Formation SQL2

SQL Avancé

17.12

Dalibo SCOP

https://dalibo.com/formations

SQL Avancé

Formation SQL2

TITRE : SQL Avancé
SOUS-TITRE : Formation SQL2

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-06-7

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, Christrophe 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

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.

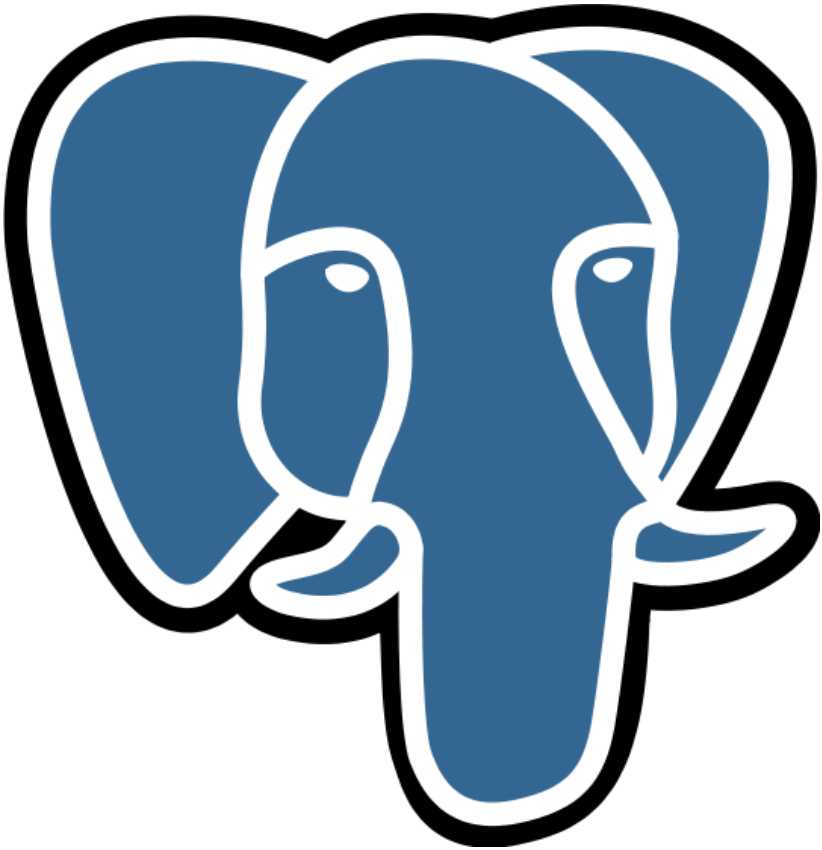
À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

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 POSTGRESQL : OPTIMISATIONS SQL



2.0.1 INTRODUCTION

- L'optimisation doit porter sur les différents composants
  - Le serveur qui héberge le SGBDR : le matériel, la distribution et le kernel
  - Le moteur de la base de données : postgresql.conf
  - La base de données : l'organisation des fichiers de PostgreSQL
  - L'application en elle-même : le schéma, les requêtes et tout ce qui s'y rapporte

- Ce module se focalise sur ce dernier point

Les bases de données sont des systèmes très complexes. Afin d'en tirer toutes les performances possibles, l'optimisation doit porter sur un très grand nombre de composants différents : le serveur qui héberge la base de données, les processus faisant fonctionner la base de données, les fichiers et disques servant à son stockage, mais aussi, et surtout, l'application elle-même. C'est sur ce dernier point que les gains sont habituellement les plus importants.

Ce module se focalise sur ce dernier point. Il n'aborde pas les plans d'exécution à proprement parler, ceux-ci étant traités par un autre module de formation.

2.1 AXES D'OPTIMISATION

- Il est illusoire d'essayer d'optimiser une application sans déterminer au préalable les sources de ralentissement
- Loi de Pareto : « Le pourcentage de la population dont la richesse est supérieure à une valeur x est proportionnel à A/x^α » (Vilfredo Pareto, économiste du XIXe siècle)
- Principe de Pareto (dérivé) : 80% des effets sont produits par 20% des causes.
- L'optimisation doit donc être ciblée :
  - il s'agit de trouver ces « 20% » de causes.

Le principe de Pareto et la loi de Pareto sont des constats empiriques. On peut définir mathématiquement une distribution vérifiant la loi de Pareto[1] . De nombreux phénomènes suivent cette distribution, comme par exemple les files d'attente (trafic internet par exemple).

2.1.1 RECHERCHE DES AXES D'OPTIMISATION

- Utilisation de profiler
  - PostgreSQL : `pgBadger`, `PoWA`, `pg_stat_statements`, `pg_stat_plans`
  - Oracle : `tkprof`, `statspack`, `AWR`...
  - SQL Server : `SQL Server Profiler`

[1] http://fr.wikipedia.org/wiki/Loi_de_Pareto_(probabilités)

Ces outils permettent rapidement d'identifier les requêtes les plus consommatrices sur un serveur. Les outils pour PostgreSQL ont le fonctionnement suivant :

- `pgBadger` est un analyseur de log. On trace donc dans les journaux applicatifs de PostgreSQL toutes les requêtes, leur durée. L'outil les analyse et retourne les requêtes les plus fréquemment exécutées, les plus gourmandes unitairement, les plus gourmandes en temps cumulé (somme des temps unitaires) ;
- `pg_stat_statements` est une vue de PostgreSQL qui trace pour chaque ordre exécuté sur l'instance son nombre d'exécution, sa durée cumulée, et un certain nombre d'autres statistiques très utiles.
- `pg_stat_plans` est une évolution de pg_stat_statements stockant en plus le plan de ces requêtes. En effet, entre le moment de l'exécution de la requête et celui de la consultation de son plan par l'utilisateur souhaitant travailler à son optimisation, le plan peut avoir changé. Elle n'est par contre pas fournie avec PostgreSQL et doit donc être installée séparément.
- `PoWA` est similaire à `Oracle AWR`. Il s'appuie sur `pg_stat_statements` pour permettre d'historiser l'activité du serveur. Une interface web permet ensuite de visualiser l'activité ainsi historisée et repérer les requêtes problématiques avec les fonctionnalités de drill-down de l'interface.

2.2 PROBLÉMATIQUES LIÉES AU SCHÉMA

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

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

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

2.2.1 QUELQUES RAPPELS SUR LE MODÈLE RELATIONNEL

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

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

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

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

2.2.2 FORMES NORMALES

Il existe une définition mathématique précise de chacune des 7 formes normales.

- La troisième forme normale peut toujours être atteinte
- La forme suivante (forme normale de Boyce-Codd, ou FNBC) ne peut pas toujours être atteinte
- La cible est donc habituellement la `3FN`
- Définition simple par **Chris Date** :
  - « *Chaque attribut dépend de la clé, de TOUTE la clé, et QUE de la clé* »
  - « *The key, the whole key, nothing but the key* »

Une relation (table) est en troisième forme normale si tous les attributs (colonnes) dépendent de la clé (primaire), de toute la clé (pas d'un sous-ensemble de ses colonnes), et de rien d'autre que de la clé (une colonne supplémentaire).

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

Voir l'article wikipedia[2] présentant l'ensemble des formes normales.

2.2.3 ATOMICITÉ

- Un attribut (colonne) doit être atomique :
  - Modifier l'attribut sans en toucher un autre
  - Donnée correcte : `boolean abs`, `boolean volant_a_gauche`, `enum` couleur, etc. **Difficile**
  - Recherche efficace : accédé en entier dans une clause `WHERE`
  - Non respect = violation de la première forme normale

On rencontre parfois des tables de cette forme :

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

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

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

ce qui sera évidemment très inefficace.

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

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

2.2.4 ATOMICITÉ - MAUVAIS EXEMPLE

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

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

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

Dans ce cas, rien n'empêche d'ajouter une ligne avec des caractéritiques similaires mais définie autrement : * ABS / antiblocage des roues * Le moteur retournera le véhicule "AD-057-GD si on veut rechercher un véhicule de couleur "bleu" `LIKE '%peinture verte%'`

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

2.2.5 ATOMICITÉ - PROPOSITION

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

Ce modèle facilite les recherches et assure la cohérence.

2.2.6 NULL

`NULL` signifie habituellement :

- Valeur non renseignée
- Valeur inconnue

Dans tous les cas, c'est une absence d'information. Ou du moins la seule information est qu'on ne sait pas.

Une table remplie de NULLs est habituellement signe d'un problème de modélisation.

17.12

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

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

Il faut avoir à l'esprit cette citation de Chris Date :

> « La valeur `NULL` telle qu'elle est implémentée dans SQL peut poser plus de problèmes qu'elle n'en résout. Son comportement est parfois étrange et est source de nombreuses erreurs et de confusions. »

2.2.7 COLONNE DE TYPE VARIABLE

Plus rarement, on rencontre aussi :

- Une colonne de type `varchar`
- qui contient :
  - quelquefois un entier
  - quelquefois une date
  - un `NULL`
  - une chaîne autre
  - etc.
- À éviter comme la peste !

On rencontre parfois ce genre de choses :

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

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

On a un gros problème de modélisation : la colonne a un type de contenu qui dépend de l'information qu'elle contient. On aurait dû avoir une colonne supplémentaire (un booléen `tournee_ok` par exemple). On va aussi avoir un problème de performance en joignant ce varchar à la clé numérique de la table `tournée`. Le moteur n'aura que deux choix : convertir le varchar en numérique, avec une exception à la clé en essayant de convertir « ANNULÉE », ou bien (ce qu'il fera) convertir le numérique de la table `tournee` en chaîne. Cette dernière méthode rendra l'accès à l'id de tournée par index impossible. D'où un parcours complet (*seq scan*) de la table `tournée` à chaque accès.

2.2.8 STOCKAGE ENTITÉ-CLÉ-VALEUR

- Entité-Attribut-Valeur (ou Entité-Clé-Valeur)
- Quel but ?
  - flexibilité du modèle de données
  - adapter sans délai ni surcoût le modèle de données
- Conséquences :
  - création d'une table : `identifiant, nom_attribut, valeur`
  - requêtes abominables et coûteuses
- Solutions :
  - revenir sur la conception du modèle de données
  - utiliser un type de données plus adapté (`hstore`)

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

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

Ainsi, pour reprendre l'exemple des informations de contacts pour un individu, une table `personnes` permet de stocker un identifiant de personne. Une table `personne_attributs` permet d'associer des données à un identifiant de personne. Le type de données de la colonne est souvent prévu largement pour faire tenir tout

type d'informations, mais sous forme textuelle. Les données ne peuvent donc pas être validées.

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

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

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

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

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

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

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

Lorsque de telles solutions sont 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` qui permet de stocker des données sous la forme `clé->valeur`. Ce type de données peut être indexé pour garantir de bons temps de réponses des requêtes qui nécessitent des recherches sur certaines clés ou certaines valeurs.

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

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

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

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

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

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

2.2.9 STOCKAGE ENTITÉ-CLÉ-VALEUR - EXEMPLE

id_pers	nom_attr	val_attr
1	nom	Prunelle
1	prenom	Léon

id_pers	nom_attr	val_attr
1	telephone	0123456789
1	fonction	dba

Comment lister tous les dba?

2.2.10 STOCKAGE ENTITÉ-CLÉ-VALEUR - REQUÊTE ASSOCIÉE...

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

2.2.11 NOMBREUSES COLONNES

Tables à plusieurs dizaines, voire centaines de colonnes :

- Les entités sont certainement trop grosses dans la modélisation
- Il y a probablement dépendance entre certaines colonnes (« *Only the key* »)
- On accède à beaucoup d'attributs inutiles (tout est stocké au même endroit)

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

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

2.2.12 ABSENCE DE CONTRAINTES

- Parfois (souvent ?) ignorées pour diverses raisons :
  - faux gains de performance
  - flexibilité du modèle de données
  - compatibilité avec d'autres SGBD
  - commodité de développement
- Les contraintes sont utiles à l'optimiseur :
  - déterminent l'unicité des valeurs
  - éradiquent des lectures de tables inutiles sur des `LEFT JOIN`
  - utilisent les contraintes `CHECK` pour exclure une partition

Les contraintes d'intégrité et notamment les clés étrangères sont parfois absentes des modèles de données. Les problématiques de performance et de flexibilité sont souvent mises en avant alors que les contraintes sont justement une aide pour l'optimisation de requêtes par le planificateur.

De plus, l'absence de contraintes va également entraîner des problèmes d'intégrité des données. Il est par exemple très compliqué de se prémunir efficacement contre une race condition[3] en l'absence de clé étrangère. Lorsque ces problèmes d'intégrité seront détectés, il s'en suivra également la création de procédures de vérification de cohérence des données qui vont aussi alourdir les développements, entraînant ainsi un travail supplémentaire pour trouver et corriger les incohérences. Ce qui a été gagné d'un côté est perdu de l'autre, mais sous une forme différente.

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

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

---

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

2.2.13 COMPLEXITÉ

Pour les performances, on envisage souvent de distribuer la base sur plusieurs nœuds.

- La complexité augmente (au niveau du code applicatif et/ou des procédures d'exploitation)
  - le risque d'erreur avec lui (programmation, fausse manipulation)
- Le retour à un état stable après un incident est complexe

Il est toujours tentant d'augmenter la quantité de ressources matérielles pour résoudre un problème de performance. Il ne faut surtout pas négliger tous les coûts de cette solution : non seulement l'achat de matériel, mais aussi les coûts humains : procédures d'exploitation, de maintenance, complexité accrue de développement, etc.

Performance et robustesse peuvent être des objectifs contradictoires.

2.3 SQL - REQUÊTES

- Le SQL est un langage déclaratif :
  - on décrit le résultat et pas la façon de l'obtenir
  - comme Prolog
  - c'est le travail de la base de déterminer le traitement à effectuer
- 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. Il s'agit de la norme ISO 9075. 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 objet pour le modèle relationnel-objet. La dernière version stable de la norme est SQL:2011.

2.3.1 OPÉRATEURS RELATIONNELS

Les opérateurs purement relationnels sont les suivants :

- Projection
  - Clause SELECT (choix des colonnes)
- Sélection

  - Clause `WHERE` (choix des enregistrements)
- Jointure
  - Clause `FROM/JOIN` (choix des tables)
- Bref, tout ce qui détermine sur quelles données on travaille

Tous ces opérateurs sont optimisables : il y a 40 ans de théorie mathématique développée afin de permettre l'optimisation de ces traitements. L'optimiseur fera un excellent travail sur ces opérations, et les organisera de façon efficace.

Par exemple : `a JOIN b JOIN c WHERE c.col=constante` sera très probablement réordonné en `c JOIN b JOIN a WHERE c.col=constante` ou `c JOIN a JOIN b WHERE c.col=constante`. Le moteur se débrouillera aussi pour choisir le meilleur algorithme de jointure pour chacune, suivant les volumétries ramenées.

2.3.2 OPÉRATEURS NON-RELATIONNELS

- Les autres opérateurs sont non-relationnels :
  - `ORDER BY`
  - `GROUP BY/DISTINCT`
  - `HAVING`
  - Sous-requête, vue
  - Fonction (classique, d'agrégat, analytique)
  - Jointure externe

Ceux-ci sont plus difficilement optimisables : ils introduisent par exemple des contraintes d'ordre dans l'exécution :

```
SELECT * FROM table1
WHERE montant > (
 SELECT avg(montant) FROM table1 WHERE departement='44'
);
```

On doit exécuter la sous-requête avant la requête.

Les jointures externes sont relationnelles, mais posent tout de même des problèmes et doivent être traitées prudemment.

```
SELECT * FROM t1 LEFT JOIN t2 on (t1.t2id=t2.id) JOIN t3 on (t1.t3id=t3.id)
;
```

Il faut faire les jointures dans l'ordre indiqué : joindre `t1` à `t3` puis le résultat à `t2` pourrait ne pas amener le même résultat (un `LEFT JOIN` peut produire des `NULL`). Il est donc

préférable de toujours mettre les jointures externes en fin de requête, sauf besoin précis : on laisse bien plus de liberté à l'optimiseur.

Le mot clé `DISTINCT` ne doit être utilisé qu'en dernière extrémité. On le rencontre très fréquemment dans des requêtes mal écrites qui produisent des doublons, afin de maquiller le résultat. C'est bien sûr extrêmement simple de produire des doublons pour ensuite dédoublonner (au moyen d'un tri de l'ensemble du résultat, bien sûr).

2.3.3 DONNÉES UTILES

Le volume de données récupéré a un impact sur les performances.

- N'accéder qu'aux tables nécessaires
- N'accéder qu'aux colonnes nécessaires
- Plus le volume de données à traiter est élevé, plus les opérations seront lentes :
  - Tris et Jointures
  - Éventuellement stockage temporaire sur disque pour certains algorithmes

Éviter donc les `SELECT *` : c'est une bonne pratique de toute façon, car la requête peut changer de résultat suite à un changement de schéma, ce qui risque d'entraîner des conséquences sur le reste du code.

Ne récupérer que les colonnes utilisées. Certains moteurs suppriment d'eux-même les colonnes qui ne sont pas retournées à l'appelant, comme par exemple dans le cas de :

```
SELECT col1, col2 FROM (SELECT * FROM t1 JOIN t2 USING (t2id) ) ;
```

PostgreSQL ne le fait pas pour le moment.

Éviter les jointures sur des tables inutiles : il n'y a que peu de cas où l'optimiseur peut supprimer de lui-même l'accès à une table inutile.

PostgreSQL le fait dans le cas d'un `LEFT JOIN` sur une table inutilisée dans le `SELECT`, au travers d'une clé étrangère, car on peut garantir que cette table est effectivement inutile.

2.3.4 LIMITER LE NOMBRE DE REQUÊTES

SQL : langage ensembliste et déclaratif

- Ne pas faire de traitement unitaire par enregistrement
- Utiliser les jointures, ne pas accéder à chaque table une-par-une
- Une seule requête, parcours de curseur

- Fréquent avec les ORM

Les bases de données relationnelles sont conçues pour manipuler des relations, pas des enregistrements unitaires.

Le langage SQL (et même les autres langages relationnels qui ont existé comme QUEL, SEQUEL) est conçu pour permettre la manipulation d'un gros volume de données, et la mise en correspondance (jointure) d'informations. Une base de données relationnelle n'est pas une simple couche de persistance.

Le fait de récupérer en une seule opération l'ensemble des informations pertinentes est aussi, indépendamment du langage, un gain de performance énorme, car il permet de s'affranchir en grande partie des latences de communication entre la base et l'application.

Préparons un jeu de test :

```
psql> CREATE TABLE test (a int, b varchar);
psql> INSERT INTO test SELECT i,i FROM generate_series (1,1000000) g(i);
psql> ALTER TABLE test ADD PRIMARY KEY (a);
```

Récupérons 10 000 enregistrements un par un (par un script perl par exemple, avec une requête préparée pour être dans le cas le plus efficace) :

```
#!/usr/bin/perl -w
print "PREPARE ps (int) AS SELECT * FROM test WHERE a=\$1;\n";
for (my $i=0;$i<=10000;$i++)
{
        print "EXECUTE ps($i);\n";
}
```

Exécutons ce code :

```
time perl demo.pl  | psql > /dev/null

real    0m1.025s
user    0m0.213s
sys     0m0.057s
```

Voici maintenant la même chose, en un seul ordre SQL :

```
time psql -c "select * from test where a >=0 and a <= 10000" > /dev/null

real    0m0.052s
user    0m0.030s
sys     0m0.003s
```

La plupart des ORM fournissent un moyen de traverser des liens entre objets. Par exemple, si une commande est liée à plusieurs articles, un ORM permettra d'écrire un code similaire à celui-ci (exemple en Java avec Hibernate) :

```
List commandes = sess.createCriteria(Commande.class);

for(Commande cmd : commandes)
{
  // Un traitement utilisant les produits
  // Génère une requête par commande !!
  System.out.println(cmd.getProduits());
}
```

Tel quel, ce code générera *N+1* requête, N étant le nombre de commandes. En effet, pour chaque accès à l'attribut "produits", l'ORM générera une requête pour récupérer les produits correspondants à la commande.

Le SQL généré sera alors similaire à celui-ci :

```
SELECT * FROM commande;
SELECT * from produits where commande_id = 1;
SELECT * from produits where commande_id = 2;
SELECT * from produits where commande_id = 3;
SELECT * from produits where commande_id = 4;
SELECT * from produits where commande_id = 5;
SELECT * from produits where commande_id = 6;
...
```

La plupart des ORM proposent des options pour personnaliser la stratégie d'accès aux collections. Il est extrêmement important de connaître celles-ci afin de permettre à l'ORM de générer des requêtes optimales.

Par exemple, dans le cas précédent, nous savons que tout les produits de toutes les commandes seront utilisés. Nous pouvons donc informer l'ORM de ce fait :

```
List commandes = sess.createCriteria(Commande.class)
  .setFetchMode('produits', FetchMode.EAGER);

for(Commande cmd : commandes)
{
  // Un traitement utilisant les produits
  System.out.println(cmd.getProduits());
}
```

Ceci générera une seule et unique requête du type :

```
SELECT * FROM commandes
LEFT JOIN produits ON commandes.id = produits.commande_id;
```

2.3.5 ÉVITER LES VUES NON-RELATIONNELLES

Une vue est simplement une requête pré-déclarée en base.

- C'est l'équivalent relationnel d'une fonction
- Quand elle est utilisée dans une autre requête, elle est initialement traitée comme une sous-requête
- Attention aux vues avec `DISTINCT`, `GROUP BY` etc.
  - Impossible de l'*inliner*
  - Barrière d'optimisation
  - Donc mauvaises performances
- Les vues sont dangereuses en termes de performance
  - masquent la complexité

Les vues sont très pratiques en SQL et en théorie permettent de séparer le modèle physique (les tables) de ce que voient les développeurs, et donc de faire évoluer le modèle physique sans impact pour le développement. En pratique, elles vont souvent être source de ralentissement : elles masquent la complexité, et peuvent rapidement conduire à l'écriture implicite de requêtes très complexes, mettant en jeu des dizaines de tables (voire des dizaines de fois les **MÊMES** tables).

Il faut donc se méfier des vues. En particulier, des vues contenant des opérations non-relationnelles, qui peuvent empêcher de nombreuses optimisations. En voici un exemple simple. La vue a été remplacée par une sous-requête équivalente :

```
EXPLAIN SELECT id,valeur FROM
 (SELECT DISTINCT ON (id) id,valeur FROM test ) AS tmp
 WHERE valeur='b' ;
```

```
QUERY PLAN
Subquery Scan on tmp (cost=76.43..85.08 rows=1 width=36)
Filter: (tmp.valeur = 'b'::text)
-> Unique (cost=76.43..82.58 rows=200 width=36)
-> Sort (cost=76.43..79.50 rows=1230 width=36)
Sort Key: test.id
-> Seq Scan on test (cost=0.00..13.30 rows=1230 width=36)
```

On constate que la condition de filtrage sur `b` n'est appliquée qu'à la fin. C'est normal, à cause du `DISTINCT ON`, l'optimiseur ne peut pas savoir si l'enregistrement qui sera retenu dans la sous-requête vérifiera `valeur='b'` ou pas, et doit donc attendre l'étape suivante

pour filtrer. Le coût en performances, même avec un volume de données raisonnable, peut être astronomique.

2.3.6 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
  - 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
  - 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,
 DECODE(index_clerecherche, 'Artiste.nom_prenom',
  SUBSTR(chaine, 0, INSTR(chaine, '_') - 1),
                                    '')        AS NOM,
 DECODE(index_clerecherche, 'Artiste.nom_prenom',
  SUBSTR(chaine, INSTR(chaine, '_') + 1),
                                    '')        AS PRENOM
FROM   ((SELECT index_idenreg AS IDARTICLE,
                C.cde_art     AS CHAINE,
                index_clerecherche
         FROM   cstd_mots M
                join cstd_index I
                  ON I.mots_id = M.mots_id
                     AND index_clerecherche =
                       'Article.codeArticle'
                join da_article C
                  ON id = index_idenreg
         WHERE  mots_mot = 'foire'
         INTERSECT
         SELECT index_idenreg AS IDARTICLE,
                C.cde_art     AS CHAINE,
                index_clerecherche
         FROM   cstd_mots M
                join cstd_index I
                  ON I.mots_id = M.mots_id
                     AND index_clerecherche =
                      'Article.codeArticle'
                join da_article C
                  ON id = index_idenreg
         WHERE  mots_mot LIKE 'vins%'
            AND 1 = 1)
        UNION ALL
        (SELECT index_idenreg AS IDARTICLE,
                C.cde_art_bal AS CHAINE,
                index_clerecherche
         FROM   cstd_mots M
                join cstd_index I
                  ON I.mots_id = M.mots_id
                     AND index_clerecherche =
                      'Article.codeArticleBalance'
                join da_article C
                  ON id = index_idenreg
         WHERE  mots_mot = 'foire'
         INTERSECT
         SELECT index_idenreg AS IDARTICLE,
```

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

```
                        WHERE  mots_mot = 'foire'
                        INTERSECT
                        SELECT idda_article AS IDARTICLE,
                               C.gtin       AS CHAINE,
                               index_clerecherche
                        FROM   cstd_mots M
                               join cstd_index I
                                 ON I.mots_id = M.mots_id
                                    AND index_clerecherche = 'Gtin.gtin'
                               join da_gtin C
                                 ON id = index_idenreg
                        WHERE  mots_mot LIKE 'vins%'
                           AND 1 = 1)
                      UNION ALL
                      (SELECT idda_article AS IDARTICLE,
                               C.ref_frn    AS CHAINE,
                               index_clerecherche
                        FROM   cstd_mots M
                               join cstd_index I
                                 ON I.mots_id = M.mots_id
                                    AND index_clerecherche =
                                   'ArbreLogistique.referenceFournisseur'
                               join da_arblogi C
                                 ON id = index_idenreg
                        WHERE  mots_mot = 'foire'
                        INTERSECT
                        SELECT idda_article AS IDARTICLE,
                               C.ref_frn    AS CHAINE,
                               index_clerecherche
                        FROM   cstd_mots M
                               join cstd_index I
                                 ON I.mots_id = M.mots_id
                                    AND index_clerecherche =
                                   'ArbreLogistique.referenceFournisseur'
                               join da_arblogi C
                                 ON id = index_idenreg
                        WHERE  mots_mot LIKE 'vins%'
                           AND 1 = 1))) INNERSEARCH
              ON INNERSEARCH.idarticle = RNA.id) SEARCHMC
  ON SEARCHMC.idarticle = Article.id
     AND 1 = 1
left join da_artcmpl ArticleComplement
       ON Article.id = ArticleComplement.idda_article
left join dp_gpdtcul GenreProduitCulturel
       ON ArticleComplement.iddp_gpdtcul = GenreProduitCulturel.id
left join df_collect Collection
```

```
        ON ArticleComplement.iddf_collect = Collection.id
left join da_gtin Gtin
        ON Article.id = Gtin.idda_article
           AND Gtin.principal = 1
           AND Gtin.db_suplog = 0
left join df_fabric Fabricant
        ON Article.iddf_fabric = Fabricant.id
left join dp_clsvtel ClassificationVenteLocale
        ON Article.iddp_clsvtel = ClassificationVenteLocale.id
left join dp_marqcom MarqueCommerciale
        ON Article.iddp_marqcom = MarqueCommerciale.id
left join da_compo Composition
        ON Composition.id = Article.idda_compo
left join dp_comptex CompositionTextile
        ON CompositionTextile.id = Article.iddp_comptex
left join da_arttrai AssoArticleInterfaceBalance
        ON AssoArticleInterfaceBalance.idda_article = Article.id
           AND AssoArticleInterfaceBalance.iddp_tinterf = 1
left join da_arttrai AssoArticleInterfaceCaisse
        ON AssoArticleInterfaceCaisse.idda_article = Article.id
           AND AssoArticleInterfaceCaisse.iddp_tinterf = 4
left join dp_raybala RayonBalance
        ON Article.iddp_raybala = RayonBalance.id
left join dp_valdico TypeTare
        ON TypeTare.id = RayonBalance.iddp_typtare
left join df_categor Categorie
        ON Categorie.id = Article.iddf_categor
left join df_grille GrilleDePrix
        ON GrilleDePrix.id = Categorie.iddf_grille
left join dp_agreage FicheAgreage
        ON FicheAgreage.id = Article.iddp_agreage
join dp_codelec Codelec
  ON Article.iddp_codelec = Codelec.id
left join dp_periact PerimetreActivite
        ON PerimetreActivite.id = Codelec.iddp_periact
left join dp_perscod DonneesPersonnalisablesCodelec
        ON Codelec.id = DonneesPersonnalisablesCodelec.iddp_codelec
           AND DonneesPersonnalisablesCodelec.db_suplog = 0
           AND DonneesPersonnalisablesCodelec.iddb_sitecl = 1012124
left join dp_clsart ClassificationArticleInterne
        ON DonneesPersonnalisablesCodelec.iddp_clsart =
          ClassificationArticleInterne.id
left join da_artdeno ArticleDenormalise
        ON Article.id = ArticleDenormalise.idda_article
left join df_clasmnt ClassementFournisseur
        ON ArticleDenormalise.iddf_clasmnt = ClassementFournisseur.id
```

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

```
              ON LienCoordonnable.id = assoALCOOR.idda_lien
       left join da_lienal assoALSTAT
              ON assoALSTAT.idda_arblogi = ArbreLogistique.id
                 AND assoALSTAT.iddp_typlien = 14897
       left join da_lien LienStatistique
              ON LienStatistique.id = assoALSTAT.idda_lien WHERE
SEARCHMC.rn = 1
   AND ( ValorisationAchat.id IS NULL
         OR ValorisationAchat.date_debut_achat = (
                    SELECT MAX(VALMAX.date_debut_achat)
                    FROM   tt_valo VALMAX
                    WHERE  VALMAX.idtt_val_dal = ValorisationAchat.idtt_val_dal
                       AND VALMAX.date_modif_retro IS NULL
                       AND VALMAX.val_ach_pab IS NOT NULL
                       AND VALMAX.date_debut_achat <= '2013-09-27') )
   AND ( Article.id IN (SELECT A.id
                        FROM   da_article A
                               join du_ucutiar AssoUcUtiAr
                                 ON AssoUcUtiAr.idda_article = A.id
                               join du_asucuti AssoUcUti
                                 ON AssoUcUti.id = AssoUcUtiAr.iddu_asucuti
                        WHERE  ( AssoUcUti.iddu_uti IN ( 90000000000022 ) )
                           AND a.iddb_sitecl = 1012124) )
   AND Article.db_suplog = 0
ORDER  BY SEARCHMC.poids ASC
```

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

2.3.7 SOUS-REQUÊTES 1/2

- Si `IN`, limiter le nombre d'enregistrements grâce à `DISTINCT`

```
SELECT * FROM t1
  WHERE val IN (SELECT DISTINCT …)
```

- Éviter une requête liée :

```
SELECT a,b
  FROM t1
  WHERE val IN (SELECT f(b))
```

2.3.8 SOUS-REQUÊTES 2/2

- Certaines sous-requêtes sont l'expression de Semi-join ou Anti-join

```
SELECT * FROM t1 WHERE fk
   [NOT] IN (SELECT pk FROM t2 WHERE xxx)
SELECT * FROM t1 WHERE [NOT] EXISTS
   (SELECT 1 FROM t2 WHERE t2.pk=t1.fk AND xxx)
SELECT t1.* FROM t1 LEFT JOIN t2 ON (t1.fk=t2.pk)
   WHERE t2.id IS [NOT] NULL`
```

- sont strictement équivalentes !
  - L'optimiseur les exécute à l'identique (sauf NOT IN)

Les seules sous-requêtes sans danger sont celles qui retournent un ensemble constant et ne sont exécutés qu'une fois, ou celles qui expriment un Semi-Join (test d'existence) ou Anti-Join (test de non-existence), qui sont presque des jointures : la seule différence est qu'elles ne récupèrent pas l'enregistrement de la table cible.

Attention toutefois à l'utilisation du prédicat NOT IN, ils peuvent générer des plans d'exécution catastrophiques :

```
tpc=# EXPLAIN SELECT *
  FROM commandes
 WHERE numero_commande NOT IN (SELECT numero_commande
                               FROM lignes_commandes);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..1196748507.51 rows=84375 width=77)
   Workers Planned: 1
   ->  Parallel Seq Scan on commandes  (cost=0.00..1196739070.01 rows=49632 width=77)
         Filter: (NOT (SubPlan 1))
         SubPlan 1
           ->  Materialize  (cost=0.00..22423.15 rows=675543 width=8)
                 ->  Seq Scan on lignes_commandes  (cost=0.00..16406.43 rows=675543 width=8)
(7 rows)

Time: 0.460 ms
```

La requête suivante est strictement équivalente et produit un plan d'exécution largement plus intéressant :

```
tpc=# EXPLAIN SELECT *
  FROM commandes
 WHERE NOT EXISTS (SELECT 1
                     FROM lignes_commandes l
                    WHERE l.numero_commande = commandes.numero_commande);
                              QUERY PLAN
```

```
------------------------------------------------------------------------
                                QUERY PLAN
------------------------------------------------------------------------------------------------
 Gather  (cost=28489.72..43299.60 rows=25860 width=77)
   Workers Planned: 1
   ->  Hash Anti Join  (cost=27489.72..39713.60 rows=15212 width=77)
         Hash Cond: (commandes.numero_commande = l.numero_commande)
         ->  Parallel Seq Scan on commandes  (cost=0.00..3403.65 rows=99265 width=77)
         ->  Hash  (cost=16406.43..16406.43 rows=675543 width=8)
               ->  Seq Scan on lignes_commandes l  (cost=0.00..16406.43 rows=675543 width=8)
(7 rows)
```

La raison, c'est que si un `NULL` est présent dans la liste du `NOT IN`, `NOT IN` vaut systématiquement `FALSE`. Nous, nous savons qu'il n'y aura pas de `numero_commandes` à `NULL`.

2.4 ACCÈS AUX DONNÉES

L'accès aux données est coûteux.

- Quelle que soit la base
- Dialogue entre client et serveur
  - Plusieurs aller/retours potentiellement
- Analyse d'un langage complexe
  - SQL PostgreSQL : `gram.y` de 14000 lignes
- Calcul de plan :
  - langage déclaratif => converti en impératif à chaque exécution

Dans les captures réseau ci-dessous, le serveur est sur le port 5932.

`SELECT * FROM test`, 0 enregistrement :

```
10:57:15.087777 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [P.], seq 109:134, ack 226, win 350,
    options [nop,nop,TS val 2270307 ecr 2269578], length 25
10:57:15.088130 IP 127.0.0.1.5932 > 127.0.0.1.39508:
    Flags [P.], seq 226:273, ack 134, win 342,
    options [nop,nop,TS val 2270307 ecr 2270307], length 47
10:57:15.088144 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [.], ack 273, win 350,
    options [nop,nop,TS val 2270307 ecr 2270307], length 0
```

`SELECT * FROM test`, 1000 enregistrements :

```
10:58:08.542660 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [P.], seq 188:213, ack 298, win 350,
    options [nop,nop,TS val 2286344 ecr 2285513], length 25
10:58:08.543281 IP 127.0.0.1.5932 > 127.0.0.1.39508:
    Flags [P.], seq 298:8490, ack 213, win 342,
    options [nop,nop,TS val 2286344 ecr 2286344], length 8192
10:58:08.543299 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [.], ack 8490, win 1002,
    options [nop,nop,TS val 2286344 ecr 2286344], length 0
10:58:08.543673 IP 127.0.0.1.5932 > 127.0.0.1.39508:
    Flags [P.], seq 8490:14241, ack 213, win 342,
    options [nop,nop,TS val 2286344 ecr 2286344], length 5751
10:58:08.543682 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [.], ack 14241, win 1012,
    options [nop,nop,TS val 2286344 ecr 2286344], length 0
```

Un client JDBC va habituellement utiliser un aller/retour de plus, en raison des requêtes préparées : un dialogue pour envoyer la requête et la préparer, et un autre pour envoyer les paramètres. Le problème est la latence du réseau, habituellement : de 50 à 300µs. Cela limite à 3 000 à 20 000 le nombre d'opérations maximum par seconde par socket. On peut bien sûr paralléliser sur plusieurs sessions, mais cela complique le traitement.

En ce qui concerne le parser : comme indiqué dans ce message[4] : `gram.o`, le parser fait 1Mo une fois compilé !

2.4.1 MAINTIEN DES CONNEXIONS

Se connecter coûte cher :

- Vérification authentification, permissions
- Création de processus, de contexte d'exécution
- Éventuellement négotiation SSL
- Acquisition de verrous

=> Maintenir les connexions coté applicatif ou utiliser un pooler.

PostgreSQL fournit un outil de benchmark synthétique. Voici les résultats :

Option `-C` : se connecter à chaque requête :

[4] http://www.postgresql.org/message-id/CA+TgmoaaYvJ7yDKJHrWN1BVk_7fcV16rvc93udSo59gfxG_t7A@mail.gmail.com

17.12

```
$ pgbench pgbench -T 20 -c 10 -j5 -S -C
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 5
duration: 20 s
number of transactions actually processed: 16972
latency average = 11.787 ms
tps = 848.383850 (including connections establishing)
tps = 1531.057609 (excluding connections establishing)
```

Sans se reconnecter à chaque requête :

```
$ pgbench pgbench -T 20 -c 10 -j5 -S
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 2
query mode: simple
number of clients: 10
number of threads: 5
duration: 20 s
number of transactions actually processed: 773963
latency average = 0.258 ms
tps = 38687.524110 (including connections establishing)
tps = 38703.239556 (excluding connections establishing)
```

On passe de 900 à 20 000 transactions par seconde.

2.4.2 PENSER RELATIONNEL

Les spécifications sont souvent procédurales, voire objet !

- Il faut prendre du recul, et réfléchir de façon ensembliste
  - On travaille sur des ensembles de données
  - On peut faire encore mieux avec les CTE (SQL:1999)

Si les spécifications disent (version simplifiée bien sûr) :

- Vérifier la présence du client

- S'il est présent, mettre à jour son adresse
- Sinon, créer le client avec la bonne adresse

Peut être écrit (pseudo-code client) :

```
SELECT count(*) from clients where client_name = 'xxx' INTO compte
IF compte > 0
  UPDATE clients set adresse='yyy' WHERE client_name='xxx'
ELSE
  INSERT client SET client_name='xxx', adresse='yyy'
END IF
```

3 requêtes, systématiquement 2 appels à la base.

On peut très bien l'écrire comme suit :

```
UPDATE clients set adresse='yyy' WHERE client_name='xxx'
IF NOT FOUND
  INSERT client SET client_name='xxx', adresse='yyy'
END IF
```

Dans ce cas, on n'aura 2 requêtes que dans le plus mauvais cas. Bien sûr, cet exemple est simpliste.

On peut aussi, grâce aux CTE, réaliser tout cela en un seul ordre SQL, qui ressemblerait à :

```
WITH
  enregistrements_a_traiter AS (
    SELECT * FROM (VALUES ('toto' , 'adresse1' ),('tata','adresse2'))
    AS val(nom_client,adresse)
  ),
  mise_a_jour AS (
    UPDATE client SET adresse=enregistrements_a_traiter.adresse
    FROM enregistrements_a_traiter
    WHERE enregistrements_a_traiter.nom_client=client.nom_client
    RETURNING client.nom_client
  )
INSERT INTO client (nom_client,adresse)
  SELECT nom_client,adresse from enregistrements_a_traiter
  WHERE NOT EXISTS (
    SELECT 1 FROM mise_a_jour
    WHERE mise_a_jour.nom_client=enregistrements_a_traiter.nom_client
  );
```

Plus d'information sur la fusion des enregistrements dans PostgreSQL avec des CTE[5] .

[5] http://vibhorkumar.wordpress.com/2011/10/26/upsertmerge-using-writable-cte-in-postgresql-9-1/

2.4.3 PAS DE DDL APPLICATIF

- Le schéma représente la modélisation des données
  - Une application n'a pas à y toucher lors de son fonctionnement normal
  - Parfois : tables temporaires locales à des sessions
  - Toujours voir si une autre solution est possible
- SQL manipule les données en flux continu :
  - chaque étape d'un plan d'exécution n'attend pas la fin de la précédente
  - Passer par une table temporaire est probablement une perte de temps

Si on reprend l'exemple précédent, il aurait pu être écrit :

