort` est très gourmande car le tri des lignes est réalisé sur plusieurs colonnes. Or, la table `commandes` a une clé primaire sur la colonne `numero_commande`. Cette clé primaire permet d'assurer que toutes les lignes sont uniques dans la table `commandes`. Si l'opération `GROUP BY` ne porte plus que la clé primaire, PostgreSQL peut utiliser le résultat de la lecture par index sur commandes pour faire le regroupement. Le temps d'exécution passe à environ 580ms :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
  FROM commandes
  JOIN lignes_commandes
       USING (numero_commande)
 GROUP BY commandes.numero_commande;
                                          QUERY PLAN
-----------------------------------------------------------------------------
 Group (actual time=0.067..580.198 rows=168749 loops=1)
   -> Merge Join (actual time=0.061..435.154 rows=675543 loops=1)
         Merge Cond: (commandes.numero_commande =
                      lignes_commandes.numero_commande)
```

```
        ->  Index Scan using orders_pkey on commandes
                      (actual time=0.027..49.784 rows=168750 loops=1)
        ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                      (actual time=0.025..131.606 rows=675543 loops=1)
              Heap Fetches: 0
 Total runtime: 584.624 ms
```

Les opérations de dédoublonnages sont régulièrement utilisées pour assurer que les lignes retournées par une requête apparaissent de manière unique. Elles sont souvent inutiles, ou peuvent à minima être largement améliorées en utilisant les propriétés du modèle de données (les clés primaires) et des opérations plus adéquates (`GROUP BY clé_primaire`). Lorsque vous rencontrez des requêtes utilisant DISTINCT, vérifiez que le DISTINCT est vraiment pertinent ou s'il ne peut pas être remplacé par un GROUP BY qui pourrait tirer partie de la lecture d'un index.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[15] .

6.9.7 INDEX INUTILISÉS

- Trop de lignes retournées
- Prédicat incluant une transformation :

```
WHERE col1 + 2 > 5
```

- Statistiques pas à jour ou peu précises
- Opérateur non-supporté par l'index :

```
WHERE col1 <> 'valeur';
```

- Paramétrage de PostgreSQL : `effective_cache_size`

PostgreSQL offre de nombreuses possibilités d'indexation des données :

- Type d'index : B-tree, GiST, GIN, SP-GiST, BRIN et hash.
- Index multi-colonnes : `CREATE INDEX ... ON (col1, col2...);`
- Index partiel : `CREATE INDEX ... WHERE colonne = valeur`
- Index fonctionnel : `CREATE INDEX ... ON (fonction(colonne))`
- Extension offrant des fonctionnalités supplémentaires : pg_trgm

Malgré toutes ces possibilités, une question revient souvent lorsqu'un index vient d'être ajouté : pourquoi cet index n'est pas utilisé ?

[15] http://www.depesz.com/index.php/2010/04/19/getting-unique-elements/

L'optimiseur de PostgreSQL est très avancé et il y a peu de cas où il est mis en défaut. Malgré cela, certains index ne sont pas utilisés comme on le souhaiterait. Il peut y avoir plusieurs raisons à cela.

Problèmes de statistiques

Le cas le plus fréquent concerne les statistiques qui ne sont pas à jour. Cela arrive souvent après le chargement massif d'une table ou une mise à jour massive sans avoir fait une nouvelle collecte des statistiques à l'issue de ces changements.

On utilisera l'ordre `ANALYZE table` pour déclencher explicitement la collecte des statistiques après un tel traitement. En effet, bien qu'autovacuum soit présent, il peut se passer un certain temps entre le moment où le traitement est fait et le moment où autovacuum déclenche une collecte de statistiques. Ou autovacuum peut ne simplement pas se déclencher car le traitement complet est imbriqué dans une seule transaction.

Un traitement batch devra comporter des ordres `ANALYZE` juste après les ordres SQL qui modifient fortement les données :

```
COPY table_travail FROM '/tmp/fichier.csv';
ANALYZE table_travail;
SELECT ... FROM table_travail;
```

Un autre problème qui peut se poser avec les statistiques concerne les tables de très forte volumétrie. Dans certain cas, l'échantillon de données ramené par `ANALYZE` n'est pas assez précis pour donner à l'optimiseur de PostgreSQL une vision suffisamment précise des données. Il choisira alors de mauvais plans d'exécution.

Il est possible d'augmenter la précision de l'échantillon de données ramené à l'aide de l'ordre :

```
ALTER TABLE ... ALTER COLUMN ... SET STATISTICS ...;
```

Problèmes de prédicats

Dans d'autres cas, les prédicats d'une requête ne permettent pas à l'optimiseur de choisir un index pour répondre à une requête. C'est le cas lorsque le prédicat inclut une transformation de la valeur d'une colonne.

L'exemple suivant est assez naïf, mais démontre bien le problème :

```
SELECT * FROM table WHERE col1 + 10 = 10;
```

Avec une telle construction, l'optimiseur ne saura pas tirer partie d'un quelconque index, à moins d'avoir créé un index fonctionnel sur `col1 + 10`, mais cet index est largement contre-productif par rapport à une réécriture de la requête.

Ce genre de problème se rencontre plus souvent sur des prédicats sur des dates :

```sql
SELECT * FROM table WHERE date_trunc('month', date_debut) = 12
```

ou encore plus fréquemment rencontré :

```sql
SELECT * FROM table WHERE extract('year' from date_debut) = 2013
```

Opérateurs non-supportés

Les index B-tree supportent la plupart des opérateurs généraux sur les variables scalaires ((entiers, chaînes, dates, mais pas types composés comme géométries, hstore…)), mais pas la différence (`<>` ou `!=`). Par nature, il n'est pas possible d'utiliser un index pour déterminer *toutes les valeurs sauf une*. Mais ce type de construction est parfois utilisé pour exclure les valeurs les plus fréquentes d'une colonne. Dans ce cas, il est possible d'utiliser un index partiel, qui en plus sera très petit car il n'indexera qu'une faible quantité de données par rapport à la totalité de la table :

```sql
CREATE TABLE test (id serial PRIMARY KEY, v integer);
INSERT INTO test (v) SELECT 0 FROM generate_series(1, 10000);
INSERT INTO test (v) SELECT 1;
ANALYZE test;
CREATE INDEX idx_test_v ON test(v);
EXPLAIN SELECT * FROM test WHERE v <> 0;
                      QUERY PLAN
------------------------------------------------------
 Seq Scan on test  (cost=0.00..170.03 rows=1 width=8)
   Filter: (v <> 0)

DROP INDEX idx_test_v;
```

La création d'un index partiel permet d'en tirer partie :

```sql
CREATE INDEX idx_test_v_partiel ON test (v) WHERE v<>0;
CREATE INDEX
Temps : 67,014 ms
postgres=# EXPLAIN SELECT * FROM test WHERE v <> 0;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_test_v_partiel on test  (cost=0.00..8.27 rows=1 width=8)
```

Paramétrage de PostgreSQL

Plusieurs paramètres de PostgreSQL influencent le choix ou non d'un index :

- `random_page_cost` : indique à PostgreSQL la vitesse d'un accès aléatoire par rapport à un accès séquentiel (`seq_page_cost`).
- `effective_cache_size` : indique à PostgreSQL une estimation de la taille du cache disque du système.

Le paramètre `random_page_cost` a une grande influence sur l'utilisation des index en général. Il indique à PostgreSQL le coût d'un accès disque aléatoire. Il est à comparer au paramètre `seq_page_cost` qui indique à PostgreSQL le coût d'un accès disque séquentiel. Ces coûts d'accès sont purement arbitraires et n'ont aucune réalité physique. Dans sa configuration par défaut, PostgreSQL estime qu'un accès aléatoire est 4 fois plus coûteux qu'un accès séquentiel. Les accès aux index étant par nature aléatoires alors que les parcours de table étant par nature séquentiels, modifier ce paramètre revient à favoriser l'un par rapport à l'autre. Cette valeur est bonne dans la plupart des cas. Mais si le serveur de bases de données dispose d'un système disque rapide, c'est-à-dire une bonne carte RAID et plusieurs disques SAS rapides en RAID10, ou du SSD, il est possible de baisser ce paramètre à 3 voir 2.

Enfin, le paramètre `effective_cache_size` indique à PostgreSQL une estimation de la taille du cache disque du système. Une bonne pratique est de positionner ce paramètre à 2/3 de la quantité totale de RAM du serveur. Sur un système Linux, il est possible de donner une estimation plus précise en s'appuyant sur la valeur de colonne `cached` de la commande `free`. Mais si le cache n'est que peu utilisé, la valeur trouvée peut être trop basse pour pleinement favoriser l'utilisation des index.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[16]

6.9.8 ÉCRITURE DU SQL

- `NOT IN` avec une sous-requête
  - à remplacer par `NOT EXISTS`
- Utilisation systématique de `UNION` au lieu de `UNION ALL`
  - entraîne un tri systématique
- Sous-requête dans le `SELECT`
  - utiliser `LATERAL`

La façon dont une requête SQL est écrite peut aussi avoir un effet négatif sur les performances. Il n'est pas possible d'écrire tous les cas possibles, mais certaines formes d'écritures reviennent souvent.

La clause `NOT IN` n'est pas performance lorsqu'elle est utilisée avec une sous-requête. L'optimiseur ne parvient pas à exécuter ce type de requête efficacement.

```
SELECT *
  FROM commandes
```

[16] http://www.depesz.com/index.php/2010/09/09/why-is-my-index-not-being-used/

```
WHERE numero_commande NOT IN (SELECT numero_commande
                              FROM lignes_commandes);
```

Il est nécessaire de la réécrire avec la clause `NOT EXISTS`, par exemple :

```
SELECT *
  FROM commandes
 WHERE NOT EXISTS (SELECT 1
                  FROM lignes_commandes l
                  WHERE l.numero_commande = commandes.numero_commande);
```

6.10 OUTILS

- pgAdmin3
- explain.depesz.com
- pev
- auto_explain
- plantuner

Il existe quelques outils intéressants dans le cadre du planificateur : deux applications externes pour mieux appréhender un plan d'exécution, deux modules pour changer le comportement du planificateur.

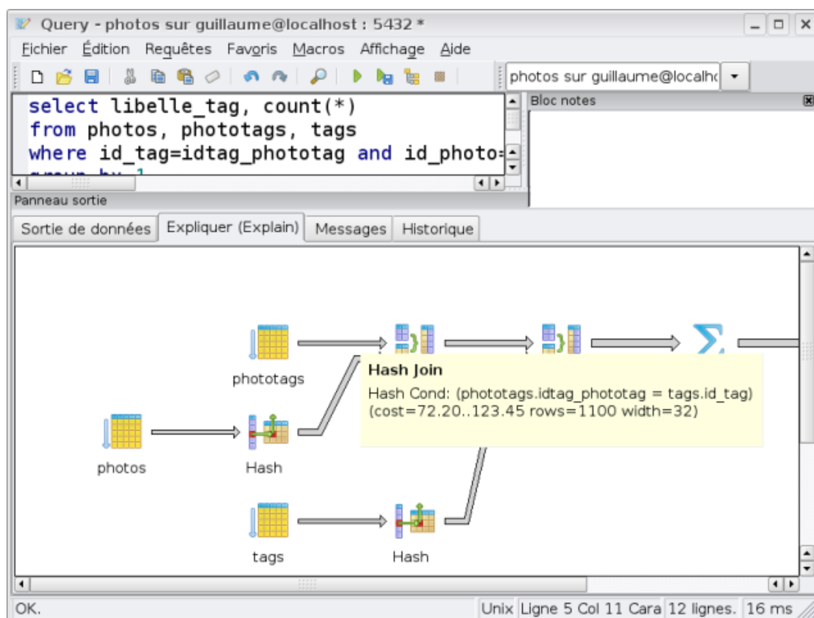
6.10.1 PGADMIN3

- Vision graphique d'un EXPLAIN
- Une icône par nœud
- La taille des flèches dépend de la quantité de données
- Le détail de chaque nœud est affiché en survolant les nœuds

pgAdmin propose depuis très longtemps un affichage graphique de l' `EXPLAIN`. Cet affichage est intéressant car il montre simplement l'ordre dans lequel les opérations sont effectuées. Chaque nœud est représenté par une icône. Les flèches entre chaque nœud indiquent où sont envoyés les flux de données, la taille de la flèche précisant la volumétrie des données.

Les statistiques ne sont affichées qu'en survolant les nœuds.

6.10.2 PGADMIN3 - COPIE D'ÉCRAN



Voici un exemple d'un `EXPLAIN` graphique réalisé par pgAdmin. En passant la souris sur les nœuds, un message affiche les informations statistiques sur le nœud.

6.10.3 SITE EXPLAIN.DEPESZ.COM

- Site web proposant un affichage particulier du EXPLAIN ANALYZE
- Il ne travaille que sur les informations réelles
- Les lignes sont colorées pour indiquer les problèmes
  - Blanc, tout va bien
  - Jaune, inquiétant
  - Marron, plus inquiétant
  - Rouge, très inquiétant
- Installable en local

Hubert Lubaczewski est un contributeur très connu dans la communauté PostgreSQL. Il publie notamment un grand nombre d'articles sur les nouveautés des prochaines versions.

17.12

Cependant, il est aussi connu pour avoir créé un site web d'analyse des plans d'exécution. Ce site web est disponible à cette adresse[17] .

Il suffit d'aller sur ce site, de coller le résultat d'un `EXPLAIN ANALYZE`, et le site affichera le plan d'exécution avec des codes couleurs pour bien distinguer les nœuds performants des autres.

Le code couleur est simple :

- Blanc, tout va bien
- Jaune, inquiétant
- Marron, plus inquiétant
- Rouge, très inquiétant

Plutôt que d'utiliser ce serveur web, il est possible d'installer ce site en local :

- le module explain en Perl[18]
- la partie site web[19]

6.10.4 EXPLAIN.DEPESZ.COM - COPIE D'ÉCRAN

HTML | TEXT | STATS

exclusive	inclusive	rows x	rows	loops	node
0.003	634.605	↑ 29.0	1	1	→ Unique (cost=115136.35..115137.73 rows=29 width=640) (actual time=634.604..634.605 rows=1 loops=1)
0.042	634.602	↑ 29.0	1	1	→ Sort (cost=115136.35..115136.42 rows=29 width=640) (actual time=634.602..634.602 rows=1 loops=1) Sort Key: modwork_beleg.due_date, modwork_beleg.id, modwork_beleg.parent_id, modwork_beleg.owner_id, modwork_beleg.gruppe_id, modwork_beleg.date, modwork_beleg.date_created, mod Sort Method: quicksort Memory: 25kB
136.959	634.560	↑ 29.0	1	1	→ Hash Left Join (cost=2749.20..115135.65 rows=29 width=640) (actual time=457.233..634.560 rows=1 loops=1) Hash Cond: (modwork_beleg.id = modwork_belegreferencesmessageid.beleg_id) Filter: (((modwork_belegreferencesmessageid.messageid)::text = '<20120913062902.175480@gmx.net>'::text) OR ((modwork_belegmessageid.messageid)::text = '<20120913062902.175
246.099	486.281	↑ 1.0	427630	1	→ Hash Left Join (cost=1824.96..52785.04 rows=428226 width=696) (actual time=28.237..486.281 rows=427630 loops=1) Hash Cond: (modwork_beleg.id = modwork_belegmessageid.beleg_id)
212.001	212.001	↑ 1.0	427630	1	→ Seq Scan on modwork_beleg (cost=0.00..45603.89 rows=428226 width=640) (actual time=0.021..212.001 rows=427630 loops=1) Filter: ((state)::text <> 'geloescht'::text)
20.197	28.181	↓ 1.0	53879	1	→ Hash (cost=1151.65..1151.65 rows=53865 width=60) (actual time=28.181..28.181 rows=53879 loops=1) Buckets: 8192 Batches: 1 Memory Usage: 4891kB
7.984	7.984	↓ 1.0	53879	1	→ Seq Scan on modwork_belegmessageid (cost=0.00..1151.65 rows=53865 width=60) (actual time=0.001..7.984 rows=53879 loops=1)
6.651	11.320	↑ 1.0	26928	1	→ Hash (cost=587.44..587.44 rows=26944 width=60) (actual time=11.320..11.320 rows=26928 loops=1) Buckets: 4096 Batches: 1 Memory Usage: 2434kB
4.669	4.669	↑ 1.0	26928	1	→ Seq Scan on modwork_belegreferencesmessageid (cost=0.00..587.44 rows=26944 width=60) (actual time=0.002..4.669 rows=26928 loops=1)

Cet exemple montre un affichage d'un plan sur le site explain.depesz.com.

Voici la signification des différentes colonnes :

[17] http://explain.depesz.com

[18] https://github.com/depesz/Pg--Explain

[19] https://github.com/depesz/explain.depesz.com

- `Exclusive`, durée passée exclusivement sur un nœud ;
- `Inclusive`, durée passée sur un nœud et ses fils ;
- `Rows x`, facteur d'échelle de l'erreur d'estimation du nombre de lignes ;
- `Rows`, nombre de lignes renvoyées ;
- `Loops`, nombre de boucles.

Sur une exécution de 600 ms, un tiers est passé à lire la table avec un parcours séquentiel.

6.10.5 SITE PEV

- Site web proposant un affichage particulier du EXPLAIN ANALYZE
  - mais différent de celui de Depesz
- Fournir un plan d'exécution en JSON
- Installable en local

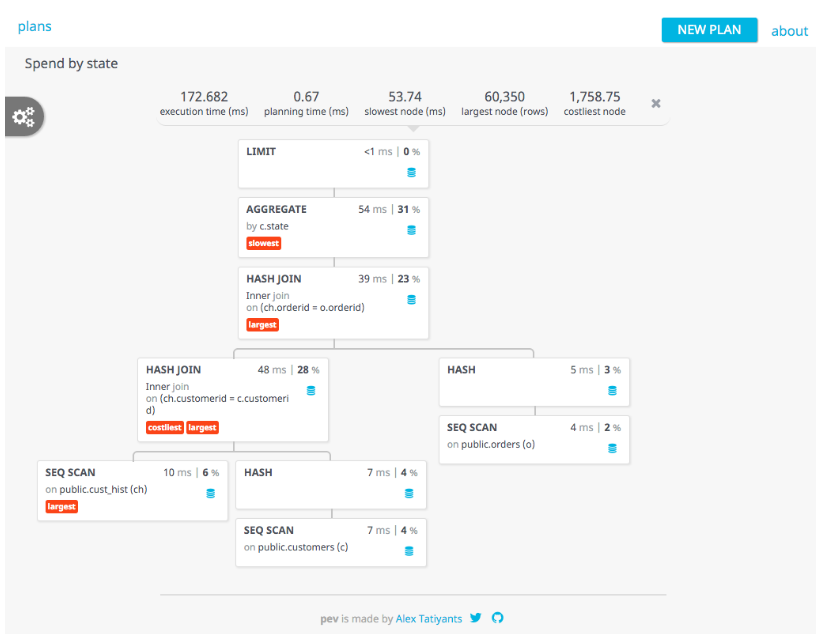
PEV est un outil librement téléchargeable sur ce dépôt github[20] . Il offre un affichage graphique du plan d'exécution et indique le nœud le plus coûteux, le plus long, le plus volumineux, etc.

Il est utilisable sur internet[21] mais aussi installable en local.

---

[20] https://github.com/AlexTatiyants/pev
[21] http://tatiyants.com/pev/#/plans

6.10.6 PEV - COPIE D'ÉCRAN



6.10.7 EXTENSION AUTO_EXPLAIN

- Extension pour PostgreSQL >= 8.4
- Connaître les requêtes lentes est bien
- Mais difficile de connaître leur plan d'exécution au moment où elles ont été lentes
- D'où le module auto_explain

Le but est donc de tracer automatiquement le plan d'exécution des requêtes. Pour éviter de trop écrire dans les fichiers de trace, il est possible de ne tracer que les requêtes dont la durée d'exécution a dépassé une certaine limite. Pour cela, il faut configurer le paramètre `auto_explain.log_min_duration`. D'autres options existent, qui permettent d'activer ou non certaines options du `EXPLAIN` : `log_analyze`, `log_verbose`, `log_buffers`, `log_format`.

6.10.8 EXTENSION PLANTUNER

- Extension, pour PostgreSQL >= 8.4
- Suivant la configuration
  - Interdit l'utilisation de certains index
  - Force à zéro les statistiques d'une table vide

Cette extension est disponible à cette adresse[22] .

Voici un exemple d'utilisation :

