Optimisation et index

Bases de données 2

Thibaut MADELAINE

octobre 2023

Chers lectrices & lecteurs,

Cette formation PostgreSQL est issue des manuels Dalibo. Ils ont été repris par Thibaut MADELAINE pour rentrer dans le format universitaire avec Cours Magistraux, Travaux Dirigés (sans ordinateurs) et Travaux Pratiques (avec ordinateur).

Au-delà du contenu technique en lui-même, l'intention des auteurs est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de cette formation 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 sur le site gitlab https://gitlab.com/madtibo/cours_dba_pg_universite/-/issues !

Optimisations et Index

Programme de ce cours

- Semaine 1 : découverte de PostgreSQL
- Semaine 2 : transactions et accès concurrents
- Semaine 3 : missions du DBA
- **Semaine 4** : optimisation et indexation
  - Principes de l'optimisation
  - Comprendre Explain
  - PostgreSQL et les statistiques
  - Problèmes courants

- Semaine 5 : *PL/PgSQL* et triggers

Introduction

- L'optimisation doit porter sur les différents composants
  - le serveur qui héberge le SGBDR : le matériel, la distribution, le noyau, les systèmes de fichiers
  - 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 et les requêtes

Pour qu'une optimisation soit réussie, il faut absolument tenir compte de tous les éléments ayant une responsabilité dans les performances. Cela commence avec le matériel. Il ne sert à rien d'améliorer la configuration du serveur PostgreSQL ou les requêtes si, physiquement, le serveur ne peut tenir la charge, que cela soit la cause des processeurs, de la mémoire, du disque ou du réseau. Le matériel est donc un point important à vérifier dans chaque tentative d'optimisation. De même, le système d'exploitation est pour beaucoup dans les performances de PostgreSQL : son choix et sa configuration ne doivent pas être laissés au hasard. La configuration du moteur a aussi son importance et cette partie permettra de faire la liste des paramètres importants dans le seul cadre des performances. Même l'organisation des fichiers dans les partitions des systèmes disques a un intérêt.

L'optimisation (aussi appelé `tuning`) doit donc être réalisée sur tous ces éléments **à la fois** pour être optimal !

Menu

- Quelques considérations générales sur l'optimisation
- Choix et configuration du matériel
- Choix et configuration du système d'exploitation
- Configuration du serveur de bases de données

Considérations générales - 1

- Deux points déterminants :
  - Vision globale du système d'information
  - Compréhension de l'utilisation de la base

Il est très difficile d'optimiser un serveur de bases de données sans savoir comment ce dernier va être utilisé. Par exemple, le nombre de requêtes à exécuter simultanément et leur complexité est un excellent indicateur pour mieux apprécier le nombre de cœurs à placer sur un serveur. Il est donc important de connaître la façon dont les applications travaillent avec les bases. Cela permet de mieux comprendre si le matériel est adéquat, s'il faut changer telle ou telle configuration, etc. Cela permet aussi de mieux configurer son système de supervision.

Considérations générales - 2

- L'optimisation n'est pas un processus unique
  - il s'agit au contraire d'un processus itératif
- La base doit être surveillée régulièrement !
  - nécessité d'installer des outils de supervision

Après avoir installé le serveur et l'avoir optimisé du mieux possible, la configuration optimale réalisée à ce moment ne sera bonne que pendant un certain temps. Si le service gagne en popularité, le nombre d'utilisateurs peut augmenter. La base va de toute façon grossir. Autrement dit, les conditions initiales vont changer. Un serveur optimisé pour dix utilisateurs en concurrence ne le sera plus pour 50 utilisateurs en concurrence. La configuration d'une base de 10 Go n'est pas la même que celle d'une base de 1 To.

Cette évolution doit donc être surveillée à travers un système de supervision et métrologie approprié et compris. Lorsqu'un utilisateur se plaint d'une impression de lenteur sur le système, ces informations collectées rendent souvent la tâche d'inspection plus rapide. Ainsi, l'identification du ou des paramètres à modifier, ou plus généralement des actions à réaliser pour corriger le problème, est plus aisée et repose sur une vision fiable et réelle de l'activité de l'instance.

Le plus important est donc de bien comprendre qu'un SGBD ne s'optimise pas qu'une seule fois, mais que ce travail d'optimisation sera à faire plusieurs fois au fur et à mesure de la vie du serveur.

À une échelle beaucoup plus petite, un travail d'optimisation sur une requête peut forcer à changer la configuration d'un paramètre. Cette modification peut faire gagner énormément sur cette requête… et perdre encore plus sur les autres. Là-aussi, tout travail d'optimisation doit être fait prudemment et ses effets surveillés sur une certaine période pour s'assurer que cette amélioration ne s'accompagne pas de quelques gros inconvénients.

Matériel

- Performances très liées aux possibilités du matériel
- Quatre composants essentiels
  - les processeurs
  - la mémoire
  - les disques
  - le système disque (RAID, SAN)

PostgreSQL est un système qui se base fortement sur le matériel et le système d'exploitation. Il est donc important que ces deux composants soient bien choisis et bien configurés pour que PostgreSQL fonctionne de façon optimale pour les performances.

Au niveau du matériel, les composants essentiels sont :

- les processeurs (CPU) ;
- la mémoire (RAM) ;
- les disques ;
- le système disque (carte RAID, baie SAN, etc).

CPU

- Trois critères importants
  - nombre de cœurs
  - fréquence
  - cache
- Privilégier

  - le nombre de cœurs si le nombre de sessions parallèles est important
  - ou la fréquence si les requêtes sont complexes

PostgreSQL est un système multi-processus. Chaque connexion d'un client est gérée par un processus, responsable de l'exécution des requêtes et du renvoi des données au client. Ce processus n'est pas multi-threadé. Par conséquent, chaque requête exécutée est traitée par un cœur de processeur. Plus vous voulez pouvoir exécuter de requêtes en parallèle, plus vous devez avoir de processeurs (ou plus exactement de cœurs). On considère habituellement qu'un cœur peut traiter de 4 à 20 requêtes simultanément. Cela dépend notamment beaucoup des requêtes, de leur complexité, de la quantité de donnée manipulée et retournée, etc. Il est donc essentiel de connaître le nombre de requêtes traitées simultanément pour le nombre d'utilisateurs connectés.
S'il s'agit d'un SGBD pour une application web, il y a de fortes chances que le nombre de requêtes en parallèle soit assez élevé. Dans ce contexte, il faut prévoir un grand nombre de cœur processeurs. En revanche, sur un entrepôt de données, nous trouvons habituellement peu d'utilisateurs avec des requêtes complexes et gourmandes en ressources. Dans ce cas, beaucoup de processeurs n'apporteront rien. Mieux vaut peu de cœur, mais que ces derniers soient plus puissants afin de répondre plus efficacement aux besoins importants de calculs complexe.

Ainsi, la fréquence (et donc la puissance) des processeurs est un point important à considérer. Il peut faire la différence si les requêtes à exécuter sont complexes : temps de planification réduit, calculs plus rapides donc plus de requêtes exécutées sur une période de temps donnée. Généralement, un système utilisé pour des calculs (financiers, scientifiques, géographiques) a intérêt à avoir des processeurs à fréquence élevée.

Depuis la version 9.6, un processus exécutant une requête peut demander l'aide d'autre processus (appelés workers) pour l'aider à traiter cette requête. Les différents processus utiliseront des CPU différents, permettant ainsi une exécution parallélisée d'une requête. Ceci est possible uniquement pour des requêtes en lecture seule.
Ceci a un impact important pour les requêtes consommatrices en temps CPU. De ce fait, le facteur principal de choix reste toujours le nombre de CPU disponibles.

Le cache processeur est une mémoire généralement petite mais excessivement rapide et située au plus près du processeur. Il en existe plusieurs niveaux. Tous les processeurs ont un cache de niveau L2, certains ont même un cache de niveau L3. Plus cette mémoire est importante, plus le processeur peut conserver de données utiles et éviter des aller-retours en mémoire RAM coûteux en temps. Le gain en performance pouvant être important, le mieux est de privilégier les processeurs avec beaucoup de cache.

Le choix processeur se fait donc suivant le type d'utilisation du serveur :

- une majorité de petites requêtes en très grande quantité : privilégier le nombre de cœurs ;

- une majorité de grosses requêtes en très petite quantité : privilégier la puissance processeur.

Dans tous les cas, choisissez la version des processeurs avec le plus de mémoire cache embarquée.

RAM

- Essentiel pour un serveur de bases de données
- Plus il y en a, mieux c'est
  - moins d'accès disque
- Pour le système comme pour PostgreSQL

Toute opération sur les données doit se faire en mémoire. Il est donc nécessaire qu'une bonne partie de la base tienne en mémoire, ou tout du moins la partie active. La partie passive est rarement présente en mémoire car généralement composée de données historiques qui sont peu ou pas lues et jamais modifiées.

Un cache disque permet de limiter les accès en lecture et écriture vers les disques. L'optimisation des accès aux disques est ainsi intimement liée à la quantité de mémoire physique disponible. Par conséquent, plus il y a de mémoire, mieux c'est. Cela permet de donner un cache disque plus important à PostgreSQL, tout en laissant de la place en mémoire aux sessions pour traiter les données (faire des calculs de hachage par exemple).

Il est à noter que, même avec l'apparition des disques SSD, l'accès à une donnée en mémoire est bien plus rapide qu'une donnée sur disque. Nous aborderons ce point dans le chapitre consacré aux disques.

Disques

- Trois grandes technologies : SATA, SAS et NVMe

Technologie	Temps d'accès	Débit en lecture
RAM (DDR4)	`~ 0.75 ns`	`~ 30 Go/s`
SSD NVMe	`~ 0.250 ms`	`~  6 Go/s`

Technologie	Temps d'accès	Débit en lecture
SSD SATA	`~ 0.1 ms`	`~ 500 Mo/s`
SCSI 15ktpm	`~ 1 ms`	`~ 150 Mo/s`
SATA	`~ 5 ms`	`~ 100 Mo/s`

Il existe actuellement trois types de modèles de disques :

- SATA, dont la principale qualité est d'être peu cher ;
- SAS, rapide, fiable, mais cher ;
- SSD, très rapide en temps d'accès, très cher.

Les temps d'accès sont très importants pour un SGBD. Effectivement, ces derniers conditionnent les performances des accès aléatoires, utilisés lors des parcours d'index. Le débit en lecture, lui, influe sur la rapidité de parcours des tables de façon séquentielle (bloc par bloc, de proche en proche).

Il est immédiatement visible que la mémoire est toujours imbattable, y compris face aux disques SSD avec un facteur 100 000 en performance de temps d'accès entre les deux ! À l'autre bout de l'échelle se trouvent les disques SATA. Leur faible performance en temps d'accès ne doit pas pour autant les disqualifier. Leur prix est là-aussi imbattable et il est souvent préférable de prendre un grand nombre de disques pour avoir de bonnes performances. Cependant, la fiabilité des disques SATA impose de les considérer comme des consommable et des toujours avoir des disques de secours prêt à remplacer une défaillance.

Il est souvent préconisé de se tourner vers des disques SAS (SCSI). Leurs temps d'accès et leur fiabilité ont fait de cette technologie un choix de prédilection dans le domaine des SGBD. Mais si le budget ne le permet pas, des disques SATA en plus grand nombre permet d'en gommer les défauts.

Dans tous les cas, le nombre de disques est un critère important car il permet de créer des groupes RAID efficaces ou de placer les fichiers de PostgreSQL suivant leur utilisation. Par exemple les journaux de transactions sur un système disque, les tables sur un autre et les index sur un dernier.

Le gros intérêt du disque SSD est d'avoir un temps d'accès très rapide. Il se démarque des disques magnétiques (comme SAS ou SATA) par une durée d'accès à une page aléatoire aussi rapide que celle à une donnée contiguë (ou séquentielle). C'est parfait pour accéder à des index.

Il existe aussi des supports de stockage moins courant, très onéreux, mais extrêment rapide; ce sont les cartes Fusion-IO. Il s'agit de stockage en mémoire Flash sur support PCIe pouvant aller au delà de 6 To en volume de stockage, avec des temps d'accès et des débits bien supérieurs aux SSD. Leur utilisation reste cependant très limité en raison du coût de cette technologie.

Système d'exploitation

- Quel système choisir ?
- Quelle configuration réaliser ?

Le choix du système d'exploitation n'est pas anodin. Les développeurs de PostgreSQL ont fait le choix de bien segmenter les rôles entre le système et le SGBD. Ainsi, PostgreSQL requiert que le système travaille de concert avec lui dans la gestion des accès disques, l'ordonnancement, etc.

PostgreSQL est principalement développé sur et pour Linux. Il fonctionne aussi sur d'autres systèmes, mais n'aura pas forcément les mêmes performances. De plus, la configuration du système et sa fiabilité jouent un grand rôle dans les performances et la robustesse de l'ensemble. Il est donc nécessaire de bien maîtriser ces points-là pour avancer dans l'optimisation.

Choix du système d'exploitation

- PostgreSQL fonctionne sur différents systèmes
  - Linux, BSD, Windows, Solaris, HPUX, etc.
- Principalement développé et testé sous Linux
- Windows possible pour les postes des développeurs
  - mais moins performant que Linux
  - moins d'outillage

PostgreSQL est écrit pour être le plus portable possible. Un grand nombre de choix dans son architecture a été fait en fonction de cette portabilité. Il est donc disponible sur la majorité des systèmes : Linux, BSD, Windows, Solaris, HPUX, etc. Cette portabilité est vérifiée en permanence avec la ferme de construction (BuildFarm, http://buildfarm.postgresql.org/).

Cela étant dit, il est malgré tout principalement développé sous Linux et la majorité des utilisateurs travaillent aussi avec Linux. Ce système est probablement le plus ouvert de tous, permettant ainsi une meilleure compréhension de ses mécaniques internes et ainsi une meilleure interaction. Ainsi, Linux est certainement le système le plus fonctionnel et performant avec PostgreSQL. La distribution Linux a généralement peu d'importance en ce qui concerne les performances. Les deux distributions les plus fréquemment utilisées sont RedHat (et ses dérivés CentOS, Scientific Linux) et Debian.

Un autre système souvent utilisé est Windows. Ce dernier est une solution pour du développement. Il est cependant moins performant avec PostgreSQL que Linux. Cela est principalement dû à sa gestion assez mauvaise de la mémoire partagée. Cela a pour conséquence qu'il est difficile d'avoir un cache disque important pour PostgreSQL sous Windows.

De plus, vous ne pouvez pas démarrer PostgreSQL en tant que service si vous avez besoin de plus de 125 connexions pour des problématiques d'espace mémoire attribuée à un processus non-interactif. Le seul moyen de contourner ce problème sera de le lancer en mode interactif, depuis la ligne de commande. La limite théorique est alors repoussée à 750 connexions (plus d'information sur le wiki PostgreSQL[1]).

On préférera donc utiliser une distribution Linux pour utiliser PostgreSQL en production.

Choix du noyau

- Choisir la version la plus récente du noyau car
  - plus stable
  - plus compatible avec le matériel
  - plus de fonctionnalités
  - plus de performances
- Utiliser la version de la distribution Linux
  - ne pas le compiler soi-même

Il est préférable de ne pas fonctionner avec une très ancienne version du noyau Linux. Les dernières versions sont les plus stables, les plus performantes, les plus compatibles avec les derniers matériels. Ce sont aussi celles qui proposent le plus de fonctionnalités intéressantes, comme la gestion complète du système de fichiers ext4, les « control groups », une supervision avancée (avec `perf` et `bpf`), etc.

Le mieux est d'utiliser la version proposée par votre distribution Linux et de mettre à jour le noyau quand cela s'avère possible.

Le compiler vous-même peut dans certains cas vous apporter un plus en terme de performances. Mais ce plus est difficilement quantifiable et est assorti d'un gros inconvénient : avoir à gérer soi-même les mises à jours, la recompilation en cas d'oubli d'un pilote, etc.

---

[1] http://wiki.postgresql.org/wiki/%20Running_%26_Installing_PostgreSQL_On_Native_Windows#I_cannot_run_with_more_than%20_about_125_connections_at_once.2C_despite_having_capable_hardware

Configuration du noyau

- En plus du choix du noyau, certains paramètres nécessitent une configuration personnalisée
  - sur-allocation de la mémoire
  - taille et comportement du swap
  - affinité entre les cœurs et les espaces mémoire
  - huge pages

Le noyau, comme tout logiciel, est configurable. Certaines configurations sont particulièrement importantes pour PostgreSQL.

Configuration du OOM :

- Supervision de la sur-allocation par le noyau
- Si cas critique, l'OOM fait un kill -9 du processus
- À désactiver pour un serveur dédié
  - vm.overcommit_memory
  - vm.overcommit_ratio

Certaines applications réservent souvent plus de mémoire que nécessaire. Plusieurs optimisations noyau permettent aussi d'économiser de l'espace mémoire. Ainsi, par défaut, le noyau Linux s'autorise à allouer aux processus plus de mémoire qu'il n'en dispose réellement, le risque de réellement utiliser cette mémoire étant faible. On appelle ce comportement l'*Overcommit Memory*. Si celui-ci peut être intéressant dans certains cas d'utilisation, il peut devenir dangereux dans le cadre d'un serveur PostgreSQL dédié.

Effectivement, si le noyau arrive réellement à court de mémoire, il décide alors de tuer certains processus en fonction de leur impact sur le système. Il est alors fort probable que ce soit un processus PostgreSQL qui soit tué. Dans ce cas, les transactions en cours seront annulées, et une perte de données est parfois possible en fonction de la configuration de PostgreSQL. Une corruption est par contre plutôt exclue.