```
=> CREATE TEMP TABLE temp_a_inserer  (nom_client text, adresse text);
=> INSERT INTO temp_a_inserer SELECT * FROM (VALUES ('toto' , 'adresse1' ),
              ('tata','adresse2')) AS tmp;
=> UPDATE client SET adresse=temp_a_inserer.adresse
   FROM temp_a_inserer
   WHERE temp_a_inserer.nom_client=client.nom_client;
=> INSERT INTO client (nom_client,adresse)
      SELECT nom_client,adresse from temp_a_inserer
      WHERE NOT EXISTS (
          SELECT 1 FROM client
          WHERE client.nom_client=temp_a_inserer.nom_client);
=> DROP TABLE temp_a_inserer;
```

1000 exécutions de cette méthode prennent 5 secondes, alors que la solution précédente ne dure que 500ms.

2.4.4 OPTIMISER CHAQUE ACCÈS

Un ordre SQL peut effectuer de nombreuses choses :

- Les moteurs SQL sont très efficaces, et évoluent en permanence
- Ils ont de nombreuses méthodes de tri, de jointure, qu'ils choisissent en fonction du contexte
- Si vous utilisez le langage SQL, votre requête profitera des futures évolutions
- Si vous codez tout dans votre programme, vous devrez le maintenir et l'améliorer
- Faites un maximum du côté SQL : agrégats, fonctions analytiques, tris, numérotations, `CASE`, etc.
- Commentez votre code avec `--` et `/* */`

L'avantage du code SQL est, encore une fois, qu'il est déclaratif. Il aura donc de nombreux avantages sur un code procédural.

L'exécution évoluera pour prendre en compte les variations de volumétrie des différentes tables.
Les optimiseurs sont la partie la plus importante d'un moteur SQL. Ils progressent en permanence. Chaque nouvelle version va donc potentiellement améliorer vos performances.

Si vous écrivez du procédural avec des appels unitaires à la base dans des boucles, le moteur ne pourra rien optimiser

Si vous faites vos tris ou regroupements côté client, vous êtes limités aux algorithmes fournis par vos langages, voire à ceux que vous aurez écrit manuellement. Une base de données bascule automatiquement entre une dizaine d'algorithmes différents suivant le volume, le type de données à trier, ce pour quoi le tri est ensuite utilisé, etc., voire évite de trier en utilisant des tables de hachage ou des index disponibles.

2.4.5 NE FAIRE QUE LE NÉCESSAIRE

Encore une fois, prendre de la distance vis-à-vis des spécifications fonctionnelles :

- Si le client existe, le mettre à jour :
  - Le mettre à jour, et regarder combien d'enregistrements ont été mis à jour
- Si le client existe :
  - Surtout pas de `COUNT(*)`, éventuellement un test de l'existence d'UN enregistrement
- Gérer les exceptions plutôt que de vérifier préalablement que les conditions sont remplies (si l'exception est rare)

Toujours coder les accès aux données pour que la base fasse le maximum de traitement, mais uniquement les traitements nécessaires : l'accès aux données est coûteux, il faut l'optimiser. Et le gros des pièges peut être évité avec les quelques règles d'« hygiène » simples qui viennent d'être énoncées.

2.5 INDEX

- La bonne utilisation d'un index est un sujet à part entière :
  - sujet effleuré ici
  - excellent livre par **Markus Winand** : *SQL Performance Explained*
- Compromis insertion/sélection
- Objet technique :

  - ni dans la théorie relationnelle
  - ni dans la norme SQL

Le site Use the index, Luke[6] , maintenu par Markus Winand, propose une version en ligne de son livre *SQL Performance Explained*. Une version française est par ailleurs disponible sous le titre *SQL : au cœur des performances*.

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

2.5.1 UTILITÉ D'UN INDEX

- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire récupérer directement l'enregistrement de l'index s'il contient toutes les colonnes nécessaires
- En complément, un index facilite :
  - certains tris
  - certains agrégats
- Et est utilisé pour les contraintes d'unicité !

PostgreSQL propose différentes formes d'index :

- index classique sur une seule colonne d'une table ;
- index composite sur plusieurs colonnes d'une table ;
- index partiel, en restreignant les données indexées avec une clause `WHERE` ;
- index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table.

2.5.2 INDEX ET SELECT

Un index améliore les `SELECT`

Sans index :

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

```
=# SELECT * FROM t1 WHERE i = 10000;
Temps : 1760,017 ms
```

Avec index :

```
=# CREATE INDEX idx_t1_i ON t1 (i);
=# SELECT * FROM t1 WHERE i = 10000;
Temps : 27,711 ms
```

L'index est une structure de données qui permet d'accéder rapidement à l'information recherchée. À l'image de l'index d'un livre, pour retrouver un thème rapidement, on préférera utiliser l'index du livre plutôt que lire l'intégralité du livre jusqu'à trouver le passage qui nous intéresse. Dans une base de données, l'index a un rôle équivalent. Plutôt que de lire une table dans son intégralité, la base de données utilisera l'index pour ne lire qu'une faible portion de la table pour retrouver les données recherchées.

Pour la requête d'exemple (avec une table de 20 millions de lignes), on remarque que l'optimiseur n'utilise pas le même chemin selon que l'index soit présent ou non. Sans index, PostgreSQL réalise un parcours séquentiel de la table :

```
postgres=# EXPLAIN SELECT * FROM t1 WHERE i = 10000;
                            QUERY PLAN
-------------------------------------------------------------------
 Gather  (cost=1000.00..193661.66 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t1  (cost=0.00..192661.56 rows=1 width=4)
         Filter: (i = 10000)
(4 rows)
```

Lorsqu'il est présent, PostgreSQL l'utilise car l'optimiseur estime que son parcours ne récupèrera qu'une seule ligne sur les 10 millions que compte la table :

```
postgres=# EXPLAIN SELECT * FROM t1 WHERE i = 10000;
                             QUERY PLAN
---------------------------------------------------------------------
 Index Only Scan using t1_i_idx on t1  (cost=0.44..8.46 rows=1 width=4)
   Index Cond: (i = 10000)
(2 rows)
```

Mais l'index n'accélère pas seulement la simple lecture de données, il permet également d'accélérer les tris et les agrégations, comme le montre l'exemple suivant sur un tri :

```
postgres=# EXPLAIN SELECT * FROM t1
WHERE i BETWEEN 1000 AND 1200 ORDER BY i DESC;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Only Scan Backward using t1_i_idx on t1  (cost=0.44..12.26 rows=191 width=4)
   Index Cond: ((i >= 1000) AND (i <= 1200))
(2 rows)
```

2.5.3 INDEX ET INSERT

La présence d'un index ralentit les mises à jour :

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

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

Compromis à trouver :

- favoriser les lectures
- mais pas au détriment des écritures

La présence d'un index ralentit les écritures sur une table. En effet, il faut non seulement ajouter ou modifier les données dans la table, mais il faut également maintenir le ou les index de cette table.

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

Soit une table t2 telle que :

```
CREATE TABLE t2 (
    id SERIAL PRIMARY KEY,
    valeur INTEGER,
    commentaire TEXT
);
```

La table est chargée avec pour seul index présent celui sur la clé primaire :

```
=# INSERT INTO t2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Temps : 148299,717 ms
```

Un index supplémentaire est créé sur une colonne de type entier :

```
=# CREATE INDEX idx_t2_valeur ON t2 (valeur);
=# INSERT INTO t2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Temps : 195933,515 ms
```

Un index supplémentaire est encore créé, mais cette fois sur une colonne de type texte :

```
=# CREATE INDEX idx_t2_commentaire ON t2 (commentaire);
=# INSERT INTO t2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Temps : 422662,401 ms
```

On peut comparer ces temps à l'insertion dans une table similaire dépourvue d'index :

```
=# CREATE TABLE t3 AS SELECT * FROM t2;
=# INSERT INTO t3 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Temps : 55945,787 ms
```

La table `t2` a été vidée préalablement pour chaque test.

2.5.4 QUELLES REQUÊTES OPTIMISER ?

- Seules un certain nombre de requêtes sont critiques
  - utilisation d'outil de profiling pour les identifier
  - le travail d'optimisation se porte sur celles-ci uniquement
- Détermination des requêtes critiques :
  - longues en temps cumulé, coûteuses en ressources serveur
  - longues et interactives, mauvais ressenti des utilisateurs

Toutes les requêtes ne sont pas critiques, seul un certain nombre d'entre elles méritent une attention particulière.

Il y a deux façon de déterminer les requêtes qui nécessitent d'être travaillées. La première dépend du ressenti utilisateur, il faudra en priorité traiter les requêtes interactives. Certaines auront déjà d'excellents temps de réponse, d'autres pourront être améliorées encore. Il faudra déterminer non seulement le temps de réponse maximal attendu pour une requête, mais vérifier aussi le temps total de réponse de l'application.

L'autre méthode pour déterminer les requêtes à optimiser consiste à utiliser des outils de profiling habituels (pgBadger, pg_stat_statements, pg_ stat_plans). Ces outils permettront de déterminer les requêtes les plus fréquemment exécutées et permettront d'établir un classement des requêtes qui ont nécessité le plus de temps cumulé à leur exécution (voir onglet *Time consuming queries (N)* d'un rapport pgBadger). Les requêtes les plus fréquemment exécutées méritent également qu'on leur porte attention, leur optimisation peut permettre d'économiser quelques ressources du serveur.

17.12

En utilisant l'extension pg_stat_statements, la requête suivante permet de déterminer les requêtes dont les temps d'exécution cumulés sont les plus importants :

```
SELECT r.rolname, d.datname, s.calls, s.total_time,
       s.calls / s.total_time AS avg_time, s.query
  FROM pg_stat_statements s
  JOIN pg_roles r
    ON (s.userid=r.oid)
  JOIN pg_database d
    ON (s.dbid = d.oid)
 ORDER BY s.total_time DESC
 LIMIT 10;
```

Toujours avec pg_stat_statements, la requête suivante permet de déterminer les requêtes les plus fréquemment appelées :

```
SELECT r.rolname, d.datname, s.calls, s.total_time,
       s.calls / s.total_time AS avg_time, s.query
  FROM pg_stat_statements s
  JOIN pg_roles r
    ON (s.userid=r.oid)
  JOIN pg_database d
    ON (s.dbid = d.oid)
 ORDER BY s.calls DESC
 LIMIT 10;
```

2.5.5 INDEX SPÉCIALISÉS

- PostgreSQL dispose d'index spécialisés :
  - `GIN` (*Generalized Inverted Index*)
  - `GiST` (*Generalized Search Tree*)
  - `BRIN` (*Block Range INdex*)

PostgreSQL propose plusieurs types d'index :

- les classiques `B-Tree`
- les index `hash`
- les index spécialisés `GIN` et `GiST`
- les index `BRIN`

Les index `B-Tree` sont les index les plus couramment utilisés, pour assurer des contraintes d'unicité ou simplement pour assurer les recherches avec les opérateurs de comparaisons habituels.

Les index hash sont au contraire très peu utilisés. Ils n'étaient pas journalisées jusqu'en version 10. Avant cette version, ils étaient souvent corrompus à chaque arrêt brutal du serveur, et jamais répliqués.

Ces deux types d'index permettent d'indexer des valeurs scalaires.

2.5.6 INDEX GIN ET GIST

- Ils permettent d'indexer des données non-scalaires :
  - Intervalles de dates
  - formes géométriques, données géographiques
  - Trigrammes, Full Text
  - Tableaux

Les index de type `GiST` sont des structures d'index généralisés. Ce sont des arbres `B+Tree`, qui autorisent un développeur, connaissant bien un type de données particulier, à créer ses propres méthodes d'accès aux données. Les index de type `GIN` sont de la famille de index inversés.

On retrouve ces deux types d'index associés à différentes fonctionnalités de PostgreSQL. La recherche plein texte (*full text search*) est la plus connue. Grâce à ces types d'index, il est aussi possible d'indexer les types de données `range`.

Ainsi, l'extension `pg_trgm` s'appuie sur des trigrammes pour répondre aux requêtes de type `LIKE '%motif%'`, qui ne peuvent bénéficier d'optimisation avec des index traditionnels :

```
base=# CREATE INDEX idx2 ON mots USING gist (mot gist_trgm_ops);
CREATE INDEX
base=# EXPLAIN ANALYZE SELECT * FROM mots WHERE mot LIKE '%jour%';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on mots  (cost=7.41..277.32 rows=75 width=10)
                           (actual time=50.362..50.677 rows=352 loops=1)
   Recheck Cond: (mot ~~ '%jour%'::text)
   ->  Bitmap Index Scan on idx2  (cost=0.00..7.39 rows=75 width=0)
                                  (actual time=50.322..50.322 rows=352 loops=1)
         Index Cond: (mot ~~ '%jour%'::text)
 Total runtime: 51.226 ms
```

Plus d'information sur :

- les index `GiST`[7]

[7] http://en.wikipedia.org/wiki/GiST

- les index GIN[8]

2.5.7 INDEX BRIN

- Les index BRIN sont utiles pour les grosses volumétries
- Les données sont corrélées avec leur emplacement physique

Un index BRIN stocke un « résumé » d'un ensemble de blocs. Cela réduit la taille de l'index et permet d'exclure un ensemble de blocs lors d'une recherche.

Cible :

- fortes volumétries, *big data* ;
- index classiques volumineux.

Plus d'information sur les index BRIN[9] .

2.5.8 INDEX FONCTIONNELS

- Si une fonction est appliquée à une colonne dans un prédicat :

```
SELECT ... FROM table WHERE f(colonne)=C
```

  - l'optimiseur n'utilise pas d'index
  - dénote un problème probable de normalisation
- Analogie : chercher dans un dictionnaire français
  - WHERE anglais(mot)='cheval' => il faut traduire chaque mot lu

---

2.5.9 INDEX FONCTIONNELS

Usage classique :

- Recherche sans la casse
- Avec un index fonctionnel, l'optimiseur sait utiliser un index :

```
CREATE INDEX index ON dictionnaire_fr(anglais(mot))
```

[8] http://en.wikipedia.org/wiki/Inverted_index
[9] https://kb.dalibo.com/conferences/index_brin/index_brin_pgday

Lorsqu'une fonction est appliquée à la valeur d'une colonne dans un prédicat, l'optimiseur ne sait pas utiliser un index normal pour répondre rapidement à une telle requête. Il utilisera donc un parcours séquentiel de la table, pour appliquer la fonction à toutes les valeurs de la colonne à laquelle elle s'applique, afin de vérifier le prédicat. Cela aura des conséquences désastreuses sur les temps de réponse : non seulement le parcours séquentiel peut être long, mais la fonction sera exécutée autant de fois qu'il y a de lignes dans la table, ajoutant encore un délai important au temps de réponse, ainsi qu'une énorme consommation de ressources CPU.

Ce problème apparaît souvent sur des manipulations de dates. En général, il est résolu en plaçant la transformation du côté de la constante. Par exemple, la requête suivante retourne toutes les commandes de l'année 2011, mais la fonction `extract` est appliquée à la colonne `date_commande`. L'optimiseur ne peut donc utiliser un index :

```
tpc=# EXPLAIN SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;
                                QUERY PLAN
--------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..5364.12 rows=844 width=77)
   Filter: (date_part('year'::text,
     (date_commande)::timestamp without time zone) = 2011::double precision)
```

En réécrivant le prédicat, l'index est bien utilisé :

```
tpc=# EXPLAIN SELECT * FROM commandes
WHERE date_commande BETWEEN '01-01-2011'::date AND '31-12-2011'::date;
                            QUERY PLAN
------------------------------------------------------------------------
 Bitmap Heap Scan on commandes  (cost=523.85..3302.80 rows=24530 width=77)
   Recheck Cond: ((date_commande >= '2011-01-01'::date)
                  AND (date_commande <= '2011-12-31'::date))
   ->  Bitmap Index Scan on idx_commandes_date_commande
           (cost=0.00..517.72 rows=24530 width=0)
         Index Cond: ((date_commande >= '2011-01-01'::date)
                      AND (date_commande <= '2011-12-31'::date))
```

Mais dans d'autres cas, une telle réécriture de la requête sera impossible. PostgreSQL permet d'indexer le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une même table. Au lieu d'indexer une simple colonne, on indexera le résultat de la fonction, par exemple :

```
CREATE INDEX index ON table(f(colonne))
```

La seule contrainte est que la fonction indexée doit être catégorisée `IMMUTABLE`, indiquant que la fonction retournera toujours le même résultat quand elle est appelée avec les mêmes arguments (elle ne dépend pas du contenu de la base, ni n'a de comportement

non-déterministe comme random ou clock_timestamp()).

2.6 IMPACT DES TRANSACTIONS

- Prise de verrous : ils ne sont relâchés qu'à la fin
  - `COMMIT`
  - `ROLLBACK`
- Validation des données sur le disque au `COMMIT`
  - Écriture synchrone : coûteux
- Faire des transactions qui correspondent au fonctionnel
- Si traitement lourd, préférer des transactions de grande taille

Réaliser des transactions permet de garantir l'atomicité des opérations : toutes les modifications sont validées (`COMMIT`), ou tout est annulé (`ROLLBACK`). Il n'y a pas d'état intermédiaire. Le `COMMIT` garantit aussi la durabilité des opérations : une fois que le `COMMIT` a réussi, la base de données garantit que les opérations ont bien été stockées, et ne seront pas perdues… sauf perte du matériel (disque) sur lequel ont été écrites ces opérations bien sûr.

L'opération `COMMIT` a donc bien sûr un coût : il faut garantir que les données sont bien écrites sur le disque, il faut les écrire sur le disque (évidemment), mais aussi attendre la confirmation du disque. Que les disques soient mécaniques ou SSD ne change pas grand chose, cette opération est coûteuse :

- Un disque dur doit se positionner au bon endroit (journal de transaction), écrire la donnée, et confirmer au système que c'est fait. Il faudra donc compter le temps de déplacement de la tête et de rotation du disque pour se positionner, et le temps d'écriture (celui-ci sera plus court). Compter 1 à 5 millisecondes.
- Un disque SSD doit écrire réellement le bloc demandé. Il faudra donc faire un `ERASE` du bloc puis une nouvelle écriture de celui-ci. C'est l'`ERASE` qui est lent (de l'ordre de 2 millisecondes, donc à peine plus rapide qu'un disque dur mécanique).

On peut limiter l'impact des écritures synchrone en utilisant un cache en écriture comme en proposent les serveurs haut de gamme.

Les transactions devant garantir l'unicité des opérations, il est nécessaire qu'elles prennent des verrous : sur les enregistrements modifiés, sur les tables accédées (pour éviter les changements de structure pendant leur manipulation), sur des prédicats (dans certains cas compliqués comme le niveau d'isolation `serializable`)… tout ceci a un impact :

- Le temps d'acquisition des verrous, bien sûr

- Mais aussi les sources de contention entre chaque session

Il est donc très difficile de déterminer la bonne durée d'une transaction. Trop courte : on génère beaucoup d'opérations synchrones. Trop longue : on risque de bloquer d'autres sessions. Le mieux est de coller au besoin fonctionnel.

2.6.1 VERROUILLAGE ET CONTENTION

- Chaque transaction prend des verrous :
  - sur les objets (tables, index, etc.) pour empêcher au moins leur suppression ou modification de structure pendant leur travail
  - sur les enregistrements
  - libérés à la fin de la transaction : les transactions très longues peuvent donc être problématiques
- Sous PostgreSQL, on peut quand même lire un enregistrement en cours de modification : on voit l'ancienne version (MVCC)

Afin de garantir une isolation correcte entre les différentes sessions, le SGBD a besoin de protéger certaines opérations. On ne peut par exemple pas autoriser une session à modifier le même enregistrement qu'une autre, tant qu'on ne sait pas si cette dernière a validé ou annulé sa modification. On a donc un verrouillage des enregistrements modifiés.

Certains SGBD verrouillent totalement l'enregistrement modifié. Celui-ci n'est plus accessible même en lecture tant que la modification n'a pas été validée ou annulée. Cela a l'avantage d'éviter aux sessions en attente de voir une ancienne version de l'enregistrement, mais le défaut de les bloquer, et donc de fortement dégrader les performances.

PostgreSQL, comme Oracle, utilise un modèle dit MVCC (Multi-Version Concurrency Control), qui permet à chaque enregistrement de cohabiter en plusieurs versions simultanées en base. Cela permet d'éviter que les écrivains ne bloquent les lecteurs ou les lecteurs ne bloquent les écrivains. Cela permet aussi de garantir un instantané de la base à une requête, sur toute sa durée, voire sur toute la durée de sa transaction si la session le demande (`BEGIN ISOLATION LEVEL REPEATABLE READ`).

Dans le cas où il est réellement nécessaire de verrouiller un enregistrement sans le mettre à jour immédiatement (pour éviter une mise à jour concurrente), il faut utiliser l'ordre SQL `SELECT FOR UPDATE`.

2.6.2 DEADLOCKS

- Du fait de ces verrous :
  - On peut avoir des `deadlocks` (verrous mortels)
  - En théorie, on peut les éviter (en prenant toujours les verrous dans le même ordre)
  - En pratique, ça n'est pas toujours possible ou commode
  - Les SGBD tuent une des transactions responsables du `deadlock`
  - Une application générant de nombreux `deadlocks` est ralentie

Les `deadlocks` se produisent quand plusieurs sessions acquièrent simultanément des verrous et s'interbloquent. Par exemple :

Session 1	Session 2	
`BEGIN`	`BEGIN`	
`UPDATE demo SET a=10`		
`WHERE a=1;`		
	`UPDATE demo SET a=11`	
	`WHERE a=2;`	
`UPDATE demo SET a=11`		*Session 1 bloquée.*
`WHERE a=2;`		*Attend session 2.*
	`UPDATE demo SET a=10`	*Session 2 bloquée.*
	`WHERE a=1;`	*Attend session 1.*

Bien sûr, la situation ne reste pas en l'état. Une session qui attend un verrou appelle au bout d'un temps court (une seconde par défaut sous PostgreSQL) le gestionnaire de `deadlock`, qui finira par tuer une des deux sessions. Dans cet exemple, il sera appelé par la session 2, ce qui débloquera la situation.

Une application qui a beaucoup de `deadlocks` a plusieurs problèmes :

- Les transactions attendent beaucoup (utilisation de toutes les ressources machine difficile)
- Certaines finissent annulées et doivent donc être rejouées (travail supplémentaire)

Dans notre exemple, on aurait pu éviter le problème, en définissant une règle simple : toujours verrouiller par valeurs de a croissante. Dans la pratique, sur des cas complexes, c'est bien sûr bien plus difficile à faire. Par ailleurs, un `deadlock` peut impliquer plus de deux transactions. Mais simplement réduire le volume de `deadlocks` aura toujours un impact très positif sur les performances.

On peut aussi déclencher plus rapidement le gestionnaire de `deadlock`. 1 seconde, c'est quelquefois une éternité dans la vie d'une application. Sous PostgreSQL, il suffit de modifier le paramètre `deadlock_timeout`. Plus cette variable sera basse, plus le traitement de détection de `deadlock` sera déclenché souvent. Et celui-ci peut être assez gourmand si de nombreux verrous sont présents, puisqu'il s'agit de détecter des cycles dans les dépendances de verrous.

2.7 BIBLIOGRAPHIE

- Ce document s'appuie sur de nombreuses sources.
- Si vous souhaitez approfondir les points abordés :
  - *The World and the Machine*, **Michael Jackson**
  - *The Art of SQL*, **Stéphane Faroult**
  - *Refactoring SQL Applications*, **Stéphane Faroult**
  - *SQL Performance Explained*, **Markus Winand**
  - *Introduction aux bases de données*, **Chris Date**
  - Vidéos de **Stéphane Faroult** (*roughsealtd*) sous Youtube

Bibliographie :

- *The World and the Machine*, **Michael Jackson** (version en ligne[10])
- *The Art of SQL*, **Stéphane Faroult**, ISBN-13: 978-0596008949
- *Refactoring SQL Applications*, **Stéphane Faroult**, ISBN-13: 978-0596514976
- *SQL Performance Explained*, **Markus Winand** :
  - site internet (fr)[11]
  - FR: ISBN-13: 978-3950307832
  - EN: ISBN-13: 978-3950307825
- *Introduction aux bases de données*, **Chris Date**
  - FR: ISBN-13: 978-2711748389 (8e édition)
  - EN: ISBN-13: 978-0321197849
- Vidéos de **Stéphane Faroult**, roughsealtd sous Youtube[12] :
  - Vidéo 1[13]
  - Vidéo 2[14]

[10] http://users.mct.open.ac.uk/mj665/icse17kn.pdf
[11] http://use-the-index-luke.com/fr
[12] http://www.youtube.com/channel/UCW6zsYGFckfczPKUUVdvYjg
[13] http://www.youtube.com/watch?v=40Lnoyv-sXg&list=PL767434BC92D459A7
[14] http://www.youtube.com/watch?v=GbZgnAINjUw&list=PL767434BC92D459A7

17.12

- Vidéo 3[15]

2.8 TRAVAUX PRATIQUES

2.8.1 ENONCÉ

- Charger le fichier de TP fourni par le formateur :

```
createdb tp
pg_restore -d tp tp.dmp
```

(Note pour le formateur : utiliser `genere_nf_1.sh` et `genere_pagination.sh`, sur une instance ayant une base « tp ».)

Exécuter aussi un `VACUUM VERBOSE ANALYZE` sur la base, afin d'avoir les statistiques à jour !

Normalisation de base

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

- Renommer la table en `voitures_orig`.
- Écrire des requêtes permettant d'éclater cette table en trois tables: `voitures`, `caracteristiques` et `caracteristiques_voitures`. Ne pas supprimer la table `voitures_orig` (nous en aurons besoin plus tard). Mettre en place les contraintes d'intégrité : clé primaire sur chaque table, et clés étrangères.
  - **Attention** : la table de départ contient des immatriculations en doublon !
- Comparer les performances d'une recherche de voiture ayant un toit ouvrant avec l'ancien et le nouveau modèle.

Entité-clé-valeur

- La table `voiture` existe aussi dans cette base au format « entité/clé/valeur » : table `voitures_ecv`. Trouvez toutes les caractéristiques de toutes les voitures ayant un toit ouvrant dans cette table.

[15] http://www.youtube.com/watch?v=y70FmugnhPU&list=PL767434BC92D459A7

- Convertir cette table pour qu'elle utilise un `hstore`, créer un index sur la colonne de type `hstore`, réécrire la requête et comparer. Il y a de nombreuses solutions pour écrire cette conversion. Se reporter à la documentation de l'extension hstore[16] .
- Afficher toutes les voitures ayant un ABS **et** un toit ouvrant, avec les deux modèles.

Sans toucher au schéma

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

- Trouver cette fonction (chercher dans les fonctions de découpage de chaîne de caractères, dans la documentation de PostgreSQL).
- Définir l'index (c'est un index sur un type tableau).
- Écrire la requête et son plan.

Pagination et index

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

Dans la base TP existe une table `posts`. C'est une version simplifiée d'une table de forum. Nous voulons afficher très rapidement les messages (*posts*) d'un article : les 10 premiers, puis du 11 au 20, etc. le plus rapidement possible. Nous allons examiner les différentes stratégies possibles.

- Écrire une requête permettant de récupérer les 10 premiers posts de l'article 12. La table a été créée sans index, la requête va être très lente. Utiliser `id_post`, pas le `timestamp` (il servira dans le prochain TP).
- Créer un index permettant d'améliorer cette requête.
- Écrire la même requête permettant de récupérer les 10 posts suivants. Puis du post 901 au 921. Que constate-t-on sur le plan d'exécution ?
- Trouver une réécriture de la requête pour trouver directement les posts 901 à 911 une fois connu le post 900 récupéré au travers de la pagination.

Clauses WHERE et pièges

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

[16] http://www.postgresql.org/docs/9.6/static/hstore.html

- La requête `select * from posts where to_char(ts,'YYYYMM')='201302'` retourne tous les enregistrements de février 2013. Examiner son plan d'exécution. Où est le problème ?
- Réécrire la clause `WHERE`.
- Plus compliqué : retourner tous les posts ayant eu lieu un dimanche, en 2013, en passant par un index et en une seule requête.
  - Indice : il est possible de générer la liste de tous les dimanches de l'année 2013 avec `generate_series()`.
- Pourquoi la requête suivante est-elle lente ?

```
select * from posts where id_article = (
 select round(random()*(select max(id_article) from posts))
)
```

2.8.2 SOLUTIONS

Normalisation de base

> Renommer la table en `voitures_orig`

```
alter table voitures rename TO voitures_orig;
```

> Écrire des requêtes permettant d'éclater cette table en trois tables: `voitures`, `caracteristiques` et `caracteristiques_voitures`, et mettre les contraintes d'intégrité en place.

```
create table voitures as
  select distinct on (immatriculation) immatriculation, modele
  from voitures_orig ;

alter table voitures add primary key (immatriculation);

create table caracteristiques
  as select *
     from (
             select distinct
              regexp_split_to_table(caracteristiques,',') caracteristique
             from voitures_orig)
          as tmp
```

```
      where caracteristique <> '' ;

  alter table caracteristiques add primary key (caracteristique);

  create table caracteristiques_voitures
    as select distinct *
    from (
         select
           immatriculation,
           regexp_split_to_table(caracteristiques,',') caracteristique
         from voitures_orig
        )
      as tmp
     where caracteristique <> '';

  alter table caracteristiques_voitures
  add primary key (immatriculation,caracteristique);

  alter table caracteristiques_voitures
  add foreign key (immatriculation)
  references voitures(immatriculation);

  alter table caracteristiques_voitures
  add foreign key (caracteristique)
  references caracteristiques(caracteristique);
```

> Comparer les performances d'une recherche de voiture ayant un toit ouvrant avec l'ancien et le nouveau modèle.

```
# EXPLAIN ANALYZE SELECT * FROM voitures_orig
  WHERE caracteristiques ~ E'[[:<:]]toit ouvrant[[:>:]]';
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on voitures_orig  (cost=0.00..639.90 rows=5 width=96)
                            (actual time=0.050..254.655 rows=8343 loops=1)
   Filter: (caracteristiques ~ '[[:<:]]toit ouvrant[[:>:]]'::text)
   Rows Removed by Filter: 91657
 Total runtime: 255.168 ms
```

ou plus simplement (mais moins sûr) :

```
# EXPLAIN ANALYZE SELECT * FROM voitures_orig
  WHERE caracteristiques like '%toit ouvrant%';
                                  QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on voitures_orig  (cost=0.00..1321.10 rows=8469 width=25)
                            (actual time=0.023..47.563 rows=8343 loops=1)
```

```
   Filter: (caracteristiques ~~ '%toit ouvrant%'::text)
   Rows Removed by Filter: 91657
 Total runtime: 48.155 ms
```

Avec le nouveau schéma :

```
# EXPLAIN ANALYZE  SELECT * FROM voitures
  WHERE EXISTS (
    SELECT 1 FROM caracteristiques_voitures
    WHERE caracteristiques_voitures.immatriculation=voitures.immatriculation
    AND caracteristique = 'toit ouvrant'
  );
                              QUERY PLAN
-----------------------------------------------------------------------
 Merge Semi Join  (cost=0.89..2501.47 rows=8263 width=16)
                  (actual time=0.095..128.244 rows=8343 loops=1)
   Merge Cond: (voitures.immatriculation =
                caracteristiques_voitures.immatriculation)
   ->  Index Scan using voitures_pkey on voitures
           (cost=0.42..1600.50 rows=99992 width=16)
           (actual time=0.023..39.480 rows=99992 loops=1)
   ->  Index Only Scan using caracteristiques_voitures_pkey
                       on caracteristiques_voitures
                          (cost=0.41..547.70 rows=8263 width=10)
                          (actual time=0.039..9.677 rows=8343 loops=1)
         Index Cond: (caracteristique = 'toit ouvrant'::text)
         Heap Fetches: 0
 Total runtime: 128.872 ms
```

On garde sensiblement le même temps d'exécution (seule la syntaxe en expression régulière est sûre, et encore cette version est-elle encore trop simple). On serait par contre beaucoup plus rapides si cette option était rare. Ici, 10% des voitures ont l'option, le filtre n'est donc pas très discriminant.

Ce qu'on gagne réellement, c'est la garantie que les caractéristiques ne seront que celles existant dans la table `caractéristique`, ce qui évite d'avoir à réparer la base plus tard.

Si on cherche une option rare ou n'existant pas :

```
EXPLAIN ANALYZE  SELECT * FROM voitures
WHERE EXISTS (
    SELECT 1
    FROM caracteristiques_voitures
    WHERE caracteristiques_voitures.immatriculation=voitures.immatriculation
    AND caracteristique = 'toit'
);
                              QUERY PLAN
-----------------------------------------------------------------------
```

```
 Nested Loop  (cost=0.84..1.19 rows=1 width=16)
              (actual time=0.017..0.017 rows=0 loops=1)
   ->  Index Only Scan using idx_caracteristique_immat
                       on caracteristiques_voitures
                            (cost=0.42..0.54 rows=1 width=10)
                            (actual time=0.016..0.016 rows=0 loops=1)
         Index Cond: (caracteristique = 'toit'::text)
         Heap Fetches: 0
   ->  Index Scan using voitures_new_pkey on voitures
             (cost=0.42..0.64 rows=1 width=16)
             (never executed)
         Index Cond: (immatriculation =
                      caracteristiques_voitures.immatriculation)
 Total runtime: 0.049 ms
(7 lignes)
```

Avec l'ancien schéma, on devait lire la table `voiture` en entier.

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

```
# EXPLAIN ANALYZE SELECT *
  FROM voitures
  JOIN caracteristiques_voitures AS cr1 USING (immatriculation)
  JOIN caracteristiques_voitures AS cr2 USING (immatriculation)
  JOIN caracteristiques_voitures AS cr3 USING (immatriculation)
  WHERE cr1.caracteristique = 'toit ouvrant'
  AND cr2.caracteristique = 'abs'
  AND cr3.caracteristique='4 roues motrices';
                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=1.66..2017.28 rows=57 width=58)
              (actual time=0.174..57.544 rows=478 loops=1)
   ->  Merge Join  (cost=1.24..1740.13 rows=603 width=72)
                   (actual time=0.144..50.162 rows=478 loops=1)
         Merge Cond: (cr1.immatriculation = cr2.immatriculation)
         ->  Merge Join  (cost=0.83..1157.84 rows=2198 width=48)
                         (actual time=0.083..33.680 rows=1529 loops=1)
               Merge Cond: (cr1.immatriculation = cr3.immatriculation)
               ->  Index Only Scan using caracteristiques_voitures_pkey
                                   on caracteristiques_voitures cr1
                                    (cost=0.41..547.70 rows=8263 width=24)
                                    (actual time=0.034..9.725 rows=8343 loops=1)
                     Index Cond: (caracteristique = 'toit ouvrant'::text)
                     Heap Fetches: 0
               ->  Index Only Scan using caracteristiques_voitures_pkey
```

```
                                        on caracteristiques_voitures cr3
                                (cost=0.41..547.01 rows=8194 width=24)
                                (actual time=0.029..10.297 rows=8216 loops=1)
                    Index Cond: (caracteristique = '4 roues motrices'::text)
                    Heap Fetches: 0
        ->  Index Only Scan using caracteristiques_voitures_pkey
                                 on caracteristiques_voitures cr2
                            (cost=0.41..549.62 rows=8455 width=24)
                            (actual time=0.028..9.818 rows=8300 loops=1)
              Index Cond: (caracteristique = 'abs'::text)
              Heap Fetches: 0
   ->  Index Scan using voitures_pkey on voitures
           (cost=0.42..0.45 rows=1 width=16)
           (actual time=0.013..0.014 rows=1 loops=478)
         Index Cond: (immatriculation = cr1.immatriculation)
 Total runtime:
Total runtime: 57.710 ms
```

Entité-clé-valeur

> La table `voiture` existe aussi dans cette base au format « entité/clé/valeur » : table `voitures_ecv`. Trouvez toutes les caractéristiques de toutes les voitures ayant un toit ouvrant dans cette table.

```
# EXPLAIN ANALYZE SELECT * FROM voitures_ecv
  WHERE EXISTS (
    SELECT 1 FROM voitures_ecv test
    WHERE test.entite=voitures_ecv.entite
    AND cle = 'toit ouvrant' and valeur = true
  );
                                QUERY PLAN
-------------------------------------------------------------------------
 Merge Semi Join  (cost=0.83..1881.36 rows=15136 width=25)
                  (actual time=0.052..94.150 rows=17485 loops=1)
   Merge Cond: (voitures_ecv.entite = test.entite)
   ->  Index Scan using voitures_ecv_pkey on voitures_ecv
           (cost=0.41..939.03 rows=57728 width=25)
           (actual time=0.013..20.755 rows=57728 loops=1)
   ->  Materialize  (cost=0.41..588.92 rows=8207 width=10)
                    (actual time=0.028..14.011 rows=8343 loops=1)
         ->  Index Scan using voitures_ecv_pkey on voitures_ecv test
                    (cost=0.41..568.40 rows=8207 width=10)
                    (actual time=0.025..11.508 rows=8343 loops=1)
               Index Cond: (cle = 'toit ouvrant'::text)
```

```
              Filter: valeur
 Total runtime: 95.398 ms
```

> Convertir cette table pour qu'elle utilise un `hstore`, créer un index sur la colonne de type `hstore`, réécrire la requête et comparer.

```
CREATE EXTENSION HSTORE;

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

ALTER TABLE voitures_hstore ADD PRIMARY KEY (immatriculation);

CREATE INDEX voitures_hstore_caracteristiques ON voitures_hstore
    USING gist (caracteristiques);

EXPLAIN (ANALYZE,BUFFERS)
    SELECT *
    FROM voitures_hstore
    WHERE caracteristiques @> '"toit ouvrant" => true';
                               QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using voitures_hstore_caracteristiques on voitures_hstore
        (cost=0.28..4.62 rows=37 width=64)
        (actual time=0.093..18.026 rows=8343 loops=1)
   Index Cond: (caracteristiques @> '"toit ouvrant"=>"true"'::hstore)
   Buffers: shared hit=6731
 Total runtime: 19.195 ms
```

> Afficher toutes les voitures ayant un ABS **et** un toit ouvrant, avec les deux modèles.

Avec `voitures_ecv` :

```
# EXPLAIN (ANALYZE,BUFFERS)
    SELECT * FROM voitures_ecv
    WHERE EXISTS (
        SELECT 1
        FROM voitures_ecv test
        WHERE test.entite=voitures_ecv.entite
        AND cle = 'toit ouvrant' AND valeur = true
    )
```

```
    AND EXISTS (
        SELECT 1 FROM voitures_ecv test
        WHERE test.entite=voitures_ecv.entite
        AND cle = 'abs' AND valeur = true
    );
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Merge Semi Join  (cost=1.24..2549.89 rows=4012 width=25)
                  (actual time=0.121..152.364 rows=5416 loops=1)
   Merge Cond: (test.entite = test_1.entite)
   Buffers: shared hit=2172
   ->  Merge Semi Join  (cost=0.83..1881.36 rows=15136 width=35)
                        (actual time=0.078..116.027 rows=17485 loops=1)
         Merge Cond: (voitures_ecv.entite = test.entite)
         Buffers: shared hit=1448
         ->  Index Scan using voitures_ecv_pkey on voitures_ecv
                   (cost=0.41..939.03 rows=57728 width=25)
                   (actual time=0.018..24.430 rows=57728 loops=1)
               Buffers: shared hit=724
         ->  Materialize  (cost=0.41..588.92 rows=8207 width=10)
                          (actual time=0.042..15.858 rows=8343 loops=1)
               Buffers: shared hit=724
               ->  Index Scan using voitures_ecv_pkey on voitures_ecv test
                         (cost=0.41..568.40 rows=8207 width=10)
                         (actual time=0.038..12.204 rows=8343 loops=1)
                     Index Cond: (cle = 'toit ouvrant'::text)
                     Filter: valeur
                     Buffers: shared hit=724
   ->  Index Scan using voitures_ecv_pkey on voitures_ecv test_1
             (cost=0.41..569.34 rows=8301 width=10)
             (actual time=0.036..11.779 rows=8300 loops=1)
         Index Cond: (cle = 'abs'::text)
         Filter: valeur
         Buffers: shared hit=724
 Total runtime: 153.151 ms
```

Avec `hstore` :

```
# EXPLAIN (ANALYZE,BUFFERS)
    SELECT * FROM voitures_hstore
    WHERE caracteristiques @> '"toit ouvrant" => true, "abs" => true';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using voitures_hstore_caracteristiques on voitures_hstore
        (cost=0.28..4.62 rows=37 width=55)
        (actual time=0.102..4.759 rows=1538 loops=1)
   Index Cond: (caracteristiques @> '"abs"=>"true",
```

```
                "toit ouvrant"=>"true"'::hstore)
   Buffers: shared hit=1415
 Total runtime: 5.063 ms
```

On voit que la solution entité/clé/valeur n'arrive pas à être optimisée correctement. Le `hstore` par contre est de plus en plus rapide au fur et à mesure que la sélectivité augmente.

Sans toucher au schéma

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

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

La fonction est `regexp_split_to_array` :

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

```
 immatriculation | modele |  regexp_split_to_array
-----------------+--------+-------------------------------------------------------------------------
 XZ-971-EA       | twingo | {climatisation,abs,"jantes aluminium"}
 JC-269-WE       | twingo | {"jantes aluminium","4 roues motrices",
                            climatisation,"regulateur de vitesse",
                            "boite automatique","toit ouvrant"}
 KU-380-XU       | kangoo | {"jantes aluminium",
                           "boite automatique",abs}
 VW-418-JM       | kangoo | {abs,"jantes aluminium","boite automatique"}
 ZK-505-FO       | clio   | {"toit ouvrant"}
 PX-156-TH       | kangoo | {"jantes aluminium","regulateur de vitesse",
                            climatisation}
 LB-711-JC       | clio   | {"regulateur de vitesse",abs,"boite automatique",
                            "jantes aluminium","4 roues motrices",
                            climatisation}
 BK-870-IB       | clio   | {"boite automatique","4 roues motrices",
                            abs,"jantes aluminium"}
 VQ-123-HK       | twingo | {""}
 OR-647-BK       | clio   | {""}
(10 lignes)
```

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

- Définir l'index :

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

- Écrire la requête et son plan :

```
# EXPLAIN ANALYZE
    SELECT * FROM voitures_orig
    WHERE regexp_split_to_array(caracteristiques,',') @> '{"toit ouvrant"}';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on voitures_orig (cost=5.28..49.78 rows=500 width=25)
                                   (actual time=4.159..9.357 rows=8343 loops=1)
   Recheck Cond: (regexp_split_to_array(caracteristiques, ','::text) @>
                  '{"toit ouvrant"}'::text[])
   ->  Bitmap Index Scan on idx_voitures_array
           (cost=0.00..5.15 rows=500 width=0)
           (actual time=3.915..3.915 rows=8343 loops=1)
         Index Cond: (regexp_split_to_array(caracteristiques, ','::text) @>
                      '{"toit ouvrant"}'::text[])
 Total runtime: 10.139 ms
```

Pagination et index

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

Dans la base TP existe une table `posts`. C'est une version simplifiée d'une table de forum. Nous voulons afficher très rapidement les messages (*posts*) d'un article : les 10 premiers, puis du 11 au 20, etc. le plus rapidement possible. Nous allons examiner les différentes stratégies possibles.

> Écrire une requête permettant de récupérer les 10 premiers posts de l'article 12. La table a été créée sans index, la requête va être très lente. Utiliser `id_post`, pas le `timestamp` (il servira dans le prochain TP).

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

```
 Limit  (cost=161809.55..161809.57 rows=10 width=261)
        (actual time=2697.841..2697.844 rows=10 loops=1)
   ->  Sort  (cost=161809.55..161812.00 rows=982 width=261)
             (actual time=2697.839..2697.841 rows=10 loops=1)
         Sort Key: id_post
         Sort Method: top-N heapsort  Memory: 29kB
         ->  Seq Scan on posts  (cost=0.00..161788.33 rows=982 width=261)
                                (actual time=2.274..2695.722 rows=964 loops=1)
               Filter: (id_article = 12)
               Rows Removed by Filter: 9999036
 Total runtime: 2697.924 ms
```

> Créer un index permettant d'améliorer cette requête.

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

EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE id_article =12
    ORDER BY  id_post
    LIMIT 10;
                        QUERY PLAN
-------------------------------------------------------------
 Limit  (cost=0.43..1.61 rows=10 width=260)
        (actual time=0.045..0.092 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
             (cost=0.43..115.90 rows=981 width=260)
             (actual time=0.043..0.090 rows=10 loops=1)
         Index Cond: (id_article = 12)
 Total runtime: 0.115 ms
(4 lignes)
```

C'est bien plus rapide : l'index retourne les enregistrements directement triés par `id_article, id_post`. On peut donc trouver le premier enregistrement ayant `id_article = 12`, puis récupérer de ce point tous les enregistrements par `id_post` croissant.