```
LOAD 'plantuner';
CREATE TABLE test(id int);
CREATE INDEX id_idx ON test(id);
CREATE INDEX id_idx2 ON test(id);
\d test
     Table "public.test"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
Indexes:
    "id_idx" btree (id)
    "id_idx2" btree (id)

EXPLAIN SELECT id FROM test WHERE id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx2  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)

SET enable_seqscan TO off;
SET plantuner.forbid_index TO 'id_idx2';
EXPLAIN SELECT id FROM test WHERE id=1;
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)

SET plantuner.forbid_index TO 'id_idx2,id_idx';
```

[22] http://www.sai.msu.su/~megera/wiki/plantuner

17.12

```
EXPLAIN SELECT id FROM test WHERE id=1;
                                QUERY PLAN
--------------------------------------------------------------------------
 Seq Scan on test  (cost=10000000000.00..10000000040.00 rows=12 width=4)
   Filter: (id = 1)
(2 rows)
```

Un des intérêts de cette extension est de pouvoir interdire l'utilisation d'un index, afin de pouvoir ensuite le supprimer de manière transparente, c'est-à-dire sans bloquer aucune requête applicative.

6.11 CONCLUSION

- Planificateur très avancé
- Mais faillible
- Cependant
  - ne pensez pas être plus intelligent que le planificateur

Certains SGBD concurrents supportent les *hints*, qui permettent au DBA de forcer l'optimiseur à choisir des plans d'exécution qu'il avait jugé trop coûteux. Ces *hints* sont exprimés sous la forme de commentaires et ne seront donc pas pris en compte par PostgreSQL, qui ne gère pas ces hints.

L'avis de la communauté PostgreSQL (voir https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion) est que les hints mènent à des problèmes de maintenabilité du code applicatif, interfèrent avec les mises à jour, risquent d'être contre-productifs au fur et à mesure que vos tables grossissent, et sont généralement inutiles. Sur le long terme il vaut mieux rapporter un problème rencontré avec l'optimiseur pour qu'il soit définitivement corrigé.

Si le plan d'exécution généré n'est pas optimal, il est préférable de chercher à comprendre d'où vient l'erreur. Nous avons vu dans ce module quelles pouvaient être les causes entrainant des erreurs d'estimation :

- Mauvaise écriture de requête
- Modèle de données pas optimal
- Statistiques pas à jour
- Colonnes corrélées
- ...

6.11.1 QUESTIONS

N'hésitez pas, c'est le moment !

6.12 ANNEXE : NŒUDS D'UN PLAN

- Quatre types de nœuds
  - Parcours (de table, d'index, de TID, etc.)
  - Jointures (Nested Loop, Sort/Merge Join, Hash Join)
  - Opérateurs sur des ensembles (Append, Except, Intersect, etc.)
  - Et quelques autres (Sort, Aggregate, Unique, Limit, Materialize)

Un plan d'exécution est un arbre. Chaque nœud de l'arbre est une opération à effectuer par l'exécuteur. Le planificateur arrange les nœuds pour que le résultat final soit le bon, et qu'il soit récupéré le plus rapidement possible.

Il y a quatre types de nœuds :

- les parcours, qui permettent de lire les données dans les tables en passant :
- soit par la table ;
- soit par l'index ;
- les jointures, qui permettent de joindre deux ensembles de données
- les opérateurs sur des ensembles, qui là aussi vont joindre deux ensembles ou plus
- et les opérations sur un seul ensemble : tri, limite, agrégat, etc.

Cette partie va permettre d'expliquer chaque type de nœuds, ses avantages et inconvénients.

6.12.1 PARCOURS

- Ne prend rien en entrée
- Mais renvoie un ensemble de données
  - Trié ou non, filtré ou non
- Exemples typiques
  - Parcours séquentiel d'une table, avec ou sans filtrage des enregistrements produits
  - Parcours par un index, avec ou sans filtrage supplémentaire

Les parcours sont les seules opérations qui lisent les données des tables (standards, temporaires ou non journalisées). Elles ne prennent donc rien en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

Il existe trois types de parcours que nous allons détailler :

- le parcours de table ;
- le parcours d'index ;
- le parcours de bitmap index, tous les trois pouvant recevoir des filtres supplémentaires en sortie.

Nous verrons aussi que PostgreSQL propose d'autres types de parcours.

6.12.2 PARCOURS DE TABLE

- Parcours séquentiel de la table (Sequential Scan, ou SeqScan)
- Aussi appelé FULL TABLE SCAN par d'autres SGBD
- La table est lue entièrement
  - Même si seulement quelques lignes satisfont la requête
  - Sauf dans le cas de la clause LIMIT sans ORDER BY
- Elle est lue séquentiellement par bloc de 8 Ko
- Optimisation synchronize_seqscans

Le parcours le plus simple est le parcours séquentiel. La table est lue complètement, de façon séquentielle, par bloc de 8 Ko. Les données sont lues dans l'ordre physique sur disque, donc les données ne sont pas envoyées triées au nœud supérieur.

Cela fonctionne dans tous les cas, car il n'y a besoin de rien de plus pour le faire (un parcours d'index nécessite un index, un parcours de table ne nécessite rien de plus que la table).

Le parcours de table est intéressant pour les performances dans deux cas :

- les très petites tables ;
- les grosses tables où la majorité des lignes doit être renvoyée.

Lors de son calcul de coût, le planificateur ajoute la valeur du paramètre `seq_page_cost` à chaque bloc lu.

Une optimisation des parcours séquentiels a eu lieu en version 8.3. Auparavant, quand deux processus parcouraient en même temps la même table de façon séquentielle, ils lisaient chacun la table. À partir de la 8.3, si le paramètre `synchronize_seqscans` est activé, le processus qui entame une lecture séquentielle cherche en premier lieu si un autre

processus ne ferait pas une lecture séquentielle de la même table. Si c'est le cas, Le second processus démarre son scan de table à l'endroit où le premier processus est en train de lire, ce qui lui permet de profiter des données mises en cache par ce processus. L'accès au disque étant bien plus lent que l'accès mémoire, les processus restent naturellement synchronisés pour le reste du parcours de la table, et les lectures ne sont donc réalisées qu'une seule fois. Le début de la table restera à être lu indépendamment. Cette optimisation permet de diminuer le nombre de blocs lus par chaque processus en cas de lectures parallèles de la même table.

Il est possible, pour des raisons de tests, ou pour maintenir la compatibilité avec du code partant de l'hypothèse (erronée) que les données d'une table sont toujours retournées dans le même ordre, de désactiver ce type de parcours en positionnant le paramètre `synchronize_seqscans` à `off`.

6.12.3 PARCOURS D'INDEX

- Parcours aléatoire de l'index
- Pour chaque enregistrement correspondant à la recherche
  - Parcours non séquentiel de la table (pour vérifier la visibilité de la ligne)
- Sur d'autres SGBD, cela revient à un
  - INDEX RANGE SCAN, suivi d'un TABLE ACCESS BY INDEX ROWID
- Gros gain en performance quand le filtre est très sélectif
- L'ensemble de lignes renvoyé est trié

Parcourir une table prend du temps, surtout quand on cherche à ne récupérer que quelques lignes de cette table. Le but d'un index est donc d'utiliser une structure de données optimisée pour satisfaire une recherche particulière (on parle de prédicat).

Cette structure est un arbre. La recherche consiste à suivre la structure de l'arbre pour trouver le premier enregistrement correspondant au prédicat, puis suivre les feuilles de l'arbre jusqu'au dernier enregistrement vérifiant le prédicat. De ce fait, et étant donné la façon dont l'arbre est stocké sur disque, cela peut provoquer des déplacements de la tête de lecture.

L'autre problème des performances sur les index (mais cette fois, spécifique à PostgreSQL) est que les informations de visibilité des lignes sont uniquement stockées dans la table. Cela veut dire que, pour chaque élément de l'index correspondant au filtre, il va falloir lire la ligne dans la table pour vérifier si cette dernière est visible pour la transaction en cours. Il est de toute façons, pour la plupart des requêtes, nécessaire d'aller inspecter l'enregistrement de la table pour récupérer les autres colonnes nécessaires au

bon déroulement de la requête, qui ne sont la plupart du temps pas stockées dans l'index. Ces enregistrements sont habituellement éparpillés dans la table, et retournés dans un ordre totalement différent de leur ordre physique par le parcours sur l'index. Cet accès à la table génère donc énormément d'accès aléatoires. Or, ce type d'activité est généralement le plus lent sur un disque magnétique. C'est pourquoi le parcours d'une large portion d'un index est très lent. PostgreSQL ne cherchera à utiliser un index que s'il suppose qu'il aura peu de lignes à récupérer.

Voici l'algorithme permettant un parcours d'index avec PostgreSQL :

- Pour tous les éléments de l'index
- Chercher l'élément souhaité dans l'index
- Lorsqu'un élément est trouvé
- Vérifier qu'il est visible par la transaction en lisant la ligne dans la table et récupérer les colonnes supplémentaires de la table

Cette manière de procéder est identique à ce que proposent d'autres SGBD sous les termes d'« INDEX RANGE SCAN », suivi d'un « TABLE ACCESS BY INDEX ROWID ».

Un parcours d'index est donc très coûteux, principalement à cause des déplacements de la tête de lecture. Le paramètre lié au coût de lecture aléatoire d'une page est par défaut quatre fois supérieur à celui de la lecture séquentielle d'une page. Ce paramètre s'appelle `random_page_cost`. Un parcours d'index n'est préférable à un parcours de table que si la recherche ne va ramener qu'un très faible pourcentage de la table. Et dans ce cas, le gain possible est très important par rapport à un parcours séquentiel de table. Par contre, il se révèle très lent pour lire un gros pourcentage de la table (les accès aléatoires diminuent spectaculairement les performances).

Il est à noter que, contrairement au parcours de table, le parcours d'index renvoie les données triées. C'est le seul parcours à le faire. Il peut même servir à honorer la clause `ORDER BY` d'une requête. L'index est aussi utilisable dans le cas des tris descendants. Dans ce cas, le nœud est nommé « Index Scan Backward ». Ce renvoi de données triées est très intéressant lorsqu'il est utilisé en conjonction avec la clause `LIMIT`.

Il ne faut pas oublier aussi le coût de mise à jour de l'index. Si un index n'est pas utilisé, il coûte cher en maintenance (ajout des nouvelles entrées, suppression des entrées obsolètes, etc).

Enfin, il est à noter que ce type de parcours est consommateur aussi en CPU.

Voici un exemple montrant les deux types de parcours et ce que cela occasionne comme lecture disque :

Commençons par créer une table, lui insérer quelques données et lui ajouter un index :

```
b1=# CREATE TABLE t1 (id integer);
CREATE TABLE
b1=# INSERT INTO t1 (id) VALUES (1), (2), (3);
INSERT 0 3
b1=# CREATE INDEX i1 ON t1(id);
CREATE INDEX
```

Réinitialisons les statistiques d'activité :

```
b1=# SELECT pg_stat_reset();
 pg_stat_reset
---------------

(1 row)
```

Essayons maintenant de lire la table avec un simple parcours séquentiel :

```
b1=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE id=2;
                              QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1.04 rows=1 width=4)
                 (actual time=0.011..0.012 rows=1 loops=1)
   Filter: (id = 2)
 Total runtime: 0.042 ms
(3 rows)
```

`Seq Scan` est le titre du nœud pour un parcours séquentiel. Profitons-en pour noter qu'il a fait de lui-même un parcours séquentiel. En effet, la table est tellement petite (8 Ko) qu'utiliser l'index couterait forcément plus cher. Maintenant regardons les statistiques sur les blocs lus :

```
b1=# SELECT relname, heap_blks_read, heap_blks_hit,
 idx_blks_read, idx_blks_hit
 FROM pg_statio_user_tables
 WHERE relname='t1';

 relname | heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
---------+----------------+---------------+---------------+--------------
 t1      |              0 |             1 |             0 |            0
(1 row)
```

Seul un bloc a été lu, et il a été lu dans la table (colonne `heap_blks_hit` à 1).

Pour faire un parcours d'index, nous allons désactiver les parcours séquentiels.