Il est possible de modifier ce comportement grâce aux paramètres `vm.overcommit_memory` et `vm.overcommit_ratio` du fichier `/etc/sysctl.conf`. En plaçant `vm.overcommit_memory` à 2, le noyau désactivera complètement l'*overcommit memory*. La taille maximum de mémoire utilisable par les applications se calcule alors grâce à la formule suivante :

```
(RAM * vm.overcommit_ratio / 100) + SWAP
```

Attention, la valeur par défaut du paramètre `vm.overcommit_ratio` est 50. Ainsi, sur un système avec 32 Go de mémoire et 2 Go de swap, nous obtenons seulement 18 Go de mémoire allouable !

Ne pas oublier de modifier ce paramètre ; avec `vm.overcommit_ratio` positionné à 75, nous obtenons 26 Go de mémoire utilisable par les applications sur les 32 Go disponibles. Avoir un tel paramétrage permet de garantir qu'il y aura toujours au moins 20% du total de la RAM disponible pour le cache disque, qui est très bénéfique à PostgreSQL.

Configuration du swap :

- Taille de la swap
  - pas plus de 2 Go
- Contrôler son utilisation
  - vm.swappiness fixé à 10

Configuration de l'affinité processeur / mémoire :

- Pour architecture NUMA (multi-sockets)
- Chaque socket travaille plus efficacement avec une zone mémoire allouée
- Peut pénaliser le cache disque système
  - vm.zone_reclaim_mode

Configuration du scheduler processeur :

- Réduire la propension du kernel à migrer les processus
  - `kernel.sched_migration_cost_ns = 5000000`
- Désactiver le regroupement par session TTY
  - `kernel.sched_autogroup_enabled = 0`

Choix du système de fichiers

- Windows :
  - NTFS
- Linux :
  - ext4, xfs
- Utiliser celui préconisé par votre système d'exploitation/distribution

- … et **éviter NFS** !

Quelque soit le système d'exploitation, les systèmes de fichiers ne manquent pas. Linux en est la preuve avec pas moins d'une dizaine de systèmes de fichiers. Le choix peut paraître compliqué mais il se révèle fort simple : il est préférable d'utiliser le système de fichiers préconisé par votre distribution Linux. Ce système est à la base de tous les tests des développeurs de la distribution : il a donc plus de chances d'avoir moins de bugs, tout en proposant plus de performances.
En règle générale, cela voudra dire le système *ext4* ou *XFS*. Côté performance, *ext4* semble un plus performant (voir notamment un comparatif ext4/xfs[2]).

Le système *btrfs* est encore expérimental. Il s'agit d'un système prometteur sur le papier. Il pourrait en effet apporter des fonctionnalités impossible à mettre en place dans *ext4*. Il y a encore trop de doutes sur sa stabilité et ses performances en écriture sont moins bonnes qu'avec *ext4* ou *XFS*. Son utilisation est déconseillée.

Les systèmes *reiserfs* et *jfs* ne sont pratiquement plus développés et doivent dans tous les cas être évités.

Pour Windows, la question ne se pose pas. Le système VFAT n'est pas suffisamment stable pour qu'il puisse être utilisé avec PostgreSQL. De plus, il ne connaît pas le concept des liens symboliques, important lors de la création de tablespaces avec PostgreSQL. La seule solution disponible sous Windows est donc NTFS. L'installeur fourni par EnterpriseDB dispose d'une protection qui empêche l'installation d'une instance PostgreSQL sur une partition VFAT.

Quant à Solaris, ZFS est un système très intéressant grâce à son panel fonctionnel et son mécanisme de Copy On Write permettant de faire une copie des fichiers sans arrêter PostgreSQL (aka. *Snapshot*). Les performances sont cependant moins bonnes qu'avec *ext4* ou *XFS* (se rapporter au comparatif ci-dessus[3]).

NFS peut sembler intéressant, vu ses fonctionnalités. Cependant, ce système de fichiers est source de nombreux problèmes avec PostgreSQL. La documentation[4] l'indique très clairement :

> Many installations create database clusters on network file systems. Sometimes this is done directly via NFS, or by using a Network Attached Storage (NAS) device that uses NFS internally. PostgreSQL does nothing special for NFS file systems, meaning it assumes NFS behaves exactly like locally-connected drives (DAS, Direct Attached Storage). If client and server NFS implementations have non-standard semantics, this can cause reliability problems (see http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html). Specifically, delayed (asynchronous) writes to the NFS server can cause reliability problems; if possible,

[2] http://blog.pgaddict.com/posts/postgresql-performance-on-ext4-and-xfs
[3] https://blog.pgaddict.com/posts/postgresql-performance-on-ext4-and-xfs
[4] http://www.postgresql.org/docs/current/static/creating-cluster.html

> mount NFS file systems synchronously (without caching) to avoid this. Also, soft-mounting NFS is not recommended. (Storage Area Networks (SAN) use a low-level communication protocol rather than NFS.)

Giuseppe Broccolo a étudié en profondeur[5] l'utilisation de NFS avec PostgreSQL. Il préconise un paramétrage précis à réaliser pour prévenir tout risque de corruption. Les conséquences sur les performances sont souvent rédhibitoires pour une utilisation intensive.

Configuration du système de fichiers

- Quelques options à connaître :
  - noatime
  - nobarrier sous conditions
  - data=writeback à éviter
- Permet de gagner un peu en performance

Quel que soit le système de fichiers choisi, il est possible de le configurer lors du montage, via le fichier `/etc/fstab`.

Certaines options sont intéressantes en termes de performances. Ainsi, `noatime` évite l'écriture de l'horodatage du dernier accès aux fichiers et aux répertoires.

L'option `nobarrier` peut être utilisée mais avec précaution. Cette dernière peut apporter une différence significative en terme de performance, mais elle mets en péril vos données en cas de coupure soudaine où les caches disques, RAID ou baies sont alors perdus. Cette option ne peut être utilisée qu'à la seule condition que tous ces différents caches soient sécurisés par une batterie.

Les options `data=writeback` et `nobarrier` sont souvent cités comme optimisation potentielle. Le mode `writeback` de journalisation des ext3 et ext4 est à **éviter**. Effectivement, dans certains cas rares, en cas d'interruption brutale, certains fichiers peuvent conserver des blocs fantômes ayant été normalement supprimés juste avant le crash.

---

[5] https://www.slideshare.net/GiuseppeBroccolo/gbroccolo-pgconfeu2016-pgnfs

Serveur de bases de données

- Version
- Emplacement des fichiers
- Configuration

Après avoir vu le matériel et le système d'exploitation, il est temps de passer au serveur de bases de données. Lors d'une optimisation, il est important de vérifier trois points essentiels :

- la version de PostgreSQL ;
- sa configuration (uniquement le fichier `postgresql.conf`) ;
- et l'emplacement des fichiers (journaux de transactions, tables, index, stats).

Version

- Chaque nouvelle version majeure a des améliorations de performance
  - mettre à jour est un bon moyen pour gagner en performances
- Ne pas compiler
  - sauf en cas de correction de bug
  - ou pour le développement de nouvelles fonctionnalités

Il est généralement conseillé de passer à une version majeure plus récente qu'à partir du moment où les fonctionnalités proposées sont suffisamment intéressantes. C'est un bon conseil en soi mais il faut aussi se rappeler qu'un gros travail est fait pour améliorer le planificateur. Ces améliorations peuvent être une raison suffisante pour changer de version majeure.

Voici quelques exemples frappants :

- La version 9.0 dispose d'une optimisation du planificateur lui permettant de supprimer une jointure `LEFT JOIN` si elle est inutile pour l'obtention du résultat. C'est une optimisation particulièrement bienvenue pour tous les utilisateurs d'ORM.
- La version 9.1 dispose du SSI, pour `Serializable Snapshot Isolation`. Il s'agit d'une implémentation très performante du mode d'isolation sérialisée. Ce mode permet d'éviter l'utilisation des `SELECT FOR UPDATE`.
- La version 9.2 dispose d'un grand nombre d'améliorations du planificateur et des processus postgres qui en font une version exceptionnelle pour les performances, notamment les parcours d'index seuls.

- La version 9.6 propose la parallélisation de l'exécution de certaines requêtes.
- La version 12 permet de partitionner les tables sans impact sur le schéma de données

Il est déconseillé de compiler PostgreSQL. Il existe des versions packagées disponibles dans chaque distribution.

On passera par une compilation dans le cas d'un bug bloquant, corrigé, mais pas encore packagé dans une version mineure. Il sera également nécessaire de compiler PostgreSQL lors de développements ou de tests de nouvelles fonctionnalités.

Emplacement des fichiers de données

- Séparer les objets suivant leur utilisation
- Tablespaces
- Quelques stratégies
  - séparer tables et index
  - séparer archives et données vivantes
- Configuration possible des tablespaces
  - `seq_page_cost`, `random_page_cost`, `effective_io_concurrency`

Il est possible de séparer les objets SQL dans des disques différents. Par défaut, PostgreSQL se charge du placement des objets sur le disque. Tout a lieu dans le répertoire des données, mais il est possible de créer des répertoires de stockage supplémentaires. Le nom de ces répertoires, au niveau SQL, est tablespace. Pour placer un objet dans un tablespace, il faut créer ce tablespace si ce n'est pas déjà fait, puis lancer l'ordre SQL de déplacement d'objet. Voici un exemple complet :

```
$ mkdir /opt/tablespace1
$ chown postgres:postgres /opt/tablespace1
$ chmod 700 /opt/tablespace1
$ psql postgres
postgres =# CREATE TABLESPACE grosdisque LOCATION '/opt/tablespace1';
postgres =# ALTER TABLE t1 TABLESPACE grosdisque;
```

L'idée est de séparer les objets suivant leur utilisation. Une configuration assez souvent utilisée est de placer les tables dans un tablespace et les index dans un autre. Cela permet des écritures quasi-simultanées sur différents fichiers.

La seule configuration possible au niveau des tablespaces se situe au niveaux des paramètres `seq_page_cost`, `random_page_cost` et `effective_io_concurrency`. Ils sont utilisés par

le planificateur pour évaluer la vitesse de récupérer une page séquentielle et une page aléatoire. C'est habituellement intéressant avec les SSD qui ont normalement une vitesse sensiblement équivalente pour les accès séquentiels et aléatoires, contrairement aux disques magnétiques.

```
ALTER TABLESPACE disque_ssd SET ( random_page_cost = 1 );
```

Emplacement des journaux de transactions

- Placer les journaux sur un autre disque
- Option -X de l'outil initdb
- Lien symbolique

Chaque donnée modifiée est écrite une première fois dans les journaux de transactions et une deuxième fois dans les fichiers de données. Cependant, les écritures dans ces deux types de fichiers sont très différentes. Les opérations dans les journaux de transactions sont uniquement des écritures séquentielles, sur de petits fichiers (d'une taille de 16 Mo), alors que celles des fichiers de données sont des lectures et des écritures fortement aléatoires, sur des fichiers bien plus gros (au maximum 1 Go). Du fait d'une utilisation très différente, avoir un système disque pour l'un et un système disque pour l'autre permet de gagner énormément en performances. Il faut donc pouvoir les séparer.

Pour une instance existante, il est nécessaire d'arrêter PostgreSQL, de déplacer le répertoire des journaux de transactions, de créer un lien vers ce répertoire, et enfin de redémarrer PostgreSQL. Voici un exemple qui montre le déplacement dans /pgxlog.

```
# systemctl stop postgresql@14-main.service
# mkdir -p /pg_wal/14/main/
# cd $PGDATA
# mv pg_wal /pg_wal/14/main/
# ln -s /pg_wal/14/main/pg_wal pg_wal
# systemctl start postgresql@14-main.service
```

Il est aussi possible de faire en sorte que la commande initdb le fasse elle-même. Pour cela, il faut utiliser l'option −X :

```
$ initdb -X /pgxlog
```

Cependant le résultat est le même. Un lien symbolique existe dans le répertoire de données pour que PostgreSQL retrouve le répertoire des journaux de transactions.

Configuration de PostgreSQL

- Installation par défaut pas prête pour la production
- Configuration nécessaire
- Présentation des paramètres importants

Configuration - mémoire

- shared_buffers
- wal_buffers
- work_mem
- maintenance_work_mem

Ces quatre paramètres concernent tous la quantité de mémoire que PostgreSQL utilisera pour ses différentes opérations.

`shared_buffers` permet de configurer la taille du cache disque de PostgreSQL. Chaque fois qu'un utilisateur veut extraire des données d'une table (par une requête `SELECT`) ou modifier les données d'une table (par exemple avec une requête `UPDATE`), PostgreSQL doit d'abord lire les lignes impliquées et les mettre dans son cache disque. Cette lecture prend du temps. Si ces lignes sont déjà dans le cache, l'opération de lecture n'est plus utile, ce qui permet de renvoyer plus rapidement les données à l'utilisateur. Ce cache est commun à tous les processus PostgreSQL, il n'existe donc qu'en un exemplaire. Généralement, il faut lui donner une grande taille, tout en conservant malgré tout la majorité de la mémoire pour le cache disque du système, à priori plus efficace pour de grosses quantités de données. Le pourcentage généralement préconisé est de 25% de la mémoire totale pour un serveur dédié. Donc, par exemple, pour un serveur contenant 8 Go de mémoire, nous configurerons le paramètre `shared_buffers` à 2 Go. Néanmoins, on veillera à ne pas dépasser 8 Go. Des études ont montré que les performances décroissaient avec plus de mémoire.

PostgreSQL dispose d'un autre cache disque. Ce dernier concerne les journaux de transactions. Il est généralement bien plus petit que `shared_buffers` mais, si le serveur est multi-processeurs et qu'il y a de nombreuses connexions simultanées au serveur PostgreSQL, il est important de l'augmenter. Le paramètre en question s'appelle `wal_buffers`. Plus cette mémoire est importante, plus les transactions seront conservées en mémoire avant le `COMMIT`. À partir du moment où le `COMMIT` d'une transaction arrive, toutes les modifications effectuées dans ce cache par cette transaction sont enregistrées dans le fichier du journal de transactions. La valeur par défaut est de 64 Ko mais une valeur de 16 Mo sera plus intéressante. Il est à noter qu'à partir de la version 9.1, cette taille est gérée automatiquement par PostgreSQL si `wal_buffers` vaut -1.

Deux autres paramètres mémoire sont essentiels pour de bonnes performances mais eux sont valables par processus. `work_mem` est utilisé comme mémoire de travail pour les tris et les hachages. S'il est nécessaire d'utiliser plus de mémoire, le contenu de cette mémoire est stocké sur disque pour permettre la réutilisation de la mémoire. Par exemple, si une jointure demande à stocker 52 Mo en mémoire alors que le paramètre `work_mem` vaut 10 Mo, à chaque utilisation de 10 Mo, cette partie de mémoire sera copiée sur disque, ce qui fait en gros 50 Mo écrit sur disque pour cette jointure. Si, par contre, le paramètre `work_mem` vaut 60 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément l'opération de jointure. Cette mémoire est utilisée par chaque processus du serveur PostgreSQL, de manière indépendante. Suivant la complexité des requêtes, il est même possible qu'un processus utilise plusieurs fois cette mémoire (par exemple si une requête fait une jointure et un tri). Il faut faire très attention à la valeur à donner à ce paramètre et le mettre en relation avec le nombre maximum de connexions (paramètre `max_connections`). Si la valeur est trop petite, cela forcera des écritures sur le disque par PostgreSQL. Si elle est trop grande, cela pourrait faire swapper le serveur. Généralement, une valeur entre 10 et 50 Mo est concevable. Au delà de 100 Mo, il y a probablement un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisé pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Quant à `maintenance_work_mem`, il est aussi utilisé par chaque processus PostgreSQL réalisant une opération particulière : un VACUUM, une création d'index ou l'ajout d'une clé étrangère. Comme il est peu fréquent que ces opérations soient effectuées en simultané, la valeur de ce paramètre est très souvent bien supérieure à celle du paramètre `work_mem`. Sa valeur se situe fréquemment entre 128 Mo et 1 Go, voire plus.

Configuration - planificateur

- effective_cache_size
- random_page_cost

Le planificateur dispose de plusieurs paramètres de configuration. Les deux principaux sont `effective_cache_size` et `random_page_cost`.

Le premier permet d'indiquer la taille du cache disque du système d'exploitation. Ce n'est donc pas une mémoire que PostgreSQL va allouer, c'est plutôt une simple indication de ce qui est disponible en dehors de la mémoire taillée par le paramètre `shared_buffers`. Le planificateur se base sur ce paramètre pour évaluer les chances de trouver des pages de données en mémoire. Une valeur plus importante aura tendance à faire en sorte que le planificateur privilégie l'utilisation des index, alors

qu'une valeur plus petite aura l'effet inverse. Généralement, il se positionne à 2/3 de la mémoire d'un serveur pour un serveur dédié.

Le paramètre `random_page_cost` permet de faire appréhender au planificateur le fait qu'une lecture aléatoire (autrement dit avec déplacement de la tête de lecture) est autrement plus coûteuse qu'une lecture séquentielle. Par défaut, la lecture aléatoire a un coût quatre fois plus important que la lecture séquentielle.
Ce n'est qu'une estimation, cela n'a pas à voir directement avec la vitesse des disques. Cela le prend en compte, mais cela prend également en compte l'effet du cache. Cette estimation peut être revue. Si elle est revue à la baisse, les parcours aléatoires seront moins coûteux et, par conséquent, les parcours d'index seront plus facilement sélectionnés. Si elle est revue à la hausse, les parcours aléatoires coûteront encore plus cher, ce qui risque d'annuler toute possibilité d'utiliser un index. La valeur 4 est une estimation basique. En cas d'utilisation de disque rapide, il ne faut pas hésiter à descendre un peu cette valeur (entre 2 et 3 par exemple). Si les données tiennent entièrement en cache où sont stockées sur des disques SSD, il est même possible de descendre encore plus cette valeur.