Écrire la même requête permettant de récupérer les 10 posts suivants. Puis du post 901 au 921. Que constate-t-on sur le plan d'exécution ?

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

```
    OFFSET 10;
                            QUERY PLAN
-----------------------------------------------------------
 Limit  (cost=1.61..2.79 rows=10 width=260)
        (actual time=0.070..0.136 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
            (cost=0.43..115.90 rows=981 width=260)
            (actual time=0.034..0.133 rows=20 loops=1)
         Index Cond: (id_article = 12)
 Total runtime: 0.161 ms
(4 lignes)
```

Tout va bien. La requête est à peine plus coûteuse.

À partir du post 900 :

```
EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE id_article = 12
    ORDER BY id_post
    LIMIT 10
    OFFSET 900;
                            QUERY PLAN
-----------------------------------------------------------
 Limit  (cost=106.37..107.55 rows=10 width=260)
        (actual time=5.310..5.362 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
            (cost=0.43..115.90 rows=981 width=260)
            (actual time=0.038..5.288 rows=910 loops=1)
         Index Cond: (id_article = 12)
 Total runtime: 5.397 ms
(4 lignes)
```

Cette requête est 50 fois plus lente. Il serait intéressant de trouver plus rapide.

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

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

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

 id_article | id_post
```

```
------------+---------
         12 | 9245182
(1 ligne)
```

Il suffit donc de récupérer les 10 articles pour lesquels `id_article = 12` et `id_post > 9245182`. (Ces valeurs peuvent être différentes pour votre TP, suivant le contenu de la table posts).

```
EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE id_article = 12
    AND id_post> 9245182
    ORDER BY id_post
    LIMIT 10;
                              QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=0.43..1.65 rows=10 width=260)
        (actual time=0.037..0.056 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
           (cost=0.43..9.41 rows=74 width=260)
           (actual time=0.035..0.050 rows=10 loops=1)
         Index Cond: ((id_article = 12) AND (id_post > 9245182))
 Total runtime: 0.094 ms
(4 lignes)
```

Nous sommes de retour à des temps d'exécution de 0,1 ms.

Attention : dans une base de données réaliste, le critère de sélection peut être plus compliqué. On peut avoir par exemple besoin d'un filtre sur plus d'une colonne. Ou on peut par exemple imaginer (ici ça n'a pas de sens) qu'on veut continuer sur l'article 13 si on arrive à la fin de l'article 12. Dans ce cas, on peut utiliser cette syntaxe (qui n'est pas supportée par tous les SGBD, alors que cela fait partie du standard SQL-92) :

```
EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE (id_article, id_post) > (12, 9245182)
    ORDER BY id_article, id_post
    LIMIT 10;
                              QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=0.44..1.57 rows=10 width=260)
        (actual time=0.039..0.057 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
           (cost=0.44..1135139.20 rows=9990333 width=260)
           (actual time=0.036..0.052 rows=10 loops=1)
```

```
         Index Cond: (ROW(id_article, id_post) > ROW(12, 9245182))
 Total runtime: 0.096 ms
(4 lignes)
```

On compare donc le n-uplet `(id_article, id_post)` à `(12, 9245182)` (l'ordre de tri étant l'ordre logique : on compare les `id_article`, puis si égalité les `id_post`).

Clauses WHERE et pièges

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

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

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

On a un scan complet de la table (*seq scan*). C'est normal : PostgreSQL ne peut pas deviner que `to_char(ts,'YYYYMM')='201302'` veut dire « toutes les dates du mois de février 2013 ». Une fonction est pour lui une boîte noire. Cela équivaut à rechercher tous les mots anglais se traduisant par « Bonjour » dans un dictionnaire Anglais => Français : la seule solution est de lire tout le dictionnaire.

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

NB : on trouve parfois le terme « sargable » dans la littérature pour définir les clauses `WHERE` pouvant être résolues par parcours d'index.

> Réécrire la clause `WHERE`.

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

```
EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE ts >= '2013-02-01'
    AND ts < '2013-03-01';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_posts_ts on posts
        (cost=0.43..2314.75 rows=19641 width=269)
        (actual time=0.054..104.416 rows=18234 loops=1)
   Index Cond: ((ts >= '2013-02-01 00:00:00+01'::timestamp with time zone)
                AND (ts < '2013-03-01 00:00:00+01'::timestamp with time zone))
 Total runtime: 105.712 ms
(3 lignes)
```

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

Construisons cette requête morceau par morceau :

```
SELECT generate_series(
    '2013-01-06 00:00:00',
    '2014-01-01 00:00:00',
    interval '7 days'
);
```

produit la liste de tous les dimanches de 2013 (le premier dimanche est le 6 janvier).

On aurait aussi pu calculer ce premier dimance ainsi, par exemple :

```
select '2013-01-01'::timestamp
       + interval '1 day'
       * (7-extract (dow from timestamp '2013-01-01'))
```

Ensuite :

```
SELECT i debut,
       i + interval '1 day' fin
FROM generate_series(
    '2013-01-06 00:00:00',
    '2014-01-01 00:00:00',
    interval '7 days'
) g(i);
```

produit la liste de tous les intervalles correspondant à ces dimanches.

Il ne nous reste plus qu'à joindre ces deux ensembles :

```
explain analyze
SELECT posts.*
FROM posts
JOIN (
    SELECT i debut,
           i+interval '1 day' fin
    FROM generate_series(
       '2013-01-06 00:00:00',
       '2014-01-01 00:00:00',
       interval '7 days'
    ) g(i)
) interval ON (posts.ts >= interval.debut AND posts.ts <= interval.fin) ;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=11680.98..79960740.50 rows=1111042000 width=268)
              (actual time=0.816..93.620 rows=33705 loops=1)
   ->  Function Scan on generate_series g
           (cost=0.00..10.00 rows=1000 width=8)
           (actual time=0.086..0.110 rows=52 loops=1)
   ->  Bitmap Heap Scan on posts
           (cost=11680.98..68850.31 rows=1111042 width=268)
           (actual time=0.482..1.547 rows=648 loops=52)
         Recheck Cond: ((ts >= g.i) AND (ts <= (g.i + '1 day'::interval)))
         ->  Bitmap Index Scan on idx_posts_ts
               (cost=0.00..11403.22 rows=1111042 width=0)
               (actual time=0.299..0.299 rows=648 loops=52)
             Index Cond: ((ts >= g.i) AND (ts <= (g.i + '1 day'::interval)))
 Total runtime: 96.472 ms
```

Attention : les inéqui-jointures entraînent forcément des *nested loops*. Ici tout va bien parce que la liste des dimanches est raisonnablement courte. De plus, pour avoir un tel plan, il faut que `shared_buffers` et/ou `effective_cache_size` soient suffisamment élevés (pour que le moteur estime qu'il peut passer par les index).

> Pourquoi la requête suivante est-elle lente ?

```
select * from posts where id_article = (
 select round(random()*(select max(id_article) from posts))
)
```

```
EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE id_article = (
        SELECT round(random()*(select max(id_article) FROM posts))
    );
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on posts  (cost=0.58..187730.18 rows=50000 width=269)
                    (actual time=11.143..3713.194 rows=990 loops=1)
   Filter: ((id_article)::double precision = $2)
   Rows Removed by Filter: 9999010
   InitPlan 3 (returns $2)
     ->  Result  (cost=0.56..0.58 rows=1 width=0)
                 (actual time=0.044..0.044 rows=1 loops=1)
           InitPlan 2 (returns $1)
             ->  Result  (cost=0.55..0.56 rows=1 width=0)
                         (actual time=0.037..0.037 rows=1 loops=1)
                   InitPlan 1 (returns $0)
                     ->  Limit  (cost=0.43..0.55 rows=1 width=4)
                                (actual time=0.033..0.033 rows=1 loops=1)
                           ->  Index Only Scan Backward
                                using posts_id_article_id_post
                                on posts posts_1
                                   (cost=0.43..1137275.29 rows=10000000 width=4)
                                   (actual time=0.033..0.033 rows=1 loops=1)
                                 Index Cond: (id_article IS NOT NULL)
                                 Heap Fetches: 1
 Total runtime: 3713.666 ms
(13 lignes)
```

Le seul nœud de cette requête à être lent est le *Seq Scan on posts*. Il prend l'essentiel de la durée de la requête. Pourquoi ? On compare pourtant `id_article` à une constante.

Certes, mais cette constante n'est pas du bon type !

Voici le prototype de le fonction `round` :

```
FUNCTION round (double precision) RETURNS double precision
```

Le problème est visible dans le plan :

```
Filter: ((id_article)::double precision = $2)
```

`(id_article)::double precision` signifie que tous les `id_article` sont convertis en `double precision` pour ensuite être comparés au résultat du `round`. Or une fonction de conversion est une fonction, ce qui rend l'index inutilisable. Il faut que nous forcions PostgreSQL à convertir le résultat du `round` en entier :

```
EXPLAIN ANALYZE
    SELECT *
    FROM posts
    WHERE id_article = (
        SELECT round(random()*(select max(id_article) from posts))
    )::int;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using posts_id_article_id_post on posts
        (cost=1.02..118.83 rows=1001 width=269)
        (actual time=0.094..5.158 rows=970 loops=1)
   Index Cond: (id_article = ($2)::integer)
   InitPlan 3 (returns $2)
     ->  Result  (cost=0.56..0.58 rows=1 width=0)
                 (actual time=0.037..0.037 rows=1 loops=1)
           InitPlan 2 (returns $1)
             ->  Result  (cost=0.55..0.56 rows=1 width=0)
                         (actual time=0.031..0.032 rows=1 loops=1)
                   InitPlan 1 (returns $0)
                     ->  Limit  (cost=0.43..0.55 rows=1 width=4)
                                (actual time=0.029..0.030 rows=1 loops=1)
                           ->  Index Only Scan Backward
                               using posts_id_article_id_post on posts posts_1
                                   (cost=0.43..1137275.29 rows=10000000 width=4)
                                   (actual time=0.027..0.027 rows=1 loops=1)
                                 Index Cond: (id_article IS NOT NULL)
                                 Heap Fetches: 1
 Total runtime: 5.293 ms
(12 lignes)
```

3 COMPRENDRE EXPLAIN

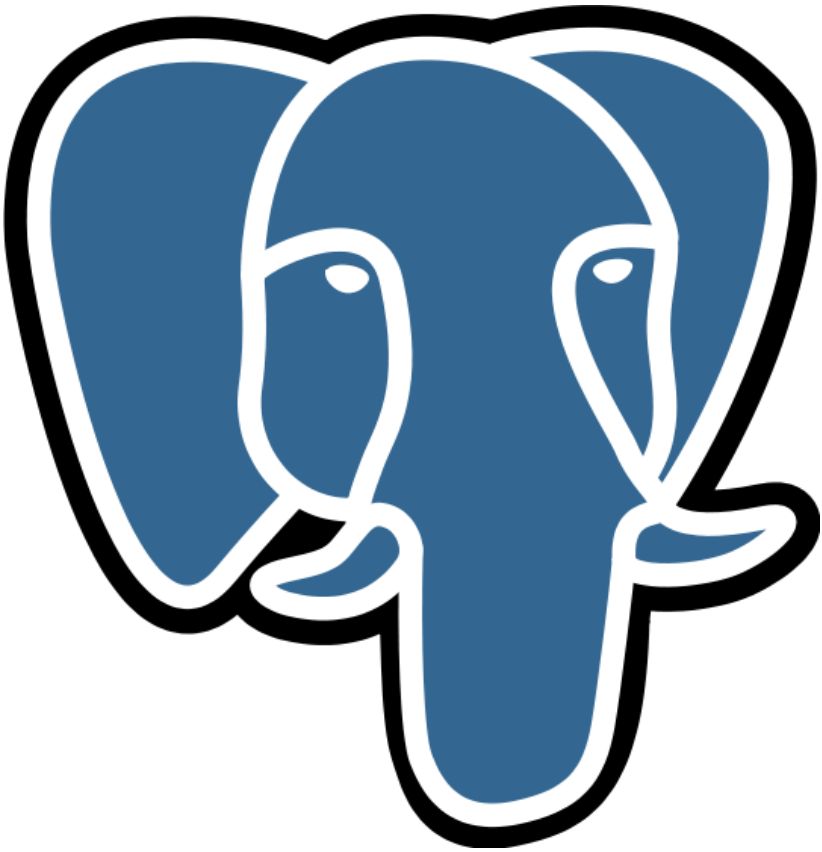


FIGURE 1: POSTGRESQL

3.1 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.

17.12

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.

3.1.1 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.

3.2 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.

3.2.1 NIVEAU SYSTÈME

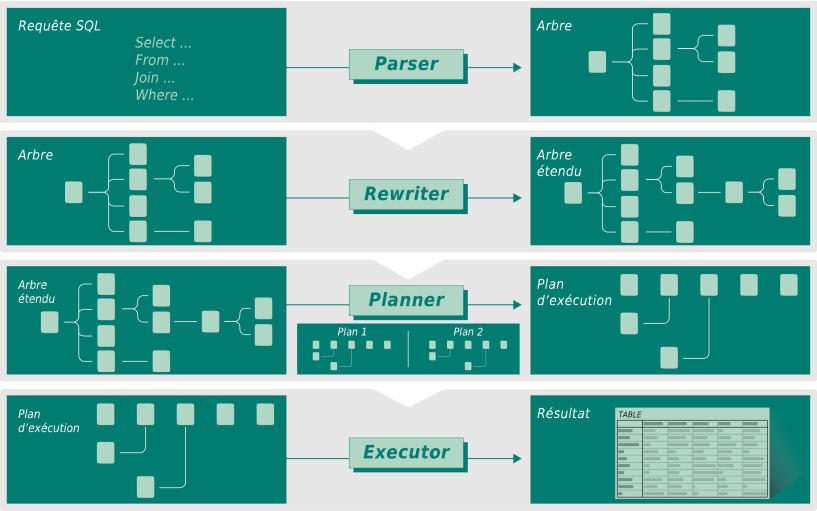
- Le client envoie une requête au serveur de bases de données
- Le serveur l'exécute
- Puis il renvoie le résultat au client

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.

3.2.2 NIVEAU SGBD

TRAITEMENT D'UNE REQUÊTE SQL



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
```

```
! 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
```

3.2.3 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.

3.3 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.

3.3.1 REQUÊTE ÉTUDIÉE

Cette requête d'exemple :

```
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 :

```
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);
```

17.12

```
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', 'Joséphа', '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);
```

3.3.2 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.

3.4 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.

3.4.1 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.

3.4.2 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).

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.5 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.

3.5.1 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.

3.6 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).

3.6.1 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.

3.6.2 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;
```

3.6.3 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.

3.6.4 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.

3.6.5 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 `<`.

3.6.6 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.

3.6.7 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;
```

`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.

3.6.8 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.

3.6.9 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.

3.6.10 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;
```

3.7 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

3.7.1 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.

3.7.2 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.

3.7.3 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)
```

3.7.4 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.

3.7.5 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[17] .

3.8 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

[17] http://docs.postgresql.fr/current/planner-stats-details.html

- chaque nœud consomme au fur et à mesure les données produites par les nœuds parents

---

3.8.1 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.

3.8.2 PARCOURS D'INDEX

- Index Scan
- Index Only Scan
- Bitmap Index Scan
- Et leurs versions parallélisées

17.12

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.

3.8.3 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.

3.8.4 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)
```

3.8.5 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.

3.9 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.

3.9.1 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`.

3.9.2 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)
```

3.9.3 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.

3.9.4 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)
```

3.9.5 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.

3.9.6 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[18] .

3.9.7 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 ?

[18] 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 :

```
SELECT * FROM table WHERE date_trunc('month', date_debut) = 12
```

ou encore plus fréquemment rencontré :

```
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 :

```
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 :

```
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[19]

3.9.8 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
```

[19] 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);
```

3.10 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.

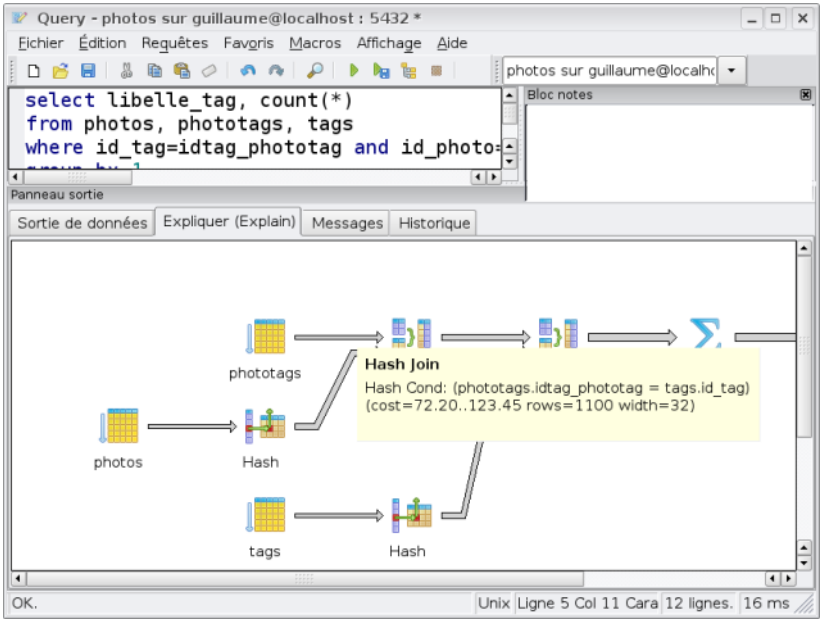
3.10.1 PGADMIN3

- 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.

3.10.2 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.

3.10.3 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.

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[20] .

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.

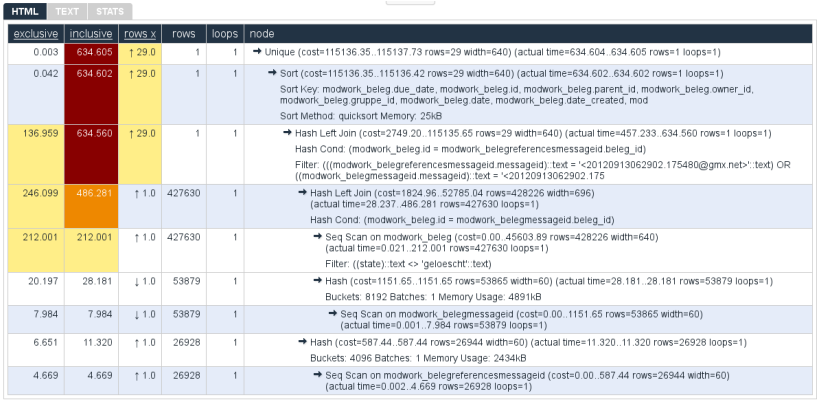
Le code couleur est simple :

- 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[21]
- la partie site web[22]

3.10.4 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 :

[20] http://explain.depesz.com
[21] https://github.com/depesz/Pg--Explain
[22] 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.

3.10.5 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

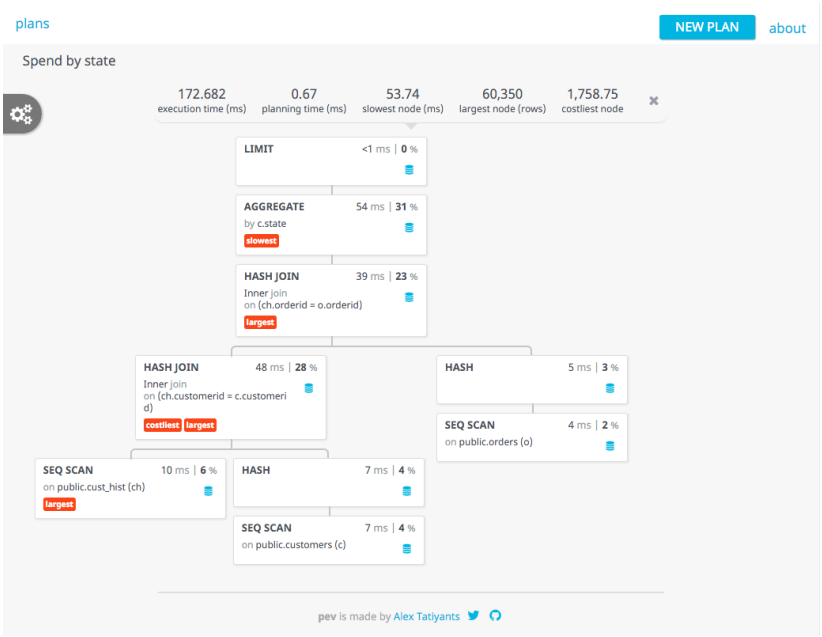
PEV est un outil librement téléchargeable sur ce dépôt github[23] . Il offre un affichage graphique du plan d'exécution et indique le nœud le plus coûteux, le plus long, le plus volumineux, etc.

Il est utilisable sur internet[24] mais aussi installable en local.

---

[23] https://github.com/AlexTatiyants/pev
[24] http://tatiyants.com/pev/#/plans

3.10.6 PEV - COPIE D'ÉCRAN



3.10.7 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`.

3.10.8 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[25] .

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';
```

[25] 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.

3.11 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
- ...

3.11.1 QUESTIONS

N'hésitez pas, c'est le moment !

3.12 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.

3.12.1 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

17.12

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.

3.12.2 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`.

3.12.3 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.

3.12.4 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.

3.12.5 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[26] .

3.12.6 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
```

[26] 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)
```

3.12.7 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.

3.12.8 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.

3.12.9 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.

3.12.10 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.

3.12.11 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.

3.12.12 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[27] 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[28] pour approfondir ce point).

[27] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f
[28] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116279

3.12.13 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.

3.12.14 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)
```

3.12.15 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

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.

3.12.16 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)
```

3.12.17 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.

3.12.18 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.

3.12.19 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)
```

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.

3.12.20 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.

3.12.21 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.

3.12.22 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)
```

3.12.23 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)
```

3.13 TRAVAUX PRATIQUES

3.13.1 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

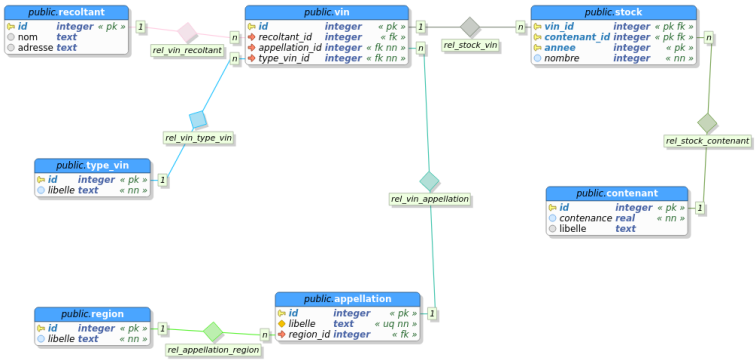


FIGURE 2: SCHÉMA DE LA BASE CAVE

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;
```

- 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 ?

3.13.2 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 ?

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)
```

17.12

```
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'
);
```

```
                                                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.

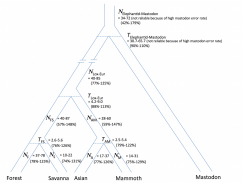
On peut aussi créer une colonne supplémentaire maintenue par un trigger, plutôt qu'un index : cela sera moins coûteux à maintenir, et permettra d'avoir la même statistique.

3.13.3 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)

4 TECHNIQUES D'INDEXATION



Droits sur l'image : CC BY 2.5 - Rohland N, Reich D, Mallick S, Meyer M, Green RE, et al.

4.1 INTRODUCTION

- Qu'est-ce qu'un index ?
- Comment indexer une base ?
- Les différents types d'index

Postgres, comme tout SGBDR, peut exploiter différentes structures de données pour renvoyer les résultats comme vu dans le module sur explain : l'indexation fournit un moyen de créer des structures de données plus adaptées pour répondre à certaines requêtes. Dans le cadre d'un développement, il est nécessaire d'avoir au minimum une connaissance basique de ces objets et concepts, de leurs implications sur les performances de la base ainsi que leurs caractéristiques.

En effet, l'équipe de développement possède généralement une vision globale des fonctionnalités demandées par l'application, et est donc mieux à même de structurer le développement et les évolutions de l'application en fonction des contraintes de performance.

Cette connaissance n'est généralement que peu accessible à un DBA d'exploitation, il est donc primordial que le développement soit conscient des enjeux de l'indexation, et des techniques permettant de réaliser celle-ci.

4.1.1 AU MENU

- Anatomie d'un index
- Les index « simples »

17.12

- Méthodologie
- Indexation avancée
- Outillage

Ce module débute par une présentation un peu théorique, permettant de comprendre ce qu'est un index, fonctionnellement et techniquement, et ce qu'il apporte.

Nous verrons ensuite les types d'index les plus fréquemment utilisés, ainsi que la méthodologie à suivre pour identifier quels index seraient bénéfiques pour la charge de travail subie par la base.

Les mécanismes d'indexation plus avancés proposés par PostgreSQL sont détaillés dans une partie suivante.

Enfin, nous aborderons l'utilisation d'outils facilitant le choix des index à mettre en place.

4.1.2 OBJECTIFS

- Comprendre ce qu'est un index
- Maîtriser le processus de création d'index
- Connaître les différents types d'index et leurs cas d'usages

---

4.2 FONCTIONNEMENT D'UN INDEX

- Analogie : index dans une publication scientifique
  - Structure séparée, associant des clés (termes) à des localisations (pages)
  - Même principe pour un index dans un SGBD
- Structure de données spécialisée, plusieurs types
- Existe en dehors de la table

Pour comprendre ce qu'est un index, l'index dans une publication scientifique au format papier offre une analogie simple.

Lorsque l'on recherche un terme particulier dans un ouvrage, il est possible de parcourir l'intégralité de l'ouvrage pour chercher les termes qui nous intéressent. Ceci prend énormément de temps, selon la taille de l'ouvrage. Ce type de recherche trouve son analogie sous la forme du parcours complet d'une table (`SeqScan`).

Une deuxième méthode pour localiser ces différents termes consiste, si l'ouvrage en dispose, à utiliser l'index de celui-ci. Un tel index associe un terme à un ensemble de pages

où celui-ci est présent. Ainsi, pour trouver le terme recherché, il est uniquement nécessaire de parcourir l'index (qui ne dépasse généralement pas quelques pages) à la recherche du terme, puis d'aller visiter les pages listées dans l'index pour extraire les informations nécessaires.

Dans un SGBD, le fonctionnement d'un index est très similaire à celui décrit ici. En effet, comme dans une publication, l'index est une structure de données à part, qui n'est pas strictement nécessaire à l'exploitation des informations, et qui est utilisée pour faciliter la recherche dans l'ensemble de données. Cette structure de données possède un coût de maintenance, dans les deux cas : toute modification des données peut entraîner des modifications afin de maintenir l'index à jour.

4.2.1 UN INDEX N'EST PAS MAGIQUE…

- Un index ne résout pas tout
- Importance de la conception du schéma de données
- Importance de l'écriture de requêtes SQL correctes

Bien souvent, la création d'index est vue comme le remède à tous les maux de performance subis par une application. Il ne faut pas perdre de vue que les facteurs principaux affectant les performances vont être liés à la conception du schéma de données, et à l'écriture des requêtes SQL.

Pour prendre un exemple caricatural, un schéma `EAV` (*Entity-Attribute-Value*, ou *entité-clé-valeur*) ne pourra jamais être performant, de part sa conception. Bien sûr, dans certains cas, une méthodologie pertinente d'indexation permettra d'améliorer un peu les performances, mais le problème réside là dans la conception même du schéma. Il est donc important dans cette phase de considérer la manière dont le modèle va influer sur les méthodes d'accès aux données, et les implications sur les performances.

De même, l'écriture des requêtes elles-mêmes conditionnera en grande partie les performances observées sur l'application. Par exemple, la mauvaise pratique (souvent mise en œuvre accidentellement via un `ORM`) dite du `N+1` ne pourra être corrigée par une indexation correcte : celle-ci consiste à récupérer une collection d'enregistrement (une requête) puis d'effectuer une requête pour chaque enregistrement afin de récupérer les enregistrements liés (N requêtes).
Dans ce type de cas, une jointure est bien plus performante. Ce type de comportement doit encore une fois être connu de l'équipe de développement, car il est plutôt difficile à détecter par une équipe d'exploitation.

17.12

De manière générale, avant d'envisager la création d'index supplémentaires, il convient de s'interroger sur les possibilités de réécriture des requêtes, voire du schéma.

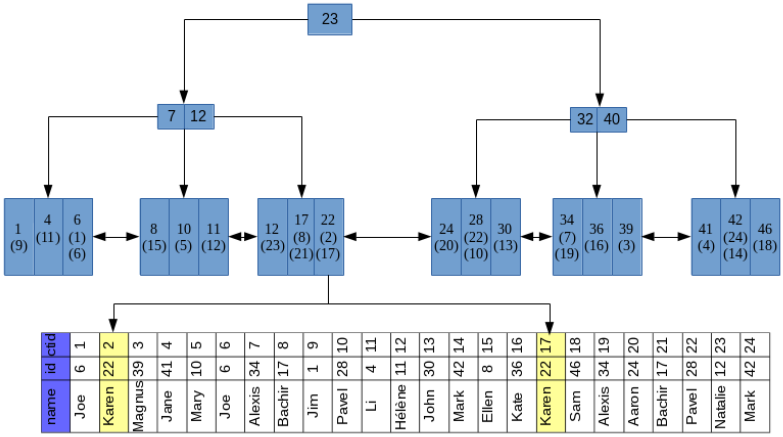
4.2.2 INDEX BTREE

- Type d'index le plus courant
- Mais aussi le plus simple
- Utilisé pour les contraintes d'unicité
- Supporte les opérateurs suivants : `<`, `<=`, `=`, `>=`, `>`
- Supporte le tri
- Ne peut pas indexer des colonnes de plus de 2.6 Ko

L'index `btree` est le plus simple conceptuellement parlant. Sans entrer dans les détails, un index `btree` est par définition équilibré : ainsi, quelle que soit la valeur recherchée, le coût est le même lors du parcours d'index. Ceci ne veut pas dire que toute requête impliquant l'index mettra le même temps ! En effet, si chaque clé n'est présente qu'une fois dans l'index, celle-ci peut être associée à une multitude de valeurs, qui devront alors être cherchées dans la table.

L'algorithme utilisé par PostgreSQL pour ce type d'index suppose que chaque page peut contenir au moins trois valeurs. Par conséquent, chaque valeur ne peut excéder un peu moins d'1/3 de bloc, soit environ 2.6 Ko. La valeur en question correspond donc à la totalité des données de toutes les colonnes de l'index pour une seule ligne. Si l'on tente de créer ou maintenir un index sur une table ne satisfaisant pas ces prérequis, une erreur sera reportée, et la création de l'index (ou l'insertion/mise à jour de la ligne) échouera. Si un index de type `btree` est tout de même nécessaire sur les colonnes en question, il est possible de créer un index fonctionnel sur une fonction de hashage des valeurs. Dans un tel cas, seul l'opérateur `=` pourra bénéficier d'un parcours d'index.

4.2.3 CONCRÈTEMENT…



Ce schéma présente une vue simplifiée d'un index `btree`. Chaque nœud présente un certain nombre de valeurs et des pointeurs vers les feuilles suivantes. C'est tout simplement une généralisation d'un arbre binaire.

La table indexée présente ici deux colonnes, sans contraintes d'unicité : `id` et `name`. Le haut du schéma représente un index sur la colonne `id`. Un tel index peut être créé par l'instruction :

```
CREATE INDEX mon_index ON ma_table (id) ;
```

Il est à noter que pour chaque ligne, PostgreSQL assigne un identifiant unique à cette ligne, appelé ctid, correspondant à sa position physique dans la table. Le fonctionnement présenté ici est simplifié : le ctid est en réalité un couple (numéro de bloc, position dans le bloc) abrégé ici en un unique entier.

Pour rechercher l'ensemble des lignes pour lesquelles la condition `WHERE id = 22` est vraie, il existe deux solutions : le parcours séquentiel classique (`SeqScan`) consiste à lire l'intégralité de la table, et tester la condition pour chaque ligne avant de la renvoyer. Selon le volume de données, et la sélectivité de la clause, cela peut représenter un travail considérable. La deuxième solution consiste à parcourir l'index. Le fonctionnement est ici un

tout petit peu plus compliqué :

- la racine ne possède qu'une valeur. En comparant la valeur recherchée (22) à la valeur de la racine, on choisit le nœud à explorer. Ici, 22 est plus petit que 23 : on explore donc le nœud de gauche ;
- ce nœud possède deux valeurs : 7 et 12. On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud : pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre. Ici, 22 est plus grand que 12, on explore donc de nouveau le nœud de droite ;
- un arbre `B-Tree` peut bien évidemment avoir une profondeur plus grande, auquel cas l'étape précédente est répétée ;
- une fois arrivé à une feuille, il suffit de parcourir celle-ci pour récupérer l'ensemble des `ctid`, ou positions physiques, des lignes correspondants au critère. Ici, l'index nous indique qu'à la valeur 22 correspondent deux lignes, situées aux positions 2 et 17 dans la table. Pour renvoyer le résultat, il suffit donc d'aller lire ces données depuis la table elle même, évitant ainsi de parcourir toute la table.

Cela implique deux choses fondamentales :

- si la valeur recherchée représente une assez petite fraction des lignes totales, le nombre d'accès disques sera fortement réduits. En revanche, au lieu d'effectuer des accès séquentiels (pour lesquels les disques durs sont relativement performants), il faudra effectuer des accès aléatoires, en *sautant* d'une position sur le disque à une autre ;
- dans tous les cas, il faudra parcourir deux structures de données : l'index, et la table elle même. Depuis PostgreSQL 9.2, dans certains cas précis, il est possible de ne parcourir que l'index. Nous reviendrons sur cette fonctionnalité ultérieurement.

Supposons désormais que nous souhaitions exécuter une requête du type :

```
SELECT id FROM ma_table ORDER BY id ;
```

L'index peut nous aider à répondre à cette requête. En effet, toutes les feuilles sont liées entre elles, et permettent ainsi un parcours ordonné. Il nous suffit donc de localiser la première feuille (la plus à gauche), et pour chaque clé, récupérer les lignes correspondantes. Une fois les clés de la feuille traitées, il suffit de suivre le pointeur vers la feuille suivante et de recommencer.

L'alternative consisterait à parcourir l'ensemble de la table, et trier toutes les lignes afin de les obtenir dans le bon ordre. Un tel tri peu être très coûteux, en mémoire comme en temps CPU. D'ailleurs, de tels tris débordent très souvent sur disque (via des fichiers temporaires) afin de ne pas garder l'intégralité des données en mémoire.

Pour les requêtes utilisant des opérateurs d'inégalité, on voit bien comment l'index peut

là aussi être utilisé. Par exemple, pour la requête suivante :

```
SELECT * FROM ma_table WHERE id <= 10 AND id >= 4 ;
```

Il suffit d'utiliser la propriété de tri de l'index pour parcourir les feuilles, en partant de la borne inférieure, jusqu'à la borne supérieure.

4.2.4 IMPACT SUR LES PERFORMANCES

- L'index n'est pas gratuit !
- Espace disque nécessaire
- Maintenance à chaque opération DML
- Mesurer la pertinence de l'index

Même si un index peut améliorer grandement les performances d'une ou plusieurs requêtes, il convient de se poser la question de la pertinence de celui-ci à chaque création. En effet, à chaque modification des données de la table, l'index devra être mis à jour pour refléter les nouvelles données. Selon les cas, par exemple lorsqu'un rééquilibrage de l'arbre est nécessaire, cette opération peut être très coûteuse. De plus, cet index étant une structure de données séparée de la table, celle-ci occupe une place disque non négligeable. Pour illustrer ce propos, voici un exemple simple.

D'abord, on crée une table, possédant 100000 lignes :

```
formation=# CREATE TABLE ma_table AS SELECT generate_series(1, 100000) c1 ;
SELECT 100000
```

Mesurons le temps nécessaire pour ajouter 1000 lignes à cette table :

```
formation=# \timing
Chronométrage activé.
formation=# INSERT INTO ma_table (c1) (SELECT generate_series(1, 100000));
INSERT 0 100000
Temps : 80,197 ms
```

Maintenant, mesurons ce même temps si la table dispose d'un index :

```
formation=# CREATE INDEX on ma_table (c1);
CREATE INDEX
Temps : 229,078 ms
formation=# INSERT INTO ma_table (c1) (SELECT generate_series(1, 100000));
INSERT 0 100000
Temps : 269,337 ms
```

17.12

On voit ici que la pénalité est loin d'être négligeable : l'insertion est une fois et demie plus lente. Bien entendu, ce facteur est purement indicatif, et variera énormément selon les cas.