```
b1=# SET enable_seqscan TO off;
SET
```

Il existe aussi un paramètre, appelé `enable_indexscan`, pour désactiver les parcours d'index.

17.12

Nous allons de nouveau réinitialiser les statistiques :

```
b1=# SELECT pg_stat_reset();
 pg_stat_reset
---------------

(1 row)
```

Maintenant relançons la requête :

```
b1=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE id=2;
                              QUERY PLAN
------------------------------------------------------------------------
 Index Scan using i1 on t1  (cost=0.00..8.27 rows=1 width=4)
                            (actual time=0.088..0.090 rows=1 loops=1)
   Index Cond: (id = 2)
 Total runtime: 0.121 ms
(3 rows)
```

Nous avons bien un parcours d'index. Vérifions les statistiques sur l'activité :

```
b1=# SELECT relname, heap_blks_read, heap_blks_hit,
 idx_blks_read, idx_blks_hit
 FROM pg_statio_user_tables
 WHERE relname='t1';
 relname | heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
---------+----------------+---------------+---------------+--------------
 t1      |              0 |             1 |             0 |            1
(1 row)
```

Une page disque a été lue dans l'index (colonne `idx_blks_hit` à 1) et une autre a été lue dans la table (colonne `heap_blks_hit` à 1). Le plus impactant est l'accès aléatoire sur l'index et la table. Il serait bon d'avoir une lecture de l'index, puis une lecture séquentielle de la table. C'est le but du Bitmap Index Scan.

6.12.4 PARCOURS D'INDEX BITMAP

- En VO, Bitmap Index Scan / Bitmap Heap Scan
- Disponible à partir de la 8.1
- Diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table
  - Lecture en un bloc de l'index
  - Lecture en un bloc de la partie intéressante de la table
- Autre intérêt : pouvoir combiner plusieurs index en mémoire
  - Nœud BitmapAnd

    - Nœud BitmapOr
- Coût de démarrage généralement important
    - Parcours moins intéressant avec une clause LIMIT

Contrairement à d'autres SGBD, un index bitmap n'a aucune existence sur disque. Il est créé en mémoire lorsque son utilisation a un intérêt. Le but est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table :

- Lecture en un bloc de l'index ;
- Lecture en un bloc de la partie intéressante de la table (dans l'ordre physique de la table, pas dans l'ordre logique de l'index).

Il est souvent utilisé quand il y a un grand nombre de valeurs à filtrer, notamment pour les clauses `IN` et `ANY`. En voici un exemple :

```
b1=# CREATE TABLE t1(c1 integer, c2 integer);
CREATE TABLE
b1=# INSERT INTO t1 SELECT i, i+1 FROM generate_series(1, 1000) AS i;
INSERT 0 1000
b1=# CREATE INDEX ON t1(c1);
CREATE INDEX
b1=# CREATE INDEX ON t1(c2);
CREATE INDEX
b1=# EXPLAIN SELECT * FROM t1 WHERE c1 IN (10, 40, 60, 100, 600);
                              QUERY PLAN
--------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=17.45..22.85 rows=25 width=8)
   Recheck Cond: (c1 = ANY ('{10,40,60,100,600}'::integer[]))
   ->  Bitmap Index Scan on t1_c1_idx  (cost=0.00..17.44 rows=25 width=0)
         Index Cond: (c1 = ANY ('{10,40,60,100,600}'::integer[]))
(4 rows)
```

La partie `Bitmap Index Scan` concerne le parcours de l'index, et la partie `Bitmap Heap Scan` concerne le parcours de table.

L'algorithme pourrait être décrit ainsi :

- Chercher tous les éléments souhaités dans l'index
- Les placer dans une structure (de TID) de type bitmap en mémoire
- Faire un parcours séquentiel partiel de la table

Ce champ de bits a deux codages possibles :

- 1 bit par ligne
- Ou 1 bit par bloc si trop de données.

Dans ce dernier (mauvais) cas, il y a une étape de revérification (`Recheck Condition`).

Ce type d'index présente un autre gros intérêt : pouvoir combiner plusieurs index en mémoire. Les bitmaps de TID se combinent facilement avec des opérations booléennes AND et OR. Dans ce cas, on obtient les nœuds `BitmapAnd` et `Nœud BitmapOr`. Voici un exemple de ce dernier :

```
b1=# EXPLAIN SELECT * FROM t1
 WHERE c1 IN (10, 40, 60, 100, 600) OR c2 IN (300, 400, 500);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=30.32..36.12 rows=39 width=8)
   Recheck Cond: ((c1 = ANY ('{10,40,60,100,600}'::integer[]))
                 OR (c2 = ANY ('{300,400,500}'::integer[])))
   ->  BitmapOr  (cost=30.32..30.32 rows=40 width=0)
         ->  Bitmap Index Scan on t1_c1_idx
               (cost=0.00..17.44 rows=25 width=0)
               Index Cond: (c1 = ANY ('{10,40,60,100,600}'::integer[]))
         ->  Bitmap Index Scan on t1_c2_idx
               (cost=0.00..12.86 rows=15 width=0)
               Index Cond: (c2 = ANY ('{300,400,500}'::integer[]))
(7 rows)
```

Le coût de démarrage est généralement important à cause de la lecture préalable de l'index et du tri des TID. Du coup, ce type de parcours est moins intéressant si une clause LIMIT est présente. Un parcours d'index simple sera généralement choisi dans ce cas.

Le paramètre `enable_bitmapscan` permet d'activer ou de désactiver l'utilisation des parcours d'index bitmap.

À noter que ce type de parcours n'est disponible qu'à partir de PostgreSQL 8.1.

6.12.5 PARCOURS D'INDEX SEUL

- Avant la 9.2, pour une requête de ce type
  - `SELECT c1 FROM t1 WHERE c1<10`
- PostgreSQL devait lire l'index et la table
  - car les informations de visibilité ne se trouvent que dans la table
- En 9.2, le planificateur peut utiliser la « Visibility Map »
  - nouveau nœud « Index Only Scan »
  - Index B-Tree (9.2+)
  - Index SP-GiST (9.2+)
  - Index GiST (9.5+) => Types : point, box, inet, range

Voici un exemple en 9.1 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM generate_series(1,10000000) a;
INSERT 0 10000000
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                              (actual time=209.569..3314.717 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                            (actual time=197.177..197.177 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 3323.497 ms
(5 rows)

b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                                (actual time=48.620..269.907 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                              (actual time=35.780..35.780 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 273.761 ms
(5 rows)
```

Donc 3 secondes pour la première exécution (avec un cache pas forcément vide), et 273 millisecondes pour la deuxième exécution (et les suivantes, non affichées ici).

Voici ce que cet exemple donne en 9.2 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM (select generate_series(1,10000000)) AS t(a);
INSERT 0 10000000
```

```
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                                (cost=0.00..3084.77 rows=86656 width=11)
                                (actual time=0.080..97.942 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
 Total runtime: 108.134 ms
(4 rows)

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
                                (cost=0.00..3084.77 rows=86656 width=11)
                                (actual time=0.024..26.954 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=347
 Total runtime: 34.352 ms
(5 rows)
```

Donc, même à froid, il est déjà pratiquement trois fois plus rapide que la version 9.1, à chaud. La version 9.2 est dix fois plus rapide à chaud.

Essayons maintenant en désactivant les parcours d'index seul :

```
b1=# SET enable_indexonlyscan TO off;
SET
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                              (actual time=29.256..2992.289 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=346 read=43834 written=2022
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                             (actual time=27.004..27.004 rows=89432 loops=1)
```

```
        Index Cond: ((a >= 10000) AND (a <= 100000))
        Buffers: shared hit=346
 Total runtime: 3000.502 ms
(8 rows)

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                              (actual time=23.533..1141.754 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=2 read=44178
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                              (actual time=21.592..21.592 rows=89432 loops=1)
        Index Cond: ((a >= 10000) AND (a <= 100000))
        Buffers: shared hit=2 read=344
 Total runtime: 1146.538 ms
(8 rows)
```

On retombe sur les performances de la version 9.1.

Maintenant, essayons avec un cache vide (niveau PostgreSQL et système) :

- en 9.1

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                          QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                             (actual time=126.624..9750.245 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Buffers: shared hit=2 read=44250
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                            (actual time=112.542..112.542 rows=89877 loops=1)
        Index Cond: ((a >= 10000) AND (a <= 100000))
        Buffers: shared hit=2 read=346
 Total runtime: 9765.670 ms
(7 rows)
```

- en 9.2

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan
b1=# WHERE a BETWEEN 10000 AND 100000;
                                          QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan
```

```
                                            (cost=0.00..3084.77 rows=86656 width=11)
                                            (actual time=11.592..63.379 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=2 read=345
 Total runtime: 70.188 ms
(5 rows)
```

La version 9.1 met 10 secondes à exécuter la requête, alors que la version 9.2 ne met que 70 millisecondes (elle est donc 142 fois plus rapide).

Voir aussi cet article de blog[23] .

6.12.6 PARCOURS : AUTRES

- TID Scan
- Function Scan
- Values
- Result

Il existe d'autres parcours, bien moins fréquents ceci dit.

`TID` est l'acronyme de `Tuple ID`. C'est en quelque sorte un pointeur vers une ligne. Un `TID Scan` est un parcours de `TID`. Ce type de parcours est généralement utilisé en interne par PostgreSQL. Notez qu'il est possible de le désactiver via le paramètre `enable_tidscan`.

Un `Function Scan` est utilisé par les fonctions renvoyant des ensembles (appelées `SRF` pour `Set Returning Functions`). En voici un exemple :

```
b1=# EXPLAIN SELECT * FROM generate_series(1, 1000);
                               QUERY PLAN
------------------------------------------------------------------------
 Function Scan on generate_series  (cost=0.00..10.00 rows=1000 width=4)
(1 row)
```

`VALUES` est une clause de l'instruction `INSERT`, mais `VALUES` peut aussi être utilisé comme une table dont on spécifie les valeurs. Par exemple :

```
b1=# VALUES (1), (2);
 column1
---------
       1
       2
```

[23] http://pgsnaga.blogspot.com/2011/10/index-only-scans-and-heap-block-reads.html

```
(2 rows)

b1=# SELECT * FROM (VALUES ('a', 1), ('b', 2), ('c', 3)) AS tmp(c1, c2);
 c1 | c2
----+----
 a  |  1
 b  |  2
 c  |  3
(3 rows)
```

Le planificateur utilise un nœud spécial appelé `Values Scan` pour indiquer un parcours sur cette clause :

```
b1=# EXPLAIN
b1-# SELECT *
b1-# FROM (VALUES ('a', 1), ('b', 2), ('c', 3))
b1-# AS tmp(c1, c2);
                          QUERY PLAN
--------------------------------------------------------------
 Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=36)
(1 row)
```

Enfin, le nœud `Result` n'est pas à proprement parler un nœud de type parcours. Il y ressemble dans le fait qu'il ne prend aucun ensemble de données en entrée et en renvoie un en sortie. Son but est de renvoyer un ensemble de données suite à un calcul. Par exemple :

```
b1=# EXPLAIN SELECT 1+2;
                QUERY PLAN
------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=0)
(1 row)
```

6.12.7 JOINTURES

- Prend deux ensembles de données en entrée
  - L'une est appelée inner (interne)
  - L'autre est appelée outer (externe)
- Et renvoie un seul ensemble de données
- Exemples typiques
  - Nested Loop, Merge Join, Hash Join

Le but d'une jointure est de grouper deux ensembles de données pour n'en produire qu'un seul. L'un des ensembles est appelé ensemble interne (`inner set`), l'autre est appelé

ensemble externe (`outer set`).

Le planificateur de PostgreSQL est capable de traiter les jointures grâce à trois nœuds :

- `Nested Loop`, une boucle imbriquée ;
- `Merge Join`, un parcours des deux ensembles triés ;
- `Hash Join`, une jointure par tests des données hachées.

6.12.8 NESTED LOOP

- Pour chaque ligne de la relation externe
  - Pour chaque ligne de la relation interne
  - Si la condition de jointure est avérée
    - Émettre la ligne en résultat
- L'ensemble externe n'est parcouru qu'une fois
- L'ensemble interne est parcouru pour chaque ligne de l'ensemble externe
  - Avoir un index utilisable sur l'ensemble interne augmente fortement les performances

Étant donné le pseudo-code indiqué ci-dessus, on s'aperçoit que l'ensemble externe n'est parcouru qu'une fois alors que l'ensemble interne est parcouru pour chaque ligne de l'ensemble externe. Le coût de ce nœud est donc proportionnel à la taille des ensembles. Il est intéressant pour les petits ensembles de données, et encore plus lorsque l'ensemble interne dispose d'un index satisfaisant la condition de jointure.

En théorie, il s'agit du type de jointure le plus lent, mais il a un gros intérêt. Il n'est pas nécessaire de trier les données ou de les hacher avant de commencer à traiter les données. Il a donc un coût de démarrage très faible, ce qui le rend très intéressant si cette jointure est couplée à une clause `LIMIT`, ou si le nombre d'itérations (donc le nombre d'enregistrements de la relation externe) est faible.

Il est aussi très intéressant, car il s'agit du seul nœud capable de traiter des jointures sur des conditions différentes de l'égalité ainsi que des jointures de type `CROSS JOIN`.

Voici un exemple avec deux parcours séquentiels :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..37.18 rows=281 width=307)
   Join Filter: (pg_class.relnamespace = pg_namespace.oid)
```

```
   ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
   ->  Materialize  (cost=0.00..1.09 rows=6 width=117)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
(5 rows)
```

Et un exemple avec un parcours séquentiel et un parcours d'index :

```
b1=# SET random_page_cost TO 0.5;
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=0.00..33.90 rows=281 width=307)
   ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
   ->  Index Scan using pg_class_relname_nsp_index on pg_class
       (cost=0.00..4.30 rows=94 width=194)
         Index Cond: (relnamespace = pg_namespace.oid)
(4 rows)
```

Le paramètre `enable_nestloop` permet d'activer ou de désactiver ce type de nœud.