Configuration - WAL

- `fsync = on`

`fsync` est le paramètre qui assure que les données sont non seulement écrites mais aussi forcées sur disque. En fait, quand PostgreSQL écrit dans des fichiers, cela passe par des appels systèmes pour le noyau qui, pour des raisons de performances, conserve dans un premier temps les données dans un cache. En cas de coupure courant, si ce cache n'est pas vidé sur disque, il est possible que des données enregistrées par un COMMIT implicite ou explicite n'aient pas atteint le disque et soient donc perdues une fois le serveur redémarré, ou pire, que des données aient été modifiées dans des fichiers de données, sans avoir été auparavant écrites dans le journal, ce qui entraînera dans ce cas des incohérences dans les fichiers de données au redémarrage. Il est donc essentiel que les données enregistrées dans les journaux de transactions soient non seulement écrites mais que le noyau soit forcé de les écrire réellement sur disque. Cette opération s'appelle `fsync`. Par défaut, ce comportement est activé. Évidemment, cela coûte en performance mais ce que ça apporte en terme de fiabilité est essentiel. Il est donc obligatoire en production de conserver ce paramètre activé.

Chaque bloc modifié dans le cache disque de PostgreSQL doit être écrit sur disque au bout d'un certain temps. Ce temps dépend du paramètre `checkpoint_timeout`. Il permet de s'assurer d'avoir au minimum un `CHECKPOINT` toutes les X minutes (5 par défaut).
Tout surplus d'activité doit aussi être géré. Un surplus d'activité engendrera des journaux de transactions supplémentaires. Le meilleur moyen dans ce cas est de préciser au bout de combien de journaux

traités il faut lancer un `CHECKPOINT`. Cela se fait via le paramètre `max_wal_size`. Ce paramètre fixe la quantité de WAL déclenchant un checkpoint (1 Go par défaut).

Le nom du paramètre `max_wal_size` peut porter à confusion. Le volume de WAL peut dépasser `max_wal_size` en cas de forte activité, ce n'est pas une valeur plafond.

Configuration - autovacuum

- `autovacuum = on`

L'autovacuum doit être activé. Ce processus supplémentaire coûte un peu en performances mais il s'acquitte de deux tâches importantes pour les performances : éviter la fragmentation dans les tables et index, et mettre à jour les statistiques sur les données.

Sa configuration est généralement trop basse pour être suffisamment efficace.

Configuration - parallélisation

- Nombreux paramètres de configuration
- Dépendra :
  - du type de requêtes à exécuter
  - du nombre de CPU disponibles

Les 6 paramètres suivants permettent de configurer finement la parallélisation des requêtes :

- max_worker_processes
- max_parallel_workers
- max_parallel_workers_per_gather
- max_parallel_maintenance_workers
- min_parallel_table_scan_size
- min_parallel_index_scan_size

Un processus PostgreSQL peut se faire aider d'autres processus pour exécuter une seule et même requête. Le nombre de processus utilisables pour une requête dépend de la valeur du paramètre `max_parallel_workers_per_gather` (à 2 par défaut). Si plusieurs processus veulent paralléliser l'exécution de leur requête, le nombre de processus d'aide ne pourra pas dépasser la valeur du paramètre `max_parallel_workers` (8 par défaut).

Il est à noter que ce nombre ne peut pas dépasser la valeur du paramètre `max_worker_processes` (par défaut à 8).

La parallélisation peut se faire sur différentes parties d'une requête, comme un parcours de table ou d'index, une jointure ou un calcul d'agrégat. Dans le cas d'un parcours, la parallélisation n'est possible que si la table ou l'index est suffisamment volumineux pour qu'une telle action soit intéressante au niveau des performances. Le volume déclencheur dépend de la valeur du paramètre `min_parallel_table_scan_size`, dont la valeur par défaut est de 8 Mo, pour une table et de la valeur du paramètre `min_parallel_index_scan_size` pour un index (valeur par défaut, 512 Ko).

Outil pgtune

- Outil écrit en `Python`, par Greg Smith
  - Repris en `Ruby` par Alexey Vasiliev
- Propose quelques meilleures valeurs pour certains paramètres
- Quelques options pour indiquer des informations systèmes
- Version web[6]
- Il existe également pgconfig[7]

Le site du projet en ruby[8] se trouve sur ghithub.

pgtune est capable de trouver la quantité de mémoire disponible sur le système. À partir de cette information et de quelques règles internes, il arrive à déduire une configuration bien meilleure que la configuration par défaut. Il est important de lui indiquer le type d'utilisation principale : Web, DW (pour DataWarehouse), mixed, etc.

Sur le serveur de tests se trouvent 8 Go de RAM. Commençons par une configuration pour une utilisation par une application web :

```
max_connections = 200
shared_buffers = 2GB
effective_cache_size = 6GB
work_mem = 10485kB
```

[6] http://pgtune.leopard.in.ua/
[7] https://www.pgconfig.org/
[8] https://github.com/le0pard/pgtune

```
maintenance_work_mem = 512MB
min_wal_size = 1GB
max_wal_size = 2GB
checkpoint_completion_target = 0.7
wal_buffers = 16MB
default_statistics_target = 100
```

Une application web, c'est beaucoup d'utilisateurs qui exécutent de petites requêtes simples, très rapides, non consommatrices. Du coup, le nombre de connexions a été doublé par rapport à sa valeur par défaut. Le paramètre `work_mem` est augmenté mais raisonnablement par rapport à la mémoire totale et au nombre de connexions. Le paramètre `shared_buffers` se trouve au quart de la mémoire, alors que le paramètre `effective_cache_size` est au deux-tiers évoqué précédemment. Le paramètre `wal_buffers` est aussi augmenté. Il arrive à 16 Mo. Il peut y avoir beaucoup de transactions en même temps, mais elles seront généralement peu coûteuses en écriture. D'où le fait que les paramètres `min_wal_size`, `max_wal_size` et `checkpoint_completion_target` sont augmentés mais là-aussi très raisonnablement.

Voyons maintenant avec un profil OLTP (*OnLine Transaction Processing*) :

```
max_connections = 300
shared_buffers = 2GB
effective_cache_size = 6GB
work_mem = 6990kB
maintenance_work_mem = 512MB
min_wal_size = 2GB
max_wal_size = 4GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
```

Une application OLTP doit gérer un plus grand nombre d'utilisateurs. Ils font autant d'opérations de lecture que d'écriture. Tout cela est transcrit dans la configuration. Un grand nombre d'utilisateurs simultanés veut dire une valeur importante pour le paramètre `max_connections` (maintenant à 300). De ce fait, le paramètre `work_mem` ne peut plus avoir une valeur si importante. Sa valeur est donc baissée tout en restant fortement au-dessus de la valeur par défaut. Dû au fait qu'il y aura plus d'écritures, la taille du cache des journaux de transactions (paramètre `wal_buffers`) est augmentée. Il faudra essayer de tout faire passer par les `CHECKPOINT`, d'où la valeur maximale pour `checkpoint_completion_target`. Quant à `shared_buffers` et `effective_cache_size`, ils restent aux valeurs définies ci-dessus (respectivement un quart et deux-tiers de la mémoire).

Et enfin avec un profil entrepôt de données (*datawarehouse*) :

```
max_connections = 20
shared_buffers = 2GB
effective_cache_size = 6GB
work_mem = 52428kB
maintenance_work_mem = 1GB
min_wal_size = 4GB
max_wal_size = 8GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 500
```

Pour un entrepôt de données, il y a généralement peu d'utilisateurs à un instant `t`.
Par contre, ils exécutent des requêtes complexes sur une grosse volumétrie. Du coup, la configuration change en profondeur cette fois. Le paramètre `max_connections` est diminué très fortement. Cela permet d'allouer beaucoup de mémoire aux tris et hachages (paramètre `work_mem` à 50 Mo). Les entrepôts de données ont souvent des scripts d'import de données (batchs). Cela nécessite de pouvoir écrire rapidement de grosses quantités de données, autrement dit une augmentation conséquente du paramètre `wal_buffers` et des `min_wal_size`/`max_wal_size`. Du fait de la grosse volumétrie des bases dans ce contexte, une valeur importante pour le `maintenance_work_mem` est essentiel pour que les créations d'index et les `VACUUM` se fassent rapidement. De même, la valeur du `default_statistics_target` est sérieusement augmentée car le nombre de lignes des tables est conséquent et nécessite un échantillon plus important pour avoir des statistiques précises sur les données des tables.

Évidemment, tout ceci n'est qu'une recommandation générale. L'expérimentation permettra de se diriger vers une configuration plus personnalisée.

Comprendre `EXPLAIN`

- Le matériel, le système et la configuration sont importants pour les performances
- Mais il est aussi essentiel de se préoccuper des requêtes et de leurs performances

Face à un problème de performances, l'administrateur se retrouve assez rapidement face à une (ou plusieurs) requête(s). Une requête en soi représente très peu d'informations. Suivant la requête, des dizaines de plans peuvent être sélectionnés pour l'exécuter. Il est donc nécessaire de pouvoir trouver

le plan d'exécution et de comprendre ce plan. Cela permet de mieux appréhender la requête et de mieux comprendre les pistes envisageables pour la corriger.

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.

Jeu de tests

- Tables
  - `services` : 4 lignes
  - `services_big` : 40 000 lignes
  - `employes` : 14 lignes
  - `employes_big` : ~500 000 lignes
- Index
  - `service*.num_service` (clés primaires)
  - `employes*.matricule` (clés primaires)
  - `employes*.date_embauche`
  - `employes_big.num_service` (clé étrangère)

Le script suivant permet de recréer le jeu d'essai :

```sql
-- suppression des tables si elles existent

DROP TABLE IF EXISTS services CASCADE;
DROP TABLE IF EXISTS services_big CASCADE;
DROP TABLE IF EXISTS employes CASCADE;
DROP TABLE IF EXISTS employes_big CASCADE;

-- définition des tables

CREATE TABLE services (
    num_service serial PRIMARY KEY,
    nom_service character varying(20),
    localisation character varying(20),
    departement integer,
    date_creation date
);

CREATE TABLE services_big (
    num_service serial PRIMARY KEY,
    nom_service character varying(20),
    localisation character varying(20),
    departement integer,
    date_creation date
);

CREATE TABLE employes (
    matricule     serial 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)
);

CREATE TABLE employes_big (
    matricule     serial 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)
```

```
);

-- ajout des données

INSERT INTO services
VALUES
    (1, 'Comptabilité', 'Paris', 75, '2006-09-03'),
    (2, 'R&D', 'Rennes', 40, '2009-08-03'),
    (3, 'Commerciaux', 'Limoges', 52, '2006-09-03'),
    (4, 'Consultants', 'Nantes', 44, '2009-08-03');

INSERT INTO services_big (nom_service, localisation, departement,
 ↪  date_creation)
VALUES
    ('Comptabilité', 'Paris', 75, '2006-09-03'),
    ('R&D', 'Rennes', 40, '2009-08-03'),
    ('Commerciaux', 'Limoges', 52, '2006-09-03'),
    ('Consultants', 'Nantes', 44, '2009-08-03');

INSERT INTO services_big (nom_service, localisation, departement,
 ↪  date_creation)
  SELECT s.nom_service, s.localisation, s.departement, s.date_creation
  FROM services s, generate_series(1, 10000);

INSERT INTO employes VALUES
    (33, 'Roy', 'Arthur', 'Consultant', 105, '2000-06-01', 4),
    (81, 'Prunelle', 'Léon', 'Commercial', 97, '2000-06-01', 3),
    (97, 'Lebowski', 'Dude', 'Responsable', 104, '2003-01-01', 3),
    (104, 'Cruchot', 'Ludovic', 'Directeur Général', NULL, '2005-03-06', 3),
    (105, 'Vacuum', 'Anne-Lise', 'Responsable', 104, '2005-03-06', 4),
    (119, 'Thierrie', 'Armand', 'Consultant', 105, '2006-01-01', 4),
    (120, 'Tricard', 'Gaston', 'Développeur', 125, '2006-01-01', 2),
    (125, 'Berlicot', 'Jules', 'Responsable', 104, '2006-03-01', 2),
    (126, 'Fougasse', 'Lucien', 'Comptable', 128, '2006-03-01', 1),
    (128, 'Cruchot', 'Joséphа', 'Responsable', 105, '2006-03-01', 1),
    (131, 'Lareine-Leroy', 'Émilie', 'Développeur', 125, '2006-06-01', 2),
    (135, 'Brisebard', 'Sylvie', 'Commercial', 97, '2006-09-01', 3),
    (136, 'Barnier', 'Germaine', 'Consultant', 105, '2006-09-01', 4),
    (137, 'Pivert', 'Victor', 'Consultant', 105, '2006-09-01', 4);

-- on copie la table employes
INSERT INTO employes_big SELECT * FROM employes;
```

```sql
-- duplication volontaire des lignes d'un des employés
INSERT INTO employes_big
  SELECT i, nom,prenom,fonction,manager,date_embauche,num_service
  FROM employes_big,
  LATERAL generate_series(1000, 500000) i
  WHERE matricule=137;

-- création des index
CREATE INDEX ON employes(date_embauche);
CREATE INDEX ON employes_big(date_embauche);
CREATE INDEX ON employes_big(num_service);

-- calcul des statistiques sur les nouvelles données
VACUUM ANALYZE;
```

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.

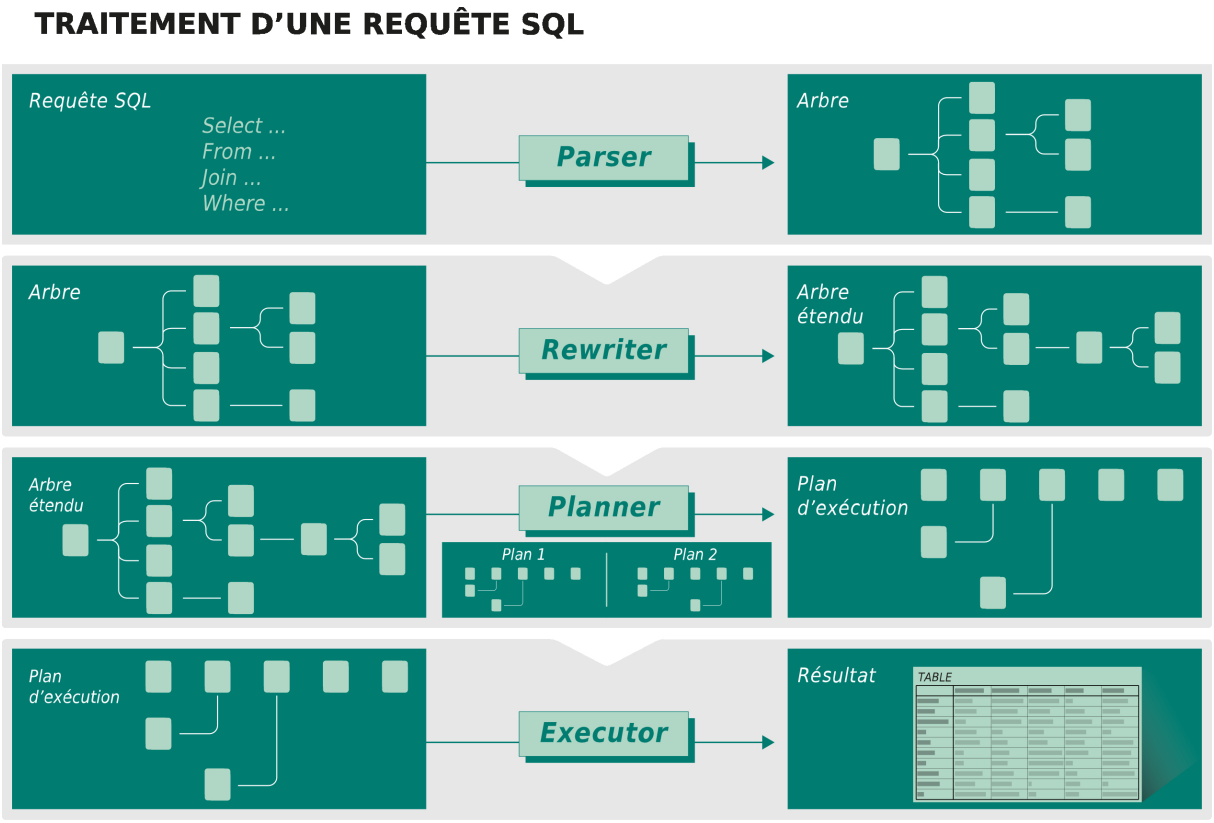
Niveau système

- 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. L'affichage peut aussi être un problème (afficher une ligne sera plus rapide qu'afficher un million de lignes, afficher un entier est plus rapide qu'afficher un document texte de 1 Mo, etc.).

Niveau SGBD



Lorsque le serveur récupère la requête, un ensemble de traitements est réalisé :

- le `parser` va réaliser une analyse syntaxique de la requête ;
- le `rewriter` va réécrire, si nécessaire la requête ;
- pour cela, il prend en compte les règles, les vues non matérialisées et les fonctions SQL ;

- si une règle demande de changer la requête, la requête envoyée est remplacée par la nouvelle ;
- si une vue non matérialisée est utilisée, la requête qu'elle contient est intégrée dans la requête envoyée ;
- si une fonction SQL intégrable 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`.

Exceptions

- Procédures stockées (appelées avec `CALL`)
- 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.

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 10 % des lignes, soit 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 retournées par la lecture d'une table après application d'un ou plusieurs prédicats.