En revanche, si l'on compare les temps d'exécution avec et sans l'index, on constate une amélioration très nette de ceux-ci :

Sans l'index :

```
formation=# SELECT * FROM ma_table WHERE c1 = 10;
 c1
----
 10
 10
 10
 10
(4 lignes)
Temps : 8,665 ms
```

Avec l'index :

```
formation=# SELECT * FROM ma_table WHERE c1 = 10;
 c1
----
 10
 10
 10
 10
(4 lignes)

Temps : 0,309 ms
```

4.2.5 INDEX MULTICOLONNES

- Possibilité d'indexer plusieurs colonnes :

  ```
  CREATE INDEX ON ma_table (id, name) ;
  ```

- L'ordre des colonnes est **primordial**
  - permet de répondre aux requêtes sur les premières colonnes de l'index
  - pour les autres, PostreSQL lira tout l'index ou ignorera l'index

Il est possible de créer un index sur plusieurs colonnes. Il faut néanmoins être conscient des requêtes supportées par un tel index. Admettons que l'on crée la table et l'index suivants :

```
CREATE TABLE t1(c1 int,c2 int,c3 int);

INSERT INTO t1(c1, c2, c3)
SELECT i, i/2, i*2 FROM generate_series (1,1e6::bigint) i ;

CREATE INDEX ON t1 (c1, c2, c3) ;

ANALYZE t1 ;
```

L'index est optimal pour répondre aux requêtes portant sur les premières colonnes de l'index :

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE c1 = 1000 ;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=2.31..2.32 rows=1 width=0) (...)
   Buffers: shared hit=4
   ->  Index Scan using t1_c1_c2_c3_idx on t1  (...)
         Index Cond: (c1 = 1000)
         Buffers: shared hit=4
 Total runtime: 0.102 ms
```

Mais si les premières colonnes de l'index ne sont pas spécifiées, alors l'index devra être parcouru en grande partie. Ce peut être toujours plus intéressant que parcourir toute la table, surtout si l'index contient toutes les données du SELECT. Mais pour limiter les aller-retours entre index et table, et en fonction de nombreux paramètres, comme les statistiques et les valeurs relatives de `seq_page_cost` et `random_page_cost`, PostgreSQL peut décider d'ignorer l'index et de parcourir directement la table :

```
--- Paramétrage par défaut
SET random_page_cost=4 ; SET seq_page_cost=1 ;

EXPLAIN (ANALYZE,BUFFERS) SELECT COUNT(*) FROM t1 WHERE c3 = 1000 ;

                                          QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=17906.00..17906.01 rows=1 width=0) (...)
   Buffers: shared hit=100 read=5306
   ->  Seq Scan on t1  (cost=0.00..17906.00 rows=1 width=0) (...)
         Filter: (c3 = 1000)
         Buffers: shared hit=100 read=5306
 Total runtime: 87.321 ms

-- Valeurs pour un SSD
SET random_page_cost=0.1 ; SET seq_page_cost=0.1 ;
```

```
EXPLAIN (ANALYZE,BUFFERS) SELECT COUNT(*) FROM t1 WHERE c3 = 1000 ;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=11354.30..11354.31 rows=1 width=0) (...)
   Buffers: shared hit=3835
   ->  Index Scan using t1_c1_c2_c3_idx on t1 (...)
         Index Cond: (c3 = 1000)
         Buffers: shared hit=3835
 Total runtime: 17.522 ms
```

Noter que tout l'index a été lu dans la dernière requête, alors que celle sur la colonne de tête `c1` se contentait de lire 4 blocs.

Concernant les *range scans* (requêtes impliquant des opérateurs d'inégalité, tels que `<`, `<=`, `>=`, `>`), celles-ci pourront être satisfaites par l'index de manière quasi-optimale si les opérateurs d'inégalité sont appliqués sur la dernière colonne requêtée, et de manière sub-optimale s'ils portent sur les premières colonnes.

Cet index pourra être utilisé pour répondre aux requêtes suivantes de manière optimale :

```
SELECT * FROM t1 WHERE c1 = 2 ;
SELECT * FROM t1 WHERE c1 = 2 AND c2 = 3 AND c3 = 8 ;
SELECT * FROM t1 WHERE c1 = 10 AND c2 <= 4 ;
```

Il pourra aussi être utilisé, mais de manière bien moins efficace, pour les requêtes suivantes, qui bénéficieraient d'un index sur un ordre alternatif des colonnes :

```
SELECT * FROM t1 WHERE c1 = 20000 AND c2 >= 10000 AND c3 = 4000 ;
SELECT * FROM t1 WHERE c1 < 10000 AND c2 = 10000 ;
```

Le plan de cette dernière requête est :

```
 Index Scan using t1_c1_c2_c3_idx on t1  (...)
   Index Cond: ((c1 < 10000) AND (c2 = 10000))
   Buffers: shared hit=41
 Total runtime: 0.360 ms
```

Les index multicolonnes peuvent aussi être utilisés pour le tri comme dans les exemples suivants. Ici le cas est optimal puisque l'index contient toutes les données nécessaires :

```
SELECT * FROM t1 ORDER BY c1 ;
SELECT * FROM t1 ORDER BY c1, c2 ;
SELECT * FROM t1 ORDER BY c1, c2, c3 ;
```

Le plan de cette dernière requête est :

```
 Index Scan using t1_c1_c2_c3_idx on t1 (cost=0.00..33380.23 rows=1000000 ...)
   Buffers: shared read=9240
 Total runtime: 198.398 ms
```

Il est donc nécessaire d'avoir une très bonne connaissance de l'application pour déterminer comment créer des index multi-colonnes pertinents pour un nombre maximum de requêtes.

4.3 MÉTHODOLOGIE DE CRÉATION D'INDEX

- On indexe pour une requête, ou idéalement une collection de requête
- On n'indexe pas « une table »

La première chose à garder en tête est que l'on indexe pas le schéma de données, c'est à dire les tables, mais en fonction de la charge de travail supportée par la base, c'est à dire les requêtes. En effet, comme nous l'avons vu précédemment, tout index superflu à un coût global pour la base de données, notamment pour les opérations DML.

4.3.1 L'INDEX ? QUEL INDEX ?

- Identifier les requêtes nécessitant un index
- Créer les index permettant de répondre à ces requêtes
- Valider le fonctionnement, en rejouant la requête avec :

```
EXPLAIN (ANALYZE, BUFFERS)
```

La méthodologie elle même est assez simple. Selon le principe qu'un index sert une (ou des) requêtes, la première chose à faire consiste à identifier celles-ci. L'équipe de développement est dans une position idéale pour réaliser ce travail : elle seule peut connaître le fonctionnement global de l'application, et donc les colonnes qui vont être utilisées, ensemble ou non, comme cible de filtres ou de tris. Au delà de la connaissance de l'application, il est possible d'utiliser des outils tels que `pgbadger`, `pg_stat_statements` et `PoWA` pour identifier les requêtes particulièrement consommatrices, et qui pourraient donc potentiellement nécessiter un index. Ces outils seront présentés plus loin dans cette formation.

Une fois les requêtes identifiées, il est nécessaire de trouver les index permettant d'améliorer celles-ci. Ils peuvent être utilisés pour les opérations de filtrage (clause `WHERE`), de tri (clauses `ORDER BY`, `GROUP BY`) ou de jointures.
Idéalement, l'étude portera sur l'ensemble des requêtes, afin notamment de pouvoir décider d'index multi-colonnes pertinents pour le plus grand nombre de requêtes, et éviter ainsi de créer des index redondants.

4.3.2 INDEX ET CLÉS ÉTRANGÈRES

- Indexation des colonnes faisant référence à une autre
- Performances des DML
- Performances des jointures

De manière générale, l'ensemble des colonnes étant la source d'une clé étrangère devraient être indéxées, et ce pour deux raisons.

La première concerne les jointures. Généralement, lorsque deux tables sont liées par des clés étrangères, il existe au moins certaines requêtes dans l'application joignant ces tables. La colonne « cible » de la clé étrangère est nécessairement indexée, c'est un prérequis dû à la contrainte unique nécessaire à celle-ci. Il est donc possible de la parcourir de manière triée.

La colonne source, elle devrait être indexée elle aussi : en effet, il est alors possible de la parcourir de manière ordonnée, et donc de réaliser la jointure selon l'algorithme `MERGE JOIN` (comme vu lors du module concernant `EXPLAIN`), et donc d'être beaucoup plus rapide. Un tel index accélèrera de la même manière les `NESTED LOOP`, en permettant de parcourir l'index une fois par ligne de la relation externe au lieu de parcourir l'intégralité de la table.

De la même manière, pour les DML sur la table cible, cet index sera d'une grande aide : pour chaque ligne modifiée ou supprimée, il convient de vérifier, soit pour interdire soit pour « cascader » la modification, la présence de lignes faisant référence à celle touchée.

S'il n'y a qu'une règle à suivre aveuglément ou presque, c'est bien celle-ci : les colonnes faisant partie d'une clé étrangère doivent être indéxées !

4.3.3 INDEX NON UTILISÉS

- Pas le bon type (`CAST`)
- Pas les bons opérateurs
- Sélectivité trop faible
- Index redondants

Parfois, un index peut exister sur une colonne, mais celui-ci n'est pas utilisé par la requête. Dans des cas très particuliers, et donc rares, il s'agit d'une limitation de l'optimiseur de

PostgreSQL. Cependant, généralement, celui-ci a une très bonne raison de ne pas utiliser l'index.

Il faut d'abord s'assurer que la requête est écrite correctement. Par exemple, s'assurer que la valeur passée en paramètre est du type correspondant à celui de la donnée indexée. Cela peut paraître contre-intuitif, mais certains transtypages ne permettent pas de garantir que les résultats d'un opérateur (par exemple l'égalité) seront les mêmes si les arguments sont convertis dans un type ou dans l'autre.

Par exemple, il en va ainsi de la comparaison entre entiers (`int`) et décimaux (`numeric`) :

```
-- Ceci est faux
SELECT 1::int::numeric = 1.2::numeric;

-- Ceci est vrai
SELECT 1::int = 1.2::numeric::int;
```

Ainsi, afin de garantir la justesse des résultats, certaines opérations qui paraissent *logiques* du point de vue de l'utilisateur ne permettent pas d'utiliser un index :

```
sql=# EXPLAIN SELECT * FROM clients WHERE client_id = 3::numeric;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on clients  (cost=0.00..2525.00 rows=500 width=51)
   Filter: ((client_id)::numeric = 3::numeric)
(2 lignes)

sql=# EXPLAIN SELECT * FROM clients WHERE client_id = 3;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using clients_pkey on clients  (cost=0.29..8.31 rows=1 width=51)
   Index Cond: (client_id = 3)
(2 lignes)
```

Ces exemples sont exagérément évidents, mais il peut être plus compliqué de trouver la cause du problème lorsqu'il s'agit d'un problème de transtypage lors d'une jointure entre des champs de types différents.

Il en est de même lors d'opérations pour lesquelles une logique existe, mais dont l'optimiseur n'a pas connaissance. Par exemple, concernant les fonctions de manipulation sur les dates : pour un utilisateur, chercher les commandes dont la date tronquée au mois correspond au 1er janvier est équivalent aux commandes dont la date est entre le 1er et le 31 janvier. Pour l'optimiseur, il en va tout à fait autrement :

```
sql=# EXPLAIN ANALYZE SELECT * FROM commandes
      WHERE date_commande BETWEEN '2015-01-01' AND '2015-01-31';
                          QUERY PLAN
```

```
-----------------------------------------------------------
 Index Scan using commandes_date_commande_idx on commandes
                                    (cost=0.42..118.82 rows=5554 width=51)
                               (actual time=0.019..0.915 rows=4882 loops=1)
   Index Cond: ((date_commande >= '2015-01-01'::date)
           AND (date_commande <= '2015-01-31'::date))
 Planning time: 0.074 ms
 Execution time: 1.098 ms
(4 lignes)

sql=# EXPLAIN ANALYZE SELECT * FROM commandes
      WHERE date_trunc('month', date_commande) = '2015-01-01';
;
                              QUERY PLAN
-----------------------------------------------------------------------
Gather  (cost=1000.00..8160.96 rows=5000 width=51)
        (actual time=17.282..192.131 rows=4882 loops=1)
 Workers Planned: 3
 Workers Launched: 3
 ->  Parallel Seq Scan on commandes (cost=0.00..6660.96 rows=1613 width=51)
                           (actual time=17.338..177.896 rows=1220 loops=4)
     Filter: (date_trunc('month'::text,
                                  (date_commande)::timestamp with time zone)
                        = '2015-01-01 00:00:00+01'::timestamp with time zone)
     Rows Removed by Filter: 248780
 Planning time: 0.215 ms
 Execution time: 196.930 ms
(8 lignes)
```

De même, il faut bien garder à l'esprit qu'un index ne sert qu'à certains opérateurs. Ceci est généralement indiqué correctement dans la documentation. Pour plus de détails à ce sujet, se référer à la section correspondant aux classes d'opérateurs[29] . Si un opérateur non supporté est utilisé, l'index ne servira à rien :

```
sql=# CREATE INDEX ON fournisseurs (commentaire);
CREATE INDEX

sql=# EXPLAIN ANALYZE SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';
                               QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on fournisseurs  (cost=0.00..225.00 rows=1 width=45)
                           (actual time=0.045..1.477 rows=47 loops=1)
   Filter: (commentaire ~~ 'ipsum%'::text)
   Rows Removed by Filter: 9953
 Planning time: 0.085 ms
```

[29] http://www.postgresql.org/docs/current/static/indexes-opclass.html

```
 Execution time: 1.509 ms
(5 lignes)
```

Nous verrons qu'il existe d'autre classes d'opérateurs, permettant d'indexer correctement la requête précédente.

Comme vu précédemment, le parcours d'un index implique à la fois des lectures sur l'index, et des lectures sur la table. Au contraire d'une lecture séquentielle de la table, l'accès aux données via l'index nécessite des lectures aléatoires. Ainsi, si l'optimiseur estime que la requête nécessitera de parcourir une grande partie de la table, il peut décider de ne pas utiliser l'index : l'utilisation de celui-ci serait alors trop coûteux.

4.4 INDEXATION AVANCÉE

De nombreuses possibilités d'indexation avancée :

- Index multi-colonnes
- Index fonctionnels
- Index partiels
- `Covering indexes`
- Classes d'opérateurs
- `GiN`
- `GiST`
- `BRIN`
- `Hash`

PostgreSQL fournit de nombreux types d'index, afin de répondre à des problématiques de recherches complexes.

L'index `btree` étudié précédemment est l'index le plus fréquemment utilisé, car il a de nombreux avantages :

- Performances se dégradant peu avec la taille de l'arbre : les temps de recherche sont en `O(log(n))`, c'est à dire qu'ils varient en fonction du logarithme du nombre d'enregistrements contenus dans l'index. Plus le nombre d'enregistrements est élevé, plus la variation est faible
- Ils permettent une excellente concurrence d'accès : on peut facilement avoir plusieurs processus en train d'insérer simultanément dans un index `btree`, avec très peu de contention entre ces processus

Toutefois ils ne permettent de répondre qu'à des questions très simples : il faut qu'elles ne portent que sur la colonne indexée, et uniquement sur des opérateurs courants (égal-

ité, comparaison). Cela couvre le gros des cas, mais connaître les autres possibilités du moteur vous permettra d'accélérer des requêtes plus complexes, ou d'indexer des types de données inhabituels.

4.4.1 INDEX PARTIELS

- N'indexe qu'une partie des données :

  ```
  CREATE INDEX ON table (colonne) WHERE condition;
  ```

- Ne sert que si la clause exacte est respectée !
- Intérêt : index beaucoup plus petit !

Un index partiel est un index ne couvrant qu'une partie des enregistrements. Ainsi, l'index est beaucoup plus petit. En contrepartie, il ne pourra être utilisé que si sa condition est définie dans la requête.

Pour prendre un exemple simple, imaginons un système de "queue", dans lequel des évènements sont entrés, et qui disposent d'une colonne "traité" indiquant si oui ou non l'évènement a été traité. Dans le fonctionnement normal de l'application, la plupart des requêtes ne s'intéressent qu'aux évènements traités :

```
sql=# CREATE TABLE evenements (
    id int primary key,
    traite bool,
    type text,
    payload text
);
CREATE TABLE

sql=# INSERT INTO evenements (id, traite, type) (
    SELECT i, true,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
    FROM generate_series(1, 10000) as i);
INSERT 0 10000

sql=# INSERT INTO evenements (id, traite, type) (
    SELECT i, false,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
```

```
        END
    FROM generate_series(10001, 10010) as i);
INSERT 0 10

sql=# \d evenements
                Table « public.evenements »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 | not null  |
 traite  | boolean |                 |           |
 type    | text    |                 |           |
 payload | text    |                 |           |
Index :
    "evenements_pkey" PRIMARY KEY, btree (id)
```

Typiquement, différents applicatifs vont être intéressés par des événèments d'un certain type, mais les évènements déjà traités ne sont quasiment jamais accédés, du moins via leur état (`traite IS true`).

Ainsi, on peut souhaiter indexer le type d'évènement, mais uniquement pour les évènements non traités :

```
ql=# CREATE INDEX index_partiel on evenements (type) WHERE NOT traite;
CREATE INDEX
```

Si on recherche les évènements dont le type est `FACTURATION`, sans plus de précision, l'index ne peut évidemment pas être utilisé :

```
sql=# EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION';
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on evenements  (cost=0.00..183.12 rows=3336 width=47)
   Filter: (type = 'FACTURATION'::text)
(2 lignes)
```

En revanche, si la condition sur l'état de l'évènement est précisée, l'index sera utilisé :

```
sql=# EXPLAIN SELECT * FROM evenements
      WHERE type = 'FACTURATION' AND NOT traite;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using index_partiel on evenements  (cost=0.27..0.48 rows=1 width=47)
    Index Cond: (type = 'FACTURATION'::text)
```

Sur ce jeu de données, on peut comparer la taille de deux index, partiels ou non :

```
CREATE INDEX index_complet ON evenements(type);

SELECT idxname, pg_size_pretty(pg_total_relation_size(idxname::text))
```

```
      FROM (VALUES ('index_complet'), ('index_partiel')) as a(idxname);

    idxname     | pg_size_pretty
----------------+----------------
 index_complet | 328 kB
 index_partiel | 16 kB
(2 lignes)
```

4.4.2 INDEX PARTIELS : CAS D'USAGE

- Données *chaudes* et *froides*
- Index pour une requête ayant une condition fixe
- Éviter les index de type :

```
CREATE INDEX ON index(une_colonne) WHERE une_colonne = 'test`
```

Le cas typique d'utilisation d'un index partiel correspond, comme dans l'exemple précédent, aux applications contenant des données *chaudes*, fréquemment accédées et traitées, et *froides*, qui sont plus destinées à de l'historisation ou de l'archivage.

Par exemple, un système de vente en ligne aura probablement intérêt à disposer d'index sur les commandes dont l'état est différent de clôturé : en effet, un tel système effectuera probablement des requêtes fréquemment sur les commandes qui sont en cours de traitement, en attente d'expédition, en cours de livraison mais très peu sur des commandes déjà livrées, qui ne serviront alors plus qu'à de l'analyse statistique. De manière générale, tout système ayant à gérer des données ayant un état, et dont seul un sous-ensemble des états est activement exploité par l'application.

Nous avons mentionné précédemment qu'un index est destiné à satisfaire une requête ou un ensemble de requêtes. De cette manière, si une requête présente fréquemment des critères de types

```
WHERE une_colonne = un_parametre_variable
   AND une_autre_colonne = une_valeur_fixe`
```

alors il peut-être intéressant de créer un index partiel pour les lignes satisfaisant le critère :

```
une_autre_colonne = une_valeur_fixe
```

Ces critères sont généralement très liés au fonctionnel de l'application : au niveau exploitation, il est potentiellement difficile d'identifier des requêtes dont une valeur est toujours fixe. Encore une fois, l'appropriation des techniques d'indexation par l'équipe de développement permet alors d'améliorer grandement les performances de l'application.

Les index partiels ne doivent généralement pas filtrer sur la colonne qui est indéxée : en effet, le contenu de l'index est alors peu utile, toutes les entrées pointant vers la même clé. Il est alors plus intéressant d'y associer une colonne sur laquelle un prédicat ou un tri est effectué.

4.4.3 INDEX FONCTIONNELS

- Il s'agit d'un index sur le résultat d'une fonction :

  ```
  WHERE upper(a)='DUPOND'
  ```

- l'index classique ne fonctionne pas

  ```
  CREATE INDEX mon_idx ON ma_table ((upper(a))
  ```

- La fonction doit être `IMMUTABLE`

À partir du moment où une clause `WHERE` applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement.

C'est comme demander à un dictionnaire Anglais vers Français : « *Quels sont les mots dont la traduction en Français est 'Fenêtre'* ». Le tri du dictionnaire ne correspond pas à la question posée. Il nous faudrait un index non plus sur les mots Anglais, mais sur leur traduction en Français.

C'est exactement ce que font les index fonctionnels : ils indexent le résultat d'une fonction appliquée à l'enregistrement.

L'exemple classique est l'indexation insensible à la casse : on crée un index sur `upper` (ou `lower`) de la chaîne à indexer, et on recherche les mots convertis à la casse souhaitée.

La fonction d'indexation utilisée doit être `IMMUTABLE` : sa valeur de retour ne doit dépendre que d'une seule chose : ses paramètres en entrée. Si elle dépend du contenu de la base (c'est à dire qu'elle exécute des requêtes d'interrogation), ou qu'elle dépend d'une variable de session, elle n'est pas utilisable pour l'index : l'endroit dans lequel la donnée devrait être insérée dans l'index dépendrait de ces paramètres, et serait donc potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Si une fonction non `IMMUTABLE` est utilisée pour crée un index fonctionnel, PostgreSQL refuse, avec l'erreur :

```
ERROR:  functions in index expression must be marked IMMUTABLE
```

4.4.4 COVERING INDEXES

On trouve parfois « index couvrants » dans la littérature française.

```
CREATE INDEX idx1 on T1 (col1,col2)
```

- Répondent à la clause `WHERE`
- **ET** contiennent toutes les colonnes demandées par la requête :

  ```
  SELECT col1,col2` FROM t1 WHERE col1>12
  ```

- Pas de visite de la table (donc peu d'accès aléatoires, l'index étant à peu près trié physiquement)

Les `Covering Indexes` sont une nouveauté de PostgreSQL 9.2. Pour pouvoir en bénéficier, il faut que toutes les colonnes retournées par la requête soient présentes dans l'index.

Un parcours d'index classique (`Index Scan`) est en fait un aller/retour entre l'index et la table : on va chercher un enregistrement dans l'index, qui nous donne son adresse dans la table, on accède à cet enregistrement dans la table, puis on passe à l'entrée d'index suivante. Le coût en entrées-sorties peut être énorme : les données de la table sont habituellement éparpillées dans tous les blocs.

Le parcours d'index permis par les `Covering Indexes` (`Index Only Scan`) n'a plus besoin de cette interrogation de la table. Les enregistrements cherchés étant contigus dans l'index (puisqu'il est trié), le nombre d'accès disque est bien plus faible, ce qui peut apporter des gains de performances énormes en sélection. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

4.4.5 CLASSES D'OPÉRATEURS

- Un index utilise des opérateurs de comparaison :
- Il peut exister plusieurs façons de comparer deux données du même type
- Par exemple, pour les chaînes de caractères
  - Différentes collations
  - Tri sans collation (pour `LIKE`)

    ```
    CREATE INDEX idx1 ON ma_table (col_varchar varchar_pattern_ops)
    ```

- Permet :

  ```
  SELECT ... FROM ma_table WHERE col_varchar LIKE 'chaine%'
  ```

Il est tout à fait possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison pour l'indexations, dans des cas particuliers.

Le cas d'utilisation le plus fréquent d'utilisation dans PostgreSQL est la comparaison de chaine `LIKE 'chaine%'`. L'indexation texte « classique » utilise la collation par défaut de la base (ou la collation de la colonne de la table, dans les versions récentes de PostgreSQL). Cette collation peut être complexe (par exemple, est-ce que le « ß » allemand est équivalent à « ss », quelles sont les règles majuscules/minuscules, etc). Cette collation n'est pas compatible avec l'opérateur `LIKE`. Si on reprend l'exemple précédent, est-ce que `LIKE 'stras'` doit retourner « straße » ?

Les règles sont différentes pour chaque collation, et il serait donc très complexe de réécrire le LIKE en un BETWEEN, comme il le fait habituellement pour tous les SGBD : `col_texte LIKE 'toto%'` peut être réécrit comme `coltexte> 'toto' and coltexte< 'totop'` en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur unicode par exemple. Cela permet d'utiliser un index sur coltexte, mais uniquement si celui-ci est aussi trié en `ASCII`.

C'est à cela que sert la classe d'opérateurs varchar_pattern_ops : l'index est construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient. Il devient alors trivial pour l'optimiseur de faire la réécriture.

Il existe quelques autres cas d'utilisation d'`opclass` alternatives, mais elles sont habituellement liées à l'utilisation d'un module comme `pg_trgm`, qui fournit des types de données complexes, et sont donc clairement documentées avec ce module.

4.4.6 GIN

`GIN` : *Generalized Inverted iNdex*

- Index inversé généralisé
- Index inversé ?
  - Index associe une valeur à la liste de ses adresses
  - Utile pour tableaux, listes…
- Pour chaque entrée du tableau
  - Liste d'adresses (`TID`) où le trouver
  - Compressée à partir de 9.4 => alternative à bitmap

Un index inversé est une structure classique, utilisée le plus souvent dans l'indexation `Full Text`. Le principe est de décomposer un document en sous-structures, qui seront indexées. Par exemple, un document sera décomposé en la liste de ses mots,

et chaque mot sera une clé de l'index. Cette clé fournira la liste des documents contenant ce mot. Pour plus de détail sur la structure elle-même, [l'article de Wikipedia]( ](http://fr.wikipedia.org/wiki/Index_inversé) est une lecture conseillée.

Les index `GIN` de PostgreSQL sont « généralisés » car ils sont capables d'indexer n'importe quel type de données, à partir du moment où on lui fournit les différentes fonctions d'API permettant le découpage et le stockage des différents `items` composant la donnée à indexer.

Nativement, `GIN` supporte dans PostgreSQL les tableaux de type natif (`int`, `float`, `text`, `date`...), les `tsvector` utilisés pour l'indexation `Full Text`, les `jsonb` (depuis PostgreSQL 9.4), ainsi que tous les types scalaires ayant un index `btree`, à partir du moment où on installe l'extension `btree_gin`.

L'extension `pg_trgm` utilise aussi les index `GIN`, pour permettre des recherches de type `SELECT * FROM ma_table WHERE ma_col_texte LIKE '%ma_chaine1%ma_chaine2%'` qui utilisent un index.

Leur structure en fait des structures lentes à la mise à jour. Par contre, elles sont extrêmement efficaces pour les interrogations multicritères, ce qui les rend très appropriées pour l'indexation `Full Text`, le `jsonb`...

Un autre cas d'utilisation, depuis PostgreSQL 9.4, est le cas d'utilisation traditionnel des index `bitmap`. Les index `bitmap` sont très compact, mais ne permettent d'indexer que peu de valeurs différentes. Un index `bitmap` est une structure utilisant 1 bit par enregistrement pour chaque valeur indexable. Par exemple, on peut définir un index `bitmap` sur le sexe : deux (ou trois si on autorise null ou indéfini) valeurs seulement sont possibles. Indexer un enregistrement nécessitera donc un ou deux bits. Le défaut de ces index est qu'ils sont très peu performants si on rajoute des nouvelles valeurs, et se dégradent avec l'ajout de nouvelles valeurs : leur taille par enregistrement devient bien plus grosse, et l'index nécessite une réécriture complète à chaque ajout de nouvelle valeur.

Les index `GIN` permettent un fonctionnement sensiblement équivalent au `bitmap` : chaque valeur indexable contient la liste des enregistrements répondant au critère. Cette liste est, depuis PostgreSQL 9.4, compressée. Voici un exemple :

```
CREATE TABLE demo_gin (id bigint, nom text, sexe varchar(1));
INSERT INTO demo_gin SELECT i, 'aaaaaaaa',
    CASE WHEN random()*2 <1 THEN 'H'
         ELSE 'F'
    END FROM generate_series(1,10000000) g(i);
CREATE INDEX idx_nom ON demo_gin(nom);
CREATE EXTENSION btree_gin;
CREATE INDEX idx_sexe ON demo_gin USING gin(sexe);
```

Voici les tailles respectives :

```
=# select pg_size_pretty(pg_table_size('demo_gin'));
 pg_size_pretty
----------------
 498 MB
(1 ligne)

=# select pg_size_pretty(pg_table_size('idx_nom'));
 pg_size_pretty
----------------
 301 MB
(1 ligne)

=# select pg_size_pretty(pg_table_size('idx_sexe'));
 pg_size_pretty
----------------
 10 MB
(1 ligne)
```

Un index `btree` sur la colonne sexe aurait occupé 214Mo. L'index est donc environ 20 fois plus compact, dans cet exemple simple.

On peut aussi utiliser un index `GIN` pour indexer le contenu d'une liste texte, si par exemple on a une table ne respectant pas la première forme normale. Imaginons le champ « `attributs` », contenant une liste d'attributs séparés par une virgule :

```
CREATE INDEX idx_attributs_array ON ma_table
    USING gin (regexp_split_to_array(attributs,','));
```

La fonction `regexp_split_to_array` découpe la chaîne attribut sur le séparateur « , » ,et retourne un tableau, qui peut donc être indexé avec `GIN`.

On peut ensuite écrire des requêtes sous la forme :

```
SELECT * FROM ma_table WHERE regexp_split_to_array(attributs,',') @>
    '{"toit ouvrant","vitres teintées"}';
```

4.4.7 GIST

`GiST` : *Generalized Search Tree*

- Arbre de recherche généralisé
- Indexation non plus des valeurs mais de la véracité de prédicats
- Moins performants car moins sélectifs que `btree`
- Mais peuvent indexer à peu près n'importe quoi

- Multi-colonnes dans n'importe quel ordre
- Sur-ensemble de `btree` et `rtree`

Initialement, les index `GiST` sont un produit de la recherche de l'université de Berkeley. L'idée fondamentale est de pouvoir indexer non plus les valeurs dans l'arbre `btree`, mais plutôt la véracité d'un prédicat : « *ce prédicat est vrai sur telle sous-branche* ». On dispose donc d'une API permettant au type de données d'informer le moteur `GiST` d'informations comme : « *quel est le résultat de la fusion de tel et tel prédicat* » (pour pouvoir déterminer le prédicat du nœud parent), quel est le surcout d'ajout de tel prédicat dans telle ou telle partie de l'arbre, comment réaliser un *split* (découpage) d'une page d'index, déterminer la distance entre deux prédicats, etc.

Tout ceci est très virtuel, et rarement re-développé par les utilisateurs. Ce qui est important, c'est :

- Que ces index sont moins performants que `btree`
- Mais qu'ils permettent d'indexer des choses bien plus complexes : on peut indexer n'importe quoi avec `GiST`, quelle que soit la dimension, le type, tant qu'on peut utiliser des prédicats sur ce type
- Il est disponible pour les types natifs suivants :
  - Les types géométriques (`box`, `circle`, `point`, `poly`)
  - Les types `range` (d'`int`, de `timestamp`...)
  - Le `Full Text` (plus rapide à maintenir qu'un index `GIN`, mais moins performant à l'interrogation)
  - Les adresses IP/CIDR
- Il est en outre utilisé par
  - Le projet PostGIS, pour répondre à des questions complexes telles que « *quels sont les routes qui coupent le Rhône* », « *quelles sont les villes adjacentes à Toulouse* », « *quels sont les restaurants situés à moins de 3 km de la Nationale 12* »
  - `pg_trgm` (moins efficace que `GIN` pour la recherche exacte, mais permet de rapidement trouver les N enregistrements les plus proches d'une chaîne donnée, sans tri, et est plus compact)
- Que les index `GiST` ont moins tendance à se fragmenter que les index `GIN`, même si c'est difficilement quantifiable, car cela dépend énormément du type de mises à jour.
- Que les index `GiST` sont moins lourds à maintenir que les index `GIN`

Il est utilisé pour les « Constraint Exclusions » (Exclusions de contraintes), par exemple pour interdire que la même salle soit réservée simultanément sur deux intervalles en intersection.

Il est aussi intéressant si on a besoin d'indexer plusieurs colonnes sans trop savoir dans

quel ordre on va les accéder. On peut faire un index GiST multi-colonnes, et voir si ses performances sont satisfaisantes.

Si on reprend l'exemple du slide précédent (la table demo_gin a été renommée demo_gist) :

```
=# CREATE EXTENSION btree_gist ;
CREATE EXTENSION
=# CREATE INDEX ON demo_gin USING gist(nom,sexe);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM demo_gin WHERE sexe='b';
                          QUERY PLAN
-----------------------------------------------------
 Index Scan using demo_gist_nom_sexe_idx on demo_gist
        (cost=0.42..8.44 rows=1 width=19)
        (actual time=0.054..0.054 rows=0 loops=1)
   Index Cond: ((sexe)::text = 'b'::text)
 Planning time: 0.214 ms
 Execution time: 0.159 ms
(4 lignes)

=# EXPLAIN ANALYZE SELECT * FROM demo_gist WHERE nom='b';
                       QUERY PLAN
-------------------------------------------------------
 Index Scan using demo_gist_nom_sexe_idx on demo_gist
        (cost=0.42..8.44 rows=1 width=19)
        (actual time=0.054..0.054 rows=0 loops=1)
   Index Cond: (nom = 'b'::text)
 Planning time: 0.224 ms
 Execution time: 0.178 ms
(4 lignes)
```

Ici, les clauses WHERE ne ramènent rien, on est donc dans un cas particulier où les performances sont forcément excellentes. Dans le cas d'une application réelle, il faudra tester de façon très minutieuse.

4.4.8 KNN

- KNN = K-Nearest neighbours, K plus proches voisins
- Requêtes de types

  ```
  ORDER BY ma_colonne <-> une_référence LIMIT 10
  ```

- Très utile pour la recherche de mots ressemblants, géographique

Depuis la version 9.1, les index GiST supportent les requêtes de type *K-plus proche voisins*, et permettent donc de répondre extrêmement rapidement à des requêtes telles que :

- quels sont les dix restaurants les plus proches d'un point particulier
- quels sont les 5 mots ressemblant le plus à "éphélant", afin de proposer des corrections à un utilisateur ayant commis une faute de frappe

Une convention veut que l'opérateur distance soit généralement nommé « <-> », mais rien n'impose ce choix.

On peut par exemple prendre l'exemple d'indexation ci-dessus, avec le type natif POINT :

```
=# CREATE TABLE mes_points (p point);
CREATE TABLE

=# INSERT INTO mes_points (SELECT point(i, j)
FROM generate_series(1, 100) i, generate_series(1,100) j WHERE random() > 0.8);
INSERT 0 2029

=# CREATE INDEX ON mes_points using gist (p);
CREATE INDEX
```

Pour trouver les 4 points les plus proches du point ayant pour coordonnées (18,36), on peut utiliser la requête suivante :

```
=# SELECT p, p <-> point(18,36)
     FROM mes_points
     ORDER BY p <-> point(18, 36)
     LIMIT 4;
    p    |     ?column?
---------+-----------------
 (18,37) |                1
 (18,35) |                1
 (16,36) |                2
 (16,35) | 2.23606797749979
(4 lignes)
```

Cette requête utilise bien l'index GiST créé plus haut :

```
                          QUERY PLAN
-----------------------------------------------------
 Limit  (cost=0.14..0.49 rows=4 width=16)
        (actual time=0.049..0.052 rows=4 loops=1)
   ->  Index Scan using mes_points_p_idx on mes_points
            (cost=0.14..176.72 rows=2029 width=16)
            (actual time=0.047..0.049 rows=4 loops=1)
         Order By: (p <-> '(18,36)'::point)
```

```
 Planning time: 0.050 ms
 Execution time: 0.075 ms
(5 lignes)
```

4.4.9 BRIN

BRIN : **B**lock **R**ange **IN**dex (9.5+)

- Utile pour les tables très volumineuses
  - L'index produit est petit
- Performant lorsque les valeurs sont corrélées à leur emplacement physique
- Types qui peuvent être triés linéairement (pour obtenir `min`/`max`)

Soit une table `brin_demo` contenant l'âge de 100 millions de personnes.

```
=# CREATE TABLE brin_demo (c1 int);
=# INSERT INTO brin_demo (SELECT trunc(random() * 90 + 1) AS i
   FROM generate_series(1,100000000));

=# \dt+ brin_demo
                        Liste des relations
 Schéma |    Nom    | Type  | Propriétaire | Taille  | Description
--------+-----------+-------+--------------+---------+-------------
 public | brin_demo | table | postgres     | 3458 MB |
```

Un index `btree` va permettre d'obtenir l'emplacement physique (bloc) d'une valeur.

Un index `BRIN` va contenir une plage des valeurs pour chaque bloc. Dans notre exemple, l'index contiendra la valeur minimale et maximale de plusieurs blocs. La conséquence est que ce type d'index prend peu de place, il peut facilement tenir en RAM (réduction des IO disques) :

```
=# CREATE INDEX demo_btree_idx ON brin_demo USING btree (c1);
=# CREATE INDEX demo_brin_idx ON brin_demo USING brin (c1);

=# \di+
                                    Liste des relations
 Schéma |     Nom          | Type  | Propriétaire |      Table      | Taille
--------+------------------+-------+--------------+-----------------+---------
 public | demo_brin_idx    | index | postgres     | brin_demo       | 128 kB
 public | demo_btree_idx   | index | postgres     | brin_demo       | 2142 MB
```

Réduisons le nombre d'enregistrements dans la table :

```
=# TRUNCATE brin_demo ;
=# INSERT INTO brin_demo SELECT trunc(random() * 90 + 1) AS i
   FROM generate_series(1,100000);
```

Créons un index BRIN avec le paramètre pages_per_range à 16. Chaque page de l'index contiendra la plage de valeurs de 16 blocs :

```
=# SELECT * FROM brin_page_items(get_raw_page('brin_demo_brin_idx_16', 2),
                                 'brin_demo_brin_idx_16');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+-----------
          1 |      0 |      1 | f        | f        | f           | {1 .. 90}
          2 |     16 |      1 | f        | f        | f           | {1 .. 90}
          3 |     32 |      1 | f        | f        | f           | {1 .. 90}
...
```

On constate que les blocs de 0 à 16 contiennent les valeurs de 1 à 90. Ceci s'explique par le fait que les valeurs que nous avons insérées étaient aléatoires. Si nous réorganisons la table :

```
=# CLUSTER brin_demo USING brin_demo_btree_idx;
=# SELECT * FROM brin_page_items(
    get_raw_page('brin_demo_brin_idx_16', 2), 'brin_demo_brin_idx_16');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 4}
          2 |     16 |      1 | f        | f        | f           | {4 .. 7}
          3 |     32 |      1 | f        | f        | f           | {7 .. 10}
...
```

Les 16 premiers blocs contiennent les valeurs de 1 à 4.

Ainsi, pour une requête du type SELECT c1 FROM brin_demo WHERE c1 BETWEEN 1 AND 9, le moteur n'aura qu'à parcourir les 32 premiers blocs de la table.

Autre exemple avec plusieurs colonnes et un type text :

```
=# CREATE TABLE test (id serial primary key, val text);

=# INSERT INTO test (val) SELECT md5(i::text)
   FROM generate_series(1, 10000000) i;
```

La colonne id sera corrélée (séquence), colonne md5 qui ne sera pas du tout corrélée.

On crée un index sur deux colonnes, un int (val) et un text (val).

```
=# CREATE INDEX brin1_idx ON test USING brin (id,val);

=# \dt+ test
                    List of relations
```

```
 Schema | Name | Type  |  Owner   |  Size  | Description
--------+------+-------+----------+--------+-------------
 cave   | test | table | postgres | 651 MB |
(1 row)

=# \di+ brin1*
                          List of relations
 Schema |   Name    | Type  |  Owner   | Table |  Size  | Description
--------+-----------+-------+----------+-------+--------+-------------
 cave   | brin1_idx | index | postgres | test  | 112 kB |
(1 row)
```

Un aperçu du contenu de l'index :

```
=# SELECT itemoffset,blknum,attnum,value
   FROM brin_page_items(get_raw_page('brin1_idx', 2),'brin1_idx') LIMIT 3;

 itemoffset | blknum | attnum |                    value
------------+--------+--------+-------------------------------------------
          1 |      0 |      1 | {1 .. 15360}
          1 |      0 |      2 | {00003e3b9e5336685200ae85d21b4f5e
                                .. fffb8ef15de06d87e6ba6c830f3b6284}
          2 |    128 |      1 | {15361 .. 30720}
          2 |    128 |      2 | {00053f5e11d1fe4e49a221165b39abc9
                                .. fffe9f664c2ddba4a37bcd35936c7422}
          3 |    256 |      1 | {30721 .. 46080}
          3 |    256 |      2 | {0002ac0d783338cfeab0b2bdbd872cda
                                .. fffffe98d0963d27015c198262d97221}
```

La colonne blknum indique le bloc. Par défaut, le nombre de pages est de 128. La colonne attnum correspond à l'attribut. On remarque bien que l'id est corrélé, contrairement à la colonne val. Ce que nous confirme bien la vue pg_stats :

```
select tablename,attname,correlation from pg_stats where tablename='test';
 tablename | attname | correlation
-----------+---------+-------------
 test      | id      |           1
 test      | val     |  0.00528745
(2 rows)
```

Testons une requête :

```
=# explain (buffers,analyze) select * from test
where val between 'a87ff679a2f3e71d9181a67b7542122c'
    and 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=27558.34..151163.08 rows=2684716 width=37)
                           (actual time=17.668..1471.960 rows=2668675 loops=1)
```

```
   Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                  AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Rows Removed by Index Recheck: 7331325
   Heap Blocks: lossy=83334
   Buffers: shared hit=1 read=83342
   ->  Bitmap Index Scan on brin1_idx
            (cost=0.00..26887.16 rows=2684716 width=0)
            (actual time=17.549..17.549 rows=834560 loops=1)
         Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                      AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
         Buffers: shared hit=1 read=8
 Planning time: 0.198 ms
 Execution time: 1558.542 ms
(10 rows)
```

83343 blocs lus. Equivalent à 651 Mo soit l'intégralité de la table !