6.12.9 MERGE JOIN

- Trier l'ensemble interne
- Trier l'ensemble externe
- Tant qu'il reste des lignes dans un des ensembles
  - Lire les deux ensembles en parallèle
  - Lorsque la condition de jointure est avérée
  - Émettre la ligne en résultat
- Parcourir les deux ensembles triés (d'où Sort-Merge Join)
- Ne gère que les conditions avec égalité
- Produit un ensemble résultat trié
- Le plus rapide sur de gros ensembles de données

Contrairement au `Nested Loop`, le `Merge Join` ne lit qu'une fois chaque ligne, sauf pour les valeurs dupliquées. C'est d'ailleurs son principal atout.

L'algorithme est assez simple. Les deux ensembles de données sont tout d'abord triés, puis ils sont parcourus ensemble. Lorsque la condition de jointure est vraie, la ligne résultante est envoyée dans l'ensemble de données en sortie.

L'inconvénient de cette méthode est que les données en entrée doivent être triées. Trier les données peut prendre du temps, surtout si les ensembles de données sont volumineux.

Cela étant dit, le `Merge Join` peut s'appuyer sur un index pour accélérer l'opération de tri (ce sera alors forcément un `Index Scan`). Une table clusterisée peut aussi accélérer l'opération de tri. Néanmoins, il faut s'attendre à avoir un coût de démarrage important pour ce type de nœud, ce qui fait qu'il sera facilement disqualifié si une clause LIMIT est à exécuter après la jointure.

Le gros avantage du tri sur les données en entrée est que les données reviennent triées. Cela peut avoir son avantage dans certains cas.

Voici un exemple pour ce nœud :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                               QUERY PLAN
-------------------------------------------------------------------------
 Merge Join  (cost=23.38..27.62 rows=281 width=307)
   Merge Cond: (pg_namespace.oid = pg_class.relnamespace)
   ->  Sort  (cost=1.14..1.15 rows=6 width=117)
         Sort Key: pg_namespace.oid
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
   ->  Sort  (cost=22.24..22.94 rows=281 width=194)
         Sort Key: pg_class.relnamespace
         ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
(8 rows)
```

Le paramètre `enable_mergejoin` permet d' activer ou de désactiver ce type de nœud.

6.12.10 HASH JOIN

- Calculer le hachage de chaque ligne de l'ensemble interne
- Tant qu'il reste des lignes dans l'ensemble externe
  - Hacher la ligne lue
  - Comparer ce hachage aux lignes hachées de l'ensemble interne
  - Si une correspondance est trouvée
  - Émettre la ligne trouvée en résultat
- Ne gère que les conditions avec égalité
- Idéal pour joindre une grande table à une petite table
- Coût de démarrage important à cause du hachage de la table

La vérification de la condition de jointure peut se révéler assez lente dans beaucoup de cas : elle nécessite un accès à un enregistrement par un index ou un parcours de la table

interne à chaque itération dans un Nested Loop par exemple. Le `Hash Join` cherche à supprimer ce problème en créant une table de hachage de la table interne. Cela sous-entend qu'il faut au préalable calculer le hachage de chaque ligne de la table interne. Ensuite, il suffit de parcourir la table externe, hacher chaque ligne l'une après l'autre et retrouver le ou les enregistrements de la table interne pouvant correspondre à la valeur hachée de la table externe. On vérifie alors qu'ils répondent bien aux critères de jointure (il peut y avoir des collisions dans un hachage, ou des prédicats supplémentaires à vérifier).

Ce type de nœud est très rapide à condition d'avoir suffisamment de mémoire pour stocker le résultat du hachage de l'ensemble interne. Du coup, le paramétrage de `work_mem` peut avoir un gros impact. De même, diminuer le nombre de colonnes récupérées permet de diminuer la mémoire à utiliser pour le hachage et du coup d'améliorer les performances d'un `Hash Join`. Cependant, si la mémoire est insuffisante, il est possible de travailler par groupes de lignes (`batch`). L'algorithme est alors une version améliorée de l'algorithme décrit plus haut, permettant justement de travailler en partitionnant la table interne (on parle de Hybrid Hash Join). Il est à noter que ce type de nœud est souvent idéal pour joindre une grande table à une petite table.

Le coût de démarrage peut se révéler important à cause du hachage de la table interne. Il ne sera probablement pas utilisé par l'optimiseur si une clause `LIMIT` est à exécuter après la jointure.

Attention, les données retournées par ce nœud ne sont pas triées.

De plus, ce type de nœud peut être très lent si l'estimation de la taille des tables est mauvaise.

Voici un exemple de `Hash Join` :

```
b1=# EXPLAIN SELECT *
    FROM pg_class, pg_namespace
 WHERE pg_class.relnamespace=pg_namespace.oid;
                               QUERY PLAN
------------------------------------------------------------------------
 Hash Join  (cost=1.14..15.81 rows=281 width=307)
   Hash Cond: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=194)
   ->  Hash  (cost=1.06..1.06 rows=6 width=117)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=117)
(5 rows)
```

Le paramètre `enable_hashjoin` permet d' activer ou de désactiver ce type de nœud.

6.12.11 SUPPRESSION D'UNE JOINTURE

```
SELECT pg_class.relname, pg_class.reltuples
FROM pg_class
LEFT JOIN pg_namespace
       ON pg_class.relnamespace=pg_namespace.oid;
```

- Un index unique existe sur la colonne oid de pg_namespace
- Jointure inutile
  - sa présence ne change pas le résultat
- PostgreSQL peut supprimer la jointure à partir de la 9.0

Sur la requête ci-dessus, la jointure est inutile. En effet, il existe un index unique sur la colonne `oid` de la table `pg_namespace`. De plus, aucune colonne de la table `pg_namespace` ne va apparaître dans le résultat. Autrement dit, que la jointure soit présente ou non, cela ne va pas changer le résultat. Dans ce cas, il est préférable de supprimer la jointure. Si le développeur ne le fait pas, PostgreSQL le fera (pour les versions 9.0 et ultérieures de PostgreSQL). Cet exemple le montre.

Voici la requête exécutée en 8.4 :

```
b1=# EXPLAIN SELECT pg_class.relname, pg_class.reltuples
    FROM pg_class
 LEFT JOIN pg_namespace ON pg_class.relnamespace=pg_namespace.oid;
                               QUERY PLAN
------------------------------------------------------------------------
 Hash Left Join  (cost=1.14..12.93 rows=244 width=68)
   Hash Cond: (pg_class.relnamespace = pg_namespace.oid)
   ->  Seq Scan on pg_class  (cost=0.00..8.44 rows=244 width=72)
   ->  Hash  (cost=1.06..1.06 rows=6 width=4)
         ->  Seq Scan on pg_namespace  (cost=0.00..1.06 rows=6 width=4)
(5 rows)
```

Et la même requête exécutée en 9.0 :

```
b1=# EXPLAIN SELECT pg_class.relname, pg_class.reltuples
    FROM pg_class
 LEFT JOIN pg_namespace ON pg_class.relnamespace=pg_namespace.oid;
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on pg_class  (cost=0.00..10.81 rows=281 width=72)
(1 row)
```

On constate que la jointure est ignorée.

Ce genre de requête peut fréquemment survenir surtout avec des générateurs de requêtes comme les ORM. L'utilisation de vues imbriquées peut aussi être la source de ce type de problème.

6.12.12 ORDRE DE JOINTURE

- Trouver le bon ordre de jointure est un point clé dans la recherche de performances
- Nombre de possibilités en augmentation factorielle avec le nombre de tables
- Si petit nombre, recherche exhaustive
- Sinon, utilisation d'heuristiques et de GEQO
  - Limite le temps de planification et l'utilisation de mémoire
  - GEQO remplacé par Simulated Annealing ? (recuit simulé en VF)

Sur une requête comme `SELECT * FROM a, b, c...`, les tables a, b et c ne sont pas forcément jointes dans cet ordre. PostgreSQL teste différents ordres pour obtenir les meilleures performances.

Prenons comme exemple la requête suivante :

```
SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
```

Avec une table a contenant un million de lignes, une table b n'en contenant que 1000 et une table c en contenant seulement 10, et une configuration par défaut, son plan d'exécution est celui-ci :

```
b1=# EXPLAIN SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
                                  QUERY PLAN
---------------------------------------------------------------------------
 Nested Loop  (cost=1.23..18341.35 rows=1 width=12)
   Join Filter: (a.id = b.id)
   ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
   ->  Materialize  (cost=1.23..18176.37 rows=10 width=8)
         ->  Hash Join  (cost=1.23..18176.32 rows=10 width=8)
               Hash Cond: (a.id = c.id)
               ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
               ->  Hash  (cost=1.10..1.10 rows=10 width=4)
                     ->  Seq Scan on c  (cost=0.00..1.10 rows=10 width=4)
(9 rows)
```

Le planificateur préfère joindre tout d' abord la table a à la table c, puis son résultat à la table b. Cela lui permet d'avoir un ensemble de données en sortie plus petit (donc moins de consommation mémoire) avant de faire la jointure avec la table b.

Cependant, si PostgreSQL se trouve face à une jointure de 25 tables, le temps de calculer tous les plans possibles en prenant en compte l'ordre des jointures sera très important. En fait, plus le nombre de tables jointes est important, et plus le temps de planification va augmenter. Il est nécessaire de prévoir une échappatoire à ce système. En fait, il en existe

plusieurs. Les paramètres from_collapse_limit et join_collapse_limit permettent de spécifier une limite en nombre de tables. Si cette limite est dépassée, PostgreSQL ne cherchera plus à traiter tous les cas possibles de réordonnancement des jointures. Par défaut, ces deux paramètres valent 8, ce qui fait que, dans notre exemple, le planificateur a bien cherché à changer l'ordre des jointures. En configurant ces paramètres à une valeur plus basse, le plan va changer :

```
b1=# SET join_collapse_limit TO 2;
SET
b1=# EXPLAIN SELECT * FROM a JOIN b ON a.id=b.id JOIN c ON b.id=c.id;
                              QUERY PLAN
--------------------------------------------------------------------
 Nested Loop  (cost=27.50..18363.62 rows=1 width=12)
   Join Filter: (a.id = c.id)
   ->  Hash Join  (cost=27.50..18212.50 rows=1000 width=8)
         Hash Cond: (a.id = b.id)
         ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
         ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
               ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
   ->  Materialize  (cost=0.00..1.15 rows=10 width=4)
         ->  Seq Scan on c  (cost=0.00..1.10 rows=10 width=4)
(9 rows)
```

Avec un join_collapse_limit à 2, PostgreSQL décide de ne pas tester l'ordre des jointures. Le plan fourni fonctionne tout aussi bien, mais son estimation montre qu'elle semble être moins performante (coût de 18363 au lieu de 18341 précédemment).

Une autre technique mise en place pour éviter de tester tous les plans possibles est GEQO (*GEnetic Query Optimizer*). Cette technique est très complexe, et dispose d'un grand nombre de paramètres que très peu savent réellement configurer. Comme tout algorithme génétique, il fonctionne par introduction de mutations aléatoires sur un état initial donné. Il permet de planifier rapidement une requête complexe, et de fournir un plan d'exécution acceptable.

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe (depuis la version 9.1, voir ce commit[24] pour plus de détails). Tant que le paramètre geqo_seed ainsi que les autres paramètres contrôlant GEQO restent inchangés, le plan obtenu pour une requête donnée restera inchangé. Il est possible de faire varier la valeur de geqo_seed pour obtenir d'autres plans (voir la documentation officielle[25] pour approfondir ce point).

---

[24] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f
[25] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116279

6.12.13 OPÉRATIONS ENSEMBLISTES

- Prend un ou plusieurs ensembles de données en entrée
- Et renvoie un ensemble de données
- Concernent principalement les requêtes sur des tables partitionnées ou héritées
- Exemples typiques
  - Append
  - Intersect
  - Except

Ce type de nœuds prend un ou plusieurs ensembles de données en entrée et renvoie un seul ensemble de données. Cela concerne surtout les requêtes visant des tables partitionnées ou héritées.

6.12.14 APPEND

- Prend plusieurs ensembles de données
- Fournit un ensemble de données en sortie
  - Non trié
- Utilisé par les requêtes
  - Sur des tables héritées (partitionnement inclus)
  - Ayant des UNION ALL et des UNION
  - Attention que le UNION sans ALL élimine les duplicats, ce qui nécessite une opération supplémentaire de tri

Un nœud `Append` a pour but de concaténer plusieurs ensembles de données pour n'en faire qu'un, non trié. Ce type de nœud est utilisé dans les requêtes concaténant explicitement des tables (clause `UNION`) ou implicitement (requêtes sur une table mère).

Supposons que la table t1 est une table mère. Plusieurs tables héritent de cette table : t1_0, t1_1, t1_2 et t1_3. Voici ce que donne un `SELECT` sur la table mère :

```
b1=# EXPLAIN SELECT * FROM t1;
                              QUERY PLAN
------------------------------------------------------------------------
 Result  (cost=0.00..89.20 rows=4921 width=36)
   ->  Append  (cost=0.00..89.20 rows=4921 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
         ->  Seq Scan on t1_0 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_1 t1  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on t1_2 t1  (cost=0.00..22.30 rows=1230 width=36)
```

```
        -> Seq Scan on t1_3 t1  (cost=0.00..22.30 rows=1230 width=36)
(7 rows)
```

Nouvel exemple avec un filtre sur la clé de partitionnement :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 off
(1 row)
b1=# EXPLAIN SELECT * FROM t1 WHERE c1>250;
                              QUERY PLAN
----------------------------------------------------------------------
 Result  (cost=0.00..101.50 rows=1641 width=36)
   ->  Append  (cost=0.00..101.50 rows=1641 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_0 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_1 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_2 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_3 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
(12 rows)
```

Le paramètre `constraint_exclusion` permet d'éviter de parcourir les tables filles qui ne peuvent pas accueillir les données qui nous intéressent. Pour que le planificateur comprenne qu'il peut ignorer certaines tables filles, ces dernières doivent avoir des contraintes `CHECK` qui assurent le planificateur qu'elles ne peuvent pas contenir les données en question :

```
b1=# SHOW constraint_exclusion ;
 constraint_exclusion
----------------------
 on
(1 row)
b1=# EXPLAIN SELECT * FROM t1 WHERE c1>250;
                              QUERY PLAN
----------------------------------------------------------------------
 Result  (cost=0.00..50.75 rows=821 width=36)
   ->  Append  (cost=0.00..50.75 rows=821 width=36)
         ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=36)
               Filter: (c1 > 250)
         ->  Seq Scan on t1_2 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
```

```
         ->  Seq Scan on t1_3 t1  (cost=0.00..25.38 rows=410 width=36)
               Filter: (c1 > 250)
(8 rows)
```

Une requête utilisant `UNION ALL` passera aussi par un nœud `Append` :

```
b1=# EXPLAIN SELECT 1 UNION ALL SELECT 2;
                      QUERY PLAN
------------------------------------------------------
 Result  (cost=0.00..0.04 rows=2 width=4)
   ->  Append  (cost=0.00..0.04 rows=2 width=4)
         ->  Result  (cost=0.00..0.01 rows=1 width=0)
         ->  Result  (cost=0.00..0.01 rows=1 width=0)
(4 rows)
```

`UNION ALL` récupère toutes les lignes des deux ensembles de données, même en cas de duplicat. Pour n'avoir que les lignes distinctes, il est possible d'utiliser `UNION` sans la clause `ALL` mais cela nécessite un tri des données pour faire la distinction (un peu comme un `Merge Join`).

