Formation DEVPG

Développer avec PostgreSQL

24.09

Table des matières

Sur ce document

Formation	Formation DEVPG
Titre	Développer avec PostgreSQL
Révision	24.09
ISBN	978-2-38168-119-1
PDF	https://dali.bo/devpg_pdf
EPUB	https://dali.bo/devpg_epub
HTML	https://dali.bo/devpg_html
Slides	https://dali.bo/devpg_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

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

Versions de PostgreSQL couvertes

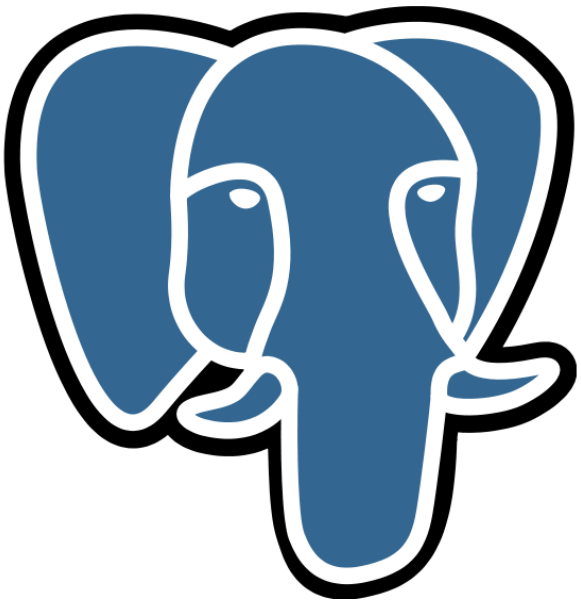
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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

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

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

1/ PostgreSQL : historique & communauté



1.1 PRÉAMBULE

- Quelle histoire !
  - parmi les plus vieux logiciels libres
  - et les plus sophistiqués
- Souvent cité comme exemple
  - qualité du code
  - indépendance des développeurs
  - réactivité de la communauté

L'histoire de PostgreSQL est longue, riche et passionnante. Au côté des projets libres Apache et Linux, PostgreSQL est l'un des plus vieux logiciels libres en activité et fait partie des SGBD les plus sophistiqués à l'heure actuelle.

Au sein des différentes communautés libres, PostgreSQL est souvent cité comme exemple à différents niveaux :

- qualité du code ;
- indépendance des développeurs et gouvernance du projet ;
- réactivité de la communauté ;
- stabilité et puissance du logiciel.

Tous ces atouts font que PostgreSQL est désormais reconnu et adopté par des milliers de grandes sociétés de par le monde.

1.1.1 Au menu

- Origines et historique du projet
- Versions et feuille de route
- Projets satellites
- Sponsors et références
- La communauté

Cette première partie est un tour d'horizon pour découvrir les multiples facettes du système de gestion de base de données libre PostgreSQL.

Les deux premières parties expliquent la genèse du projet et détaillent les différences entre les versions successives du logiciel. PostgreSQL est un des plus vieux logiciels libres ! Comprendre son histoire permet de mieux réaliser le chemin parcouru et les raisons de son succès.

Nous verrons ensuite certains projets satellites et nous listerons plusieurs utilisateurs renommés et cas d'utilisations remarquables.

Enfin, nous terminerons par une découverte de la communauté.

1.2 UN PEU D'HISTOIRE...

- La licence
- L'origine du nom
- Les origines du projet
- Les principes

1.2.1 Licence

- Licence PostgreSQL
  - libre (BSD/MIT)
  - https://www.postgresql.org/about/licence/
- Droit, sans coûts de licence, de :
  - utiliser, copier, modifier, distribuer (et même revendre)
- Reconnue par l'Open Source Initiative
- Utilisée par un grand nombre de projets de l'écosystème

PostgreSQL est distribué sous une licence spécifique, la licence PostgreSQL[1], combinant la licence BSD et la licence MIT. Elle est reconnue comme une licence libre par l'Open Source Initiative[2].

Cette licence vous donne le droit de distribuer PostgreSQL, de l'installer, de le modifier… et même de le vendre. Certaines sociétés, comme EnterpriseDB et PostgresPro, produisent leur version propriétaire de PostgreSQL de cette façon.

PostgreSQL n'est pas pour autant complètement gratuit : il peut y avoir des frais et du temps de formation, des projets de migration depuis d'autres bases, ou d'intégration des différents outils périphériques indispensables en production.

Cette licence a ensuite été reprise par de nombreux projets de la communauté : pgAdmin, pgCluu, pgstat, etc.

[1] https://www.postgresql.org/about/licence/
[2] https://opensource.org/licenses/PostgreSQL

1.2.2 PostgreSQL ?!?!

- 1985 : Michael Stonebraker recode Ingres
- post « ingres » ⇒ post~~in~~gres ⇒ postgres
- postgres ⇒ PostgreSQL

PostgreSQL a une origine universitaire.

L'origine du nom PostgreSQL remonte au système de gestion de base de données Ingres, développé à l'université de Berkeley par Michael Stonebraker. En 1985, il prend la décision de reprendre le développement à partir de zéro et nomme ce nouveau logiciel **Postgres**, comme raccourci de post-Ingres.

En 1995, avec l'ajout du support du langage SQL, Postgres fut renommé **Postgres95** puis **PostgreSQL**.

Aujourd'hui, le nom officiel est « PostgreSQL » (prononcé « post - gresse - Q - L »). Cependant, le nom « Postgres » reste accepté.

Pour aller plus loin :

- Fil de discussion sur les listes de discussion[3] ;
- Article sur le wiki officiel[4].

1.2.3 Principes fondateurs

- Sécurité des données (ACID)
- Respect des normes (ISO SQL)
- Portabilité
- Fonctionnalités intéressant le plus grand nombre
- Performances
  - si pas de péril pour les données
- Simplicité du code
- Documentation

Depuis son origine, PostgreSQL a toujours privilégié la stabilité et le respect des standards plutôt que les performances.

La sécurité des données est un point essentiel. En premier lieu, un utilisateur doit être certain qu'à partir du moment où il a exécuté l'ordre `COMMIT` d'une transaction, les données modifiées relatives à

cette transaction se trouvent bien sur disque et que même un crash ne pourra pas les faire disparaître. PostgreSQL est très attaché à ce concept et fait son possible pour forcer le système d'exploitation à ne pas conserver les données en cache, mais à les écrire sur disque dès l'arrivée d'un `COMMIT`.

L'intégrité des données, et le respect des contraintes fonctionnelles et techniques qui leur sont imposées, doivent également être garanties par le moteur à tout moment, quoi que fasse l'utilisateur. Par exemple, insérer 1000 caractères dans un champ contraint à 200 caractères maximum doit mener à une erreur explicite et non à l'insertion des 200 premiers caractères en oubliant les autres, comme cela s'est vu ailleurs. De même, un champ avec le type `date` ne contiendra jamais un 31 février, et un champ `NOT NULL` ne sera jamais vide. Tout ceci est formalisé par les propriétés (ACID[5]) que possèdent toute bonne base de données relationnelle.

Le respect des normes est un autre principe au cœur du projet. Les développeurs de PostgreSQL cherchent à coller à la norme SQL[6] le plus possible. PostgreSQL n'est pas compatible à cette norme à 100 %, aucun moteur ne l'est, mais il cherche à s'en approcher. Tout nouvel ajout d'une syntaxe ne sera accepté que si la syntaxe de la norme est ajoutée. Des extensions sont acceptées pour différentes raisons (performances, fonctionnalités en avance sur le comité de la norme, facilité de transition d'un moteur de bases de données à un autre) mais si une fonctionnalité existe dans la norme, une syntaxe différente ne peut être acceptée que si la syntaxe de la norme est elle-aussi présente.

La portabilité est importante : PostgreSQL tourne sur l'essentiel des systèmes d'exploitation : Linux (plate-forme à privilégier), macOS, les Unix propriétaires, Windows… Tout est fait pour que cela soit encore le cas dans le futur.

Ajouter des fonctionnalités est évidemment l'un des buts des développeurs de PostgreSQL. Cependant, comme il s'agit d'un projet libre, rien n'empêche un développeur de proposer une fonctionnalité, de la faire intégrer, puis de disparaître laissant aux autres la responsabilité de la corriger le cas échéant. Comme le nombre de développeurs de PostgreSQL est restreint, il est important que les fonctionnalités ajoutées soient vraiment utiles au plus grand nombre pour justifier le coût potentiel du débogage. Donc ne sont ajoutées dans PostgreSQL que ce qui est vraiment le cœur du moteur de bases de données et que ce qui sera utilisé vraiment par le plus grand nombre. Une fonctionnalité qui ne sert que une à deux personnes aura très peu de chances d'être intégrée. (Le système des extensions offre une élégante solution aux problèmes très spécifiques.)

Les performances ne viennent qu'après tout ça. En effet, rien ne sert d'avoir une modification du code qui permet de gagner énormément en performances si cela met en péril le stockage des données. Cependant, les performances de PostgreSQL sont excellentes et le moteur permet d'opérer des centaines de tables, des milliards de lignes pour plusieurs téraoctets de données, sur une seule instance, pour peu que la configuration matérielle soit correctement dimensionnée.

La simplicité du code est un point important. Le code est relu scrupuleusement par différents contributeurs pour s'assurer qu'il est facile à lire et à comprendre. En effet, cela facilitera le débogage plus tard si cela devient nécessaire.

Enfin, la documentation est là-aussi un point essentiel dans l'admission d'une nouvelle fonctionnalité. En effet, sans documentation, peu de personnes pourront connaître cette fonctionnalité. Très

[5] https://dali.bo/a2_html#ACID
[6] https://fr.wikipedia.org/wiki/Structured_Query_Language

peu sauront exactement ce qu'elle est supposée faire, et il serait donc très difficile de déduire si un problème particulier est un manque actuel de cette fonctionnalité ou un bug.

Tous ces points sont vérifiés à chaque relecture d'un patch (nouvelle fonctionnalité ou correction).

1.2.4 Origines

- Années 1970 : Michael Stonebraker développe **Ingres** à Berkeley
- 1985 : **Postgres** succède à Ingres
- 1995 : Ajout du langage SQL
- 1996 : Libération du code : Postgres devient **PostgreSQL**
- 1996 : Création du PostgreSQL Global Development Group

L'histoire de PostgreSQL remonte au système de gestion de base de données Ingres[7], développé dès 1973 à l'Université de Berkeley (Californie) par Michael Stonebraker[8].

Lorsque ce dernier décide en 1985 de recommencer le développement de zéro, il nomme le logiciel Postgres, comme raccourci de post-Ingres. Des versions commencent à être diffusées en 1989, puis commercialisées.

Postgres utilise alors un langage dérivé de QUEL[9], hérité d'Ingres, nommé POSTQUEL[10]. En 1995, lors du remplacement par le langage SQL par Andrew Yu and Jolly Chen, deux étudiants de Berkeley, Postgres est renommé Postgres95.

En 1996, Bruce Momijan et Marc Fournier convainquent l'Université de Berkeley de libérer complètement le code source. Est alors fondé le PGDG (*PostgreSQL Development Group*), entité informelle — encore aujourd'hui — regroupant l'ensemble des contributeurs. Le développement continue donc hors tutelle académique (et sans son fondateur historique Michael Stonebraker) : PostgreSQL 6.0 est publié début 1997.

Plus d'informations :

- Page associée sur le site officiel[11].

[7] https://en.wikipedia.org/wiki/Ingres_(database)
[8] https://en.wikipedia.org/wiki/Michael_Stonebraker
[9] https://en.wikipedia.org/wiki/QUEL_query_languages
[10] La trace se retrouve encore dans le nom de la librairie C pour les clients, la **libpq**.

1.2.5 Apparition de la communauté internationale

- ~ 2000: Communauté japonaise (JPUG)
- 2004 : PostgreSQLFr
- 2006 : SPI
- 2007 : Communauté italienne
- 2008 : PostgreSQL Europe et US
- 2009 : Boom des PGDay
- 2011 : Postgres Community Association of Canada
- 2017 : Community Guidelines
- …et ça continue

Les années 2000 voient l'apparition de communautés locales organisées autour d'association ou de manière informelle. Chaque communauté organise la promotion, la diffusion d'information et l'entraide à son propre niveau.

En 2000 apparaît la communauté japonaise (JPUG). Elle dispose déjà d'un grand groupe, capable de réaliser des conférences chaque année, d'éditer des livres et des magazines. Elle compte, au dernier recensement connu, plus de 3000 membres.

En 2004 naît l'association française (loi 1901) appelée PostgreSQL Fr. Cette association a pour but de fournir un cadre légal pour pouvoir participer à certains événements comme Solutions Linux, les RMLL ou d'en organiser comme le pgDay.fr (qui a déjà eu lieu à Toulouse, Nantes, Lyon, Toulon, Marseille). Elle permet aussi de récolter des fonds pour aider à la promotion de PostgreSQL.

En 2006, le PGDG intègre Software in the Public Interest, Inc.(SPI)[12], une organisation à but non lucratif chargée de collecter et redistribuer des financements. Elle a été créée à l'initiative de Debian et dispose aussi de membres comme LibreOffice.org.

Jusque là, les événements liés à PostgreSQL apparaissaient plutôt en marge de manifestations, congrès, réunions… plus généralistes. En 2008, douze ans après la création du projet, des associations d'utilisateurs apparaissent pour soutenir, promouvoir et développer PostgreSQL à l'échelle internationale. PostgreSQL UK organise une journée de conférences à Londres, PostgreSQL Fr en organise une à Toulouse. Des « sur-groupes » apparaissent aussi pour aider les groupes locaux : PGUS rassemble les différents groupes américains, plutôt organisés géographiquement, par État ou grande ville. De même, en Europe, est fondée PostgreSQL Europe, association chargée d'aider les utilisateurs de PostgreSQL souhaitant mettre en place des événements. Son principal travail est l'organisation d'un événement majeur en Europe tous les ans : pgconf.eu[13], d'abord à Paris en 2009, puis dans divers pays d'Europe jusque Milan en 2019. Cependant, elle aide aussi les communautés allemande, française et suédoise à monter leur propre événement (respectivement PGConf.DE[14], pgDay Paris[15]

[12] https://fr.wikipedia.org/wiki/Software_in_the_Public_Interest
[13] https://pgconf.eu/
[14] https://pgconf.de/
[15] https://pgday.paris/

et Nordic PGday[16]).

Dès 2010, nous dénombrons plus d'une conférence par mois consacrée uniquement à PostgreSQL dans le monde. Ce mouvement n'est pas prêt de s'arrêter :

- communauté japonaise[17] ;
- communauté francophone[18] ;
- communauté italienne[19] ;
- communauté européenne[20] ;
- communauté américaine (États-Unis)[21].

En 2011, l'association Postgres Community Association of Canada voit le jour[22]. Elle est créée par quelques membres de la *Core Team* pour gérer le nom déposé PostgreSQL, le logo, le nom de domaine sur Internet, etc.

Vu l'émergence de nombreuses communautés internationales, la communauté a décidé d'écrire quelques règles pour ces communautés. Il s'agit des *Community Guidelines*, apparues en 2017, et disponibles sur le site officiel[23].

1.2.6 Progression du code

- 1,6 millions de lignes
  - dont 1/4 de commentaires
  - le reste surtout en C
- Nombres de commit par mois :

[16] https://nordicpgday.org/
[17] https://www.postgresql.jp/
[18] https://www.postgresql.fr/
[19] https://www.itpug.org/
[20] https://www.postgresql.eu/
[21] https://www.postgresql.us/
[22] https://www.postgresql.org/message-id/4DC440BE.5040104%40agliodbs.com%3E
[23] https://www.postgresql.org/community/recognition/

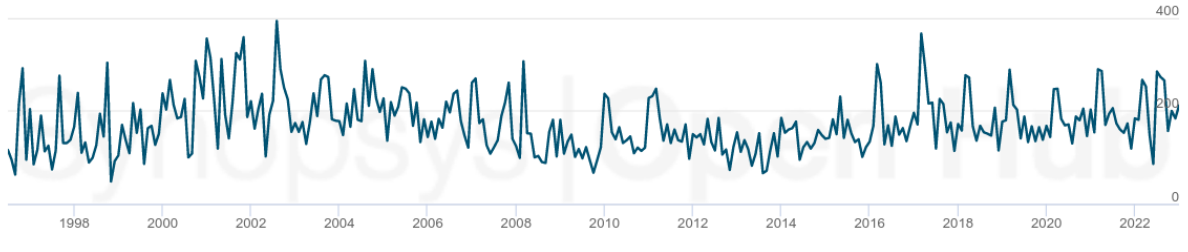


Figure 1/ .1: Évolution du nombre de commit dans le dépôt PostgreSQL

Le dépôt principal de PostgreSQL a été un dépôt CVS, passé depuis à git[24]. Il est en accès public en lecture.

Le graphe ci-dessus (source[25]) représente l'évolution du nombre de commit dans les sources de PostgreSQL. L'activité ne se dément pas. Le plus intéressant est certainement de noter que l'évolution est constante. Il n'y a pas de gros pic, ni dans un sens, ni dans l'autre.

Début 2023, PostgreSQL est composé d'1,6 millions de lignes de code, dont un quart de commentaires. Ce ratio montre que le code est très commenté, très documenté. Ceci fait qu'il est facile à lire, et donc pratique à déboguer. Et le ratio ne change pas au fil des ans. Le code est essentiellement en C, pour environ 200 développeurs actifs, à environ 200 commits par mois ces dernières années.

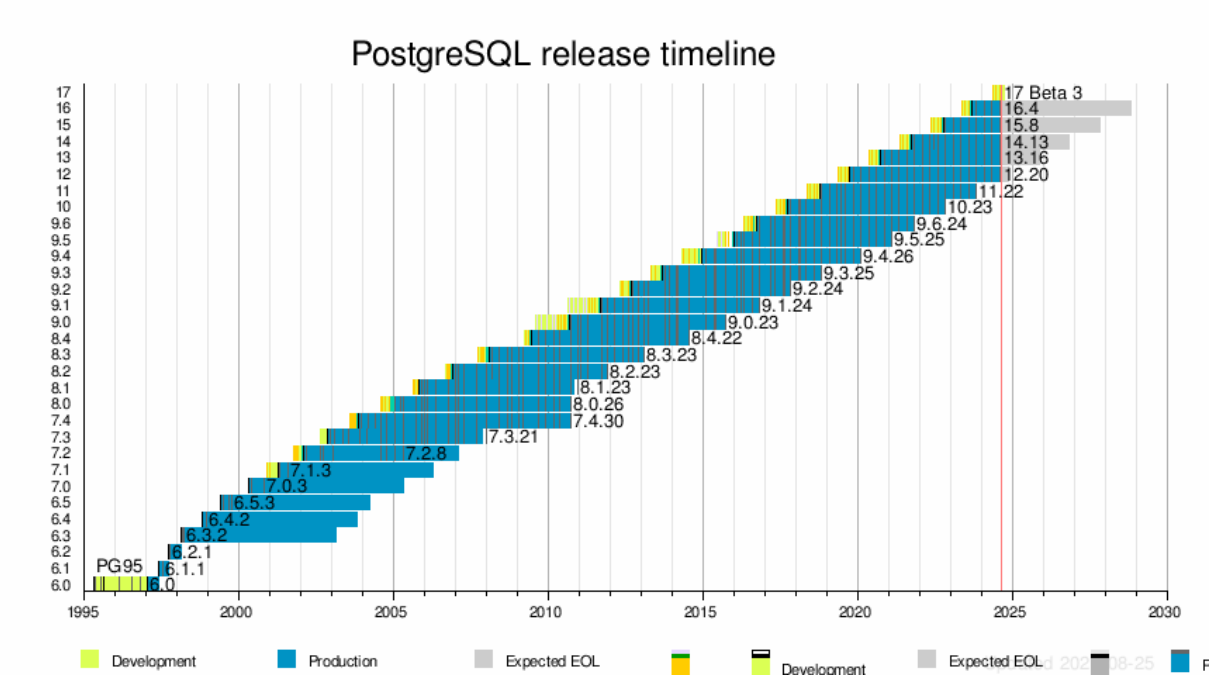
[24] https://git.postgresql.org/
[25] https://www.openhub.net/p/postgres/analyses/latest/

1.3 LES VERSIONS DE POSTGRESQL

Quelle version utiliser ?

- Historique
- Numérotation
- Mises à jour mineures et majeures
- Les versions courantes
- Quelle version en production ?
- Les forks & dérivés

1.3.1 Historique



Sources : page Wikipédia de PostgreSQL[26] et PostgreSQL Versioning Policy[27]

[26] https://en.wikipedia.org/wiki/PostgreSQL
[27] https://www.postgresql.org/support/versioning/

1.3.2 Versions & fonctionnalités

- 1996 : v6.0 -> première version publiée
- 2003 : v7.4 -> première version *réellement* stable
- 2005 : v8.0 -> arrivée sur Windows
- 2008 : v8.3 -> performances et fonctionnalités, organisation (commitfests)
- 2010 : v9.0 -> réplication physique
- 2016 : v9.6 -> parallélisation
- 2017 : v10 -> réplication logique, partitionnement déclaratif
- 2023 : v16 -> performances, fonctionnalités, administration…

La version 7.4 est la première version réellement stable. La gestion des journaux de transactions a été nettement améliorée, et de nombreuses optimisations ont été apportées au moteur.

La version 8.0 marque l'entrée tant attendue de PostgreSQL dans le marché des SGDB de haut niveau, en apportant des fonctionnalités telles que les tablespaces, les routines stockées en Java, le *Point In Time Recovery*, ainsi qu'une version native pour Windows.

La version 8.3 se focalise sur les performances et les nouvelles fonctionnalités. C'est aussi la version qui a causé un changement important dans l'organisation du développement pour encourager les contributions : gestion des commitfests, création de l'outil web associé, etc.

Les versions 9.x sont axées réplication physique. La 9.0 intègre un système de réplication asynchrone asymétrique. La version 9.1 ajoute une réplication synchrone et améliore de nombreux points sur la réplication (notamment pour la partie administration et supervision). La version 9.2 apporte la réplication en cascade. La 9.3 et la 9.4 ajoutent quelques améliorations supplémentaires. La version 9.4 intègre surtout les premières briques pour l'intégration de la réplication logique dans PostgreSQL. La version 9.6 apporte la parallélisation, ce qui était attendu par de nombreux utilisateurs.

La version 10 propose beaucoup de nouveautés, comme une amélioration nette de la parallélisation et du partitionnement (le partitionnement déclaratif complète l'ancien partitionnement par héritage), mais aussi l'ajout de la réplication logique.

Les améliorations des versions 11 à 16 sont plus incrémentales, et portent sur tous les plans. Le partitionnement déclaratif et la réplication logique sont progressivement améliorés, en performances comme en facilité de développement. Les performances s'améliorent encore grâce à la compilation *Just In Time*, la parallélisation de plus en plus d'opérations, les index couvrants, l'affinement des statistiques. La facilité d'administration s'améliore : nouvelles vues système, rôles supplémentaires pour réduire l'utilisation du superutilisateur, outillage de réplication, activation des sommes de contrôle sur une instance existante.

Il est toujours possible de télécharger les sources depuis la version 1.0 jusqu'à la version courante sur postgresql.org[28].

[28] https://www.postgresql.org/ftp/source/

1.3.3 Numérotation

- Version récentes (10+)
  - X : version majeure (10, 11, … 16)
  - X.Y : version mineure (14.8, 15.3)
- Avant la version 10 (toutes périmées !)
  - X.Y : version majeure (8.4, 9.6)
  - X.Y.Z : version mineure (9.6.24)

Une version majeure apporte de nouvelles fonctionnalités, des changements de comportement, etc. Une version majeure sort généralement tous les ans à l'automne. Une migration majeure peut se faire directement depuis n'importe quelle version précédente. Le numéro est incrémenté chaque année (version 12 en 2019, version 16 en 2023).

Une version mineure ne comporte que des corrections de bugs ou de failles de sécurité. Les publications de versions mineures sont plus fréquentes que celles de versions majeures, avec un rythme de sortie trimestriel, sauf bug majeur ou faille de sécurité. Chaque bug est corrigé dans toutes les versions stables actuellement maintenues par le projet. Le numéro d'une version mineure porte deux chiffres. Par exemple, en mai 2023 sont sorties les versions 15.3, 14.8, 13.11, 12.15 et 11.20.

Avant la version 10, les versions majeures annuelles portaient deux chiffres : 9.0 en 2010, 9.6 en 2016. Les mineures avaient un numéro de plus (par exemple 9.6.24). Cela a entraîné quelques confusions, d'où le changement de numérotation. Il va sans dire que ces versions sont totalement périmées et ne sont plus supportées, mais beaucoup continuent de fonctionner.

1.3.4 Mises à jour mineure

De M.m à M.m+n :

- En général chaque trimestre
- Et sans souci
  - *Release notes*
  - tests
  - mise à jour des binaires
  - redémarrage

Une mise à jour mineure consiste à mettre à jour vers une nouvelle version de la même branche ma-

jeure, par exemple de 14.8 à 14.9, ou de 16.0 à 16.1 (mais pas d'une version 14.x à une version 16.x). Les mises à jour des versions mineures sont cumulatives : vous pouvez mettre à jour une instance 15.0 en version 15.5 sans passer par les versions 15.1 à 15.4 intermédiaires.

En général, les mises à jour mineures se font sans souci et ne nécessitent que le remplacement des binaires et un redémarrage (et donc une courte interruption). Les fichiers de données conservent le même format. Des opérations supplémentaires sont possibles mais rarissimes. Mais comme pour toute mise à jour, il convient d'être prudent sur d'éventuels effets de bord. En particulier, il faudra lire les *Release Notes* et, si possible, effectuer les tests ailleurs qu'en production.

1.3.5 Versions courantes

- 1 version majeure par an
  - maintenue 5 ans
- Dernières mises à jour mineures
  - https://www.postgresql.org/support/versioning/ (au 25 août 2024) :
  - version 12.20
  - version 13.16
  - version 14.13
  - version 15.8
  - version 16.4
- Prochaine sortie de versions mineures prévue : 14 novembre 2024

La philosophie générale des développeurs de PostgreSQL peut se résumer ainsi :

« Notre politique se base sur la qualité, pas sur les dates de sortie. »

Toutefois, même si cette philosophie reste très présente parmi les développeurs, en pratique une version stable majeure paraît tous les ans, habituellement à l'automne. Pour ne pas sacrifier la qualité des versions, toute fonctionnalité supposée insuffisamment stable est repoussée à la version suivante. Il est déjà arrivé que la sortie de la version majeure soit repoussée à cause de bugs inacceptables.

La tendance actuelle est de garantir un support pour chaque version majeure pendant une durée minimale de 5 ans. Ainsi ne sont plus supportées les versions 10 depuis novembre 2022 et 11 depuis novembre 2023. Il n'y aura pour elles plus aucune mise à jour mineure, donc plus de correction de bug ou de faille de sécurité. Le support de la version 12 s'arrêtera en novembre 2024. Le support de la dernière version majeure, la 16, devrait durer jusqu'en 2028.

Pour plus de détails :

- Politique de versionnement[29] ;
- Dates prévues des futures versions[30].

1.3.6 Versions 9.4 à 11

- `jsonb`
- Row Level Security
- Index BRIN, bloom
- Fonctions OLAP
- Parallélisation
- SQL/MED : accès distants
- Réplication logique
- Partitionnement déclaratif
- Réduction des inconvénients de MVCC
- JIT
- Index couvrants

Ces versions ne sont plus supportées !

La version 9.4 (décembre 2014) a apporté le type `jsonb`, binaire, facilitant la manipulation des objets en JSON.

La 9.5 parue en janvier 2016 apportait notamment les index BRIN et des possibilités OLAP plus avancées que `GROUP BY`. Pour plus de détails :

- Page officielle des nouveautés de la version 9.5[31] ;
- Workshop Dalibo sur la version 9.5[32].

En 9.6, la nouvelle fonctionnalité majeure est certainement la parallélisation de certaines parties de l'exécution d'une requête. Le `VACUUM FREEZE` devient beaucoup moins gênant.

- Page officielle des nouveautés de la version 9.6[33] ;
- Workshop Dalibo sur la version 9.6[34].

[29] https://www.postgresql.org/support/versioning/
[30] https://www.postgresql.org/developer/roadmap/
[31] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.5
[32] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.5
[33] https://wiki.postgresql.org/wiki/NewIn96
[34] https://github.com/dalibo/workshops/tree/master/fr

En version 10, les fonctionnalités majeures sont l'intégration de la réplication logique et le partitionnement déclaratif, longtemps attendus, améliorés dans les versions suivantes. Sont notables aussi les tables de transition ou les améliorations sur la parallélisation.

La version 10 a aussi été l'occasion de renommer plusieurs répertoires et fonctions système, et même des outils. Attention donc si vous rencontrez des requêtes ou des scripts adaptés aux versions précédentes. Entre autres :

- le répertoire `pg_xlog` est devenu `pg_wal` ;
- le répertoire `pg_clog` est devenu `pg_xact` ;
- dans les noms de fonctions, `xlog` a été remplacé par `wal` (par exemple `pg_switch_xlog` est devenue `pg_switch_wal`) ;
- toujours dans les fonctions, `location` a été remplacé par `lsn`.

Pour plus de détails :

- Page officielle des nouveautés de la version 10[35] ;
- Workshop Dalibo sur la version 10[36].

La version 11 (octobre 2018) améliore le partitionnement de la version 10, le parallélisme, la réplication logique… et de nombreux autres points. Elle comprend aussi une première version du JIT (*Just In Time compilation*) pour accélérer les requêtes les plus lourdes en CPU, ou encore les index couvrants.

Pour plus de détails, voir notre workshop sur la version 11[37].

1.3.7 Version 12

- Octobre 2019 - Novembre 2024
- Amélioration du partitionnement déclaratif
- Amélioration des performances
  - sur la gestion des index
  - sur les CTE (option MATERIALIZED)
- Colonnes générées
- Nouvelles vues de visualisation de la progression des commandes
- Refonte de la configuration de la réplication

La version 12 est sortie le 3 octobre 2019. Elle améliore de nouveau le partitionnement et elle fait surtout un grand pas au niveau des performances et de la supervision.

[35] https://wiki.postgresql.org/wiki/New_in_postgres_10
[36] https://dali.bo/workshop10_pdf
[37] https://dali.bo/workshop11_pdf

Le fichier `recovery.conf` (pour la réplication et les restaurations physiques) disparaît. Il est maintenant intégré au fichier `postgresql.conf`. Une source fréquente de ralentissement disparaît, avec l'intégration des CTE (clauses `WITH`) dans la requête principale. Des colonnes d'une table peuvent être automatiquement générées à partir d'autres colonnes.

Pour plus de détails, voir notre workshop sur la version 12[38].

1.3.8 Version 13

- Septembre 2020 - Septembre 2025
- Améliorations :
  - partitionnement déclaratif
  - réplication logique
- Amélioration des performances :
  - index B-tree, objet statistique, tri et agrégat
- Amélioration de l'autovacuum et du VACUUM :
  - gestion complète des tables en insertion seule
  - traitement parallélisé des index lors d'un VACUUM
- Amélioration des sauvegardes :
  - génération d'un fichier manifeste, outil pg_verifybackup
- Nouvelles vues de progression de commandes :
  - pg_stat_progress_basebackup, pg_stat_progress_analyze

La version 13 est sortie le 24 septembre 2020. Elle est remplie de nombreuses petites améliorations sur différents domaines : partitionnement déclaratif, autovacuum, sauvegarde, etc. Les performances sont aussi améliorées grâce à un gros travail sur l'optimiseur, ou la réduction notable de la taille de certains index.

Pour plus de détails, voir notre workshop sur la version 13[39].

[38] https://dali.bo/workshop12_pdf
[39] https://dali.bo/workshop13_pdf

1.3.9 Version 14

- Septembre 2021 - Novembre 2026
- Nouvelles vues système & améliorations
  - `pg_stat_progress_copy`, `pg_stat_wal`, `pg_lock.waitstart`, `query_id`…
- Lecture asynchrone des tables distantes
- Paramétrage par défaut adapté aux machines plus récentes
- Améliorations diverses :
  - réplications physique et logique
  - quelques facilités de syntaxe (triggers, tableaux en PL/pgSQL)
- Performances :
  - connexions en lecture seule plus nombreuses
  - index…

La version 14 est remplie de nombreuses petites améliorations sur différents domaines listés ci-dessus.

Pour plus de détails, voir notre workshop sur la version 14[40].