```
=# CREATE INDEX brin_btree_idx ON test USING btree (val);

=# \di+ brin_btree_idx
                              List of relations
 Schema |      Name      | Type  |  Owner   | Table |  Size  | Description
--------+----------------+-------+----------+-------+--------+-------------
 cave   | brin_btree_idx | index | postgres | test  | 563 MB |
```

Au passage, notre index est presque aussi gros que notre table !

Après la commande `cluster`, notre table est bien corrélée avec `val` (mais plus avec id) :

```
=# ANALYZE TEST;
ANALYZE
=# SELECT tablename,attname,correlation
   FROM pg_stats WHERE tablename='test';
 tablename | attname | correlation
-----------+---------+-------------
 test      | id      | -0.00373068
 test      | val     |           1
(2 rows)
```

La requête après le `cluster` :

```
=# EXPLAIN (buffers,analyze) SELECT * FROM test
where val between 'a87ff679a2f3e71d9181a67b7542122c'
    and 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=27558.34..151163.08 rows=2684716 width=37)
                           (actual time=2.458..476.453 rows=2668675 loops=1)
   Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
```

```
                AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Rows Removed by Index Recheck: 19325
   Heap Blocks: lossy=22400
   Buffers: shared hit=9 read=22400
   ->  Bitmap Index Scan on brin1_idx
            (cost=0.00..26887.16 rows=2684716 width=0)
            (actual time=1.719..1.719 rows=224000 loops=1)
         Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                     AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
         Buffers: shared hit=9
 Planning time: 0.293 ms
 Execution time: 557.283 ms
(10 rows)
```

22409 blocs lus soit 175 Mo.

On supprime notre index BRIN et on garde l'index btree :

```
=# DROP INDEX brin1_idx;
=# EXPLAIN (buffers,analyze) SELECT * FROM test
where val between 'a87ff679a2f3e71d9181a67b7542122c'
and 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using brin_btree_idx on test
            (cost=0.56..151908.16 rows=2657080 width=37)
            (actual time=0.032..449.482 rows=2668675 loops=1)
   Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Buffers: shared read=41306 written=10
 Planning time: 0.137 ms
 Execution time: 531.724 ms
```

Même durée d'exécution mais le nombre de blocs lus est beaucoup plus important :

41306 blocs soit 322 Mo. Presque deux fois plus de blocs lus.

En résumé, les index BRIN sont intéressants pour les tables volumineuses et où il y a une forte corrélation entre les valeurs et leur emplacement physique.

Plus d'information sur les index BRIN[30] .

---

[30] https://kb.dalibo.com/conferences/index_brin/index_brin_pgday

4.4.10 HASH

Index Hash : * Journalisés uniquement depuis la version 10 * donc facilement corrompus sur les versions antérieures * Moins performants que les `btree` * Ne gèrent que les égalités, pas « < » et « > » * Mais plus compacts * À ne pas utiliser

Les index Hash n'étaient pas journalisés avant la version 10. Cela sous-entend qu'ils risquaient une corruption à chaque arrêt brutal. Ils étaient aussi peu performant par rapport à des index `btree`. Ceci explique le peu d'utilisation de ce type d'index jusqu'à maintenant.

L'utilisation d'un index Hash est donc une mauvaise idée avant la version 10.

4.5 OUTILS

- pour l'identification des requêtes
- pour l'identification des prédicats et des requêtes liées
- pour la validation de l'index à créer

Différents outils permettent d'aider le développeur ou le DBA à identifer plus facilement les index à créer. On peut classer ceux-ci en trois groupes, selon l'étape de la méthodologie à laquelle ils s'appliquent.

4.5.1 IDENTIFIER LES REQUÊTES

- `PgBadger`
- `pg_stat_statements`
- `PoWA`

Pour identifier les requêtes les plus lentes, et donc potentiellement nécessitant une réécriture ou un nouvel index, `PgBadger` permet d'analyser les logs une fois ceux-ci configurés pour tracer toutes les requêtes. Il est disponible à l'adresse suivante : https://dalibo.github.io/pgbadger.

Pour une vision plus *temps-réel* de ces requêtes, l'extension `pg_stat_statements`, fournie avec les « contrib » PostgreSQL, permet de garder trace des N-requêtes les plus fréquemment exécutées, et conserve ainsi le temps d'éxécution total de celle-ci, ainsi que les accès au cache de PostgreSQL ou au système de fichiers.

Le projet PoWA ajoute de la valeur à ces statistiques en les historisant, et en fournissant une interface web permettant de les exploiter. Voir la documentation[31] du projet PoWA pour plus d'information.

4.5.2 IDENTIFIER LES PRÉDICATS ET DES REQUÊTES LIÉES

- pg_qualstats
  - avec PoWa

Pour identifier quels sont les prédicats (clause WHERE ou condition de jointure à identifier en priorité), l'extension pg_qualstats permet de pousser l'analyse offerte par pg_stat_statements au niveau du prédicat lui-même. Ainsi, on peut détecter quelles sont les requêtes utilisant les mêmes colonnes, ce qui peut aider notamment à déterminer des index multi-colonnes ou des index partiels.

De même que pg_stat_statements, cette extension peut être historisée et exploitée par le biais du projet PoWA.

4.5.3 ÉTUDE DES INDEX À CRÉER

- PoWA
- HypoPG

Le projet PoWA propose une fonctionnalité, encore rudimentaire, de suggestion d'index à créer.

Pour éviter de créer un index sur la base pour valider son utilisation, l'extension HypoPG permet de créer des index hypothétiques, et donc de répondre à la question « Quel serait le plan d'exécution de ma requête si cet index existait ? ».

L'intégration d'HypoPG dans PoWA permet là aussi une souplesse d'utilisation, en présentant les plans espérés avec ou sans les index suggérés.

Pour montrer l'utilité de ces outils, nous pouvons utiliser le script suivant :

```
./run.pl --conf=conf --requetes=run.txt
```

Ensuite, en ouvrant l'interface de PoWA, on peut étudier les différentes requêtes, et les suggestions d'index réalisées par l'outil. À partir de ces suggestions, on peut créer les

[31] https://powa.readthedocs.org/en/latest

nouveaux index, et enfin relancer le bench pour constater les améliorations de performances.

4.5.4 CONCLUSION

- Responsabilité de l'indexation
- Compréhension des mécanismes
- Différents types d'index, différentes stratégies
- Outillage

L'indexation d'une base de données est souvent un sujet qui est traité trop tard dans le cycle de l'application. Lorsque celle-ci est géré à l'étape du développement, il est possible de bénéficier de l'expérience et de la connaissance des développeurs. La maîtrise de cette compétence est donc idéalement transverse entre le développement et l'exploitation.

Le fonctionnement d'un index est somme toute assez simple, mais il est important de l'appréhender pour comprendre les enjeux d'une bonne stratégie d'indexation.

Nous avons pu voir des types d'index assez avancés : le but n'était bien évidemment pas de maîtriser leurs syntaxes mais plutôt connaître leur existence ainsi que leurs cas d'application.

Enfin, de nouveaux outils open-source arrivent sur le marché pour faciliter la mise en place d'une stratégie d'indexation efficace. Ceux-ci ne peuvent que s'améliorer pour le futur.

4.6 TRAVAUX PRATIQUES

4.6.1 ENONCÉS

Cette série de question utilise la base `magasin`, qui est disponible dans le schéma `magasin` de la base de TP.

Considérons le cas d'usage d'une recherche de commandes par date. Le besoin fonctionnel est le suivant : renvoyer l'intégralité des commandes passées au mois de janvier 2014.

Index "simples"

- Écrire cette requête.

- Afficher le plan de celle-ci, en utilisant `EXPLAIN (ANALYZE, BUFFERS)`. Que constatez-vous ?

Nous souhaitons désormais afficher les résultats à l'utilisateur par ordre de date croissante.

- Réécrire la requête.
- Afficher de nouveau son plan. Que constatez-vous ?

Maintenant, nous allons essayer d'optimiser ces deux requêtes.

- Créer un index permettant de répondre à ces requêtes.
- Afficher de nouveau le plan de celles-ci. Que constatez-vous ?

Maintenant, étudions l'impact des index pour une opération de jointure. Le besoin fonctionnel est désormais de lister toutes les commandes associées à un client (admettons, dont le `client_id` vaut 3), avec les informations du client lui-même.

- Écrire cette requête.
- Afficher son plan. Que constatez-vous ?
- Créer un index pour accélérer cette requête.
- Afficher de nouveau son plan. Que constatez-vous ?

Selectivité

- Écrivez une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').
- Ajoutez un index sur la colonne `type_client`, et rejouez les requêtes précédentes.
- Affichez leurs plans d'éxécution. Que se passe-t-il ? Pourquoi ?

Index partiels

Sur la base fournie pour les TPs, les lots non livrés sont constamment requêtés. Notamment, un système d'alerte est mis en place afin d'assurer un suivi qualité sur les lots à l'état suivant :

- En dépôt depuis plus de 12h, mais non expédié.
- Expédié, mais non réceptionné depuis plus de 3 jours.

17.12

Écrire les requêtes correspondant à ce besoin fonctionnel. Quel index peut-on créer pour optimiser celles-ci ? Affichez les plans d'éxécution. Que constate-t-on ?

Index fonctionnels

Pour répondre aux exigences de stockage, l'application a besoin de pouvoir trouver rapidement les produits dont le volume est compris entre certaines bornes (nous négligeons ici le facteur de forme, qui est problématique dans le cadre d'un véritable stockage en entrepôt !).

- Écrivez une requête permettant de renvoyer l'ensemble des produits dont le volume ne dépasse pas 1 L (les unités sont en mm, 1 L = 1 000 000 mm3)
- Comment peut-on optimiser cette requête ? (Astuce : les index *fonctionnels* ne permettent pas d'indexer une expression qui n'est pas une fonction !)

Cas d'index non utilisés

Un développeur cherche à récupérer les commandes dont le numéro d'expédion est 190774 avec cette requête :

```
select * from lignes_commandes WHERE numero_lot_expedition = '190774'::numeric;
```

- Créer un index pour améliorer son exécution.
- L'index est-il utilisé? Quel est le problème?

Ecrivez une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

- Créez un index pour améliorer l'exécution de cette requête.
- Pourquoi celui-ci n'est pas utilisé? Conseil : regardez la table `pg_stats`
- Faites le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

Index Gin

- Créer deux index de type btree et GIN sur `lignes_commandes(quantite)`.
- Comparer leur taille.

4.6.2 SOLUTIONS

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

Index "simples"

> Renvoyer l'intégralité des commandes passées au mois de janvier 2014.

Pour renvoyer l'ensemble de ces produits, la requête est très simple :

```
SELECT * FROM commandes date_commande where date_commande >= '2014-01-01'
AND date_commande < '2014-02-01';
```

Le plan de celle-ci est le suivant :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01';
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..25158.00 rows=19674 width=50)
                        (actual time=2.436..102.300 rows=19204 loops=1)
   Filter: ((date_commande >= '2014-01-01'::date)
            AND (date_commande < '2014-02-01'::date))
   Rows Removed by Filter: 980796
   Buffers: shared hit=10158
 Planning time: 0.057 ms
 Execution time: 102.929 ms
(6 lignes)
```

> Afficher les résultats à l'utilisateur par ordre de date croissante.

Ajoutons la clause `ORDER BY` :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01'
ORDER BY date_commande;
                               QUERY PLAN
-------------------------------------------------------------------------
 Sort  (cost=26561.15..26610.33 rows=19674 width=50)
       (actual time=103.895..104.726 rows=19204 loops=1)
   Sort Key: date_commande
   Sort Method: quicksort  Memory: 2961kB
   Buffers: shared hit=10158
```

```
   ->  Seq Scan on commandes  (cost=0.00..25158.00 rows=19674 width=50)
                              (actual time=2.801..102.181
                               rows=19204 loops=1)
         Filter: ((date_commande >= '2014-01-01'::date)
                  AND (date_commande < '2014-02-01'::date))
         Rows Removed by Filter: 980796
         Buffers: shared hit=10158
 Planning time: 0.096 ms
 Execution time: 105.410 ms
(10 lignes)
```

On constate ici que lors du parcours séquentiel, 980796 lignes ont été lues, puis écartées car ne correspondant pas au prédicat, nous laissant ainsi avec un total de 19204 lignes. Les valeurs précises peuvent changer, les données étant générées aléatoirement. De plus, le tri a été réalisé en mémoire. On constate de plus que 10158 blocs ont été parcourus, ici depuis le cache, mais ils auraient pu l'être depuis le disque.

> Créer un index permettant de répondre à ces requêtes.

Création de l'index :

```
CREATE INDEX idx_commandes_date_commande ON commandes(date_commande);
```

On constate que le plan utilise désormais l'index :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01';
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using idx_commandes_date_commande on commandes
        (cost=0.42..822.60 rows=19674 width=50)
        (actual time=0.015..3.311 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
                AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
 Planning time: 0.074 ms
 Execution time: 4.133 ms
(5 lignes)
```

Le temps d'exécution a été réduit considérablement : la requête est 25 fois plus rapide. On constate notamment que seuls 254 blocs ont été parcourus.

Pour la requête avec la clause `ORDER BY`, nous obtenons le plan d'éxécution suivant :

```
                        QUERY PLAN
-----------------------------------------------------------
```

```
 Index Scan using idx_commandes_date_commande on commandes
       (cost=0.42..822.60 rows=19674 width=50)
       (actual time=0.032..3.378 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
               AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
 Planning time: 0.516 ms
 Execution time: 4.049 ms
(5 lignes)
```

Celui-ci est identique ! En effet, l'index permettant un parcours trié, l'opération de tri est ici « gratuite ».

> Lister toutes les commandes associées à un client (admettons, dont le `client_id` vaut 3), avec les informations du client lui-même.

Pour récupérer toutes les commandes associées à un client :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
      INNER JOIN clients ON commandes.client_id = clients.client_id
    WHERE clients.client_id = 3;
                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.29..22666.42 rows=11 width=101)
              (actual time=8.799..80.771 rows=14 loops=1)
   Buffers: shared hit=10161
   ->  Index Scan using clients_pkey on clients
           (cost=0.29..8.31 rows=1 width=51)
           (actual time=0.017..0.018 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=3
   ->  Seq Scan on commandes  (cost=0.00..22658.00 rows=11 width=50)
                              (actual time=8.777..80.734 rows=14 loops=1)
         Filter: (client_id = 3)
         Rows Removed by Filter: 999986
         Buffers: shared hit=10158
 Planning time: 0.281 ms
 Execution time: 80.853 ms
(11 lignes)
```

Avec un index sur la colonne concernée par la clé étrangère :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
      INNER JOIN clients on commandes.client_id = clients.client_id
```

```
      WHERE clients.client_id = 3;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=4.80..55.98 rows=11 width=101)
              (actual time=0.064..0.189 rows=14 loops=1)
   Buffers: shared hit=23
   ->  Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.032..0.032 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=6
   ->  Bitmap Heap Scan on commandes  (cost=4.51..47.56 rows=11 width=50)
                                      (actual time=0.029..0.147
                                       rows=14 loops=1)
         Recheck Cond: (client_id = 3)
         Heap Blocks: exact=14
         Buffers: shared hit=17
         ->  Bitmap Index Scan on commandes_client_id_idx
                  (cost=0.00..4.51 rows=11 width=0)
                  (actual time=0.013..0.013 rows=14 loops=1)
               Index Cond: (client_id = 3)
               Buffers: shared hit=3
 Planning time: 0.486 ms
 Execution time: 0.264 ms
(14 lignes)
```

On constate ici un temps d'exécution divisé par 160 : en effet, on ne lit plus que 17 blocs pour la commande (3 pour l'index, 14 pour les données) au lieu de 10158.

Selectivité

> Écrivez une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').

Les requêtes :

```
SELECT * FROM clients WHERE type_client = 'P';
SELECT * FROM clients WHERE type_client = 'E';
```

Pour créer l'index :

```
CREATE INDEX ON clients (type_client);
```

Les plans d'éxécution :

```
EXPLAIN ANALYZE SELECT * FROM clients WHERE type_client = 'P';
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on clients  (cost=0.00..2276.00 rows=89803 width=51)
                      (actual time=0.006..12.877 rows=89800 loops=1)
   Filter: (type_client = 'P'::bpchar)
   Rows Removed by Filter: 10200
 Planning time: 0.374 ms
 Execution time: 16.063 ms
(5 lignes)

EXPLAIN ANALYZE SELECT * FROM clients WHERE type_client = 'E';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=154.50..1280.84 rows=8027 width=51)
                              (actual time=2.094..4.287 rows=8111 loops=1)
   Recheck Cond: (type_client = 'E'::bpchar)
   Heap Blocks: exact=1026
   ->  Bitmap Index Scan on clients_type_client_idx
           (cost=0.00..152.49 rows=8027 width=0)
           (actual time=1.986..1.986 rows=8111 loops=1)
         Index Cond: (type_client = 'E'::bpchar)
 Planning time: 0.152 ms
 Execution time: 4.654 ms
(7 lignes)
```

L'optimiseur sait estimer, à partir des statistiques (consultables via la vue `pg_stats`), qu'il y a approximativement 89000 clients particuliers, contre 8000 clients entreprise.

Dans le premier cas, la majorité de la table sera parcourue, et renvoyée : il n'y a aucun intérêt à utiliser l'index.

Dans l'autre, le nombre de lignes étant plus faible, l'index est bel et bien utilisé (via un BitmapScan, ici).

Index partiels

> Requêtes pour les besoins suivants :
>
> - En dépôt depuis plus de 12h, mais non expédié.
> - Expédié, mais non réceptionné depuis plus de 3 jours.

Les requêtes correspondantes sont les suivantes :

```
SELECT * FROM lots
    WHERE date_expedition IS NULL
```

```
    AND date_depot < now() - '12h'::interval
SELECT * FROM lots
    WHERE date_reception IS NULL
    AND date_expedition < now() - '3d'::interval;
```

On peut donc optimiser ces requêtes à l'aide des index partiels suivants :

```
CREATE INDEX ON lots (date_depot) WHERE date_expedition IS NULL;
CREATE INDEX ON lots (date_expedition) WHERE date_reception IS NULL;
```

Si l'on regarde les plans d'exécution de ces requêtes avec les nouveaux index, on voit qu'ils sont utilisés :

```
EXPLAIN (ANALYZE) SELECT * FROM lots
      WHERE date_reception IS NULL
      AND date_expedition < now() -  '3d'::interval;
                          QUERY PLAN
-----------------------------------------------------------------
 Index Scan using lots_date_expedition_idx on lots
        (cost=0.13..4.15 rows=1 width=43)
        (actual time=0.006..0.006 rows=0 loops=1)
   Index Cond: (date_expedition < (now() - '3 days'::interval))
 Planning time: 0.078 ms
 Execution time: 0.036 ms
(4 lignes)
```

Il est intéressant de noter que seul le test sur la condition indéxée (`date_receptioni`) est présent dans le plan : la condition `date_reception IS NULL` est implicitement validée par l'index.

Attention, il peut être tentant d'utiliser une formulation de la sorte pour ces requêtes :

```
SELECT * FROM lots
WHERE date_reception IS NULL
AND now() - date_expedition > '3d'::interval;
```

D'un point de vue logique, c'est la même chose, mais l'optimiseur n'est pas capable de réécrire cette requête correctement. Ici, l'index sera tout de même utilisé, le volume de lignes satisfaisant au critère étant très faible, mais il ne sera pas utilisé pour filtrer sur la date :

```
EXPLAIN ANALYZE SELECT * FROM lots
      WHERE date_reception IS NULL
      AND now() - date_expedition > '3d'::interval;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using lots_date_expedition_idx on lots
        (cost=0.12..4.15 rows=1 width=43)
```

```
        (actual time=0.007..0.007 rows=0 loops=1)
   Filter: ((now() - (date_expedition)::timestamp with time zone) >
           '3 days'::interval)
 Planning time: 0.204 ms
 Execution time: 0.132 ms
(4 lignes)
```

La ligne importante et différente ici concerne le `Filter` en lieu et place du `Index Cond` du plan précédent.

C'est une autre illustration des points vus précédemment sur les index non utilisés.

Index fonctionnels

> Pour répondre aux exigences de stockage, l'application a besoin de pouvoir trouver rapidement les produits dont le volume est compris entre certaines bornes (nous négligeons ici le facteur de forme, qui est problématique dans le cadre d'un véritable stockage en entrepôt !).
>
> - Écrivez une requête permettant de renvoyer l'ensemble des produits dont le volume ne dépasse pas 1 L (les unités sont en mm, 1 L = 1 000 000 mm3)
> - Comment peut-on optimiser cette requête ? (Astuce : les index *fonctionnels* ne permettent pas d'indexer une expression qui n'est pas une fonction !)

Concernant le volume des produits, la requête est assez simple :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000;
```

Pour optimiser cette requête, il va nous falloir créer une fonction `IMMUTABLE` :

```
CREATE OR REPLACE function volume(p produits) RETURNS numeric
AS $$
SELECT p.longueur * p.hauteur * p.largeur;
$$ language SQL
IMMUTABLE;
```

On peut ensuite indexer le résultat de cette fonction :

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

Il est ensuite possible d'écrire la requête de plusieurs manières, la fonction étant ici écrite en SQL et non en PL/pgSQL ou autre langage procédural :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000;
SELECT * FROM produits WHERE volume(produits) < 1000000;
```

En effet, l'optimiseur est capable de « regarder » à l'intérieur de la fonction SQL pour déterminer que les clauses sont les mêmes, ce qui n'est pas vrai pour les autres langages.

De part l'origine « relationnel-objet » de PostgreSQL, on peut même écrire la requête de la manière suivante :

```
SELECT * FROM produits WHERE produits.volume < 1000000;
```

Cas d'index non utilisés

> Un développeur cherche à récupérer les commandes dont le numéro d'expédion est 190774 avec cette requête :

```
select * from lignes_commandes WHERE numero_lot_expedition = '190774'::numeric;
```

```
explain (analyze,buffers) SELECT * FROM lignes_commandes
                            WHERE numero_lot_expedition = '190774'::numeric;
                      QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on lignes_commandes
                        (cost=0.00..89331.51 rows=15710 width=74)
                        (actual time=0.024..1395.705 rows=6 loops=1)
   Filter: ((numero_lot_expedition)::numeric = '190774'::numeric)
   Rows Removed by Filter: 3141961
   Buffers: shared hit=97 read=42105
 Planning time: 0.109 ms
 Execution time: 1395.741 ms
(6 rows)
```

Le moteur fait un parcours séquentiel et retire la plupart des enregistrements pour n'en conserver que 6.

> - Créer un index pour améliorer son exécution.

```
create index ON lignes_commandes (numero_lot_expedition );
```

> - L'index est-il utilisé? Quel est le problème?

L'index n'est pas utilisé à cause de la conversion bigint vers numeric. Il est important d'utiliser les bons types :

```
explain (analyze,buffers) SELECT * FROM lignes_commandes
                                  WHERE numero_lot_expedition = '190774';
                           QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using lignes_commandes_numero_lot_expedition_idx
    on lignes_commandes
        (cost=0.43..8.52 rows=5 width=74)
        (actual time=0.054..0.071 rows=6 loops=1)
   Index Cond: (numero_lot_expedition = '190774'::bigint)
   Buffers: shared hit=1 read=4
 Planning time: 0.325 ms
 Execution time: 0.100 ms
(5 rows)
```

Sans conversion la requête est bien plus rapide. Faites également le test sans index, le seqscan sera également plus rapide, le moteur n'ayant pas à convertir toutes les lignes parcourues.

> Ecrivez une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

```
explain (analyze,buffers) select * from lignes_commandes
                                   where quantite between 1 and 8;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on lignes_commandes
        (cost=0.00..89331.51 rows=2504357 width=74)
        (actual time=0.108..873.666 rows=2512740 loops=1)
   Filter: ((quantite >= 1) AND (quantite <= 8))
   Rows Removed by Filter: 629227
   Buffers: shared hit=16315 read=25887
 Planning time: 0.369 ms
 Execution time: 1009.537 ms
(6 rows)
```

> Créez un index pour améliorer l'exécution de cette requête.

```
create index ON lignes_commandes(quantite);
```

> Pourquoi celui-ci n'est pas utilisé? Conseil : regardez la table `pg_stats`

La table `pg_stats` nous donne des inforamtions de statistiques. Par exemple, pour la répartition des valeurs pour la colonne quantite:

```
n_distinct              | 10
most_common_vals        | {0,6,1,8,2,4,7,9,5,3}
most_common_freqs       | {0.1037,0.1018,0.101067,0.0999333,0.0999,0.0997,
                                    0.0995,0.0992333,0.0978333,0.0973333}
```

Ces quelques lignes nous indiquent qu'il y a 10 valeurs distinctes et qu'il y a environ 10% d'enregistrements correspondant à chaque valeur.

Avec le prédicat `quantite between 1 and 8` le moteur estime récupérer environ 80% de la table. Il est donc bien plus couteux de lire l'index et la table pour récupérer 80% de la table. C'est pourquoi le moteur fait un seqscan qui moins couteux.

> Faites le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

```
explain (analyze,buffers) SELECT * from lignes_commandes
                                  WHERE quantite between 1 and 4;
                           QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on lignes_commandes
            (cost=26538.09..87497.63 rows=1250503 width=74)
            (actual time=206.705..580.854 rows=1254886 loops=1)
   Recheck Cond: ((quantite >= 1) AND (quantite <= 4))
   Heap Blocks: exact=42202
   Buffers: shared read=45633
   ->  Bitmap Index Scan on lignes_commandes_quantite_idx
            (cost=0.00..26225.46 rows=1250503 width=0)
            (actual time=194.250..194.250 rows=1254886 loops=1)
         Index Cond: ((quantite >= 1) AND (quantite <= 4))
         Buffers: shared read=3431
 Planning time: 0.271 ms
 Execution time: 648.414 ms
(9 rows)
```

Cette fois la sélectivité est différente et le nombre d'enregistrent moins élevé. Le moteur passe donc par un parcours d'index.

Cet exemple montre qu'on indexe selon une requête et non selon une table.

Index Gin

> Comparer la taille d'un index sur `lignes_commandes(quantite)`

```
CREATE index ON lignes_commandes using gin (fournisseur_id );
ERROR:  data type bigint has no default operator class for access method "gin"
```

```
HINT:  You must specify an operator class for the index
    or define a default operator class for the data type.
CREATE extension btree_gin;
CREATE index lignes_commandes_fournisseur_id_gin ON lignes_commandes
    USING gin (fournisseur_id );
CREATE index lignes_commandes_fournisseur_id_btree ON lignes_commandes
    (fournisseur_id );
```

Les index GIN sont utiles pour indexer des types tableau ou JSON.

Il est necessaire d'utiliser l'extension `btree_gin` afin d'indexer des types scalaires (int ...)

Depuis la version 9.3 ces index sont compressés, ainsi la clé n'est indexée qu'une fois. Le gain est donc intéressant dès lors que la table comprend des valeurs identiques.

```
SELECT
pg_size_pretty(pg_relation_size('lignes_commandes_fournisseur_id_btree'));
 pg_size_pretty
----------------
 67 MB
(1 row)

SELECT
pg_size_pretty(pg_relation_size('lignes_commandes_fournisseur_id_gin'));
 pg_size_pretty
----------------
 15 MB
(1 row)
```

L'index est bien plus compact.

5 SQL AVANCÉ POUR LE TRANSACTIONNEL

5.0.1 PRÉAMBULE

- SQL et PostgreSQL proposent de nombreuses possibilités avancées
  - normes SQL:99, 2003, 2008 et 2011
  - parfois, extensions propres à PostgreSQL

La norme SQL a continué d'évoluer et a bénéficié d'un grand nombre d' améliorations. Beaucoup de requêtes qu'il était difficile d'exprimer avec les premières incarnations de la norme sont maintenant faciles à réaliser avec les dernières évolutions.

Ce module a pour objectif de voir les fonctionnalités pouvant être utiles pour développer une application transactionnelle.

5.0.2 MENU

- `LIMIT`/`OFFSET`
- jointures `LATERAL`
- `UPSERT` : INSERT ou UPDATE
- *Common Table Expressions*
- Serializable Snapshot Isolation

5.0.3 OBJECTIFS

- Aller au-delà de SQL:92
- Concevoir des requêtes simples pour résoudre des problèmes complexes

---

5.1 LIMIT

- Clause `LIMIT`
- ou syntaxe en norme SQL : `FETCH FIRST xx ROWS`
- Utilisation :
  - limite le nombre de lignes du résultat

La clause LIMIT, ou sa déclinaison normalisée par le comité ISO FETCH FIRST xx ROWS, permet de limiter le nombre de lignes résultant d'une requête SQL. La syntaxe normalisée vient de DB2 d'IBM et va être amenée à apparaître sur la plupart des bases de données. La syntaxe LIMIT reste néanmoins disponible sur de nombreux SGBD et est plus concise.

5.1.1 LIMIT : EXEMPLE

```
SELECT *
  FROM employes
 LIMIT 2;

 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
(2 lignes)
```

L'exemple ci-dessous s'appuie sur le jeu d'essai suivant :

```
SELECT *
  FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   |             | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

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 garantit en aucune façon l'ordre des résultats à moins d'employer la clause ORDER BY, et que l'ensemble des champs sur lequel on trie soit unique et non null.

Si une ligne était modifiée, changeant sa position physique dans la table, le résultat de la requête ne serait pas le même. Par exemple, en réalisant une mise à jour fictive de la ligne correspondant au matricule 00000001 :

```
UPDATE employes
   SET nom = nom
 WHERE matricule = '00000001';
```

L'ordre du résultat n'est pas garanti :

17.12

```
SELECT *
  FROM employes
 LIMIT 2;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+---------
 00000004  | Fantasio | Courrier    | 4500.00
 00000006  | Prunelle | Publication | 4000.00
(2 lignes)
```

L'application d'un critère de tri explicite permet d'obtenir la sortie souhaitée :

```
SELECT *
  FROM employes
 ORDER BY matricule
 LIMIT 2;
 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
```

5.1.2 OFFSET

- Clause OFFSET
  - à utiliser avec `LIMIT`
- Utilité :
  - pagination de résultat
  - sauter les *n* premières lignes avant d'afficher le résultat

Ainsi, en reprenant le jeu d'essai utilisé précédemment :

```
SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   |             | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

5.1.3 OFFSET : EXEMPLE (1/2)

- Sans offset :

```
SELECT *
  FROM employes
 LIMIT 2
 ORDER BY matricule;
```

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
```

5.1.4 OFFSET : EXEMPLE (2/2)

- En sautant les deux premières lignes :

```
SELECT *
  FROM employes
 ORDER BY matricule
 LIMIT 2
 OFFSET 2;
```

```
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+---------
 00000006  | Prunelle | Publication | 4000.00
 00000020  | Lagaffe  | Courrier    | 3000.00
```

5.1.5 OFFSET : ALTERNATIVE

- OFFSET peut être problématique :
  - beaucoup de données lues
  - temps de réponse dégradés
- Alternative possible
  - utilisation d'un index
  - couplé aux données composites
- Article sur le sujet[32]

[32] http://use-the-index-luke.com/fr/no-offset

Cependant, sur un jeu de données conséquent et une pagination importante, ce principe de fonctionnement peut devenir contre-performant. En effet, la base de données devra lire malgré tout les enregistrements qui n'apparaîtront pas dans le résultat de la requête, simplement dans le but de les compter.

Soit la table `posts` suivante :

```
sql2=# \d posts
                Table « public.posts »
  Colonne   |           Type           | Modificateurs
------------+--------------------------+---------------
 id_article | integer                  |
 id_post    | integer                  |
 ts         | timestamp with time zone |
 message    | text                     |
```

Si l'on souhaite récupérer les 10 premiers enregistrements

```
SELECT id_article, id_post, ts, message
  FROM posts
 WHERE id_article = 1907
 ORDER BY id_post
 LIMIT 10;
```

Cette requête génère le plan d'exécution suivant :

```
 Limit  (cost=0.43..40.69 rows=10 width=269)
        (actual time=0.075..0.102 rows=10 loops=1)
   Buffers: shared hit=13
   ->  Index Scan using posts_id_article_id_post_idx on posts
            (cost=0.43..3941.56 rows=979 width=269)
            (actual time=0.072..0.095 rows=10 loops=1)
         Index Cond: (id_article = 1907)
         Buffers: shared hit=13
 Planning time: 0.204 ms
 Execution time: 0.183 ms
```

La requête est rapide et ne lit que peu de données, ce qui est bien.

En revanche, si l'on saute un nombre conséquent d'enregistrements, la situation devient problématique :

```
SELECT id_article, id_post, ts, message
  FROM posts
 WHERE id_article = 1907
 ORDER BY id_post
 LIMIT 10
 OFFSET 1000;
```

Le plan d'exécution n'est plus le même. Pour répondre à la requête, PostgreSQL choisit la lecture de l'ensemble des résultats, puis leur tri, pour enfin appliquer la limite. Le temps d'exécution est alors de 4 milliseconde sur le serveur de test, et 1014 blocs sont accédés, ce qui reste encore raisonnable. Voici le plan généré :

```
Limit  (cost=3844.89..3844.89 rows=1 width=269)
       (actual time=4.655..4.661 rows=10 loops=1)
  Buffers: shared hit=1009 read=5
  I/O Timings: read=0.033
  ->  Sort  (cost=3842.44..3844.89 rows=979 width=269)
            (actual time=4.348..4.502 rows=1010 loops=1)
        Sort Key: id_post
        Sort Method: quicksort  Memory: 420kB
        Buffers: shared hit=1009 read=5
        I/O Timings: read=0.033
        ->  Bitmap Heap Scan on posts
                  (cost=20.02..3793.81 rows=979 width=269)
                  (actual time=1.096..3.345 rows=1011 loops=1)
              Recheck Cond: (id_article = 1907)
              Heap Blocks: exact=1009
              Buffers: shared hit=1009 read=5
              I/O Timings: read=0.033
              ->  Bitmap Index Scan on posts_id_article_idx
                        (cost=0.00..19.78 rows=979 width=0)
                        (actual time=0.655..0.655 rows=1011 loops=1)
                    Index Cond: (id_article = 1907)
                    Buffers: shared read=5
                    I/O Timings: read=0.033
Planning time: 0.242 ms
Execution time: 4.766 ms
```

Le problème de ce plan est que, plus le jeu de résultat sera important, plus les temps de réponse seront importants. Ils seront encore plus importants si le tri déclenche un tri sur disque. Il faut donc trouver une solution pour les minimiser.

Les problèmes de l'utilisation de la clause `OFFSET` sont parfaitement expliqués sur cet article[33] . Le TP qui suit ce chapitre propose d'implémenter les solutions proposées dans l'article par Markus Winand.

[33] http://use-the-index-luke.com/fr/no-offset

5.2 RETURNING

- Clause RETURNING
- Utilité :
  - récupérer les enregistrements modifiés
  - avec INSERT
  - avec UPDATE
  - avec DELETE

La clause RETURNING permet de récupérer les valeurs modifiées par un ordre de modification. Ainsi, la clause RETURNING associée à l'ordre INSERT permet d'obtenir une ou plusieurs colonnes des lignes insérées.

5.2.1 RETURNING : EXEMPLE

```
CREATE TABLE test_returning (id serial primary key, val integer);

INSERT INTO test_returning (val)
  VALUES (10)
RETURNING id, val;

 id | val
----+-----
  1 |  10
(1 ligne)
```

Cela permet par exemple de récupérer la valeur de colonnes portant une valeur par défaut, comme la valeur affectée par une séquence, comme sur l'exemple ci-dessus.

La clause RETURNING permet également de récupérer les valeurs des colonnes mises à jour :

```
UPDATE test_returning
   SET val = val + 10
 WHERE id = 1
 RETURNING id, val;
 id | val
----+-----
  1 |  20
(1 ligne)
```

Associée à l'ordre DELETE, il est possible d'obtenir les lignes supprimées :

```
DELETE FROM test_returning
 WHERE val < 30
 RETURNING id, val;
 id | val
----+-----
  1 |  20
(1 ligne)
```

5.3 UPSERT

- INSERT ou UPDATE ?
  - `INSERT ... ON CONFLICT DO { NOTHING | UPDATE }`
  - À partir de la version 9.5
- Utilité :
  - mettre à jour en cas de conflit sur un INSERT
  - ne rien faire en cas de conflit sur un INSERT

L'implémentation de l'UPSERT peut poser des questions sur la concurrence d'accès. L'implémentation de PostgreSQL de `ON CONFLICT DO UPDATE` est une opération atomique, c'est à dire que PostgreSQL garantit qu'il n'y aura pas de conditions d'exécution qui pourront amener à des erreurs. L'utilisation d'une contrainte d'unicité n'est pas étrangère à cela, elle permet en effet de pouvoir vérifier que la ligne n'existe pas, et si elle existe déjà, de verrouiller la ligne à mettre à jour de façon atomique.

En comparaison, plusieurs approches naïves présentent des problèmes de concurrences d'accès. Les différentes approches sont décrites dans cet article de depesz[34] . Elle présente toutes des problèmes de *race conditions* qui peuvent entraîner des erreurs. Une autre possibilité aurait été d'utiliser une CTE en écriture, mais elle présente également les problèmes de concurrence d'accès décrits dans l'article.

Sur des traitements d'intégration de données, il s'agit d'un comportement qui n'est pas toujours souhaitable. La norme SQL propose l'ordre `MERGE` pour palier à des problèmes de ce type, mais il est peu probable de le voir un jour implémenté dans PostgreSQL[35]. L'ordre `INSERT` s'est toutefois vu étendu avec PostgreSQL 9.5 pour gérer les conflits à l'insertion.

Les exemples suivants s'appuient sur le jeu de données suivant :

[34] http://www.depesz.com/2012/06/10/why-is-upsert-so-complicated/

[35] La solution actuelle semble techniquement meilleure et la solution actuelle a donc été choisie. Le wiki du projet PostgreSQL montre que l'ordre `MERGE` a été étudié et qu'un certains nombres d' aspects cruciaux n'ont pas été spécifiés, amenant le projet PostgreSQL a utiliser sa propre version. Voir la documentation : https://wiki.postgresql.org/wiki/UPSERT#MERGE_disadvantages.