Attention que le UNION sans ALL élimine les duplicats, ce qui nécessite une opération supplémentaire de tri :

```
b1=# EXPLAIN SELECT 1 UNION SELECT 2;
                         QUERY PLAN
------------------------------------------------------------
 Unique  (cost=0.05..0.06 rows=2 width=0)
   ->  Sort  (cost=0.05..0.06 rows=2 width=0)
         Sort Key: (1)
         ->  Append  (cost=0.00..0.04 rows=2 width=0)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
(6 rows)
```

6.12.15 MERGEAPPEND

- Append avec optimisation
- Fournit un ensemble de données en sortie trié
- Utilisé par les requêtes
  - UNION ALL ou partitionnement/héritage
  - Utilisant des parcours triés
  - Idéal avec Limit

17.12

Le nœud MergeAppend est une optimisation spécifiquement conçue pour le partitionnement, introduite en 9.1.

Cela permet de répondre plus efficacement aux requêtes effectuant un tri sur un UNION ALL, soit explicite, soit induit par un héritage/partitionnement. Considérons la requête suivante :

```
SELECT *
FROM (
 SELECT t1.a, t1.b FROM t1
 UNION ALL
 SELECT t2.a, t2.c FROM t2
) t
ORDER BY a;
```

Il est facile de répondre à cette requête si l'on dispose d'un index sur les colonnes a des tables t1 et t2: il suffit de parcourir chaque index en parallèle (assurant le tri sur a), en renvoyant la valeur la plus petite.

Pour comparaison, avant la 9.1 et l'introduction du nœud MergeAppend, le plan obtenu était celui-ci :

```
                                         QUERY PLAN
--------------------------------------------------------------------------------
 Sort  (cost=24129.64..24629.64 rows=200000 width=22)
       (actual time=122.705..133.403 rows=200000 loops=1)
   Sort Key: t1.a
   Sort Method:  quicksort  Memory: 21770kB
   ->  Result  (cost=0.00..6520.00 rows=200000 width=22)
               (actual time=0.013..76.527 rows=200000 loops=1)
         ->  Append  (cost=0.00..6520.00 rows=200000 width=22)
                     (actual time=0.012..54.425 rows=200000 loops=1)
               ->  Seq Scan on t1  (cost=0.00..2110.00 rows=100000 width=23)
                                   (actual time=0.011..19.379 rows=100000 loops=1)
               ->  Seq Scan on t2  (cost=0.00..4410.00 rows=100000 width=22)
                                   (actual time=1.531..22.050 rows=100000 loops=1)
 Total runtime: 141.708 ms
```

Depuis la 9.1, l'optimiseur est capable de détecter qu'il existe un **parcours paramétré**, renvoyant les données triées sur la clé demandée (a), et utilise la stratégie MergeAppend :

```
                                        QUERY PLAN
--------------------------------------------------------------------------------
 Merge Append  (cost=0.72..14866.72 rows=300000 width=23)
               (actual time=0.040..76.783 rows=300000 loops=1)
   Sort Key: t1.a
   ->  Index Scan using t1_pkey on t1  (cost=0.29..3642.29 rows=100000 width=22)
                                       (actual time=0.014..18.876 rows=100000 loops=1)
```

```
   ->  Index Scan using t2_pkey on t2  (cost=0.42..7474.42 rows=200000 width=23)
                                 (actual time=0.025..35.920 rows=200000 loops=1)
 Total runtime: 85.019 ms
```

Cette optimisation est d'autant plus intéressante si l'on utilise une clause LIMIT.

Sans MergeAppend :

```
                                            QUERY PLAN
--------------------------------------------------------------------------------
 Limit  (cost=9841.93..9841.94 rows=5 width=22)
        (actual time=119.946..119.946 rows=5 loops=1)
   ->  Sort  (cost=9841.93..10341.93 rows=200000 width=22)
             (actual time=119.945..119.945 rows=5 loops=1)
         Sort Key: t1.a
         Sort Method:  top-N heapsort  Memory: 25kB
         ->  Result  (cost=0.00..6520.00 rows=200000 width=22)
                     (actual time=0.008..75.482 rows=200000 loops=1)
               ->  Append  (cost=0.00..6520.00 rows=200000 width=22)
                           (actual time=0.008..53.644 rows=200000 loops=1)
                     ->  Seq Scan on t1
                                 (cost=0.00..2110.00 rows=100000 width=23)
                                 (actual time=0.006..18.819 rows=100000 loops=1)
                     ->  Seq Scan on t2
                                 (cost=0.00..4410.00 rows=100000 width=22)
                                 (actual time=1.550..22.119 rows=100000 loops=1)
 Total runtime: 119.976 ms
(9 lignes)
```

Avec MergeAppend :

```
 Limit  (cost=0.72..0.97 rows=5 width=23)
        (actual time=0.055..0.060 rows=5 loops=1)
   ->  Merge Append  (cost=0.72..14866.72 rows=300000 width=23)
                     (actual time=0.053..0.058 rows=5 loops=1)
         Sort Key: t1.a
         ->  Index Scan using t1_pkey on t1
                           (cost=0.29..3642.29 rows=100000 width=22)
                           (actual time=0.033..0.036 rows=3 loops=1)
         ->  Index Scan using t2_pkey on t2
                           (cost=0.42..7474.42 rows=200000 width=23) =
                           (actual time=0.019..0.021 rows=3 loops=1)
 Total runtime: 0.117 ms
```

On voit ici que chacun des parcours d'index renvoie 3 lignes, ce qui est suffisant pour renvoyer les 5 lignes ayant la plus faible valeur pour a.

6.12.16 AUTRES

- Nœud HashSetOp Except
  - instructions EXCEPT et EXCEPT ALL
- Nœud HashSetOp Intersect
  - instructions INTERSECT et INTERSECT ALL

La clause `UNION` permet de concaténer deux ensembles de données. Les clauses `EXCEPT` et `INTERSECT` permettent de supprimer une partie de deux ensembles de données.

Voici un exemple basé sur `EXCEPT` :

```
b1=# EXPLAIN SELECT oid FROM pg_proc
    EXCEPT  SELECT oid FROM pg_proc;
                               QUERY PLAN
--------------------------------------------------------------------
 HashSetOp Except  (cost=0.00..219.39 rows=2342 width=4)
   ->  Append  (cost=0.00..207.68 rows=4684 width=4)
         ->  Subquery Scan on "*SELECT* 1"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
         ->  Subquery Scan on "*SELECT* 2"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
(6 rows)
```

Et un exemple basé sur `INTERSECT` :

```
b1=# EXPLAIN   SELECT oid FROM pg_proc
    INTERSECT SELECT oid FROM pg_proc;
                               QUERY PLAN
--------------------------------------------------------------------
 HashSetOp Intersect  (cost=0.00..219.39 rows=2342 width=4)
   ->  Append  (cost=0.00..207.68 rows=4684 width=4)
         ->  Subquery Scan on "*SELECT* 1"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
         ->  Subquery Scan on "*SELECT* 2"
             (cost=0.00..103.84 rows=2342 width=4)
               ->  Seq Scan on pg_proc
                   (cost=0.00..80.42 rows=2342 width=4)
(6 rows)
```

6.12.17 DIVERS

- Prend un ensemble de données en entrée
- Et renvoie un ensemble de données
- Exemples typiques
  - Sort
  - Aggregate
  - Unique
  - Limit
  - InitPlan, SubPlan

Tous les autres nœuds que nous allons voir prennent un seul ensemble de données en entrée et en renvoient un aussi. Ce sont des nœuds d'opérations simples comme le tri, l'agrégat, l'unicité, la limite, etc.

6.12.18 SORT

- Utilisé pour le ORDER BY
  - Mais aussi DISTINCT, GROUP BY, UNION
  - Les jointures de type Merge Join
- Gros délai de démarrage
- Trois types de tri
  - En mémoire, tri quicksort
  - En mémoire, tri top-N heapsort (si clause LIMIT)
  - Sur disque

PostgreSQL peut faire un tri de trois façons.

Les deux premières sont manuelles. Il lit toutes les données nécessaires et les trie en mémoire. La quantité de mémoire utilisable dépend du paramètre `work_mem`. S'il n'a pas assez de mémoire, il utilisera un stockage sur disque. La rapidité du tri dépend principalement de la mémoire utilisable mais aussi de la puissance des processeurs. Le tri effectué est un tri quicksort sauf si une clause `LIMIT` existe, auquel cas, le tri sera un top-N heapsort. La troisième méthode est de passer par un index Btree. En effet, ce type d'index stocke les données de façon triée. Dans ce cas, PostgreSQL n'a pas besoin de mémoire.

Le choix entre ces trois méthodes dépend principalement de `work_mem`. En fait, le pseudo-code ci-dessous explique ce choix :

```
Si les données de tri tiennent dans work_mem
    Si une clause LIMIT est présente
```

```
        Tri top-N heapsort
    Sinon
        Tri quicksort
Sinon
    Tri sur disque
```

Voici quelques exemples :

- un tri externe

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
                              QUERY PLAN
----------------------------------------------------------------------
 Sort  (cost=150385.45..153040.45 rows=1062000 width=4)
       (actual time=807.603..941.357 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: external sort  Disk: 17608kB
   ->  Seq Scan on t2  (cost=0.00..15045.00 rows=1062000 width=4)
                (actual time=0.050..143.918 rows=1000000 loops=1)
 Total runtime: 1021.725 ms
(5 rows)
```

- un tri en mémoire

```
b1=# SET work_mem TO '100MB';
SET
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id;
                              QUERY PLAN
----------------------------------------------------------------------
 Sort  (cost=121342.45..123997.45 rows=1062000 width=4)
       (actual time=308.129..354.035 rows=1000000 loops=1)
   Sort Key: id
   Sort Method: quicksort  Memory: 71452kB
   ->  Seq Scan on t2  (cost=0.00..15045.00 rows=1062000 width=4)
                (actual time=0.088..142.787 rows=1000000 loops=1)
 Total runtime: 425.160 ms
(5 rows)
```

- un tri en mémoire

```
b1=# EXPLAIN ANALYZE SELECT 1 FROM t2 ORDER BY id LIMIT 10000;
                                 QUERY PLAN
----------------------------------------------------------------------
 Limit  (cost=85863.56..85888.56 rows=10000 width=4)
        (actual time=271.674..272.980 rows=10000 loops=1)
   ->  Sort  (cost=85863.56..88363.56 rows=1000000 width=4)
             (actual time=271.671..272.240 rows=10000 loops=1)
         Sort Key: id
         Sort Method: top-N heapsort  Memory: 1237kB
```

```
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                        (actual time=0.031..146.306 rows=1000000 loops=1)
 Total runtime: 273.665 ms
(6 rows)
```

- un tri par un index

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

Le paramètre `enable_sort` permet de défavoriser l'utilisation d'un tri. Dans ce cas, le planificateur tendra à préférer l'utilisation d'un index, qui retourne des données déjà triées.

Augmenter la valeur du paramètre `work_mem` aura l'effet inverse : favoriser un tri plutôt que l'utilisation d'un index.

6.12.19 AGGREGATE

- Agrégat complet
- Pour un seul résultat

Il existe plusieurs façons de réaliser un agrégat :

- l'agrégat standard;
- l'agrégat par tri des données;
- et l'agrégat par hachage;

ces deux derniers sont utilisés quand la clause `SELECT` contient des colonnes en plus de la fonction d'agrégat.

Par exemple, pour un seul résultat count(*), nous aurons ce plan d'exécution :

```
b1=# EXPLAIN SELECT count(*) FROM pg_proc;
                           QUERY PLAN
-----------------------------------------------------------------
 Aggregate  (cost=86.28..86.29 rows=1 width=0)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=0)
(2 rows)
```

17.12

Seul le parcours séquentiel est possible ici car count() doit compter toutes les lignes.

Autre exemple avec une fonction d'agrégat max.

```
b1=# EXPLAIN SELECT max(proname) FROM pg_proc;
                               QUERY PLAN
------------------------------------------------------------------
 Aggregate  (cost=92.13..92.14 rows=1 width=64)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
(2 rows)
```

Il existe une autre façon de récupérer la valeur la plus petite ou la plus grande : passer par l'index. Ce sera très rapide car l'index est trié.