1.3.10 Version 15

- Octobre 2022 - Novembre 2027
- Nombreuses améliorations incrémentales
  - dont en réplication logique
- Commande `MERGE`
- Performances :
  - `DISTINCT` parallélisable
  - pg_dump & sauvegardes, recovery, partitionnement
- Changements notables :
  - `public` n'est plus accessible en écriture à tous
  - sauvegarde PITR exclusive disparaît

[40] https://dali.bo/workshop14_pdf

La version 15 est également une mise à jour sans grande nouveauté fracassante, mais contenant de très nombreuses améliorations et optimisations sur de nombreux plans, comme par exemple la commande `MERGE` ou l'accélération du *recovery* sur une reprise de restauration.

Signalons deux changements de comportement importants : pour renforcer la sécurité, le schéma `public` n'est plus accessible en écriture par défaut à tous les utilisateurs ; et la sauvegarde physique en mode exclusif n'est plus disponible.

Pour plus de détails, voir notre workshop sur la version 15[41].

1.3.11 Version 16

- Septembre 2023 - Novembre 2028
- Plus de tris incrémentaux (`DISTINCT`…)
- Réplication logique depuis un secondaire
- Expressions régulières dans `pg_hba.conf`
- Vues systèmes améliorées : `pg_stat_io`…
- Compression lz4 ou zstd pour pg_dump
- Optimisation et améliorations diverses (parallélisation…)

La version 16 est parue le 14 septembre 2023, et est considérée comme bonne pour la production. Là encore, les améliorations sont incrémentales.

On notera la possibilité de rajouter des expressions régulières dans `pg_hba.conf` pour faciliter la gestion des accès. En réplication logique, un abonnement peut se faire auprès d'un serveur secondaire. La réplication logique peut devenir parallélisable. `pg_dump` acquiert des algorithmes de compression plus modernes. Le travail de parallélisation de nouveaux nœuds se poursuit. Une nouvelle vue de suivi des entrées-sorties apparaît : `pg_stat_io`.

Pour plus de détails :

- workshop Dalibo sur la version 16[42] ;
- blog Dalibo[43] ;
- présentation de Magnus Hagander au PGDay UK 2023[44]

[41] https://dali.bo/workshop15_pdf
[42] https://dali.bo/workshop16_pdf
[43] https://blog.dalibo.com/2023/09/15/release_postgresql_16.html
[44] https://www.hagander.net/talks/PostgreSQL%2016.pdf

1.3.12 Petit résumé

- Versions 7.x :
  - fondations
  - durabilité
- Versions 8.x :
  - fonctionnalités
  - performances
- Versions 9.x :
  - réplication physique
  - extensibilité
- Versions 10 à 16 :
  - réplication logique
  - parallélisation
  - partitionnement
- … et la 17 approche

Si nous essayons de voir cela avec de grosses mailles, les développements des versions 7 ciblaient les fondations d'un moteur de bases de données stable et durable. Ceux des versions 8 avaient pour but de rattraper les gros acteurs du marché en fonctionnalités et en performances. Enfin, pour les versions 9, on est plutôt sur la réplication et l'extensibilité.

La version 10 se base principalement sur la parallélisation des opérations (développement mené principalement par EnterpriseDB) et la réplication logique (par 2ndQuadrant). Les versions 11 à 16 améliorent ces deux points, entre mille autres améliorations en différents points du moteur, notamment les performances et la facilité d'administration.

1.3.13 Quelle version utiliser en production ?

- 12 et inférieures
  - **Danger !**
  - planifier une migration urgemment !
- 12, 13, 14, 15, 16
  - mises à jour mineures uniquement
- 16
  - nouvelles installations
  - nouveaux développements (ou future 17 ?)
- Tableau comparatif des versions[45]

La version 12 ne sera plus supportée dès novembre 2024.

Si vous avez une version 12 ou inférieure, planifiez le plus rapidement possible une migration vers une version plus récente, comme la 16. (Si votre migration va être longue, vous pouvez même envisager la 17.)

La 10 et la 11 ne sont plus maintenues. Elles fonctionneront toujours aussi bien, mais il n'y aura plus de correction de bug, y compris pour les failles de sécurité ! Si vous utilisez ces versions ou des versions antérieures, il est impératif d'étudier une migration de version dès que possible.

Les versions 13 à 16 sont celles recommandées pour une production. Le plus important est d'appliquer les mises à jour correctives.

La version 16 est conseillée pour les nouvelles installations en production, et les nouveaux développements.

La 17.0 est annoncée pour l'automne 2024. Par expérience, quand une version x.0 paraît à l'automne, elle est généralement stable. Nombre de DBA préfèrent prudemment attendre les premières mises à jour mineures (en novembre généralement) pour la mise en production. Elle est envisageable pour un nouveau projet qui pourrait durer quelques mois. Cette prudence est à mettre en balance avec l'intérêt pour les nouvelles fonctionnalités.

Pour plus de détails, voir le tableau comparatif des versions[46].

[46] https://www.postgresql.org/about/featurematrix

1.3.14 Versions dérivées / Forks

Entre de nombreux autres :

- Compatibilité Oracle :
  - EnterpriseDB
- Data warehouse :
  - Greenplum, Netezza
- Forks :
  - Amazon RedShift, Aurora, Neon…
  - attenton : support, extensions…
- Extensions :
  - Citus
  - timescaledb
- Packages avec des outils & support
- Bases compatibles

Il existe de nombreuses versions dérivées de PostgreSQL. Elles sont en général destinées à des cas d'utilisation très spécifiques et offrent des fonctionnalités non proposées par la version communautaire. Leur code est souvent fermé et nécessite l'acquisition d'une licence payante. La licence de PostgreSQL permet cela, et le phénomène existait déjà dès les années 1990 avec divers produits commerciaux comme Illustra.

Modifier le code de PostgreSQL a plusieurs conséquences négatives. Certaines fonctionnalités de PostgreSQL peuvent être désactivées. Il est donc difficile de savoir ce qui est réellement utilisable. De plus, chaque nouvelle version mineure demande une adaptation de leur ajout de code. Chaque nouvelle version majeure demande une adaptation encore plus importante de leur code. C'est un énorme travail, qui n'apporte généralement pas suffisamment de plus-value à la société éditrice pour qu'elle le réalise. La seule société qui le fait de façon complète est EnterpriseDB, qui arrive à proposer des mises à jour régulièrement. Par contre, si on revient sur l'exemple de Greenplum, ils sont restés bloqués pendant un bon moment sur la version 8.0. Ils ont cherché à corriger cela. Fin 2021, Greenplum 6.8 est au niveau de la version 9.4[47], version considérée alors comme obsolète par la communauté depuis plus de deux ans. En janvier 2023, Greenplum 7.0 bêta n'est toujours parvenu qu'au niveau de PostgreSQL 12.12…

Rien ne dit non plus que la société ne va pas abandonner son fork. Par exemple, il a existé quelques forks créés lorsque PostgreSQL n'était pas disponible en natif sous Windows : ces forks ont majoritairement disparu lors de l'arrivée de la version 8.0, qui supportait officiellement Windows dans la version communautaire.

[47] https://web.archive.org/web/20211018012643/https://docs.greenplum.org/6-8/security-guide/topics/preface.html

Il y a eu aussi quelques forks créés pour gérer la réplication. Là aussi, la majorité de ces forks ont été abandonnés (et leurs clients avec) quand PostgreSQL a commencé à proposer de la réplication en version 9.0.

Il faut donc bien comprendre qu'à partir du moment où un utilisateur choisit une version dérivée, il dépend fortement (voire uniquement) de la bonne volonté de la société éditrice pour maintenir son produit, le mettre à jour avec les dernières corrections et les dernières nouveautés de la version communautaire, et le rendre compatible avec la myriadr d'extensions existantes. Pour éviter ce problème, certaines sociétés ont décidé de transformer leur fork en une extension. C'est beaucoup plus simple à maintenir et n'enferme pas leurs utilisateurs. C'est le cas par exemple de citusdata (racheté par Microsoft) pour son extension de *sharding* ; ou encore de TimescaleDB, avec leur extension spécialisée dans les séries temporelles.

Dans les exemples de fork dédiés aux entrepôts de données, les plus connus historiquement sont Greenplum, de Pivotal (racheté par WMWare), et Netezza (racheté par IBM). Autant Greenplum tente de se raccrocher au PostgreSQL communautaire toutes les quelques années, autant ce n'est pas le cas de Netezza, optimisé pour du matériel dédié, et qui a forké de PostgreSQL 7.2.

Amazon, avec notamment les versions Redshift[48] ou Aurora, a la particularité de modifier profondément PostgreSQL pour l'adapter à son infrastructure, mais ne diffuse pas ses modifications. Même si certaines incompatibilités sont listées, il est très difficile de savoir où ils en sont et l'impact qu'a leurs modifications.

Neon.tech[49] est un autre *fork* ayant réécrit la couche de stockage et permettant de dupliquer des bases rapidement, notamment à l'usage des développeurs.

EDB Postgres Advanced Server est une distribution PostgreSQL d'EnterpriseDB. Elle permet de faciliter la migration depuis Oracle. Son code est propriétaire et soumis à une licence payante. Certaines fonctionnalités finissent par atterrir dans le code communautaire (une fois qu'EnterpriseDB le souhaite et que la communauté a validé l'intérêt de cette fonctionnalité et sa possible intégration).

Supabase est un exemple de société intégrant PostgreSQL dans une plateforme plus vaste pour du développement web.

BDR, anciennement de 2nd Quadrant, maintenant EnterpriseDB, est un *fork* visant à fournir une version multimaître de PostgreSQL, mais le code a été refermé dans les dernières versions. Il est très difficile de savoir où ils en sont. Son utilisation implique de prendre le support chez eux.

La société russe Postgres Pro, tout comme EnterpriseDB, propose diverses fonctionnalités dans sa version propre, tout en proposant souvent leur inclusion dans la version communautaire — ce qui n'est pas automatique.

Face au leadership de PostgreSQL, une tendance récente pour certaines bases de données est de se revendiquer « compatibles PostgreSQL », par exemple YugabyteDB. Certains éditeurs de solutions de bases de données distribuées propriétaires disent que leur produit peut remplacer PostgreSQL sans modification de code côté application. Il convient de rester critique et prudent face à cette affirmation, car ces produits n'ont parfois rien à voir avec PosgreSQL. Leurs évolutions n'intégreront sans doute pas la version communautaire.

[48] https://www.stitchdata.com/blog/how-redshift-differs-from-postgresql/
[49] https://neon.tech/

(Cet historique provient en partie de la liste exhaustive des « forks »[50], ainsi de que cette conférence de Josh Berkus[51] de 2009 et des références en bibliographie.)

Sauf cas très précis, il est recommandé d'utiliser la version officielle, libre et gratuite. Vous savez exactement ce qu'elle propose et vous choisissez librement vos partenaires (pour les formations, pour le support, pour les audits, etc).

[50] https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases
[51] https://www.slideshare.net/pgconf/elephant-roads-a-tour-of-postgres-forks

1.4 QUELQUES PROJETS SATELLITES

PostgreSQL n'est que le moteur ! Besoin d'outils pour :

- Administration
- Sauvegarde
- Supervision
- Migration
- SIG

PostgreSQL n'est qu'un moteur de bases de données. Quand vous l'installez, vous n'avez que ce moteur. Vous disposez de quelques outils en ligne de commande (détaillés dans nos modules « Outils graphiques et consoles » et « Tâches courantes ») mais aucun outil graphique n'est fourni.

Du fait de ce manque, certaines personnes ont décidé de développer ces outils graphiques. Ceci a abouti à une grande richesse grâce à la grande variété de projets « satellites » qui gravitent autour du projet principal.

Par choix, nous ne présenterons ici que des logiciels libres et gratuits. Pour chaque problématique, il existe aussi des solutions propriétaires. Ces solutions peuvent parfois apporter des fonctionnalités inédites. Il faut néanmoins considérer que l'offre de la communauté Open-Source répond à la plupart des besoins des utilisateurs de PostgreSQL.

1.4.1 Administration, Développement, Modélisation

Entre autres, dédiés ou pas :

- Administration :
  - pgAdmin4
  - temBoard
- Développement :
  - DBeaver
- Modélisation :
  - pgModeler

Il existe différents outils graphiques pour l'administration, le développement et la modélisation. Une liste plus exhaustive est disponible sur le wiki PostgreSQL[52].

[52] https://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools

pgAdmin4[53] est un outil d'administration dédié à PostgreSQL, qui permet aussi de requêter. (La version 3 est considérée comme périmée.)

temBoard[54] est une console d'administration plus complète. temBoard intègre de la supervision, des tableaux de bord, la gestion des sessions en temps réel, du bloat, de la configuration et l'analyse des performances.

DBeaver[55] est un outil de requêtage courant, utilisable avec de nombreuses bases de données différentes, et adapté à PostgreSQL.

Pour la modélisation, pgModeler[56] est dédié à PostgreSQL. Il permet la modélisation, la rétro-ingénierie d'un schéma existant, la génération de scripts de migration.

1.4.2 Sauvegardes

- Export logique :
  - pg_back : https://github.com/orgrim/pg_back/
- Sauvegarde physique (PITR) :
  - pgBackRest : https://pgbackrest.org/
  - barman : https://www.pgbarman.org/

Les outils listés ci-dessus sont les outils principaux et que nous recommandons pour la réalisation des sauvegardes et la gestion de leur rétention.

Ils se basent sur les outils standards de PostgreSQL de sauvegarde physique ou logique.

Liens :

- pg_back[57]
- pgBackRest[58]
- barman[59]

Il existe plusieurs outils propriétaires, notamment pour une sauvegarde par snapshot au niveau de la baie.

[53] https://www.pgadmin.org/
[54] https://labs.dalibo.com/temboard
[55] https://dbeaver.io/
[56] https://pgmodeler.io/
[57] https://github.com/orgrim/pg_back/releases
[58] https://pgbackrest.org/
[59] https://www.pgbarman.org

1.4.3 Supervision

- Nagios/Icinga2 :
  - check_pgactivity
  - check_postgres
- Prometheus : postgres_exporter
- PoWA

Pour ne citer que quelques projets libres et matures :

check_pgactivity[60] est une sonde Nagios pouvant récupérer un grand nombre de statistiques d'activités renseignées par PostgreSQL. Il faut de ce fait un serveur Nagios (ou un de ses nombreux forks ou surcharges) pour gérer les alertes et les graphes. Il existe aussi check_postgres[61].

postgres_exporter[62] est l'exporteur de métriques pour Prometheus.

PoWA[63] est composé d'une extension qui historise les statistiques récupérées par l'extension `pg_stat_statements` et d'une application web qui permet de récupérer les requêtes et leur statistiques facilement.

1.4.4 Audit

- pgBadger
  - https://pgbadger.darold.net/
- pgCluu
  - https://pgcluu.darold.net/

pgBadger[64] est l'outil de base pour les analyses (à posteriori) des traces de PostgreSQL, dont notamment les requêtes.

pgCluu[65] permet une analyse du système et de PostgreSQL.

[60] https://github.com/OPMDG/check_pgactivity
[61] https://bucardo.org/check_postgres/
[62] https://github.com/prometheus-community/postgres_exporter
[63] https://powa.readthedocs.io/en/latest/
[64] https://pgbadger.darold.net/
[65] https://pgcluu.darold.net/

1.4.5 Migration

- Oracle, MySQL, SQL Server : Ora2Pg
  - https://ora2pg.darold.net/
- MySQL, SQL Server : pgloader
  - https://pgloader.io/

Il existe de nombreux outils pour migrer vers PostgreSQL une base de données utilisant un autre moteur. Ce qui pose le plus problème en pratique est le code applicatif (procédures stockées).

Plusieurs outils libres ou propriétaires, plus ou moins efficaces, existent - ou ont existé. Citons les plus importants :

Ora2Pg[66], de Gilles Darold, convertit le schéma de données, migre les données, et tente même de convertir le code PL/SQL en PL/pgSQL. Il convertit aussi des bases MySQL ou SQL Server.

pgloader[67], de Dimitri Fontaine, permet de migrer depuis MySQL, SQLite ou MS SQL Server, et importe les fichiers CSV, DBF (dBase) ou IXF (fichiers d'échange indépendants de la base).

Ces outils sont libres. Des sociétés vivant de la prestation de service autour de la migration ont également souvent développé les leurs.

1.4.6 PostGIS

[66] https://ora2pg.darold.net/
[67] https://pgloader.io/

- Projet indépendant, GPL, https://postgis.net/
- Module spatial pour PostgreSQL
  - Extension pour types géométriques/géographiques & outils
  - La référence des bases de données spatiales
  - « quelles sont les routes qui coupent le Rhône ? »
  - « quelles sont les villes adjacentes à Toulouse ? »
  - « quels sont les restaurants situés à moins de 3 km de la Nationale 12 ? »

PostGIS ajoute le support d'objets géographiques à PostgreSQL. C'est un projet totalement indépendant développé par la société Refractions Research sous licence GPL, soutenu par une communauté active, utilisée par des spécialistes du domaine géospatial (IGN, BRGM, AirBNB, Mappy, Openstreetmap, Agence de l'eau…), mais qui peut convenir pour des projets plus modestes.

Techniquement, c'est une extension transformant PostgreSQL en serveur de données spatiales, qui sera utilisé par un Système d'Information Géographique (SIG), tout comme le SDE de la société ESRI ou bien l'extension Oracle Spatial. PostGIS se conforme aux directives du consortium OpenGIS et a été certifié par cet organisme comme tel, ce qui est la garantie du respect des standards par PostGIS.

PostGIS permet d'écrire des requêtes de ce type :

```
SELECT restaurants.geom, restaurants.name FROM restaurants
  WHERE EXISTS (SELECT 1 FROM routes
                  WHERE ST_DWithin(restaurants.geom, routes.geom, 3000)
                AND route.name = 'Nationale 12')
```

PostGIS fournit les fonctions d'indexation qui permettent d'accéder rapidement aux objets géométriques, au moyen d'index GiST. La requête ci-dessus n'a évidemment pas besoin de parcourir tous les restaurants à la recherche de ceux correspondant aux critères de recherche.

La liste des fonctionnalités comprend le support des coordonnées géodésiques ; des projections et reprojections dans divers systèmes de coordonnées locaux (Lambert93 en France par exemple) ; des opérateurs d'analyse géométrique (enveloppe convexe, simplification…)

PostGIS est intégré aux principaux serveurs de carte, ETL, et outils de manipulation.

La version 3.0 apporte la gestion du parallélisme, un meilleur support de l'indexation SP-GiST et GiST, ainsi qu'un meilleur support du type GeoJSON.

1.5 SPONSORS & RÉFÉRENCES

- https://www.postgresql.org/about/sponsors
- Références :
  - françaises
  - et internationales

Au-delà de ses qualités, PostgreSQL suscite toujours les mêmes questions récurrentes :

- qui finance les développements ? (et pourquoi ?)
- qui utilise PostgreSQL ?

1.5.1 Sponsors principaux

- Sociétés se consacrant à PostgreSQL :
  - Crunchy Data (USA) : Tom Lane, Stephen Frost, Joe Conway…
  - EnterpriseDB (USA) : Bruce Momjian, Robert Haas, Dave Page…
  - 2nd Quadrant (R.U.) : Simon Riggs, Peter Eisentraut…
    - racheté par EDB
  - PostgresPro (Russie) : Oleg Bartunov, Alexander Korotkov
  - Cybertec (Autriche), Dalibo (France), Redpill Linpro (Suède), Credativ (Allemagne)…
- Sociétés vendant un fork ou une extension :
  - Citusdata (Microsoft), Pivotal (VMWare), TimescaleDB

La liste des sponsors de PostgreSQL contribuant activement au développement figure sur la liste officielle des sponsors[68]. Ce qui suit n'est qu'un aperçu.

EnterpriseDB est une société américaine qui a décidé de fournir une version de PostgreSQL propriétaire fournissant une couche de compatibilité avec Oracle. Ils emploient plusieurs développeurs importants du projet PostgreSQL (dont trois font partie de la *Core Team*), et reversent un certain nombre de leurs travaux au sein du moteur communautaire. Ils ont aussi un poids financier qui leur permet de sponsoriser la majorité des grands événements autour de PostgreSQL : PGEast et PGWest aux États-Unis, PGDay en Europe.

[68] https://www.postgresql.org/about/sponsors/

En 2020, EnterpriseDB rachète 2nd Quadrant, une société anglaise fondée par Simon Riggs, développeur PostgreSQL de longue date. 2nd Quadrant développe de nombreux outils autour de PostgreSQL comme pglogical, des versions dérivées comme Postgres-XL ou BDR, ou des outils annexes comme barman ou repmgr.

Crunchy Data offre sa propre version certifiée et finance de nombreux développements.

De nombreuses autres sociétés dédiées à PostgreSQL existent dans de nombreux pays. Parmi les sponsors officiels, nous pouvons compter Cybertec en Autriche ou Redpill Linpro en Suède. En Russie, PostgresPro maintient une version locale et reverse aussi de nombreuses contributions à la communauté.

En Europe francophone, Dalibo participe pleinement à la communauté. La société est Major Sponsor du projet PostgreSQL[69], ce qui indique un support de longue date. Elle développe et maintient plusieurs outils plébiscités par la communauté, comme autrefois Open PostgreSQL Monitoring (OPM) ou la sonde check_pgactivity[70], plus récemment la console d'administration temBoard[71], avec de nombreux autres projets en cours[72], et une participation active au développement de patchs pour PostgreSQL. Dalibo sponsorise également des événements comme les PGDay français et européens, ainsi que la communauté francophone.

Des sociétés comme Citusdata (racheté par Microsoft), Pivotal (VMWare) ou TimescaleDB proposent ou ont proposé leur version dérivée sous une forme ou une autre, mais « jouent le jeu » et participent au développement de la version communautaire, notamment en cherchant à ce que leur produit n'en diverge pas.

1.5.2 Autres sponsors

- Autres sociétés :
  - VMWare, Rackspace, Heroku, Conova, Red Hat, Microsoft
  - NTT (*streaming replication*), Fujitsu, NEC
- Cloud
  - nombreuses

Contribuent également à PostgreSQL nombre de sociétés non centrées autour des bases de données.

NTT a financé de nombreux patchs pour PostgreSQL.

Fujitsu a participé à de nombreux développements aux débuts de PostgreSQL, et emploie Amit Kapila.

[69] https://www.postgresql.org/about/sponsors/
[70] https://github.com/OPMDG/check_pgactivity
[71] https://labs.dalibo.com/temboard
[72] https://labs.dalibo.com/about

VMWare a longtemps employé le développeur finlandais Heikki Linnakangas, parti ensuite un temps chez Pivotal. VMWare emploie aussi Michael Paquier ou Julien Rouhaud.

Red Hat a longtemps employé Tom Lane à plein temps pour travailler sur PostgreSQL. Il a pu dédier une très grande partie de son temps de travail à ce projet, bien qu'il ait eu d'autres affectations au sein de Red Hat. Tom Lane a travaillé également chez SalesForce, ensuite il a rejoint Crunchy Data Solutions fin 2015.

Il y a déjà plus longtemps, Skype a offert un certain nombre d'outils très intéressants : pgBouncer (pooler de connexion), Londiste (réplication par trigger), etc. Ce sont des outils utilisés en interne et publiés sous licence BSD comme retour à la communauté. Malgré le rachat par Microsoft, certains sont encore utiles et maintenus.

Zalando est connu pour l'outil de haute disponibilité patroni.

De nombreuses sociétés liées au cloud figurent aussi parmi les sponsors, comme Conova (Autriche), Heroku ou Rackspace (États-Unis), ou les mastodontes Google, Amazon Web Services et, à nouveau, Microsoft.

1.5.3 Références

- Météo France
- IGN
- RATP, SNCF
- CNAF
- MAIF, MSA
- Le Bon Coin
- Air France-KLM
- Société Générale
- Carrefour, Leclerc, Leroy Merlin
- Instagram, Zalando, TripAdvisor
- Yandex
- CNES
- …et plein d'autres

Météo France utilise PostgreSQL depuis plus d'une décennie pour l'essentiel de ses bases, dont des instances critiques de plusieurs téraoctets (témoignage sur postgresql.fr[73]).

L'IGN utilise PostGIS et PostgreSQL depuis 2006[74].

La RATP a fait ce choix depuis 2007 également[75].

[73] https://www.postgresql.fr/temoignages/meteo_france
[74] https://www.postgresql.fr/temoignages/ign
[75] https://www.journaldunet.com/solutions/dsi/1013631-la-ratp-integre-postgresql-a-son-systeme-d-information/

La Caisse Nationale d'Allocations Familiales a remplacé ses mainframes par des instances PostgreSQL[76] dès 2010 (4 To et 1 milliard de requêtes par jour).

Instagram utilise PostgreSQL depuis le début[77].

Zalando a décrit plusieurs fois son infrastructure PostgreSQL[78] et annonçait en 2018[79] utiliser pas moins de 300 bases de données en interne et 650 instances dans un cloud AWS. Zalando contribue à la communauté, notamment par son outil de haute disponibilité patroni[80].

Le DBA de TripAdvisor témoigne de leur utilisation de PostgreSQL dans l'interview suivante[81].

Dès 2009, Leroy Merlin migrait vers PostgreSQL des milliers de logiciels de caisse[82].

Yandex, équivalent russe de Google a décrit en 2016 la migration des 300 To de données de Yandex.Mail depuis Oracle vers PostgreSQL[83].

La Société Générale a publié son outil de migration d'Oracle à PostgreSQL[84].

Autolib à Paris utilisait PostgreSQL. Le logiciel est encore utilisé dans les autres villes où le service continue. Ils ont décrit leur infrastructure au PG Day 2018 à Marseille[85].

De nombreuses autres sociétés participent au Groupe de Travail Inter-Entreprises de PostgreSQLFr[86] : Air France, Carrefour, Leclerc, le CNES, la MSA, la MAIF, PeopleDoc, EDF…

Cette liste ne comprend pas les innombrables sociétés qui n'ont pas communiqué sur le sujet. PostgreSQL étant un logiciel libre, il n'existe nulle part de dénombrement des instances actives.

1.5.4 Le Bon Coin

- Site de petites annonces
- 4è site le plus consulté en France (2017)
- 27 millions d'annonces en ligne, 800 000 nouvelles chaque jour
- Instance PostgreSQL principale : 3 To de volume, 3 To de RAM
- 20 serveurs secondaires

PostgreSQL tient la charge sur de grosses bases de données et des serveurs de grande taille.

[76] https://www.silicon.fr/cnaf-debarrasse-mainframes-149897.html?inf_by=5bc488a1671db858728b4c35
[77] https://media.postgresql.org/sfpug/instagram_sfpug.pdf
[78] http://gotocon.com/dl/goto-berlin-2013/slides/HenningJacobs_and_ValentineGogichashvili_WhyZalandoTrustsInPostgreSQL.pdf
[79] https://www.postgresql.eu/events/pgconfeu2018/schedule/session/2135-highway-to-hell-or-stairway-to-cloud/
[80] https://jobs.zalando.com/tech/blog/zalandos-patroni-a-template-for-high-availability-postgresql/
[81] https://www.citusdata.com/blog/25-terry/285-matthew-kelly-tripadvisor-talks-about-pgconf-silicon-valley
[82] https://wiki.postgresql.org/images/6/63/Adeo_PGDay.pdf
[83] https://www.pgcon.org/2016/schedule/attachments/426_2016.05.19%20Yandex.Mail%20success%20story.pdf
[84] https://github.com/societe-generale/code2pg
[85] https://www.youtube.com/watch?v=vd8B7B-Zca8
[86] https://www.postgresql.fr/entreprises/accueil

Le Bon Coin privilégie des serveurs physiques dans ses propres datacenters.

Pour plus de détails et l'évolution de la configuration, voir les témoignages de ses directeurs technique[87] (témoignage de juin 2012) et infrastructure[88] (juin 2017), ou la conférence de son DBA Flavio Gurgel au pgDay Paris 2019[89].

Ce dernier s'appuie sur les outils classiques fournis par la communauté : pg_dump (pour archivage, car ses exports peuvent être facilement restaurés), barman, pg_upgrade.

[87] https://www.postgresql.fr/temoignages:le_bon_coin
[88] https://web.archive.org/web/20171222173630/https://www.kissmyfrogs.com/jean-louis-bergamo-leboncoin-ce-qui-a-ete-fait-maison-est-ultra-performant/
[89] https://www.postgresql.eu/events/pgdayparis2019/schedule/session/2376-large-databases-lots-of-servers

1.6 À LA RENCONTRE DE LA COMMUNAUTÉ

- Cartographie du projet
- Pourquoi participer
- Comment participer

1.6.1 PostgreSQL, un projet mondial

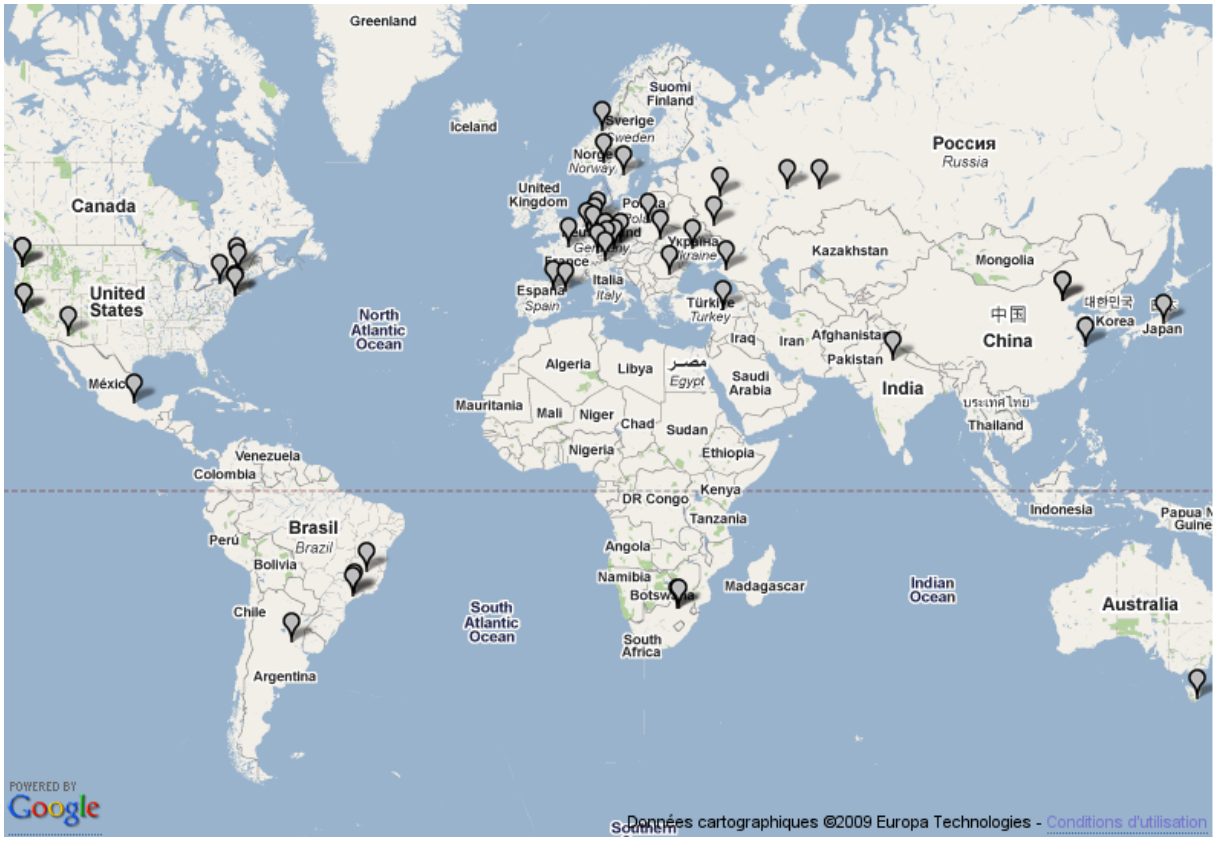


Figure 1/ .2: Carte des hackers

On le voit, PostgreSQL compte des contributeurs sur tous les continents.

Le projet est principalement anglophone. Les *core hackers* sont surtout répartis en Amérique, Europe, Asie (Japon surtout).

Il existe une très grande communauté au Japon, et de nombreux développeurs en Russie.

La communauté francophone est très dynamique, s'occupe beaucoup des outils, mais il n'y a que

quelques développeurs réguliers du *core* francophones : Michael Paquier, Julien Rouhaud, Fabien Coelho…

La communauté hispanophone est naissante.

1.6.2 PostgreSQL Core Team

Figure 1/ .3: Core team

Le terme *Core Hackers* désigne les personnes qui sont dans la communauté depuis longtemps. Ces personnes désignent directement les nouveaux membres.