Requête étudiée

```sql
SELECT matricule, nom, prenom, nom_service, fonction, localisation
FROM   employes emp
JOIN   services ser ON (emp.num_service = ser.num_service)
WHERE  ser.localisation = 'Nantes';
```

Cette requête nous servira d'exemple. Elle permet de déterminer les employés basés à Nantes et pour résultat :

```
 matricule |   nom    |  prenom   | nom_service |  fonction   | localisation
-----------+----------+-----------+-------------+-------------+------------
 ↪  --
        33 | Roy      | Arthur    | Consultants | Consultant  | Nantes
       105 | Vacuum   | Anne-Lise | Consultants | Responsable | Nantes
       119 | Thierrie | Armand    | Consultants | Consultant  | Nantes
       136 | Barnier  | Germaine  | Consultants | Consultant  | Nantes
       137 | Pivert   | Victor    | Consultants | Consultant  | Nantes
```

En fonction du cache, elle dure de 1 à quelques millisecondes.

Plan de la requête étudiée

L'objet de ce module est de comprendre son plan d'exécution :

```
 Hash Join  (cost=1.06..2.28 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.

Ce plan montre une jointure par hachage. La table `services` est parcourue intégralement (`Seq Scan`), mais elle est filtrée sur le critère sur « Nantes ». Un *hash* de la colonne `num_service` des lignes résultantes de ce filtrage est effectué, et comparé aux valeurs rencontrées lors d'un parcours complet de `employes`.

S'affichent également les coûts estimés des opérations et le nombre de lignes que PostgreSQL s'attend à trouver à chaque étape.

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).

Lors de cette énumération des différents plans, il calcule leur coût. Cela lui permet d'en ignorer certains alors qu'ils sont incomplets si leur plan d'exécution est déjà plus coûteux que les autres. Pour calculer le coût, 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.

À la fin de l'énumération et du calcul de coût, il ne lui reste plus qu'à sélectionner le plan qui a le plus petit coût.

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 employés dont le nom commence par la lettre B en triant les employés par leur service, il pourrait utiliser une requête du type :

```
# SELECT nom, prenom, num_service
  FROM employes
  WHERE nom LIKE 'B%'
  ORDER BY num_service ;

    nom    |  prenom  | num_service
-----------+----------+-------------
 Berlicot  | Jules    |           2
 Brisebard | Sylvie   |           3
```

```
 Barnier   | Germaine |           4
(3 lignes)
```

Un moteur de bases de données peut récupérer les données de plusieurs façons :

- faire un parcours séquentiel de la table `employes` en filtrant les enregistrements d'après leur nom, puis trier les données grâce à un algorithme ;
- faire un parcours d'index (s'il y en a un) sur la colonne `nom` pour trouver plus rapidement les enregistrements de la table `employes` satisfaisant le filtre `'B%'`, puis trier les données grâce à un algorithme ;
- faire un parcours d'index sur la colonne `num_service` pour récupérer les enregistrements déjà triés par service, et ne retourner que ceux vérifiant le prédicat `nom like 'B%'`.

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'opérations de base. Ces opérations travaillent sur des ensembles de lignes, généralement un ou deux. Chaque opération renvoie un seul ensemble de lignes. Le planificateur peut combiner ces opérations suivant certaines règles. Une opération 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.

Règles

- 1re règle : Récupérer le bon résultat
- 2e 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 (charge des CPU utilisés et nombre de CPU utilisés) 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).

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 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.

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 jointure (trois au total) ;
- agréger un ensemble de lignes avec une opération d'agrégation (trois là-aussi) ;
- trier un ensemble de lignes ;
- etc.

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.

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
  - les coûts liés à la parallélisation et au JIT
- Paramètres modifiables dynamiquement avec SET

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.

Divers paramètres permettent d'ajuster les coûts relatifs :

- `seq_page_cost` (défaut : 1) représente le coût relatif d'un accès séquentiel à un bloc sur le disque, c'est-à-dire à un bloc lu en même temps que ses voisins dans la table ;
- `random_page_cost` (défaut : 4) représente le coût relatif d'un accès aléatoire (isolé) à un bloc : 4 signifie que le temps d'accès de déplacement de la tête de lecture de façon aléatoire est estimé 4 fois plus important que le temps d'accès en séquentiel - ce sera moins avec un bon disque, voire 1.2 pour un SSD ;
- `cpu_tuple_cost` (défaut 0,01) représente le coût relatif de la manipulation d'une ligne en mémoire ;
- `cpu_index_tuple_cost` (défaut 0,005) répercute le coût de traitement d'une donnée issue d'un index ;
- `cpu_operator_cost` (défaut 0,0025) indique le coût d'application d'un opérateur sur une donnée ;
- `parallel_tuple_cost` (défaut 0,1) indique le coût de traitement d'une ligne lors d'un parcours parallélisé ;
- `parallel_setup_cost` (défaut : 1000) indique le coût de mise en place d'un parcours parallélisé, une procédure assez lourde qui ne se rentabilise pas pour les petites requêtes ;
- `jit_above_cost` (défaut 100 000), `jit_inline_above_cost` (500 000), `jit_optimize_above_cos` (500 000) représentent les seuils d'activation de divers niveaux du JIT (*Just In Time* ou compilation à la volée des requêtes), qui ne se rentabilise que sur les gros volumes.

En général, on ne modifie pas ces paramètres sans justification sérieuse. Le plus fréquemment, 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 constants. 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.

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.

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 planificateur 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).

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.

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, `date_embauche = '2006-09-01'`, la requête va ramener pratiquement l'intégralité de la table. PostgreSQL choisira donc une lecture séquentielle de la table, ou `Seq Scan` :

```
EXPLAIN (ANALYZE, TIMING OFF)
  SELECT *
  FROM employes_big
  WHERE date_embauche='2006-09-01';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..10901.69 rows=498998 width=40)
                                        (actual rows=499004 loops=1)
   Filter: (date_embauche = '2006-09-01'::date)
   Rows Removed by Filter: 11
 Planning time: 0.027 ms
 Execution time: 42.624 ms
```

La partie `cost` montre que l'optimiseur estime que la lecture va ramener 498 998 lignes. Comme on peut le voir, ce n'est pas exact : elle en récupère 499 004. Ce n'est qu'une estimation basée sur des statistiques selon la répartition des données et ces estimations seront la plupart du temps un peu erronées. L'important est de savoir si l'erreur est négligeable ou si elle est importante. Dans notre cas, elle est négligeable. On lit aussi que 11 lignes ont été filtrées pendant le parcours (et 499004+11 correspond bien aux 499015 lignes de la table).

Avec un filtre sur une valeur beaucoup plus sélective, la requête ne ramènera que 2 lignes. L'optimiseur préférera donc passer par l'index que l'on a créé :

```
EXPLAIN (ANALYZE, TIMING OFF)
  SELECT *
  FROM employes_big
  WHERE date_embauche='2006-01-01';
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
        (cost=0.42..4.44 rows=1 width=41) (actual rows=2 loops=1)
   Index Cond: (date_embauche = '2006-01-01'::date)
 Planning Time: 0.213 ms
 Execution Time: 0.090 ms
```

Dans ce deuxième essai, l'optimiseur estime ramener 1 ligne. En réalité, il en ramène 2. L'estimation reste relativement précise étant donné le volume de données.

Dans le premier cas, l'optimiseur prévoit de sélectionner l'essentiel de la table et 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.

Statistiques : table et index

- Taille
- Cardinalité
- Stocké dans `pg_class`
  - colonnes `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;
```

Statistiques

- 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.

Vue pg_stats

- Une ligne par colonne de chaque table et par index fonctionnel
- 3 colonnes d'identification
  - `schemaname`, `tablename`, `attname`
- 8 colonnes d'informations statistiques

La vue `pg_stats` a été créée pour faciliter la compréhension des statistiques récupérées par la commande ANALYZE.

Par exemple, pour une colonne donnée de la petite table `employes` :

```
# SELECT * FROM pg_stats
  WHERE  schemaname = 'public'
  AND tablename     = 'employes'
```

```
  AND attname       = 'date_embauche' \gx

-[ RECORD 1
 ↪  ]----------+------------------------------------------------------------
schemaname             | public
tablename              | employes
attname                | date_embauche
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -0.5
most_common_vals       |
 ↪  {2006-03-01,2006-09-01,2000-06-01,2005-03-06,2006-01-01}
most_common_freqs      | {0.214286,0.214286,0.142857,0.142857,0.142857}
histogram_bounds       | {2003-01-01,2006-06-01}
correlation            | 1
most_common_elems      | ¤
most_common_elem_freqs | ¤
elem_count_histogram   | ¤
```

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. Ce n'est pas le cas ici.

null_frac

Cette statistique correspond au pourcentage de valeurs `NULL` dans l'échantillon considéré. Elle est toujours calculée. Il n'y a pas de valeurs nulles dans l'exemple ci-dessus.

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`, `boolean`, `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 (c'est en moyenne le cas ici).

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,
- ou `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 égales (aussi appelées classes), 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 <.

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.

Échantillon statistique

- Se configure dans postgresql.conf

  - `default_statistics_target = 100`

- Configurable par colonne `sql 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 :

```sql
ALTER TABLE nom_table ALTER [ COLUMN ] nom_colonne SET STATISTICS valeur;
```

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

Nœud d'exécution

- Nœud

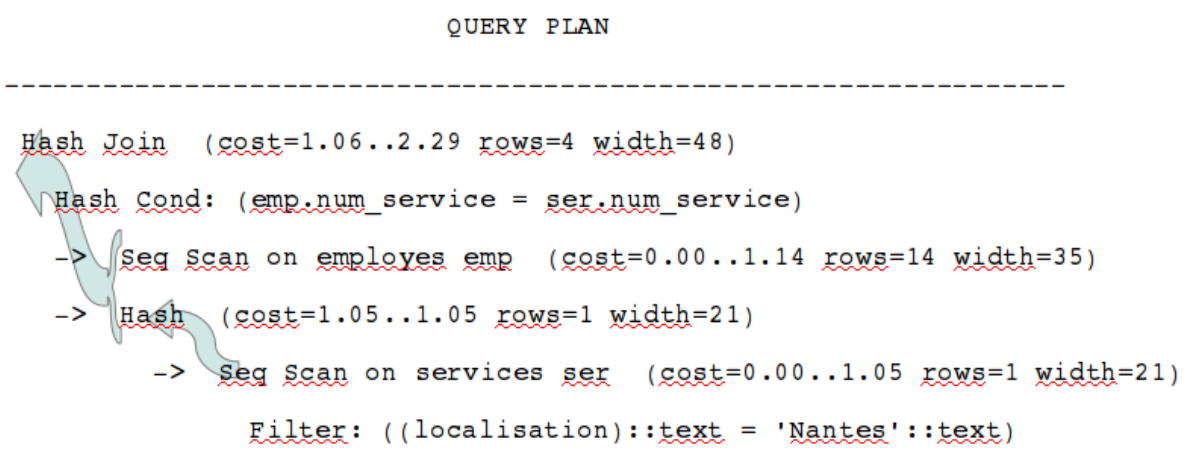
  - opération simple : lectures, jointures, tris, etc.
  - unité de traitement
  - produit et consomme des données

- 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 ensemble 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.

Lecture d'un plan



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 nœud 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.

Options de l'EXPLAIN

- Des options supplémentaires, dont :
  - ANALYZE
  - BUFFERS
- 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 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
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 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 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.

Option FORMAT

L'option `FORMAT` 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 :

Outils

- pev2
- explain.depesz.com
- auto_explain

Il existe quelques outils intéressants dans le cadre du planificateur : deux applications externes pour mieux appréhender un plan d'exécution, un module pour changer le comportement du planificateur.

Site pev2

- Site web proposant un affichage particulier du EXPLAIN ANALYZE

  - mais différent de celui de Depesz

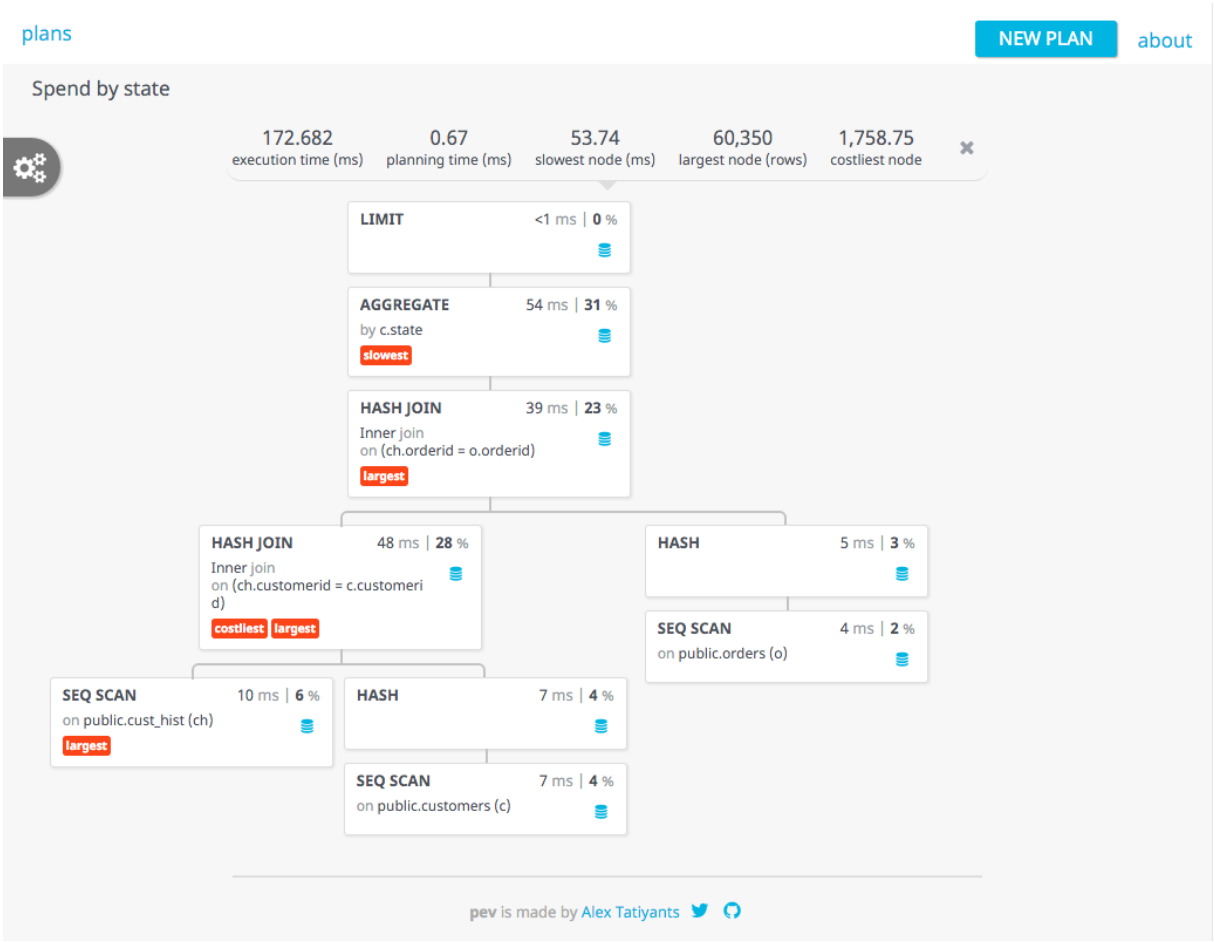
- Installable en local

PEV2 est un outil librement téléchargeable sur ce dépôt github[9]. 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[10] mais aussi installable en local.

---

pev2 - copie d'écran



---

[9] https://github.com/dalibo/pev2
[10] https://explain.dalibo.com

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[11].

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

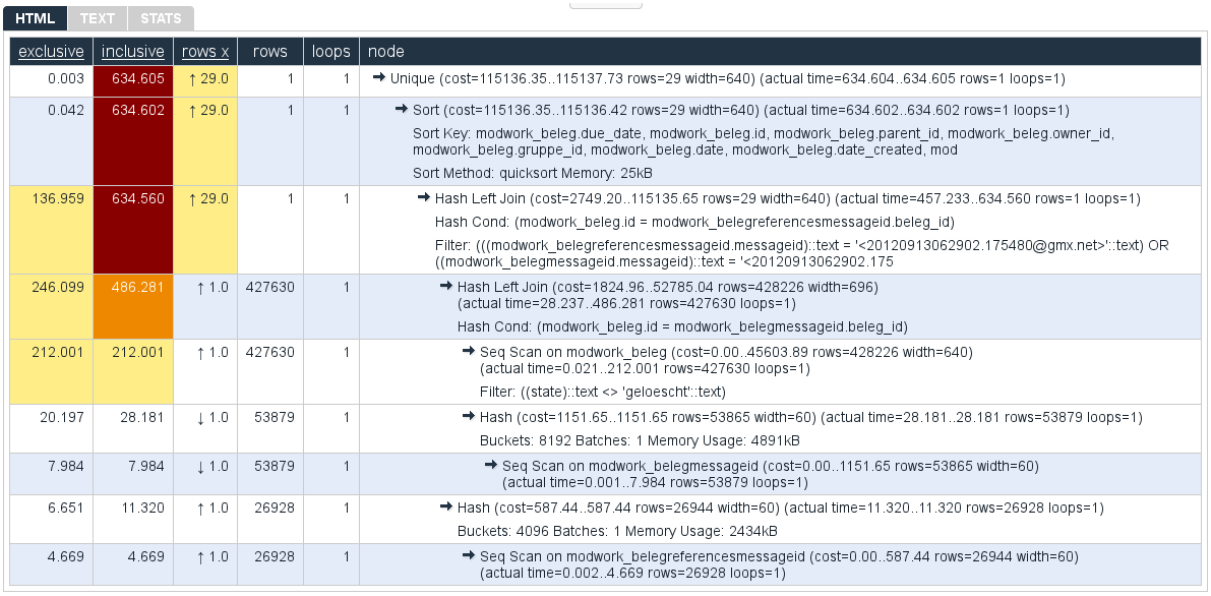
Plutôt que d'utiliser ce serveur web, il est possible d'installer ce site en local :

- le module explain en Perl[12]
- la partie site web[13]

---

[11] http://explain.depesz.com
[12] https://github.com/depesz/Pg--Explain
[13] https://github.com/depesz/explain.depesz.com

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 :

- `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.