```
\d employes
       Table "public.employes"
  Column   |     Type      | Modifiers
-----------+---------------+-----------
 matricule | character(8)  | not null
 nom       | text          | not null
 service   | text          |
 salaire   | numeric(7,2)  |
Indexes:
    "employes_pkey" PRIMARY KEY, btree (matricule)

SELECT * FROM employes ;
 matricule |   nom    |   service    | salaire
-----------+----------+--------------+----------
 00000001  | Dupuis   | Direction    | 10000.00
 00000004  | Fantasio | Courrier     |  4500.00
 00000006  | Prunelle | Publication  |  4000.00
 00000020  | Lagaffe  | Courrier     |  3000.00
 00000040  | Lebrac   | Publication  |  3000.00
(5 lignes)
```

5.3.1 UPSERT : PROBLÈME À RÉSOUDRE

- Insérer une ligne déjà existante provoque une erreur :

```
INSERT INTO employes (matricule, nom, service, salaire)
 VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00);
ERROR:  duplicate key value violates unique constraint
        "employes_pkey"
DETAIL:  Key (matricule)=(00000001) already exists.
```

Si l'on souhaite insérer une ligne contenant un matricule déjà existant, une erreur de clé dupliquée est levée et le reste de la transaction est annulé.

5.3.2 ON CONFLICT DO NOTHING

- la clause `ON CONFLICT DO NOTHING` évite d'insérer une ligne existante :

```
=# INSERT INTO employes (matricule, nom, service, salaire)
   VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00)
   ON CONFLICT DO NOTHING;
INSERT 0 0
```

Les données n'ont pas été modifiées :

```
=# SELECT * FROM employes ;
 matricule |   nom     |   service    | salaire
-----------+-----------+--------------+----------
 00000004  | Fantasio | Courrier     |  4500.00
 00000006  | Prunelle | Publication  |  4000.00
 00000020  | Lagaffe  | Courrier     |  3000.00
 00000040  | Lebrac   | Publication  |  3000.00
 00000001  | Dupuis   | Direction    | 10000.00
(5 rows)
```

5.3.3 ON CONFLICT DO NOTHING : SYNTAXE

```
INSERT ....
ON CONFLICT
  DO NOTHING;
```

Il suffit d'indiquer à PostgreSQL de ne rien faire en cas de conflit sur une valeur dupliquée avec la clause `ON CONFLICT DO NOTHING` placée à la fin de l'ordre INSERT qui peut poser problème.

Dans ce cas, si une rupture d'unicité est détectée, alors PostgreSQL ignorera l'erreur, silencieusement. En revanche, si une erreur apparaît sur une autre contrainte, l'erreur sera levée.

En prenant l'exemple suivant :

```
CREATE TABLE test_upsert (
  i serial PRIMARY KEY,
  v text UNIQUE,
  x integer CHECK (x > 0)
);

INSERT INTO test_upsert (v, x) VALUES ('x', 1);
```

L'insertion d'une valeur dupliquée provoque bien une erreur d'unicité :

```
INSERT INTO test_upsert (v, x) VALUES ('x', 1);
ERROR:  duplicate key value violates unique constraint "test_upsert_v_key"
```

L'erreur d'unicité est bien ignorée si la ligne existe déjà, le résultat est `INSERT 0 0` qui indique qu'aucune ligne n'a été insérée :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 1)
 ON CONFLICT DO NOTHING;
INSERT 0 0
```

L' insertion est aussi ignorée si l'on tente d'insérer des lignes rompant la contrainte d'unicité mais ne comportant pas les mêmes valeurs pour d'autres colonnes :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 4)
 ON CONFLICT DO NOTHING;
INSERT 0 0
```

Si l'on insère une valeur interdite par la contrainte `CHECK`, une erreur est bien levée :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 0)
 ON CONFLICT DO NOTHING;
ERROR:  new row for relation "test_upsert" violates check constraint
        "test_upsert_x_check"
DETAIL:  Failing row contains (4, x, 0).
```

5.3.4 ON CONFLICT DO UPDATE

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'M. Pirate', 'Direction', 0.00)
ON CONFLICT (matricule)
   DO UPDATE SET salaire = employes.salaire,
                 nom = excluded.nom
RETURNING *;

 matricule |    nom    |   service   | salaire
-----------+-----------+-------------+----------
 00000001  | M. Pirate | Direction   | 50000.00
```

La clause `ON CONFLICT` permet de déterminer une colonne sur laquelle le conflit peut arriver. Cette colonne ou ces colonnes doivent porter une contrainte d'unicité ou une contrainte d'exclusion, c'est à dire une contrainte portée par un index. La clause `DO UPDATE` associée fait référence aux valeurs rejetées par le conflit à l'aide de la pseudo-table `excluded`. Les valeurs courantes sont accessibles en préfixant les colonnes avec le nom de la table. L'exemple montre cela.

Avec la requête de l'exemple, on voit que le salaire du directeur n'a pas été modifié, mais son nom l'a été :

```
SELECT * FROM employes ;
 matricule |    nom    |   service   | salaire
-----------+-----------+-------------+----------
 00000004  | Fantasio  | Courrier    |  4500.00
 00000006  | Prunelle  | Publication |  4000.00
 00000020  | Lagaffe   | Courrier    |  3000.00
 00000040  | Lebrac    | Publication |  3000.00
 00000001  | M. Pirate | Direction   | 10000.00
(5 rows)
```

La clause `ON CONFLICT` permet également de définir une constrainte d'intégrité sur laquelle on réagit en cas de conflit :

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00)
ON CONFLICT ON CONSTRAINT employes_pkey
   DO UPDATE SET salaire = excluded.salaire;
```

On remarque que seul le salaire du directeur a changé :

```
SELECT * FROM employes ;
 matricule |   nom     |   service   | salaire
-----------+-----------+-------------+----------
 00000004  | Fantasio  | Courrier    |  4500.00
 00000006  | Prunelle  | Publication |  4000.00
 00000020  | Lagaffe   | Courrier    |  3000.00
 00000040  | Lebrac    | Publication |  3000.00
 00000001  | M. Pirate | Direction   | 50000.00
(5 rows)
```

5.3.5 ON CONFLICT DO UPDATE

- Avec plusieurs lignes insérées :

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000002', 'Moizelle Jeanne', 'Publication', 3000.00),
       ('00000040', 'Lebrac', 'Publication', 3100.00)
ON CONFLICT (matricule)
   DO UPDATE SET salaire = employes.salaire,
                 nom = excluded.nom
RETURNING *;

 matricule |       nom       |   service   | salaire
```

```
-----------+-----------------+-------------+----------
 00000002  | Moizelle Jeanne | Publication |  3000.00
 00000040  | Lebrac          | Publication |  3000.00
```

Bien sûr, on peut insérer plusieurs lignes, INSERT ON CONFLICT réagira uniquement sur les doublons :

La nouvelle employée, *Moizelle Jeanne* a été intégrée dans la tables des employés, et *Lebrac* a été traité comme un doublon, en appliquant la règle de mise à jour vue plus haut : seul le nom est mis à jour et le salaire est inchangé.

```
SELECT * FROM employes ;
 matricule |       nom       |   service   | salaire
-----------+-----------------+-------------+----------
 00000004  | Fantasio        | Courrier    |  4500.00
 00000006  | Prunelle        | Publication |  4000.00
 00000020  | Lagaffe         | Courrier    |  3000.00
 00000001  | M. Pirate       | Direction   | 50000.00
 00000002  | Moizelle Jeanne | Publication |  3000.00
 00000040  | Lebrac          | Publication |  3000.00
(6 rows)
```

À noter que la clause `SET salaire = employes.salaire` est inutile, c'est ce que fait PostgreSQL implicitement.

5.3.6 ON CONFLICT DO UPDATE : SYNTAXE

- colonne(s) portant(s) une contrainte d'unicité
- pseudo-table *excluded*

```
INSERT ....
ON CONFLICT (<colonne clé>)
  DO UPDATE
        SET colonne_a_modifier = excluded.colonne,
            autre_colonne_a_modifier = excluded.autre_colonne,
            ...;
```

Si l'on choisit de réaliser une mise à jour plutôt que de générer une erreur, on utilisera la clause `ON CONFLICT DO UPDATE`. Il faudra dans ce cas préciser la ou les colonnes qui portent une contrainte d'unicité. Cette contrainte d'unicité permettra de détecter la duplication de valeur, PostgreSQL pourra alors appliquer la règle de mise à jour édictée.

La règle de mise à jour permet de définir très finement quelles sont les colonnes que l'on met à jour et quelles sont les colonnes que l'on ne met pas à jour. Dans ce contexte, la pseudo-table *excluded* représente l'ensemble rejeté par l'`INSERT`. Il faudra explicitement

indiquer les colonnes dont la valeur sera mise à jour à partir des valeurs que l'on tente d'insérer, reprise de la pseudo-table *excluded* :

```
ON CONFLICT (...)
  DO UPDATE
  SET colonne = excluded.colonne,
      autre_colonne = excluded.autre_colonne,
      ...
```

En alternative, il est possible d'indiquer un nom de contrainte plutôt que le nom d'une colonne portant une contrainte d'unicité :

```
INSERT ....
ON CONFLICT ON CONSTRAINT nom_contrainte
  DO UPDATE
        SET colonne_a_modifier = excluded.colonne,
            autre_colonne_a_modifier = excluded.autre_colonne,
            ...;
```

De plus amples informations quant à la syntaxe sont disponibles dans la documentation[36]
.

5.4 LATERAL

- Jointures `LATERAL`
  - SQL:99
  - PostgreSQL 9.3
  - équivalent d'une boucle `foreach`
- Utilisations
  - top-N à partir de plusieurs tables
  - jointure avec une fonction retournant un ensemble

`LATERAL` apparaît dans la révision de la norme SQL de 1999. Elle permet d'appliquer une requête ou une fonction sur le résultat d'une table.

5.4.1 LATERAL : AVEC UNE SOUS-REQUÊTE

- Jointure LATERAL
  - équivalent de *foreach*
- Utilité :

[36] http://www.postgresql.org/docs/9.5/static/sql-insert.html

- Top-N à partir de plusieurs tables
- exemple : *afficher les 5 derniers messages des 5 derniers sujets actifs d'un forum*

La clause LATERAL existe dans la norme SQL depuis plusieurs années. L'implémentation de cette clause dans la plupart des SGBD reste cependant relativement récente.

Elle permet d'utiliser les données de la requête principale dans une sous-requête. La sous-requête sera appliquée à chaque enregistrement retourné par la requête principale.

5.4.2 LATERAL : EXEMPLE

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets,
  LATERAL(SELECT date_publication,
                 substr(message, 0, 100) AS extrait
            FROM messages
           WHERE sujets.sujet_id = messages.sujet_id
           ORDER BY date_publication DESC
           LIMIT 5) top_5_messages
 ORDER BY sujets.date_modification DESC,
          top_5_messages.date_publication DESC
 LIMIT 25;
```

L'exemple ci-dessus montre comment afficher les 5 derniers messages postés sur les 5 derniers sujets actifs d'un forum avec la clause LATERAL.

Une autre forme d'écriture emploie le mot clé JOIN, inutile dans cet exemple. Il peut avoir son intérêt si l'on utilise une jointure externe (LEFT JOIN par exemple si un sujet n'impliquait pas forcément la présence d'un message) :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets
  JOIN LATERAL(SELECT date_publication, substr(message, 0, 100) AS extrait
            FROM messages
           WHERE sujets.sujet_id = messages.sujet_id
           ORDER BY date_publication DESC
           LIMIT 5) top_5_messages
    ON (true) -- condition de jointure toujours vraie
 ORDER BY sujets.date_modification DESC, top_5_messages.date_publication DESC
 LIMIT 25;
```

Il aurait été possible de réaliser cette requête par d'autres moyens, mais LATERAL permet d'obtenir la requête la plus performante. Une autre approche quasiment aussi performante

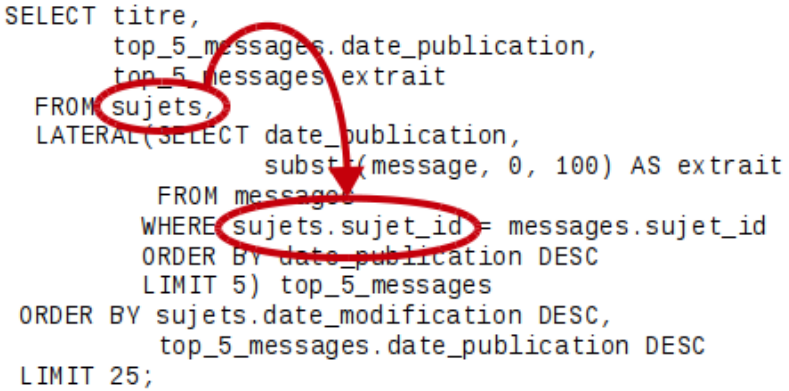
aurait été de faire appel à une fonction retournant les 5 enregistrements souhaités.

À noter qu'une colonne `date_modification` a été ajouté à la table `sujets` afin de déterminer rapidement les derniers sujets modifiés. Sans cela, il faudrait parcourir l'ensemble des sujets, récupérer la date de publication des derniers messages avec une jointure LATERAL et récupérer les 5 derniers sujets actifs. Cela nécessite de lire beaucoup de données. Un trigger positionné sur la table `messages` permettra d'entretenir la colonne `date_modification` sur la table `sujets` sans difficulté. Il s'agit donc ici d'une entorse aux règles de modélisation en vue d'optimiser les traitements.

Un index sur les colonnes sujet_id et date_publication permettra de minimiser les accès pour cette requête :

```
CREATE INDEX ON messages (sujet_id, date_publication DESC);
```

5.4.3 LATERAL : PRINCIPE



Si nous n'avions pas la clause `LATERAL`, nous pourrions être tentés d'écrire la requête suivante :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets
  JOIN (SELECT date_publication, substr(message, 0, 100) AS extrait
          FROM messages
         WHERE sujets.sujet_id = messages.sujet_id
         ORDER BY date_message DESC
         LIMIT 5) top_5_messages
```

```
 ORDER BY sujets.date_modification DESC
 LIMIT 25;
```

Cependant, la norme SQL interdit une telle construction, il n 'est pas possible de référencer la table principale dans une sous-requête. Mais avec la clause LATERAL, la sous-requête peut faire appel à la table principale.

5.4.4 LATERAL : AVEC UNE FONCTION

- Utilisation avec une fonction retournant un ensemble
  - clause LATERAL optionnelle
- Utilité :
  - extraire les données d'un tableau ou d'une structure JSON sous la forme tabulaire
  - utiliser une fonction métier qui retourne un ensemble X selon un ensemble Y fourni

L'exemple ci-dessous montre qu'il est possible d'utiliser une fonction retournant un ensemble (SRF pour *Set Returning Functions*).

5.4.5 LATERAL : EXEMPLE AVEC UNE FONCTION

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets,
       get_top_5_messages(sujet_id) AS top_5_messages
 ORDER BY sujets.date_modification DESC
 LIMIT 25;
```

La fonction get_top_5_messages est la suivante :

```
CREATE OR REPLACE FUNCTION get_top_5_messages (p_sujet_id integer)
RETURNS TABLE (date_publication timestamp, extrait text)
AS $PROC$
BEGIN
  RETURN QUERY SELECT date_publication, substr(message, 0, 100) AS extrait
    FROM messages
   WHERE messages.sujet_id = p_sujet_id
   ORDER BY date_publication DESC
   LIMIT 5;
```

```
END;
$PROC$ LANGUAGE plpgsql;
```

La clause LATERAL n'est pas obligatoire, mais elle s'utiliserait ainsi : :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets, LATERAL get_top_5_messages(sujet_id) AS top_5_messages
 ORDER BY sujets.date_modification DESC LIMIT 25;
```

5.5 COMMON TABLE EXPRESSIONS

- Common Table Expressions
  - clauses WITH et WITH RECURSIVE
- Utilité :
  - factoriser des sous-requêtes

5.5.1 CTE ET SELECT

- Utilité
  - factoriser des sous-requêtes
  - améliorer la lisibilité d'une requête

Les CTE permettent de factoriser la définition d'une sous-requête qui pourrait être appelée plusieurs fois.

Une CTE est exprimée avec la clause WITH. Cette clause permet de définir des vues éphémères qui seront utilisées les unes après les autres et au final utilisées dans la requête principale.

5.5.2 CTE ET SELECT : EXEMPLE

```
WITH resultat AS (
   /* requête complexe */
)
SELECT *
  FROM resultat
 WHERE nb < 5;
```

On utilise principalement une CTE pour factoriser la définition d'une sous-requête commune, comme dans l'exemple ci-dessus.

Un autre exemple un peu plus complexe :

```
WITH resume_commandes AS (
SELECT c.numero_commande, c.client_id, quantite*prix_unitaire AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
)
SELECT type_client, NULL AS pays, SUM(montant) AS montant_total_commande
  FROM resume_commandes
  JOIN clients
    ON (resume_commandes.client_id = clients.client_id)
 GROUP BY type_client
UNION ALL
SELECT NULL, code_pays AS pays, SUM(montant)
  FROM resume_commandes r
  JOIN clients cl
    ON (r.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 GROUP BY code_pays;
```

Le plan d'exécution de la requête montre que la vue resume_commandes est exécutée une seule fois et son résultat est utilisé par les deux opérations de regroupements définies dans la requête principale :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=244618.50..323855.66 rows=12 width=67)
   CTE resume_commandes
     ->  Hash Join  (cost=31886.90..174241.18 rows=1216034 width=26)
           Hash Cond: (l.numero_commande = c.numero_commande)
           ->  Seq Scan on lignes_commandes l
                 (cost=0.00..73621.47 rows=3141947 width=18)
           ->  Hash  (cost=25159.00..25159.00 rows=387032 width=16)
                 ->  Seq Scan on commandes c
                       (cost=0.00..25159.00 rows=387032 width=16)
                     Filter: ((date_commande >= '2014-01-01'::date)
                             AND (date_commande <= '2014-12-31'::date))
   ->  HashAggregate  (cost=70377.32..70377.36 rows=3 width=34)
         Group Key: clients.type_client
         ->  Hash Join  (cost=3765.00..64297.15 rows=1216034 width=34)
               Hash Cond: (resume_commandes.client_id = clients.client_id)
```

```
            ->  CTE Scan on resume_commandes
                    (cost=0.00..24320.68 rows=1216034 width=40)
            ->  Hash  (cost=2026.00..2026.00 rows=100000 width=10)
                  ->  Seq Scan on clients
                        (cost=0.00..2026.00 rows=100000 width=10)
   ->  HashAggregate  (cost=79236.89..79237.00 rows=9 width=35)
         Group Key: co.code_pays
         ->  Hash Join  (cost=12624.57..73156.72 rows=1216034 width=35)
               Hash Cond: (r.client_id = cl.client_id)
               ->  CTE Scan on resume_commandes r
                     (cost=0.00..24320.68 rows=1216034 width=40)
               ->  Hash  (cost=10885.57..10885.57 rows=100000 width=11)
                     ->  Hash Join
                           (cost=3765.00..10885.57 rows=100000 width=11)
                           Hash Cond: (co.contact_id = cl.contact_id)
                           ->  Seq Scan on contacts co
                                 (cost=0.00..4143.05 rows=110005 width=11)
                           ->  Hash (cost=2026.00..2026.00 rows=100000 width=16)
                                 ->  Seq Scan on clients cl
                                       (cost=0.00..2026.00 rows=100000 width=16)
```

Si la requête avait été écrite sans CTE, donc en exprimant deux fois la même sous-requête, le coût d'exécution aurait été multiplié par deux car il aurait fallu exécuter la sous-requête deux fois au lieu d'une.

On utilise également les CTE pour améliorer la lisibilité des requêtes complexes, mais cela peut poser des problèmes d'optimisations, comme cela sera discuté plus bas.

5.5.3 CTE ET SELECT : SYNTAXE

```
WITH nom_vue1 AS (
 <requête pour générer la vue 1>
)
SELECT *
  FROM nom_vue1;
```

La syntaxe de définition d'une vue est donnée ci-dessus.

On peut néanmoins enchaîner plusieurs vues les unes à la suite des autres :

```
WITH nom_vue1 AS (
 <requête pour générer la vue 1>
), nom_vue2 AS (
 <requête pour générer la vue 2, pouvant utiliser la vue 1>
```

```
)
<requête principale utilisant vue 1 et/ou vue2>;
```

5.5.4 CTE ET BARRIÈRE D'OPTIMISATION

- Attention, une CTE est une barrière d'optimisation
  - pas de transformations
  - pas de propagation des prédicats

Il faut néanmoins être vigilant car l'optimiseur n'inclut pas la définition des CTE dans la requête principale quand il réalise les différentes passes d'optimisations.

Par exemple, si un prédicat appliqué dans la requête principale peut être remonté au niveau d'une sous-requête, l'optimiseur de PostgreSQL le réalisera :

```
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 considérer dès le départ. En anglais, on parle de *predicate push-down*.

Une requête similaire, mais basée sur une CTE ne permet pas d'appliquer le filtre au plus tôt comme dans le cas d'une sous-requête :

```
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)
```

Ce mécanisme permet néanmoins de contourner certaines limitations de l'optimiseur de PostgreSQL en vue de contrôler précisément le plan d'exécution d'une requête. Ce principe de fonctionnement pourra être amené à changer dans des versions futures de PostgreSQL.

5.5.5 CTE EN ÉCRITURE

- CTE avec des requêtes en modification
  - avec `INSERT`/`UPDATE`/`DELETE`
  - et éventuellement `RETURNING`
  - obligatoirement exécuté sur PostgreSQL
- Exemple d'utilisation :
  - archiver des données
  - partitionner les données d'une table
  - débugger une requête complexe

5.5.6 CTE EN ÉCRITURE : EXEMPLE

```
WITH donnees_a_archiver AS (
DELETE FROM donnes_courantes
 WHERE date < '2015-01-01'
 RETURNING *
)
INSERT INTO donnes_archivees
SELECT * FROM donnees_a_archiver;
```

La requête d'exemple permet d'archiver des données dans une table dédiée à l'archivage en utilisant une CTE en écriture. L'emploi de la clause `RETURNING` permet de récupérer les lignes purgées.

Le même principe s'applique pour une table que l'on vient de partitionner. Les enregistrements se trouvent initialement dans la table mère, il faut les répartir sur les différentes partitions. On utilisera une requête reposant sur le même principe que la précédente. L'ordre INSERT visera la table principale si l'on souhaite utiliser le trigger de partition pour répartir les données. Il pourra également viser une partition donnée afin d'éviter le surcoût du trigger de partition.

En plus de ce cas d'usage simple, il est possible d'utiliser cette fonctionnalité pour débugger une requête complexe.

```
WITH sous-requete1 AS (

),
debug_sous-requete1 AS (
INSERT INTO debug_sousrequete1
SELECT * FROM sous-requete1
), sous-requete2 AS (
SELECT ...
  FROM sous-requete1
  JOIN ....
 WHERE ....
 GROUP BY ...
),
debug_sous-requete2 AS (
INSERT INTO debug_sousrequete2
SELECT * FROM sous-requete2
)
SELECT *
  FROM sous-requete2;
```

On peut également envisager une requête CTE en écriture pour émuler une requête MERGE pour réaliser une intégration de données complexe, là où l'UPSERT ne serait pas suffisant. Il faut toutefois avoir à l'esprit qu'une telle requête présente des problèmes de concurrences d'accès, pouvant entraîner des résultats inattendus si elle est employée alors que d'autres sessions modifient les données. On se contentera d'utiliser une telle requête dans des traitements batchs.

Il est important de noter que sur PostgreSQL, chaque sous-partie d'une CTE qui exécute une opération de mise à jour sera exécutée, même si elle n'est pas explicitement appelé. Par exemple :

```
WITH del AS (DELETE FROM nom_table),
fonction_en_ecriture AS (SELECT * FROM fonction_en_ecriture())
SELECT 1;
```

supprimera l'intégralité des données de la table nom_table, mais n'appellera pas la fonction `fonction_en_ecriture()`, même si celle-ci effectue des écritures.

5.5.7 CTE RÉCURSIVE

- SQL permet d'exprimer des récursions
  - WITH RECURSIVE
- Utilité :

- récupérer une arborescence de menu hiérarchique
- parcourir des graphes (réseaux sociaux, etc.)

Le langage SQL permet de réaliser des récursions avec des CTE récursives. Son principal intérêt est de pouvoir parcourir des arborescences, comme par exemple des arbres généalogiques, des arborescences de service ou des entrées de menus hiérarchiques.

Il permet également des réaliser des parcours de graphes, mais les possibilités de SQL sont plus limitées de ce côté là. En effet, SQL utilise un algorithme de type "Breadth First" (parcours en largeur) où PostgreSQL produit tout le niveau courant, et approfondit ensuite la récursion. Ce fonctionnement est à l'opposé d'un algorithme "Depth First" (parcours en profondeur) où chaque branche est explorée à fond individuellement avant de passer à la branche suivante. Ce principe de fonctionnement de l'implémentation dans SQL peut poser des problèmes sur des recherches de types réseaux sociaux où des bases de données orientées graphes, tel que Neo4J, seront bien plus efficaces. À noter que l'extension pgRouting implémente des algorithmes de parcours de graphes plus efficace, cela permet de rester dans PostgreSQL mais nécessite un certain formalisme et il faut avoir conscience que pgRouting n'est pas l'outil le plus efficace car il génère un graphe en mémoire à chaque requête à résoudre et ce graphe généré en mémoire est perdu après l'appel.

5.5.8 CTE RÉCURSIVE : EXEMPLE (1/2)

```
WITH RECURSIVE suite AS (
SELECT 1 AS valeur
UNION ALL
SELECT valeur + 1
  FROM suite
 WHERE valeur < 10
)
SELECT * FROM suite;
```

Voici le résultat de cette requête :

```
valeur
--------
      1
      2
      3
      4
      5
```

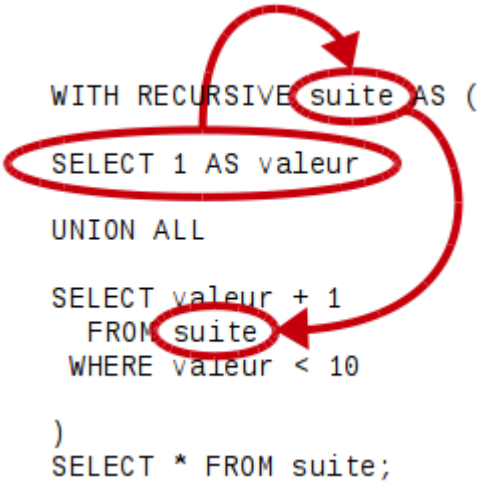
```
      6
      7
      8
      9
     10
(10 rows)
```

L'exécution de cette requête commence avec le `SELECT 1 AS valeur` (la requête avant le `UNION ALL`), d'où la première ligne avec la valeur 1. Puis PostgreSQL exécute le `SELECT valeur+1 FROM suite WHERE valeur < 10` tant que cette requête renvoie des lignes. À la première exécution, il additionne 1 avec la valeur précédente (1), ce qui fait qu'il renvoie 2. A la deuxième exécution, il additionne 1 avec la valeur précédente (2), ce qui fait qu'il renvoie 3. Etc. La récursivité s'arrête quand la requête ne renvoie plus de ligne, autrement dit quand la colonne vaut 10.

Cet exemple n'a aucun autre intérêt que de présenter la syntaxe permettant de réaliser une récursion en langage SQL.

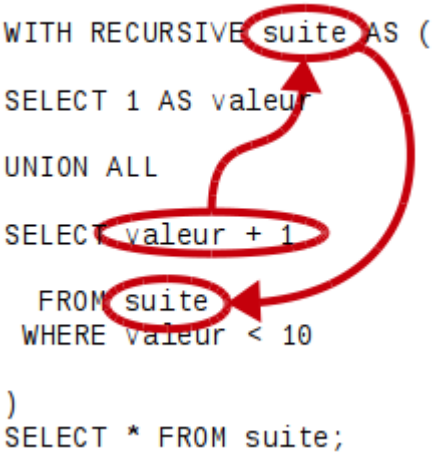
5.5.9 CTE RÉCURSIVE : PRINCIPE

- 1ère étape : initialisation de la récursion



5.5.10 CTE RÉCURSIVE : PRINCIPE

- récursion : la requête s'appelle elle-même



5.5.11 CTE RÉCURSIVE : EXEMPLE (2/2)

```
WITH RECURSIVE parcours_menu AS (
SELECT menu_id, libelle, parent_id,
       libelle AS arborescence
  FROM entrees_menu
 WHERE libelle = 'Terminal'
   AND parent_id IS NULL
UNION ALL
SELECT menu.menu_id, menu.libelle, menu.parent_id,
       arborescence || '/' || menu.libelle
  FROM entrees_menu menu
  JOIN parcours_menu parent
    ON (menu.parent_id = parent.menu_id)
)
SELECT * FROM parcours_menu;
```

17.12

Cet exemple suivant porte sur le parcours d'une arborescence de menu hiérarchique.

Une table entrees_menu est crée :

```
CREATE TABLE entrees_menu (menu_id serial primary key, libelle text not null,
                           parent_id integer);
```

Elle dispose du contenu suivant :

```
SELECT * FROM entrees_menu;
 menu_id |          libelle           | parent_id
---------+----------------------------+-----------
       1 | Fichier                    |
       2 | Edition                    |
       3 | Affichage                  |
       4 | Terminal                   |
       5 | Onglets                    |
       6 | Ouvrir un onglet           |         1
       7 | Ouvrir un terminal         |         1
       8 | Fermer l'onglet            |         1
       9 | Fermer la fenêtre          |         1
      10 | Copier                     |         2
      11 | Coller                     |         2
      12 | Préférences                |         2
      13 | Général                    |        12
      14 | Apparence                  |        12
      15 | Titre                      |        13
      16 | Commande                   |        13
      17 | Police                     |        14
      18 | Couleur                    |        14
      19 | Afficher la barre d'outils |         3
      20 | Plein écran                |         3
      21 | Modifier le titre          |         4
      22 | Définir l'encodage         |         4
      23 | Réinitialiser              |         4
      24 | UTF-8                      |        22
      25 | Europe occidentale         |        22
      26 | Europe centrale            |        22
      27 | ISO-8859-1                 |        25
      28 | ISO-8859-15                |        25
      29 | WINDOWS-1252               |        25
      30 | ISO-8859-2                 |        26
      31 | ISO-8859-3                 |        26
      32 | WINDOWS-1250               |        26
      33 | Onglet précédent           |         5
      34 | Onglet suivant             |         5
(34 rows)
```

Nous allons définir une CTE récursive qui va afficher l'arborescence du menu *Terminal*. La récursion va donc commencer par chercher la ligne correspondant à cette entrée de menu dans la table `entrees_menu`. Une colonne calculée arborescence est créée, elle servira plus tard dans la récursion :

```
SELECT menu_id, libelle, parent_id, libelle AS arborescence
  FROM entrees_menu
 WHERE libelle = 'Terminal'
   AND parent_id IS NULL
```

La requête qui réalisera la récursion est une jointure entre le résultat de l'itération précédente, obtenu par la vue `parcours_menu` de la CTE, qui réalisera une jointure avec la table `entrees_menu` sur la colonne entrees_ menu.parent_id qui sera jointe à la colonne menu_id de l'itération précédente.

La condition d'arrêt de la récursion n'a pas besoin d'être exprimée. En effet, les entrées terminales des menus ne peuvent pas être jointe avec de nouvelles entrées de menu, car il n'y a pas d'autre correspondance avec `parent_id`).

On obtient ainsi la requête CTE récursive présentée ci-dessus.

À titre d'exemple, voici l'implémentation du jeu des six degrés de Kevin Bacon en utilisant pgRouting :

```
WITH dijkstra AS (
SELECT seq, id1 AS node, id2 AS edge, cost
  FROM pgr_dijkstra('
  SELECT f.film_id AS id,
         f.actor_id::integer AS source,
         f2.actor_id::integer AS target,
         1.0::float8 AS cost
    FROM film_actor f
    JOIN film_actor f2
      ON (f.film_id = f2.film_id and f.actor_id <> f2.actor_id)'
  , 29539, 29726, false, false)
)
SELECT *
  FROM actors
  JOIN dijkstra
  on (dijkstra.node = actors.actor_id) ;

 actor_id |   actor_name    | seq | node  | edge | cost
----------+-----------------+-----+-------+------+------
    29539 | Kevin Bacon     |   0 | 29539 | 1330 |    1
    29625 | Robert De Niro  |   1 | 29625 |   53 |    1
    29726 | Al Pacino       |   2 | 29726 |   -1 |    0
(3 lignes)
```

5.6 CONCURRENCE D'ACCÈS

- Problèmes pouvant se poser :
  - UPDATE perdu
  - lecture non répétable
- Plusieurs solutions possibles
  - versionnement des lignes
  - SELECT FOR UPDATE
  - SERIALIZABLE

Plusieurs problèmes de concurrences d'accès peuvent se poser quand plusieurs transactions modifient les mêmes données en même temps.

Tout d'abord, des UPDATE peuvent être perdus, dans le cas où plusieurs transactions lisent la même ligne, puis la mettent à jour sans concertation. Par exemple, si la transaction 1 ouvre une transaction et effectue une lecture d 'une ligne donnée :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '0000004';
```

La transaction 2 effectue les mêmes traitements :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '0000004';
```

Après un traitement applicatif, la transaction 1 met les données à jour pour noter l'augmentation de 5% du salarié. La transaction est validée dans la foulée avec `COMMIT` :

```
UPDATE employes
   SET salaire = <valeur récupérée préalablement * 1.05>
 WHERE matricule = '00000004';
COMMIT;
```

Après un traitement applicatif, la transaction 2 met également les données à jour pour noter une augmentation exceptionnelle de 100 € :

```
UPDATE employes
   SET salaire = <valeur récupérée préalablement + 100>
 WHERE matricule = '00000004';
COMMIT;
```

Le salarié a normalement droit à son augmentation de 100 € ET l'augmentation de 5%, or l'augmentation de 5% a été perdue car écrasée par la transaction n°2. Ce problème aurait pu être évité de trois façons différentes : * en effectuant un `UPDATE` utilisant la valeur

lue par l'ordre `UPDATE`, * en verrouillant les données lues avec `SELECT FOR UPDATE`, * en utilisant le niveau d'isolation `SERIALIZABLE`.

La première solution n'est pas toujours envisageable, il faut donc se tourner vers les deux autres solutions.

Le problème des lectures sales (*dirty reads*) ne peut pas se poser car PostgreSQL n'implémente pas le niveau d'isolation `READ UNCOMMITTED`. Si ce niveau d'isolation est sélectionné, PostgreSQL utilise alors le niveau `READ COMMITTED`.

5.6.1 SELECT FOR UPDATE

- `SELECT FOR UPDATE`
- Utilité :
  - "réserver" des lignes en vue de leur mise à jour
  - éviter les problèmes de concurrence d'accès

L'ordre `SELECT FOR UPDATE` permet de lire des lignes tout en les réservant en posant un verrou dessus en vue d'une future mise à jour. Le verrou permettra une lecture parallèle, mais mettra toute mise à jour en attente.

Reprenons l'exemple précédent et utilisons `SELECT FOR UPDATE` pour voir si le problème de concurrence d'accès peut être résolu.

session 1

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE;
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

La requête SELECT a retourné les données souhaitées.

session 2

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE;
```

La requête SELECT ne rend pas la main, elle est mise en attente.

session 3

Une troisième session effectue une lecture, sans poser de verrou explicite :

```
SELECT * FROM employes WHERE matricule = '00000004';
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

Le SELECT n'a pas été bloqué par la session 1. Seule la session 2 est bloquée car elle tente d'obtenir le même verrou.

session 1

L'application a effectué ses calculs et met à jour les données en appliquant l'augmentation de 5% :

```
UPDATE employes
   SET salaire = 4725
 WHERE matricule = '00000004';
```

Les données sont vérifiées :

```
SELECT * FROM employes WHERE matricule = '00000004';
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4725.00
(1 row)
```

Enfin, la transaction est validée :

```
COMMIT;
```

session 2

La session 2 a rendu la main, le temps d'attente a été important pour réaliser ces calculs complexes :

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4725.00
(1 row)

Time: 128127,105 ms
```

Le salaire obtenu est bien le salaire mis à jour par la session 1. Sur cette base, l'application applique l'augmentation de 100 € :

```
UPDATE employes
   SET salaire = 4825.00
 WHERE matricule = '00000004';

SELECT * FROM employes WHERE matricule = '00000004';
 matricule |   nom    | service  | salaire
```

```
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4825.00
```

La transaction est validée :

```
COMMIT;
```

Les deux transactions ont donc été effectuée de manière sérialisée, l'augmentation de 100 € ET l'augmentation de 5% ont été accordée à Fantasio. En contre-partie, l'une des deux transactions concurrentes a été mise en attente afin de pouvoir sérialiser les transactions. Cela implique de penser les traitements en verrouillant les ressources auxquelles on souhaite accéder.

L'ordre `SELECT FOR UPDATE` dispose également d'une option `NOWAIT` qui permet d'annuler la transaction courante si un verrou ne pouvait être acquis. Si l'on reprend les premières étapes de l'exemple précédent :

session 1

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE NOWAIT;
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

Aucun verrou préalable n'avait été posé, la requête SELECT a retourné les données souhaitées.

session 2

On effectue la même chose sur la session n°2 :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004' FOR UPDATE NOWAIT;
ERROR:  could not obtain lock on row in relation "employes"
```

Comme la session n°1 possède déjà un verrou sur la ligne qui nous intéresse, l'option `NOWAIT` sur le `SELECT` a annulé la transaction.

Il faut maintenant effectuer un `ROLLBACK` explicite pour pouvoir recommencer les traitements au risque d'obtenir le message suivant :

```
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
```

5.6.2 SKIP LOCKED

- `SELECT FOR UPDATE SKIP LOCKED`
  - PostgreSQL 9.5
- Utilité :
  - implémente des files d'attentes parallélisables

Une dernière fonctionnalité intéressante de `SELECT FOR UPDATE`, apparue avec PostgreSQL 9.5, permet de mettre en oeuvre différents workers qui consomment des données issues d'une table représentant une file d'attente. Il s'agit de la clause `SKIP LOCKED`, dont le principe de fonctionnement est identique à son équivalent sous Oracle.

En prenant une table représentant la file d'attente suivante, peuplée avec des données générées :

```
CREATE TABLE test_skiplocked (id serial primary key, val text);
INSERT INTO test_skiplocked (val) SELECT md5(i::text)
FROM generate_series(1, 1000) i;
```

Une première transaction est ouverte et tente d'obtenir un verrou sur les 10 premières lignes :

```
BEGIN TRANSACTION;

SELECT *
  FROM test_skiplocked
 LIMIT 10
   FOR UPDATE SKIP LOCKED;

 id |               val
----+----------------------------------
  1 | c4ca4238a0b923820dcc509a6f75849b
  2 | c81e728d9d4c2f636f067f89cc14862c
  3 | eccbc87e4b5ce2fe28308fd9f2a7baf3
  4 | a87ff679a2f3e71d9181a67b7542122c
  5 | e4da3b7fbbce2345d7772b0674a318d5
  6 | 1679091c5a880faf6fb5e6087eb1b2dc
  7 | 8f14e45fceea167a5a36dedd4bea2543
  8 | c9f0f895fb98ab9159f51fd0297e236d
  9 | 45c48cce2e2d7fbdea1afc51c7c6ad26
 10 | d3d9446802a44259755d38e6d163e820
(10 rows)
```

Si on démarre une seconde transaction en parallèle, avec la première transaction toujours ouverte, le fait d'exécuter la requête `SELECT FOR UPDATE` sans la clause `SKIP LOCKED` aurait pour effet de la mettre en attente. L'ordre `SELECT` rendra la main lorsque la transaction

#1 se terminera.

Avec la clause `SKIP LOCKED`, les 10 premières verrouillées par la transaction n°1 seront passées et ce sont les 10 lignes suivantes qui seront verrouillées et retournées par l'ordre `SELECT` :

```
BEGIN TRANSACTION;

SELECT *
  FROM test_skiplocked
 LIMIT 10
   FOR UPDATE SKIP LOCKED;

 id |               val
----+----------------------------------
 11 | 6512bd43d9caa6e02c990b0a82652dca
 12 | c20ad4d76fe97759aa27a0c99bff6710
 13 | c51ce410c124a10e0db5e4b97fc2af39
 14 | aab3238922bcc25a6f606eb525ffdc56
 15 | 9bf31c7ff062936a96d3c8bd1f8f2ff3
 16 | c74d97b01eae257e44aa9d5bade97baf
 17 | 70efdf2ec9b086079795c442636b55fb
 18 | 6f4922f45568161a8cdf4ad2299f6d23
 19 | 1f0e3dad99908345f7439f8ffabdffc4
 20 | 98f13708210194c475687be6106a3b84
(10 rows)
```

Ensuite, la première transaction supprime les lignes verrouillées et valide la transaction :

```
DELETE FROM test_skiplocked
 WHERE id IN (...);
COMMIT;
```

De même pour la seconde transaction, qui aura traité d'autres lignes en parallèle que la transaction #1.

5.7 SERIALIZABLE SNAPSHOT ISOLATION

SSI : Serializable Snapshot Isolation (9.1+)

- Chaque transaction est seule sur la base
- Si on ne peut maintenir l'illusion
  - Une des transactions en cours est annulée
- Sans blocage
- On doit être capable de rejouer la transaction

- Toutes les transactions impliquées doivent être `serializable`
- `default_transaction_isolation=serializable` dans la configuration

PostgreSQL fournit depuis la version 9.1 un mode d'isolation appelé `SERIALIZABLE`. Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base. Dès que cette garantie ne peut plus être apportée, une des transactions est annulée.

Toute transaction non déclarée comme `SERIALIZABLE` peut en théorie s'exécuter n'importe quand, ce qui rend inutile le mode `SERIALIZABLE` sur les autres. C'est donc un mode qui doit être mis en place globalement.

Voici un exemple :

Dans cet exemple, il y a des enregistrements avec une colonne couleur contenant 'blanc' ou 'noir'. Deux utilisateurs essayent simultanément de convertir tous les enregistrements vers une couleur unique, mais chacun dans une direction opposée. Un veut passer tous les blancs en noir, et l'autre tous les noirs en blanc.

L'exemple peut être mis en place avec ces ordres :

```
create table points
  (
    id int not null primary key,
    couleur text not null
  );
insert into points
  with x(id) as (select generate_series(1,10))
  select id, case when id % 2 = 1 then 'noir'
    else 'blanc' end from x;
```

Session 1 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'noir'
where couleur = 'blanc';
```

Session 2 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'blanc'
where couleur = 'noir';
```

À ce moment, une des deux transaction est condamnée à mourir.

Session 2 :

```
commit;
```

Le premier à valider gagne.

```
select * from points order by id;
```

```
 id | couleur
----+-------
  1 | blanc
  2 | blanc
  3 | blanc
  4 | blanc
  5 | blanc
  6 | blanc
  7 | blanc
  8 | blanc
  9 | blanc
 10 | blanc
(10 rows)
```

Session 1 : Celle-ci s'est exécutée comme si elle était seule.

```
commit;
```

```
ERROR:  could not serialize access
        due to read/write dependencies
        among transactions
DETAIL:  Cancelled on identification
         as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Une erreur de sérialisation. On annule et on réessaye.

```
rollback;
begin;
update points set couleur = 'noir'
  where couleur = 'blanc';
commit;
```

Il n'y a pas de transaction concurrente pour gêner.