NB : Le terme *hacker* peut porter à confusion, il s'agit bien ici de la définition « universitaire » : https://fr.wikipedia.org/wiki/Hacker_(programmation)

La *Core Team* est un ensemble de personnes doté d'un pouvoir assez limité. Ils ne doivent pas appartenir en majorité à la même société. Ils peuvent décider de la date de sortie d'une version. Ce sont les personnes qui sont immédiatement au courant des failles de sécurité du serveur PostgreSQL. Exceptionnellement, elles tranchent certains débats si un consensus ne peut être atteint dans la communauté. Tout le reste des décisions est pris par la communauté dans son ensemble après discussion, généralement sur la liste pgsql-hackers.

Les membres actuels de la *Core Team* sont[90] :

- **Tom Lane** (Crunchy Data, Pittsburgh, États-Unis) : certainement le développeur le plus aguerri avec la vision la plus globale, notamment sur l'optimiseur ;
- **Bruce Momjian** (EnterpriseDB, Philadelphie, États-Unis) : a lancé le projet en 1995, écrit du code (pg_upgrade notamment) et s'est beaucoup occupé de la promotion ;
- **Magnus Hagander** (Redpill Linpro, Stockholm, Suède) : développeur, a participé notamment au portage Windows, à l'outil pg_basebackup, à l'administration des serveurs, président de PostgreSQL Europe ;

[90] https://www.postgresql.org/community/contributors/

- **Andres Freund** (Microsoft, San Francisco, États-Unis) : contributeur depuis des années de nombreuses fonctionnalités (JIT, réplication logique, performances…) ;
- **Dave Page** (EnterpriseDB, Oxfordshire, Royaume-Uni) : leader du projet pgAdmin, version Windows, administration des serveurs, secrétaire de PostgreSQL Europe ;
- **Peter Eisentraut** (EnterpriseDB, Dresde, Allemagne) : développement du moteur (internationalisation, SQL/Med…), respect de la norme SQL, etc. ;
- **Jonathan Katz** (Crunchy Data, New York, États-Unis) : promotion du projet, modération, revues de patchs.

1.6.3 Contributeurs

Figure 1/ .4: Contributeurs

Actuellement, les « contributeurs » se répartissent quotidiennement les tâches suivantes :

- développement des projets satellites (pgAdmin…) ;
- promotion du logiciel ;
- administration de l'infrastructure des serveurs ;
- rédaction de documentation ;
- conférences ;
- traductions ;
- organisation de groupes locaux.

Le *PGDG* a fêté son 10e anniversaire à Toronto en juillet 2006. Ce « PostgreSQL Anniversary Summit » a réuni pas moins de 80 membres actifs du projet. La photo ci-dessus a été prise à l'occasion.

PGCon2009 a réuni 180 membres actifs à Ottawa, et environ 220 en 2018 et 2019.

Voir la liste des contributeurs officiels[91].

1.6.4 Qui contribue du code ?

- Principalement des personnes payées par leur société
- 30 committers
  - Tom Lane
  - Andres Freund
  - Peter Eisentraut
  - Nikita Glukhov
  - Álvaro Herrera
  - Michael Paquier
  - Robert Haas
  - …et beaucoup d'autres
- Commitfests[92] : tous les 2 mois
  - https://commitfest.postgresql.org/

Au printemps 2024, on compte 30 committers[93], c'est-à-dire personnes pouvant écrire dans tout ou partie du dépôt de PostgreSQL. Il ne s'agit pas que de leur travail, mais pour une bonne partie de patchs d'autres contributeurs après discussion et validation des fonctionnalités mais aussi des standards propres à PostgreSQL, de la documentation, de la portabilité, de la simplicité, de la sécurité, etc. Ces autres contributeurs peuvent être potentiellement n'importe qui. En général, un patch est relu par plusieurs personnes avant d'être transmis à un *committer*.

Les discussions quant au développement ont lieu principalement (mais pas uniquement) sur la liste pgsql-hackers[94]. Les éventuels bugs sont transmis à la liste pgsql-bugs[95]. Puis les patchs en cours sont revus au moins tous les deux mois lors des Commitfests[96]. Il n'y a pas de *bug tracker* car le fonctionnement actuel est jugé satisfaisant.

Robert Haas publie chaque année une analyse sur les contributeurs de code et les participants aux discussions sur le développement de PostgreSQL sur la liste pgsql-hackers :

- 2023 : http://rhaas.blogspot.com/2024/01/who-contributed-to-postgresql.html

[91] https://www.postgresql.org/community/contributors/
[93] https://www.postgresql.org/developer/committers/
[94] https://www.postgresql.org/list/pgsql-hackers/
[95] https://www.postgresql.org/list/pgsql-bugs/
[96] https://commitfest.postgresql.org/

- 2022 : http://rhaas.blogspot.com/2023/04/who-contributed-to-postgresql.html
- 2020/2021 : http://rhaas.blogspot.com/2022/01/who-contributed-to-postgresql.html
- 2019 : http://rhaas.blogspot.com/2020/05/who-contributed-to-postgresql.html
- 2018 : http://rhaas.blogspot.com/2019/01/who-contributed-to-postgresql.html
- 2017 : http://rhaas.blogspot.com/2018/06/who-contributed-to-postgresql.html
- 2016 : http://rhaas.blogspot.com/2017/04/who-contributes-to-postgresql.html

1.6.5 Répartition des développeurs

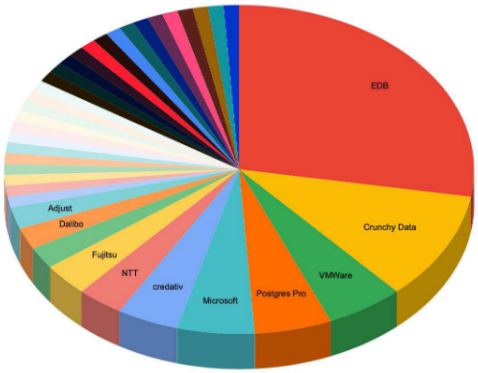


Figure 1/ .5: Répartition des développeurs

Voici une répartition des différentes sociétés qui ont contribué aux améliorations de la version 13. On y voit qu'un grand nombre de sociétés prend part à ce développement. La plus importante est EDB, mais même elle n'est responsable que d'un petit tiers des contributions.

(Source : Future Postgres Challenges[97], Bruce Momjian, 2021)

1.6.6 Utilisateurs

- Vous !
- **Le succès d'un logiciel libre dépend de ses utilisateurs.**

Il est impossible de connaître précisément le nombre d'utilisateurs de PostgreSQL. Toutefois ce nombre est en constante augmentation.

Il existe différentes manières de s'impliquer dans une communauté Open-Source. Dans le cas de PostgreSQL, vous pouvez :

[97] https://momjian.us/main/writings/pgsql/challenges.pdf

- déclarer un bug ;
- tester les versions bêta ;
- témoigner.

1.6.7 Pourquoi participer

- Rapidité des corrections de bugs
- Préparer les migrations / tester les nouvelles versions
- Augmenter la visibilité du projet
- Créer un réseau d'entraide

Au-delà de motivations idéologiques ou technologiques, il y a de nombreuses raisons objectives de participer au projet PostgreSQL.

Envoyer une description d'un problème applicatif aux développeurs est évidemment le meilleur moyen d'obtenir sa correction. Attention toutefois à être précis et complet lorsque vous déclarez un bug sur pgsql-bugs[98] ! Assurez-vous que vous pouvez le reproduire.

Tester les versions « candidates » dans votre environnement (matériel et applicatif) est la meilleure garantie que votre système d'information sera compatible avec les futures versions du logiciel.

Les retours d'expérience et les cas d'utilisations professionnelles sont autant de preuves de la qualité de PostgreSQL. Ces témoignages aident de nouveaux utilisateurs à opter pour PostgreSQL, ce qui renforce la communauté.

S'impliquer dans les efforts de traductions, de relecture ou dans les forums d'entraide ainsi que toute forme de transmission en général est un très bon moyen de vérifier et d'approfondir ses compétences.

1.6.8 Ressources web de la communauté

- Site officiel : https://www.postgresql.org/
- Actualité : https://planet.postgresql.org/
- Des extensions : https://pgxn.org/

Le site officiel de la communauté se trouve sur https://www.postgresql.org/. Ce site contient des informations sur PostgreSQL, la documentation des versions maintenues, les archives des listes de discussion, etc.

[98] https://www.postgresql.org/list/pgsql-bugs/

Le site « Planet PostgreSQL » est un agrégateur réunissant les blogs des *Core Hackers*, des contributeurs, des traducteurs et des utilisateurs de PostgreSQL.

Le site PGXN est l'équivalent pour PostgreSQL du CPAN de Perl, une collection en ligne de librairies et extensions accessibles depuis la ligne de commande.

1.6.9 Documentation officielle

- LA référence, même au quotidien
- Anglais : https://www.postgresql.org/docs/
- Français : https://docs.postgresql.fr/ <

La documentation officielle sur https://www.postgresql.org/docs/current est maintenue au même titre que le code du projet, et sert aussi au quotidien, pas uniquement pour des cas obscurs.

Elle est versionnée pour chaque version majeure.

La traduction française suit de près les mises à jour de la documentation officielle : https://docs.postgresql.fr/.

1.6.10 Serveurs francophones

- Site officiel : https://www.postgresql.fr/
- Documentation traduite : https://docs.postgresql.fr/
- Forum : https://forums.postgresql.fr/
- Actualité : https://planete.postgresql.fr/
- Association PostgreSQLFr : https://www.postgresql.fr/asso/accueil
- Groupe de Travail Inter-Entreprises (PGGTIE) : https://www.postgresql.fr/entreprises/accueil

Le site postgresql.fr est le site de l'association des utilisateurs francophones du logiciel. La communauté francophone se charge de la traduction de toutes les documentations.

1.6.11 Listes de discussions / Listes d'annonces

- pgsql-announce
- pgsql-general
- pgsql-admin
- pgsql-sql
- pgsql-performance
- pgsql-fr-generale
- pgsql-advocacy
- pgsql-bugs

Les mailing-lists sont les outils principaux de gouvernance du projet. Toute l'activité de la communauté (bugs, promotion, entraide, décisions) est accessible par ce canal. Les développeurs principaux du projets répondent parfois eux-mêmes. Si vous avez une question ou un problème, la réponse se trouve probablement dans les archives ! Pour s'inscrire ou consulter les archives : https://www.postgresql.org/list/.

Si vous pensez avoir trouvé un bug, vous pouvez le remonter sur la liste anglophone pgsql-bugs[99], par le formulaire dédié[100]. Pour faciliter la tâche de ceux qui tenteront de vous répondre, suivez bien les consignes sur les rapports de bug[101] : informations complètes, reproductibilité…

1.6.12 IRC

- Réseau LiberaChat
- IRC anglophone :
  - #postgresql
  - #postgresql-eu
- IRC francophone :
  - #postgresqlfr

Le point d'entrée principal pour le réseau LiberaChat est le serveur **irc.libera.chat**. La majorité des développeurs sont disponibles sur IRC et peuvent répondre à vos questions.

Des canaux de discussion spécifiques à certains projets connexes sont également disponibles, comme par exemple #slony.

[99] https://www.postgresql.org/list/pgsql-bugs/
[100] https://www.postgresql.org/account/submitbug/
[101] https://docs.postgresql.fr/current/bogue-reporting.html

Attention ! Vous devez poser votre question en public et ne pas solliciter de l'aide par message privé.

1.6.13 Wiki

- https://wiki.postgresql.org/

Le wiki est un outil de la communauté qui met à disposition une véritable mine d'informations.

Au départ, le wiki avait pour but de récupérer les spécifications écrites par des développeurs pour les grosses fonctionnalités à développer à plusieurs. Cependant, peu de développeurs l'utilisent dans ce cadre. L'utilisation du wiki a changé en passant plus entre les mains des utilisateurs qui y intègrent un bon nombre de pages de documentation (parfois reprises dans la documentation officielle). Le wiki est aussi utilisé par les organisateurs d'événements pour y déposer les slides des conférences. Elle n'est pas exhaustive et, hélas, souffre fréquemment d'un manque de mises à jour.

1.6.14 L'avenir de PostgreSQL

- PostgreSQL est devenu la base de données de référence
- Grandes orientations :
  - réplication logique
  - meilleur parallélisme
  - gros volumes
- Prochaine version, la 17
- Stabilité économique
- De plus en plus de (gros) clients
- Le futur de PostgreSQL dépend de vous !

Le projet avance grâce à de plus en plus de contributions. Les grandes orientations actuelles sont :

- une réplication de plus en plus sophistiquée ;
- une gestion plus étendue du parallélisme ;
- une volumétrie acceptée de plus en plus importante ;
- etc.

PostgreSQL est là pour durer. Le nombre d'utilisateurs, de toutes tailles, augmente tous les jours. Il n'y a pas qu'une seule entreprise derrière ce projet. Il y en a plusieurs, petites et grosses sociétés, qui

s'impliquent pour faire avancer le projet, avec des modèles économiques et des marchés différents, garants de la pérennité du projet.

1.7 CONCLUSION

- Un projet de grande ampleur
- Un SGBD complet
- Souplesse, extensibilité
- De belles références
- Une solution **stable**, **ouverte**, **performante** et **éprouvée**
- Pas de dépendance envers UN éditeur

Certes, la licence PostgreSQL implique un coût nul (pour l'acquisition de la licence), un code source disponible et aucune contrainte de redistribution. Toutefois, il serait erroné de réduire le succès de PostgreSQL à sa gratuité.

Beaucoup d'acteurs font le choix de leur SGBD sans se soucier de son prix. En l'occurrence, ce sont souvent les qualités intrinsèques de PostgreSQL qui séduisent :

- sécurité des données (reprise en cas de crash et résistance aux bogues applicatifs) ;
- facilité de configuration ;
- montée en puissance et en charge progressive ;
- gestion des gros volumes de données ;
- pas de dépendance envers un unique éditeur ou prestataire.

1.7.1 Bibliographie

- Documentation officielle (préface)
- Articles fondateurs de M. Stonebracker (1987)
- *Présentation du projet PostgreSQL* (Guillaume Lelarge, 2008)
- *Looking back at PostgreSQL* (J.M. Hellerstein, 2019)

Quelques références :

- Préface de la documentation officielle : 2. Bref historique de PostgreSQL[102]
- *The Design of POSTGRES*[103] , Michael Stonebraker & Lawrence A. Rowe, 1987
- Présentation du projet PostgreSQL[104], Guillaume Lelarge, RMLL 2008
- *Looking Back at PostgreSQL*[105] , Joseph M. Hellerstein, 2019

[102] https://docs.postgresql.fr/current/history.html
[103] http://db.cs.berkeley.edu/papers/ERL-M85-95.pdf
[104] https://web.archive.org/web/20160322070704/2008.rmll.info/Presentation-de-PostgreSQL.html
[105] https://arxiv.org/pdf/1901.01973.pdf

Iconographie : La photo initiale est le logo officiel de PostgreSQL[106].

1.7.2 Questions

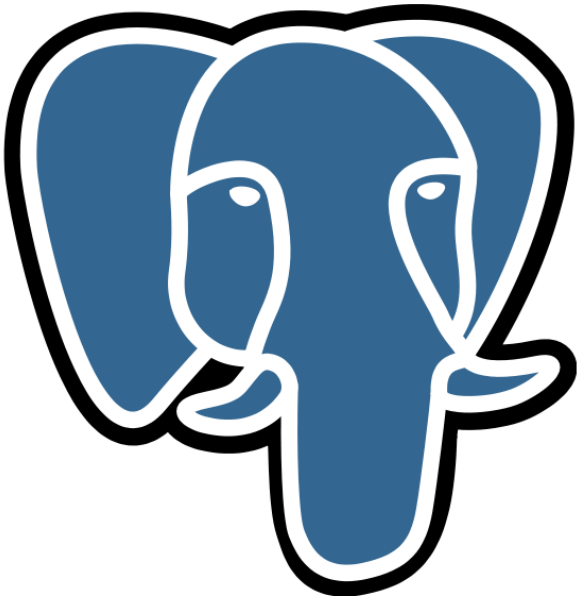
N'hésitez pas, c'est le moment !

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

1.8 QUIZ

https://dali.bo/a1_quiz

2/ Découverte des fonctionnalités



2.1 AU MENU

- Fonctionnalités du moteur
- Objets SQL
- Connaître les différentes fonctionnalités et possibilités
- Découvrir des exemples concrets

Ce module propose un tour rapide des fonctionnalités principales du moteur : ACID, MVCC, transactions, journaux de transactions… ainsi que des objets SQL gérés (schémas, index, tablespaces, triggers…). Ce rappel des concepts de base permet d'avancer plus facilement lors des modules suivants.

2.2 FONCTIONNALITÉS DU MOTEUR

- Standard SQL
- ACID : la gestion transactionnelle
- Niveaux d'isolation
- Journaux de transactions
- Administration
- Sauvegardes
- Réplication
- Supervision
- Sécurité
- Extensibilité

Cette partie couvre les différentes fonctionnalités d'un moteur de bases de données. Il ne s'agit pas d'aller dans le détail de chacune, mais de donner une idée de ce qui est disponible. Les modules suivants de cette formation et des autres formations détaillent certaines de ces fonctionnalités.

2.2.1 Respect du standard SQL

- Excellent support du SQL ISO
- Objets SQL
  - tables, vues, séquences, routines, triggers
- Opérations
  - jointures, sous-requêtes, requêtes CTE, requêtes de fenêtrage, etc.

La dernière version du standard SQL est SQL:2023[1]. À ce jour, aucun SGBD ne la supporte complètement, *mais* :

- PostgreSQL progresse et s'en approche au maximum, au fil des versions ;
- la majorité de la norme est supportée, parfois avec des syntaxes différentes ;
- PostgreSQL est le SGDB le plus respectueux du standard.

[1] https://en.wikipedia.org/wiki/SQL:2023

2.2.2 ACID

Gestion transactionnelle : la force des bases de données relationnelles :

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

Les propriétés ACID sont le fondement même de toute bonne base de données. Il s'agit de l'acronyme des quatre règles que toute transaction (c'est-à-dire une suite d'ordres modifiant les données) doit respecter :

- **A** : Une transaction est appliquée en « tout ou rien ».
- **C** : Une transaction amène la base d'un état stable à un autre.
- **I** : Les transactions n'agissent pas les unes sur les autres.
- **D** : Une transaction validée sera conservée de manière permanente.

Les bases de données relationnelles les plus courantes depuis des décennies (PostgreSQL bien sûr, mais aussi Oracle, MySQL, SQL Server, SQLite…) se basent sur ces principes, même si elles font chacune des compromis différents suivant leurs cas d'usage, les compromis acceptés à chaque époque avec la performance et les versions.

Atomicité :

Une transaction doit être exécutée entièrement ou pas du tout, et surtout pas partiellement, même si elle est longue et complexe, même en cas d'incident majeur sur la base de données. L'exemple basique est une transaction bancaire : le montant d'un virement doit être sur un compte ou un autre, et en cas de problème ne pas disparaître ou apparaître en double. Ce principe garantit que les données modifiées par des transactions valides seront toujours visibles dans un état stable, et évite nombre de problèmes fonctionnels comme techniques.

Cohérence :

Un état cohérent respecte les règles de validité définies dans le modèle, c'est-à-dire les contraintes définies dans le modèle : types, plages de valeurs admissibles, unicité, liens entre tables (clés étrangères), etc. Le non-respect de ces règles par l'applicatif entraîne une erreur et un rejet de la transaction.

Isolation :

Des transactions simultanées doivent agir comme si elles étaient seules sur la base. Surtout, elles ne voient pas les données *non validées* des autres transactions. Ainsi une transaction peut travailler sur un état stable et fixe, et durer assez longtemps sans risque de gêner les autres transactions.

Il existe plusieurs « niveaux d'isolation » pour définir précisément le comportement en cas de lectures ou écritures simultanées sur les mêmes données et pour arbitrer avec les contraintes de performances ; le niveau le plus contraignant exige que tout se passe comme si toutes les transactions se déroulaient successivement.

Durabilité :

Une fois une transaction validée par le serveur (typiquement : `COMMIT` ne retourne pas d'erreur, ce qui valide la cohérence et l'enregistrement physique), l'utilisateur doit avoir la garantie que la donnée ne sera pas perdue ; du moins jusqu'à ce qu'il décide de la modifier à nouveau. Cette garantie doit valoir même en cas d'événement catastrophique : plantage de la base, perte d'un disque… C'est donc au serveur de s'assurer autant que possible que les différents éléments (disque, système d'exploitation…) ont bien rempli leur office. C'est à l'humain d'arbitrer entre le niveau de criticité requis et les contraintes de performances et de ressources adéquates (et fiables) à fournir à la base de données.

NoSQL :

À l'inverse, les outils de la mouvance (« NoSQL », par exemple MongoDB ou Cassandra), ne fournissent pas les garanties ACID. C'est le cas de la plupart des bases non-relationnelles, qui reprennent le modèle BASE[2] (*Basically Available*, *Soft State*, *Eventually Consistent*, soit succintement : disponibilité d'abord ; incohérence possible entre les réplicas ; cohérence… à terme, après un délai). Un intérêt est de débarasser le développeur de certaines lourdeurs apparentes liées à la modélisation assez stricte d'une base de données relationnelle. Cependant, la plupart des applications ont d'abord besoin des garanties de sécurité et cohérence qu'offrent un moteur transactionnel classique, et la décision d'utiliser un système ne les garantissant pas ne doit pas être prise à la légère ; sans parler d'autres critères comme la fragmentation du domaine par rapport au monde relationnel et son SQL (à peu près) standardisé. Avec le temps, les moteurs transactionnels ont acquis des fonctionnalités qui faisaient l'intérêt des bases NoSQL (en premier lieu la facilité de réplication et le stockage de JSON), et ces dernières ont tenté d'intégrer un peu plus de sécurité dans leur modèle.

2.2.3 MVCC

- MultiVersion Concurrency Control
- Le « noyau » de PostgreSQL
- Garantit les propriétés ACID
- Permet les accès concurrents sur la même table
  - une lecture ne bloque pas une écriture
  - une écriture ne bloque pas une lecture
  - une écriture ne bloque pas les autres écritures…
  - …sauf pour la mise à jour de la **même ligne**

MVCC (Multi Version Concurrency Control) est le mécanisme interne de PostgreSQL utilisé pour garantir la cohérence des données lorsque plusieurs processus accèdent simultanément à la même table.

MVCC maintient toutes les versions nécessaires de chaque ligne, ainsi **chaque transaction voit une image figée de la base** (appelée *snapshot*). Cette image correspond à l'état de la base lors du dé-

[2] https://en.wikipedia.org/wiki/Eventual_consistency

marrage de la requête ou de la transaction, suivant le niveau d'*isolation* demandé par l'utilisateur à PostgreSQL pour la transaction.

MVCC fluidifie les mises à jour en évitant les blocages trop contraignants (verrous sur `UPDATE`) entre sessions et par conséquent de meilleures performances en contexte transactionnel.

C'est notamment MVCC qui permet d'exporter facilement une base *à chaud* et d'obtenir un export cohérent alors même que plusieurs utilisateurs sont potentiellement en train de modifier des données dans la base.

C'est la qualité de l'implémentation de ce système qui fait de PostgreSQL un des meilleurs SGBD au monde : chaque transaction travaille dans son image de la base, cohérent du début à la fin de ses opérations. Par ailleurs, les écrivains ne bloquent pas les lecteurs et les lecteurs ne bloquent pas les écrivains, contrairement aux SGBD s'appuyant sur des verrous de lignes. Cela assure de meilleures performances, moins de contention et un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

2.2.4 Transactions

- Une transaction = ensemble **atomique** d'opérations
- « Tout ou rien »
- `BEGIN` obligatoire pour grouper des modifications
- `COMMIT` pour valider
  - y compris le DDL
- Perte des modifications si :
  - `ROLLBACK` / perte de la connexion / arrêt (brutal ou non) du serveur
- `SAVEPOINT` pour sauvegarde des modifications d'une transaction à un instant `t`
- Pas de transactions imbriquées

L'exemple habituel et très connu des transactions est celui du virement d'une somme d'argent du compte de Bob vers le compte d'Alice. Le total du compte de Bob ne doit pas montrer qu'il a été débité de X euros tant que le compte d'Alice n'a pas été crédité de X euros. Nous souhaitons en fait que les deux opérations apparaissent aux yeux du reste du système comme une seule opération unitaire. D'où l'emploi d'une transaction explicite. En voici un exemple :

```
BEGIN;
UPDATE comptes SET solde=solde-200 WHERE proprietaire='Bob';
UPDATE comptes SET solde=solde+200 WHERE proprietaire='Alice';
COMMIT;
```

Contrairement à d'autres moteurs de bases de données, PostgreSQL accepte aussi les instructions DDL dans une transaction. En voici un exemple :

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);

SELECT age FROM capitaines;

 age
  35

ROLLBACK;
SELECT age FROM capitaines;

ERROR:  relation "capitaines" does not exist
LINE 1: SELECT age FROM capitaines;
                        ^
```

Nous voyons que la table `capitaines` a existé **à l'intérieur** de la transaction. Mais puisque cette transaction a été annulée (`ROLLBACK`), la table n'a pas été créée au final.

Cela montre aussi le support du DDL transactionnel au sein de PostgreSQL : PostgreSQL n'effectue aucun `COMMIT` implicite sur des ordres DDL tels que `CREATE TABLE`, `DROP TABLE` ou `TRUNCATE TABLE`. De ce fait, ces ordres peuvent être annulés au sein d'une transaction.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);
SAVEPOINT insert_sp;
UPDATE capitaines SET age = 45 WHERE nom = 'Haddock';
ROLLBACK TO SAVEPOINT insert_sp;
COMMIT;

SELECT age FROM capitaines WHERE nom = 'Haddock';

 age
  35
```

Malgré le `COMMIT` après l'`UPDATE`, la mise à jour n'est pas prise en compte. En effet, le `ROLLBACK TO SAVEPOINT` a permis d'annuler cet `UPDATE` mais pas les opérations précédant le `SAVEPOINT`.

À partir de la version 12, il est possible de chaîner les transactions avec `COMMIT AND CHAIN` ou `ROLLBACK AND CHAIN`. Cela veut dire terminer une transaction et en démarrer une autre immédiatement après avec les mêmes propriétés (par exemple, le niveau d'isolation).

2.2.5 Niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Nous pouvons spécifier le niveau d'isolation au démarrage d'une transaction
  - `BEGIN ISOLATION LEVEL xxx;`
- Niveaux d'isolation supportés
  - `read commited` (défaut)
  - `repeatable read`
  - `serializable`

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé sera un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

Le standard SQL spécifie quatre niveaux, mais PostgreSQL n'en supporte que trois (il n'y a pas de `read uncommitted` : les lignes non encore committées par les autres transactions sont toujours invisibles).

2.2.6 Fiabilité : journaux de transactions

- *Write Ahead Logs* (WAL)
- Chaque donnée est écrite **2 fois** sur le disque !
- Sécurité quasiment infaillible
- Avantages :
  - WAL : écriture séquentielle
  - un seul *sync* sur le WAL
  - fichiers de données : en asynchrone
  - sauvegarde PITR et de la réplication fiables

Les journaux de transactions (appelés souvent WAL, autrefois XLOG) sont une garantie contre les pertes de données.

Il s'agit d'une technique standard de journalisation appliquée à toutes les transactions. Ainsi lors d'une modification de donnée, l'écriture au niveau du disque se fait en deux temps :

- écriture immédiate dans le journal de transactions ;
- écriture dans le fichier de données, plus tard, lors du prochain *checkpoint*.

Ainsi en cas de crash :

- PostgreSQL redémarre ;
- PostgreSQL vérifie s'il reste des données non intégrées aux fichiers de données dans les journaux (mode *recovery*) ;
- si c'est le cas, ces données sont recopiées dans les fichiers de données afin de retrouver un état stable et cohérent.

Plus d'informations, lire cet article[3].

Les écritures dans le journal se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture (sur un disque dur classique, c'est l'opération la plus coûteuse).

De plus, comme nous n'écrivons que dans un seul fichier de transactions, la synchronisation sur disque peut se faire sur ce seul fichier, si le système de fichiers le supporte.

L'écriture définitive dans les fichiers de données, asynchrone et généralement de manière lissée, permet là aussi de gagner du temps.

Mais les performances ne sont pas la seule raison des journaux de transactions. Ces journaux ont aussi permis l'apparition de nouvelles fonctionnalités très intéressantes, comme le PITR et la réplication physique, basés sur le rejeu des informations stockées dans ces journaux.

2.2.7 Sauvegardes

- Sauvegarde des fichiers à froid
  - outils système
- Import/Export logique
  - `pg_dump`, `pg_dumpall`, `pg_restore`
- Sauvegarde physique à chaud
  - `pg_basebackup`
  - sauvegarde PITR

PostgreSQL supporte différentes solutions pour la sauvegarde.

La plus simple revient à sauvegarder à froid tous les fichiers des différents répertoires de données mais cela nécessite d'arrêter le serveur, ce qui occasionne une mise hors production plus ou moins longue, suivant la volumétrie à sauvegarder.

L'export logique se fait avec le serveur démarré. Plusieurs outils sont proposés : `pg_dump` pour sauvegarder une base, `pg_dumpall` pour sauvegarder toutes les bases. Suivant le format de l'export, l'import se fera avec les outils `psql` ou `pg_restore`. Les sauvegardes se font à chaud et sont cohérentes sans blocage de l'activité (seuls la suppression des tables et le changement de leur définition sont interdits).

Enfin, il est possible de sauvegarder les fichiers à chaud. Cela nécessite de mettre en place l'archivage des journaux de transactions. L'outil `pg_basebackup` est conseillé pour ce type de sauvegarde.

Il est à noter qu'il existe un grand nombre d'outils développés par la communauté pour faciliter encore plus la gestion des sauvegardes avec des fonctionnalités avancées comme le PITR (*Point In Time Recovery*) ou la gestion de la rétention, notamment pg_back (sauvegarde logique), pgBackRest ou barman (sauvegarde physique).

2.2.8 Réplication

- Réplication physique
  - instance complète
  - même architecture
- Réplication logique (PG 10+)
  - table par table / colonne par colonne avec ou sans filtre (PG 15)
  - voire opération par opération
- Asynchrones ou synchrone
- Asymétriques

PostgreSQL dispose de la réplication depuis de nombreuses années.

Le premier type de réplication intégrée est la réplication physique. Il n'y a pas de granularité, c'est forcément l'instance complète (toutes les bases de données), et au niveau des fichiers de données. Cette réplication est asymétrique : un seul serveur primaire effectue lectures comme écritures, et les serveurs secondaires n'acceptent que des lectures.

Le deuxième type de réplication est bien plus récent vu qu'il a été ajouté en version 10. Il s'agit d'une réplication logique, où les données elles-mêmes sont répliquées. Cette réplication est elle aussi asymétrique. Cependant, ceci se configure table par table (et non pas au niveau de l'instance comme pour la réplication physique). Avec la version 15, il devient possible de choisir quelles colonnes sont publiées et de filtre les lignes à publier.

La réplication logique n'est pas intéressante quand nous voulons un serveur sur lequel basculer en cas de problème sur le primaire. Dans ce cas, il vaut mieux utiliser la réplication physique. Par contre, c'est le bon type de réplication pour une réplication partielle ou pour une mise à jour de version majeure.

Dans les deux cas, les modifications sont transmises en asynchrone (avec un délai possible). Il est cependant possible de la configurer en synchrone pour tous les serveurs ou seulement certains.

2.2.9 Extensibilité

- Extensions
  - `CREATE EXTENSION monextension ;`
  - nombreuses : contrib, packagées… selon provenance
  - notion de confiance (v13+)
  - dont langages de procédures stockées !
- Système des *hooks*
- *Background workers*

Faute de pouvoir intégrer toutes les fonctionnalités demandées dans PostgreSQL, ses développeurs se sont attachés à permettre à l'utilisateur d'étendre lui-même les fonctionnalités sans avoir à modifier le code principal.

Ils ont donc ajouté la possibilité de créer des extensions. Une extension contient un ensemble de types de données, de fonctions, d'opérateurs, etc. en un seul objet logique. Il suffit de créer ou de supprimer cet objet logique pour intégrer ou supprimer tous les objets qu'il contient. Cela facilite grandement l'installation et la désinstallation de nombreux objets. Les extensions peuvent être codées en différents langages, généralement en C ou en PL/SQL. Elles ont eu un grand succès.

La possibilité de développer des routines dans différents langages en est un exemple : perl, python, PHP, Ruby ou JavaScript sont disponibles. PL/pgSQL est lui-même une extension à proprement parler, toujours présente.