Extension auto_explain

- Extension pour PostgreSQL
- 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`.

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 ré-actualisées 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.

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 ligne 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 bloc à accéder pour lire la table intégralement. Le paramètre `seq_page_cost` sera appliqué ensuite pour indiquer le coût de l'opération :

```
SELECT relname, relpages * current_setting('seq_page_cost')::float AS
 ↪  cout_acces
  FROM pg_class
 WHERE relname = 'pays';
 relname | cout_acces
---------+------------
 pays    |          1
```

Cependant, le coût d'accès seul ne représente pas le coût de la lecture des données. Une fois que le bloc est monté en mémoire, PostgreSQL doit décoder chaque ligne individuellement. L'optimiseur utilise `cpu_tuple_cost` pour estimer le coût de manipulation des lignes :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float AS cout
  FROM pg_class
```

```
 WHERE relname = 'pays';
 relname | cout
---------+------
 pays    | 1.25
```

On peut vérifier que le calcul est bon :

```
EXPLAIN SELECT * FROM pays;
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.25 rows=25 width=53)
(1 ligne)
```

Si l'on applique un filtre à la requête, les traitements seront plus lourds. Par exemple, en ajoutant le prédicat `WHERE pays = 'FR'`.

Il faut non seulement extraire les lignes les unes après les autres, mais il faut également appliquer l'opérateur de comparaison utilisé. L'optimiseur utilise le paramètre `cpu_operator_cost` pour déterminer le coût d'application d'un filtre :

```
SELECT relname,
       relpages * current_setting('seq_page_cost')::float
       + reltuples * current_setting('cpu_tuple_cost')::float
       + reltuples * current_setting('cpu_operator_cost')::float AS cost
  FROM pg_class
 WHERE relname = 'pays';
 relname |  cost
---------+--------
 pays    | 1.3125
```

En récupérant le plan d'exécution de la requête à laquelle est appliqué le filtre `WHERE pays = 'FR'`, on s'aperçoit que le calcul est juste, à l'arrondi près :

```
EXPLAIN SELECT * FROM pays WHERE code_pays = 'FR';
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on pays  (cost=0.00..1.31 rows=1 width=53)
   Filter: (code_pays = 'FR'::text)
(2 lignes)
```

Pour aller plus loin dans le calcul de sélectivité, de cardinalité et de coût, la documentation de PostgreSQL montre un exemple complet de calcul de sélectivité et indique les références des fichiers sources dans lesquels fouiller pour en savoir plus :Comment le planificateur utilise les statistiques[14].

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

Conclusion

PostgreSQL propose de nombreuses voies d'optimisation.

- choix du matériel
  - configuration pointilleuse
  - surveillance du bon fonctionnement

Cela passe en priorité par un bon choix des composants matériels et par une configuration pointilleuse. Mais ceci ne peut se faire qu'en connaissance de l'ensemble du système, et notamment des applications utilisant les bases de l'instance.

Lors du fonctionnement de votre instance, en cas de ralentissements, Utilisation de profiler :

- `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 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.

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.

Parcours

- Ne prend rien en entrée
- Mais renvoie un ensemble de données
  - Trié ou non, filtré ou non
- Exemples typiques
  - Parcours séquentiel d'une table, avec ou sans filtrage des enregistrements produits
  - Parcours par un index, avec ou sans filtrage supplémentaire

Les parcours sont les seules opérations qui lisent les données des tables (standards, temporaires ou non journalisées). Elles ne prennent donc rien en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

Il existe trois types de parcours que nous allons détailler :

- le parcours de table ;
- le parcours d'index ;
- le parcours de bitmap index, tous les trois pouvant recevoir des filtres supplémentaires en sortie.

Nous verrons aussi que PostgreSQL propose d'autres types de parcours.

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

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
- Parallélisation possible pour les index B-Tree

Parcours d'index bitmap

- En VO, Bitmap Index Scan / Bitmap Heap Scan
- 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
- Indes Btree uniquement

Parcours d'index seul

```
* Nœud Index Only Scan pour les index :
    * Index B-Tree
    * Index SP-GiST
    * Index GiST
```

Types de jointures

- Nested Loop : boucle imbriqué
- Hash Join : hachage de la table interne
- Merge Join : tri-fusion
- Parallélisation possible pour ces jointures
- Hash Anti et Semi Join

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

---

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

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

---

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

Autres

- Nœud HashSetOp Except
  - instructions EXCEPT et EXCEPT ALL
- Nœud HashSetOp Intersect
  - instructions INTERSECT et INTERSECT ALL

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.

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 :
  - quicksort
  - Incremental Sort (grâce à un parcours d'index)

  - top-N heapsort (si clause LIMIT)
- Le tri sur disque : external merge

Aggregate

- Agrégat complet
- Pour un seul résultat

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

---

Group Aggregate

- Reçoit des données déjà triées
- Parcours des données
  - Regroupement du groupe précédent arrivé à une donnée différente

---

Unique

- Reçoit des données déjà triées
- Parcours des données
  - Renvoi de la donnée précédente une fois arrivé à une donnée différente
- Résultat trié

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)

Memoize

- Disponible en version 14
- Cache de résultat pour les Nested Loop

Parallélisation

- Parcours séquentiel
- Jointures Nested Loop, Hash Join et Merge Join
- Agrégats
- Parcours d'index (Btree uniquement)
- Création d'index BTree (v11)
- Certaines créations de table et vues matérialisées (v11)
- `DISTINCT` (v15)

La parallélisation de l'exécution d'une requête est disponible depuis la version 9.6 de PostgreSQL. Elle permet de paralléliser les parcours de table (`SeqScan`), les jointures (`Nested Loop` et `Hash Join`), ainsi que certaines fonctions d'agrégat (comme `min`, `max`, `avg`, `sum`, etc.).

La version 10 active le parallélisme par défaut et l'améliore en parallélisant les parcours d'index B-Tree (`Index Scan`, `Index Only Scan` et `Bitmap Scan`) et les jointures de type `Merge Join`.

La version 11 est une nouvelle source d'améliorations avec la possibilité de créer des index B-Tree de façon parallélisée. La parallélisation est disponible pour les autres types d'index mais ils n'en font pas usage pour l'instant. Certaines créations de table, `CREATE TABLE ... AS`, `SELECT INTO`

et `CREATE MATERIALIZED VIEW`, sont aussi parallélisables. La clause `LIMIT` est passée aux processus de parallélisation.

Limites actuelles de la parallélisation

- Pas sur les écritures de données
- Très peu d'opérations DDL gérées
- Pas en cas de verrous
- Pas sur les curseurs
- En évolution à chaque version

Même si cette fonctionnalité évolue au fil des versions majeures, des limitations assez fortes restent présentes, notamment :

- pas de parallélisation pour les écritures de données (`INSERT`, `UPDATE`,`DELETE`, etc.),
- peu de parallélisation sur les opérations DDL (par exemple un `ALTER TABLE` ne peut pas être parallélisé)

Il y a des cas particuliers, notamment `CREATE TABLE...AS` ou `CREATE MATERIALIZED VIEW`, parallélisables à partir de la v11 ; ou le niveau d'isolation *serializable*: avant la v12, il ne permet aucune parallélisation.

Action de l'optimiseur

- 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

Suppression des jointures externes inutiles

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.

Sous certaines conditions, PostgreSQL peut supprimer des jointures externes, à condition que le résultat ne soit pas modifié. Dans l'exemple suivant, il ne sert à rien d'aller consulter la table `services` (ni données à récupérer, ni filtrage à faire, et même si la table est vide, le `LEFT JOIN` ne provoquera la disparition d'aucune ligne) :

```
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=19)
   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=0.00..2.27 rows=5 width=19)
   ->  Seq Scan on services s  (cost=0.00..1.05 rows=1 width=4)
         Filter: (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=0.00..2.25 rows=2 width=64)
   ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=21)
         Filter: (num_service = 1)
   ->  Seq Scan on employes emp  (cost=0.00..1.18 rows=2 width=43)
         Filter: (num_service = 1)
```

La sous-requête `ser` a été remontée dans l'arbre de requête pour être intégrée 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))
```

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 cas de problème, il est possible d'utiliser une CTE avec le mot-clé `MATERIALIZED` (*Common Table Expression*, clause `WITH`) pour bloquer cette optimisation :

```
EXPLAIN
  WITH e AS MATERIALIZED (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

Voici deux fonctions, la première écrite en SQL, la seconde en PL/pgSQL :

```
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 :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes
  WHERE date_embauche = add_months_plpgsql(now()::date, -1);

                                QUERY PLAN
-------------------------------------------------------------------------
 ↪  ---
 Seq Scan on employes (actual time=0.354..0.354 rows=0 loops=1)
   Filter: (date_embauche = add_months_plpgsql((now())::date,
    ↪  '-1'::integer))
   Rows Removed by Filter: 14
   Buffers: shared hit=1
 Planning Time: 0.199 ms
 Execution Time: 0.509 ms
```

Effectivement, 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 est directement intégrée dans la clause de filtrage de la requête :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes
  WHERE date_embauche = add_months_sql(now()::date, -1);
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on employes (actual time=0.014..0.014 rows=0 loops=1)
   Filter: (date_embauche = (((now())::date + '-1 mons'::interval))::date)
   Rows Removed by Filter: 14
   Buffers: shared hit=1
 Planning Time: 0.111 ms
 Execution Time: 0.027 ms
```

Le temps d'exécution a été divisé presque par 2 sur ce jeu de donnée très réduit, montrant l'impact de l'appel d'une fonction dans une clause de filtrage.

Dans les deux cas ci-dessus, PostgreSQL a négligé l'index sur `date_embauche` : la table ne faisait de toute façon qu'un bloc ! Mais pour de plus grosses tables l'index sera nécessaire, et la différence entre fonctions PL/pgSQL et SQL devient alors encore plus flagrant. Avec la même requête sur la table `employes_big`, beaucoup plus grosse, on obtient ceci :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes_big
  WHERE date_embauche = add_months_plpgsql(now()::date, -1);
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 ↪  ---
 Seq Scan on employes_big (actual time=464.531..464.531 rows=0 loops=1)
   Filter: (date_embauche = add_months_plpgsql((now())::date,
    ↪  '-1'::integer))
   Rows Removed by Filter: 499015
   Buffers: shared hit=4664
 Planning Time: 0.176 ms
 Execution Time: 465.848 ms
```

La fonction portant sur une « boîte noire », l'optimiseur n'a comme possibilité que le parcours complet de la table.

```
EXPLAIN (ANALYZE, BUFFERS, COSTS off)
  SELECT *
  FROM employes_big
  WHERE date_embauche = add_months_sql(now()::date, -1);

                                  QUERY PLAN
------------------------------------------------------------------------------
 ↪  ----
 Index Scan using employes_big_date_embauche_idx on employes_big
                                     (actual time=0.016..0.016 rows=0 loops=1)
   Index Cond: (date_embauche = (((now())::date + '-1
    ↪  mons'::interval))::date)
   Buffers: shared hit=3
 Planning Time: 0.143 ms
 Execution Time: 0.032 ms
```

La fonction SQL est intégrée, l'optimiseur voit le critère dans `date_embauche` et peut donc se poser la question de l'utiliser (et ici, la réponse est oui : 3 blocs contre 4664, tous présents dans le cache dans cet exemple).

D'où une exécution beaucoup plus rapide.

Travaux Dirigés 4

- Optimisation et index
  - Les problèmes les plus courants
  - Comprendre EXPLAIN

Question de cours

- Quels sont les différents moyens disponibles pour optimiser le fonctionnement d'une instance ?
- Quelle est la méthode utilisée par PostgreSQL pour sélectionner le meilleur plan afin d'obtenir le résultat d'une requête ?
- Comment obtenir ce meilleur plan à partir d'une requête ?

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 lent voire catastrophiques de certaines requêtes.

Colonnes corrélées

```sql
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`.

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)
```

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
- Attention au collationnement
- Ne pas oublier pg_trgm 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);
```

Une autre possibilité pour la non utilisation d'un index est le collationnement. Si le collationnement de la requête diffère du collationnement de la colonne de l'index, l'index ne pourra pas être utilisé.

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.

Dédoublonnage

```
SELECT DISTINCT t1.* FROM t1 JOIN t2 ON (t1.id=t2.t1_id);
```

- DISTINCT est souvent utilisé pour dédoublonner les lignes de t1
  - mais génère un tri qui pénalise les performances
- GROUP BY est plus rapide
- Une clé primaire permet de dédoublonner efficacement des lignes
  - à utiliser avec GROUP BY

L'exemple ci-dessous montre une requête qui récupère les commandes qui ont des lignes de commandes et réalise le dédoublonnage avec DISTINCT. Le plan d'exécution montre une opération de tri qui

a nécessité un fichier temporaire de 60Mo. Toutes ces opérations sont assez gourmandes, la requête répond en 5,9s :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
tpc-# SELECT DISTINCT commandes.* FROM commandes
sql1=# JOIN lignes_commandes USING (numero_commande);
                                    QUERY PLAN
-------------------------------------------------------------------------------
 ↪  ----
 Unique (actual time=5146.904..5833.600 rows=168749 loops=1)
   ->  Sort (actual time=5146.902..5307.633 rows=675543 loops=1)
         Sort Key: commandes.numero_commande, commandes.client_id,
                   commandes.etat_commande, commandes.prix_total,
                   commandes.date_commande, commandes.priorite_commande,
                   commandes.vendeur, commandes.priorite_expedition,
                   commandes.commentaire
         Sort Method: external sort  Disk: 60760kB
         ->  Merge Join (actual time=0.061..601.674 rows=675543 loops=1)
               Merge Cond: (commandes.numero_commande =
                            lignes_commandes.numero_commande)
               ->  Index Scan using orders_pkey on commandes
                         (actual time=0.026..71.544 rows=168750 loops=1)
               ->  Index Only Scan using lignes_com_pkey on lignes_commandes
                         (actual time=0.025..175.321 rows=675543 loops=1)
                     Heap Fetches: 0
 Total runtime: 5849.996 ms
```

En restreignant les colonnes récupérées à celle réellement intéressante et en utilisant GROUP BY au lieu du DISTINCT, le temps d'exécution tombe à 4,5s :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
 FROM commandes
 JOIN lignes_commandes
      USING (numero_commande)
 GROUP BY commandes.numero_commande, commandes.etat_commande,
          commandes.prix_total, commandes.date_commande,
          commandes.priorite_commande, commandes.vendeur,
          commandes.priorite_expedition;
                                        QUERY PLAN
```

```
------------------------------------------------------------------------------
 ↪  ----
 Group (actual time=4025.069..4663.992 rows=168749 loops=1)
   ->  Sort (actual time=4025.065..4191.820 rows=675543 loops=1)
         Sort Key: commandes.numero_commande, commandes.etat_commande,
                   commandes.prix_total, commandes.date_commande,
                   commandes.priorite_commande, commandes.vendeur,
                   commandes.priorite_expedition
         Sort Method: external sort  Disk: 46232kB
         ->  Merge Join (actual time=0.062..579.852 rows=675543 loops=1)
               Merge Cond: (commandes.numero_commande =
                            lignes_commandes.numero_commande)
               ->  Index Scan using orders_pkey on commandes
                            (actual time=0.027..70.212 rows=168750 loops=1)
               ->  Index Only Scan using lignes_com_pkey on lignes_commandes
                            (actual time=0.026..170.555 rows=675543 loops=1)
                     Heap Fetches: 0
 Total runtime: 4676.829 ms
```

Mais, à partir de PostgreSQL 9.1, il est possible d'améliorer encore les temps d'exécution de cette requête. Dans le plan d'exécution précédent, on voit que l'opération `Sort` est très gourmande car le tri des lignes est réalisé sur plusieurs colonnes. Or, la table `commandes` a une clé primaire sur la colonne `numero_commande`. Cette clé primaire permet d'assurer que toutes les lignes sont uniques dans la table `commandes`. Si l'opération GROUP BY ne porte plus que la clé primaire, PostgreSQL peut utiliser le résultat de la lecture par index sur commandes pour faire le regroupement. Le temps d'exécution passe à environ 580ms :

```
tpc=# EXPLAIN (ANALYZE on, COSTS off)
SELECT commandes.numero_commande, commandes.etat_commande,
    commandes.prix_total, commandes.date_commande,
    commandes.priorite_commande, commandes.vendeur,
    commandes.priorite_expedition
  FROM commandes
  JOIN lignes_commandes
       USING (numero_commande)
 GROUP BY commandes.numero_commande;
                                          QUERY PLAN
------------------------------------------------------------------------------
 ↪  ----
 Group (actual time=0.067..580.198 rows=168749 loops=1)
   ->  Merge Join (actual time=0.061..435.154 rows=675543 loops=1)
         Merge Cond: (commandes.numero_commande =
                      lignes_commandes.numero_commande)
```

```
        ->  Index Scan using orders_pkey on commandes
                    (actual time=0.027..49.784 rows=168750 loops=1)
        ->  Index Only Scan using lignes_commandes_pkey on lignes_commandes
                    (actual time=0.025..131.606 rows=675543 loops=1)
              Heap Fetches: 0
Total runtime: 584.624 ms
```

Les opérations de dédoublonnages sont régulièrement utilisées pour assurer que les lignes retournées par une requête apparaissent de manière unique. Elles sont souvent inutiles, ou peuvent à minima être largement améliorées en utilisant les propriétés du modèle de données (les clés primaires) et des opérations plus adéquates (`GROUP BY clé_primaire`). Lorsque vous rencontrez des requêtes utilisant DISTINCT, vérifiez que le DISTINCT est vraiment pertinent ou s'il ne peut pas être remplacé par un GROUP BY qui pourrait tirer partie de la lecture d'un index.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[15].