```
b1=# EXPLAIN SELECT max(oid) FROM pg_proc;
                               QUERY PLAN
------------------------------------------------------------------------
 Result  (cost=0.13..0.14 rows=1 width=0)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.13 rows=1 width=4)
           ->  Index Scan Backward using pg_proc_oid_index on pg_proc
               (cost=0.00..305.03 rows=2330 width=4)
                 Index Cond: (oid IS NOT NULL)
(5 rows)
```

Il est à noter que ce n'est pas valable pour les valeurs de type booléen jusqu'en 9.2.

6.12.20 HASH AGGREGATE

- Hachage de chaque n-uplet de regroupement (group by)
- accès direct à chaque n-uplet pour appliquer fonction d'agrégat
- Intéressant si l'ensemble des valeurs distinctes tient en mémoire, dangereux sinon

Voici un exemple de ce type de nœud :

```
b1=# EXPLAIN SELECT proname, count(*) FROM pg_proc GROUP BY proname;
                               QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=92.13..111.24 rows=1911 width=64)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
(2 rows)
```

Le hachage occupe de la place en mémoire, le plan n'est choisi que si PostgreSQL estime que si la table de hachage générée tient dans work_mem. **C'est le seul type de nœud qui peut dépasser work_mem :** la seule façon d'utiliser le HashAggregate est en mémoire, il est donc agrandi s'il est trop petit.

Quant au paramètre `enable_hashagg`, il permet d'activer et de désactiver l'utilisation de ce type de nœud.

6.12.21 GROUP AGGREGATE

- Reçoit des données déjà triées
- Parcours des données
  - Regroupement du groupe précédent arrivé à une donnée différente

Voici un exemple de ce type de nœud :

```
b1=# EXPLAIN SELECT proname, count(*) FROM pg_proc GROUP BY proname;
                                QUERY PLAN
------------------------------------------------------------------------
 GroupAggregate  (cost=211.50..248.17 rows=1911 width=64)
   ->  Sort  (cost=211.50..217.35 rows=2342 width=64)
         Sort Key: proname
         ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=64)
(4 rows)
```

Un parcours d'index est possible pour remplacer le parcours séquentiel et le tri.

6.12.22 UNIQUE

- Reçoit des données déjà triées
- Parcours des données
  - Renvoi de la donnée précédente une fois arrivé à une donnée différente
- Résultat trié

Le nœud `Unique` permet de ne conserver que les lignes différentes. L'opération se réalise en triant les données, puis en parcourant le résultat trié. Là aussi, un index aide à accélérer ce type de nœud.

En voici un exemple :

```
b1=# EXPLAIN SELECT DISTINCT pronamespace FROM pg_proc;
                               QUERY PLAN
-----------------------------------------------------------------------
 Unique  (cost=211.57..223.28 rows=200 width=4)
   ->  Sort  (cost=211.57..217.43 rows=2343 width=4)
         Sort Key: pronamespace
```

```
        ->  Seq Scan on sample4  (cost=0.00..80.43 rows=2343 width=4)
(4 rows)
```

6.12.23 LIMIT

- Permet de limiter le nombre de résultats renvoyés
- Utilisé par
  - clauses LIMIT et OFFSET d'une requête SELECT
  - fonctions min() et max() quand il n'y a pas de clause WHERE et qu'il y a un index
- Le nœud précédent sera de préférence un nœud dont le coût de démarrage est peu élevé (SeqScan, NestedLoop)

Voici un exemple de l'utilisation d'un nœud `Limit` :

```
b1=# EXPLAIN SELECT 1 FROM pg_proc LIMIT 10;
                           QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=0.00..0.34 rows=10 width=0)
   ->  Seq Scan on pg_proc  (cost=0.00..80.42 rows=2342 width=0)
(2 rows)
```

6.13 TRAVAUX PRATIQUES

6.13.1 ÉNONCÉS

Préambule

- Utilisez `\timing` dans `psql` pour afficher les temps d'exécution de la recherche.

- Afin d'éviter tout effet dû au cache, autant du plan que des pages de données, nous utilisons parfois une sous-requête avec un résultat non déterministe (`random`).

- N'oubliez pas de lancer plusieurs fois les requêtes. Vous pouvez les rappeler avec `\g`, ou utiliser la touche **flèche haut** du clavier si votre installation utilise `readline` ou `libedit`.

- Vous devrez disposer de la base `cave` pour ce TP.
- Les valeurs (taille, temps d'exécution) varieront à cause de plusieurs critères :
  - les machines sont différentes ;
  - le jeu de données peut avoir partiellement changé depuis la rédaction du TP.

Affichage de plans de requêtes simples

Recherche de motif texte

- Affichez le plan de cette requête (sur la base `cave`) :

```
SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
```

Que constatez-vous ?

- Affichez maintenant le nombre de blocs accédés par cette requête.
- Cette requête ne passe pas par un index. Essayez de lui forcer la main.
- L'index n'est toujours pas utilisé. L'index « par défaut » n'est pas capable de répondre à des questions sur motif.
- Créez un index capable de réaliser ces opérations. Testez à nouveau le plan.
- Réactivez `enable_seqscan`. Testez à nouveau le plan.
- Quelle est la conclusion ?

Recherche de motif texte avancé

La base `cave` ne contient pas de données textuelles appropriées, nous allons en utiliser une autre.

- Lancez `textes.sql` ou `textes_10pct.sql` (préférable sur une machine peu puissante, ou une instance PostgreSQL non paramétrée).

```
psql < textes_10pct.sql
```

Ce script crée une table `textes`, contenant le texte intégral d'un grand nombre de livres en français du projet Gutenberg, soit 10 millions de lignes pour 85 millions de mots.

Nous allons rechercher toutes les références à « Fantine » dans les textes. On devrait trouver beaucoup d'enregistrements provenant des « Misérables ».

- La méthode SQL standard pour écrire cela est :

17.12

```
SELECT * FROM textes WHERE contenu ILIKE '%fantine%';
```

Exécutez cette requête, et regardez son plan d'exécution.

Nous lisons toute la table à chaque fois. C'est normal et classique avec une base de données : non seulement la recherche est insensible à la casse, mais elle commence par `%`, ce qui est incompatible avec une indexation btree classique.

Nous allons donc utiliser l'extension `pg_trgm` :

- Créez un index trigramme :

```
textes=# CREATE EXTENSION pg_trgm;
CREATE INDEX idx_trgm ON textes USING gist (contenu gist_trgm_ops);
-- ou CREATE INDEX idx_trgm ON textes USING gin (contenu gin_trgm_ops);
```

- Quelle est la taille de l'index ?
- Réexécutez la requête. Que constatez-vous ?
- Suivant que vous ayez opté pour GiST ou Gin, refaites la manipulation avec l'autre méthode d'indexation.
- Essayez de créer un index « Full Text » à la place de l'index trigramme. Quels sont les résultats ?

Optimisation d'une requête

Schéma de la base cave

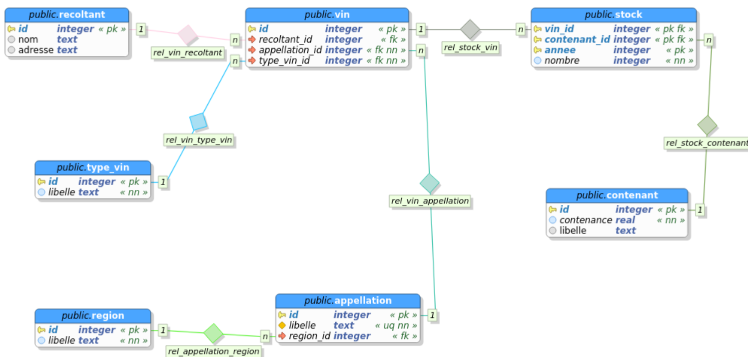


FIGURE 7: SCHÉMA DE LA BASE CAVE

DALIBO

Optimisation 1

Nous travaillerons sur la requête contenue dans le fichier `requete1.sql` pour cet exercice :

```sql
-- \timing

-- explain analyze
select
       m.annee||' - '||a.libelle as millesime_region,
       sum(s.nombre) as contenants,
       sum(s.nombre*c.contenance) as litres
from
      contenant c
      join stock s
        on s.contenant_id = c.id
      join (select round(random()*50)+1950 as annee) m
        on s.annee =  m.annee
      join vin v
        on s.vin_id = v.id
      left join appellation a
      on v.appellation_id = a.id
group by m.annee||' - '||a.libelle;
```

- Exécuter la requête telle quelle et noter le plan et le temps d'exécution.
- Créer un index sur la colonne `stock.annee`.
- Exécuter la requête juste après la création de l'index
- Faire un `ANALYZE stock`.
- Exécuter à nouveau la requête.
- Interdire à PostgreSQL les *sequential scans* avec la commande `set enable_seqscan to off ;` dans votre session dans `psql`.
- Exécuter à nouveau la requête.
- Tenter de réécrire la requête pour l'optimiser.

Optimisation 2

L'exercice nous a amené à la réécriture de la requête

- Voici la requête que nous avons à présent :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     left join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Cette écriture n'est pas optimale, pourquoi ?

Indices

- Vérifiez le schéma de données de la base `cave`.
- Faites les requêtes de vérification nécessaires pour vous assurer que vous avez bien trouvé une anomalie dans la requête.
- Réécrivez la requête une nouvelle fois et faites un `EXPLAIN ANALYZE` pour vérifier que le plan d'exécution est plus simple et plus rapide avec cette nouvelle écriture.

Optimisation 3

Un dernier problème existe dans cette requête. Il n'est visible qu'en observant le plan d'exécution de la requête précédente.

Indice

Cherchez une opération présente dans le plan qui n'apparaît pas dans la requête. Comment modifier la requête pour éviter cette opération ?

Corrélation entre colonnes

- Importez le fichier `correlations.sql`.

Dans la table `villes`, on trouve les villes et leur code postal. Ces colonnes sont très fortement corrélées, mais pas identiques : plusieurs villes peuvent partager le même code postal, et une ville peut avoir plusieurs codes postaux. On peut aussi, bien sûr, avoir

plusieurs villes avec le même nom, mais pas le même code postal (dans des départements différents par exemple). Pour obtenir la liste des villes pouvant poser problème :

```
SELECT *
FROM villes
WHERE localite IN
    (SELECT localite
     FROM villes
     GROUP BY localite HAVING count(*) >1)
  AND codepostal IN
    (SELECT codepostal
     FROM villes
     GROUP BY codepostal HAVING count(*) >1);
```

Avec cette requête, on récupére toutes les villes ayant plusieurs occurrences et dont au moins une possède un code postal partagé. Ces villes ont donc besoin du code postal ET du nom pour être identifiées.

Un exemple de requête problématique est le suivant :

```
SELECT * FROM colis
WHERE id_ville IN
   (SELECT id_ville FROM villes
    WHERE localite ='PARIS'
      AND codepostal LIKE '75%')
```

- Exécutez cette requête, et regardez son plan d'exécution. Où est le problème ?
- Exécutez cette requête sans la dernière clause `AND codepostal LIKE '75%'`. Que constatez-vous ?
- Quelle solution pourrait-on adopter, si on doit réellement spécifier ces deux conditions ?

6.13.2 SOLUTIONS

Affichage de plans de requêtes simples

Recherche de motif texte

- Affichez le plan de cette requête (sur la base `cave`).

```
SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
```

```
cave=# explain SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=1 width=24)
```

```
   Filter: (libelle ~~ 'Brouilly%'::text)
(2 lignes)
```

Que constatez-vous ?

- Affichez maintenant le nombre de blocs accédés par cette requête.

```
cave=# explain (analyze,buffers) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                      QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=1 width=24)
                          (actual time=0.066..0.169 rows=1 loops=1)
   Filter: (libelle ~~ 'Brouilly%'::text)
   Rows Removed by Filter: 318
   Buffers: shared hit=3
 Total runtime: 0.202 ms
(5 lignes)
```

- Cette requête ne passe pas par un index. Essayez de lui forcer la main.

```
cave=# set enable_seqscan TO off;
SET
cave=# explain (analyze,buffers) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                      QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on appellation  (cost=10000000000.00..10000000006.99 rows=1 width=24)
                          (actual time=0.073..0.197 rows=1 loops=1)
   Filter: (libelle ~~ 'Brouilly%'::text)
   Rows Removed by Filter: 318
   Buffers: shared hit=3
 Total runtime: 0.238 ms
(5 lignes)
```

Passer `enable_seqscan` à « off » n'interdit pas l'utilisation des scans séquentiels. Il ne fait que les défavoriser fortement : regardez le coût estimé du scan séquentiel.

- L'index n'est toujours pas utilisé. L'index « par défaut » n'est pas capable de répondre à des questions sur motif.

En effet, l'index par défaut trie les données par la collation de la colonne de la table. Il lui est impossible de savoir que `libelle LIKE 'Brouilly%'` est équivalent à `libelle >= 'Brouilly' AND libelle < 'Brouillz'`. Ce genre de transformation n'est d'ailleurs pas forcément trivial, ni même possible. Il existe dans certaines langues des équivalences (ß et ss en allemand par exemple) qui rendent ce genre de transformation au mieux hasardeuse.

- Créez un index capable de ces opérations. Testez à nouveau le plan.

Pour pouvoir répondre à cette question, on doit donc avoir un index spécialisé, qui compare les chaînes non plus par rapport à leur collation, mais à leur valeur binaire (octale en fait).

```
CREATE INDEX appellation_libelle_key_search
    ON appellation (libelle text_pattern_ops);
```

On indique par cette commande à PostgreSQL de ne plus utiliser la classe d'opérateurs habituelle de comparaison de texte, mais la classe `text_pattern_ops`, qui est spécialement faite pour les recherches `LIKE 'xxxx%'` : cette classe ne trie plus les chaînes par leur ordre alphabétique, mais par leur valeur octale.

Si on redemande le plan :

```
cave=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using appellation_libelle_key_search on appellation
                              (cost=0.27..8.29 rows=1 width=24)
                              (actual time=0.057..0.059 rows=1 loops=1)
   Index Cond: ((libelle ~>=~ 'Brouilly'::text)
                AND (libelle ~<~ 'Brouillz'::text))
   Filter: (libelle ~~ 'Brouilly%'::text)
   Buffers: shared hit=1 read=2
 Total runtime: 0.108 ms
(5 lignes)
```

On utilise enfin un index.

- Réactivez `enable_seqscan`. Testez à nouveau le plan.

```
cave=# reset enable_seqscan ;
RESET
cave=# explain (analyze,buffers) SELECT * FROM appellation
cave=# WHERE libelle LIKE 'Brouilly%';
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on appellation  (cost=0.00..6.99 rows=1 width=24)
                          (actual time=0.063..0.172 rows=1 loops=1)
   Filter: (libelle ~~ 'Brouilly%'::text)
   Rows Removed by Filter: 318
   Buffers: shared hit=3
 Total runtime: 0.211 ms
(5 lignes)
```

- Quelle est la conclusion ?

PostgreSQL choisit de ne pas utiliser cet index. Le temps d'exécution est pourtant un peu

meilleur avec l'index (60 microsecondes contre 172 microsecondes). Néanmoins, cela n'est vrai que parce que les données sont en cache. En cas de données hors du cache, le plan par parcours séquentiel (*seq scan*) est probablement meilleur. Certes il prend plus de temps CPU puisqu'il doit consulter 318 enregistrements inutiles. Par contre, il ne fait qu'un accès à 3 blocs séquentiels (les 3 blocs de la table), ce qui est le plus sûr.

La table est trop petite pour que PostgreSQL considère l'utilisation d'un index.

Recherche de motif texte avancé

La base `cave` ne contient pas de données textuelles appropriées, nous allons en utiliser une autre.

- Lancez `textes.sql` ou `textes_10pct.sql` (préférable sur une machine peu puissante, ou une instance PostgreSQL non paramétrée).

```
psql < textes.sql
```

Ce script crée une table `textes`, contenant le texte intégral d'un grand nombre de livres en français du projet Gutenberg, soit 10 millions de lignes pour 85 millions de mots.

Nous allons rechercher toutes les références à « Fantine » dans les textes. On devrait trouver beaucoup d'enregistrements provenant des « Misérables ».

- La méthode SQL standard pour écrire cela est :

```
SELECT * FROM textes WHERE contenu ILIKE '%fantine%';
```

Exécutez cette requête, et regardez son plan d'exécution.

```
textes=# explain (analyze,buffers) SELECT * FROM textes
textes=# WHERE contenu ILIKE '%fantine%';
                                         QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on textes  (cost=0.00..325809.40 rows=874 width=102)
                     (actual time=224.634..22567.231 rows=921 loops=1)
   Filter: (contenu ~~* '%fantine%'::text)
   Rows Removed by Filter: 11421523
   Buffers: shared hit=130459 read=58323
 Total runtime: 22567.679 ms
(5 lignes)
```

Cette requête ne peut pas être optimisée avec les index standard (`btree`) : c'est une recherche insensible à la casse et avec plusieurs % dont un au début.

Avec GiST

- Créez un index trigramme:

```
textes=# CREATE EXTENSION pg_trgm;

textes=# CREATE INDEX idx_trgm ON textes USING gist (contenu gist_trgm_ops);
CREATE INDEX
Temps : 962794,399 ms
```

- Quelle est la taille de l'index ?

L'index fait cette taille (pour une table de 1,5Go) :

```
textes=# select pg_size_pretty(pg_relation_size('idx_trgm'));
 pg_size_pretty
----------------
 2483 MB
(1 ligne)
```

- Réexécutez la requête. Que constatez-vous ?