Autre exemple : la possibilité d'ajouter des types de données, des routines et des opérateurs a permis l'émergence de la couche spatiale de PostgreSQL (appelée PostGIS).

Les provenances, rôle et niveau de finition des extensions sont très variables. Certaines sont des utilitaires éprouvés fournis avec PostgreSQL (parmi les « contrib »). D'autres sont des utilitaires aussi complexes que PostGIS ou un langage de procédures stockées. Des éditeurs diffusent leur produit comme une extension plutôt que *forker* PostgreSQL (Citus, timescaledb…). Beaucoup d'extensions peuvent être installées très simplement depuis des paquets disponibles dans les dépôts habituels (de la distribution ou du PGDG), ou le site du concepteur. Certaines sont diffusées comme code source à compiler. Comme tout logiciel, il faut faire attention à en vérifier la source, la qualité, la réputation et la pérennité.

Une fois les binaires de l'extension en place sur le serveur, l'ordre `CREATE EXTENSION` suffit généralement dans la base cible, et les fonctionnalités sont immédiatement exploitables.

Les extensions sont habituellement installées par un administrateur (un utilisateur doté de l'attribut `SUPERUSER`). À partir de la version 13, certaines extensions sont déclarées de confiance `trusted`). Ces extensions peuvent être installées par un utilisateur standard (à condition qu'il dispose des droits de création dans la base et le ou les schémas concernés).

Les développeurs de PostgreSQL ont aussi ajouté des *hooks* pour accrocher du code à exécuter sur certains cas. Cela a permis entre autres de créer l'extension `pg_stat_statements` qui s'accroche au code de l'exécuteur de requêtes pour savoir quelles sont les requêtes exécutées et pour récupérer des statistiques sur ces requêtes.

Enfin, les *background workers* ont vu le jour. Ce sont des processus spécifiques lancés par le serveur PostgreSQL lors de son démarrage et stoppés lors de son arrêt. Cela a permis la création de PoWA (outil qui historise les statistiques sur les requêtes) et une amélioration très intéressante de `pg_prewarm` (sauvegarde du contenu du cache disque à l'arrêt de PostgreSQL, restauration du contenu au démarrage).

Des exemples d'extensions sont décrites dans nos modules Extensions PostgreSQL pour l'utilisateur[4], Extensions PostgreSQL pour la performance[5], Extensions PostgreSQL pour les DBA[6].

2.2.10 Sécurité

- Fichier `pg_hba.conf`
- Filtrage IP
- Authentification interne (MD5, SCRAM-SHA-256)
- Authentification externe (identd, LDAP, Kerberos…)
- Support natif de SSL

Le filtrage des connexions se paramètre dans le fichier de configuration `pg_hba.conf`. Nous pouvons y définir quels utilisateurs (déclarés auprès de PostgreSQL) peuvent se connecter à quelles bases, et depuis quelles adresses IP.

L'authentification peut se baser sur des mots de passe chiffrés propres à PostgreSQL (`md5` ou le plus récent et plus sécurisé `scram-sha-256` en version 10), ou se baser sur une méthode externe (auprès de l'OS, ou notamment LDAP ou Kerberos qui couvre aussi Active Directory).

Si PostgreSQL interroge un service de mots de passe centralisé, vous devez toujours créer les rôle dans PostgreSQL. Seule l'option `WITH PASSWORD` est inutile. Pour créer, configurer mais aussi supprimer

[4] https://dali.bo/x1_html
[5] https://dali.bo/x2_html
[6] https://dali.bo/x3_html

les rôles depuis un annuaire, l'outil ldap2pg[7] existe.

L'authentification et le chiffrement de la connexion par SSL sont couverts.

[7] https://labs.dalibo.com/ldap2pg

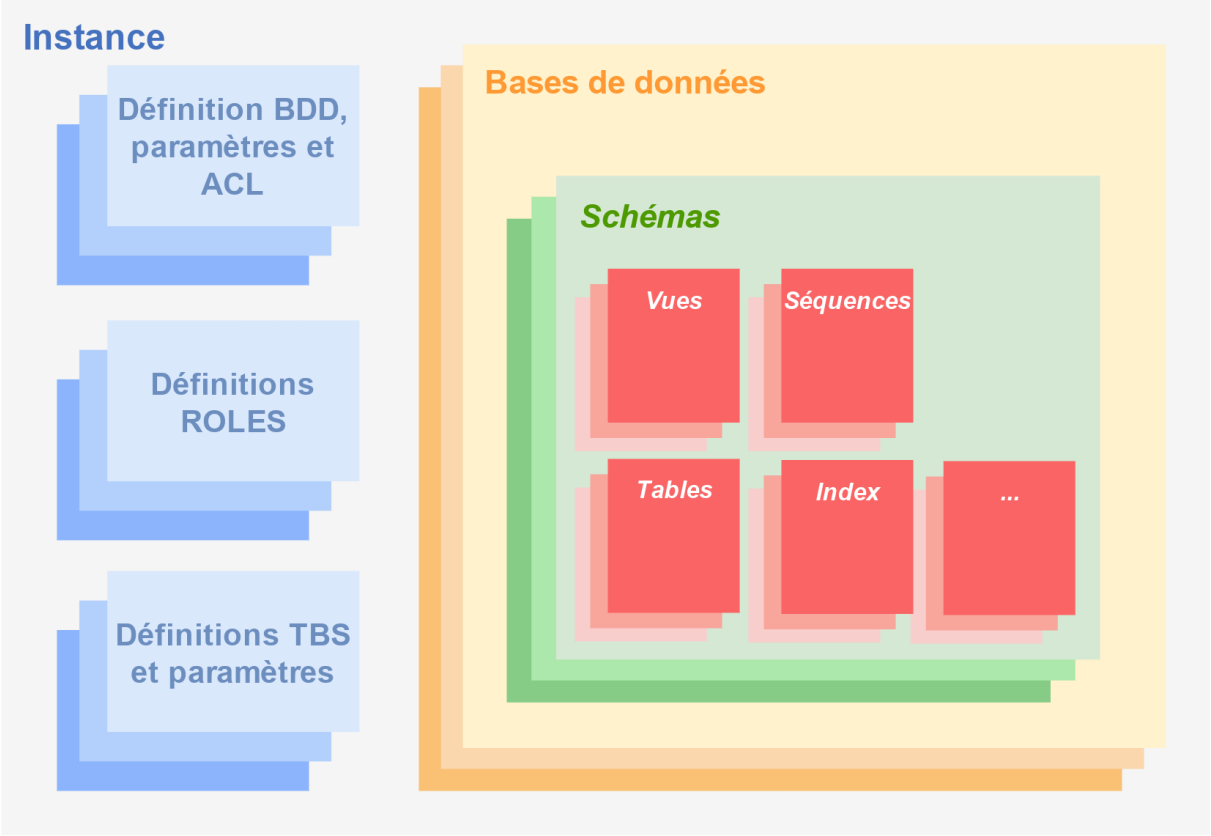
2.3 OBJETS SQL

- Instances
- Objets globaux :
  - Bases
  - Rôles
  - Tablespaces
- Objets locaux :
  - Schémas
  - Tables
  - Vues
  - Index
  - Routines
  - …

Le but de cette partie est de passer en revue les différents objets logiques maniés par un moteur de bases de données PostgreSQL.

Nous allons donc aborder la notion d'instance, les différents objets globaux et les objets locaux. Tous ne seront pas vus, mais le but est de donner une idée globale des objets et des fonctionnalités de PostgreSQL.

2.3.1 Organisation logique



Il est déjà important de bien comprendre une distinction entre les objets. Une instance est un ensemble de bases de données, de rôles et de tablespaces. Ces objets sont appelés des objets globaux parce qu'ils sont disponibles quelque soit la base de données de connexion. Chaque base de données contient ensuite des objets qui lui sont propres. Ils sont spécifiques à cette base de données et accessibles uniquement lorsque l'utilisateur est connecté à la base qui les contient. Il est donc possible de voir les bases comme des conteneurs hermétiques en dehors des objets globaux.

2.3.2 Instances

- Une instance
  - un répertoire de données
  - un port TCP
  - une configuration
  - plusieurs bases de données
- Plusieurs instances possibles sur un serveur

Une instance est un ensemble de bases de données. Après avoir installé PostgreSQL, il est nécessaire de créer un répertoire de données contenant un certain nombre de répertoires et de fichiers qui permettront à PostgreSQL de fonctionner de façon fiable. Le contenu de ce répertoire est créé initialement par la commande `initdb`. Ce répertoire stocke ensuite tous les objets des bases de données de l'instance, ainsi que leur contenu.

Chaque instance a sa propre configuration. Il n'est possible de lancer qu'un seul `postmaster` par instance, et ce dernier acceptera les connexions à partir d'un port TCP spécifique.

Il est possible d'avoir plusieurs instances sur le même serveur, physique ou virtuel. Dans ce cas, chaque instance aura son répertoire de données dédié et son port TCP dédié. Ceci est particulièrement utile quand l'on souhaite disposer de plusieurs versions de PostgreSQL sur le même serveur (par exemple pour tester une application sur ces différentes versions).

2.3.3 Rôles

- Utilisateurs / Groupes
  - Utilisateur : Permet de se connecter
- Différents attributs et droits

Une instance contient un ensemble de rôles. Certains sont prédéfinis et permettent de disposer de droits particuliers (lecture de fichier avec `pg_read_server_files`, annulation d'une requête avec `pg_signal_backend`, etc). Cependant, la majorité est composée de rôles créés pour permettre la connexion des utilisateurs.

Chaque rôle créé peut être utilisé pour se connecter à n'importe quelle base de l'instance, à condition que ce rôle en ait le droit. Ceci se gère directement avec l'attribution du droit `LOGIN` au rôle, et avec la configuration du fichier d'accès `pg_hba.conf`.

Chaque rôle peut être propriétaire d'objets, auquel cas il a tous les droits sur ces objets. Pour les objets dont il n'est pas propriétaire, il peut se voir donner des droits, en lecture, écriture, exécution, etc par le propriétaire.

Nous parlons aussi d'utilisateurs et de groupes. Un utilisateur est un rôle qui a la possibilité de se connecter aux bases alors qu'un groupe ne le peut pas. Un groupe sert principalement à gérer plus simplement les droits d'accès aux objets.

2.3.4 Tablespaces

- Répertoire physique contenant les fichiers de données de l'instance
- Une base peut
  - se trouver sur un seul tablespace
  - être répartie sur plusieurs tablespaces
- Permet de gérer l'espace disque et les performances
- Pas de quota

Toutes les données des tables, vues matérialisées et index sont stockées dans le répertoire de données principal. Cependant, il est possible de stocker des données ailleurs que dans ce répertoire. Il faut pour cela créer un tablespace. Un tablespace est tout simplement la déclaration d'un autre répertoire de données utilisable par PostgreSQL pour y stocker des données :

```
CREATE TABLESPACE chaud LOCATION '/SSD/tbl/chaud';
```

Il est possible d'avoir un tablespace par défaut pour une base de données, auquel cas tous les objets logiques créés dans cette base seront enregistrés physiquement dans le répertoire lié à ce tablespace. Il est aussi possible de créer des objets en indiquant spécifiquement un tablespace, ou de les déplacer d'un tablespace à un autre. Un objet spécifique ne peut appartenir qu'à un seul tablespace (autrement dit, un index ne pourra pas être enregistré sur deux tablespaces). Cependant, pour les objets partitionnés, le choix du tablespace peut se faire partition par partition.

Le but des tablespaces est de fournir une solution à des problèmes d'espace disque ou de performances. Si la partition où est stocké le répertoire des données principal se remplit fortement, il est possible de créer un tablespace dans une autre partition et donc d'utiliser l'espace disque de cette partition. Si de nouveaux disques plus rapides sont à disposition, il est possible de placer les objets fréquemment utilisés sur le tablespace contenant les disques rapides. Si des disques SSD sont à disposition, il est très intéressant d'y placer les index, les fichiers de tri temporaires, des tables de travail…

Par contre, contrairement à d'autres moteurs de bases de données, PostgreSQL n'a pas de notion de quotas. Les tablespaces ne peuvent donc pas être utilisés pour contraindre l'espace disque utilisé par certaines applications ou certains rôles.

2.3.5 Bases

- Conteneur hermétique
- Un rôle ne se connecte pas à une instance
  - il se connecte forcément à une base
- Une fois connecté, il ne voit que les objets de cette base
  - contournement : foreign data wrappers, dblink

Une base de données est un conteneur hermétique. En dehors des objets globaux, le rôle connecté à une base de données ne voit et ne peut interagir qu'avec les objets contenus dans cette base. De même, il ne voit pas les objets locaux des autres bases. Néanmoins, il est possible de lui donner le droit d'accéder à certains objets d'une autre base (de la même instance ou d'une autre instance) en utilisant les *Foreign Data Wrappers* (`postgres_fdw`) ou l'extension `dblink` .

Un rôle ne se connecte pas à l'instance. Il se connecte forcément à une base spécifique.

2.3.6 Schémas

- Espace de noms
- Sous-ensemble de la base
- Non lié à un utilisateur
- Résolution des objets : `search_path`
- `pg_catalog` , `information_schema`
  - pour catalogues système (lecture seule !)

Les schémas sont des espaces de noms à l'intérieur d'une base de données permettant :

- de grouper logiquement les objets d'une base de données ;
- de séparer les utilisateurs entre eux ;
- de contrôler plus efficacement les accès aux données ;
- d'éviter les conflits de noms dans les grosses bases de données.

Un schéma n'a à priori aucun lien avec un utilisateur donné.

Un schéma est un espace logique sans lien avec les emplacements physiques des données (ne pas confondre avec les *tablespaces*).

Un utilisateur peut avoir accès à tous les schémas ou à un sous-ensemble, tout dépend des droits dont il dispose. Depuis la version 15, un nouvel utilisateur n'a le droit de créer d'objet nulle part. Dans

les versions précédentes, il avait accès au schéma `public` de chaque base et pouvait y créer des objets.

Lorsque le schéma n'est pas indiqué explicitement pour les objets d'une requête, PostgreSQL recherche les objets dans les schémas listés par le paramètre `search_path` valable pour la session en cours .

Voici un exemple d'utilisation des schémas :

```
-- Création de deux schémas
CREATE SCHEMA s1;
CREATE SCHEMA s2;

-- Création d'une table sans spécification du schéma
CREATE TABLE t1 (id integer);

-- Comme le montre la méta-commande \d, la table est créée dans le schéma public

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres

-- Ceci est dû à la configuration par défaut du paramètre search_path
-- modification du search_path
SET search_path TO s1;

-- création d'une nouvelle table sans spécification du schéma
CREATE TABLE t2 (id integer);

-- Cette fois, le schéma de la nouvelle table est s1
-- car la configuration du search_path est à s1
-- Nous pouvons aussi remarquer que les tables capitaines et s1
-- ne sont plus affichées
-- Ceci est dû au fait que le search_path ne contient que le schéma s1 et
-- n'affiche donc que les objets de ce schéma.

postgres=# \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 s1     | t2   | table | postgres

-- Nouvelle modification du search_path
SET search_path TO s1, public;

-- Cette fois, les deux tables apparaissent

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
```

```
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
CREATE TABLE s2.t3 (id integer);

-- changement du search_path pour voir la table
SET search_path TO s1, s2, public;

-- La table apparaît bien, et le schéma d'appartenance est bien s2

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
-- attention, cette table a un nom déjà utilisé par une autre table
CREATE TABLE s2.t2 (id integer);

-- La création se passe bien car, même si le nom de la table est identique,
-- le schéma est différent
-- Par contre, \d ne montre que la première occurence de la table
-- ici, nous ne voyons t2 que dans s1

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Changeons le search_path pour placer s2 avant s1
SET search_path TO s2, s1, public;

-- Maintenant, la seule table t2 affichée est celle du schéma s2

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s2     | t2                | table    | postgres
 s2     | t3                | table    | postgres
```

Tous ces exemples se basent sur des ordres de création de table. Cependant, le comportement serait identique sur d'autres types de commande (`SELECT`, `INSERT`, etc) et sur d'autres types d'objets

locaux.

Pour des raisons de sécurité, il est très fortement conseillé de laisser le schéma `public` en toute fin du `search_path`. En effet, avant la version 15, s'il est placé au début, comme tout le monde avait le droit de créer des objets dans `public`, quelqu'un de mal intentionné pouvait placer un objet dans le schéma `public` pour servir de proxy à un autre objet d'un schéma situé après `public`. Même si la version 15 élimine ce risque, il reste la bonne pratique d'adapter le `search_path` pour placer les schémas applicatifs en premier.

Les schémas `pg_catalog` et `information_schema` contiennent des tables utilitaires (« catalogues système ») et des vues. Les catalogues système représentent l'endroit où une base de données relationnelle stocke les métadonnées des schémas, telles que les informations sur les tables, et les colonnes, et des données de suivi interne. Dans PostgreSQL, ce sont de simples tables. Un simple utilisateur lit fréquemment ces tables, plus ou moins directement, mais n'a aucune raison d'y modifier des données. Toutes les opérations habituelles pour un utilisateur ou administrateur sont disponibles sous la forme de commandes SQL.

Ne modifiez jamais directement les tables et vues système dans les schémas `pg_catalog` et `information_schema` ; n'y ajoutez ni n'y effacez jamais rien !

Même si cela est techniquement possible, seules des exceptions particulièrement ésotériques peuvent justifier une modification directe des tables systèmes (par exemple, une correction de vue système, suite à un bug corrigé dans une version mineure). Ces tables n'apparaissent d'ailleurs pas dans une sauvegarde logique (`pg_dump`).

2.3.7 Tables

Par défaut, une table est :

- Permanente
  - si temporaire, vivra le temps de la session (ou de la transaction)
- Journalisée
  - si *unlogged*, perdue en cas de crash, pas de réplication
- Non partitionnée
  - partitionnement possible par intervalle, valeur ou hachage

Par défaut, les tables sont permanentes, journalisées et non partitionnées.

Il est possible de créer des tables temporaires (`CREATE TEMPORARY TABLE`). Celles-ci ne sont visibles que par la session qui les a créées et seront supprimées par défaut à la fin de cette session. Il est

aussi possible de les supprimer automatiquement à la fin de la transaction qui les a créées. Il n'existe pas dans PostgreSQL de notion de table temporaire globale. Cependant, une extension[8] existe pour combler leur absence.

Pour des raisons de performance, il est possible de créer une table non journalisée (`CREATE UNLOGGED TABLE`). La définition de la table est journalisée mais pas son contenu. De ce fait, en cas de crash, il est impossible de dire si la table est corrompue ou non, et donc, au redémarrage du serveur, PostgreSQL vide la table de tout contenu. De plus, n'étant pas journalisée, la table n'est pas présente dans les sauvegardes PITR, ni repliquée vers d'éventuels serveurs secondaires.

Enfin, depuis la version 10, il est possible de partitionner les tables suivant un certain type de partitionnement : par intervalle, par valeur ou par hachage.

2.3.8 Vues

- Masquer la complexité
  - structure : interface cohérente vers les données, même si les tables évoluent
  - sécurité : contrôler l'accès aux données de manière sélective
- Vues matérialisées
  - à rafraîchir à une certaine fréquence

Le but des vues est de masquer une complexité, qu'elle soit du côté de la structure de la base ou de l'organisation des accès. Dans le premier cas, elles permettent de fournir un accès qui ne change pas même si les structures des tables évoluent. Dans le second cas, elles permettent l'accès à seulement certaines colonnes ou certaines lignes. De plus, les vues étant exécutées avec les mêmes droits que l'utilisateur qui les a créées, cela permet un changement temporaire des droits d'accès très appréciable dans certains cas.

Voici un exemple d'utilisation :

```
SET search_path TO public;

-- création de l'utilisateur guillaume
-- il n'aura pas accès à la table capitaines
-- par contre, il aura accès à la vue capitaines_anon
CREATE ROLE guillaume LOGIN;

-- ajoutons une colonne à la table capitaines
-- et ajoutons-y des données
ALTER TABLE capitaines ADD COLUMN num_cartecredit text;
INSERT INTO capitaines (nom, age, num_cartecredit)
  VALUES ('Robert Surcouf', 20, '1234567890123456');
```

[8] https://github.com/darold/pgtt

```
-- création de la vue
CREATE VIEW capitaines_anon AS
  SELECT nom, age, substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- ajout du droit de lecture à l'utilisateur guillaume
GRANT SELECT ON TABLE capitaines_anon TO guillaume;

-- connexion en tant qu'utilisateur guillaume
SET ROLE TO guillaume;

-- vérification qu'on lit bien la vue mais pas la table
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+-----------------
 Robert Surcouf |  20 | 123456789******

-- tentative de lecture directe de la table
SELECT * FROM capitaines;
ERROR:  permission denied for relation capitaines
```

Il est possible de modifier une vue en lui ajoutant des colonnes à la fin, au lieu de devoir les détruire et recréer (ainsi que toutes les vues qui en dépendent, ce qui peut être fastidieux).

Par exemple :

```
SET ROLE postgres;

CREATE OR REPLACE VIEW capitaines_anon AS SELECT
  nom,age,substring(num_cartecredit,0,10)||'******' AS num_cc_anon,
  md5(substring(num_cartecredit,0,10)) AS num_md5_cc
  FROM capitaines;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon    |            num_md5_cc
----------------+-----+-----------------+----------------------------------
 Robert Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

Nous pouvons aussi modifier les données au travers des vues simples, sans ajout de code et de trigger :

```
UPDATE capitaines_anon SET nom = 'Nicolas Surcouf' WHERE nom = 'Robert Surcouf';

SELECT * from capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon    |            num_md5_cc
-----------------+-----+-----------------+----------------------------------
 Nicolas Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b

UPDATE capitaines_anon SET num_cc_anon = '123456789xxxxxx'
  WHERE nom = 'Nicolas Surcouf';

ERROR:  cannot update column "num_cc_anon" of view "capitaines_anon"
DETAIL:  View columns that are not columns of their base relation
         are not updatable.
```

PostgreSQL gère le support natif des vues matérialisées (`CREATE MATERIALIZED VIEW nom_vue_mat AS SELECT …`). Les vues matérialisées sont des vues dont le contenu est figé sur disque, permettant de ne pas re-calculer leur contenu à chaque appel. De plus, il est possible de les indexer pour accélérer leur consultation. Il faut cependant faire attention à ce que leur contenu reste synchrone avec le reste des données.

Les vues matérialisées ne sont pas mises à jour automatiquement, il faut demander explicitement le rafraîchissement (`REFRESH MATERIALIZED VIEW`). Avec la clause `CONCURRENTLY`, s'il y a un index d'unicité, le rafraîchissement ne bloque pas les sessions lisant en même temps les données d'une vue matérialisée.

```
-- Suppression de la vue
DROP VIEW capitaines_anon;

-- Création de la vue matérialisée
CREATE MATERIALIZED VIEW capitaines_anon AS
  SELECT nom,
    age,
    substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- Les données sont bien dans la vue matérialisée
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******

-- Mise à jour d'une ligne de la table
-- Cette mise à jour est bien effectuée, mais la vue matérialisée
-- n'est pas impactée
UPDATE capitaines SET nom = 'Robert Surcouf' WHERE nom = 'Nicolas Surcouf';

SELECT * FROM capitaines WHERE nom LIKE '%Surcouf';

 id |      nom       | age | num_cartecredit
----+----------------+-----+------------------
  1 | Robert Surcouf |  20 | 1234567890123456

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+-----------------
 Nicolas Surcouf |  20 | 123456789******

-- Le résultat est le même mais le plan montre bien que PostgreSQL ne passe
-- plus par la table mais par la vue matérialisée :
EXPLAIN SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

                        QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on capitaines_anon  (cost=0.00..20.62 rows=1 width=68)
   Filter: (nom ~~ '%Surcouf'::text)

-- Après un rafraîchissement explicite de la vue matérialisée,
-- cette dernière contient bien les bonnes données
```

```
REFRESH MATERIALIZED VIEW capitaines_anon;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+-----------------
 Robert Surcouf |  20 | 123456789******

-- Pour rafraîchir la vue matérialisée sans bloquer les autres sessions :

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;

ERROR:  cannot refresh materialized view "public.capitaines_anon" concurrently
HINT:  Create a unique index with no WHERE clause on one or more columns
       of the materialized view.

-- En effet, il faut un index d'unicité pour faire un rafraîchissement
-- sans bloquer les autres sessions.
CREATE UNIQUE INDEX ON capitaines_anon(nom);

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;
```

2.3.9 Index

- Algorithmes supportés
  - B-tree (par défaut)
  - Hash
  - GiST / SP-GiST
  - GIN
  - BRIN
  - Bloom
- Type
  - Mono ou multicolonne
  - Partiel
  - Fonctionnel
  - Couvrant

PostgreSQL propose plusieurs algorithmes d'index.

Pour une indexation standard, nous utilisons en général un index B-tree, de par ses nombreuses possibilités et ses très bonnes performances.

Les index hash sont peu utilisés, essentiellement dans la comparaison d'égalité de grandes chaînes de caractères.

Moins simples d'abord, les index plus spécifiques (GIN, GIST) sont spécialisés pour les grands volumes de données complexes et multidimensionnelles : indexation textuelle, géométrique, géographique, ou de tableaux de données par exemple.

Les index BRIN sont des index très compacts destinés aux grandes tables où les données sont fortement corrélées par rapport à leur emplacement physique sur les disques.

Les index bloom sont des index probabilistes visant à indexer de nombreuses colonnes interrogées simultanément. Ils nécessitent l'ajout d'une extension (nommée `bloom`). Contrairement aux index btree, les index bloom ne dépendant pas de l'ordre des colonnes.

Le module `pg_trgm` permet l'utilisation d'index dans des cas habituellement impossibles, comme les expressions rationnelles et les `LIKE '%...%'`.

Généralement, l'indexation porte sur la valeur d'une ou plusieurs colonnes. Il est néanmoins possible de n'indexer qu'une partie des lignes (index partiel) ou le résultat d'une fonction sur une ou plusieurs colonnes en paramètre. Enfin, il est aussi possible de modifier les index de certaines contraintes (unicité et clé primaire) pour inclure des colonnes supplémentaires.

Plus d'informations :

- Article Wikipédia sur les arbres B[9] ;
- Article Wikipédia sur les tables de hachage[10] ;
- Documentation officielle française[11].

2.3.10 Types de données

- Types de base
  - natif : `int`, `float`
  - standard SQL : `numeric`, `char`, `varchar`, `date`, `time`, `timestamp`, `bool`
- Type complexe
  - tableau
  - JSON (`jsonb`), XML
  - vecteur (données LLM, FTS)
- Types métier
  - réseau, géométrique, etc.
- Types créés par les utilisateurs
  - structure SQL, C, Domaine, Enum

PostgreSQL dispose d'un grand nombre de types de base, certains natifs (comme la famille des `integer` et celle des `float`), et certains issus de la norme SQL (`numeric`, `char`, `varchar`, `date`, `time`, `timestamp`, `bool`).

Il dispose aussi de types plus complexes. Les tableaux (`array`) permettent de lister un ensemble de valeurs discontinues. Les intervalles (`range`) permettent d'indiquer toutes les valeurs comprises entre une valeur de début et une valeur de fin. Ces deux types dépendent évidemment d'un type de base : tableau d'entiers, intervalle de dates, etc. Existent aussi les types complexes les données XML et JSON (préférer le type optimisé `jsonb`).

PostgreSQL sait travailler avec des vecteurs pour des calculs avancé. De base, le type `tsvector` permet la recherche plein texte, avec calcul de proximité de mots dans un texte, pondération des résultats, etc. L'extension `pgvector` permet de stocker et d'indexer des vecteurs utilisé par les algorithmes LLM implémentés dans les IA génératives.

Enfin, il existe des types métiers ayant trait principalement au réseau (adresse IP, masque réseau), à la géométrie (point, ligne, boite). Certains sont apportés par des extensions.

Tout ce qui vient d'être décrit est natif. Il est cependant possible de créer ses propres types de données, soit en SQL soit en C. Les possibilités et les performances ne sont évidemment pas les mêmes.

Voici comment créer un type en SQL :

```
CREATE TYPE serveur AS (
  nom             text,
  adresse_ip      inet,
  administrateur  text
);
```

Ce type de données va pouvoir être utilisé dans tous les objets SQL habituels : table, routine, opérateur (pour redéfinir l'opérateur `+` par exemple), fonction d'agrégat, contrainte, etc.

Voici un exemple de création d'un opérateur :

```
CREATE OPERATOR + (
    leftarg = stock,
    rightarg = stock,
    procedure = stock_fusion,
    commutator = +
);
```

(Il faut au préalable avoir défini le type `stock` et la fonction `stock_fusion`.)

Il est aussi possible de définir des domaines. Ce sont des types créés par les utilisateurs à partir d'un type de base et en lui ajoutant des contraintes supplémentaires.

En voici un exemple :

```
CREATE DOMAIN code_postal_francais AS text CHECK (value ~ '^\d{5}$');
ALTER TABLE capitaines ADD COLUMN cp code_postal_francais;
UPDATE capitaines SET cp = '35400' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET cp = '1420' WHERE nom = 'Haddock';

ERROR:  value for domain code_postal_francais violates check constraint
        "code_postal_francais_check"
```

```
UPDATE capitaines SET cp = '01420' WHERE nom = 'Haddock';
SELECT * FROM capitaines;
```

```
 id |      nom       | age | num_cartecredit  |  cp
----+----------------+-----+------------------+-------
  1 | Robert Surcouf |  20 | 1234567890123456 | 35400
  1 | Haddock        |  35 |                  | 01420
```

Les domaines permettent d'intégrer la déclaration des contraintes à la déclaration d'un type, et donc de simplifier la maintenance de l'application si ce type peut être utilisé dans plusieurs tables : si la définition du code postal est insuffisante pour une évolution de l'application, il est possible de la modifier par un `ALTER DOMAIN`, et définir de nouvelles contraintes sur le domaine. Ces contraintes seront vérifiées sur l'ensemble des champs ayant le domaine comme type avant que la nouvelle version du type ne soit considérée comme valide.

Le défaut par rapport à des contraintes `CHECK` classiques sur une table est que l'information ne se trouvant pas dans la table, les contraintes sont plus difficiles à lister sur une table.

Enfin, il existe aussi les enums. Ce sont des types créés par les utilisateurs composés d'une liste ordonnée de chaînes de caractères.

En voici un exemple :

```
CREATE TYPE jour_semaine
  AS ENUM ('Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi',
  'Samedi', 'Dimanche');

ALTER TABLE capitaines ADD COLUMN jour_sortie jour_semaine;

UPDATE capitaines SET jour_sortie = 'Mardi' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET jour_sortie = 'Samedi' WHERE nom LIKE 'Haddock';

SELECT * FROM capitaines WHERE jour_sortie >= 'Jeudi';
```

```
 id |   nom   | age | num_cartecredit | cp | jour_sortie
----+---------+-----+-----------------+----+-------------
  1 | Haddock |  35 |                 |    | Samedi
```

Les *enums* permettent de déclarer une liste de valeurs statiques dans le dictionnaire de données plutôt que dans une table externe sur laquelle il faudrait rajouter des jointures : dans l'exemple, nous aurions pu créer une table `jour_de_la_semaine`, et stocker la clé associée dans `planning`. Nous aurions pu tout aussi bien positionner une contrainte `CHECK`, mais nous n'aurions plus eu une liste ordonnée.

Conférence de Heikki Linakangas sur la création d'un type color[12].

2.3.11 Contraintes

- `CHECK`
  - `prix > 0`
- `NOT NULL`
  - `id_client NOT NULL`
- Unicité
  - `id_client UNIQUE`
- Clés primaires
  - `UNIQUE NOT NULL` ==> `PRIMARY KEY (id_client)`
- Clés étrangères
  - `produit_id REFERENCES produits(id_produit)`
- `EXCLUDE`
  - `EXCLUDE USING gist (room WITH =, during WITH &&)`

Les contraintes sont la garantie de conserver des données de qualité ! Elles permettent une vérification qualitative des données, beaucoup plus fine qu'en définissant uniquement un type de données.

Les exemples ci-dessus reprennent :

- un prix qui doit être strictement positif ;
- un identifiant qui ne doit pas être vide (sinon des jointures filtreraient des lignes) ;
- une valeur qui doit être unique (comme des numéros de clients ou de facture) ;
- une clé primaire (unique non nulle), qui permet d'identifier précisément une ligne ;
- une clé étrangère vers la clé primaire d'une autre table (là encore pour garantir l'intégrité des jointures) ;
- une contrainte d'exclusion interdisant que deux plages temporelles se recouvrent dans la réservation de la même salle de réunion.

Les contraintes d'exclusion permettent un test sur plusieurs colonnes avec différents opérateurs (et non uniquement l'égalité, comme dans le cas d'une contrainte unique, qui n'est qu'une contrainte d'exclusion très spécialisée). Si le test se révèle positif, la ligne est refusée.

Une contrainte peut porter sur plusieurs champs et un champ peut être impliqué dans plusieurs contraintes :

```
CREATE TABLE commandes (
    no_commande     varchar(16) CHECK (no_commande ~ '^[A-Z0-9]*$'),
    id_entite_commerciale int REFERENCES entites_commerciales,
    id_client       int       REFERENCES clients,
    date_commande   date      NOT NULL,
    date_livraison  date      CHECK (date_livraison >= date_commande),
    PRIMARY KEY (no_commande, id_entite_commerciale)
);

\d commandes
                              Table « public.commandes »
        Colonne        |         Type          | … | NULL-able | Par défaut
-----------------------+-----------------------+---+-----------+------------
 no_commande           | character varying(16) |   | not null  |
 id_entite_commerciale | integer               |   | not null  |
 id_client             | integer               |   |           |
 date_commande         | date                  |   | not null  |
 date_livraison        | date                  |   |           |
Index :
    "commandes_pkey" PRIMARY KEY, btree (no_commande, id_entite_commerciale)
Contraintes de vérification :
    "commandes_check" CHECK (date_livraison >= date_commande)
    "commandes_no_commande_check" CHECK (no_commande::text ~ '^[A-Z0-9]*$'::text)
Contraintes de clés étrangères :
    "commandes_id_client_fkey" FOREIGN KEY (id_client) REFERENCES clients(id_client)
    "commandes_id_entite_commerciale_fkey" FOREIGN KEY (id_entite_commerciale)
 ↪  REFERENCES entites_commerciales(id_entite_commerciale)
```

Les contraintes doivent être vues comme la dernière ligne de défense de votre application face aux bugs. En effet, le code d'une application change beaucoup plus souvent que le schéma, et les données survivent souvent à l'application, qui peut être réécrite entretemps. Quoi qu'il se passe, des contraintes judicieuses garantissent qu'il n'y aura pas d'incohérence logique dans la base.

Si elles sont gênantes pour le développeur (car elles imposent un ordre d'insertion ou de mise à jour), il faut se rappeler que les contraintes peuvent être « débrayées » le temps d'une transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED ;
…
COMMIT ;
```

Les contraintes ne seront validées qu'au `COMMIT`.