Index inutilisés

- Trop de lignes retournées
- Prédicat incluant une transformation : `sql WHERE col1 + 2 > 5`
- Statistiques pas à jour ou peu précises
- Opérateur non-supporté par l'index : `sql 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 ?

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

[15] http://www.depesz.com/index.php/2010/04/19/getting-unique-elements/

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[16]

Écriture du SQL

- `NOT IN` avec une sous-requête
  - à remplacer par `NOT EXISTS`
- Utilisation systématique de `UNION ALL` au lieu de `UNION`
  - `UNION` 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
 WHERE numero_commande NOT IN (SELECT numero_commande
                                 FROM lignes_commandes);
```

[16] http://www.depesz.com/index.php/2010/09/09/why-is-my-index-not-being-used/

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);
```

Enoncé

Comprendre EXPLAIN

Schéma de la base cave

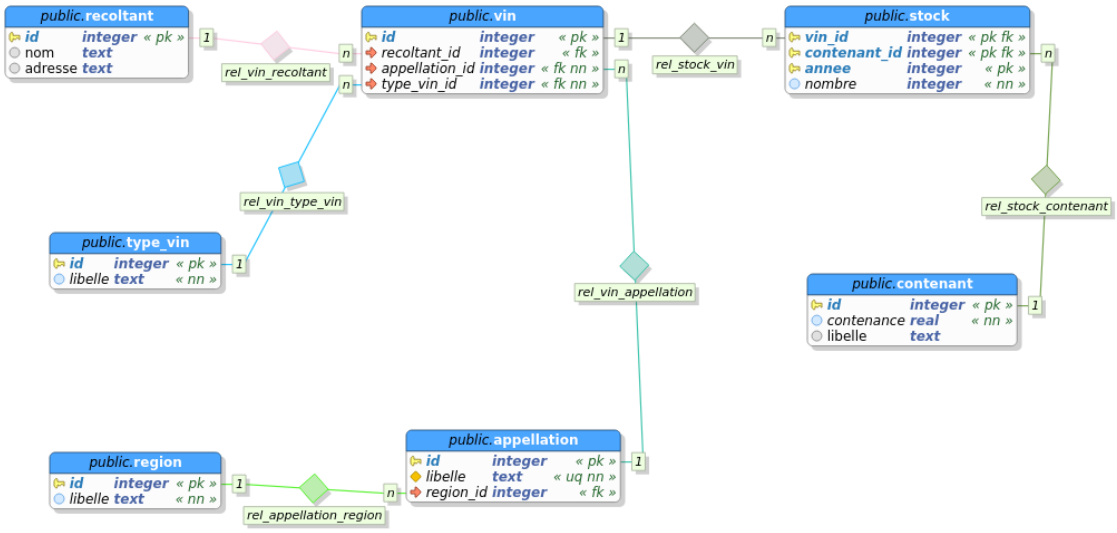


Figure 1: Schéma de la base cave

Exercice 1

```
cave=> EXPLAIN SELECT s.nombre
  FROM stock s
  JOIN vin v ON s.vin_id = v.id
  LEFT JOIN appellation a ON v.appellation_id = a.id;
                          QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=169.60..25290.86 rows=861611 width=4)
   Hash Cond: (s.vin_id = v.id)
   ->  Seq Scan on stock s  (cost=0.00..13274.11 rows=861611 width=8)
   ->  Hash  (cost=93.71..93.71 rows=6071 width=8)
         ->  Seq Scan on vin v  (cost=0.00..93.71 rows=6071 width=8)
(5 lignes)
```

1. Combien de noeuds sont présent dans ce plan ?
2. Quel est le nombre de lignes qui seront remontés par cette requête ?
3. Ce résultat est-il certain ? Pourquoi ?
4. Que remarque-t-on au niveau des jointures ?

```
cave=> EXPLAIN (ANALYSE,BUFFERS) SELECT nombre
  FROM stock s
  JOIN vin v ON s.vin_id = v.id
  LEFT JOIN appellation a ON v.appellation_id = a.id;
                                QUERY PLAN
------------------------------------------------------------------------------
 ↪  ---
 Hash Join  (cost=169.60..25290.86 rows=861611 width=4)
            (actual time=7.378..461.477 rows=861611 loops=1)
   Hash Cond: (s.vin_id = v.id)
   Buffers: shared hit=3 read=4691
   ->  Seq Scan on stock s  (cost=0.00..13274.11 rows=861611 width=8)
                            (actual time=0.060..134.329 rows=861611 loops=1)
         Buffers: shared read=4658
   ->  Hash  (cost=93.71..93.71 rows=6071 width=8)
             (actual time=7.168..7.168 rows=6071 loops=1)
         Buckets: 8192  Batches: 1  Memory Usage: 302kB
         Buffers: shared read=33
         ->  Seq Scan on vin v  (cost=0.00..93.71 rows=6071 width=8)
                                (actual time=0.047..3.505 rows=6071 loops=1)
               Buffers: shared read=33
 Planning time: 1.146 ms
 Execution time: 511.313 ms
(12 lignes)

cave=> EXPLAIN (ANALYSE,BUFFERS) SELECT nombre
  FROM stock s
  JOIN vin v ON s.vin_id = v.id
  LEFT JOIN appellation a ON v.appellation_id = a.id;
                                QUERY PLAN
------------------------------------------------------------------------------
 ↪  ---
 Hash Join  (cost=169.60..25290.86 rows=861611 width=4)
            (actual time=5.291..370.185 rows=861611 loops=1)
   Hash Cond: (s.vin_id = v.id)
   Buffers: shared hit=4691
   ->  Seq Scan on stock s  (cost=0.00..13274.11 rows=861611 width=8)
                            (actual time=0.022..93.276 rows=861611 loops=1)
         Buffers: shared hit=4658
   ->  Hash  (cost=93.71..93.71 rows=6071 width=8)
             (actual time=5.228..5.228 rows=6071 loops=1)
         Buckets: 8192  Batches: 1  Memory Usage: 302kB
         Buffers: shared hit=33
```

```
         ->  Seq Scan on vin v  (cost=0.00..93.71 rows=6071 width=8)
                                (actual time=0.021..2.392 rows=6071 loops=1)
               Buffers: shared hit=33
 Planning time: 0.303 ms
 Execution time: 414.245 ms
(12 lignes)
```

5. Combien de noeuds sont présent dans ces plans ?
6. Comment peut-on expliquer la différence dans les temps d'exécution ?

Exercice 2

```
cave=> EXPLAIN ANALYSE SELECT annee FROM stock s WHERE annee<1952;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on stock s  (cost=0.00..15429.62 rows=287243 width=4)
                      (actual time=0.146..208.687 rows=34388 loops=1)
   Filter: (annee < 1952)
   Rows Removed by Filter: 827223
 Planning time: 0.295 ms
 Execution time: 212.568 ms
(5 lignes)
```

1. Quelle est l'opération choisie par l'optimiseur ?
2. Que remarque-t-on entre le plan choisi et l'opération effectuée ?

```
cave=> EXPLAIN ANALYSE SELECT annee FROM stock s WHERE annee<1952;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..13606.36 rows=34608 width=4)
         (actual time=0.326..77.223 rows=34388 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on stock s
             (cost=0.00..9145.56 rows=14420 width=4)
             (actual time=0.091..59.782 rows=11463 loops=3)
         Filter: (annee < 1952)
         Rows Removed by Filter: 275741
 Planning time: 0.152 ms
 Execution time: 82.376 ms
(8 lignes)
```

3. Quelle est cette fois l'opération choisie par l'optimiseur ?
4. Que remarque-t-on entre le plan choisi et l'opération effectuée ?
5. Que s'est-il produit entre les deux exécutions des requêtes ?

Exercice 3

```
cave=> EXPLAIN ANALYSE SELECT annee
  FROM stock s
   WHERE annee<1950;
                                QUERY PLAN
-------------------------------------------------------------------------
 Gather  (cost=1000.00..10145.66 rows=1 width=4)
         (actual time=89.613..89.613 rows=0 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on stock s
          (cost=0.00..9145.56 rows=1 width=4)
          (actual time=85.824..85.824 rows=0 loops=3)
         Filter: (annee < 1950)
         Rows Removed by Filter: 287204
 Planning time: 0.155 ms
 Execution time: 92.065 ms
(8 lignes)

cave=> (...)

cave=> EXPLAIN ANALYSE SELECT annee
  FROM stock s
   WHERE annee<1950;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using idx_stock_annee on stock s
      (cost=0.42..5.52 rows=1 width=4)
      (actual time=0.024..0.024 rows=0 loops=1)
   Index Cond: (annee < 1950)
   Heap Fetches: 0
 Planning time: 0.195 ms
 Execution time: 0.071 ms
(5 lignes)
```

1. Que s'est-il produit entre les deux exécutions des requêtes ?

2. Quelle opération a pu être effectuée pour gagner un facteur 1200 sur le temps d'exécution de cette requête ?

Exercice 4

```
cave=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM stock
  WHERE annee>1980 ORDER BY (annee,nombre);
                                QUERY PLAN
-------------------------------------------------------------------------------
 ↪  --
 Sort  (cost=56382.10..57217.79 rows=334276 width=48)
       (actual time=2832.406..2967.067 rows=338057 loops=1)
   Sort Key: (ROW(annee, nombre))
   Sort Method: external merge  Disk: 18176kB
   Buffers: shared hit=4658, temp read=2273 written=2273
   ->  Seq Scan on stock  (cost=0.00..15428.14 rows=334276 width=48)
                          (actual time=116.801..248.923 rows=338057 loops=1)
         Filter: (annee > 1980)
         Rows Removed by Filter: 523554
         Buffers: shared hit=4658
 Planning time: 0.220 ms
 Execution time: 3003.787 ms
(10 lignes)

cave=> (...)

cave=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM stock
  WHERE annee>1980 ORDER BY (annee,nombre);
                                QUERY PLAN
-------------------------------------------------------------------------------
 ↪  --
 Sort  (cost=46099.10..46934.79 rows=334276 width=48)
       (actual time=984.267..1022.301 rows=338057 loops=1)
   Sort Key: (ROW(annee, nombre))
   Sort Method: quicksort  Memory: 38227kB
   Buffers: shared hit=4658
   ->  Seq Scan on stock  (cost=0.00..15428.14 rows=334276 width=48)
                          (actual time=112.079..248.697 rows=338057 loops=1)
         Filter: (annee > 1980)
         Rows Removed by Filter: 523554
```

```
         Buffers: shared hit=4658
 Planning time: 0.200 ms
 Execution time: 1058.208 ms
(10 lignes)
```

1. Que s'est-il produit entre les deux exécutions des requêtes ?
2. Quelle opération a été effectuée pour gagner un facteur 4 sur le temps d'exécution de cette requête ?

Exercice 5

```
cave=> EXPLAIN ANALYSE SELECT nombre
  FROM stock s
   WHERE annee>1990;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on stock s  (cost=0.00..15428.14 rows=167095 width=4)
                      (actual time=117.058..160.155 rows=169437 loops=1)
   Filter: (annee > 1990)
   Rows Removed by Filter: 692174
 Planning time: 0.137 ms
 Execution time: 167.216 ms
(5 lignes)

cave=> CREATE INDEX idx_stock_nombre_annee ON stock (nombre,annee) ;
CREATE INDEX
cave=> EXPLAIN ANALYSE SELECT nombre
  FROM stock s
   WHERE annee>1990;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on stock s  (cost=0.00..15428.14 rows=167095 width=4)
                      (actual time=145.586..192.560 rows=169437 loops=1)
   Filter: (annee > 1990)
   Rows Removed by Filter: 692174
 Planning time: 0.395 ms
 Execution time: 200.287 ms
(5 lignes)
```

1. Pourquoi le plannificateur n'a-t-il pas utilisé l'index créé lors de l'exécution de la seconde requête ?

2. Proposez un autre index pour optimiser la première requête.

```
cave=> EXPLAIN ANALYSE SELECT nombre
  FROM stock s
   WHERE annee>1990 AND nombre >20;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using idx_stock_nombre_annee on stock s
      (cost=0.42..8.44 rows=1 width=4)
      (actual time=0.031..0.031 rows=0 loops=1)
   Index Cond: ((nombre > 20) AND (annee > 1990))
   Heap Fetches: 0
 Planning time: 0.294 ms
 Execution time: 0.084 ms
(5 lignes)
```

3. Pourquoi le plannificateur a-t-il utilisé l'index créé lors de l'exécution de cette requête ?

```
cave=> EXPLAIN ANALYSE SELECT nombre
  FROM stock s
   WHERE annee>1960 AND nombre>12;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on stock s  (cost=0.00..17582.17 rows=308922 width=4)
                      (actual time=40.806..190.476 rows=309201 loops=1)
   Filter: ((annee > 1960) AND (nombre > 12))
   Rows Removed by Filter: 552410
 Planning time: 0.198 ms
 Execution time: 202.977 ms
(5 lignes)
```

4. Pourquoi le plannificateur n'a-t-il pas utilisé l'index créé lors de l'exécution de cette requête ?

Exercice 6

```
cave=> CREATE INDEX idx_stock_annee ON stock (annee);
CREATE INDEX
cave=> EXPLAIN ANALYZE SELECT annee,nombre
  FROM stock s WHERE (annee%4=0) AND ((annee%100<>0) OR (annee%400=0));
                                QUERY PLAN
-------------------------------------------------------------------------
 ↪  -----
```

```
 Gather  (cost=1000.00..15061.81 rows=4287 width=8)
         (actual time=10.898..158.395 rows=219937 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on stock_bis s
          (cost=0.00..13633.11 rows=1786 width=8)
          (actual time=6.077..96.088 rows=73312 loops=3)
         Filter: (((annee%4) = 0) AND (((annee%100) <> 0) OR ((annee%400) =
          ↪  0)))
         Rows Removed by Filter: 213891
 Planning time: 0.166 ms
 Execution time: 172.671 ms
(8 lignes)
```

1. Quelle a été l'impact de la création de l'index *idx_stock_annee* ?
2. Comment pourrait-on améliorer les performances de cette requête ?

Exercice 7 Écrire la requête permettant de récupérer la somme du nombre de bouteilles en stock pour chaque region dont le stock pour chaque vin est inférieur à 10, l'adresse du récoltant devant contenir un 33 et le libellé de l'appellation la chaine de caractère eau. Le résultat devra être ordonné par le somme du nombre de bouteille.

Solutions du TD 4

Question de cours

- Quels sont les différents moyens disponibles pour optimiser le fonctionnement d'une instance ?
  - Pour optimiser le fonctionnement d'une instance, on doit rechercher le ou les goulots d'étranglement gênant la bonne marche du système. Cela nécessite une analyse de tous les composants : requêtes et schéma SQL, configuration physique et logicielle de l'instance, configuration du système de fichiers et du noyau et enfin le matériel utilisé.
- Quelle est la méthode utilisée par PostgreSQL pour sélectionner le meilleur plan afin d'obtenir le résultat d'une requête ?

  - Le planificateur de PostgreSQL dispose de nombreux algorithmes pour calculer la bonne réponse à une requête. En se basant sur un mécanisme de coût, il fixe un prix à tous les plans et choisit le moins cher.

- Comment obtenir ce meilleur plan à partir d'une requête ?
  - Le mot clé `EXPLAIN` placé avant le texte de la requête permet d'obtenir le plan optimum choisi par le planificateur.

Exercice 1

1. Combien de noeuds sont présent dans ce plan ?

4 noeuds d'exécution sont présents :

- Scan séquentiel sur vin,
- Hachage du noeud précédent (pour préparer la jointure),
- Scan séquentiel sur stock,
- Jonction par hachage des 2 noeuds précédents sur `s.vin_id = v.id`

2. Quel est le nombre de lignes qui seront remontés par cette requête ?

861 611 lignes seront remontées.

3. Ce résultat est-il certain ? Pourquoi ?

L'opération effectuée est un `EXPLAIN`. Le nombre de ligne retourné est calculé en utilisant des statistiques sur les données. Le nombre de lignes estimé est donc approximatif. Un `EXPLAIN ANALYSE` effectue l'opération et permet d'obtenir une certitude sur le résultat.

4. Que remarque-t-on au niveau des jointures ?

Un jointure a été effectuée sur *appellation*. Cette table n'étant d'aucune utilité pour remonter le bon résultat, le planificateur a supprimé cette joinure.

5. Combien de noeuds sont présent dans ce plan ?

Comme précédemment, 4 noeuds d'exécution sont présents :

- Scan séquentiel sur vin,
- Hachage du noeud précédent (pour préparer la jointure),
- Scan séquentiel sur stock,
- Jonction par hachage des 2 noeuds précédents sur `s.vin_id = v.id`

6. Comment peut-on expliquer la différence dans les temps d'exécution ?

Dans le premier cas, les données ont dûes être lues par PostgreSQL et chargées dans son cache. Ces données ont pu être lues à partir des disques ou bien depuis le cache de l'OS.

Dans le deuxième plan d'exécution, les données étaient présentes dans le cache de PostgreSQL. La plannification et l'exécution de la requête ont été plus rapide.

Pour tester les temps pris par une requête, il est important de la lancer plusieurs fois de suite. Afin de ne pas prendre en compte le temps pris pour la lecture des données.

Si votre serveur a des latences à cause du besoin de lectures de données, la solution le plus simple est d'ajouter de la RAM.