```
textes=# explain (analyze,buffers) SELECT * FROM textes
textes=# WHERE contenu ILIKE '%fantine%';
                                          QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=111.49..3573.39 rows=912 width=102)
                             (actual time=1942.872..1949.393 rows=922 loops=1)
   Recheck Cond: (contenu ~~* '%fantine%'::text)
   Rows Removed by Index Recheck: 75
   Buffers: shared hit=16030 read=144183 written=14741
   ->  Bitmap Index Scan on idx_trgm  (cost=0.00..111.26 rows=912 width=0)
                            (actual time=1942.671..1942.671 rows=997 loops=1)
         Index Cond: (contenu ~~* '%fantine%'::text)
         Buffers: shared hit=16029 read=143344 written=14662
 Total runtime: 1949.565 ms
(8 lignes)

Temps : 1951,175 ms
```

PostgreSQL dispose de mécanismes spécifiques avancés pour certains types de données. Ils ne sont pas toujours installés en standard, mais leur connaissance peut avoir un impact énorme sur les performances.

Le mécanisme GiST est assez efficace pour répondre à ce genre de questions. Il nécessite quand même un accès à un grand nombre de blocs, d'après le plan : 160 000 blocs lus, 15 000 écrits (dans un fichier temporaire, on pourrait s'en débarrasser en augmentant le `work_mem`). Le gain est donc conséquent, mais pas gigantesque : le plan initial lisait 190 000 blocs. On gagne surtout en temps de calcul, car on accède directement aux bons enregistrements. Le parcours de l'index, par contre, est coûteux.

Avec Gin

- Créez un index trigramme:

```
textes=# CREATE EXTENSION pg_trgm;

textes=# CREATE INDEX idx_trgm ON textes USING gin (contenu gin_trgm_ops);
CREATE INDEX
Temps : 591534,917 ms
```

L'index fait cette taille (pour une table de 1,5Go) :

```
textes=# select pg_size_pretty(pg_total_relation_size('textes'));
 pg_size_pretty
----------------
 4346 MB
(1 ligne)
```

L'index est très volumineux.

- Réexécutez la requête. Que constatez-vous ?

```
textes=# explain (analyze,buffers) SELECT * FROM textes
textes=# WHERE contenu ILIKE '%fantine%';
                                              QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=103.06..3561.22 rows=911 width=102)
                             (actual time=777.469..780.834 rows=921 loops=1)
   Recheck Cond: (contenu ~~* '%fantine%'::text)
   Rows Removed by Index Recheck: 75
   Buffers: shared hit=2666
   ->  Bitmap Index Scan on idx_trgm  (cost=0.00..102.83 rows=911 width=0)
                            (actual time=777.283..777.283 rows=996 loops=1)
         Index Cond: (contenu ~~* '%fantine%'::text)
         Buffers: shared hit=1827
 Total runtime: 780.954 ms
(8 lignes)
```

PostgreSQL dispose de mécanismes spécifiques avancés pour certains types de données. Ils ne sont pas toujours installés en standard, mais leur connaissance peut avoir un impact énorme sur les performances. Le mécanisme Gin est vraiment très efficace pour répondre à ce genre de questions. Il s'agit de répondre en moins d'une seconde à « quelles lignes contiennent la chaîne "fantine" ? » sur 12 millions de lignes de texte. Les Index Gin sont par contre très coûteux à maintenir. Ici, on n'accède qu'à 2 666 blocs, ce qui est vraiment excellent. Mais l'index est bien plus volumineux que l'index GiST.

Avec le Full Text Search

Le résultat sera bien sûr différent, et le FTS est moins souple.

Version GiST :

```
textes=# create index idx_fts
           on textes
           using gist (to_tsvector('french',contenu));
CREATE INDEX
Temps : 1807467,811 ms


textes=# EXPLAIN (analyze,buffers) SELECT * FROM textes
textes=# WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');
                 QUERY PLAN
------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=2209.51..137275.87 rows=63109 width=97)
                             (actual time=648.596..659.733 rows=311 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu) @@
                  '''fantin'''::tsquery)
   Buffers: shared hit=37165
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..2193.74 rows=63109 width=0)
                                (actual time=648.493..648.493 rows=311 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu) @@
                      '''fantin'''::tsquery)
         Buffers: shared hit=37016
 Total runtime: 659.820 ms
(7 lignes)

Temps : 660,364 ms
```

Et la taille de l'index :

```
textes=# select pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 671 MB
(1 ligne)
```

Version Gin :

```
textes=# CREATE INDEX idx_fts ON textes
textes=# USING gin (to_tsvector('french',contenu));
CREATE INDEX
Temps : 491499,599 ms
textes=# EXPLAIN (analyze,buffers) SELECT * FROM textes
textes=# WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');
                                                         QUERY PLAN
```

```
------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=693.10..135759.45 rows=63109 width=97)
                             (actual time=0.278..0.699 rows=311 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu) @@
                 '''fantin'''::tsquery)
   Buffers: shared hit=153
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..677.32 rows=63109 width=0)
                                     (actual time=0.222..0.222 rows=311 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu) @@
                     '''fantin'''::tsquery)
         Buffers: shared hit=4
 Total runtime: 0.793 ms
(7 lignes)

Temps : 1,534 ms
```

Taille de l'index :

```
textes=# select pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 593 MB
(1 ligne)
```

On constate donc que le Full Text Search est bien plus efficace que le trigramme, du moins pour le Full Text Search + Gin : trouver 1 mot parmi plus de cent millions, dans 300 endroits différents dure 1,5 ms.

Par contre, le trigramme permet des recherches floues (orthographe approximative), et des recherches sur autre chose que des mots, même si ces points ne sont pas abordés ici.

Optimisation d'une requête

Optimisation 1

Nous travaillerons sur la requête contenue dans le fichier `requete1.sql` pour cet exercice:

```
-- \timing

-- explain analyze
select
      m.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
```

```
from
      contenant c
      join stock s
        on s.contenant_id = c.id
      join (select round(random()*50)+1950 as annee) m
        on s.annee =  m.annee
      join vin v
        on s.vin_id = v.id
      left join appellation a
      on v.appellation_id = a.id
group by m.annee||' - '||a.libelle;
```

L'exécution de la requête donne le plan suivant, avec un temps qui peut varier en fonction de la machine utilisée et de son activité:

```
HashAggregate  (cost=12763.56..12773.13 rows=319 width=32)
               (actual time=1542.472..1542.879 rows=319 loops=1)
   ->  Hash Left Join  (cost=184.59..12741.89 rows=2889 width=32)
                       (actual time=180.263..1520.812 rows=11334 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=174.42..12663.10 rows=2889 width=20)
                        (actual time=179.426..1473.270 rows=11334 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=173.37..12622.33 rows=2889 width=20)
                              (actual time=179.401..1446.687 rows=11334 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Hash Join  (cost=0.04..12391.22 rows=2889 width=20)
                            (actual time=164.388..1398.643 rows=11334 loops=1)
                           Hash Cond: ((s.annee)::double precision =
                                ((round((random() * 50::double precision)) +
                                1950::double precision)))
                           ->  Seq Scan on stock s
                                (cost=0.00..9472.86 rows=577886 width=16)
                                (actual time=0.003..684.039 rows=577886 loops=1)
                           ->  Hash  (cost=0.03..0.03 rows=1 width=8)
                                     (actual time=0.009..0.009 rows=1 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                 ->  Result  (cost=0.00..0.02 rows=1 width=0)
                                     (actual time=0.005..0.006 rows=1 loops=1)
                     ->  Hash  (cost=97.59..97.59 rows=6059 width=8)
                               (actual time=14.987..14.987 rows=6059 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 237kB
```

```
                                ->  Seq Scan on vin v
                                      (cost=0.00..97.59 rows=6059 width=8)
                                      (actual time=0.009..7.413 rows=6059 loops=1)
                  ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                            (actual time=0.013..0.013 rows=2 loops=1)
                        Buckets: 1024  Batches: 1  Memory Usage: 1kB
                        ->  Seq Scan on contenant c
                               (cost=0.00..1.02 rows=2 width=8)
                               (actual time=0.003..0.005 rows=2 loops=1)
            ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                      (actual time=0.806..0.806 rows=319 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 17kB
                  ->  Seq Scan on appellation a
                       (cost=0.00..6.19 rows=319 width=20)
                       (actual time=0.004..0.379 rows=319 loops=1)
 Total runtime: 1543.242 ms
(23 rows)
```

Nous créons à présent un index sur `stock.annee` comme suit :

```
create index stock_annee on stock (annee) ;
```

Et exécutons à nouveau la requête. Hélas nous constatons que rien ne change, ni le plan, ni le temps pris par la requête.

Nous n'avons pas lancé `ANALYZE`, cela explique que l'optimiseur n'utilise pas l'index : il n'en a pas encore la connaissance.

```
ANALYZE STOCK ;
```

Le plan n'a toujours pas changé ! Ni le temps d'exécution ?!

Interdisons donc de faire les *seq scans* à l'optimiseur :

```
SET ENABLE_SEQSCAN TO OFF;
```

Nous remarquons que le plan d'exécution est encore pire :

```
 HashAggregate  (cost=40763.39..40772.96 rows=319 width=32)
                (actual time=2022.971..2023.390 rows=319 loops=1)
   ->  Hash Left Join  (cost=313.94..40741.72 rows=2889 width=32)
                       (actual time=18.149..1995.889 rows=11299 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=290.92..40650.09 rows=2889 width=20)
                        (actual time=17.172..1937.644 rows=11299 loops=1)
               Hash Cond: (s.vin_id = v.id)
```

```
               ->  Nested Loop  (cost=0.04..40301.43 rows=2889 width=20)
                                (actual time=0.456..1882.531 rows=11299 loops=1)
                     Join Filter: (s.contenant_id = c.id)
                     ->  Hash Join  (cost=0.04..40202.48 rows=2889 width=20)
                                (actual time=0.444..1778.149 rows=11299 loops=1)
                           Hash Cond: ((s.annee)::double precision =
                                ((round((random() * 50::double precision)) +
                                1950::double precision))
                           ->  Index Scan using stock_pkey on stock s
                                (cost=0.00..37284.12 rows=577886 width=16)
                                (actual time=0.009..1044.061 rows=577886 loops=1)
                           ->  Hash  (cost=0.03..0.03 rows=1 width=8)
                                     (actual time=0.011..0.011 rows=1 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                 ->  Result  (cost=0.00..0.02 rows=1 width=0)
                                      (actual time=0.005..0.006 rows=1 loops=1)
                     ->  Materialize  (cost=0.00..12.29 rows=2 width=8)
                                (actual time=0.001..0.003 rows=2 loops=11299)
                           ->  Index Scan using contenant_pkey on contenant c
                                    (cost=0.00..12.28 rows=2 width=8)
                                    (actual time=0.004..0.010 rows=2 loops=1)
               ->  Hash  (cost=215.14..215.14 rows=6059 width=8)
                         (actual time=16.699..16.699 rows=6059 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 237kB
                     ->  Index Scan using vin_pkey on vin v
                             (cost=0.00..215.14 rows=6059 width=8)
                             (actual time=0.010..8.871 rows=6059 loops=1)
         ->  Hash  (cost=19.04..19.04 rows=319 width=20)
                   (actual time=0.936..0.936 rows=319 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 17kB
               ->  Index Scan using appellation_pkey on appellation a
                        (cost=0.00..19.04 rows=319 width=20)
                        (actual time=0.016..0.461 rows=319 loops=1)
 Total runtime: 2023.742 ms
(22 rows)
```

Que faire alors ?

Il convient d'autoriser à nouveau les *seq scan*, puis, peut-être, de réécrire la requête.

Nous réécrivons la requête comme suit (fichier `requete2.sql`) :

17.12

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     left join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Il y a une jointure en moins, ce qui est toujours appréciable. Nous pouvons faire cette réécriture parce que la requête `select round(random()*50)+1950 as annee` ne ramène qu'un seul enregistrement.

Voici le résultat :

```
 HashAggregate  (cost=12734.64..12737.10 rows=82 width=28)
                (actual time=265.899..266.317 rows=319 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=0)
                 (actual time=0.005..0.006 rows=1 loops=1)
   ->  Hash Left Join  (cost=184.55..12712.96 rows=2889 width=28)
                       (actual time=127.787..245.314 rows=11287 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=174.37..12634.17 rows=2889 width=16)
                        (actual time=126.950..208.077 rows=11287 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=173.33..12593.40 rows=2889 width=16)
                              (actual time=126.925..181.867 rows=11287 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Seq Scan on stock s
                            (cost=0.00..12362.29 rows=2889 width=16)
                            (actual time=112.101..135.932 rows=11287 loops=1)
                           Filter: ((annee)::double precision = $0)
                     ->  Hash  (cost=97.59..97.59 rows=6059 width=8)
                               (actual time=14.794..14.794 rows=6059 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 237kB
```

```
                              ->  Seq Scan on vin v
                                    (cost=0.00..97.59 rows=6059 width=8)
                                    (actual time=0.010..7.321 rows=6059 loops=1)
                  ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                            (actual time=0.013..0.013 rows=2 loops=1)
                        Buckets: 1024  Batches: 1  Memory Usage: 1kB
                        ->  Seq Scan on contenant c
                               (cost=0.00..1.02 rows=2 width=8)
                               (actual time=0.004..0.006 rows=2 loops=1)
            ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                      (actual time=0.815..0.815 rows=319 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 17kB
                  ->  Seq Scan on appellation a
                       (cost=0.00..6.19 rows=319 width=20)
                       (actual time=0.004..0.387 rows=319 loops=1)
 Total runtime: 266.663 ms
(21 rows)
```

Nous sommes ainsi passés de 2 s à 250 ms : la requête est donc environ 10 fois plus rapide.

Que peut-on conclure de cet exercice ?

- que la création d'un index est une bonne idée ; cependant l'optimiseur peut ne pas l'utiliser, pour de bonnes raisons ;
- qu'interdire les *seq scan* est toujours une mauvaise idée (ne présumez pas de votre supériorité sur l'optimiseur !)

Optimisation 2

Voici la requête 2 telle que nous l'avons trouvé dans l'exercice précédent :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
```

```
     left join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

On peut se demander si la jointure externe (LEFT JOIN) est fondée… On va donc vérifier l'utilité de la ligne suivante :

```
vin v left join appellation a on v.appellation_id = a.id
```

Cela se traduit par « récupérer tous les tuples de la table vin, et pour chaque correspondance dans appellation, la récupérer, si elle existe ».

En regardant la description de la table `vin` (`\d vin` dans `psql`), on remarque la contrainte de clé étrangère suivante :

```
« vin_appellation_id_fkey »
  FOREIGN KEY (appellation_id)
  REFERENCES appellation(id)
```

Cela veut dire qu'on a la certitude que pour chaque vin, si une référence à la table appellation est présente, elle est nécessairement vérifiable.

De plus, on remarque :

```
appellation_id | integer | not null
```

Ce qui veut dire que la valeur de ce champ ne peut être nulle. Elle contient donc obligatoirement une valeur qui est présente dans la table `appellation`.

On peut vérifier au niveau des tuples en faisant un `COUNT(*)` du résultat, une fois en `INNER JOIN` et une fois en `LEFT JOIN`. Si le résultat est identique, la jointure externe ne sert à rien :