Sur le sujet, voir par exemple *Constraints: a Developer's Secret Weapon*[13] de Will Leinweber (pgDay Paris 2018) (slides[14], vidéo[15]).

Du point de vue des performances, les contraintes permettent au planificateur d'optimiser les requêtes. Par exemple, le planificateur sait ne pas prendre en compte certaines jointures, notamment grâce à l'existence d'une contrainte d'unicité. (Sur ce point, la version 15 améliore les contraintes d'unicité en permettant de choisir si la valeur NULL est considérée comme unique ou pas. Par défaut

[13] https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2192-constraints-a-developers-secret-weapon/
[14] https://www.postgresql.eu/events/pgdayparis2018/sessions/session/1835/slides/70/2018-03-15%20constraints%20a%20developers%20secret%20weapon%20pgday%20paris.pdf
[15] https://youtu.be/hWh8QoV8z8k

et historiquement, une valeur NULL n'étant pas égal à une valeur NULL, les valeurs NULL sont considérées distinctes, et donc on peut avoir plusieurs valeurs NULL dans une colonne ayant une contrainte d'unicité.)

2.3.12 Colonnes à valeur générée

- Valeur calculée à l'insertion
- `DEFAULT`
- Identité
  - `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY`
- Expression
  - `GENERATED ALWAYS AS ( generation_expr ) STORED`

Une colonne a par défaut la valeur `NULL` si aucune valeur n'est fournie lors de l'insertion de la ligne. Il existe néanmoins trois cas où le moteur peut substituer une autre valeur.

Le plus connu correspond à la clause `DEFAULT`. Dans ce cas, la valeur insérée correspond à la valeur indiquée avec cette clause si aucune valeur n'est indiquée pour la colonne. Si une valeur est précisée, cette valeur surcharge la valeur par défaut. L'exemple suivant montre cela :

```
CREATE TABLE t2 (c1 integer, c2 integer, c3 integer DEFAULT 10);
INSERT INTO t2 (c1, c2, c3) VALUES (1, 2, 3);
INSERT INTO t2 (c1) VALUES (2);
SELECT * FROM t2;
```

```
 c1 | c2 | c3
----+----+----
  1 |  2 |  3
  2 |    | 10
```

La clause `DEFAULT` ne peut pas être utilisée avec des clauses complexes, notamment des clauses comprenant des requêtes.

Pour aller un peu plus loin, à partir de PostgreSQL 12, il est possible d'utiliser `GENERATED ALWAYS AS ( expression )`. Cela permet d'avoir une valeur calculée pour la colonne, valeur qui ne peut pas être surchargée, ni à l'insertion, ni à la mise à jour (mais qui est bien stockée sur le disque).

Comme exemple, nous allons reprendre la table `capitaines` et lui ajouter une colonne ayant comme valeur la version modifiée du numéro de carte de crédit :

```
ALTER TABLE capitaines
  ADD COLUMN num_cc_anon text
  GENERATED ALWAYS AS (substring(num_cartecredit, 0, 10) || '******') STORED;

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;
```

```
      nom       | num_cartecredit  |   num_cc_anon
----------------+------------------+-----------------
 Robert Surcouf | 1234567890123456 | 123456789******
 Haddock        |                  |
```

```
INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi', 'test');
```

```
ERROR:  cannot insert into column "num_cc_anon"
DETAIL:  Column "num_cc_anon" is a generated column.
```

```
INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi');
```

```
SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;
```

```
          nom          | num_cartecredit  |   num_cc_anon
-----------------------+------------------+-----------------
 Robert Surcouf        | 1234567890123456 | 123456789******
 Haddock               |                  |
 Joseph Pradere-Niquet | 9876543210987654 | 987654321******
```

Enfin, `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY` permet d'obtenir une colonne d'identité, bien meilleure que ce que le pseudo-type `serial` propose. Si `ALWAYS` est indiqué, la valeur n'est pas modifiable.

```
ALTER TABLE capitaines
  ADD COLUMN id2 integer GENERATED ALWAYS AS IDENTITY;
```

```
SELECT nom, id2 FROM capitaines;
```

```
          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3
```

```
INSERT INTO capitaines (nom) VALUES ('Tom Souville');
```

```
SELECT nom, id2 FROM capitaines;
```

```
          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3
 Tom Souville          |   4
```

Le type `serial` est remplacé par le type `integer` et une séquence comme le montre l'exemple suivant. C'est un problème dans la mesure ou la déclaration qui est faite à la création de la table produit un résultat différent en base et donc dans les exports de données.

```
CREATE TABLE tserial(s serial);
```

```
                            Table "public.tserial"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-----------------------------------
 s      | integer |           | not null | nextval('tserial_s_seq'::regclass)
```

2.3.13 Langages

- Procédures & fonctions en différents langages
- Par défaut : SQL, C et PL/pgSQL
- Extensions officielles : Perl, Python
- Mais aussi Java, Ruby, Javascript…
- Intérêts : fonctionnalités, performances

Les langages officiellement supportés par le projet sont :

- PL/pgSQL ;
- PL/Perl[16] ;
- PL/Python[17] ;
- PL/Tcl.

Voici une liste non exhaustive des langages procéduraux disponibles, à différents degrés de maturité :

- PL/sh[18] ;
- PL/R[19] ;
- PL/Java[20] ;
- PL/lolcode ;
- PL/Scheme ;
- PL/PHP ;
- PL/Ruby ;
- PL/Lua[21] ;
- PL/pgPSM ;
- PL/v8[22] (Javascript).

Tableau des langages supportés[23].

Pour qu'un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL. Les autres doivent être ajoutés à partir des paquets de la distribution ou du PGDG, ou compilés à la main, puis l'extension installée dans la base :

[16] https://docs.postgresql.fr/current/plperl.html
[17] https://docs.postgresql.fr/current/plpython.html
[18] https://github.com/petere/plsh
[19] https://github.com/postgres-plr/plr
[20] https://tada.github.io/pljava/
[21] https://github.com/pllua/pllua
[22] https://github.com/plv8/plv8

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
-- etc.
```

Ces fonctions peuvent être utilisées dans des index fonctionnels et des triggers comme toute fonction SQL ou PL/pgSQL.

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n'est pas performant quand il s'agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage. S'ajoute l'intérêt de ne pas avoir à réécrire en PL/pgSQL des fonctions existantes dans d'autres langages ou d'accéder à des modules bien établis de ces langages.

2.3.14 Fonctions & procédures

- Fonction
  - renvoie une ou plusieurs valeurs
  - `SETOF` ou `TABLE` pour plusieurs lignes
- Procédure (v11+)
  - ne renvoie rien
  - peut gérer le transactionnel dans certains cas

Historiquement, PostgreSQL ne proposait que l'écriture de fonctions. Depuis la version 11, il est aussi possible de créer des procédures. Le terme « routine » est utilisé pour signifier procédure ou fonction.

Une fonction renvoie une donnée. Cette donnée peut comporter une ou plusieurs colonnes. Elle peut aussi avoir plusieurs lignes dans le cas d'une fonction `SETOF` ou `TABLE`.

Une procédure ne renvoie rien. Elle a cependant un gros avantage par rapport aux fonctions dans le fait qu'elle peut gérer le transactionnel. Elle peut valider ou annuler la transaction en cours. Dans ce cas, une nouvelle transaction est ouverte immédiatement après la fin de la transaction précédente.

2.3.15 Opérateurs

- Dépend d'un ou deux types de données
- Utilise une fonction prédéfinie :

```
CREATE OPERATOR //
  (FUNCTION=division0,
  LEFTARG=integer,
  RIGHTARG=integer);
```

Il est possible de créer de nouveaux opérateurs sur un type de base ou sur un type utilisateur. Un opérateur exécute une fonction, soit à un argument pour un opérateur unitaire, soit à deux arguments pour un opérateur binaire.

Voici un exemple d'opérateur acceptant une division par zéro sans erreur :

```
-- définissons une fonction de division en PL/pgSQL
CREATE FUNCTION division0 (p1 integer, p2 integer) RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
  IF p2 = 0 THEN
      RETURN NULL;
  END IF;

  RETURN p1 / p2;
END
$$;

-- créons l'opérateur
CREATE OPERATOR // (FUNCTION = division0, LEFTARG = integer, RIGHTARG = integer);

-- une division normale se passe bien

SELECT 10/5;

 ?column?
----------
        2

SELECT 10//5;

 ?column?
----------
        2

-- une division par 0 ramène une erreur avec l'opérateur natif
SELECT 10/0;

ERROR:  division by zero
```

```
-- une division par 0 renvoie NULL avec notre opérateur
SELECT 10//0;

 ?column?
----------

(1 row)
```

2.3.16 Triggers

- Opérations : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`
- Trigger sur :
  - une colonne, et/ou avec condition
  - une vue
  - DDL
- Tables de transition
- Effet sur :
  - l'ensemble de la requête (`FOR STATEMENT`)
  - chaque ligne impactée (`FOR EACH ROW`)
- N'importe quel langage supporté

Les triggers peuvent être exécutés avant (`BEFORE`) ou après (`AFTER`) une opération.

Il est possible de les déclencher pour chaque ligne impactée (`FOR EACH ROW`) ou une seule fois pour l'ensemble de la requête (`FOR STATEMENT`). Dans le premier cas, il est possible d'accéder à la ligne impactée (ancienne et nouvelle version). Dans le deuxième cas, il a fallu attendre la version 10 pour disposer des tables de transition qui donnent à l'utilisateur une vision des lignes avant et après modification.

Par ailleurs, les triggers peuvent être écrits dans n'importe lequel des langages de routine supportés par PostgreSQL (C, PL/pgSQL, PL/Perl, etc.)

Exemple :

```
ALTER TABLE capitaines ADD COLUMN salaire integer;

CREATE FUNCTION verif_salaire()
RETURNS trigger AS $verif_salaire$
BEGIN
  -- Nous verifions que les variables ne sont pas vides
  IF NEW.nom IS NULL THEN
    RAISE EXCEPTION 'Le nom ne doit pas être null.';
  END IF;
```

```
  IF NEW.salaire IS NULL THEN
    RAISE EXCEPTION 'Le salaire ne doit pas être null.';
  END IF;

  -- pas de baisse de salaires !
  IF NEW.salaire < OLD.salaire THEN
    RAISE EXCEPTION 'Pas de baisse de salaire !';
  END IF;

  RETURN NEW;
END;
$verif_salaire$ LANGUAGE plpgsql;

CREATE TRIGGER verif_salaire BEFORE INSERT OR UPDATE ON capitaines
  FOR EACH ROW EXECUTE PROCEDURE verif_salaire();

UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 3000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';

ERROR:  pas de baisse de salaire !
CONTEXTE : PL/pgSQL function verif_salaire() line 13 at RAISE
```

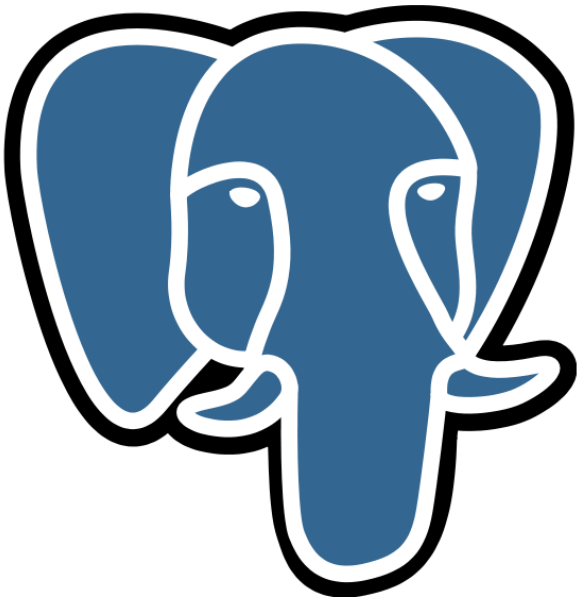
2.3.17 Questions

N'hésitez pas, c'est le moment !

2.4 QUIZ

https://dali.bo/a2_quiz

3/ Introduction aux plans d’exécution



3.1 INTRODUCTION

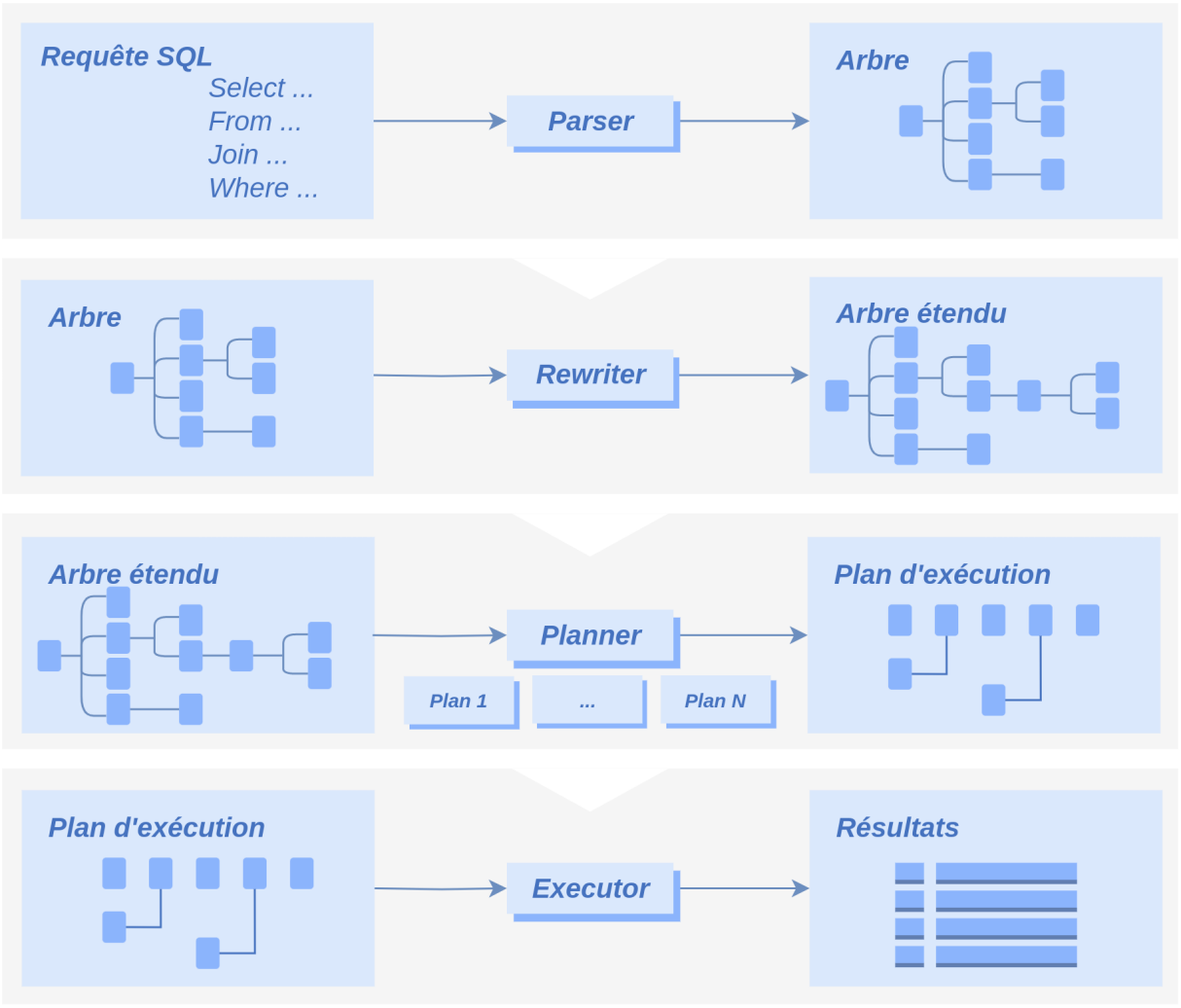
- Qu'est-ce qu'un plan d'exécution ?
- Quels outils peuvent aider

Ce module a pour but de faire une présentation très rapide de l'optimiseur et des plans d'exécution. Il contient surtout une introduction sur la commande `EXPLAIN` et sur différents outils en relation.

3.1.1 Au menu

- Exécution globale d'une requête
- Optimiseur
- `EXPLAIN`
- Nœuds d'un plan
- Outils

3.1.2 Niveau SGBD



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

Tout d'abord, le *parser* va réaliser une analyse syntaxique de la requête.

Puis 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. Il en est de même pour une fonction SQL intégrable.

Ensuite, 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.

Enfin, 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.

En général, le principal souci pour les performances sur ce type d'instructions est donc l'obtention des verrous et l'exécution réelle de la requête. Il existe quelques ordres (comme `TRUNCATE` ou `COPY`) exécutés beaucoup plus directement.

3.2 OPTIMISEUR

- SQL est un langage déclaratif
- Une requête décrit le résultat à obtenir
  - mais pas la façon pour l'obtenir
- C'est à l'optimiseur de déduire le moyen de parvenir au résultat demandé : comment ?

Les moteurs de base de données utilisent un langage SQL qui permet à l'utilisateur de décrire le résultat qu'il souhaite obtenir, mais pas la manière. C'est à la base de données de se débrouiller pour obtenir ce résultat le plus rapidement possible.

3.2.1 Principe de l'optimiseur

Le modèle vise à minimiser un coût :

- Énumérer tous les plans d'exécution
  - ou presque tous…
- Statistiques + configuration + règles → coût calculé
- Coût le plus bas = meilleur plan

Le but de l'optimiseur 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).

Pour calculer le « coût » d'un plan, PostgreSQL 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.

Le coût d'un plan est une valeur calculée sans unité ni signification physique.

3.2.2 Exemple de requête et son résultat

```
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
```

La requête en exemple permet de récupérer des informations sur tous les employés dont le nom commence par la lettre B en triant les employés par leur service.

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 à la fois 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.

3.2.3 Décisions de l'optimiseur

- Comment accéder aux lignes ?
  - parcours de table, d'index, de fonction, etc.
- Comment joindre les tables ?
  - ordre
  - type
- Comment agréger ?
  - brut, tri, hachage…

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut utiliser un parcours de table (une lecture complète du fichier), un parcours d'index ou encore d'autres types de parcours. Ce sont généralement les premières opérations utilisées.

Pour joindre les tables, l'ordre dans lequel ce sera fait est très important. Pour la jointure elle-même, il existe plusieurs méthodes différentes. Il existe aussi plusieurs algorithmes d'agrégation de lignes. Un tri peut être nécessaire pour une jointure, une agrégation, ou pour un `ORDER BY`, et là encore il y a plusieurs algorithmes possibles, ou des techniques pour éviter de le faire.

3.3 MÉCANISME DE CALCUL DE COÛTS

- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne
  - appliquer un opérateur
  - …
- et généralement un paramètre associé

L'optimiseur statistique de PostgreSQL utilise un modèle de calcul de coût. Les coûts calculés sont des indications arbitraires de la charge nécessaire pour répondre à une requête. Chaque facteur de coût représente une unité de travail : lecture d'un bloc, manipulation d'une ligne en mémoire, application d'un opérateur sur un champ.

3.3.1 Statistiques

- Connaître le coût de traitement d'une ligne est bien
  - mais combien de lignes à traiter ?
- Statistiques sur les données
  - mises à jour : `ANALYZE`
- Sans bonnes statistiques, pas de bons plans !

Connaître le coût unitaire de traitement d'une ligne est une bonne chose, mais si on ne sait pas le nombre de lignes à traiter, on ne peut pas calculer le coût total. L'optimiseur a donc besoin de statistiques sur les données, comme par exemple le nombre de blocs et de lignes d'une table, les valeurs les plus fréquentes et leur fréquence pour chaque colonne de chaque table. Les statistiques sur les données sont calculées lors de l'exécution de la commande SQL `ANALYZE`. L'autovacuum exécute généralement cette opération en arrière-plan.

Des statistiques périmées ou pas assez fines sont une source fréquente de plans non optimaux !

3.3.2 Exemple - parcours d'index

```
CREATE TABLE t1 (c1 integer, c2 integer);
INSERT INTO t1 SELECT i, i FROM generate_series(1, 1000) i;
CREATE INDEX ON t1(c1);
ANALYZE t1;

EXPLAIN SELECT * FROM t1 WHERE c1=1 ;

                          QUERY PLAN
---------------------------------------------------------------
Index Scan using t1_c1_idx on t1 (cost=0.28..8.29 rows=1 width=8)
  Index Cond: (c1 = 1)
```

L'exemple crée une table et lui ajoute 1000 lignes. Chaque ligne a une valeur différente dans les colonnes `c1` et `c2` (de 1 à 1000).

```
SELECT * FROM t1 ;
```

```
  c1  |  c2
------+------
    1 |    1
    2 |    2
    3 |    3
    4 |    4
    5 |    5
    6 |    6
…
  996 |  996
  997 |  997
  998 |  998
  999 |  999
 1000 | 1000
(1000 lignes)
```

Dans cette requête :

```
EXPLAIN SELECT * FROM t1 WHERE c1=1 ;
```

nous savons qu'un `SELECT` filtrant sur la valeur 1 pour la colonne `c1` ne ramènera qu'une ligne. Grâce aux statistiques relevées par la commande `ANALYZE` exécutée juste avant, l'optimiseur estime lui aussi qu'une seule ligne sera récupérée. Une ligne sur 1000, c'est un bon ratio pour faire un parcours d'index. C'est donc ce que recommande l'optimiseur.

3.3.3 Exemple - parcours de table

```
UPDATE t1 SET c1=1 ;   /* 1000 lignes identiques */

ANALYZE t1 ;           /* ne pas oublier ! */

EXPLAIN SELECT * FROM t1 WHERE c1=1;

                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..21.50 rows=1000 width=8)
   Filter: (c1 = 1)
```

La même table, mais avec 1000 lignes ne contenant plus que la valeur 1. Un `SELECT` filtrant sur cette valeur 1 ramènera dans ce cas toutes les lignes. L'optimiseur s'en rend compte et décide qu'un parcours séquentiel de la table est préférable à un parcours d'index. C'est donc ce que recommande l'optimiseur.

Dans cet exemple, l'ordre `ANALYZE` garantit que les statistiques sont à jour (le démon autovacuum n'est pas forcément assez rapide).

3.3.4 Exemple - parcours d'index forcé

```
SET enable_seqscan TO off ;

EXPLAIN SELECT * FROM t1 WHERE c1=1;

                          QUERY PLAN
---------------------------------------------------------------
Index Scan using t1_c1_idx on t1 (cost=0.28..57.77 rows=1000 width=8)
  Index Cond: (c1 = 1)

RESET enable_seqscan ;
```

Le coût du parcours de table était de 21,5 pour la récupération des 1000 lignes, donc un coût bien supérieur au coût du parcours d'index, qui lui était de 8,29, mais pour une seule ligne. On pourrait se demander le coût du parcours d'index pour 1000 lignes. À titre expérimental, on peut désactiver (ou plus exactement désavantager) le parcours de table en configurant le paramètre `enable_seqscan` à `off`.

En faisant cela, on s'aperçoit que le plan passe finalement par un parcours d'index, tout comme le premier. Par contre, le coût n'est plus de 8,29, mais de 57,77, donc supérieur au coût du parcours

de table. C'est pourquoi l'optimiseur avait d'emblée choisi un parcours de table. Un index n'est pas forcément le chemin le plus court.

3.4 QU'EST-CE QU'UN 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

L'optimiseur transforme une grosse action (exécuter une requête) en plein de petites actions unitaires (trier un ensemble de données, lire une table, parcourir un index, joindre deux ensembles de données, etc). Ces petites actions sont liées les unes aux autres. Par exemple, pour exécuter cette requête :

```
SELECT * FROM une_table ORDER BY une_colonne;
```

peut se faire en deux actions :

- récupérer les enregistrements de la table ;
- trier les enregistrements provenant de la lecture de la table.

Mais ce n'est qu'une des possibilités.

3.4.1 Nœud d'exécution

- Nœud
  - opération simple : lectures, jointures, tris, etc.
  - unité de traitement
  - produit et consomme des données
- Enchaînement des opérations
  - chaque nœud produit les données consommées par le nœud parent
  - le nœud final retourne les données à l'utilisateur

Les nœuds correspondent à des unités de traitement qui réalisent des opérations simples sur un ou deux ensembles de données : lecture d'une table, jointures entre deux tables, tri d'un ensemble, etc. Si le plan d'exécution était une recette, chaque nœud serait une étape de la recette.

Les nœuds peuvent produire et consommer des données.

3.4.2 Récupérer un plan d'exécution

- Commande `EXPLAIN`
  - suivi de la requête complète
- Uniquement le plan finalement retenu

Pour récupérer le plan d'exécution d'une requête, il suffit d'utiliser la commande `EXPLAIN`. Cette commande est suivie de la requête pour laquelle on souhaite le plan d'exécution.

Seul le plan sélectionné est affichable. Les plans ignorés du fait de leur coût trop important ne sont pas récupérables. Ceci est dû au fait que les plans en question peuvent être abandonnés avant d'avoir été totalement développés si leur coût partiel est déjà supérieur à celui de plans déjà considérés.

3.4.3 Exemple de requête

```
EXPLAIN SELECT * FROM t1  WHERE c2<10 ORDER BY c1;
```

Cette requête va récupérer tous les enregistrements de t1 pour lesquels la valeur de la colonne c2 est inférieure à 10. Les enregistrements sont triés par rapport à la colonne c1.

3.4.4 Plan pour cette requête

```
                          QUERY PLAN
-------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
   Sort Key: c1
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
         Filter: (c2 < 10)
```

L'optimiseur envoie ce plan à l'exécuteur. Ce dernier voit qu'il a une opération de tri à effectuer (nœud `Sort`). Pour cela, il a besoin de données que le nœud suivant va lui donner. Il commence donc l'opération de lecture (nœud `SeqScan`). Il envoie chaque enregistrement valide au nœud `Sort` pour que ce dernier les trie.

Chaque nœud dispose d'un certain nombre d'informations placées soit sur la même ligne entre des parenthèses, soit sur la ou les lignes du dessous. La différence entre une ligne de nœud et une ligne d'informations est que la ligne de nœud contient une flèche au début (`->`). Par exemple, le nœud `Sort` contient des informations entre des parenthèses et une information supplémentaire sur la ligne suivante indiquant la clé de tri (la colonne `c1`). Par contre, la troisième ligne n'est pas une ligne d'informations du nœud `Sort` mais un nouveau nœud (`SeqScan`).

3.4.5 Informations sur la ligne nœud

```
->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
      Filter: (c2 < 10)
```

- `cost` : coûts de récupération
  - de la **première** ligne
  - de **toutes** les lignes
- `rows`
  - nombre de lignes en sortie du nœud
- `width`
  - largeur moyenne d'un enregistrement (octets)

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 (elle dépend essentiellement de la taille de la table lue, mais aussi d'opération de filtrage). `rows` correspond au nombre de lignes que le planificateur pense récupérer à la sortie de ce nœud. Dans le cas d'une nouvelle table traitée par `ANALYZE`, les versions antérieures à la version 14 calculaient une valeur probable du nombre de lignes en se basant sur la taille moyenne d'une ligne et sur une table faisant 10 blocs. La version 14 corrige cela en ayant une meilleure idée du nombre de lignes d'une nouvelle table. `width` est la largeur en octets de la ligne.

3.4.6 Informations sur les lignes suivantes

```
Sort  (cost=21.64..21.67 rows=9 width=8)
  Sort Key: c1
Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
  Filter: (c2 < 10)
```

- `Sort`
  - `Sort Key` : clé de tri
- `Seq Scan`
  - `Filter` : filtre (si besoin)
- Dépend
  - du type de nœud
  - des options de `EXPLAIN`
  - des paramètres de configuration
  - de la version de PostgreSQL

Les informations supplémentaires dépendent de beaucoup d'éléments. Elles peuvent différer suivant le type de nœud, les options de la commande `EXPLAIN`, et certains paramètres de configuration. De même la version de PostgreSQL joue un rôle majeur : les nouvelles versions peuvent apporter des informations supplémentaires pour que le plan soit plus lisible et que l'utilisateur soit mieux informé.

3.4.7 Option ANALYZE

```
EXPLAIN (ANALYZE)  /* exécution !! */
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1;

                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
       (actual time=0.493..0.498 rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
                 (actual time=0.061..0.469 rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
 Planning Time: 0.239 ms
 Execution Time: 0.606 ms
```

Le but de cette option est d'obtenir les informations sur l'exécution réelle de la requête.

Avec `ANALYZE`, la requête est réellement exécutée ! Attention donc aux `INSERT` / `UPDATE` / `DELETE`. N'oubliez pas non plus qu'un `SELECT` peut appeler des fonctions qui écrivent dans la base. Dans le doute, pensez à englober l'appel dans une transaction que vous annulerez après coup.

Quatre nouvelles informations apparaissent dans un nouveau bloc de parenthèses. Elles sont 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'exécutions de ce nœud, soit dans le cadre d'une jointure, soit dans le cadre d'une requête parallélisée.

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).

3.4.8 Option BUFFERS

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1;

                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=17.64..17.67 rows=9 width=8)
       (actual time=0.126..0.127 rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   Buffers: shared hit=3 read=5
   ->  Seq Scan on t1  (cost=0.00..17.50 rows=9 width=8)
                       (actual time=0.017..0.106 rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
         Buffers: shared read=5
```

`BUFFERS` fait apparaître le nombre de blocs (*buffers*) impactés par chaque nœud du plan d'exécution, en lecture comme en écriture.

`shared read=5` en bas signifie que 5 blocs ont été trouvés et lus **hors** du cache de PostgreSQL (*shared buffers*). 5 blocs est ici la taille de `t1` sur le disque. Le cache de l'OS est peut-être intervenu, ce n'est pas visible ici. Un peu plus haut, `shared hit=3 read=5` indique que 3 blocs ont été lus dans ce cache, et 5 autres toujours hors du cache. Les valeurs exactes dépendent donc de l'état du cache. Si on relance la requête, pour une telle petite table, les relectures se feront uniquement en `shared hit`.

`BUFFERS` compte aussi les blocs de fichiers ou tables temporaires (`temp` ou `local`), ou les blocs écrits sur disque (`written`).

`EXPLAIN (ANALYZE, BUFFERS)` n'affiche que des données réelles, pas des estimations. Depuis PostgreSQL 13, `EXPLAIN (BUFFERS)` sans `ANALYZE` peut être utilisé, mais il ne montre que les quelques blocs utilisés par la planification, plutôt que tous ceux auxquels la requête accéderait réellement.