```
select * from points order by id;
```

```
 id | couleur
----+-------
  1 | noir
  2 | noir
  3 | noir
  4 | noir
  5 | noir
  6 | noir
```

```
   7 | noir
   8 | noir
   9 | noir
  10 | noir
 (10 rows)
```

La transaction s'est exécutée seule, après l'autre.

Le mode `SERIALIZABLE` permet de s'affranchir des `SELECT FOR UPDATE` qu'on écrit habituellement, dans les applications en mode `READ COMMITTED`. Toutefois, il fait bien plus que ça, puisqu'il réalise du verrouillage de prédicats. Un enregistrement qui «apparaît» ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation. Il permet aussi de gérer les problèmes ci-dessus avec plus de deux sessions.

Pour des exemples plus complets, le mieux est de consulter la documentation officielle[37]
.

5.8 CONCLUSION

- SQL est un langage très riche
- Connaître les nouveautés des versions de la norme depuis 20 ans permet de
  - gagner énormément de temps de développemment
  - mais aussi de performance

5.9 TRAVAUX PRATIQUES

5.9.1 ÉNONCÉS

Jointure latérale

Cette série de question utilise la base `magasin`, qui est disponible dans le schéma `magasin` de la base de TP.

Afficher les 10 derniers articles commandés.

Pour chacune des 10 dernières commandes passées, afficher le premier article commandé :

CTE récursive

[37] https://wiki.postgresql.org/wiki/SSI/fr

Généalogie

Cet exercice propose de manipuler des données généalogiques, disposées dans le schéma `genealogie` de l'environnement de TP.

Voici la description de la table genealogie qui sera utilisée :

```
\d genealogie
                              Table "public.genealogie"
     Column      |  Type   |                Modifiers
-----------------+---------+-----------------------------------
 id              | integer | not null default                  +
                 |         | nextval('genealogie_id_seq'::regclass)
 nom             | text    |
 prenom          | text    |
 date_naissance | date    |
 pere            | integer |
 mere            | integer |
Indexes:
    "genealogie_pkey" PRIMARY KEY, btree (id)
```

À partir de la table `genealogie`, déterminer qui sont les descendants de Fernand DEVAUX.

À l'inverse, déterminer qui sont les ancètres de Adèle TAILLANDIER

Réseau social

Cet exercice est assez similaire au précédent et propose de manipuler des arborescences. Les tables de travail sont disponibles dans le schéma `socialnet`.

Les tableaux et la fonction unnest peuvent être utiles pour résoudre plus facilement ce problème

La table `personnes` contient la liste de toutes les personnes d'un réseau social.

```
                        Table "public.personnes"
 Column |  Type   |                       Modifiers
--------+---------+-------------------------------------------------------
 id     | integer | not null default nextval('personnes_id_seq'::regclass)
 nom    | text    | not null
 prenom | text    | not null
Indexes:
    "personnes_pkey" PRIMARY KEY, btree (id)
```

La table `relation` contient les connexions entre ces personnes.

```
   Table "public.relation"
 Column |  Type   | Modifiers
--------+---------+-----------
 gauche | integer | not null
 droite | integer | not null
Indexes:
    "relation_droite_idx" btree (droite)
    "relation_gauche_idx" btree (gauche)
```

Déterminer le niveau de connexions entre Sadry Luettgen et Yelsi Kerluke et afficher le chemin de relation le plus court qui permet de les connecter ensemble.

Dépendance de vues

Les dépendances entre objets est un problème classique dans les bases de données :

- dans quel ordre charger des tables selon les clés étrangères ?
- dans quel ordre recréer des vues ?
- etc.

Le catalogue de PostgreSQL décrit l'ensemble des objets de la base de données. Deux tables vont nous intéresser pour mener à bien cet exercice :

- `pg_depend` liste les dépendances entre objets
- `pg_rewrite` stocke les définitions des règles de réécritures des vues (RULES)
- `pg_class` liste les objets que l'on peut interroger comme une table, hormis les fonctions retournant des ensembles

La définition d'une vue peut être obtenue à l'aide de la fonction `pg_get_viewdef`.

Pour plus d'informations sur ces tables, se référer à la documentation :

- Catalogue pg_depend[38]
- Catalogue pg_rewrite[39]
- Catalogue pg_class[40]
- Fonction d'inforamtion du catalogue système[41]

Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de recréation des vues.

[38] http://www.postgresql.org/docs/9.4/static/catalog-pg-depend.html
[39] http://www.postgresql.org/docs/9.4/static/catalog-pg-rewrite.html
[40] http://www.postgresql.org/docs/9.4/static/catalog-pg-class.html
[41] http://www.postgresql.org/docs/9.4/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE

5.9.2 CORRECTIONS DU TP

Jointure latérale

Afficher les 10 derniers articles commandés.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

On commence par afficher les 10 dernières commandes :

```
SELECT *
  FROM commandes
 ORDER BY numero_commande DESC
 LIMIT 10;
```

Une simple jointure nous permet de retrouver les 10 derniers articles commandés :

```
SELECT lc.produit_id, p.nom
  FROM commandes c
  JOIN lignes_commandes lc
    ON (c.numero_commande = lc.numero_commande)
  JOIN produits p
    ON (lc.produit_id = p.produit_id)
 ORDER BY c.numero_commande DESC, numero_ligne_commande DESC
 LIMIT 10;
```

Pour chacune des 10 dernières commandes passées, afficher le premier article commandé :

La requête précédente peut être dérivée pour répondre à la question demandée. Ici, pour chacune des dix dernières commandes, nous vous récupérer le nom du dernier article commandé, ce qui sera transcrit sous la forme d'une jointure latérale :

```
SELECT numero_commande, produit_id, nom
  FROM commandes c,
  LATERAL (SELECT p.produit_id, p.nom
             FROM lignes_commandes lc
             JOIN produits p
               ON (lc.produit_id = p.produit_id)
            WHERE (c.numero_commande = lc.numero_commande)
            ORDER BY numero_ligne_commande ASC
            LIMIT 1
 ) premier_article_par_commande
 ORDER BY c.numero_commande DESC
 LIMIT 10;
```

CTE récursive

17.12

Généalogie

Cet exercice propose de manipuler des données généalogiques.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `genealogie` :

```
SET search_path = genealogie;
```

Voici la description de la table genealogie qui sera utilisée :

```
\d genealogie
                              Table "public.genealogie"
     Column      |  Type   |                Modifiers
-----------------+---------+-----------------------------------
 id              | integer | not null default                  +
                 |         | nextval('genealogie_id_seq'::regclass)
 nom             | text    |
 prenom          | text    |
 date_naissance  | date    |
 pere            | integer |
 mere            | integer |
Indexes:
    "genealogie_pkey" PRIMARY KEY, btree (id)
```

À partir de la table genealogie, déterminer qui sont les descendants de Fernand DEVAUX.

```
WITH RECURSIVE arbre_genealogique AS (
SELECT id, nom, prenom, date_naissance, pere, mere
  FROM genealogie
 WHERE nom = 'DEVAUX'
   AND prenom = 'Fernand'
UNION ALL
SELECT g.*
  FROM arbre_genealogique ancetre
  JOIN genealogie g
    ON (g.pere = ancetre.id OR g.mere = ancetre.id)
)
SELECT id, nom, prenom, date_naissance
  FROM arbre_genealogique
```

À l'inverse, déterminer qui sont les ancètres de Adèle TAILLANDIER

```
WITH RECURSIVE arbre_genealogique AS (
SELECT id, nom, prenom, date_naissance, pere, mere
  FROM genealogie
 WHERE nom = 'TAILLANDIER'
   AND prenom = 'Adèle'
```

```
UNION ALL
SELECT ancetre.id, ancetre.nom, ancetre.prenom, ancetre.date_naissance,
       ancetre.pere, ancetre.mere
  FROM arbre_genealogique descendant
  JOIN genealogie ancetre
    ON (descendant.pere = ancetre.id OR descendant.mere = ancetre.id)
)
SELECT id, nom, prenom, date_naissance
  FROM arbre_genealogique;
```

Réseau social

Cet exercice est assez similaire au précédent.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `socialnet` :

```
SET search_path = socialnet;
```

Les tableaux et la fonction unnest peuvent être utiles pour résoudre plus facilement ce problème

La table `personnes` contient la liste de toutes les personnes d'un réseau social.

```
                        Table "public.personnes"
 Column |  Type   |                       Modifiers
--------+---------+-------------------------------------------------------
 id     | integer | not null default nextval('personnes_id_seq'::regclass)
 nom    | text    | not null
 prenom | text    | not null
Indexes:
    "personnes_pkey" PRIMARY KEY, btree (id)
```

La table `relation` contient les connexions entre ces personnes.

```
   Table "public.relation"
 Column |  Type   | Modifiers
--------+---------+-----------
 gauche | integer | not null
 droite | integer | not null
Indexes:
    "relation_droite_idx" btree (droite)
    "relation_gauche_idx" btree (gauche)
```

Déterminer le niveau de connexions entre Sadry Luettgen et Yelsi Kerluke et afficher le chemin de relation le plus court qui permet de les connecter ensemble.

17.12

La requête suivante permet de répondre à cette question :

```
WITH RECURSIVE connexions AS (
SELECT gauche, droite, ARRAY[gauche] AS personnes_connectees,0::integer AS level
  FROM relation
 WHERE gauche = 1
UNION ALL
SELECT p.gauche, p.droite, personnes_connectees || p.gauche, level + 1 AS level
  FROM connexions c
  JOIN relation p ON (c.droite = p.gauche)
 WHERE level < 4
   AND p.gauche <> ANY (personnes_connectees)
), plus_courte_connexion AS (
SELECT *
  FROM connexions
 WHERE gauche = (
    SELECT id FROM personnes WHERE nom = 'Kerluke' AND prenom = 'Yelsi'
 )
 ORDER BY level ASC
 LIMIT 1
)
SELECT list.id, p.nom, p.prenom, list.level - 1 AS level
  FROM plus_courte_connexion,
       unnest(personnes_connectees) WITH ORDINALITY AS list(id, level)
  JOIN personnes p on (list.id = p.id)
 ORDER BY list.level;
```

Cet exemple fonctionne sur une faible volumétrie, mais les limites des bases relationnelles sont rapidement atteintes sur de telles requêtes.

Une solution consisterait à implémenter un algorithme de parcours de graphe avec pgRouting[42] , mais cela nécessitera de présenter les données sous une forme particulière.

Pour les problématiques de traitement de graphe, notamment sur de grosses volumétries, une base de données orientée graphe comme Neo4J sera probablement plus adaptée.

Dépendance de vues

Les dépendances entre objets est un problème classique dans les bases de données :

- dans quel ordre charger des tables selon les clés étrangères ?
- dans quel ordre recréer des vus ?
- etc.

Le catalogue de PostgreSQL décrit l'ensemble des objets de la base de données. Deux tables vont nous intéresser pour mener à bien cet exercice :

[42] http://docs.pgrouting.org/2.1/src/dijkstra/doc/index.html#pgr-dijkstra

- `pg_depend` liste les dépendances entre objets
- `pg_rewrite` stocke les définitions des règles de réécritures des vues (RULES)
- `pg_class` liste les objets que l'on peut interroger comme une table, hormis les fonctions retournant des ensembles

La définition d'une vue peut être obtenue à l'aide de la fonction `pg_get_viewdef`.

Pour plus d'informations sur ces tables, se référer à la documentation :

- Catalogue pg_depend[43]
- Catalogue pg_rewrite[44]
- Catalogue pg_class[45]
- Fonction d'inforamtion du catalogue système[46]

Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de recréation des vues.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

Si la jointure entre `pg_depend` et `pg_rewrite` est possible pour l'objet de départ, alors il s'agit probablement d'une vue. En discriminant sur les objets qui référencent la vue `pilotes_brno`, nous arrivons à la requête de départ suivante :

```
SELECT DISTINCT pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
```

La présence de doublons nous oblige à utiliser la clause DISTINCT.

Nous pouvons donc créer un graphe de dépendances à partir de cette requête de départ, transformée en requête récursive :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
```

[43] http://www.postgresql.org/docs/9.4/static/catalog-pg-depend.html
[44] http://www.postgresql.org/docs/9.4/static/catalog-pg-rewrite.html
[45] http://www.postgresql.org/docs/9.4/static/catalog-pg-class.html
[46] http://www.postgresql.org/docs/9.4/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE

```
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
)
SELECT * FROM graph;
```

Il faut maintenant résoudre les OID pour déterminer les noms des vues et leur schéma. Pour cela, nous ajoutons une vue `resolved` telle que :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
),
resolved AS (
  SELECT n.nspname AS dependent_schema, d.relname as dependent,
         n2.nspname AS dependee_schema, d2.relname as dependee,
         depth
  FROM graph
  JOIN pg_class d ON d.oid = objid
  JOIN pg_namespace n ON d.relnamespace = n.oid
  JOIN pg_class d2 ON d2.oid = refobjid
  JOIN pg_namespace n2 ON d2.relnamespace = n2.oid
)
SELECT * FROM resolved;
```

Nous pouvons maintenant présenter les ordres de suppression et de recréation des vues, dans le bon ordre. Les vues doivent être supprimées selon le numéro d'ordre décroissant et recrées selon le numéro d'ordre croissant :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
```

```
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
),
resolved AS (
  SELECT n.nspname AS dependent_schema, d.relname as dependent,
         n2.nspname AS dependee_schema, d2.relname as dependee,
         depth
  FROM graph
  JOIN pg_class d ON d.oid = objid
  JOIN pg_namespace n ON d.relnamespace = n.oid
  JOIN pg_class d2 ON d2.oid = refobjid
  JOIN pg_namespace n2 ON d2.relnamespace = n2.oid
)
(SELECT 'DROP VIEW ' || dependent_schema || '.' || dependent || ';'
   FROM resolved
  GROUP BY dependent_schema, dependent
  ORDER BY max(depth) DESC)
UNION ALL
(SELECT 'CREATE OR REPLACE VIEW ' || dependent_schema || '.' || dependent ||
        ' AS ' || pg_get_viewdef(dependent) || '; '
   FROM resolved
  GROUP BY dependent_schema, dependent
  ORDER BY max(depth));
```

6 SQL POUR L'ANALYSE DE DONNÉES

6.1 PRÉAMBULE

- Analyser des données est facile avec PostgreSQL
  - opérations d'agrégation disponibles
  - fonctions OLAP avancées

6.1.1 MENU

- agrégation de données
- clause FILTER
- fonctions window
- GROUPING SETS, ROLLUP, CUBE
- WITHIN GROUPS

6.1.2 OBJECTIFS

- Écrire des requêtes encore plus complexes
- Analyser les données en amont
  - pour ne récupérer que le résultat

6.2 AGRÉGATS

- SQL dispose de fonctions de calcul d'agrégats
- Utilité :
  - calcul de sommes, moyennes, valeur minimale et maximale
  - nombreuses fonctions statistiques disponibles

À l'aide des fonctions de calcul d'agrégats, on peut réaliser un certain nombre de calculs permettant d'analyser les données d'une table.

La plupart des exemples utilisent une table `employes` définie telle que :

```
CREATE TABLE employes (
  matricule char(8) primary key,
  nom       text    not null,
```

```
  service   text,
  salaire   numeric(7,2)
);

INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000001', 'Dupuis', 'Direction', 10000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000004', 'Fantasio', 'Courrier', 4500.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000006', 'Prunelle', 'Publication', 4000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000020', 'Lagaffe', 'Courrier', 3000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000040', 'Lebrac', 'Publication', 3000.00);

SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

Ainsi, on peut déduire le salaire moyen avec la fonction `avg()`, les salaires maximum et minimum versés par la société avec les fonctions `max()` et `min()`, ainsi que la somme totale des salaires versés avec la fonction `sum()` :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
  FROM employes;
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 4900.0000000000000000 |        10000.00 |         3000.00 |       24500.00
```

La base de données réalise les calculs sur l'ensemble des données de la table et n'affiche que le résultat du calcul.

Si l'on applique un filtre sur les données, par exemple pour ne prendre en compte que le service *Courrier*, alors PostgreSQL réalise le calcul uniquement sur les données issues de la lecture :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
```

```
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
  FROM employes
 WHERE service = 'Courrier';
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 3750.0000000000000000 |         4500.00 |         3000.00 |        7500.00
(1 ligne)
```

En revanche, il n'est pas possible de référencer d'autres colonnes pour les afficher à côté du résultat d'un calcul d'agrégation à moins de les utiliser comme critère de regroupement :

```
SELECT avg(salaire), nom FROM employes;
ERROR:  column "employes.nom" must appear in the GROUP BY clause or be used in
        an aggregate function
LIGNE 1 : SELECT avg(salaire), nom FROM employes;
                               ^
```

6.2.1 AGRÉGATS AVEC GROUP BY

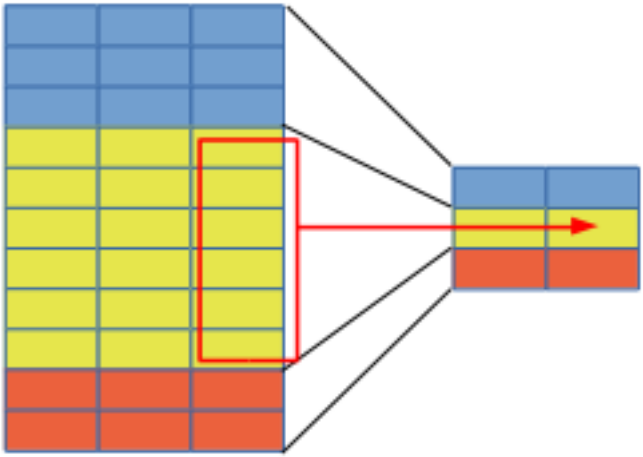
- agrégat + `GROUP BY`
- Utilité
  - effectue des calculs sur des regroupements : moyenne, somme, comptage, etc.
  - regroupement selon un critère défini par la clause `GROUP BY`
  - exemple : calcul du salaire moyen de chaque service

L'opérateur d'agrégat `GROUP BY` indique à la base de données que l'on souhaite regrouper les données selon les mêmes valeurs d'une colonne.

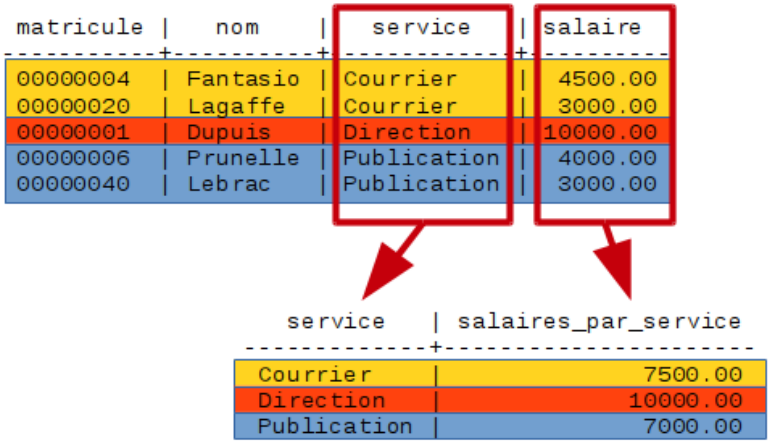
matricule	nom	service	salaire
00000004	Fantasio	Courrier	4500.00
00000020	Lagaffe	Courrier	3000.00
00000001	Dupuis	Direction	10000.00
00000006	Prunelle	Publication	4000.00
00000040	Lebrac	Publication	3000.00

Des calculs pourront être réalisés sur les données agrégées selon le critère de regroupement donné. Le résultat sera alors représenté en n' affichant que les colonnes de re-

groupement puis les valeurs calculées par les fonctions d'agrégation :



6.2.2 GROUP BY : PRINCIPE



L'agrégation est ici réalisée sur la colonne `service`. En guise de calcul d'agrégation, une

somme est réalisée sur les salaires payés dans chaque service.

6.2.3 GROUP BY : EXEMPLES

```
SELECT service,
       sum(salaire) AS salaires_par_service
  FROM employes
 GROUP BY service;

   service   | salaires_par_service
-------------+----------------------
 Courrier    |              7500.00
 Direction   |             10000.00
 Publication |              7000.00
(3 lignes)
```

SQL permet depuis le début de réaliser des calculs d'agrégation. Pour cela, la base de données observe les critères de regroupement définis dans la clause `GROUP BY` de la requête et effectue l'opération sur l'ensemble des lignes qui correspondent au critère de regroupement.

On peut bien entendu combiner plusieurs opérations d'agrégations :

```
SELECT service,
       sum(salaire) salaires_par_service,
       avg(salaire) AS salaire_moyen_service
  FROM employes
 GROUP BY service;

   service   | salaires_par_service | salaire_moyen_service
-------------+----------------------+-----------------------
 Courrier    |              7500.00 |  3750.0000000000000000
 Direction   |             10000.00 | 10000.0000000000000000
 Publication |              7000.00 |  3500.0000000000000000
(3 lignes)
```

On peut combiner le résultat de deux requêtes d'agrégation avec `UNION ALL`, si les ensembles retournées sont de même type :

```
SELECT service,
       sum(salaire) AS salaires_par_service
  FROM employes GROUP BY service
UNION ALL
SELECT 'Total' AS service,
       sum(salaire) AS salaires_par_service
```

```
  FROM employes;

   service    | salaires_par_service
--------------+----------------------
 Courrier     |              7500.00
 Direction    |             10000.00
 Publication  |              7000.00
 Total        |             24500.00
(4 lignes)
```

On le verra plus loin, cette dernière requête peut être écrite plus simplement avec les `GROUPING SETS`, mais qui nécessitent au minimum PostgreSQL 9.5.

6.2.4 AGRÉGATS ET ORDER BY

- Extension propriétaire de PostgreSQL
  - `ORDER BY` dans la fonction d'agrégat
- Utilité :
  - ordonner les données agrégées
  - surtout utile avec `array_agg`, `string_agg` et `xmlagg`

Les fonctions `array_agg`, `string_agg` et `xmlagg` permettent d'agréger des éléments dans un tableau, dans une chaîne ou dans une arborescence XML. Autant l'ordre dans lequel les données sont utilisées n'a pas d'importance lorsque l'on réalise un calcul d'agrégat classique, autant cet ordre va influencer la façon dont les données seront produites par les trois fonctions citées plus haut. En effet, le tableau généré par `array_agg` est composé d'éléments ordonnés, de même que la chaîne de caractères ou l'arborescence XML.

6.2.5 UTILISER ORDER BY AVEC UN AGRÉGAT

```
SELECT service,
       string_agg(nom, ', ' ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |  liste_employes
--------------+------------------
 Courrier     | Fantasio, Lagaffe
 Direction    | Dupuis
 Publication | Lebrac, Prunelle
(3 lignes)
```

17.12

La requête suivante permet d'obtenir, pour chaque service, la liste des employés dans un tableau, trié par ordre alphabétique :

```
SELECT service,
       string_agg(nom, ', ' ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |  liste_employes
--------------+-------------------
 Courrier     | Fantasio, Lagaffe
 Direction    | Dupuis
 Publication | Lebrac, Prunelle
(3 lignes)
```

Il est possible de réaliser la même chose mais pour obtenir un tableau plutôt qu'une chaîne de caractère :

```
SELECT service,
       array_agg(nom ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |   liste_employes
--------------+--------------------
 Courrier     | {Fantasio,Lagaffe}
 Direction    | {Dupuis}
 Publication | {Lebrac,Prunelle}
```

6.3 CLAUSE FILTER

- Clause `FILTER`
- Utilité :
  - filtrer les données sur les agrégats
  - évite les expressions `CASE` complexes
- SQL:2003
- Intégré dans la version 9.4

La clause `FILTER` permet de remplacer des expressions complexes écrites avec `CASE` et donc de simplifier l'écriture de requêtes réalisant un filtrage dans une fonction d'agrégat.

6.3.1 FILTRER AVEC CASE

- La syntaxe suivante était utilisée :

```
SELECT count(*) AS compte_pays,
       count(CASE WHEN r.nom_region='Europe' THEN 1
                  ELSE 0
             END) AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

Avec cette syntaxe, dès que l'on a besoin d'avoir de multiples filtres ou de filtres plus complexes, la requête devient très rapidement peu lisible et difficile à maintenir. Le risque d'erreur est également élevé.

6.3.2 FILTRER AVEC FILTER

- La même requête écrite avec la clause FILTER :

```
SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe')
                AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

L'exemple suivant montre l'utilisation de la clause `FILTER` et son équivalent écrit avec une expression `CASE` :

```
sql=# SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe') AS compte_pays_europeens,
       count(CASE WHEN r.nom_region='Europe' THEN 1 END)
           AS oldschool_compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
 compte_pays | compte_pays_europeens | oldschool_compte_pays_europeens
-------------+-----------------------+---------------------------------
          25 |                     5 |                               5
(1 ligne)
```

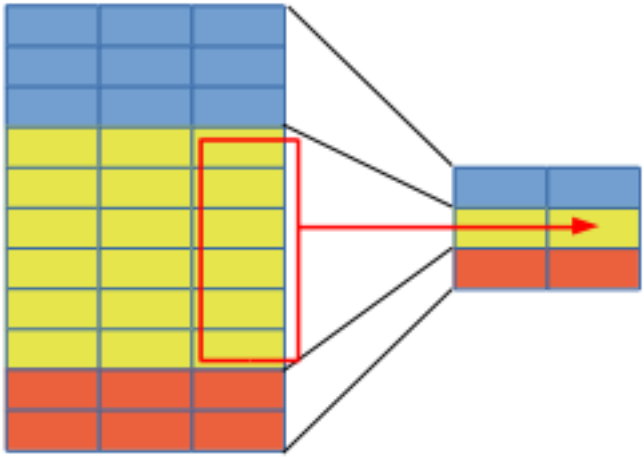
6.4 FONCTIONS DE FENÊTRAGE

- Fonctions *window*
  - travaille sur des ensembles de données regroupés et triés indépendamment de la requête principale
- Utilisation :
  - utiliser plusieurs critères d'agrégation dans la même requête
  - utiliser des fonctions de classement
  - faire référence à d'autres lignes de l'ensemble de données

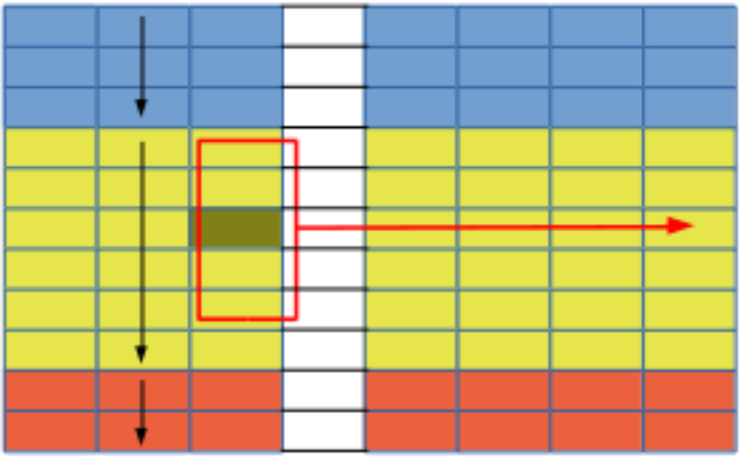
PostgreSQL supporte les fonctions de fenêtrage depuis la version 8.4. Elles apportent des fonctionnalités analytiques à PostgreSQL, et permettent d'écrire beaucoup plus simplement certaines requêtes.

Prenons un exemple.

```
SELECT service, AVG(salaire)
FROM employe
GROUP BY service
```



```
SELECT service, id_employe, salaire,
  AVG(salaire) OVER (
    PARTITION BY service
    ORDER BY age
    ROWS BETWEEN 2 PRECEEDING AND 2 FOLLOWING
  )
FROM employes
```



6.4.1 REGROUPEMENT

- Regroupement
  - clause `OVER (PARTITION BY ...)`
- Utilité :
  - plusieurs critères de regroupement différents
  - avec des fonctions de calcul d'agrégats

La clause `OVER` permet de définir la façon dont les données sont regroupées - uniquement pour la colonne définie - avec la clause `PARTITION BY`.

Les exemples vont utiliser cette table `employes` :

```
exemple=# SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

6.4.2 REGROUPEMENT : EXEMPLE

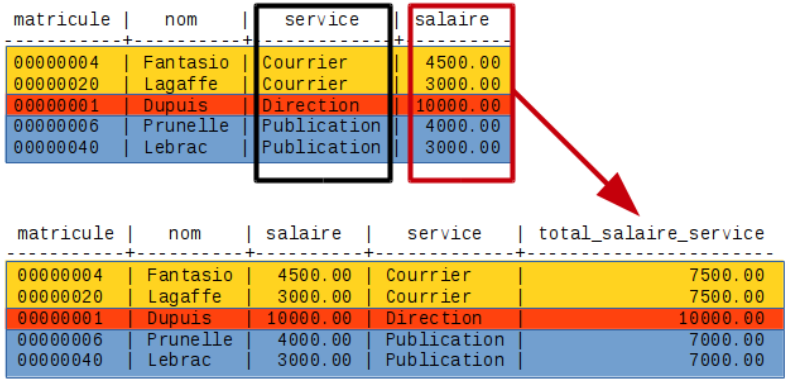
```
SELECT matricule, salaire, service,
       SUM(salaire) OVER (PARTITION BY service)
                 AS total_salaire_service
  FROM employes;
```

```
 matricule | salaire  |   service    | total_salaire_service
-----------+----------+--------------+-----------------------
 00000004  |  4500.00 | Courrier     |               7500.00
 00000020  |  3000.00 | Courrier     |               7500.00
 00000001  | 10000.00 | Direction    |              10000.00
 00000006  |  4000.00 | Publication  |               7000.00
 00000040  |  3000.00 | Publication  |               7000.00
```

Les calculs réalisés par cette requête sont identiques à ceux réalisés avec une agrégation utilisant `GROUP BY`. La principale différence est que l'on évite de ici de perdre le détail des données tout en disposant des données agrégées dans le résultat de la requête.

6.4.3 REGROUPEMENT : PRINCIPE

```
SUM(salaire) OVER (PARTITION BY service)
```



Entouré de noir, le critère de regroupement et entouré de rouge, les données sur lesquelles sont appliqués le calcul d'agrégat.

6.4.4 REGROUPEMENT : SYNTAXE

```
SELECT ...
 agregation OVER (PARTITION BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Le terme `PARTITION BY` permet d'indiquer les critères de regroupement de la fenêtre sur laquelle on souhaite travailler.

6.4.5 TRI

- Tri
  - `OVER (ORDER BY …)`
- Utilité :
  - numéroter les lignes : `row_number()`
  - classer des résultats : `rank()`, `dense_rank()`
  - faire appel à d'autres ligne du résultat : `lead()`, `lag()`

6.4.6 TRI : EXEMPLE

- Pour numéroter des lignes :

```
SELECT row_number() OVER (ORDER BY matricule),
       matricule, nom
  FROM employes;

 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          3 | 00000006  | Prunelle
          4 | 00000020  | Lagaffe
          5 | 00000040  | Lebrac
(5 lignes)
```

La fonction `row_number()` permet de numéroter les lignes selon un critère de tri défini dans la clause `OVER`.

L'ordre de tri de la clause `OVER` n'influence pas l'ordre de tri explicite d'une requête :

```
SELECT row_number() OVER (ORDER BY matricule),
       matricule, nom
  FROM employes
 ORDER BY nom;
 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          4 | 00000020  | Lagaffe
          5 | 00000040  | Lebrac
          3 | 00000006  | Prunelle
(5 lignes)
```

On dispose aussi de fonctions de classement, pour déterminer par exemple quels sont les employés les moins bien payés :

```
SELECT matricule, nom, salaire, service,
       rank() OVER (ORDER BY salaire),
       dense_rank() OVER (ORDER BY salaire)
  FROM employes ;
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

La fonction de fenêtrage `rank()` renvoie le classement en autorisant des trous dans la numérotation, et `dense_rank()` le classement sans trous.

6.4.7 TRI : EXEMPLE AVEC UNE SOMME

- Calcul d'une somme glissante :

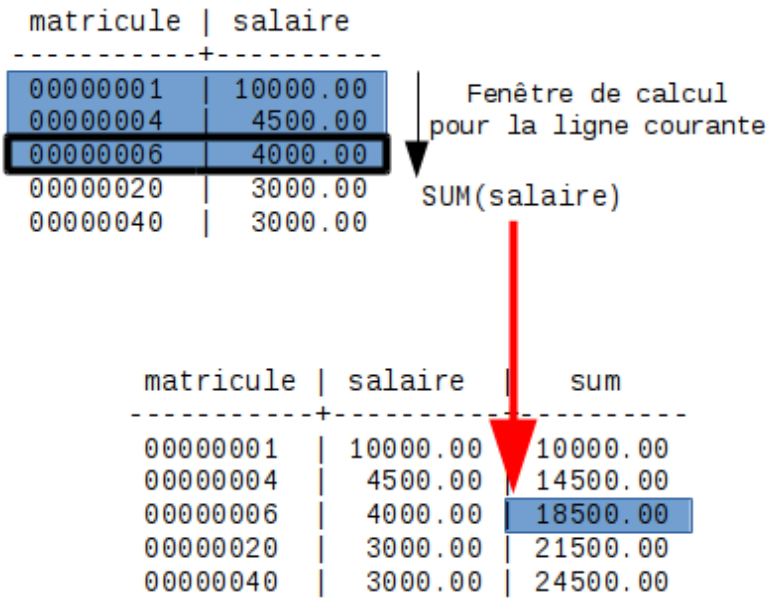
```
SELECT matricule, salaire,
       SUM(salaire) OVER (ORDER BY matricule)
  FROM employes;

 matricule | salaire  |   sum
-----------+----------+----------
 00000001  | 10000.00 | 10000.00
 00000004  |  4500.00 | 14500.00
 00000006  |  4000.00 | 18500.00
```

```
00000020  |  3000.00 | 21500.00
00000040  |  3000.00 | 24500.00
```

6.4.8 TRI : PRINCIPE

```
SUM(salaire) OVER (ORDER BY matricule)
```



Lorsque l'on utilise une clause de tri, la portion de données visible par l'opérateur d'agrégat correspond aux données comprises entre la première ligne examinée et la ligne courante. La fenêtre est définie selon le critère `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Nous verrons plus loin que nous pouvons modifier ce comportement.

6.4.9 TRI : SYNTAXE

```
SELECT ...
 agregation OVER (ORDER BY >colonnes>)
```

```
  FROM <liste_tables>
 WHERE <predicats>
```

Le terme `ORDER BY` permet d'indiquer les critères de tri de la fenêtre sur laquelle on souhaite travailler.

6.4.10 REGROUPEMENT ET TRI

- On peut combiner les deux
  - `OVER (PARTITION BY .. ORDER BY ..)`
- Utilité :
  - travailler sur des jeux de données ordonnés et isolés les uns des autres

Il est possible de combiner les clauses de fenêtrage `PARTITION BY` et `ORDER BY`. Cela permet d'isoler des jeux de données entre eux avec la clause `PARTITION BY`, tout en appliquant un critère de tri avec la clause `ORDER BY`. Beaucoup d'applications sont possibles si l'on associe à cela les nombreuses fonctions analytiques disponibles.

6.4.11 REGROUPEMENT ET TRI : EXEMPLE

```
SELECT continent, pays, population,
       rank() OVER (PARTITION BY continent
                    ORDER BY population DESC)
              AS rang
  FROM population;
```

```
    continent     |       pays          | population | rang
------------------+---------------------+------------+------
 Afrique          | Nigéria             |      173.6 |    1
 Afrique          | Éthiopie            |       94.1 |    2
 Afrique          | Égypte              |       82.1 |    3
 Afrique          | Rép. dém. du Congo  |       67.5 |    4
(...)
 Amérique du Nord | États-Unis          |      320.1 |    1
 Amérique du Nord | Canada              |       35.2 |    2
(...)
```

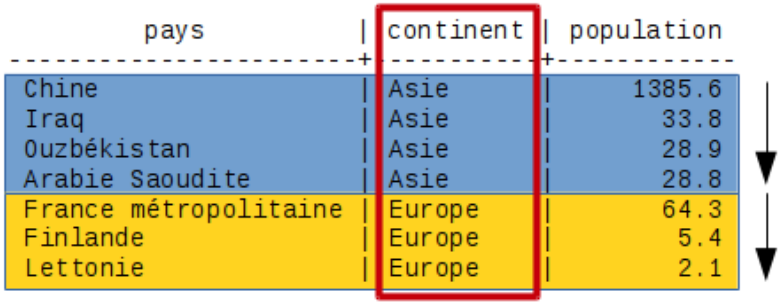
Si l'on applique les deux clauses `PARTITION BY` et `ORDER BY` à une fonction de fenêtrage, alors le critère de tri est appliqué dans la partition et chaque partition est indépendante l'une de l'autre.

Voici un extrait plus complet du résultat de la requête présentée ci-dessus :

```
        continent         |         pays          | population | rang_population
--------------------------+-----------------------+------------+-----------------
 Afrique                  | Nigéria               |      173.6 |               1
 Afrique                  | Éthiopie              |       94.1 |               2
 Afrique                  | Égypte                |       82.1 |               3
 Afrique                  | Rép. dém. du Congo    |       67.5 |               4
 Afrique                  | Afrique du Sud        |       52.8 |               5
 Afrique                  | Tanzanie              |       49.3 |               6
 Afrique                  | Kenya                 |       44.4 |               7
 Afrique                  | Algérie               |       39.2 |               8
 Afrique                  | Ouganda               |       37.6 |               9
 Afrique                  | Maroc                 |       33.0 |              10
 Afrique                  | Ghana                 |       25.9 |              11
 Afrique                  | Mozambique            |       25.8 |              12
 Afrique                  | Madagascar            |       22.9 |              13
 Afrique                  | Côte-d'Ivoire         |       20.3 |              14
 Afrique                  | Niger                 |       17.8 |              15
 Afrique                  | Burkina Faso          |       16.9 |              16
 Afrique                  | Zimbabwe              |       14.1 |              17
 Afrique                  | Soudan                |       14.1 |              17
 Afrique                  | Tunisie               |       11.0 |              19
 Amérique du Nord         | États-Unis            |      320.1 |               1
 Amérique du Nord         | Canada                |       35.2 |               2
 Amérique latine. Caraïbes | Brésil               |      200.4 |               1
 Amérique latine. Caraïbes | Mexique              |      122.3 |               2
 Amérique latine. Caraïbes | Colombie             |       48.3 |               3
 Amérique latine. Caraïbes | Argentine            |       41.4 |               4
 Amérique latine. Caraïbes | Pérou                |       30.4 |               5
 Amérique latine. Caraïbes | Venezuela            |       30.4 |               5
 Amérique latine. Caraïbes | Chili                |       17.6 |               7
 Amérique latine. Caraïbes | Équateur             |       15.7 |               8
 Amérique latine. Caraïbes | Guatemala            |       15.5 |               9
 Amérique latine. Caraïbes | Cuba                 |       11.3 |              10
(...)
```

6.4.12 REGROUPEMENT ET TRI : PRINCIPE

```
OVER (PARTITION BY continent
      ORDER BY population DESC)
```



6.4.13 REGROUPEMENT ET TRI : SYNTAXE

```
SELECT ...
 <agregation> OVER (PARTITION BY <colonnes>
                    ORDER BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Cette construction ne pose aucune difficulté syntaxique. La norme impose de placer la clause `PARTITION BY` avant la clause `ORDER BY`, c'est la seule chose à retenir au niveau de la syntaxe.

6.4.14 FONCTIONS ANALYTIQUES

- PostgreSQL dispose d'un certain nombre de fonctions analytiques
- Utilité :
  - faire référence à d'autres lignes du même ensemble
  - évite les auto-jointures complexes et lentes

Sans les fonctions analytiques, il était difficile en SQL d'écrire des requêtes nécessitant de faire appel à des données provenant d'autres lignes que la ligne courante.

Par exemple, pour renvoyer la liste détaillée de tous les employés ET le salaire le plus élevé du service auquel il appartient, on peut utiliser la fonction `first_value()` :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
 matricule |   nom    | salaire  |   service   | salaire_maximum_service
-----------+----------+----------+-------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier    |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier    |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction   |                10000.00
 00000006  | Prunelle |  4000.00 | Publication |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication |                 4000.00
(5 lignes)
```

Il existe également les fonctions suivantes :

- `last_value(colonne)` : renvoie la dernière valeur pour la colonne ;
- `nth(colonne, n)` : renvoie la n-ème valeur (en comptant à partir de **1**) pour la colonne ;
- `lag(colonne, n)` : renvoie la valeur située en n-ème position **avant** la ligne en cours pour la colonne ;
- `lead(colonne, n)` : renvoie la valeur située en n-ème position **après** la ligne en cours pour la colonne ;
  - pour ces deux fonction, le n est facultatif et vaut **1** par défaut ;
  - ces deux fonctions acceptent un 3ème argument facultatif spécifiant la valeur à renvoyer si aucune valeur n'est trouvée en n-ème position avant ou après. Par défaut, `NULL` sera renvoyé.

6.4.15 LEAD() ET LAG()

- `lead(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **après** la ligne courante
- `lag(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **avant** la ligne courante

La construction `lead(colonne)` est équivalente à `lead(colonne, 1)`. De même, la construction `lag(colonne)` est équivalente à `lag(colonne, 1)`. Il s'agit d'un raccourci pour utiliser la valeur précédente ou la valeur suivante d'une colonne dans la fenêtre définie.