Exercice 2

1. Quelle est l'opération choisie par l'optimiseur ?

L'optimiseur a choisi un parcours séquentiel de la table stock.

2. Que remarque-t-on entre le plan choisi et l'opération effectuée ?

L'optimiseur avait prévu, en utilisant ses statistiques, de remonter 287 243 lignes. L'option `ANALYSE` du `EXPLAIN` remonte le nombre réel de lignes soit 34 388.

3. Quelle est cette fois l'opération choisie par l'optimiseur ?

L'optimiseur a choisi de parcourir la table stock avec un scan séquentiel, mais cette fois-ci, de façon parallèle avec 2 *workers*.

4. Que remarque-t-on entre le plan choisi et l'opération effectuée ?

Le nombre de lignes estimées est comme précédement faux. Il est cependant bien plus précis (moins d'1% d'erreur).

5. Que s'est-il produit entre les deux exécutions des requêtes ?

Les statistiques n'étaient pas à jour lors de la première exécution de la requête. Un `ANALYZE stock` a permis de créer ou mettre à jour les statistiques. Dans le premier cas, les statistiques étaient vides car la table venait d'être créée. L'optimiseur doit évidemment fournir un plan d'exécution même non optimal. Il estime que le filtre remontera 30% des lignes.

Exercice 3

1. Que s'est-il produit entre les deux exécutions des requêtes ?

L'optimiseur a choisi d'utiliser un scan d'index seul à la place d'un scan séquentiel parallélisé.

2. Quelle opération a pu être effectuée pour gagner un facteur 1200 sur le temps d'exécution de cette requête ?

Un index a été créé sur la table *stock*. Cet index comprenait en premier la colonne *annee*. Voici une possibilité d'ordre SQL :

```
cave=> CREATE INDEX idx_stock_annee ON stock (annee);
```

Exercice 4

1. Que s'est-il produit entre les deux exécutions des requêtes ?

Le temps d'exécution est passé de 4,5 secondes à moins de 1 seconde. Ce gain de temps est dû au passage d'un tri sur disque (*external merge Disk*) a un tri en mémoire (*quicksort Memory*).

Dans le premier cas, 2 273 x 8 Ko soit 18 Mo de données ont dûes être écrites et lues sur disque.

2. Quelle opération a été effectuée pour gagner un facteur 4 sur le temps d'exécution de cette requête ?

Les tris utilisent la mémoire de travail de la connection (`work_mem`) et non la mémoire partagée de PostgreSQL (`shared_buffers`). La méthode pour augmenter La mémoire de travail pour une requête particulière est d'exécuter la requête suivante au préalable :

```
cave=> SET work_mem='40MB';
```

Ce paramètre peut être configuré au niveau global, au niveau d'une base de données, pour un role spécifique, pour une connection.

Exercice 5

1. Pourquoi le plannificateur n'a-t-il pas utilisé l'index créé ?

L'index créé est un index *btree* sur les colonnes *nombre* et *annee*. L'odre des colonnes est important. Si la première colonne de l'index n'est pas utilisé dans le filtre, il ne peut pas être utilisé.

2. Proposez un autre index pour optimiser le première requête.

Un index sur la colonne *annee* uniquement pourra être utilisé :

```
cave=> CREATE INDEX idx_stock_annee ON stock (annee);
```

3. Pourquoi le plannificateur a-t-il utilisé l'index créé lors de l'exécution de la troisième requête ?

La requête possédait un prédicat sur les 2 colonnes de l'index, il pouvait donc l'utiliser pour trouver le bon résultat.

4. Pourquoi le plannificateur n'a-t-il pas utilisé l'index créé lors de l'exécution de cette requête ?

Il est plus coûteux de lire toute une table en passant par un index plutôt qu'en effectuant un scan séquentiel. En effet le coût de lecture aléatoire est par défaut de 4 pour un coût de lecture séquentiel à 1.

L'optimiseur a vu que la sélectivité du filtre était faible (36% des lignes remontées). Il a donc préféré passer par un scan séquentiel.

Si vous possédez des disques en RAID 10, des disques SSD ou un SAN performant, vous pouvez baisser le coût de la lecture aléatoire. Le plannificateur utilisera alors plus souvent les index :

```
cave=> set random_page_cost=3;
SET
cave=> EXPLAIN ANALYSE SELECT nombre
  FROM stock s
   WHERE annee>1960 AND nombre >12;
                           QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on stock s
      (cost=7247.47..16539.30 rows=308922 width=4)
      (actual time=30.905..82.448 rows=309201 loops=1)
   Recheck Cond: ((nombre > 12) AND (annee > 1960))
   Heap Blocks: exact=3351
   ->  Bitmap Index Scan on idx_stock_nombre_annee
            (cost=0.00..7170.23 rows=308922 width=0)
            (actual time=30.340..30.340 rows=309201 loops=1)
         Index Cond: ((nombre > 12) AND (annee > 1960))
 Planning time: 0.092 ms
 Execution time: 93.748 ms
(7 lignes)
```

Exercice 6

1. Quelle a été l'impact de la création de l'index *idx_stock_annee* ?

L'index n'a pas été utilisé.

2. Comment pourrait-on améliorer les performances de cette requête ?

Nous pouvons passer par un index fonctionnel :

```
cave=> CREATE INDEX idx_stock_annee_bisextile ON stock (annee)
       WHERE (annee%4=0) AND ((annee%100<>0) OR (annee%400=0));
CREATE INDEX
```

```
cave=> EXPLAIN ANALYZE SELECT annee,nombre
  FROM stock s WHERE (annee%4=0) AND ((annee%100<>0) OR (annee%400=0));
                          QUERY PLAN
-------------------------------------------------------------------
 Index Scan using idx_stock_annee_bisextile on stock s
    (cost=0.42..3327.68 rows=4287 width=8)
    (actual time=0.020..70.139 rows=219937 loops=1)
 Planning time: 0.117 ms
 Execution time: 82.230 ms
(3 lignes)
```

Exercice 7 Voici la requête demandée :

```
SELECT r.libelle, SUM(nombre)
  FROM stock s
  INNER JOIN vin v ON s.vin_id=v.id
  INNER JOIN appellation a ON v.appellation_id=a.id
  INNER JOIN region r ON a.region_id=r.id
  INNER JOIN recoltant re ON v.recoltant_id=re.id
  WHERE s.nombre<10
    AND re.adresse LIKE '%33%'
    AND a.libelle LIKE '%eau%'
  GROUP BY r.libelle ORDER BY SUM(nombre);
```

Son plan d'exécution est le suivant :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
----------
 Sort  (cost=1091.43..1091.48 rows=19 width=40)
  Sort Key: (sum(s.nombre))
  ->  HashAggregate  (cost=1090.84..1091.03 rows=19 width=40)
    Group Key: r.libelle
    ->  Nested Loop  (cost=10.43..1068.95 rows=4378 width=36)
       ->  Hash Join  (cost=10.00..132.55 rows=106 width=36)
          Hash Cond: (a.region_id = r.id)
          ->  Hash Join  (cost=8.58..129.66 rows=106 width=8)
            Hash Cond: (v.appellation_id = a.id)
            ->  Hash Join  (cost=1.10..117.88 rows=867 width=8)
               Hash Cond: (v.recoltant_id = re.id)
             ->  Seq Scan on vin v  (cost=0.00..93.71 rows=6071 width=12)
```

```
                ->  Hash  (cost=1.09..1.09 rows=1 width=4)
              ->  Seq Scan on recoltant re  (cost=0.00..1.09 rows=1 width=4)
                    Filter: (adresse ~~ '%33%'::text)
              ->  Hash  (cost=6.99..6.99 rows=39 width=8)
            ->  Seq Scan on appellation a  (cost=0.00..6.99 rows=39 width=8)
                  Filter: (libelle ~~ '%eau%'::text)
            ->  Hash  (cost=1.19..1.19 rows=19 width=36)
             ->  Seq Scan on region r  (cost=0.00..1.19 rows=19 width=36)
         ->  Index Scan using stock_pkey on stock s
               (cost=0.42..8.42 rows=41 width=8)
            Index Cond: (vin_id = v.id)
            Filter: (nombre < 10)
(23 lignes)
```

Travaux Pratiques 4

- Optimisation avec PostgreSQL

Rappel

Durant ces travaux pratiques, nous allons utiliser la machine virtuelle du TP 1 pour héberger notre serveur de base de données PostgreSQL.

Effectuez les manipulations nécessaires pour réaliser les actions listées dans la section *Énoncés*.

Vous pouvez vous aider du cours, de l'annexe de ce TP, des derniers TP, ainsi que de l'aide en ligne ou des pages de manuels (`man`).

Énoncés

Utilisation de EXPLAIN et indexation Ces exercices sont à effectuer sur la base `cave` nouvellement restaurée dans votre instance en version 16.

Indexation simple

1. Écrire la requête permettant de sélectionner le nombre de bouteilles en stock de l'année 2012. Quel est son plan d'exécution ?
2. Rajoutez un index pour que la requête soit plus rapide.

Index partiels

1. Créer un index se limitant aux vins avec moins de 10 bouteilles en stock.
2. Exécuter une requête pour rechercher les vins possédant un stock inférieur à 7 bouteilles. Que remarquez-vous ?
3. Exécuter une requête pour rechercher les vins possédant un stock inférieur à 11 bouteilles. Que remarquez-vous ?

Indexation multi-colonnes

1. Créez l'index optimum pour cette requête :

```
SELECT * FROM stock
WHERE vin_id IN (SELECT vin_id FROM stock TABLESAMPLE BERNOULLI (0.01))
  AND annee BETWEEN 2004 AND 2007;
```

2. Quel est la taille sur disque de la table *stock* ainsi que pour les index sur cette table ?

Recherche de motif texte

1. Affichez le plan de cette requête (sur la base `cave`) :

```
SELECT * FROM appellation WHERE libelle LIKE 'Brouilly%';
```

2. Que constatez-vous ?
3. Affichez maintenant le nombre de blocs accédés par cette requête.
4. Cette requête ne passe pas par un index. Essayez de lui forcer la main.
5. L'index n'est toujours pas utilisé. L'index « par défaut » n'est pas capable de répondre à des questions sur motif.
6. Créez un index capable de réaliser ces opérations (indice : vérifier les classes d'opérateurs de votre index). Testez à nouveau le plan.
7. Réactivez `enable_seqscan`. Testez à nouveau le plan.
8. Quelle est la conclusion ?

Index trigrammes

Pour cet exercice, nous allons utiliser la base de données *magasin*

1. Effectuer une recherche textuelle dans les commentaires de la table *magasin.commandes* en sélectionnant les lignes commençant par `qui`.
2. Créer un index btree capable d'aider à exécuter la requête.
3. Quels sont les gains de performances ?
4. Tenter de chercher les lignes se terminant par `qui`, puis les lignes contenant `qui`. Que remarquez vous ?
5. Tenter de chercher les lignes commençant par `qui` mais sans tenir compte de la casse ? Que remarquez vous ?
6. Créer un index `GIN` par trigramme pour des recherche textuelle sur la colonne *commentaire* de la table *magasin.commandes*.
7. Quel est la taille sur disque de la table *magasin.commandes* ainsi que celle des index que l'on vient de créer ?
8. Relancer toutes les requêtes précédentes. Que remarquez-vous ?

Optimisation de requête Soit la requête suivante :

```
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)+1970 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;
```

1. Exécutez la requête et étudiez son plan d'exécution. 2 Utilisez le site explain.depesz.com pour clarifier le plan d'exécution.
2. Sans créer de nouvel index, optimisez cette requête pour diviser son temps d'exécution par 4 (3 optimisations possibles).

Indices :

- une jointure avec une sous-requête peut-être déplacée,
- joindre de façon externe n'est pas utile dans une relation 1-1,
- les opérateurs d'index ont un type.

Machine Virtuelle

Si vous ne l'avez pas encore fait, la machine virtuelle du TP 1 doit-être déployée.
Se référer à ce TP pour la déployer.

Pour lancer votre machine virtuelle, lancer la commande suivante dans votre terminal :

```
vboxmanage startvm TP_bdd --type headless
```

A la fin de votre TP, n'oubliez pas de l'éteindre, elle restera sinon allumée et accessible à quiconque se connecte sur la machine que vous avez utilisé. Vous pouvez éteindre votre machine virtuelle avec la commande :

```
vboxmanage controlvm TP_bdd poweroff
```

Pour se connecter à votre machine virtuelle en ssh, utilisez la commande suivante :

```
ssh -p 2222 tp@127.0.0.1
```

Les mots de passes sont les mêmes que les logins (*tp* et *login*).

```
sudo -iu postgres
```

Annexe : index dans Postgresql

De nombreuses fonctionnalités d'indexation sont disponibles dans PostgreSQL :

- 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 classique, créé comme ceci :

```
CREATE INDEX mon_index ON TABLE t1(a) ;
```

est l'index le plus fréquemment utilisé. C'est un index « BTree », c'est-à-dire un index stocké sous forme d'arbre balancé.

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 (égalité, 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.

Index Multi-Colonnes Un index peut référencer plus d'une colonne :

- CREATE INDEX idx ON ma_table (col1,col2,col3)
- Index trié sur le n-uplet (col1,col2,col3)
- Accès direct à n'importe quelle valeur de
  - (col1,col2,col3)
  - (col1,col2)
  - (col1)

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

Index partiel

- Un index partiel n'indexe qu'une partie des données d'une table, en précisant une clause WHERE à la création de l'index :

```
CREATE INDEX idx_partiel ON trapsnmp (date_reception)
WHERE est_acquitte=false;
```

- Beaucoup plus petit que l'index complet.
- Souvent dédié à une requête précise :

```
SELECT * FROM trapsnmp WHERE est_acquitte=false
ORDER BY date_reception
```

- La clause WHERE ne porte pas forcément sur la colonne indexée, c'est même souvent plus intéressant de la faire porter sur une autre colonne.

Covering Indexes Les *Covering Indexes* (on trouve parfois « index couvrants » dans la littérature française) :

- Répondent à la clause WHERE
- ET contiennent toutes les colonnes demandées par la requête
- `SELECT col1,col2 FROM t1 WHERE col1>12`
- `CREATE INDEX idx1 on T1 (col1,col2)`
- Pas de visite de la table (donc peu d'accès aléatoires, l'index étant à peu près trié physiquement)

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%'

Index GIN GIN : **G**eneralized **I**nverted i**N**dex

- 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

Index GiST GiST : **G**enera**l**i**z**ed **S**earch **T**ree

- 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

Contraintes d'Exclusion avec les Index GiST Contrainte d'exclusion : Une extension du concept d'unicité

- Unicité : `n-uplet1 = n-uplet2` interdit dans une table
- Contrainte d'exclusion : `n-uplet1 op n-uplet2` interdit dans une table
- op est n'importe quel opérateur indexable par GiST

```
CREATE TABLE circles
    ( c circle,
      EXCLUDE USING gist (c WITH &&));
```

Index BRIN BRIN : **B**lock **R**ange **IN**dex

- 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)—–

Index Hash

- Journalisés uniquement depuis la version 10
- Moins performants que les Btree
- Ne gèrent que les égalités, pas < et >
- Mais plus compacts
- Ne pas utiliser avant la version 10

Utilisation d'index Index inutilisé :

- L'optimiseur pense qu'il n'est pas rentable
  - Il a le plus souvent raison
  - S'il se trompe : statistiques ? bug ?
- La requête n'est pas compatible
  - Clause WHERE avec fonction ?
  - Cast ?
- C'est souvent tout à fait normal

Solutions du TP 4

Utilisation de EXPLAIN et indexation **Indexation simple**

1. Écrire la requête permettant de sélectionner le nombre de bouteilles en stock de l'année 2012. Quel est son plan d'exécution ?

```
cave=> EXPLAIN ANALYZE SELECT count(*) FROM stock WHERE annee=2012;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=10144.90..10144.91 rows=1 width=8)
                     (actual time=62.750..66.031 rows=1 loops=1)
   ->  Gather  (cost=10144.69..10144.90 rows=2 width=8)
               (actual time=62.586..66.005 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate
                 (cost=9144.69..9144.70 rows=1 width=8)
                 (actual time=58.837..58.845 rows=1 loops=3)
```

```
               ->  Parallel Seq Scan on stock
                        (cost=0.00..9140.20 rows=1794 width=0)
                        (actual time=8.417..47.055 rows=5621 loops=3)
                     Filter: (annee = 2012)
                     Rows Removed by Filter: 281432
 Planning time: 0.126 ms
 Execution time: 69.812 ms
(10 lignes)
```

2. Rajoutez un index pour que la requête soit plus rapide.

```
cave=> CREATE INDEX idx_stock_annee ON stock(annee);
CREATE INDEX
cave=> EXPLAIN ANALYZE SELECT count(*) FROM stock WHERE annee=2012;
                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=10144.90..10144.91 rows=1 width=8)
            (actual time=67.279..70.598 rows=1 loops=1)
   ->  Bitmap Heap Scan on stock
            (cost=10144.69..10144.90 rows=2 width=8)
            (actual time=67.124..70.574 rows=3 loops=1)
         Recheck Cond: (annee = 2012)
         Heap Blocks: exact=93
         ->  Bitmap Index Scan on idx_stock_annee
                  (cost=0.00..289.17 rows=15566 width=0)
                  (actual time=3.265..3.265 rows=16894 loops=1)
               Index Cond: (annee = 2012)
 Planning time: 0.469 ms
 Execution time: 10.538 ms
(8 lignes)
```

Index partiels

1. Créer un index se limitant aux vins avec moins de 10 bouteilles en stock.

```
cave=> CREATE INDEX idx_stock_nombre_moins_10 ON stock(nombre) WHERE
 ↪  nombre<10;
```

2. Exécuter une requête pour rechercher les vins possédant un stock inférieur à 7 bouteilles. Que remarquez-vous ?