3.4.9 Option SETTINGS

```
SET enable_seqscan TO off ;
SET work_mem TO '100MB';

EXPLAIN (SETTINGS)
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1;

                          QUERY PLAN
---------------------------------------------------------------
Index Scan using t1_c1_idx on t1  (cost=0.28..57.77 rows=9 width=8)
  Filter: (c2 < 10)
Settings: enable_seqscan = 'off', work_mem = '100MB'

RESET ALL ;
```

Désactivée par défaut, l'option `SETTINGS` permet d'obtenir les valeurs des paramètres qui ne sont pas à leur valeur par défaut dans la session de la requête. Elle est pratique quand il faut transmettre le plan à un collègue ou un prestataire qui n'a pas forcément accès à la machine.

3.4.10 Option WAL

```
EXPLAIN (ANALYZE, WAL)
INSERT INTO t1 SELECT i, i FROM generate_series(1,1000) i ;

                        QUERY PLAN
-----------------------------------------------------
 Insert on t1  (cost=0.00..10.00 rows=1000 width=8)
          (actual time=8.078..8.079 rows=0 loops=1)
   WAL: records=2017 fpi=3 bytes=162673
   ->  Function Scan on generate_series i
       (cost=0.00..10.00 rows=1000 width=8)
       (actual time=0.222..0.522 rows=1000 loops=1)
 Planning Time: 0.076 ms
 Execution Time: 8.141 ms
```

Désactivée par défaut et nécessitant l'option `ANALYZE`, l'option `WAL` permet d'obtenir le nombre d'enregistrements et le nombre d'octets écrits dans les journaux de transactions. (Rappelons que les écritures dans les fichiers de données se font généralement plus tard, en arrière-plan.)

3.4.11 Option GENERIC_PLAN

Quel plan générique pour les requêtes préparées ?

```
EXPLAIN (GENERIC_PLAN)
SELECT * FROM t1 WHERE c1 < $1 ;
```

- PostgreSQL 16

L'option `GENERIC_PLAN` n'est malheureusement pas disponible avant PostgreSQL 16. Elle est pourtant très pratique quand on cherche le plan d'une requête préparée sans connaître ses paramètres, ou pour savoir quel est le plan générique que prévoit PostgreSQL pour une requête préparée.

En effet, les plans des requêtes préparées ne sont pas forcément recalculés à chaque appel avec les paramètres exacts (le système est assez complexe et dépend du paramètre `plan_cache_mode`). La requête ne peut être exécutée sans vraie valeur de paramètre, donc l'option `ANALYZE` est inutilisable, mais en activant `GENERIC_PLAN` on peut tout de même voir le plan générique que PostgreSQL peut choisir (`SUMMARY ON` affiche en plus le temps de planification) :

```
EXPLAIN (GENERIC_PLAN, SUMMARY ON)
SELECT * FROM t1 WHERE c1 < $1 ;

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

```
 Index Scan using t1_c1_idx on t1  (cost=0.15..14.98 rows=333 width=8)
   Index Cond: (c1 < $1)
 Planning Time: 0.195 ms
```

C'est effectivement le plan qui serait optimal pour `$1` =1. Mais pour la valeur 1000, qui ramène toute la table, un *Seq Scan* serait plus pertinent.

3.4.12 Autres options

- `COSTS OFF`
  - masquer les coûts
- `TIMING OFF`
  - désactiver le chronométrage & des informations vues/calculées par l'optimiseur
- `VERBOSE`
  - affichage verbeux : schémas, colonnes, workers
- `SUMMARY`
  - affichage du temps de planification et exécution (si applicable)
- `FORMAT`
  - sortie en texte, JSON, XML, YAML

Ces options sont moins utilisées, mais certaines restent intéressantes dans des cas précis.

Option COSTS

Cette option est activée par défaut. Il peut être intéressant de la désactiver pour n'avoir que le plan.

```
EXPLAIN (COSTS OFF) SELECT * FROM t1 WHERE c2<10 ORDER BY c1 ;
```

```
        QUERY PLAN
--------------------------
 Sort
   Sort Key: c1
   ->  Seq Scan on t1
         Filter: (c2 < 10)
```

Option TIMING

Cette option est activée par défaut. Il peut être intéressant de le désactiver sur les systèmes où le chronométrage prend beaucoup de temps et allonge inutilement la durée d'exécution de la requête. Mais de ce fait, le résultat devient beaucoup moins intéressant.

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t1 WHERE c2<10 ORDER BY c1 ;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8) (actual rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8) (actual rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
 Planning Time: 0.155 ms
 Execution Time: 0.381 ms
```

Option VERBOSE

Désactivée par défaut, l'option `VERBOSE` permet d'afficher des informations supplémentaires comme :

- la liste des colonnes en sortie ;
- le nom des objets qualifiés par le nom du schéma ;
- des statistiques sur les workers (pour les requêtes parallélisées) ;
- le code SQL envoyé à un serveur distant (pour les tables distantes avec `postgres_fdw` notamment).

Dans l'exemple suivant, le nom du schéma est ajouté au nom de la table. La nouvelle ligne `Output` indique la liste des colonnes de l'ensemble de données en sortie du nœud.

```
EXPLAIN (VERBOSE) SELECT * FROM t1 WHERE c2<10 ORDER BY c1 ;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
   Output: c1, c2
   Sort Key: t1.c1
   ->  Seq Scan on public.t1  (cost=0.00..21.50 rows=9 width=8)
         Output: c1, c2
         Filter: (t1.c2 < 10)
```

Option SUMMARY

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 (la durée de planification est pourtant parfois importante). Par contre, un `EXPLAIN ANALYZE` l'affiche par défaut.

```
EXPLAIN (SUMMARY ON) SELECT * FROM t1 WHERE c2<10 ORDER BY c1;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
   Sort Key: c1
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
         Filter: (c2 < 10)
 Planning Time: 0.185 ms
```

```
EXPLAIN (ANALYZE, SUMMARY OFF) SELECT * FROM t1 WHERE c2<10 ORDER BY c1;
```

```
                     QUERY PLAN
-----------------------------------------------------------
 Sort  (cost=21.64..21.67 rows=9 width=8)
       (actual time=0.343..0.346 rows=9 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t1  (cost=0.00..21.50 rows=9 width=8)
                 (actual time=0.031..0.331 rows=9 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 991
```

Option FORMAT

L'option `FORMAT` permet de préciser le format du texte en sortie. Par défaut, il s'agit du format texte habituel, mais il est possible de choisir un format semi-structuré parmi JSON, XML et YAML. Les formats semi-structurés sont utilisés principalement par des outils d'analyse comme explain.dalibo.com[1], car le contenu est plus facile à analyser, et même un peu plus complet. Voici ce que donne la commande `EXPLAIN` avec le format JSON :

```
psql -X -AtX \
-c 'EXPLAIN (FORMAT JSON) SELECT * FROM t1 WHERE c2<10 ORDER BY c1' | jq '.[]'
```

```
{
  "Plan": {
    "Node Type": "Sort",
    "Parallel Aware": false,
    "Async Capable": false,
    "Startup Cost": 34.38,
    "Total Cost": 34.42,
    "Plan Rows": 18,
    "Plan Width": 8,
    "Sort Key": [
      "c1"
    ],
    "Plans": [
      {
        "Node Type": "Seq Scan",
        "Parent Relationship": "Outer",
        "Parallel Aware": false,
        "Async Capable": false,
        "Relation Name": "t1",
        "Alias": "t1",
        "Startup Cost": 0,
        "Total Cost": 34,
        "Plan Rows": 18,
        "Plan Width": 8,
        "Filter": "(c2 < 10)"
      }
    ]
  }
}
```

[1] https://explain.dalibo.com

3.4.13 Paramètre track_io_timing

```
SET track_io_timing TO on;
EXPLAIN (ANALYZE, BUFFERS)
SELECT   *   FROM t1   WHERE c2<10   ORDER BY c1 ;

                              QUERY PLAN
-----------------------------------------------------------
 Sort  (cost=52.14..52.21 rows=27 width=8) (actual time=1.359..1.366
↪  rows=27 loops=1)
   …
   Buffers: shared hit=3 read=14
   I/O Timings: read=0.388
   ->  Seq Scan on t1  (cost=0.00..51.50 rows=27 width=8) (actual
↪  time=0.086..1.233 rows=27 loops=1)
         Filter: (c2 < 10)
         Rows Removed by Filter: 2973
         Buffers: shared read=14
         I/O Timings: read=0.388
 Planning:
   Buffers: shared hit=43 read=14
   I/O Timings: read=0.469
 Planning Time: 1.387 ms
 Execution Time: 1.470 ms
```

La configuration du paramètre `track_io_timing` permet de demander le chronométrage des opérations d'entrée/sortie disque. Sur ce plan, nous pouvons voir que 14 blocs ont été lus en dehors du cache de PostgreSQL et que cela a pris 0,388 ms pour les lire (ils étaient certainement dans le cache du système d'exploitation).

Cette information permet de voir si le temps d'exécution de la requête est dépensé surtout dans la demande de blocs au système d'exploitation (donc hors du cache de PostgreSQL) ou dans l'exécution même de la requête (donc interne à PostgreSQL).

3.4.14 Détecter les problèmes

- Temps d'exécution de chaque opération
- Différence entre l'estimation du nombre de lignes et la réalité
- Boucles
  - appels, même rapides, nombreux
- Opérations utilisant beaucoup de blocs (`BUFFERS`)
- Opérations lentes de lecture/écriture (`track_io_timing`)

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.

Par exemple, une différence importante entre le nombre estimé de lignes et le nombre réel de lignes laisse un doute sur les statistiques présentes. Soit elles n'ont pas été réactualisées récemment, soit l'échantillon n'est pas suffisamment important pour que les statistiques donnent une vue proche du réel du contenu de la table.

Les boucles sont à surveiller. Par exemple, un accès à une ligne par un index est généralement très rapide, mais répété des millions de fois à cause d'une boucle, le total est parfois plus long qu'une lecture complète de la table indexée. C'est notamment l'enjeu du réglage entre `seq_page_cost` et `random_page_cost`.

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 du cache de PostgreSQL. Sachant qu'un bloc fait généralement 8 kilo-octets, il est aisé de déterminer le volume de données manipulé par une requête.

3.5 NŒUDS D'EXÉCUTION LES PLUS COURANTS (INTRODUCTION)

- Parcours
- Jointures
- Agrégats
- Tri

Nous n'allons pas détailler tous les nœuds existants, mais évoquer simplement les plus importants. Une analyse plus poussée des nœuds et une référence complète sont disponibles dans les modules J2[2] et J6[3].

3.5.1 Parcours

- Table
  - *Seq Scan*, *Parallel Seq Scan*
- Index
  - *Index Scan*, *Bitmap Scan*, *Index Only Scan*
  - et les variantes parallélisées
- Autres
  - *Function Scan*, *Values Scan*

Plusieurs types d'objets peuvent être parcourus. Pour chacun, l'optimiseur peut choisir entre plusieurs types de parcours.

Les tables passent par un *Seq Scan* qui est une lecture simple de la table, bloc par bloc, ligne par ligne. Ce parcours peut filtrer les données mais ne les triera pas. Une variante parallélisée existe sous le nom de *Parallel Seq Scan*.

Les index disposent de plusieurs parcours, principalement suivant la quantité d'enregistrements à récupérer :

- *Index Scan* quand il y a très peu d'enregistrements à récupérer ;
- *Bitmap Scan* quand il y en a un peu plus ou quand on veut lire plusieurs index d'une même table pour satisfaire plusieurs conditions de filtre ;

[2] https://dali.bo/j2_html
[3] https://dali.bo/j6_html

- *Index Only Scan* quand les champs de la requête correspondent aux colonnes de l'index (ce qui permet d'éviter la lecture de tout ou partie de la table).

Ces différents parcours sont parallélisables. Ils ont dans ce cas le mot *Parallel* ajouté en début du nom du nœud.

Enfin, il existe des parcours moins fréquents, comme les parcours de fonction (*Function Scan*) ou de valeurs (*Values Scan*).

3.5.2 Jointures

- Algorithmes
  - *Nested Loop*
  - *Hash Join*
  - *Merge Join*
- Parallélisation possible
- Pour `EXISTS`, `IN` et certaines jointures externes
  - *Semi Join*
  - *Anti Join*

Trois nœuds existent pour les jointures.

Le *Nested Loop* est utilisé pour toutes les conditions de jointure n'utilisant pas l'opérateur d'égalité. Il est aussi utilisé quand un des deux ensembles de données renvoie très peu de données.

Le *Hash Join* est certainement le nœud le plus commun. Il est utilisé un peu dans tous les cas, sauf si les deux ensembles de données arrivent déjà triés. Dans ce cas, il est préférable de passer par un *Merge Join* qui réclame deux ensembles de données déjà triés.

Les *Semi Join* et *Anti Join* sont utilisés dans des cas très particuliers et peu fréquents.

3.5.3 Agrégats

- Un résultat au total
  - *Aggregate*
- Un résultat par regroupement
  - *Hash Aggregate*
  - *Group Aggregate*
  - *Mixed Aggregate*
- Parallélisation
  - *Partial Aggregate*
  - *Finalize Aggregate*

De même il existe plusieurs algorithmes d'agrégation qui s'occupent des sommes, des moyennes, des regroupements divers, etc. Ils sont souvent parallélisables.

3.5.4 Opérations unitaires

- *Sort*
- *Incremental Sort*
- *Limit*
- *Unique* (`DISTINCT`)
- *Append* (`UNION ALL`), *Except*, *Intersect*
- *Gather* (parallélisme)
- *Memoize* (14+)

Un grand nombre de petites opérations ont leur propre nœud, comme le tri avec *Sort* et *Incremental Sort*, la limite de lignes (`LIMIT`) avec *Limit*, la clause `DISTINCT` avec *Unique*), etc. Elles prennent généralement un ensemble de données et renvoient un autre ensemble de données issu du traitement du premier.

Le groupe des nœuds *Append*, *Except* et *Intersect* ne se comporte pas ainsi. Notamment, *Append* est le seul nœud à prendre potentiellement plus de deux ensembles de données en entrée.

Apparu avec PostgreSQL 14, le nœud *Memoize* est un cache de résultat qui permet d'optimiser les performances d'autres nœuds en mémorisant des données qui risquent d'être accédées plusieurs fois

de suite. Pour le moment, ce nœud n'est utilisable que pour les données de l'ensemble interne d'un *Nested Loop*.

3.6 OUTILS GRAPHIQUES

- pgAdmin
- explain.depesz.com
- explain.dalibo.com

L'analyse de plans complexes devient très vite fastidieuse. Nous n'avons vu ici que des plans d'une dizaine de lignes au maximum, mais les plans de requêtes réellement problématiques peuvent faire plusieurs centaines, voire milliers de lignes. L'analyse manuelle devient impossible. Des outils ont été créés pour mieux visualiser les parties intéressantes des plans.

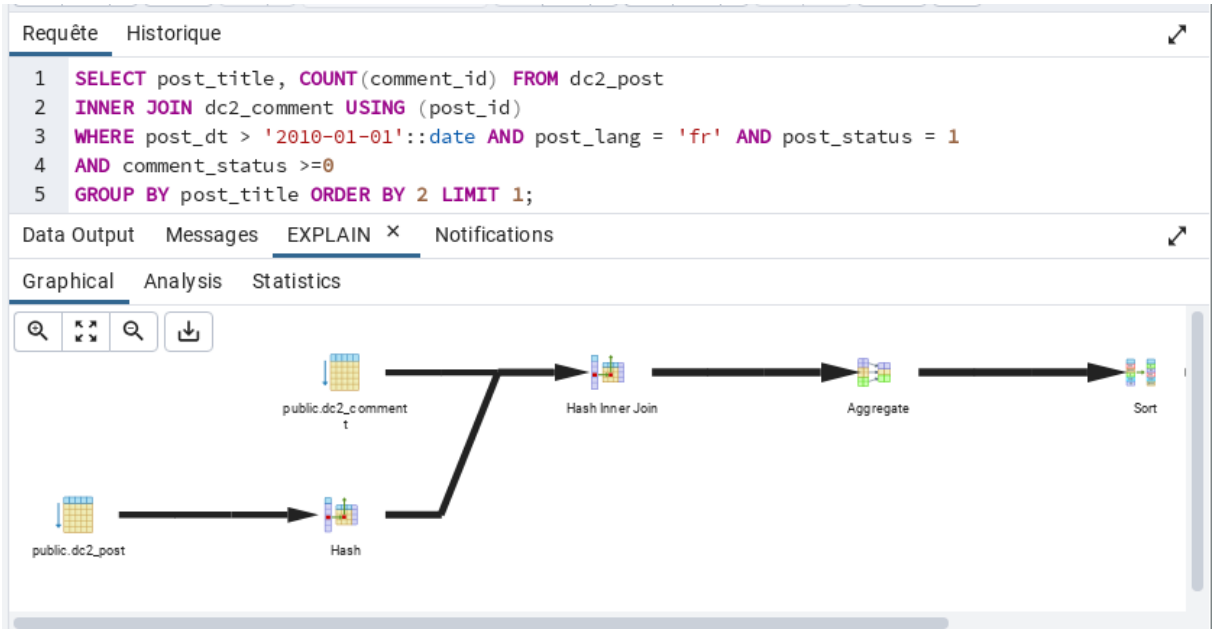
3.6.1 pgAdmin

- Vision graphique d'un `EXPLAIN`
- Une icône par nœud
- La taille des flèches dépend de la quantité de données
- Le détail de chaque nœud est affiché en survolant les nœuds

pgAdmin propose depuis très longtemps un affichage graphique de l'`EXPLAIN`. Cet affichage est intéressant car il montre simplement l'ordre dans lequel les opérations sont effectuées. Chaque nœud est représenté par une icône. Les flèches entre chaque nœud indiquent où sont envoyés les flux de données, la taille de la flèche précisant la volumétrie des données.

Les statistiques ne sont affichées qu'en survolant les nœuds.

3.6.2 pgAdmin - copie d'écran



Voici un exemple d'un `EXPLAIN` graphique réalisé par pgAdmin 4. En cliquant sur un nœud, un message affiche les informations statistiques sur le nœud.

3.6.3 explain.depesz.com

- Site web avec affichage amélioré du `EXPLAIN ANALYZE`
- Lignes colorées pour indiquer les problèmes
- 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 sur https://explain.depesz.com/

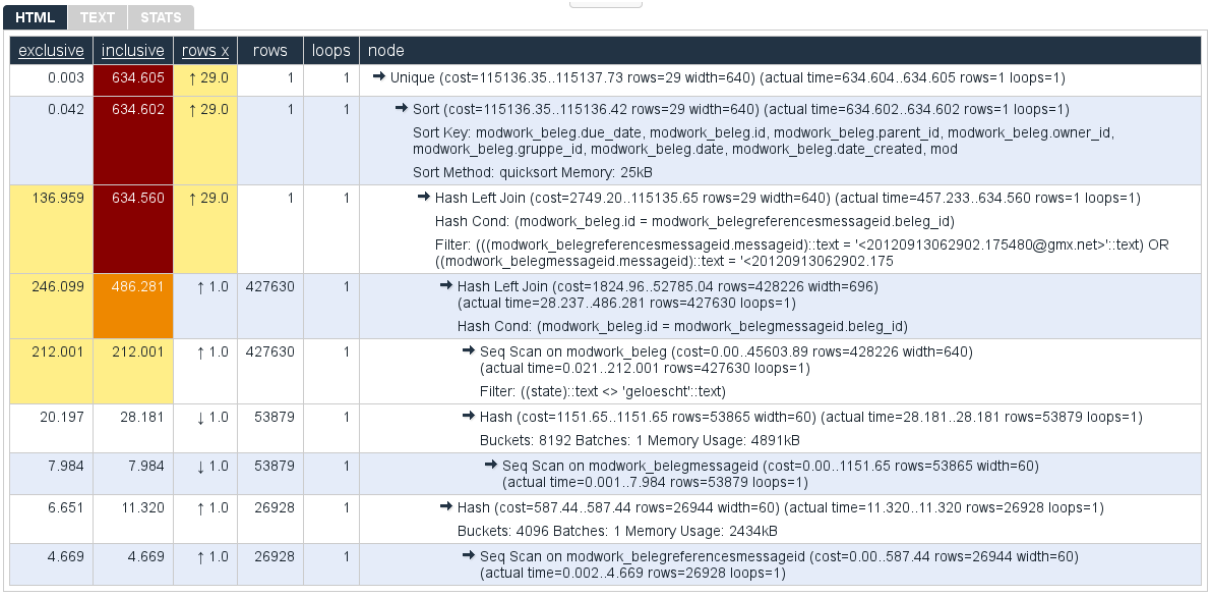
Il suffit d'aller sur ce site, de coller le résultat d'un `EXPLAIN ANALYZE`, et le site affichera le plan d'exécution avec des codes couleurs pour bien distinguer les nœuds performants des autres.

Le code couleur est simple : blanc indique que tout va bien, jaune est inquiétant, marron est plus inquiétant, et rouge très inquiétant.

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

- le module explain en Perl[4]
- la partie site web[5]

3.6.4 explain.depesz.com - exemple



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 l'affichage d'un plan sur le site explain.depesz.com[6].

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.

[4] https://gitlab.com/depesz/Pg--Explain
[5] https://gitlab.com/depesz/explain.depesz.com
[6] https://explain.depesz.com/

3.6.5 explain.dalibo.com

- Reprise de **pev** d'Alex Tatiyants, par Pierre Giraud (Dalibo)
- Page web avec affichage graphique d'un `EXPLAIN [ANALYZE]`
- Repérage des nœuds longs, lourds…
- Affichage flexible
- explain.dalibo.com
- Installable en local

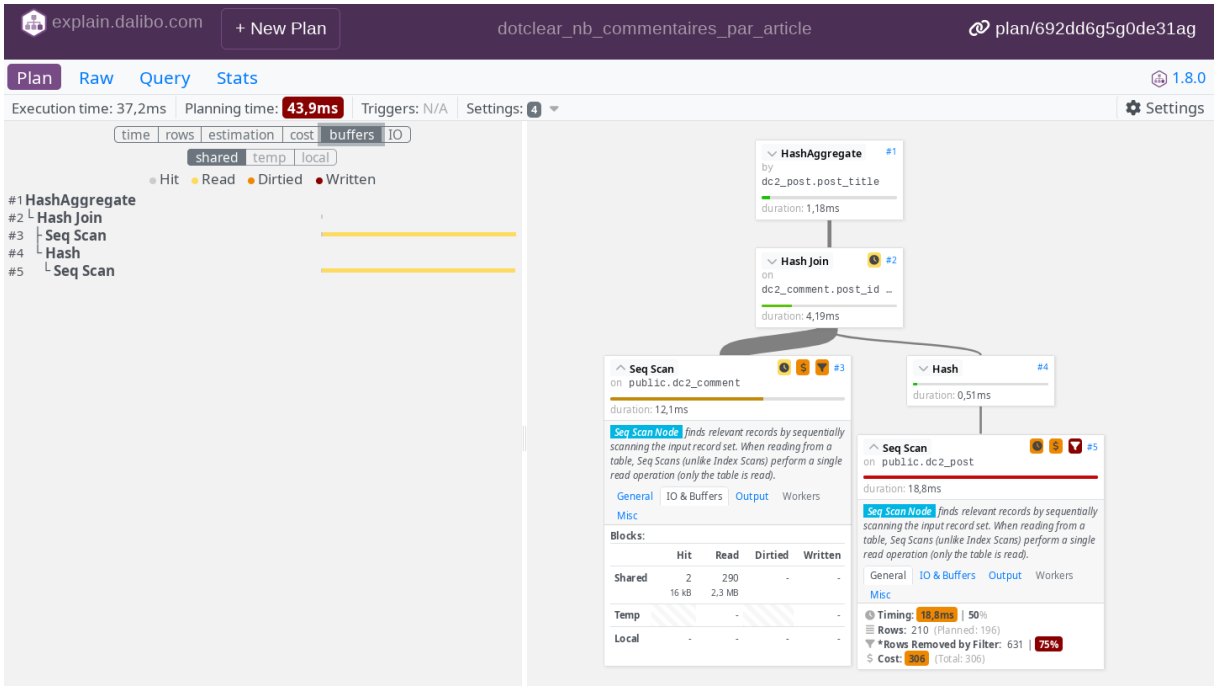
À l'origine, pev (*PostgreSQL Explain Visualizer*) est un outil libre[7] offrant un affichage graphique du plan d'exécution et pointant le nœud le plus coûteux, le plus long, le plus volumineux, etc. Utilisable en ligne[8], il n'est hélas plus maintenu depuis plusieurs années.

explain.dalibo.com[9] en est un *fork*, très étendu et activement maintenu par Pierre Giraud de Dalibo. Les plans au format texte comme JSON sont acceptés. Les versions récentes de PostgreSQL sont supportées, avec leurs spécificités : nouvelles options d'`EXPLAIN`, nouveaux types de nœuds… Tout se passe en ligne. Les plans peuvent être partagés. Si vous ne souhaitez pas qu'ils soient stockés chez Dalibo, utilisez la version strictement locale de pev2[10].

Le code[11] est sous licence PostgreSQL. Techniquement, c'est un composant VueJS qui peut être intégré à vos propres outils.

[7] https://github.com/AlexTatiyants/pev
[8] https://tatiyants.com/pev/#/plans
[9] https://explain.dalibo.com
[10] https://www.github.com/dalibo/pev2/releases/latest/download/index.html
[11] https://github.com/dalibo/pev2

3.6.6 explain.dalibo.com - exemple



explain.dalibo.com permet de repérer d'un coup d'œil les parties les plus longues du plan, celles utilisant le plus de lignes, les écarts d'estimation, les dérives du temps de planification… Les nœuds peuvent être repliés. Plusieurs modes d'affichage sont disponibles.

Un grand nombre de plans d'exemple sont disponibles sur le site.

3.7 CONCLUSION

- Un optimiseur très avancé
- Ne vous croyez pas plus malin que lui
- Mais il est important de savoir comment il fonctionne

Cette introduction à l'optimiseur de PostgreSQL permet de comprendre comment il fonctionne et sur quoi il se base. Cela permet de pointer certains des problèmes. C'est aussi un prérequis indispensable pour voir plus tard l'intérêt des différents index et nœuds d'exécution de PostgreSQL.

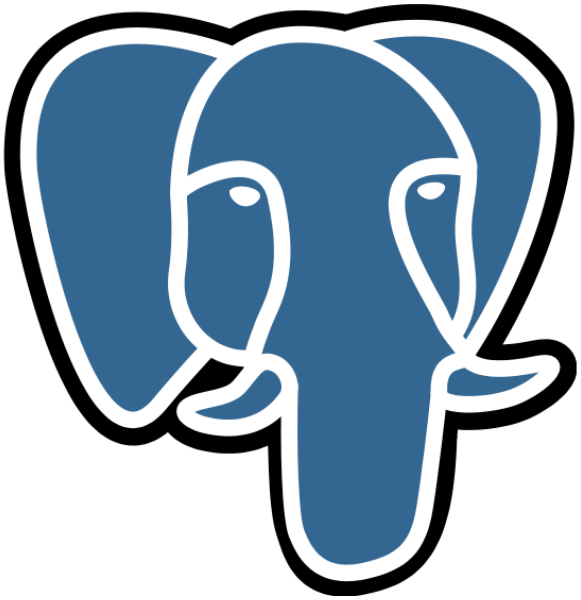
3.7.1 Questions

N'hésitez pas, c'est le moment !

3.8 QUIZ

https://dali.bo/j0_quiz

4/ PostgreSQL : Optimisations SQL



4.0.1 Introduction

L'optimisation doit porter sur :

- Le matériel
  - serveur, distribution, kernel, stockage, réseau…
- Le moteur de la base :
  - `postgresql.conf` & co
  - l'organisation des fichiers de PostgreSQL
- L'application
  - schéma, requêtes, vues…

Les bases de données sont des systèmes très complexes. Afin d'en tirer toutes les performances possibles, l'optimisation doit porter sur un très grand nombre de composants différents : le serveur qui héberge la base de données, et de manière générale tout l'environnement matériel, les processus faisant fonctionner la base de données, les fichiers et disques servant à son stockage, le partitionnement, mais aussi, et surtout, l'application elle-même. C'est sur ce dernier point que les gains sont habituellement les plus importants. (Les autres sont traités entre autres dans le module de formation J1[1].)

[1] https://dali.bo/j1_html

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

4.1 AXES D'OPTIMISATION

« 80% des effets sont produits par 20% des causes. » (Principe de Pareto)

- Il est illusoire d'optimiser une application sans connaître les sources de ralentissement
- Cibler l'optimisation :
  - trouver ces « 20% »
  - ne pas micro-optimiser ce qui n'a pas d'influence

Le principe de Pareto et la loi de Pareto sont des constats empiriques. On peut définir mathématiquement une distribution vérifiant le principe de Pareto[2].

> Le pourcentage de la population dont la richesse est supérieure à une valeur x est proportionnel à A/x^α »
>
> Vilfredo Pareto[a], économiste du XIXe siècle
>
> [a] https://fr.wikipedia.org/wiki/Vilfredo_Pareto

De nombreux phénomènes suivent cette distribution. Dans le cadre de PostgreSQL, cela se vérifie souvent. Une poignée de requêtes peut être responsable de l'essentiel du ralentissement. Une table mal conçue peut être responsable de la majorité de vos problèmes. Le temps de développement n'était pas infini, il ne sert à rien de passer beaucoup de temps à optimiser chaque paramètre sans avoir identifié les principaux consommateurs de ressource.

4.1.1 Quelles requêtes optimiser ?

Seul un certain nombre de requêtes sont critiques

- Identification (outil de profiling)
  - à optimiser prioritairement
- Différencier
  - longues en temps cumulé = coûteuses en ressources serveur
  - longues et interactives = mauvais ressenti des utilisateurs

Toutes les requêtes ne sont pas critiques, seul un certain nombre d'entre elles méritent une attention particulière. Il y a deux façon de déterminer les requêtes qui nécessitent d'être travaillées.

[2] https://fr.wikipedia.org/wiki/Principe_de_Pareto

La première dépend du ressenti utilisateur : un utilisateur devant son écran est notoirement impatient. Il faudra donc en priorité traiter les requêtes interactives. Certaines auront déjà d'excellents temps de réponse, d'autres pourront être améliorées.

L'autre méthode pour déterminer les requêtes à optimiser consiste à utiliser les outils de profiling habituels, dont nous allons voir quelques exemples. Ces outils permettent de déterminer les requêtes les plus fréquemment exécutées, et d'établir un classement des requêtes qui ont nécessité le plus de temps cumulé à leur exécution (voir l'onglet *Time consuming queries (N)* d'un rapport pgBadger). Les requêtes les plus fréquemment exécutées méritent également qu'on leur porte attention. Même si leur temps d'exécution cumulé est acceptable, leur optimisation peut permettre d'économiser quelques ressources du serveur, et une dérive peut avoir vite de gros impacts.

4.1.2 Recherche des axes d'optimisation

Quelques profilers :

- pgBadger[3]
- pg_stat_statements[4]
- PoWA[5]

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

pgBadger :

pgBadger est un analyseur de fichiers de traces PostgreSQL. Il nécessite de tracer dans les journaux applicatifs de PostgreSQL toutes les requêtes et leur durée. L'outil sait repérer des requêtes identiques avec des paramètres différents. Il les analyse et retourne les requêtes les plus fréquemment exécutées, les plus gourmandes unitairement, les plus gourmandes en temps cumulé (somme des temps unitaires).

Pour plus de détails, voir https://dali.bo/h1_html#pgbadger.

pg_stat_statements :

L'extension `pg_stat_statements`[6] est livrée avec PostgreSQL. Elle trace pour chaque ordre (même SQL, paramètres différents) exécuté sur l'instance son nombre d'exécutions, sa durée cumulée, et un certain nombre d'autres statistiques très utiles.

Si elle est présente, la requête suivante permet de déterminer les requêtes dont les temps d'exécution cumulés sont les plus importants :

```
SELECT r.rolname, d.datname, s.calls, s.total_exec_time,
       s.total_exec_time / s.calls AS avg_time, s.query
  FROM pg_stat_statements s
  JOIN pg_roles r     ON (s.userid=r.oid)
  JOIN pg_database d  ON (s.dbid = d.oid)
```

[6] https://dali.bo/x2_html#pg_stat_statements

```
 ORDER BY s.total_exec_time DESC
 LIMIT 10 ;
```

Et la requête suivante permet de déterminer les requêtes les plus fréquemment appelées :

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

Pour plus de détails sur les métriques relevées, voir https://dali.bo/h2_html#pg_stat_statements, et pour l'installation et des exemples de requêtes, voir https://dali.bo/x2_html#pg_stat_statements, ou encore la documentation officielle[7]

PoWA :

PoWA est une extension disponible dans les dépôts du PGDG, qui s'appuie sur `pg_stat_statements` pour 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.