6.4.16 LEAD() ET LAG() : EXEMPLE

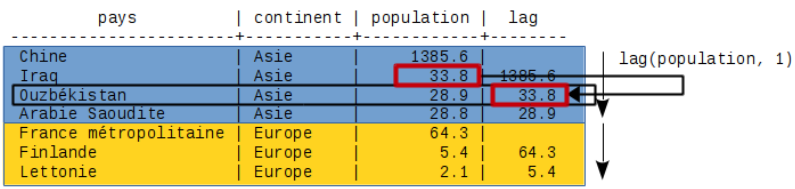
```
SELECT pays, continent, population,
       lag(population) OVER (PARTITION BY continent
                             ORDER BY population DESC)
  FROM population;
```

```
        pays          | continent | population |  lag
----------------------+-----------+------------+--------
 Chine                | Asie      |     1385.6 |
 Iraq                 | Asie      |       33.8 | 1385.6
 Ouzbékistan          | Asie      |       28.9 |   33.8
 Arabie Saoudite      | Asie      |       28.8 |   28.9
 France métropolitaine | Europe    |       64.3 |
 Finlande             | Europe    |        5.4 |   64.3
 Lettonie             | Europe    |        2.1 |    5.4
```

La requête présentée en exemple ne s'appuie que sur un jeu réduit de données afin de montrer un résultat compréhensible.

6.4.17 LEAD() ET LAG() : PRINCIPE

```
lag(population) OVER (PARTITION BY continent
                      ORDER BY population DESC)
```



NULL est renvoyé lorsque la valeur n'est pas accessible dans la fenêtre de données, comme par exemple si l'on souhaite utiliser la valeur d'une colonne appartenant à la ligne précédant la première ligne de la partition.

6.4.18 FIRST/LAST/NTH_VALUE

- `first_value(colonne)`
  - retourne la dernière valeur pour la colonne

- `last_value(colonne)`
  - retourne la dernière valeur pour la colonne
- `nth_value(colonne, n)`
  - retourne la n-ème valeur (en comptant à partir de 1) pour la colonne

Utilisé avec `ORDER BY` et `PARTITION BY`, la fonction `first_value()` permet par exemple d'obtenir le salaire le plus élevé d'un service :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
```

```
 matricule |   nom    | salaire  |   service   | salaire_maximum_service
-----------+----------+----------+-------------+------------------------
 00000004  | Fantasio |  4500.00 | Courrier    |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier    |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction   |                10000.00
 00000006  | Prunelle |  4000.00 | Publication |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication |                 4000.00
(5 lignes)
```

6.4.19 FIRST/LAST/NTH_VALUE : EXEMPLE

```
SELECT pays, continent, population,
       first_value(population)
           OVER (PARTITION BY continent
                 ORDER BY population DESC)
  FROM population;
```

```
       pays       | continent | population | first_value
------------------+-----------+------------+-------------
 Chine            | Asie      |     1385.6 |      1385.6
 Iraq             | Asie      |       33.8 |      1385.6
 Ouzbékistan      | Asie      |       28.9 |      1385.6
 Arabie Saoudite  | Asie      |       28.8 |      1385.6
 France           | Europe    |       64.3 |        64.3
 Finlande         | Europe    |        5.4 |        64.3
 Lettonie         | Europe    |        2.1 |        64.3
```

Lorsque que la clause `ORDER BY` est utilisée pour définir une fenêtre, la fenêtre visible depuis la ligne courante commence par défaut à la première ligne de résultat et s'arrête à la ligne courante.

Par exemple, si l'on exécute la même requête en utilisant `last_value()` plutôt que `first_value()`, on récupère à chaque fois la valeur de la colonne sur la ligne courante :

```
SELECT pays, continent, population,
       last_value(population) OVER (PARTITION BY continent
                                    ORDER BY population DESC)
  FROM population;
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Chine                 | Asie      |     1385.6 |     1385.6
 Iraq                  | Asie      |       33.8 |       33.8
 Ouzbékistan           | Asie      |       28.9 |       28.9
 Arabie Saoudite       | Asie      |       28.8 |       28.8
 France métropolitaine | Europe    |       64.3 |       64.3
 Finlande              | Europe    |        5.4 |        5.4
 Lettonie              | Europe    |        2.1 |        2.1
(7 rows)
```

Il est alors nécessaire de redéfinir le comportement de la fenêtre visible pour que la fonction se comporte comme attendu, en utilisant `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` - cet aspect sera décrit dans la section sur les possibilités de modification de la définition de la fenêtre.

6.4.20 CLAUSE WINDOW

- Pour factoriser la définition d'une fenêtre :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
 WINDOW w AS (ORDER BY salaire);
```

Il arrive que l'on ait besoin d'utiliser de plusieurs fonctions de fenêtrage au sein d'une même requête qui utilisent la même définition de fenêtre (même clause `PARTITION BY` et/ou `ORDER BY`). Afin d'éviter de dupliquer cette clause, il est possible de définir une fenêtre nommée et de l'utiliser à plusieurs endroits de la requête. Par exemple, l'exemple précédent des fonctions de classement pourrait s'écrire :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
  WINDOW w AS (ORDER BY salaire);
```

```
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

À noter qu'il est possible de définir de multiples définitions de fenêtres au sein d'une même requête, et qu'une définition de fenêtre peut surcharger la clause ORDER BY si la définition parente ne l'a pas définie. Par exemple, la requête SQL suivante est correcte :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w_asc,
       dense_rank() OVER w_desc
  FROM employes
  WINDOW w AS (PARTITION BY service),
         w_asc AS (w ORDER BY salaire),
         w_desc AS (w ORDER BY salaire DESC);
```

6.4.21 CLAUSE WINDOW : SYNTAXE

```
SELECT fonction_agregat OVER nom,
       fonction_agregat_2 OVER nom ...
       ...
  FROM <liste_tables>
 WHERE <predicats>
 WINDOW nom AS (PARTITION BY ... ORDER BY ...)
```

6.4.22 DÉFINITION DE LA FENÊTRE

- La fenêtre de travail par défaut est :

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

- Deux modes possibles :
  - RANGE
  - ROWS
- Nécessite une clause ORDER BY

6.4.23 DÉFINITION DE LA FENÊTRE : RANGE

- Indique un intervalle à bornes *flou*
- Borne de départ :
  - `UNBOUNDED PRECEDING`: depuis le début de la partition
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `UNBOUNDED FOLLOWING` : jusqu'à la fin de la partition
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
  ORDER BY ...
  RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

6.4.24 DÉFINITION DE LA FENÊTRE : ROWS

- Indique un intervalle borné par un nombre de ligne défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx valeurs devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx valeurs derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
  ORDER BY ...
  ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

6.4.25 DÉFINITION DE LA FENÊTRE : EXEMPLE

```
SELECT pays, continent, population,
       last_value(population)
        OVER (PARTITION BY continent ORDER BY population
              RANGE BETWEEN UNBOUNDED PRECEDING
                        AND UNBOUNDED FOLLOWING)
  FROM population;
```

```
        pays          | continent | population | last_value
----------------------+-----------+------------+------------
 Arabie Saoudite      | Asie      |       28.8 |     1385.6
```

```
 Ouzbékistan           | Asie      |       28.9 |     1385.6
 Iraq                  | Asie      |       33.8 |     1385.6
 Chine (4)             | Asie      |     1385.6 |     1385.6
 Lettonie              | Europe    |        2.1 |       64.3
 Finlande              | Europe    |        5.4 |       64.3
 France métropolitaine | Europe    |       64.3 |       64.3
```

6.5 WITHIN GROUP

- `WITHIN GROUP`
  - PostgreSQL 9.4
- Utilité :
  - calcul de médianes, centiles

La clause `WITHIN GROUP` est une nouvelle clause pour les agrégats utilisant des fonctions dont les données doivent être triées. Quelques fonctions ont été ajoutées pour profiter au mieux de cette nouvelle clause.

6.5.1 WITHIN GROUP : EXEMPLE

```
SELECT continent,
  percentile_disc(0.5)
    WITHIN GROUP (ORDER BY population) AS "mediane",
  percentile_disc(0.95)
    WITHIN GROUP (ORDER BY population) AS "95pct",
  ROUND(AVG(population), 1) AS moyenne
FROM population
 GROUP BY continent;
```

```
         continent          | mediane | 95pct   | moyenne
----------------------------+---------+---------+---------
 Afrique                    |    33.0 |  173.6  |    44.3
 Amérique du Nord           |    35.2 |  320.1  |   177.7
 Amérique latine. Caraïbes  |    30.4 |  200.4  |    53.3
 Asie                       |    53.3 | 1252.1  |   179.9
 Europe                     |     9.4 |   82.7  |    21.8
```

Cet exemple permet d'afficher le continent, la médiane de la population par continent et la population du pays le moins peuplé parmi les 5% de pays les plus peuplés de chaque continent.

17.12

Pour rappel, la table contient les données suivantes :

```
postgres=# SELECT * FROM population ORDER BY continent, population;
         pays          | population | superficie | densite | continent
-----------------------+------------+------------+---------+-----------
 Tunisie               |       11.0 |        164 |      67 | Afrique
 Zimbabwe              |       14.1 |        391 |      36 | Afrique
 Soudan                |       14.1 |        197 |      72 | Afrique
 Burkina Faso          |       16.9 |        274 |      62 | Afrique
(...)
```

En ajoutant le support de cette clause, PostgreSQL améliore son support de la norme SQL 2008 et permet le développement d'analyses statistiques plus élaborées.

6.6 GROUPING SETS

- `GROUPING SETS/ROLLUP/CUBE`
- Extension de `GROUP BY`
- PostgreSQL 9.5
- Utilité :
  - présente le résultat de plusieurs agrégations différentes
  - réaliser plusieurs agrégations différentes dans la même requête

Les `GROUPING SETS` permettent de définir plusieurs clauses d'agrégation `GROUP BY`. Les résultats seront présentés comme si plusieurs requêtes d'agrégation avec les clauses `GROUP BY` mentionnées étaient assemblées avec `UNION ALL`.

6.6.1 GROUPING SETS : JEU DE DONNÉES

---

stock		
piece	region	quantite
ecrous	est	50
ecrous	ouest	0
ecrous	sud	40
clous	est	70
clous	nord	40
vis	ouest	50
vis	sud	50
vis	nord	60

stock				
piece/region	est	ouest	sud	nord
ecrous	50	0	40	
clous	70			0
vis		50	50	60

```
CREATE TABLE stock AS SELECT * FROM (
    VALUES ('ecrous', 'est',   50),
           ('ecrous', 'ouest',  0),
           ('ecrous', 'sud',   40),
           ('clous',  'est',   70),
           ('clous',  'nord',   0),
           ('vis',    'ouest', 50),
           ('vis',    'sud',   50),
           ('vis',    'nord',  60)
) AS VALUES(piece, region, quantite);
```

6.6.2 GROUPING SETS : EXEMPLE VISUEL

sum (quantite) … grouping sets (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total	120	50	90	60	

6.6.3 GROUPING SETS : EXEMPLE ORDRE SQL

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
 piece  | region | sum
--------+--------+-----
 clous  |        |  70
 ecrous |        |  90
 vis    |        | 160
        | est    | 120
        | nord   |  60
        | ouest  |  50
        | sud    |  90
```

6.6.4 GROUPING SETS : ÉQUIVALENT

- On peut se passer de la clause `GROUPING SETS`
  - mais la requête sera plus lente

```
SELECT piece,NULL as region,sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region,sum(quantite)
  FROM STOCK
  GROUP BY region;
```

Le comportement de la clause `GROUPING SETS` peut être émulée avec deux requêtes utilisant chacune une clause `GROUP BY` sur les colonnes de regroupement souhaitées.

Cependant, le plan d'exécution de la requête équivalente conduit à deux lectures et peut être particulièrement coûteux si le jeu de données sur lequel on souhaite réaliser les agrégations est important :

```
EXPLAIN SELECT piece,NULL as region,sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region,sum(quantite)
  FROM STOCK
  GROUP BY region;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=1.12..2.38 rows=7 width=44)
   ->  HashAggregate  (cost=1.12..1.15 rows=3 width=45)
         Group Key: stock.piece
         ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=9)
   ->  HashAggregate  (cost=1.12..1.16 rows=4 width=44)
         Group Key: stock_1.region
         ->  Seq Scan on stock stock_1  (cost=0.00..1.08 rows=8 width=8)
```

La requête utilisant la clause `GROUPING SETS` propose un plan bien plus efficace :

```
EXPLAIN SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
                            QUERY PLAN
-------------------------------------------------------------------
 GroupAggregate  (cost=1.20..1.58 rows=14 width=17)
   Group Key: piece
   Sort Key: region
     Group Key: region
   ->  Sort  (cost=1.20..1.22 rows=8 width=13)
         Sort Key: piece
         ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=13)
```

6.6.5 ROLLUP

- `ROLLUP`
- PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête

La clause `ROLLUP` est une fonctionnalité d'analyse type OLAP du langage SQL. Elle s'utilise dans la clause `GROUP BY`, tout comme `GROUPING SETS`

6.6.6 ROLLUP : EXEMPLE VISUEL

sum (quantite) … ROLLUP (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total					320

6.6.7 ROLLUP : EXEMPLE ORDRE SQL

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY ROLLUP (piece,region);
```

Cette requête est équivalente à la requête suivante utilisant GROUPING SETS :

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY GROUPING SETS ((),(piece),(piece,region));
```

Sur une requête un peu plus intéressante, effectuant des statistiques sur des ventes :

```
SELECT type_client, code_pays, SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
```

```
 A           | DE         |    6642061.57
 A           | DZ         |    6404425.16
 A           | FR         |   55261295.52
 A           | IN         |    7224008.95
 A           | PE         |    7356239.93
 A           | RU         |    6766644.98
 A           | US         |    7700691.07
 A           |            |  111557177.00
(...)
 P           | RU         |  287605812.99
 P           | US         |  296424154.49
 P           |            | 4692152751.08
             |            | 5217862160.65
```

Une fonction GROUPING, associée à ROLLUP, permet de déterminer si la ligne courante correspond à un regroupement donné. Elle est de la forme d'un masque de bit converti au format décimal :

```
SELECT row_number()
         OVER ( ORDER BY grouping(piece,region)) AS ligne,
       grouping(piece,region)::bit(2) AS g,
       piece,
       region,
       sum(quantite)
FROM stock
GROUP BY CUBE (piece,region)
ORDER BY g ;
```

```
 ligne | g  | piece  | region | sum
-------+----+--------+--------+-----
     1 | 00 | clous  | est    | 150
     2 | 00 | clous  | nord   |  10
     3 | 00 | ecrous | est    | 110
     4 | 00 | ecrous | ouest  |  10
     5 | 00 | ecrous | sud    |  90
     6 | 00 | vis    | nord   | 130
     7 | 00 | vis    | ouest  | 110
     8 | 00 | vis    | sud    | 110
     9 | 01 | vis    |        | 350
    10 | 01 | ecrous |        | 210
    11 | 01 | clous  |        | 160
    12 | 10 |        | ouest  | 120
    13 | 10 |        | sud    | 200
    14 | 10 |        | est    | 260
    15 | 10 |        | nord   | 140
    16 | 11 |        |        | 720
```

Voici un autre exemple :

```
SELECT COALESCE(service,
    CASE
        WHEN GROUPING(service) = 0 THEN 'Unknown' ELSE 'Total'
    END) AS service,
       sum(salaire) AS salaires_service, count(*) AS nb_employes
  FROM employes
 GROUP BY ROLLUP (service);
  service    | salaires_service | nb_employes
-------------+------------------+-------------
 Courrier    |          7500.00 |           2
 Direction   |         50000.00 |           1
 Publication |          7000.00 |           2
 Total       |         64500.00 |           5
(4 rows)
```

Ou appliqué à l'exemple un peu plus complexe :

```
SELECT COALESCE(type_client,
       CASE
         WHEN GROUPING(type_client) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS type_client,
       COALESCE(code_pays,
       CASE
         WHEN GROUPING(code_pays) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
```

```
 A            | RU         |    6766644.98
 A            | US         |    7700691.07
 A            | Total      |  111557177.00
(...)
 P            | US         |  296424154.49
 P            | Total      | 4692152751.08
 Total        | Total      | 5217862160.65
```

6.6.8 CUBE

- `CUBE`
  - PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête
  - sur toutes les clauses de regroupement

La clause `CUBE` est une autre fonctionnalité d'analyse type OLAP du langage SQL. Tout comme `ROLLUP`, elle s'utilise dans la clause `GROUP BY`.

6.6.9 CUBE : EXEMPLE VISUEL

sum (quantite) … CUBE (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total	120	50	90	60	320

6.6.10 CUBE : EXEMPLE ORDRE SQL

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY CUBE (piece,region);
```

Cette requête est équivalente à la requête suivante utilisant GROUPING SETS :

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY GROUPING SETS (
  (),
  (piece),
  (region),
  (piece,region)
  );
```

Elle permet de réaliser des regroupements sur l'ensemble des combinaisons possibles des clauses de regroupement indiquées. Pour de plus amples détails, se référer à cet article[47]
.

En reprenant la requête de l'exemple précédent :

```
SELECT type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle retournera le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
```

[47] http://www.wikiwand.com/en/OLAP_cube#/overview

```
A          | RU        |    6766644.98
A          | US        |    7700691.07
A          |           |  111557177.00
E          | CA        |   28457655.81
E          | CN        |   25537539.68
E          | DE        |   25508815.68
E          | DZ        |   24821750.17
E          | FR        |  209402443.24
E          | IN        |   26788642.27
E          | PE        |   24541974.54
E          | RU        |   25397116.39
E          | US        |   23696294.79
E          |           |  414152232.57
P          | CA        |  292975985.52
P          | CN        |  287795272.87
P          | DE        |  287337725.21
P          | DZ        |  302501132.54
P          | FR        | 2341977444.49
P          | IN        |  295256262.73
P          | PE        |  300278960.24
P          | RU        |  287605812.99
P          | US        |  296424154.49
P          |           | 4692152751.08
           |           | 5217862160.65
           | CA        |  327706809.65
           | CN        |  321261454.05
           | DE        |  319488602.46
           | DZ        |  333727307.87
           | FR        | 2606641183.25
           | IN        |  329268913.95
           | PE        |  332177174.71
           | RU        |  319769574.36
           | US        |  327821140.35
```

Dans ce genre de contexte, lorsque le regroupement est réalisé sur l'ensemble des valeurs d'un critère de regroupement, alors la valeur qui apparaît est `NULL` pour la colonne correspondante. Si la colonne possède des valeurs `NULL` légitimes, il est alors difficile de les distinguer. On utilise alors la fonction `GROUPING()` qui permet de déterminer si le regroupement porte sur l'ensemble des valeurs de la colonne. L'exemple suivant montre une requête qui exploite cette fonction :

17.12

```
SELECT GROUPING(type_client,code_pays)::bit(2),
       GROUPING(type_client)::boolean g_type_cli,
       GROUPING(code_pays)::boolean g_code_pays,
       type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 grouping | g_type_cli | g_code_pays | type_client | code_pays |    montant
----------+------------+-------------+-------------+-----------+---------------
 00       | f          | f           | A           | CA        |    6273168.32
 00       | f          | f           | A           | CN        |    7928641.50
 00       | f          | f           | A           | DE        |    6642061.57
 00       | f          | f           | A           | DZ        |    6404425.16
 00       | f          | f           | A           | FR        |   55261295.52
 00       | f          | f           | A           | IN        |    7224008.95
 00       | f          | f           | A           | PE        |    7356239.93
 00       | f          | f           | A           | RU        |    6766644.98
 00       | f          | f           | A           | US        |    7700691.07
 01       | f          | t           | A           |           |  111557177.00
(...)
 01       | f          | t           | P           |           | 4692152751.08
 11       | t          | t           |             |           | 5217862160.65
 10       | t          | f           |             | CA        |  327706809.65
 10       | t          | f           |             | CN        |  321261454.05
 10       | t          | f           |             | DE        |  319488602.46
 10       | t          | f           |             | DZ        |  333727307.87
 10       | t          | f           |             | FR        | 2606641183.25
 10       | t          | f           |             | IN        |  329268913.95
 10       | t          | f           |             | PE        |  332177174.71
 10       | t          | f           |             | RU        |  319769574.36
 10       | t          | f           |             | US        |  327821140.35
```

```
(40 rows)
```

L'application sera alors à même de gérer la présentation des résultats en fonction des valeurs de `grouping` ou `g_type_client` et `g_code_pays`.

6.7 TRAVAUX PRATIQUES

6.7.1 ÉNONCÉS

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

```
     Table "public.brno_2015"
  Column   |   Type   | Modifiers
-----------+----------+-----------
 no_tour   | integer  |
 no_pilote | integer  |
 lap_time  | interval |
```

Une table `pilotes` permet de connaître les détails d'un pilote :

```
      Table "public.pilotes"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 no          | integer |
 nom         | text    |
 nationalite | text    |
 ecurie      | text    |
 moto        | text    |
```

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

Agrégation

1. Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?
2. Déterminer quel est le pilote le plus régulier (écart-type).

Window Functions

3. Afficher la place sur le podium pour chaque coureur.
4. À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.
5. Pour chaque tour, afficher :

- le nom du pilote ;
- son rang dans le tour ;
- son temps depuis le début de la course ;
- dans le tour, la différence de temps par rapport au premier.

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?
7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.
8. En quelle position a terminé le coureur qui a doublé le plus de personnes ? Combien de personnes a-t-il doublées ?

Grouping Sets

Ce TP nécessite PostgreSQL 9.5 ou supérieur. Il s'appuie sur les tables présentes dans le schéma `magasin`.

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.
10. Ajouter également le montant total des commandes depuis le début de l'activité.
11. Ajouter également le montant total des commandes par pays.

6.7.2 CORRECTIONS

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

```
     Table "public.brno_2015"
  Column   |   Type   | Modifiers
-----------+----------+-----------
 no_tour   | integer  |
 no_pilote | integer  |
 lap_time  | interval |
```

Une table `pilotes` permet de connaître les détails d'un pilote :

```
      Table "public.pilotes"
   Column     |  Type   | Modifiers
--------------+---------+-----------
 no           | integer |
 nom          | text    |
 nationalite  | text    |
 ecurie       | text    |
 moto         | text    |
```

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

Agrégation

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

1. Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?

Le coureur :

```
SELECT nom, max(lap_time) - min(lap_time) as ecart
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 GROUP BY 1
 ORDER BY 2
 LIMIT 1;
```

La requête donne le résultat suivant :

```
       nom        |    ecart
------------------+--------------
 Jorge LORENZO | 00:00:04.661
(1 row)
```

2. Déterminer quel est le pilote le plus régulier (écart-type).

Nous excluons le premier tour car il s'agit d'une course avec départ arrêté, donc ce tour est plus lent que les autres, ici d'au moins 8 secondes :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
```

```
 WHERE no_tour > 1
 GROUP BY 1
 ORDER BY 2
 LIMIT 1;
```

Le résultat montre le coureur qui a abandonné en premier :

```
       nom        |       stddev
------------------+--------------------
 Alex DE ANGELIS | 0.130107647741847
(1 row)
```

On s'aperçoit qu'Alex De Angelis n'a pas terminé la course. Il semble donc plus intéressant de ne prendre en compte que les pilotes qui ont terminé la course et toujours en excluant le premier tour (il y a 22 tours sur cette course, on peut le positionner soit en dur dans la requête, soit avec un sous-select permettant de déterminer le nombre maximum de tours) :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 WHERE no_tour > 1
   AND no_pilote in (SELECT no_pilote FROM brno_2015 WHERE no_tour=22)
GROUP BY 1
ORDER BY 2
LIMIT 1;
```

Le pilote 19 a donc été le plus régulier :

```
       nom        |       stddev
------------------+--------------------
 Alvaro BAUTISTA | 0.222825823492654
```

Window Functions

Si ce n'est pas déjà fait, nous positionnons le search_path pour chercher les objets du schéma brno2015 :

```
SET search_path = brno2015;
```

3. Afficher la place sur le podium pour chaque coureur.

Les coureurs qui ne franchissent pas la ligne d'arrivée sont dans le classement malgré tout. Il faut donc tenir compte de cela dans l'affichage des résultats.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
       nom, ecurie, total_time
  FROM (SELECT no_pilote,
```

```
            sum(lap_time) over (PARTITION BY no_pilote) as total_time,
            max(no_tour) over (PARTITION BY no_pilote) as max_lap
        FROM brno_2015
    ) AS race_data
  JOIN pilotes
    ON (race_data.no_pilote = pilotes.no)
 GROUP BY nom, ecurie, max_lap, total_time
 ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

```
 rang |       nom         |           ecurie             |  total_time
------+-------------------+------------------------------+--------------
    1 | Jorge LORENZO     | Movistar Yamaha MotoGP       | 00:42:53.042
    2 | Marc MARQUEZ      | Repsol Honda Team            | 00:42:57.504
    3 | Valentino ROSSI   | Movistar Yamaha MotoGP       | 00:43:03.439
    4 | Andrea IANNONE    | Ducati Team                  | 00:43:06.113
    5 | Dani PEDROSA      | Repsol Honda Team            | 00:43:08.692
    6 | Andrea DOVIZIOSO  | Ducati Team                  | 00:43:08.767
    7 | Bradley SMITH     | Monster Yamaha Tech 3        | 00:43:14.863
    8 | Pol ESPARGARO     | Monster Yamaha Tech 3        | 00:43:16.282
    9 | Aleix ESPARGARO   | Team SUZUKI ECSTAR           | 00:43:36.826
   10 | Danilo PETRUCCI   | Octo Pramac Racing           | 00:43:38.303
   11 | Yonny HERNANDEZ   | Octo Pramac Racing           | 00:43:43.015
   12 | Scott REDDING     | EG 0,0 Marc VDS              | 00:43:43.216
   13 | Alvaro BAUTISTA   | Aprilia Racing Team Gresini  | 00:43:47.479
   14 | Stefan BRADL      | Aprilia Racing Team Gresini  | 00:43:47.666
   15 | Loris BAZ         | Forward Racing               | 00:43:53.358
   16 | Hector BARBERA    | Avintia Racing               | 00:43:54.637
   17 | Nicky HAYDEN      | Aspar MotoGP Team            | 00:43:55.43
   18 | Mike DI MEGLIO    | Avintia Racing               | 00:43:58.986
   19 | Jack MILLER       | CWM LCR Honda                | 00:44:04.449
   20 | Claudio CORTI     | Forward Racing               | 00:44:43.075
   21 | Karel ABRAHAM     | AB Motoracing                | 00:44:55.697
   22 | Maverick VIÑALES  | Team SUZUKI ECSTAR           | 00:29:31.557
   23 | Cal CRUTCHLOW     | CWM LCR Honda                | 00:27:38.315
   24 | Eugene LAVERTY    | Aspar MotoGP Team            | 00:08:04.096
   25 | Alex DE ANGELIS   | E-Motion IodaRacing Team     | 00:06:05.782
(25 rows)
```

4. À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.

17.12

La requête n'est pas beaucoup modifiée, seule la fonction `first_value()` est utilisée pour déterminer le temps du vainqueur, temps qui sera ensuite retranché au temps du coureur courant.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
       nom, ecurie, total_time,
       total_time - first_value(total_time)
         OVER (ORDER BY max_lap desc, total_time asc) AS difference
  FROM (SELECT no_pilote,
               sum(lap_time) over (PARTITION BY no_pilote) as total_time,
               max(no_tour) over (PARTITION BY no_pilote) as max_lap
          FROM brno_2015
       ) AS race_data
  JOIN pilotes
    ON (race_data.no_pilote = pilotes.no)
 GROUP BY nom, ecurie, max_lap, total_time
 ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

```
r |       nom       |          ecurie       | total_time  |  difference
--+-----------------+-----------------------+-------------+---------------
 1| Jorge LORENZO   | Movistar Yamaha [...]|00:42:53.042 | 00:00:00
 2| Marc MARQUEZ    | Repsol Honda Team     |00:42:57.504 | 00:00:04.462
 3| Valentino ROSSI | Movistar Yamaha [...]|00:43:03.439 | 00:00:10.397
 4| Andrea IANNONE  | Ducati Team           |00:43:06.113 | 00:00:13.071
 5| Dani PEDROSA    | Repsol Honda Team     |00:43:08.692 | 00:00:15.65
 6| Andrea DOVIZIOSO| Ducati Team           |00:43:08.767 | 00:00:15.725
 7| Bradley SMITH   | Monster Yamaha Tech 3|00:43:14.863 | 00:00:21.821
 8| Pol ESPARGARO   | Monster Yamaha Tech 3|00:43:16.282 | 00:00:23.24
 9| Aleix ESPARGARO | Team SUZUKI ECSTAR    |00:43:36.826 | 00:00:43.784
10| Danilo PETRUCCI | Octo Pramac Racing    |00:43:38.303 | 00:00:45.261
11| Yonny HERNANDEZ | Octo Pramac Racing    |00:43:43.015 | 00:00:49.973
12| Scott REDDING   | EG 0,0 Marc VDS       |00:43:43.216 | 00:00:50.174
13| Alvaro BAUTISTA | Aprilia Racing [...] |00:43:47.479 | 00:00:54.437
14| Stefan BRADL    | Aprilia Racing [...] |00:43:47.666 | 00:00:54.624
15| Loris BAZ       | Forward Racing        |00:43:53.358 | 00:01:00.316
16| Hector BARBERA  | Avintia Racing        |00:43:54.637 | 00:01:01.595
17| Nicky HAYDEN    | Aspar MotoGP Team     |00:43:55.43  | 00:01:02.388
18| Mike DI MEGLIO  | Avintia Racing        |00:43:58.986 | 00:01:05.944
19| Jack MILLER     | CWM LCR Honda         |00:44:04.449 | 00:01:11.407
20| Claudio CORTI   | Forward Racing        |00:44:43.075 | 00:01:50.033
21| Karel ABRAHAM   | AB Motoracing         |00:44:55.697 | 00:02:02.655
```

```
22| Maverick VIÑALES| Team SUZUKI ECSTAR   |00:29:31.557 | -00:13:21.485
23| Cal CRUTCHLOW    | CWM LCR Honda        |00:27:38.315 | -00:15:14.727
24| Eugene LAVERTY   | Aspar MotoGP Team    |00:08:04.096 | -00:34:48.946
25| Alex DE ANGELIS | E-Motion Ioda[...]     |00:06:05.782 | -00:36:47.26
(25 rows)
```

5. Pour chaque tour, afficher :

- le nom du pilote ;
- son rang dans le tour ;
- son temps depuis le début de la course ;
- dans le tour, la différence de temps par rapport au premier.

Pour construire cette requête, nous avons besoin d'obtenir le temps cumulé tour après tour pour chaque coureur. Nous commençons donc par écrire une première requête :

```
SELECT *,
       SUM(lap_time)
         OVER (PARTITION BY no_pilote ORDER BY no_tour) AS temps_tour_glissant
 FROM brno_2015
```

Elle retourne le résultat suivant :

```
 no_tour | no_pilote |   lap_time   | temps_tour_glissant
---------+-----------+--------------+---------------------
       1 |         4 | 00:02:02.209 | 00:02:02.209
       2 |         4 | 00:01:57.57  | 00:03:59.779
       3 |         4 | 00:01:57.021 | 00:05:56.8
       4 |         4 | 00:01:56.943 | 00:07:53.743
       5 |         4 | 00:01:57.012 | 00:09:50.755
       6 |         4 | 00:01:57.011 | 00:11:47.766
       7 |         4 | 00:01:57.313 | 00:13:45.079
       8 |         4 | 00:01:57.95  | 00:15:43.029
       9 |         4 | 00:01:57.296 | 00:17:40.325
      10 |         4 | 00:01:57.295 | 00:19:37.62
      11 |         4 | 00:01:57.185 | 00:21:34.805
      12 |         4 | 00:01:57.45  | 00:23:32.255
      13 |         4 | 00:01:57.457 | 00:25:29.712
      14 |         4 | 00:01:57.362 | 00:27:27.074
      15 |         4 | 00:01:57.482 | 00:29:24.556
      16 |         4 | 00:01:57.358 | 00:31:21.914
      17 |         4 | 00:01:57.617 | 00:33:19.531
      18 |         4 | 00:01:57.594 | 00:35:17.125
```

```
      19 |          4 | 00:01:57.412 | 00:37:14.537
      20 |          4 | 00:01:57.786 | 00:39:12.323
      21 |          4 | 00:01:58.087 | 00:41:10.41
      22 |          4 | 00:01:58.357 | 00:43:08.767
(...)
```

Cette requête de base est ensuite utilisée dans une CTE qui sera utilisée par la requête répondant à la question de départ. La colonne `temps_tour_glissant` est utilisée pour calculer le rang du pilote dans la course, est affiché et le temps cumulé du meilleur pilote est récupéré avec la fonction `first_value` :

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time)
      OVER (PARTITION BY no_pilote
            ORDER BY no_tour
            )  as temps_tour_glissant
    FROM brno_2015
    ORDER BY no_pilote, no_tour
)

SELECT no_tour, nom,
rank() OVER (PARTITION BY no_tour
             ORDER BY temps_tour_glissant ASC
            ) as place_course,
temps_tour_glissant,
temps_tour_glissant - first_value(temps_tour_glissant)
OVER (PARTITION BY no_tour
      ORDER BY temps_tour_glissant asc
    ) AS difference
FROM temps_glissant t
JOIN pilotes p ON p.no = t.no_pilote;
```

On pouvait également utiliser une simple sous-requête pour obtenir le même résultat :

```
SELECT no_tour,
  nom,
  rank()
    OVER (PARTITION BY no_tour
          ORDER BY temps_tour_glissant ASC
          ) AS place_course,
  temps_tour_glissant,
  temps_tour_glissant - first_value(temps_tour_glissant)
    OVER (PARTITION BY no_tour
           ORDER BY temps_tour_glissant asc
          ) AS difference
FROM (
```

```
  SELECT *, SUM(lap_time)
    OVER (PARTITION BY no_pilote
          ORDER BY no_tour)
          AS temps_tour_glissant
  FROM brno_2015) course
  JOIN pilotes
    ON (pilotes.no = course.no_pilote)
ORDER BY no_tour;
```

La requête fournit le résultat suivant :

```
no.|       nom         | place_c. | temps_tour_glissant |  difference
---+-------------------+----------+---------------------+--------------
 1 | Jorge LORENZO     |        1 | 00:02:00.83         | 00:00:00
 1 | Marc MARQUEZ      |        2 | 00:02:01.058        | 00:00:00.228
 1 | Andrea DOVIZIOSO  |        3 | 00:02:02.209        | 00:00:01.379
 1 | Valentino ROSSI   |        4 | 00:02:02.329        | 00:00:01.499
 1 | Andrea IANNONE    |        5 | 00:02:02.597        | 00:00:01.767
 1 | Bradley SMITH     |        6 | 00:02:02.861        | 00:00:02.031
 1 | Pol ESPARGARO     |        7 | 00:02:03.239        | 00:00:02.409
(...)
 2 | Jorge LORENZO     |        1 | 00:03:57.073        | 00:00:00
 2 | Marc MARQUEZ      |        2 | 00:03:57.509        | 00:00:00.436
 2 | Valentino ROSSI   |        3 | 00:03:59.696        | 00:00:02.623
 2 | Andrea DOVIZIOSO  |        4 | 00:03:59.779        | 00:00:02.706
 2 | Andrea IANNONE    |        5 | 00:03:59.9          | 00:00:02.827
 2 | Bradley SMITH     |        6 | 00:04:00.355        | 00:00:03.282
 2 | Pol ESPARGARO     |        7 | 00:04:00.87         | 00:00:03.797
 2 | Maverick VIÑALES  |        8 | 00:04:01.187        | 00:00:04.114
(...)
(498 rows)
```

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?

Il est ici nécessaire de sélectionner pour chaque tour le temps du meilleur tour. On peut alors sélectionner les tours pour lequels le temps du tour est égal au meilleur temps :

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time)
        OVER (PARTITION BY no_pilote
              ORDER BY no_tour
             ) as temps_tour_glissant
    FROM brno_2015
```

```
    ORDER BY no_pilote, no_tour
),

classement_tour AS (
    SELECT no_tour, no_pilote, lap_time,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant,
    min(lap_time) OVER (PARTITION BY no_pilote) as meilleur_temps
    FROM temps_glissant
)

SELECT no_tour, nom, place_course, lap_time
FROM classement_tour t
JOIN pilotes p ON p.no = t.no_pilote
WHERE lap_time = meilleur_temps;
```

Ce qui donne le résultat suivant :

```
 no_tour |       nom        | place_course |   lap_time
---------+------------------+--------------+--------------
       4 | Jorge LORENZO    |            1 | 00:01:56.169
       4 | Marc MARQUEZ     |            2 | 00:01:56.048
       4 | Valentino ROSSI  |            3 | 00:01:56.747
       6 | Andrea IANNONE   |            5 | 00:01:56.86
       6 | Dani PEDROSA     |            7 | 00:01:56.975
       4 | Andrea DOVIZIOSO |            4 | 00:01:56.943
       3 | Bradley SMITH    |            6 | 00:01:57.25
      17 | Pol ESPARGARO    |            8 | 00:01:57.454
       4 | Aleix ESPARGARO  |           12 | 00:01:57.844
       4 | Danilo PETRUCCI  |           11 | 00:01:58.121
       9 | Yonny HERNANDEZ  |           14 | 00:01:58.53
       2 | Scott REDDING    |           14 | 00:01:57.976
       3 | Alvaro BAUTISTA  |           21 | 00:01:58.71
       3 | Stefan BRADL     |           16 | 00:01:58.38
       3 | Loris BAZ        |           19 | 00:01:58.679
       2 | Hector BARBERA   |           15 | 00:01:58.405
       2 | Nicky HAYDEN     |           16 | 00:01:58.338
       3 | Mike DI MEGLIO   |           18 | 00:01:58.943
       4 | Jack MILLER      |           22 | 00:01:59.007
       2 | Claudio CORTI    |           24 | 00:02:00.377
```

```
      14 | Karel ABRAHAM    |           23 | 00:02:01.716
       3 | Maverick VIÑALES |            8 | 00:01:57.436
       3 | Cal CRUTCHLOW    |           11 | 00:01:57.652
       3 | Eugene LAVERTY   |           20 | 00:01:58.977
       3 | Alex DE ANGELIS  |           23 | 00:01:59.257
(25 rows)
```

7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.

```
WITH nb_tour AS (
    SELECT max(no_tour) FROM brno_2015
),
temps_glissant AS (
    SELECT  no_tour, no_pilote, lap_time,
    sum(lap_time) OVER (
        PARTITION BY no_pilote
        ORDER BY no_tour
    ) as temps_tour_glissant,
    max(no_tour) OVER (PARTITION BY no_pilote) as total_tour
    FROM brno_2015
),
classement_tour AS (
    SELECT no_tour, no_pilote, lap_time, total_tour,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course
    FROM temps_glissant
)
SELECT no_pilote
FROM classement_tour t
JOIN nb_tour n ON n.max = t.total_tour
GROUP BY no_pilote
HAVING count(DISTINCT place_course) = 1;
```

Elle retourne le résultat suivant :

```
 no_pilote
-----------
        93
        99
(2 lignes)
```

8. En quelle position a terminé le coureur qui a doublé le plus de personnes. Combien de personnes a-t-il doublées ?

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time) OVER (
        PARTITION BY no_pilote
        ORDER BY no_tour
    ) as temps_tour_glissant
    FROM brno_2015
),
classement_tour AS (
    SELECT no_tour, no_pilote, lap_time,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant
    FROM temps_glissant
),
depassement  AS (
    SELECT no_pilote,
      last_value(place_course) OVER (PARTITION BY no_pilote) as rang,
    CASE
        WHEN lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course < 0
        THEN 0
        ELSE lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course
        END AS depasse
    FROM classement_tour t
)

SELECT no_pilote, rang, sum(depasse)
FROM depassement
GROUP BY no_pilote, rang
ORDER BY sum(depasse) DESC
LIMIT 1;
```

Grouping Sets

La suite de ce TP est maintenant réalisé avec la base de formation habituelle. Attention, ce TP nécessite l'emploi d'une version 9.5 ou supérieure de PostgreSQL.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY GROUPING SETS (
   (extract('year' from date_commande), code_pays),
   (extract('year' from date_commande))
 );
```

Le résultat attendu est :

```
 annee | code_pays | montant_total_commande
-------+-----------+------------------------
  2003 | DE        |                49634.24
  2003 | FR        |                10003.98
  2003 |           |                59638.22
  2008 | CA        |              1016082.18
  2008 | CN        |               801662.75
  2008 | DE        |               694787.87
  2008 | DZ        |               663045.33
  2008 | FR        |              5860607.27
  2008 | IN        |               741850.87
  2008 | PE        |              1167825.32
  2008 | RU        |               577164.50
  2008 | US        |               928661.06
  2008 |           |             12451687.15
(...)
```

10. Ajouter également le montant total des commandes depuis le début de l'activité.

L'opérateur de regroupement `ROLL UP` amène le niveau d'agrégation sans regroupement :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
```

```
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY ROLLUP (extract('year' from date_commande), code_pays);
```

11. Ajouter également le montant total des commandes par pays.

Cette fois, l'opérateur `CUBE` permet d'obtenir l'ensemble de ces informations :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

12. À partir de la requête précédente, ajouter une colonne par critère de regroupement, de type booléen, qui est positionnée à `true` lorsque le regroupement est réalisé sur l'ensemble des valeurs de la colonne.

Ces colonnes booléennes permettent d'indiquer à l'application comment gérer la présentation des résultats.

```
SELECT grouping(extract('year' from date_commande))::boolean AS g_annee,
       grouping(code_pays)::boolean AS g_pays,
       extract('year' from date_commande) AS annee,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY CUBE (extract('year' from date_commande), code_pays);
```