```
cave=> EXPLAIN SELECT * FROM stock WHERE nombre < 7;
                            QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on stock  (cost=660.20..5756.55 rows=35068 width=16)
```

```
   Recheck Cond: (nombre < 7)
   ->  Bitmap Index Scan on idx_stock_nombre_moins_10
         (cost=0.00..651.43 rows=35068 width=0)
       Index Cond: (nombre < 7)
(4 lignes)
```

L'index que nous venons de créer est utilisé, même si la condition n'est pas la même. L'optimiseur sait que tous les résultats sont accessibles via cet index.

2. Exécuter une requête pour rechercher les vins possédant un stock inférieur à 11 bouteilles. Que remarquez-vous ?

```
cave=> EXPLAIN SELECT * FROM stock WHERE nombre < 11;
                                QUERY PLAN
---------------------------------------------------------------------------
 ↪  ----
 Bitmap Heap Scan on stock  (cost=6056.93..14754.46 rows=323162 width=16)
   Recheck Cond: (nombre < 11)
   ->  Bitmap Index Scan on idx_stock_nombre_annee
         (cost=0.00..5976.14 rows=323162 width=0)
       Index Cond: (nombre < 11)
(4 lignes)
```

L'index que nous venons de créer n'est pas utilisé. En effet, l'optimiseur sait que tous les résultats ne pourront pas être trouvés via notre index partiel.

Indexation multi-colonnes

1. Créez l'index optimum pour cette requête :

```
SELECT * FROM stock
WHERE vin_id IN (SELECT vin_id FROM stock TABLESAMPLE BERNOULLI (0.01))
  AND annee BETWEEN 2004 AND 2007;
```

```
cave=> CREATE INDEX idx_stock_vin_id_entre_2004_2007 ON stock(vin_id)
  WHERE annee>= 2004 AND annee <= 2007;
```

2. Quel est la taille sur disque de la table *stock* ainsi que pour les index sur cette table ?

```
cave=> SELECT
  pg_size_pretty(pg_total_relation_size('stock')) total_size
  ,pg_size_pretty(pg_relation_size('stock')) table_stock
 ,pg_size_pretty(pg_relation_size('idx_stock_annee')) idx_annee
 ,pg_size_pretty(pg_relation_size('idx_stock_nombre_moins_10')) idx_annee_nb
 ,pg_size_pretty(pg_relation_size('idx_stock_vin_id_entre_2004_2007))
 ↪  idx_vin_id;
```

```
 total_size | table_stock | idx_annee | idx_annee_nb | idx_vin_id
------------+-------------+-----------+--------------+------------
 106 MB     | 36 MB       | 18 MB     | 5528 kB      | 1496 kB
(1 ligne)
```

Recherche de motif texte

1. 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)
```

2. Que constatez-vous ?

La requête ne passe pas par un index.

3. Affichez maintenant le nombre de blocs accédés par cette requête.

```
cave=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM appellation
 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)
```

4. 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
  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.

5. 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.

6. Créez un index capable de réaliser ces opérations (indice : *Index Operator Classes*). 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).

```
cave=> 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
  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.

7. Réactivez `enable_seqscan`. Testez à nouveau le plan.

```
cave=> RESET enable_seqscan;
RESET
cave=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM appellation
  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)
```

8. 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

1. Effectuer une recherche textuelle dans les commentaires de la table *magasin.commandes* en sélectionnant les lignes commençant par `qui`.

```
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire LIKE 'qui%';
                                                QUERY PLAN
-----------------------------------------------------------------------------------
 ↪  ----
 Gather  (cost=1000.00..21251.06 rows=60598 width=51)
         (actual time=0.481..81.811 rows=60311 loops=1)
   Workers Planned: 3
   Workers Launched: 3
   Buffers: shared hit=10414
   ->  Parallel Seq Scan on commandes
            (cost=0.00..14191.26 rows=19548 width=51)
            (actual time=0.118..67.941 rows=15078 loops=4)
         Filter: (commentaire ~~ 'qui%'::text)
         Rows Removed by Filter: 234922
         Buffers: shared hit=10195
 Planning time: 0.318 ms
 Execution time: 91.008 ms
(10 lignes)
```

2. Créer un index btree capable d'aider à exécuter la requête.

```
magasin=> CREATE INDEX idx_commandes_commentaire_btree
    ON magasin.commandes (commentaire text_pattern_ops);
CREATE INDEX
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire LIKE 'qui%';
                                                QUERY PLAN
-----------------------------------------------------------------------------------
 ↪  ----
 Bitmap Heap Scan on commandes  (cost=1730.50..12528.17 rows=60598 width=51)
                               (actual time=17.027..45.687 rows=60311 loops=1)
   Filter: (commentaire ~~ 'qui%'::text)
   Heap Blocks: exact=10135
   Buffers: shared hit=10478
   ->  Bitmap Index Scan on idx_commandes_commentaire_btree
            (cost=0.00..1715.36 rows=51093 width=0)
            (actual time=15.130..15.130 rows=60311 loops=1)
         Index Cond: ((commentaire ~>=~ 'qui'::text)
                      AND (commentaire ~<~ 'quj'::text))
         Buffers: shared hit=343
 Planning time: 0.232 ms
 Execution time: 48.689 ms
(9 lignes)
```

3. Quels sont les gains de performances ?

Le temps d'exécution est divisé par 2.

4. Tenter de chercher les lignes se terminant par `qui`, puis les lignes contenant `qui`. Que remarquez vous ?

```
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire LIKE '%qui';
                                        QUERY PLAN
-------------------------------------------------------------------------------
 ↪  ----
 Gather  (cost=1000.00..20241.06 rows=50498 width=51)
         (actual time=0.534..103.126 rows=27866 loops=1)
   Workers Planned: 3
   Workers Launched: 3
   Buffers: shared hit=10414
   ->  Parallel Seq Scan on commandes
           (cost=0.00..14191.26 rows=16290 width=51)
           (actual time=0.113..91.170 rows=6966 loops=4)
         Filter: (commentaire ~~ '%qui'::text)
         Rows Removed by Filter: 243034
         Buffers: shared hit=10195
 Planning time: 0.272 ms
 Execution time: 107.653 ms
(10 lignes)

magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire LIKE '%qui%';
                                        QUERY PLAN
-------------------------------------------------------------------------------
 ↪  ----
 Seq Scan on commandes  (cost=0.00..22659.00 rows=323189 width=51)
                        (actual time=0.028..308.985 rows=245799 loops=1)
   Filter: (commentaire ~~ '%qui%'::text)
   Rows Removed by Filter: 754201
   Buffers: shared hit=10159
 Planning time: 0.198 ms
 Execution time: 324.977 ms
(6 lignes)
```

L'index construit précédemment n'est pas utilisé.

5. Tenter de chercher les lignes commençant par `qui` mais sans tenir compte de la casse ? Que

remarquez vous ?

```
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire ILIKE 'qui%';
                                        QUERY PLAN
-------------------------------------------------------------------------------
 ↪  ----
 Gather  (cost=1000.00..21251.06 rows=60598 width=51)
         (actual time=0.648..385.081 rows=60311 loops=1)
   Workers Planned: 3
   Workers Launched: 3
   Buffers: shared hit=10414
   ->  Parallel Seq Scan on commandes
            (cost=0.00..14191.26 rows=19548 width=51)
            (actual time=0.334..370.751 rows=15078 loops=4)
         Filter: (commentaire ~~* 'qui%'::text)
         Rows Removed by Filter: 234922
         Buffers: shared hit=10195
 Planning time: 0.714 ms
 Execution time: 393.533 ms
(10 lignes)
```

L'index construit précédemment n'est pas utilisé.

6. Créer un index GIN par trigramme pour des recherche textuelle sur la colonne *commentaire* de la table *magasin.commandes*.

```
magasin=> CREATE INDEX idx_commandes_commentaire_gin ON magasin.commandes
   USING GIN (commentaire gin_trgm_ops);
CREATE INDEX
```

7. Quel est la taille sur disque de la table *magasin.commandes* ainsi que celle des index que l'on vient de créer ?

```
magasin=> SELECT
  pg_size_pretty(pg_total_relation_size('magasin.commandes')) total_size
  ,pg_size_pretty(pg_relation_size('magasin.commandes')) table_commandes

 ↪  ,pg_size_pretty(pg_relation_size('magasin.idx_commandes_commentaire_btree'))
 ↪  idx_btree
 ,pg_size_pretty(pg_relation_size('magasin.idx_commandes_commentaire_gin'))
 ↪  idx_gin;
 total_size | table_commandes | idx_btree | idx_gin
------------+-----------------+-----------+---------
```

```
 208 MB     | 79 MB           | 46 MB     | 40 MB
(1 ligne)
```

8. Relancer toutes les requêtes précédentes. Que remarquez-vous ?

```
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
   WHERE commentaire LIKE '%qui';
                                QUERY PLAN
-------------------------------------------------------------------------------
 ↪  -
 Bitmap Heap Scan on commandes
      (cost=475.36..11265.59 rows=50498 width=51)
      (actual time=59.380..136.851 rows=27866 loops=1)
   Recheck Cond: (commentaire ~~ '%qui'::text)
   Rows Removed by Index Recheck: 79357
   Heap Blocks: exact=10159
   Buffers: shared hit=10247
   ->  Bitmap Index Scan on idx_commandes_commentaire_gin
            (cost=0.00..462.74 rows=50498 width=0)
            (actual time=55.715..55.715 rows=107223 loops=1)
         Index Cond: (commentaire ~~ '%qui'::text)
         Buffers: shared hit=88
 Planning time: 0.381 ms
 Execution time: 139.196 ms
(10 lignes)
```

L'optimiseur fait le choix d'utiliser l'index GIN qui est moins performant que l'index btree.

```
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire LIKE 'qui%';
                                QUERY PLAN
-------------------------------------------------------------------------------
 ↪  -
 Bitmap Heap Scan on commandes  (cost=573.63..11490.11 rows=60598 width=51)
  ↪  (actual time=56.604..125.665 rows=60311 loops=1)
   Recheck Cond: (commentaire ~~ 'qui%'::text)
   Rows Removed by Index Recheck: 162216
   Heap Blocks: exact=10159
   Buffers: shared hit=10361
   ->  Bitmap Index Scan on idx_commandes_commentaire_gin  (cost=0.00..558.48
    ↪  rows=60598 width=0) (actual time=54.436..54.436 rows=222527 loops=1)
         Index Cond: (commentaire ~~ 'qui%'::text)
         Buffers: shared hit=202
 Planning time: 0.185 ms
```

```
 Execution time: 128.731 ms
(10 lignes)
```

L'optimiseur fait le choix d'utiliser l'index GIN qui est un peu moins performant que la scan séquentiel parallelisé.

```
magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire LIKE '%qui%';
                                QUERY PLAN
-------------------------------------------------------------------------------
 ↪  -
 Bitmap Heap Scan on commandes
      (cost=2984.72..17183.58 rows=323189 width=51)
      (actual time=36.787..126.013 rows=245799 loops=1)
   Recheck Cond: (commentaire ~~ '%qui%'::text)
   Heap Blocks: exact=10159
   Buffers: shared hit=10197
   ->  Bitmap Index Scan on idx_commandes_commentaire_gin
            (cost=0.00..2903.92 rows=323189 width=0)
            (actual time=34.788..34.788 rows=245799 loops=1)
         Index Cond: (commentaire ~~ '%qui%'::text)
         Buffers: shared hit=38
 Planning time: 0.145 ms
 Execution time: 138.501 ms
(9 lignes)

magasin=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM magasin.commandes
  WHERE commentaire ILIKE 'qui%';
                                QUERY PLAN
-------------------------------------------------------------------------------
 ↪  -
 Bitmap Heap Scan on commandes
      (cost=573.63..11490.11 rows=60598 width=51)
      (actual time=103.630..374.094 rows=60311 loops=1)
   Recheck Cond: (commentaire ~~* 'qui%'::text)
   Rows Removed by Index Recheck: 162216
   Heap Blocks: exact=10159
   Buffers: shared hit=10361
   ->  Bitmap Index Scan on idx_commandes_commentaire_gin
            (cost=0.00..558.48 rows=60598 width=0)
            (actual time=100.851..100.851 rows=222527 loops=1)
         Index Cond: (commentaire ~~* 'qui%'::text)
         Buffers: shared hit=202
```

```
 Planning time: 0.645 ms
 Execution time: 377.434 ms
(10 lignes)
```

Le choix de l'optimiseur est ici valide concernant les performances des requêtes par rapport à une exécution parallelisée.

Le choix de l'optimiseur est cependant valide dans tous les cas. En effet, le coût théorique est bien inférieur dans le cas de l'index GIN par rapport à tous les autres opérations.

Optimisation de requête Soit la requête suivante :

```
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)+1970 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;
```

1. Exécutez la requête et étudiez son plan d'exécution.
2. Utilisez le site explain.depesz.com pour clarifier le plan d'exécution.
3. Sans créer de nouvel index, optimisez cette requête pour diviser son temps d'exécution par 4 (3 optimisations possibles).

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)) +
                       1970::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)
 Planning time: 4.348 ms
 Execution time: 309.337 ms
(25 lignes)
```

L'index `idx_stock_annee` n'est pas utilisé.

Interdisons de faire les scans séquentiel à l'optimiseur :

```
SET enable_seqscan TO off;
```

Nous remarquons que le plan d'exécution est encore pire :

```
 GroupAggregate  (cost=50217.50..50288.50 rows=319 width=48)
            (actual time=669.962..675.969 rows=319 loops=1)
  Group Key: ((((((round((random() * '50'::double precision))
       + '1970'::double precision)))::text || ' - '::text) || a.libelle))
  ->  Sort  (cost=50217.50..50228.27 rows=4308 width=40)
          (actual time=669.925..671.213 rows=16838 loops=1)
      Sort Key: ((((((round((random() * '50'::double precision))
                      +  '1970'::double  precision)))::text  ||  '  -
'::text) || a.libelle))
      Sort Method: quicksort  Memory: 2106kB
      ->  Hash Left Join  (cost=398.75..49957.45 rows=4308 width=40)
                     (actual time=8.359..633.349 rows=16838 loops=1)
          Hash Cond: (v.appellation_id = a.id)
          ->  Hash Join  (cost=375.83..49832.22 rows=4308 width=20)
                      (actual time=7.880..603.339 rows=16838 loops=1)
              Hash Cond: (s.vin_id = v.id)
              ->  Nested Loop
                            (cost=0.60..49397.81 rows=4308 width=20)
                      (actual time=0.263..588.333 rows=16838 loops=1)
                  Join Filter: (s.contenant_id = c.id)
                  Rows Removed by Join Filter: 33676
                  ->  Hash Join
                         (cost=0.47..49191.77 rows=4308 width=20)
                      (actual time=0.243..571.664 rows=16838 loops=1)
                      Hash Cond:
                        ((s.annee)::double precision
                        = ((round((random() * '50'::double precision))
```

```
                                + '1970'::double precision)))
                        ->  Index Scan using stock_pkey on stock s
                              (cost=0.42..44840.59 rows=861611 width=16)
                        (actual time=0.038..430.953 rows=861611 loops=1)
                        ->  Hash
                              (cost=0.03..0.03 rows=1 width=8)
                              (actual time=0.023..0.023 rows=1 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                            ->  Result
                                  (cost=0.00..0.02 rows=1 width=8)
                                 (actual time=0.016..0.017 rows=1 loops=1)
                    ->  Materialize
                           (cost=0.13..12.19 rows=3 width=8)
                          (actual time=0.000..0.000 rows=3 loops=16838)
                      ->  Index Scan using contenant_pkey on contenant c
                            (cost=0.13..12.18 rows=3 width=8)
                           (actual time=0.009..0.013 rows=3 loops=1)
                ->  Hash  (cost=299.35..299.35 rows=6071 width=8)
                        (actual time=7.589..7.589 rows=6071 loops=1)
                    Buckets: 8192  Batches: 1  Memory Usage: 302kB
                    ->  Index Scan using vin_pkey on vin v
                            (cost=0.28..299.35 rows=6071 width=8)
                         (actual time=0.020..4.501 rows=6071 loops=1)
            ->  Hash  (cost=18.93..18.93 rows=319 width=20)
                    (actual time=0.440..0.440 rows=319 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 25kB
                ->  Index Scan using appellation_pkey on appellation a
                        (cost=0.15..18.93 rows=319 width=20)
                       (actual time=0.018..0.259 rows=319 loops=1)
 Planning time: 1.701 ms
 Execution time: 676.301 ms
(28 lignes)
```

Que faire alors ?

Il convient d'autoriser à nouveau les scans séquentiel, puis, peut-être, de réécrire la requête.

Nous réécrivons la requête comme suit :

```
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)+1970 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)+1970 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)
 Planning time: 1.385 ms
 Execution time: 232.251 ms
(21 lignes)
```

Nous sommes ainsi passés de 300 ms à 250 ms : la requête est donc environ 20% 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)+1970 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
-------
  6071
```

```
select count(*)
from vin v
  left join appellation a on (v.appellation_id = a.id);
```

```
 count
-------
  6071
```

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)+1970 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)
 Planning time: 1.369 ms
 Execution time: 227.893 ms
(21 lignes)
```

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 comparé à $0, une valeur filtre appliquée à la table. Cette valeur est le résultat du calcul :

```
select round(random()*50)+1970 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 (ROUND(RANDOM()*50)+1970)::int 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)
 Planning time: 0.490 ms
 Execution time: 31.118 ms
(21 rows)
```

On constate qu'on utilise enfin l'index de `stock`. Le temps d'exécution a encore été divisé par sept.

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 80 millisecondes.