[7] https://docs.postgresql.fr/current/pgstatstatements.html

4.2 SQL ET REQUÊTES

Le SQL :

- est un langage déclaratif :
  - on décrit le résultat, pas la façon de l'obtenir
  - c'est le travail de la base de déterminer le traitement à effectuer
- décrit un traitement ensembliste :
  - ≠ traitement procédural
  - « *on effectue des opérations sur des relations pour obtenir des relations* »
- est normalisé

Le langage SQL[8] a été normalisé par l'ANSI en 1986 et est devenu une norme ISO internationale en 1987. Il s'agit de la norme ISO 9075. Elle a subi plusieurs évolutions dans le but d'ajouter des fonctionnalités correspondant aux attentes de l'industrie logicielle. Parmi ces améliorations, notons l'intégration de quelques fonctionnalités objet pour le modèle relationnel-objet. La dernière version stable de la norme est SQL:2023[9] (juin 2023).

4.2.1 Opérateurs relationnels

Les opérateurs purement relationnels :

- Projection = `SELECT`
  - choix des colonnes
- Sélection = `WHERE`
  - choix des enregistrements
- Jointure = `FROM/JOIN`
  - choix des tables
- Bref : tout ce qui détermine sur quelles données on travaille

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

[8] https://fr.wikipedia.org/wiki/Structured_Query_Language
[9] https://en.wikipedia.org/wiki/SQL:2023

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

4.2.2 Opérateurs non-relationnels

Les autres opérateurs sont non-relationnels :

- `ORDER BY`
- `GROUP BY/DISTINCT`
- `HAVING`
- sous-requête, vue
- fonction (classique, d'agrégat, analytique)
- jointure externe

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

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

On doit exécuter la sous-requête avant la requête.

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

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

Il faut faire les jointures dans l'ordre indiqué : joindre `t1` à `t3` puis le résultat à `t2` pourrait ne pas amener le même résultat (un `LEFT JOIN` peut produire des `NULL`). Il est donc préférable de toujours mettre les jointures externes en fin de requête, sauf besoin précis : on laisse bien plus de liberté à l'optimiseur.

Le mot clé `DISTINCT` ne doit être utilisé qu'en dernière extrémité. On le rencontre très fréquemment dans des requêtes mal écrites qui produisent des doublons, afin de corriger le résultat — souvent en passant par un tri de l'ensemble du résultat, ce qui est coûteux.

4.2.3 Données utiles

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

- N'accéder qu'aux tables nécessaires
- N'accéder qu'aux colonnes nécessaires
  - viser *Index Only Scan*
  - se méfier : stockage TOAST
- Plus le volume de données à traiter est élevé, plus les opérations seront lentes :
  - tris et Jointures
  - éventuellement stockage temporaire sur disque pour certains algorithmes

Éviter les `SELECT *` :

C'est une bonne pratique, car la requête peut changer de résultat suite à un changement de schéma, ce qui risque d'entraîner des conséquences sur le reste du code.

Ne récupérer que les colonnes utilisées :

Dans beaucoup de cas, PostgreSQL sait repérer des colonnes qui figurent dans la requête mais sont finalement inutiles. Mais il n'est pas parfait. Surtout, il ne pourra pas repérer que vous n'avez pas réellement besoin d'une colonne issue d'une requête. En précisant uniquement les colonnes nécessaires, le moteur peut parfois utiliser des parcours plus simples, notamment des *Index Only Scans*. Il peut aussi éviter de lire les colonnes à gros contenu qui sont généralement déportés dans la partie TOAST[10] d'une table (des fichiers séparés de la table principale pour certains grands champs, transparents à l'utilisation mais dont l'accès n'est pas gratuit).

Éviter les jointures sur des tables inutiles :

Il n'y a que peu de cas où l'optimiseur peut supprimer de lui-même l'accès à une table inutile. Notamment, PostgreSQL le fait dans le cas d'un `LEFT JOIN` sur une table inutilisée dans le `SELECT`, au travers d'une clé étrangère, car on peut garantir que cette table est effectivement inutile.

4.2.4 Limiter le nombre de requêtes

SQL : langage ensembliste

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

[10] https://dali.bo/m4_html#mécanisme-toast

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

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

Préparons un jeu de test :

```
CREATE TABLE test (id int, valeur varchar);
INSERT INTO test SELECT i,chr(i%94+32) FROM generate_series (1,1000000) g(i);
ALTER TABLE test ADD PRIMARY KEY (id);
VACUUM ANALYZE test ;
```

Le script perl génère 1000 ordres pour récupérer des enregistrements un par un, avec une requête préparée pour être dans le cas le plus efficace :

```
#!/usr/bin/perl -w
print "PREPARE ps (int) AS SELECT * FROM test WHERE id=\$1;\n";
for (my $i=0;$i<=1000;$i++)
{
        print "EXECUTE ps($i);\n";
}
```

Exécutons ce code :

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

real    0m44,476s
user    0m0,137s
sys     0m0,040s
```

La durée totale de ces 1000 requêtes dépend fortement de la distance au serveur de base de données. Si le serveur est sur le même sous-réseau, on peut descendre à une seconde. Noter que l'établissement de la connexion au serveur n'a lieu qu'une fois.

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

```
time psql -c "SELECT * FROM test WHERE id BETWEEN 0 AND 1000" > /dev/null

real    0m0,129s
user    0m0,063s
sys     0m0,018s
```

Les allers-retours au serveur sont donc très coûteux dès qu'ils se cumulent.

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

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

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

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

Le SQL généré sera alors similaire à celui-ci :

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

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

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

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

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

Ceci générera une seule et unique requête du type :

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

4.2.5 Sous-requêtes dans un IN

Un *Semi Join* peut être très efficace (il ne lit pas tout)

```
SELECT * FROM t1
  WHERE val1  IN (  SELECT val2 …  )
```

- Sinon attention s'il y a beaucoup de valeurs dans la sous-requête !
  - dédoublonner :

```
SELECT * FROM t1
  WHERE val1 IN (  SELECT  DISTINCT  val2 …  )
```

  - surtout : réécriture avec `EXISTS` (si index disponible)

De la même manière que pour la clause `EXISTS`, un des intérêts du `IN` est de savoir quand il n'y a pas besoin de lire toute la sous-requête, comme ici, où seuls 5 blocs de la grande table `lots` sont lus, qui contiennent déjà toutes les valeurs possibles de `transporteur_id` :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT nom
FROM transporteurs
WHERE transporteur_id IN (
        SELECT transporteur_id
        FROM lots
        WHERE date_depot IS NOT NULL
        ) ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.00..19478.49 rows=5 width=12) (actual
↪  time=0.032..0.042 rows=5 loops=1)
   Join Filter: (transporteurs.transporteur_id = lots.transporteur_id)
   Rows Removed by Join Filter: 13
   Buffers: shared hit=5 read=1
   ->  Seq Scan on transporteurs  (cost=0.00..1.05 rows=5 width=20) (actual
↪  time=0.004..0.005 rows=5 loops=1)
         Buffers: shared hit=1
   ->  Seq Scan on lots  (cost=0.00..19476.04 rows=1006704 width=8) (actual
↪  time=0.005..0.006 rows=4 loops=5)
         Filter: (date_depot IS NOT NULL)
         Buffers: shared hit=4 read=1
 Planning:
   Buffers: shared hit=173 read=1
 Planning Time: 0.664 ms
 Execution Time: 0.069 ms
```

Mais la requête dans le `IN` peut être arbitrairement complexe, l'optimisation peut échouer et PostgreSQL peut basculer sur une forme de jointure plus lourde avec un regroupement des valeurs de la sous-requête :

```
EXPLAIN (ANALYZE,COSTS OFF)
SELECT nom
FROM transporteurs
WHERE transporteur_id IN (
        SELECT transporteur_id + 0  /* modification du critère */
        FROM lots
        WHERE date_depot IS NOT NULL
        ) ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Hash Join (actual time=139.280..139.283 rows=5 loops=1)
   Hash Cond: (transporteurs.transporteur_id = (lots.transporteur_id + 0))
   ->  Seq Scan on transporteurs (actual time=0.010..0.011 rows=5 loops=1)
   ->  Hash (actual time=139.265..139.265 rows=5 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  HashAggregate (actual time=139.259..139.261 rows=5 loops=1)
               Group Key: (lots.transporteur_id + 0)
               Batches: 1  Memory Usage: 24kB
               ->  Seq Scan on lots (actual time=0.010..70.184 rows=1006704 loops=1)
                     Filter: (date_depot IS NOT NULL)
 Planning Time: 0.201 ms
 Execution Time: 139.325 ms
```

Le dédoublonnage explicite au sein même de la sous-requête est alors parfois une bonne idée, même s'il vient forcément avec un certain coût :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT nom
FROM transporteurs
WHERE transporteur_id IN (
        SELECT   DISTINCT   transporteur_id + 0
        FROM lots
        WHERE date_depot IS NOT NULL
        ) ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Hash Join (actual time=46.385..48.943 rows=5 loops=1)
   Hash Cond: (transporteurs.transporteur_id = ((lots.transporteur_id + 0)))
   ->  Seq Scan on transporteurs (actual time=0.005..0.006 rows=5 loops=1)
   ->  Hash (actual time=46.375..48.931 rows=5 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Unique (actual time=46.367..48.925 rows=5 loops=1)
               ->  Sort (actual time=46.366..48.922 rows=15 loops=1)
                     Sort Key: ((lots.transporteur_id + 0))
                     Sort Method: quicksort  Memory: 25kB
                     ->  Gather (actual time=46.306..48.914 rows=15 loops=1)
                           Workers Planned: 2
                           Workers Launched: 2
                        ->  HashAggregate (actual time=44.707..44.708 rows=5 loops=3)
                              Group Key: (lots.transporteur_id + 0)
                              Batches: 1  Memory Usage: 24kB
                              Worker 0:  Batches: 1  Memory Usage: 24kB
                              Worker 1:  Batches: 1  Memory Usage: 24kB
                              ->  Parallel Seq Scan on lots (actual
↪  time=0.008..23.330 rows=335568 loops=3)
```

```
                                        Filter: (date_depot IS NOT NULL)
 Planning Time: 0.096 ms
 Execution Time: 48.979 ms
```

Mais on ne retrouve pas les performances de la première version.

Une autre possibilité est de réécrire la requête avec `EXISTS` mais cela n'a d'intérêt ici que si on peut indexer le champ calculé ; auquel cas les requêtes ci-dessus peuvent redevenir efficaces.

```
SELECT nom
FROM transporteurs t
WHERE EXISTS (
        SELECT 'ok'
        FROM lots l
        WHERE date_depot IS NOT NULL
        AND l.transporteur_id+0 /* à indexer */
                        = t.transporteur_id
        ) ;
```

4.2.6 Sous-requêtes liées

À éviter :

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

- un appel de fonction ou sous-requête par ligne !
- est-ce voulu ?
- transformer en clause `WHERE`
- penser à la clause `LATERAL`

Dans l'exemple ci-dessus, le résultat de la sous-requête dépend de la valeur `b` de chaque ligne de `t1`. On a donc autant d'appels, ce qui peut être une catastrophe. L'expérience montre que ce n'est parfois pas vraiment voulu…

Selon la complexité de la sous-requête, elle peut être réécrite en une simple clause `WHERE`. Il faut connaître aussi la clause `LATERAL`[] (https://docs.postgresql.fr/current/queries-table-expressions.html#QUERIES-LATERAL) dédiée à ce genre de chose et qui a au moins le mérite d'être lisible et explicite.

4.2.7 Sous-requêtes : équivalences IN/EXISTS/LEFT JOIN

Ces sous-requêtes sont strictement équivalentes (*Semi-join*) :

```
SELECT * FROM t1
WHERE fk IN ( SELECT pk FROM t2 WHERE … )

SELECT * FROM t1
WHERE EXISTS  ( SELECT 1 FROM t2 WHERE t2.pk = t1.fk AND … )

SELECT t1.*
FROM   t1   LEFT JOIN t2   ON (t1.fk=t2.pk)
WHERE
     t2.id IS NULL
```

(Et *Anti-join* pour les variantes avec `NOT`)

- Attention à `NOT IN` : préférer `NOT EXISTS`

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

Attention toutefois à l'utilisation du prédicat `NOT IN`, ils peuvent générer des plans d'exécution catastrophiques, avec une exécution de la sous-requête par ligne.

```
EXPLAIN SELECT *
FROM    commandes c
WHERE   numero_commande NOT IN (SELECT l.numero_commande
                                FROM   lignes_commandes l );

                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..22803529388.17 rows=500000 width=51)
   Workers Planned: 2
   ->  Parallel Seq Scan on commandes c  (cost=0.00..22803478388.17 rows=208333
↪  width=51)
         Filter: (NOT (SubPlan 1))
         SubPlan 1
           ->  Materialize  (cost=0.00..101602.11 rows=3141807 width=8)
                 ->  Seq Scan on lignes_commandes l  (cost=0.00..73620.07
↪  rows=3141807 width=8)
```

La raison est la suivante : si un `NULL` est présent dans la liste du `NOT IN`, `NOT IN` vaut systématiquement *false*. Or, nous savons qu'il n'y aura pas de `numero_commandes` à `NULL`. (Dans cette requête précise, PostgreSQL aurait pu le deviner car le champ `lignes_commandes.numero_commande` est `NOT NULL`, mais il n'en est pas encore capable.) Une réécriture avec `EXISTS` est strictement équivalente et produit un plan d'exécution largement plus intéressant avec un *Hash Right Anti Join* :

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

                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=24604.00..148053.15 rows=419708 width=51)
   Workers Planned: 2
   ->  Parallel Hash Right Anti Join  (cost=23604.00..105082.35 rows=174878 width=51)
         Hash Cond: (l.numero_commande = commandes.numero_commande)
         ->  Parallel Seq Scan on lignes_commandes l  (cost=0.00..55292.86
 ↪  rows=1309086 width=8)
         ->  Parallel Hash  (cost=14325.67..14325.67 rows=416667 width=51)
               ->  Parallel Seq Scan on commandes  (cost=0.00..14325.67 rows=416667
 ↪  width=51)
```

4.2.8 Les vues

Une vue est une requête pré-déclarée en base.

- Équivalent relationnel d'une fonction
- Si utilisée dans une autre requête, elle est traitée comme une sous-requête
- et *inlinée*
- Pas de problème si elle est relationnelle…

Les vues sont des requêtes dont le code peut être inclus dans une autre requête, comme s'il s'agissait d'une sous-requête. Les vues sont très pratiques en SQL et, en théorie, permettent de séparer le modèle physique (les tables) de ce que voient les développeurs, et donc de faire évoluer le modèle physique sans impact pour le développement. Elles sont surtout très pratiques pour rendre les requêtes plus lisibles, permettre la réutilisation de code SQL, et masquer la complexité à des utilisateurs peu avertis.

Dans le cas idéal, une vue reste relationnelle et donc ne contient que `SELECT`, `FROM` et `WHERE`, et elle peut être fusionnée avec le reste de la requête ; y compris avec des vues basées sur des vues. Ici les critères de deux vues imbriquées se retrouvent dans un même nœud :

```
CREATE OR REPLACE VIEW v_test_az
AS SELECT * FROM test
WHERE valeur BETWEEN 'A' AND 'Z' ;

CREATE OR REPLACE VIEW v_test_1000_az
AS SELECT * FROM v_test_az
WHERE id < 100 ;

EXPLAIN SELECT * FROM v_test_1000_az ;

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

```
 Index Scan using test_pkey on test  (cost=0.42..10.68 rows=54 width=6)
   Index Cond: (id < 100)
   Filter: (((valeur)::text >= 'A'::text) AND ((valeur)::text <= 'Z'::text))
```

4.2.9 Éviter les vues non-relationnelles

- Attention aux vues avec `DISTINCT`, `GROUP BY` etc.
  - tous les problèmes des sous-requêtes déjà vus
  - impossible de l'*inliner*
  - barrière d'optimisation
  - …et mauvaises performances
- Les vues sont dangereuses en termes de performance
  - masquent la complexité
- Penser aux vues matérialisées si la requête est lourde

En pratique, les vues sont souvent sources de ralentissement : elles masquent la complexité, et l'utilisateur crée alors sans le savoir des requêtes très complexes, mettant en jeu des dizaines de tables (voire des dizaines de fois les **mêmes** tables !).

Avant d'utiliser une vue, il faut s'intéresser un peu à son contenu et ce qu'elle fait.

On retrouve aussi toute la complexité liée aux sous-requêtes, puisqu'une vue en est l'équivalent. En particulier, il faut se méfier des vues contenant des opérations non-relationnelles, qui peuvent empêcher de nombreuses optimisations. En voici un exemple simple :

```
CREATE OR REPLACE VIEW v_test_did
AS SELECT DISTINCT ON (id) id,valeur FROM test ;

EXPLAIN (ANALYZE, COSTS OFF)
  SELECT id,valeur
  FROM v_test_did
  WHERE valeur='b' ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Subquery Scan on v_test_did (actual time=0.070..203.584 rows=10638 loops=1)
   Filter: ((v_test_did.valeur)::text = 'b'::text)
   Rows Removed by Filter: 989362
   ->  Unique (actual time=0.017..158.458 rows=1000000 loops=1)
         ->  Index Scan using test_pkey on test (actual time=0.015..98.200
↪  rows=1000000 loops=1)
 Planning Time: 0.202 ms
 Execution Time: 203.872 ms
```

On constate que la condition de filtrage sur `b` n'est appliquée qu'à la fin. C'est normal : à cause du `DISTINCT ON`, l'optimiseur ne peut pas savoir si l'enregistrement qui sera retenu dans la sous-requête vérifiera `valeur='b'` ou pas, et doit donc attendre l'étape suivante pour filtrer. Le coût en performances, même avec un volume de données raisonnable, peut être astronomique.

4.3 ACCÈS AUX DONNÉES

L'accès aux données est coûteux.

- Quelle que soit la base
- Dialogue entre client et serveur
  - plusieurs aller/retours potentiellement
- Analyse d'un langage complexe
  - SQL PostgreSQL : `gram.y` de 19000 lignes
- Calcul de plan :
  - langage déclaratif => converti en impératif à chaque exécution

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

`SELECT * FROM test`, 0 enregistrement :

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

`SELECT * FROM test`, 1000 enregistrements :

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

Un client JDBC va habituellement utiliser un aller/retour de plus, en raison des requêtes préparées : un dialogue pour envoyer la requête et la préparer, et un autre pour envoyer les paramètres. Le problème est la latence du réseau, habituellement : de 50 à 300 µs. Cela limite à 3 000 à 20 000 le nombre

d'opérations maximum par seconde par socket. On peut bien sûr paralléliser sur plusieurs sessions, mais cela complique le traitement.

En ce qui concerne le parser : comme indiqué dans ce message[11] : `gram.o`, le parser fait 1 Mo une fois compilé !

4.3.1 Coût des connexions

Se connecter coûte cher :

- Authentification, permissions
- Latence réseau
- Négociation SSL
- Création de processus & contexte d'exécution
- Acquisition de verrous

→ Maintenir les connexions côté applicatif, ou utiliser un pooler.

L'établissement d'une connexion client au serveur est coûteuse, en temps comme ressource. Il faut plusieurs allers-retours réseau pour établir la connexion. Il y a souvent une négociation pour le chiffrement SSL. PostgreSQL doit authentifier l'utilisateur, puis créer son processus, le contexte d'exécution, poser quelques verrous, avant que les requêtes puissent arriver. Pour des requêtes répétées, il est beaucoup plus efficace d'ouvrir une connexion et de la réutiliser pour de nombreuses requêtes.

On peut tester l'impact d'une connexion/déconnexion avec `pgbench`, dont l'option `-C` lui demande de se connecter à chaque requête :

```
$ pgbench pgbench -T 20 -c 10 -j5 -S -C

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

Sans se reconnecter à chaque requête :

```
$ pgbench pgbench -T 20 -c 10 -j5 -S
```

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

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

On passe de 900 à 38 000 transactions par seconde.

Un pooler est souvent intégré d'office aux serveurs d'applications (par exemple Tomcat). Sinon, PgBouncer[12] est l'outil généralement recommandé. Pour les détails, voir notre module de formation W6[13].

4.3.2 Penser relationnel

- Les spécifications sont souvent procédurales, voire objet !
- Prendre du recul
- Réfléchir de façon ensembliste
  - on travaille sur des ensembles de données
  - penser aux CTE (`WITH`)

Si les spécifications disent (version simplifiée bien sûr) :

- vérifier la présence du client ;
- s'il est présent, mettre à jour son adresse ;
- sinon, créer le client avec la bonne adresse,

on peut être tenter d'écrire (pseudo-code client) :

```
SELECT count(*) from clients where client_name = 'xxx'
INTO compte

IF compte > 0
  UPDATE clients set adresse='yyy' WHERE client_name='xxx'
ELSE
  INSERT client SET client_name='xxx', adresse='yyy'
END IF
```

D'où 3 requêtes, systématiquement 2 appels à la base. On peut facilement économiser une requête :

[12] https://www.pgbouncer.org/
[13] https://dali.bo/w6_html

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

Les versions modernes de PostgreSQL permettent de tout faire en un seul ordre. L'exemple suivant utilise une fusion des enregistrements dans PostgreSQL avec des CTE[14].

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

Dans beaucoup de cas on peut faire encore plus simple grâce à la clause `ON CONFLICT … DO UPDATE` (« upsert[15] ») :

```
INSERT INTO client (nom_client,adresse) VALUES ('toto' , 'adresse1' ),
↪  ('tata','adresse2')
    ON CONFLICT (nom_client) DO UPDATE
    SET adresse = EXCLUDED.adresse
    WHERE client.nom_client = EXCLUDED.nom_client;
```

PostgreSQL 15 apporte même une commande MERGE[16]. Par rapport à `INSERT ON CONFLICT`, `MERGE` permet aussi des suppressions, et possède un mécanisme différent.

[14] https://vibhorkumar.wordpress.com/2011/10/26/upsertmerge-using-writable-cte-in-postgresql-9-1/
[15] https://docs.postgresql.fr/16/sql-insert.html
[16] https://public.dalibo.com/exports/formation/workshops/fr/ws15/150-postgresql_15.handout.html#ajout-de-la-commande-sql-merge

4.3.3 Pas de DDL applicatif

Le schéma est la modélisation des données

- Une application n'a pas à y toucher lors de son fonctionnement normal + exception : tables temporaires
- SQL manipule les données en flux continu :
  - chaque étape d'un plan d'exécution n'attend pas la fin de la précédente
  - donc : une table temporaire est souvent une perte de temps

Il est fortement déconseillé qu'une application modifie le schéma de données pendant son fonctionnement, notamment qu'elle crée des tables ou ajoute des colonnes. Une exception fréquente concerne les tables « temporaires », qui n'existent que le temps d'une session. Elles sont inévitables dans certaines circonstances, assez courantes pendant des batchs, mais dans le flux normal de l'applicatif l'utilisation de tables temporaires ne sert généralement qu'à multiplier les étapes et à poser des sortes de points d'arrêt artificiels dans le maniement des données. C'est très net sur cette réécriture de l'exemple précédent :

```
CREATE TEMP TABLE temp_a_inserer  (nom_client text, adresse text);

INSERT INTO temp_a_inserer
  SELECT * FROM (VALUES ('toto' , 'adresse1' ), ('tata','adresse2')) AS tmp;

UPDATE client SET adresse=temp_a_inserer.adresse
FROM temp_a_inserer
WHERE temp_a_inserer.nom_client=client.nom_client;

INSERT INTO client (nom_client,adresse)
SELECT nom_client,adresse from temp_a_inserer
WHERE NOT EXISTS (
          SELECT 1 FROM client
          WHERE client.nom_client=temp_a_inserer.nom_client
          );
DROP TABLE temp_a_inserer;
```

4.3.4 Optimiser chaque accès

Les moteurs SQL sont très efficaces, et évoluent en permanence

- Ils ont de nombreuses méthodes de tri, de jointure, choisies en fonction du contexte
- En SQL :
  - optimisation selon volume & configuration
  - évolution avec le moteur
- Dans l'application cliente : vous devrez le maintenir et l'améliorer
- Faites le maximum côté SQL :
  - agrégats, fonctions analytiques, tris, numérotations, `CASE`, etc.
  - Commentez avec `--` et `/* */`

L'avantage du code SQL est, encore une fois, qu'il est déclaratif. Il aura donc de nombreux avantages sur un code procédural côté applicatif, quel que soit le langage. L'exécution par le moteur évoluera pour prendre en compte les variations de volumétrie des différentes tables. Les optimiseurs sont la partie la plus importante d'un moteur SQL. Ils progressent en permanence. Chaque nouvelle version va donc potentiellement améliorer vos performances.

Si vous écrivez du procédural avec des appels unitaires à la base dans des boucles, le moteur ne pourra rien optimiser. Si vous faites vos tris ou regroupements côté client, vous êtes limités aux algorithmes fournis par vos langages, voire à ceux que vous aurez écrit manuellement à un moment donné. Alors qu'une base de données bascule automatiquement entre une dizaine d'algorithmes différents suivant le volume, le type de données à trier, ce pour quoi le tri est ensuite utilisé, etc., voire évite de trier en utilisant des tables de hachage ou des index disponibles. La migration à une nouvelle version du moteur peut vous apporter d'autres techniques prises alors en compte de manière transparente.

4.3.5 Ne faire que le nécessaire

Prendre de la distance vis-à-vis des spécifications fonctionnelles (bis) :

- Ex : mise à jour ou insertion ?
  - tenter la mise à jour, et regarder combien d'enregistrements ont été mis à jour
  - surtout pas de `COUNT(*)`
  - éventuellement un test de l'existence d'un seul enregistrement
  - gérer les exceptions plutôt que de vérifier préalablement que les conditions sont remplies (si l'exception est rare)
  - et se renseigner sur la syntaxe

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

4.4 INDEX

- Objets destinés à l'optimisation des accès
- À poser par les développeurs :

```
CREATE INDEX ON ma_table (nom colonne) ;
```

Les index sont des objets uniquement destinés à accélérer les requêtes (filtrage mais aussi jointures et tris, ou respect des contraintes d'unicité). Ils ne modifient jamais le résultat d'une requête (tout au plus : ils peuvent changer l'ordre des lignes résultantes si celui-ci est indéfini.) Il est capital pour un développeur d'en maîtriser les bases, car il est celui qui sait quels sont les champs interrogés dans son application. Les index sont un sujet en soi qui sera traité par ailleurs.

4.5 IMPACT DES TRANSACTIONS

- Verrous : relâchés à la **fin** de la transaction
  - `COMMIT`
  - `ROLLBACK`
- Validation des données sur le disque au `COMMIT`
  - écriture synchrone : coûteux
- Contournements :
  - tables temporaires/unlogged ?
  - parfois : `synchronous_commit = off` (…si perte possible)
- → Faire des transactions qui correspondent au fonctionnel
  - pas trop nombreuses
  - courtes, pas de travail inutile une fois des verrous posés

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

L'opération `COMMIT` a bien sûr un coût : il faut garantir que les données sont bien écrites sur le disque, il faut les écrire sur le disque (évidemment), mais aussi attendre la confirmation du disque, voire de serveurs répliqués distants parfois. Même avec des disques SSD, plus performants que les disques classiques, cette opération reste coûteuse. Un disque dur classique doit attendre la rotation du disque et placer sa tête au bon endroit (dans le journal de transaction), écrire la donnée, et confirmer au système que c'est fait. Un disque SSD doit écrire réellement le bloc demandé, c'est-à-dire l'effacer (relativement lent) puis le réécrire. Dans les deux cas, il faut compter de l'ordre de la milliseconde.

Attention aux caches en écriture des disques ou cartes RAID : son contenu ne doit pas disparaître en cas de panne de courant. Cela dépend des disques…

Il est parfois acceptable de perdre les dernières données en cas de panne de courant (par défaut, celles committées pendant les derniers 300 ms). On peut donc réduire la fréquence de la synchronisation des journaux avec :

```
SET synchronous_commit TO off ;          /* pour une session */
SET LOCAL synchronous_commit TO off ;    /* pour une transaction */
```

La synchronisation n'aura plus lieu aussi fréquemment. L'impact peut être énorme sur les petites transactions nombreuses.

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

- par le temps d'acquisition des verrous, bien sûr ;
- par la contention entre sessions : certaines risquent d'en bloquer d'autres.

Les verrous étant posés lors des ordres d'écriture et relâchés en fin de transaction, on fera en sorte que la transaction fasse le minimum de choses après les premières écritures.

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

4.5.1 Verrouillage et contention

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

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

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

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

Dans le cas où il est réellement nécessaire de verrouiller un enregistrement sans le mettre à jour immédiatement (pour éviter une mise à jour concurrente), il faut utiliser l'ordre SQL `SELECT FOR UPDATE`. Ce dernier possède une très intéressante option `SKIP LOCKED`[17] pour ne pas être bloqué par une ligne déjà verrouillée.

4.5.2 Deadlocks

« Verrous mortels » : comment les éviter ?

- Théorie : prendre toujours les verrous dans le même ordre
- Pratique, ça n'est pas toujours possible ou commode
- Conséquence : une des transactions est tuée
  - erreurs, ralentissements

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

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

[17] https://docs.postgresql.fr/16/sql-select.html#SQL-FOR-UPDATE-SHARE

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

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

- les transactions, mêmes celles qui réussissent, attendent beaucoup (ce qui bride l'utilisation de toutes les ressources machine) ;
- certaines finissent annulées et doivent donc être rejouées (travail supplémentaire).

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

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

4.6 BASE DISTRIBUÉE

Écrire sur plusieurs nœuds ?

- Complexité (applicatif/exploitation)
  - → risque d'erreur (programmation, fausse manipulation)
  - reprise d'incident complexe
- Essayez avec un seul serveur plus gros
  - après avoir optimisé bien sûr
  - PostgreSQL peut vous étonner

Il y a plusieurs variantes de l'utilisation de plusieurs nœuds. PostgreSQL permet nativement la lecture sur des réplicas d'une instance (et la technologie est fiable), mais pas l'écriture. Il est cependant possible de faire du *sharding* en répartissant les données sur plusieurs instances indépendantes (*sharding*). L'extension Citus[18] permet de faire cela de manière transparente. Mais il ne faut surtout pas négliger tous les coûts de cette solution : non seulement le coût du matériel, mais aussi les coûts humains : procédures d'exploitation, de maintenance, complexité accrue de développement, etc.

Performance et robustesse peuvent être des objectifs contradictoires.

Avant de complexifier votre système, pensez à augmenter les ressources du serveur (après avoir optimisé autant que possible et identifié quelle ressource pose problème : RAM, CPU, disque… ?)

[18] https://www.citusdata.com/product

4.7 BIBLIOGRAPHIE

- Quelques références :
  - *The Art of SQL*, **Stéphane Faroult**
  - *Refactoring SQL Applications*, **Stéphane Faroult**
  - *SQL Performance Explained*, **Markus Winand**
  - *Introduction aux bases de données*, **Chris Date**
  - *The Art of PostgreSQL*, **Dimitri Fontaine**
  - Vidéos de **Stéphane Faroult** sous Youtube

Il existe bien des livres sur le développement en SQL. Voici quelques sources intéressantes parmi bien d'autres :

Livres pratiques non propres à PostgreSQL :

- *The Art of SQL*, **Stéphane Faroult**, 2006 (ISBN-13: 978-0596008949)
- *Refactoring SQL Applications*, **Stéphane Faroult**, 2008 (ISBN-13: 978-0596514976)
- *SQL Performance Explained*, **Markus Winand**, 2012 : même si ce livre ne tient pas compte des dernières nouveautés des index de PostgreSQL, il contient l'essentiel de ce qu'un développeur doit savoir sur les index B-tree sur diverses bases de données courantes
  - site internet (fr)[19]
  - ISBN-13 en français : 978-3950307832, en anglais : 978-3950307825

En vidéos :

- **Stéphane Faroult** (roughsealtd sur Youtube[20]) : *SQL Best Practices in less than 20 minutes* partie 1[21], partie 2[22], partie 3[23].

Livres spécifiques à PostgreSQL :

- *The Art of PostgreSQL*[24] de **Dimitri Fontaine** (2020) ; l'ancienne édition de 2017 se nommait *Mastering PostgreSQL in Application Development*.