```
select count(*)
from vin v
  inner join appellation a on (v.appellation_id = a.id);

 count
-------
  6057

select count(*)
from vin v
  left join appellation a on (v.appellation_id = a.id);

 count
-------
  6057
```

On peut donc réécrire la requête 2 sans la jointure externe inutile, comme on vient de le démontrer :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     join appellation a
       on v.appellation_id = a.id
where s.annee =  (select round(random()*50)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Voici le résultat :

```
 HashAggregate  (cost=12734.64..12737.10 rows=82 width=28)
                (actual time=266.916..267.343 rows=319 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=0)
                 (actual time=0.005..0.006 rows=1 loops=1)
   ->  Hash Join  (cost=184.55..12712.96 rows=2889 width=28)
                  (actual time=118.759..246.391 rows=11299 loops=1)
         Hash Cond: (v.appellation_id = a.id)
         ->  Hash Join  (cost=174.37..12634.17 rows=2889 width=16)
                        (actual time=117.933..208.503 rows=11299 loops=1)
               Hash Cond: (s.contenant_id = c.id)
               ->  Hash Join  (cost=173.33..12593.40 rows=2889 width=16)
                              (actual time=117.914..182.501 rows=11299 loops=1)
                     Hash Cond: (s.vin_id = v.id)
                     ->  Seq Scan on stock s
                            (cost=0.00..12362.29 rows=2889 width=16)
                            (actual time=102.903..135.451 rows=11299 loops=1)
                           Filter: ((annee)::double precision = $0)
                     ->  Hash  (cost=97.59..97.59 rows=6059 width=8)
                               (actual time=14.979..14.979 rows=6059 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 237kB
                           ->  Seq Scan on vin v
```

```
                                (cost=0.00..97.59 rows=6059 width=8)
                                (actual time=0.010..7.387 rows=6059 loops=1)
                 ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                           (actual time=0.009..0.009 rows=2 loops=1)
                       Buckets: 1024  Batches: 1  Memory Usage: 1kB
                       ->  Seq Scan on contenant c
                              (cost=0.00..1.02 rows=2 width=8)
                              (actual time=0.002..0.004 rows=2 loops=1)
           ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                     (actual time=0.802..0.802 rows=319 loops=1)
                 Buckets: 1024  Batches: 1  Memory Usage: 17kB
                 ->  Seq Scan on appellation a
                      (cost=0.00..6.19 rows=319 width=20)
                      (actual time=0.004..0.397 rows=319 loops=1)
 Total runtime: 267.688 ms
(21 rows)
```

Cette réécriture n'a pas d'effet sur le temps d'exécution de la requête dans notre cas. Mais il est probable qu'avec des cardinalités différentes dans la base, cette réécriture aurait eu un impact. Remplacer un `LEFT JOIN` par un `JOIN` est le plus souvent intéressant, car il laisse davantage de liberté au moteur sur le sens de planification des requêtes.

Optimisation 3

Si on observe attentivement le plan, on constate qu'on a toujours le parcours séquentiel de la table `stock`, qui est notre plus grosse table. Pourquoi a-t-il lieu ?

Si on regarde le filtre (ligne `Filter`) du parcours de la table `stock`, on constate qu'il est écrit :

```
  Filter: ((annee)::double precision = $0)
```

Ceci signifie que pour tous les enregistrements de la table, l'année est convertie en nombre en double précision (un nombre à virgule flottante), afin d'être comparée à $0, une valeur filtre appliquée à la table. Cette valeur est le résultat du calcul :

```
select round(random()*50)+1950 as annee
```

comme indiquée par le début du plan (les lignes de l'initplan 1).

Pourquoi compare-t-il l'année, déclarée comme un entier (`integer`), en la convertissant en un nombre à virgule flottante ?

DALIBO

Parce que la fonction `round()` retourne un nombre à virgule flottante. La somme d'un nombre à virgule flottante et d'un entier est évidemment un nombre à virgule flottante. Si on veut que la fonction `round()` retourne un entier, il faut forcer explicitement sa conversion, via `CAST(xxx as int)` ou `::int`.

Réécrivons encore une fois cette requête :

```
explain analyze
select
      s.annee||' - '||a.libelle as millesime_region,
      sum(s.nombre) as contenants,
      sum(s.nombre*c.contenance) as litres
from
     contenant c
     join stock s
       on s.contenant_id = c.id
     join vin v
       on s.vin_id = v.id
     join appellation a
       on v.appellation_id = a.id
where s.annee =  (select cast(round(random()*50) as int)+1950 as annee)
group by s.annee||' - '||a.libelle;
```

Voici son plan :

```
 HashAggregate  (cost=1251.12..1260.69 rows=319 width=28)
                (actual time=138.418..138.825 rows=319 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=0)
                 (actual time=0.005..0.006 rows=1 loops=1)
   ->  Hash Join  (cost=267.86..1166.13 rows=11329 width=28)
                  (actual time=31.108..118.193 rows=11389 loops=1)
         Hash Cond: (s.contenant_id = c.id)
         ->  Hash Join  (cost=266.82..896.02 rows=11329 width=28)
                        (actual time=31.071..80.980 rows=11389 loops=1)
               Hash Cond: (s.vin_id = v.id)
               ->  Index Scan using stock_annee on stock s
                    (cost=0.00..402.61 rows=11331 width=16)
                    (actual time=0.049..17.191 rows=11389 loops=1)
                     Index Cond: (annee = $0)
               ->  Hash  (cost=191.08..191.08 rows=6059 width=20)
                         (actual time=31.006..31.006 rows=6059 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 313kB
                     ->  Hash Join  (cost=10.18..191.08 rows=6059 width=20)
```

```
                                        (actual time=0.814..22.856 rows=6059 loops=1)
                                   Hash Cond: (v.appellation_id = a.id)
                                   ->  Seq Scan on vin v
                                        (cost=0.00..97.59 rows=6059 width=8)
                                        (actual time=0.005..7.197 rows=6059 loops=1)
                                   ->  Hash  (cost=6.19..6.19 rows=319 width=20)
                                             (actual time=0.800..0.800 rows=319 loops=1)
                                         Buckets: 1024  Batches: 1  Memory Usage: 17kB
                                         ->  Seq Scan on appellation a
                                            (cost=0.00..6.19 rows=319 width=20)
                                            (actual time=0.002..0.363 rows=319 loops=1)
                 ->  Hash  (cost=1.02..1.02 rows=2 width=8)
                           (actual time=0.013..0.013 rows=2 loops=1)
                       Buckets: 1024  Batches: 1  Memory Usage: 1kB
                       ->  Seq Scan on contenant c  (cost=0.00..1.02 rows=2 width=8)
                                            (actual time=0.003..0.006 rows=2 loops=1)
 Total runtime: 139.252 ms
(21 rows)
```

On constate qu'on utilise enfin l'index de `stock`. Le temps d'exécution a encore été divisé par deux.

NB : ce problème d'incohérence de type était la cause du plus gros ralentissement de la requête. En reprenant la requête initiale, et en ajoutant directement le cast, la requête s'exécute déjà en 160 millisecondes.

Corrélation entre colonnes

Importez le fichier `correlations.sql`.

```
createdb correlations
psql correlations < correlations.sql
```

- Exécutez cette requête, et regardez son plan d'exécution. Où est le problème ?

Cette requête a été exécutée dans un environnement où le cache a été intégralement vidé, pour être dans la situation la plus défavorable possible. Vous obtiendrez probablement des performances meilleures, surtout si vous réexécutez cette requête.

```
explain (analyze,buffers)
   SELECT * FROM colis WHERE id_ville IN (
     SELECT id_ville
     FROM villes
     WHERE localite ='PARIS'
```

```
        AND codepostal LIKE '75%'
);
                                                          QUERY PLAN
--------------------------------------------------------------------------------
 Nested Loop  (cost=6.75..13533.81 rows=3265 width=16)
              (actual time=38.020..364383.516 rows=170802 loops=1)
   Buffers: shared hit=91539 read=82652
   I/O Timings: read=359812.828
   ->  Seq Scan on villes  (cost=0.00..1209.32 rows=19 width=
                           (actual time=23.979..45.383 rows=940 loops=1)
         Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
         Rows Removed by Filter: 54015
         Buffers: shared hit=1 read=384
         I/O Timings: read=22.326
   ->  Bitmap Heap Scan on colis (cost=6.75..682.88 rows=181 width=16)
                                 (actual time=1.305..387.239 rows=182 loops=940)
         Recheck Cond: (id_ville = villes.id_ville)
         Buffers: shared hit=91538 read=82268
         I/O Timings: read=359790.502
         ->  Bitmap Index Scan on idx_colis_ville
               (cost=0.00..6.70 rows=181 width=0)
               (actual time=0.115..0.115 rows=182 loops=940)
               Index Cond: (id_ville = villes.id_ville)
               Buffers: shared hit=2815 read=476
               I/O Timings: read=22.862
 Total runtime: 364466.458 ms
(17 lignes)
```

On constate que l'optimiseur part sur une boucle extrêmement coûteuse : 940 parcours sur `colis`, par `id_ville`. En moyenne, ces parcours durent environ 400 ms. Le résultat est vraiment très mauvais.

Il fait ce choix parce qu'il estime que la condition

```
localite ='PARIS' AND codepostal LIKE '75%'
```

va ramener 19 enregistrements. En réalité, elle en ramène 940, soit 50 fois plus, d'où un très mauvais choix. Pourquoi PostgreSQL fait-il cette erreur ?

```
marc=# EXPLAIN SELECT * FROM villes;
                          QUERY PLAN
--------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..934.55 rows=54955 width=27)
(1 ligne)
```

```
marc=# EXPLAIN SELECT * FROM villes WHERE localite='PARIS';
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1071.94 rows=995 width=27)
   Filter: (localite = 'PARIS'::text)
(2 lignes)

marc=# EXPLAIN SELECT * FROM villes WHERE codepostal LIKE '75%';
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1071.94 rows=1042 width=27)
   Filter: (codepostal ~~ '75%'::text)
(2 lignes)


marc=# EXPLAIN SELECT * FROM villes WHERE localite='PARIS'
marc=# AND codepostal LIKE '75%';
                               QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on villes  (cost=0.00..1209.32 rows=19 width=27)
   Filter: ((codepostal ~~ '75%'::text) AND (localite = 'PARIS'::text))
(2 lignes)
```

D'après les statistiques, villes contient 54955 enregistrements, 995 contenant PARIS (presque 2%), 1042 commençant par 75 (presque 2%).

Il y a donc 2% d'enregistrements vérifiant chaque critère (c'est normal, ils sont presque équivalents). PostgreSQL, n'ayant aucune autre information, part de l'hypothèse que les colonnes ne sont pas liées, et qu'il y a donc 2% de 2% (soit environ 0,04%) des enregistrements qui vérifient les deux.

Si on fait le calcul exact, on a donc :

```
(995/54955)*(1042/54955)*54955
```

soit 18,8 enregistrements (arrondi à 19) qui vérifient le critère. Ce qui est évidemment faux.

- Exécutez cette requête sans la dernière clause `AND codepostal LIKE '75%'`. Que constatez-vous ?

```
explain (analyze,buffers) select * from colis where id_ville in (
    select id_ville from villes where localite ='PARIS'
);
```


DALIBO

```
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Hash Semi Join  (cost=1083.86..183312.59 rows=173060 width=16)
                 (actual time=48.975..4362.348 rows=170802 loops=1)
   Hash Cond: (colis.id_ville = villes.id_ville)
   Buffers: shared hit=7 read=54435
   I/O Timings: read=1219.212
   ->  Seq Scan on colis  (cost=0.00..154053.55 rows=9999955 width=16)
                          (actual time=6.178..2228.259 rows=9999911 loops=1)
         Buffers: shared hit=2 read=54052
         I/O Timings: read=1199.307
   ->  Hash  (cost=1071.94..1071.94 rows=954 width=
             (actual time=42.676..42.676 rows=940 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 37kB
         Buffers: shared hit=2 read=383
         I/O Timings: read=19.905
         ->  Seq Scan on villes  (cost=0.00..1071.94 rows=954 width=
               (actual time=35.900..41.957 rows=940 loops=1)
               Filter: (localite = 'PARIS'::text)
               Rows Removed by Filter: 54015
               Buffers: shared hit=2 read=383
               I/O Timings: read=19.905
 Total runtime: 4375.105 ms
(17 lignes)
```

Cette fois-ci le plan est bon, et les estimations aussi.

- Quelle solution pourrait-on adopter, si on doit réellement spécifier ces deux conditions ?

On pourrait indexer sur une fonction des deux. C'est maladroit, mais malheureusement la seule solution sûre :

```
CREATE FUNCTION test_ville (ville text,codepostal text) RETURNS text
IMMUTABLE LANGUAGE SQL as $$
SELECT ville || '-' || codepostal
$$ ;

CREATE INDEX idx_test_ville ON villes (test_ville(localite , codepostal));

ANALYZE villes;

EXPLAIN (analyze,buffers) SELECT * FROM colis WHERE id_ville IN (
    SELECT id_ville
    FROM villes
    WHERE test_ville(localite,codepostal) LIKE 'PARIS-75%'
);
```

```
                                  QUERY PLAN
--------------------------------------------------------------------------------
 Hash Semi Join  (cost=1360.59..183924.46 rows=203146 width=16)
                 (actual time=46.127..3530.348 rows=170802 loops=1)
   Hash Cond: (colis.id_ville = villes.id_ville)
   Buffers: shared hit=454 read=53989
   ->  Seq Scan on colis  (cost=0.00..154054.11 rows=9999911 width=16)
                          (actual time=0.025..1297.520 rows=9999911 loops=1)
         Buffers: shared hit=66 read=53989
   ->  Hash  (cost=1346.71..1346.71 rows=1110 width=8)
             (actual time=46.024..46.024 rows=940 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 37kB
         Buffers: shared hit=385
         ->  Seq Scan on villes  (cost=0.00..1346.71 rows=1110 width=8)
                                 (actual time=37.257..45.610 rows=940 loops=1)
               Filter: (((localite || '-'::text) || codepostal) ~~
                         'PARIS-75%'::text)
               Rows Removed by Filter: 54015
               Buffers: shared hit=385
 Total runtime: 3543.838 ms
```

On constate qu'avec cette méthode il n'y a plus d'erreur d'estimation. Elle est bien sûr très pénible à utiliser, et ne doit donc être réservée qu'aux quelques rares requêtes ayant été identifiées comme ayant un comportement pathologique.

On peut aussi créer une colonne supplémentaire maintenue par un trigger, plutôt qu'un index : cela sera moins coûteux à maintenir, et permettra d'avoir la même statistique.

6.13.3 CONCLUSION

Que peut-on conclure de cet exercice ?

- que la ré-écriture est souvent la meilleure des solutions : interrogez-vous toujours sur la façon dont vous écrivez vos requêtes, plutôt que de mettre en doute PostgreSQL **a priori** ;
- que la ré-écriture de requête est souvent complexe - néanmoins, surveillez un certain nombre de choses :
  - casts implicites suspects ;
  - jointures externes inutiles ;
  - sous-requêtes imbriquées ;
  - jointures inutiles (données constantes)