Sur la théorie des bases de données :

- *An Introduction to Database Systems*, **Chris Date** (8è édition de 2003, ISBN-13 en français : 978-2711748389 ; en anglais : 978-0321197849) ;
- *The World and the Machine*, **Michael Jackson** (version en ligne[25] .

[19] https://use-the-index-luke.com/fr
[20] https://www.youtube.com/channel/UCW6zsYGFckfczPKUUVdvYjg
[21] https://www.youtube.com/watch?v=40Lnoyv-sXg&list=PL767434BC92D459A7
[22] https://www.youtube.com/watch?v=GbZgnAINjUw&list=PL767434BC92D459A7
[23] https://www.youtube.com/watch?v=y70FmugnhPU&list=PL767434BC92D459A7
[24] https://theartofpostgresql.com/
[25] https://dl.acm.org/doi/10.1145/225014.225041

4.8 QUIZ

https://dali.bo/j3_quiz

5/ Techniques d’indexation

Photo de Maksym Kaharlytskyi[1], Unsplash licence

[1] https://unsplash.com/@qwitka

5.1 INTRODUCTION

- Qu'est-ce qu'un index ?
- Comment indexer une base ?
- Les index B-tree dans PostgreSQL

5.1.1 Objectifs

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

5.1.2 Introduction aux index

- Uniquement destinés à l'optimisation
- À gérer d'abord par le développeur
    - **Markus Winand** : *SQL Performance Explained*

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

La création des index est à la charge du développeur ou du DBA, leur création n'est pas automatique, sauf exception.

Pour Markus Winand, c'est d'abord au développeur de poser les index, car c'est lui qui sait comment ses données sont utilisées. Un DBA d'exploitation n'a pas cette connaissance, mais il connaît généralement mieux les différents types d'index et leurs subtilités, et voit comment les requêtes réagissent en production. Développeur et DBA sont complémentaires dans l'analyse d'un problème de performance.

Le site de Markus Winand, Use the index, Luke[2], propose une version en ligne de son livre *SQL Performance Explained*, centré sur les index B-tree (les plus courants). Une version française est par ailleurs disponible sous le titre *SQL : au cœur des performances*.

[2] https://use-the-index-luke.com

5.1.3 Utilité d'un index

- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire tout récupérer dans l'index (*Index Only Scan*)
- Un index facilite :
  - certains tris
  - certains agrégats
- Obligatoires et automatique pour clés primaires & unicité
  - conseillé pour clés étrangères (FK)

Les index ne changent pas le résultat d'une requête, mais l'accélèrent. L'index permet de pointer l'endroit de la table où se trouve une donnée, pour y accéder directement. Parfois c'est toute une plage de l'index, voire sa totalité, qui sera lue, ce qui est généralement plus rapide que lire toute la table.

Le cas le plus favorable est l'*Index Only Scan* : toutes les données nécessaires sont contenues dans l'index, lui seul sera lu et PostgreSQL ne lira pas la table elle-même.

PostgreSQL propose différentes formes d'index :

- index classique sur une seule colonne d'une table ;
- index composite sur plusieurs colonnes d'une table ;
- index partiel, en restreignant les données indexées avec une clause `WHERE` ;
- index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table ;
- index couvrants, contenant plus de champs que nécessaire au filtrage, pour ne pas avoir besoin de lire la table, et obtenir un *Index Only Scan*.

La création des index est à la charge du développeur. Seules exceptions : ceux créés automatiquement quand on déclare des contraintes de clé primaire ou d'unicité. La création est alors automatique.

Les contraintes de clé étrangère imposent qu'il existe déjà une clé primaire sur la table pointée, mais ne crée pas d'index sur la table portant la clé.

5.1.4 Index et lectures

Un index améliore les `SELECT`

- Sans index :

```
=# SELECT * FROM test WHERE id = 10000;
Temps : 1760,017 ms
```

- Avec index :

```
=# CREATE INDEX idx_test_id ON test (id);

=# SELECT * FROM test WHERE id = 10000;
Temps : 27,711 ms
```

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

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

```
EXPLAIN SELECT * FROM test WHERE id = 10000;

                            QUERY PLAN
-------------------------------------------------------------------
 Gather  (cost=1000.00..193661.66 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on test  (cost=0.00..192661.56 rows=1 width=4)
         Filter: (id = 10000)
```

Lorsqu'il est présent, PostgreSQL l'utilise car l'optimiseur estime que son parcours ne récupérera qu'une seule ligne sur les 20 millions que compte la table :

```
EXPLAIN SELECT * FROM test WHERE id = 10000;

                             QUERY PLAN
---------------------------------------------------------------------
 Index Only Scan using idx_test_id on test  (cost=0.44..8.46 rows=1 width=4)
   Index Cond: (id = 10000)
```

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

```
EXPLAIN SELECT id FROM test
        WHERE id BETWEEN 1000 AND 1200 ORDER BY id DESC;
```

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Only Scan Backward using idx_test_id on test
                                                (cost=0.44..12.26 rows=191 width=4)
   Index Cond: ((id >= 1000) AND (id <= 1200))
```

5.1.5 Index : inconvénients

- L'index n'est pas gratuit !
- Ralentit les écritures
  - maintenance
- Place disque
- Compromis à trouver

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

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

Soit une table `test2` telle que :

```
CREATE TABLE test2 (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    valeur INTEGER,
    commentaire TEXT
);
```

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

```
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 35253,228 ms (00:35,253)
```

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

```
CREATE INDEX idx_test2_valeur ON test2 (valeur);
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 44410,775 ms (00:44,411)
```

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

```
CREATE INDEX idx_test2_commentaire ON test2 (commentaire);
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 207075,335 ms (03:27,075)
```

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

```
CREATE TABLE test3 AS SELECT * FROM test2;
INSERT INTO test3 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
```

```
INSERT 0 10000000
Durée : 14758,503 ms (00:14,759)
```

La table `test2` a été vidée préalablement pour chaque test.

Enfin, la place disque utilisée par ces index n'est pas négligeable :

```
\di+ *test2*
                                 Liste des relations
 Schéma |          Nom           | Type  | Propriétaire | Table | Taille |  …
--------+------------------------+-------+--------------+-------+--------+-
 public | idx_test2_commentaire  | index | postgres     | test2 | 387 MB |
 public | idx_test2_valeur       | index | postgres     | test2 | 214 MB |
 public | test2_pkey             | index | postgres     | test2 | 214 MB |
```

```
SELECT pg_size_pretty(pg_relation_size('test2')),
       pg_size_pretty(pg_indexes_size('test2')) ;
```

```
 pg_size_pretty | pg_size_pretty
----------------+----------------
 574 MB         | 816 MB
```

Pour ces raisons, on ne posera pas des index systématiquement avant de se demander s'ils seront utilisés. L'idéal est d'étudier les plans de ses requêtes et de chercher à optimiser.

5.1.6 Index : contraintes pratiques à la création

- Lourd…

```
-- bloque les écritures !
CREATE INDEX ON matable ( macolonne ) ;
-- ne bloque pas, peut échouer
CREATE INDEX CONCURRENTLY ON matable ( macolonne ) ;
```

- Si fragmentation :

```
REINDEX INDEX nomindex ;
REINDEX TABLE CONCURRENTLY nomtable ;
```

- Paramètres :
  - `maintenance_work_mem` (sinon : fichier temporaire !)
  - `max_parallel_maintenance_workers`

Création d'un index :

Bien sûr, la durée de création de l'index dépend fortement de la taille de la table. PostgreSQL va lire toutes les lignes et trier les valeurs rencontrées. Ce peut être lourd et impliquer la création de fichiers temporaires.

Si l'on utilise la syntaxe classique, toutes les écritures sur la table sont bloquées (mises en attente) pendant la durée de la création de l'index (verrou *ShareLock*). Les lectures restent possibles, mais cette contrainte est parfois rédhibitoire pour les grosses tables.

Clause CONCURRENTLY :

Ajouter le mot clé `CONCURRENTLY` permet de rendre la table accessible en écriture. Malheureusement, cela nécessite au minimum deux parcours de la table, et donc alourdit et ralentit la construction de l'index. Dans quelques cas défavorables (entre autres l'interruption de la création de l'index), la création échoue et l'index existe mais est invalide :

```
pgbench=# \d pgbench_accounts
                Table « public.pgbench_accounts »
 Colonne  |     Type      | Collationnement | NULL-able | Par défaut
----------+---------------+-----------------+-----------+------------
 aid      | integer       |                 | not null  |
 bid      | integer       |                 |           |
 abalance | integer       |                 |           |
 filler   | character(84) |                 |           |
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
    "pgbench_accounts_bid_idx" btree (bid) INVALID
```

L'index est inutilisable et doit être supprimé et recréé, ou bien réindexé. Pour les détails, voir la documentation officielle[3].

Une supervision peut détecter des index invalides avec cette requête, qui ne doit jamais rien ramener :

```
SELECT indexrelid::regclass AS index, indrelid::regclass AS table
FROM pg_index
WHERE indisvalid = false ;
```

Réindexation :

Comme les tables, les index sont soumis à la fragmentation. Celle-ci peut cependant monter assez haut sans grande conséquence pour les performances. De plus, le nettoyage des index est une des étapes des opérations de VACUUM[4].

Une reconstruction de l'index est automatique lors d'un `VACUUM FULL` de la table.

Certaines charges provoquent une fragmentation assez élevée, typiquement les tables gérant des files d'attente. Une réindexation reconstruit totalement l'index. Voici quelques variantes de l'ordre :

```
REINDEX INDEX pgbench_accounts_bid_idx ;  -- un seul index
REINDEX TABLE pgbench_accounts ;          -- tous les index de la table
REINDEX (VERBOSE) DATABASE pgbench ;      -- tous ceux de la base, avec détails
```

Il existe là aussi une clause `CONCURRENTLY` :

```
REINDEX (VERBOSE) INDEX CONCURRENTLY pgbench_accounts_bid_idx ;
```

(En cas d'échec, on trouvera là aussi des index invalides, suffixés avec `_ccnew`, à côté des index préexistants toujours fonctionnels et que PostgreSQL n'a pas détruits.)

Paramètres :

La rapidité de création d'un index dépend essentiellement de la mémoire accordée, définie dans `maintenance_work_mem`. Si elle ne suffit pas, le tri se fera dans des fichiers temporaires plus lents. Sur les serveurs modernes, le défaut de 64 Mo est ridicule, et on peut monter aisément à :

```
SET maintenance_work_mem = '2GB' ;
```

Attention de ne pas saturer la mémoire en cas de création simultanée de nombreux gros index (lors d'une restauration avec `pg_restore` notamment).

Si le serveur est bien doté en CPU, la parallélisation de la création d'index peut apporter un gain en temps appréciable. La valeur par défaut est :

```
SET max_parallel_maintenance_workers = 2 ;
```

et devrait même être baissée sur les plus petites configurations.

[3] https://docs.postgresql.fr/current/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY
[4] https://dali.bo/m5_html#fonctionnement-de-vacuum

5.1.7 Types d'index dans PostgreSQL

- Défaut : B-tree classique (équilibré)
- `UNIQUE` (préférer la contrainte)
- Mais aussi multicolonne, fonctionnel, partiel, couvrant
- Index spécialisés : hash, GiST, GIN, BRIN, HNSW….

Par défaut un `CREATE INDEX` créera un index de type B-tree, de loin le plus courant. Il est stocké sous forme d'arbre équilibré, avec de nombreux avantages :

- les performances se dégradent peu avec la taille de l'arbre (les temps de recherche sont en O(log(n)), donc fonction du logarithme du nombre d'enregistrements dans l'index) ;
- l'accès concurrent est excellent, avec très peu de contention entre processus qui insèrent simultanément.

Toutefois les B-tree ne permettent de répondre qu'à des questions très simples, portant sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre tout de même la majorité des cas.

Contrainte d'unicité et index :

Un index peut être déclaré `UNIQUE` pour provoquer une erreur en cas d'insertion de doublons. Mais on préférera généralement déclarer une *contrainte* d'unicité (notion fonctionnelle), qui techniquement, entraînera la création d'un index.

Par exemple, sur cette table `personne` :

```
$ CREATE TABLE personne (id int, nom text);

$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
```

on peut créer un index unique :

```
$ CREATE UNIQUE INDEX ON personne (id);

$ \d personne
                 Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "personne_id_idx" UNIQUE, btree (id)
```

La contrainte d'unicité est alors implicite. La suppression de l'index se fait sans bruit :

```
DROP INDEX personne_id_idx;
```

Définissons une contrainte d'unicité sur la colonne plutôt qu'un index :

```
ALTER TABLE personne ADD CONSTRAINT unique_id UNIQUE (id);
```

```
$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "unique_id" UNIQUE CONSTRAINT, btree (id)
```

Un index est également créé. La contrainte empêche sa suppression :

```
DROP INDEX unique_id ;
```

```
ERREUR:  n'a pas pu supprimer index unique_id car il est requis par contrainte
 unique_id sur table personne
ASTUCE : Vous pouvez supprimer contrainte unique_id sur table personne à la
 place.
```

Le principe est le même pour les clés primaires.

Indexation avancée :

Il faut aussi savoir que PostgreSQL permet de créer des index B-tree :

- sur plusieurs colonnes ;
- sur des résultats de fonction ;
- sur une partie des valeurs indexées ;
- intégrant des champs non indexés mais souvent récupérés avec les champs indexés (index couvrants).

D'autres types d'index que B-tree existent, destinés à certains types de données ou certains cas d'optimisation précis.

5.2 FONCTIONNEMENT D'UN INDEX

5.2.1 Structure d'un index

- Structure associant des clés (termes) à des localisations (pages)
- Structure de données spécialisée, plusieurs types
- Séparée de la table
- Analogies :
  - fiches en carton des bibliothèques avant l'informatique (B-tree)
  - index d'un livre technique (GIN)

Les fiches en carton des anciennes bibliothèques sont un bon équivalent du type d'index le plus courant utilisé par les bases de données en général et PostgreSQL en particulier : le B-tree.

Lorsque l'on recherche des ouvrages dans la bibliothèque, il est possible de parcourir l'intégralité du bâtiment pour chercher les livres qui nous intéressent. Ceci prend énormément de temps. La bibliothèque peut être triée, mais ce tri ne permet pas forcément de trouver facilement le livre. Ce type de recherche trouve son analogie sous la forme du parcours complet d'une table (*Seq Scan*).

Une deuxième méthode pour localiser l'ouvrage consiste à utiliser un index. Sur fiche carton ou sous forme informatique, cet index associe par exemple le nom d'auteur à un ensemble de références (emplacements dans les rayonnages) où celui-ci est présent. Ainsi, pour trouver les œuvres de Proust avec l'index en carton, il suffit de parcourir les fiches, dont l'intégralité tient devant l'utilisateur. La fiche indique des références dans plusieurs rayons et il faudra aller se déplacer pour trouver les œuvres, en allant directement aux bons rayons.

Dans une base de données, le fonctionnement d'un index est très similaire. En effet, comme dans une bibliothèque, l'index est une structure de données à part, qui n'est pas strictement nécessaire à l'exploitation des informations, et qui est principalement utilisée pour la recherche dans l'ensemble de données. Cette structure de données possède un coût de maintenance, dans les deux cas : toute modification des données entraîne des modifications de l'index afin de le maintenir à jour. Et un index qui n'est pas à jour peut provoquer de gros problèmes. Dans le doute, on peut jeter l'index et le recréer de zéro sans problème d'intégrité des données originales.

Il peut y avoir plusieurs index suivant les besoins. L'index trié par auteur ne permet pas de trouver un livre dont on ne connaît que le titre (sauf à lire toutes les fiches). Il faut alors un autre index classé par titre.

Pour filer l'analogie : un index peut être multicolonne (les fiches en carton triées par auteur le sont car elles contiennent le titre, et pas que la référence dans les rayons). L'index peut répondre à une demande à lui seul : il suffit pour compter le nombre de livres de Marcel Proust (c'est le principe des *Index Only Scans*). Une fiche d'un index peut contenir des informations supplémentaires (dates de publication, éditeur…) pour faciliter d'autres recherches sans aller dans les rayons (index « couvrant »).

Dans la réalité comme dans une base de données, il y a un dilemme quand il faut récupérer de très nombreuses données : soit aller chercher de nombreux livres un par un dans les rayons, soit balayer tous les livres systématiquement dans l'ordre où ils viennent pour éviter trop d'allers-retours.

Autres types d'index non informatiques similaires au B-tree :

- les tables décennales de l'État civil, qui pointent vers un endroit précis des registres des actes de naissance, mariage ou décès d'une commune ;
- l'index d'un catalogue papier.

L'index d'un livre technique ou d'un livre de recettes cible des parties des données et non les données elles-mêmes (comme le titre). Il s'approche plus d'un autre type d'index, le GIN, qui existe aussi dans PostgreSQL.

Un annuaire téléphonique papier présente les données sous un mode strictement ordonné. Cette intégration entre table et index n'a pas d'équivalent sous PostgreSQL mais existe dans d'autres moteurs de bases de données.

5.2.2 Un index n'est pas magique

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

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

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

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

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

5.2.3 Index B-tree

- Type d'index le plus courant
  - et le plus simple
- Utilisable pour les contraintes d'unicité
- Supporte les opérateurs : `<`, `<=`, `=`, `>=`, `>`
- Supporte le tri
- Ne peut pas indexer des colonnes de plus de 2,6 ko

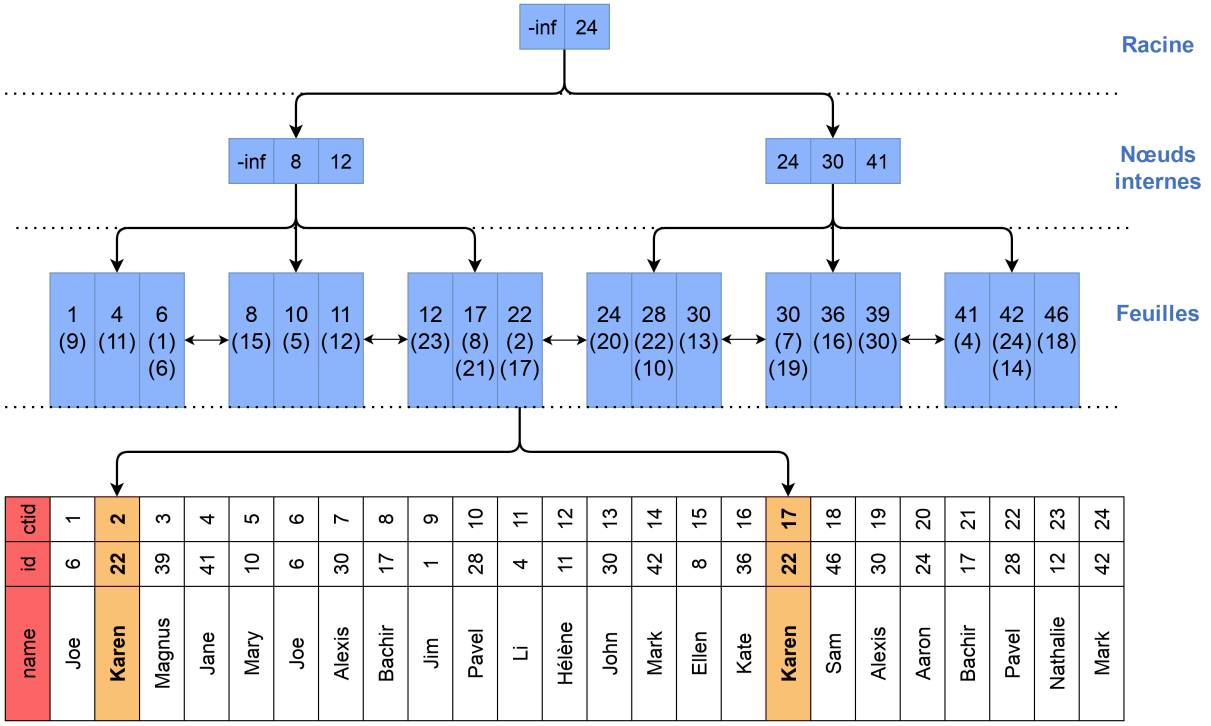
L'index B-tree est le plus simple conceptuellement parlant. Sans entrer dans les détails, un index B-tree est par définition équilibré : ainsi, quelle que soit la valeur recherchée, le coût est le même lors du

parcours d'index. Ceci ne veut pas dire que toute requête impliquant l'index mettra le même temps ! En effet, si chaque clé n'est présente qu'une fois dans l'index, celle-ci peut être associée à une multitude de valeurs, qui devront alors être cherchées dans la table.

L'algorithme utilisé par PostgreSQL pour ce type d'index suppose que chaque page peut contenir au moins trois valeurs. Par conséquent, chaque valeur ne peut excéder un peu moins d'⅓ de bloc, soit environ 2,6 ko. La valeur en question correspond donc à la totalité des données de toutes les colonnes de l'index pour une seule ligne. Si l'on tente de créer ou maintenir un index sur une table ne satisfaisant pas ces prérequis, une erreur sera renvoyée, et la création de l'index (ou l'insertion/mise à jour de la ligne) échouera. Ces champs sont souvent des longs textes ou des champs composés dont on cherchera plutôt des parties, et un index B-tree n'est de toute façon pas adapté. Si un index de type B-tree est tout de même nécessaire sur les colonnes en question, pour des recherches sur l'intégralité de la ligne, les index de type hash sont plus adaptés (mais ils ne supportent que l'opérateur `=`).

5.2.4 Exemple de structure d'index

```
SELECT name FROM ma_table WHERE id = 22
```



Ce schéma présente une vue très simplifiée d'une table (en blanc, avec ses champs `id` et `name`) et d'un index B-tree sur `id` (en bleu), tel que le créerait :

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

Un index B-tree peut contenir trois types de nœuds :

- la racine : elle est unique c'est la base de l'arbre ;
- des nœuds internes : il peut y en avoir plusieurs niveaux ;
- des feuilles : elles contiennent :
  - les valeurs indexées (triées !) ;
  - les valeurs incluses (si applicable) ;
  - les positions physiques (`ctid`), ici entre parenthèses et sous forme abrégée, car la forme réelle est (numéro de bloc, position de la ligne dans le bloc) ;
  - l'adresse de la feuille précédente et de la feuille suivante.

La racine et les nœuds internes contiennent des enregistrements qui décrivent la valeur minimale de chaque bloc du niveau inférieur et leur adresse (`ctid`).

Lors de la création de l'index, il ne contient qu'une feuille. Lorsque cette feuille se remplit, elle se divise en deux et un nœud racine est créé au-dessus. Les feuilles se remplissent ensuite progressivement et se séparent en deux quand elles sont pleines. Ce processus remplit progressivement la racine. Lorsque la racine est pleine, elle se divise en deux nœuds internes, et une nouvelle racine est crée au-dessus. Ce processus permet de garder un arbre équilibré.

Recherchons le résultat de :

```
SELECT name FROM ma_table WHERE id = 22
```

en passant par l'index.

- En parcourant la racine, on cherche un enregistrement dont la valeur est strictement supérieure à la valeur que l'on recherche. Ici, 22 est plus petit que 24 : on explore donc le nœud de gauche.
- Ce nœud référence trois nœuds inférieurs (ici des feuilles). On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud : pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre. Ici, 22 est plus grand que 12, on explore donc le nœud de droite au niveau inférieur.
- Un arbre B-tree peut bien évidemment avoir une profondeur plus grande, auquel cas l'étape précédente est répétée.
- Une fois arrivé sur une feuille, il suffit de la parcourir pour récupérer l'ensemble des positions physiques des lignes correspondants au critère. Ici, la feuille nous indique qu'à la valeur 22 correspondent deux lignes aux positions 2 et 17. Lorsque la valeur recherchée est supérieure ou égale à la plus grande valeur du bloc, PostgreSQL va également lire le bloc suivant. Ce cas de figure peut se produire si PostgreSQL a divisé une feuille en deux avant ou même pendant la recherche que nous exécutons. Ce serait par exemple le cas si on cherchait la valeur 30.
- Pour trouver les valeurs de `name`, il faut aller chercher dans la table même les lignes aux positions trouvées dans l'index. D'autre part, les informations de visibilité des lignes doivent aussi être trouvées dans la table. (Il existe des cas où la recherche peut éviter cette dernière étape : ce sont les *Index Only Scan*.)

Même en parcourant les deux structures de données, si la valeur recherchée représente une assez petite fraction des lignes totales, le nombre d'accès disques sera donc fortement réduit. En revanche, au lieu d'effectuer des accès séquentiels (pour lesquels les disques durs classiques sont relativement performants), il faudra effectuer des accès aléatoires, en *sautant* d'une position sur le disque à une autre. Le choix est fait par l'optimiseur.

Supposons désormais que nous souhaitions exécuter une requête sans filtre, mais exigeant un tri, du type :

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

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

L'alternative consisterait à parcourir l'ensemble de la table, et trier toutes les lignes afin de les obtenir dans le bon ordre. Un tel tri peut être très coûteux, en mémoire comme en temps CPU. D'ailleurs,

de tels tris débordent très souvent sur disque (via des fichiers temporaires) afin de ne pas garder l'intégralité des données en mémoire.

Pour les requêtes utilisant des opérateurs d'inégalité, on voit bien comment l'index peut là aussi être utilisé. Par exemple, pour la requête suivante :

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

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

Dernière remarque : ce schéma ne montre qu'une entrée d'index pour 22, bien qu'il pointe vers deux lignes. En fait, il y avait bien deux entrées pour 22 avant PostgreSQL 13. Depuis cette version, PostgreSQL sait dédupliquer les entrées pour économiser de la place.

5.2.5 Index multicolonnes

- Possibilité d'indexer plusieurs colonnes :

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

- Ordre des colonnes **primordial**
  - accès direct aux premières colonnes de l'index
  - pour les autres, PostgreSQL lira tout l'index ou ignorera l'index

Il est possible de créer un index sur plusieurs colonnes. Il faut néanmoins être conscient des requêtes supportées par un tel index. Admettons que l'on crée une table d'un million de lignes avec un index sur trois champs :

```
CREATE TABLE t1 (c1 int, c2 int, c3 int, c4 text);

INSERT INTO t1 (c1, c2, c3, c4)
SELECT i*10,j*5,k*20, 'text'||i||j||k
FROM generate_series (1,100) i
CROSS JOIN generate_series(1,100) j
CROSS JOIN generate_series(1,100) k ;

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

VACUUM ANALYZE t1 ;

-- Figer des paramètres pour l'exemple
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

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

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 1000 and c2=500 and c3=2000 ;

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..8.45 rows=1 width=22)
   Index Cond: ((c1 = 1000) AND (c2 = 500) AND (c3 = 2000))
```

Et encore plus quand l'index permet de répondre intégralement au contenu de la requête :

```
EXPLAIN SELECT c1,c2,c3 FROM t1 WHERE c1 = 1000 and c2=500 ;

                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Only Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..6.33 rows=95 width=12)
   Index Cond: ((c1 = 1000) AND (c2 = 500))
```

Mais si les premières colonnes de l'index ne sont pas spécifiées, alors l'index devra être parcouru en grande partie.

Cela reste plus intéressant que parcourir toute la table, surtout si l'index est petit et contient toutes les données du `SELECT`. Mais le comportement dépend alors de nombreux paramètres, comme les statistiques, les estimations du nombre de lignes ramenées et les valeurs relatives de `seq_page_cost` et `random_page_cost` :

```
SET random_page_cost TO 0.1 ; SET seq_page_cost TO 0.1 ;  -- SSD

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (...) (...)
   Index Cond: (c3 = 2000)
   Buffers: shared hit=3899
 Planning:
   Buffers: shared hit=15
 Planning Time: 0.218 ms
 Execution Time: 67.081 ms
```

Noter que tout l'index a été lu.

Mais pour limiter les aller-retours entre index et table, PostgreSQL peut aussi décider d'ignorer l'index et de parcourir directement la table :

```
SET random_page_cost TO 4 ; SET seq_page_cost TO 1 ;  -- défaut (disque mécanique)

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                    QUERY PLAN
-----------------------------------------------------------------------------------
Seq Scan on t1  (cost=0.00..18871.00 rows=9600 width=22) (...)
   Filter: (c3 = 2000)
   Rows Removed by Filter: 990000
   Buffers: shared hit=6371
 Planning Time: 0.178 ms
 Execution Time: 114.572 ms
```

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

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

```
SELECT * FROM t1 WHERE c1 = 20 ;
SELECT * FROM t1 WHERE c1 = 20 AND c2 = 50 AND c3 = 400 ;
SELECT * FROM t1 WHERE c1 = 10 AND c2 <= 4 ;
```

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

```
SELECT * FROM t1 WHERE c1 = 100 AND c2 >= 80 AND c3 = 40 ;
SELECT * FROM t1 WHERE c1 < 100 AND c2 = 100 ;
```

Le plan de cette dernière requête est :

```
 Bitmap Heap Scan on t1  (cost=2275.98..4777.17 rows=919 width=22) (...)
   Recheck Cond: ((c1 < 100) AND (c2 = 100))
   Heap Blocks: exact=609
   Buffers: shared hit=956
   ->  Bitmap Index Scan on t1_c1_c2_c3_idx  (cost=0.00..2275.76 rows=919 width=0)
↪  (...)
         Index Cond: ((c1 < 100) AND (c2 = 100))
         Buffers: shared hit=347
 Planning Time: 0.227 ms
 Execution Time: 15.596 ms
```

Les index multicolonnes peuvent aussi être utilisés pour le tri comme dans les exemples suivants. Il n'y a pas besoin de trier (ce peut être très coûteux) puisque les données de l'index sont triées. Ici le cas est optimal puisque l'index contient toutes les données nécessaires :

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

Le plan de cette dernière requête est :

```
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..55893.66 rows=1000000 width=22)
↪  (...)
   Buffers: shared hit=1003834
 Planning Time: 0.282 ms
 Execution Time: 425.520 ms
```

Il est donc nécessaire d'avoir une bonne connaissance de l'application (ou de passer du temps à observer les requêtes consommatrices) pour déterminer comment créer des index multicolonnes pertinents pour un nombre maximum de requêtes.

5.2.6 Nœuds des index

- *Index Scan*
- *Bitmap Scan*
- *Index Only Scan*
  - idéal pour les performances
- et les variantes parallélisées

L'optimiseur a le choix entre plusieurs parcours pour utiliser un index, principalement suivant la quantité d'enregistrements à récupérer :

5.2.6.1 Index Scan

Un *Index Scan* est optimal quand il y a peu d'enregistrements à récupérer. Noter qu'il comprend l'accès à l'index *et* celui à la table ensuite.

5.2.6.2 Bitmap Scan

Le *Bitmap Scan* est utile quand il y a plus de lignes, ou quand on veut lire plusieurs index d'une même table pour satisfaire plusieurs conditions de filtre.

Il se décompose en deux nœuds : un *Bitmap Index Scan* qui récupère des blocs d'index, et un *Bitmap Heap Scan* qui va chercher les blocs dans la table.

Typiquement, ce nœud servira pour des recherches de plages de valeurs ou de grandes quantités de lignes. Il est favorisé par une bonne corrélation des données avec leur emplacement physique.

5.2.6.3 Index Only Scan

L'*Index Only Scan* est utile quand les champs de la requête correspondent aux colonnes de l'index. Ce nœud permet d'éviter la lecture de tout ou partie de la table et est donc très performant.

Autre intérêt de l'*Index Only Scan* : les enregistrements cherchés sont contigus dans l'index (puisqu'il est trié), et le nombre d'accès disque est bien plus faible. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

Si peu de champs de la table sont impliqués dans la requête, il faut penser à viser un *Index Only Scan*.

5.2.6.4 Parallélisation

Chacun de ses nœuds a une version parallélisable si l'index est assez grand et que l'optimiseur pense que paralléliser est utile. Il apparaît alors un nœud *Gather* pour rassembler les résultats des différents *workers*.

5.3 MÉTHODOLOGIE DE CRÉATION D'INDEX

- On indexe pour une requête
  - ou idéalement une collection de requêtes
- Et pas « une table »

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

5.3.1 L'index ? Quel index ?

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

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

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

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

5.3.2 Index et clés étrangères

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

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

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

La colonne source devrait être indexée elle aussi : en effet, il est alors possible de la parcourir de manière ordonnée, et donc de réaliser la jointure selon l'algorithme *Merge Join* (comme vu lors du module sur les plans d'exécution[5]), et donc d'être beaucoup plus rapide. Un tel index accélérera de la même manière les *Nested Loop*, en permettant de parcourir l'index une fois par ligne de la relation externe au lieu de parcourir l'intégralité de la table.

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

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

Deux exceptions : les champs ayant une cardinalité très faible et homogène (par exemple, un champ homme/femme dans une population équilibrée) ; et ceux dont on constate l'inutilité après un certain temps, par des valeurs à zéro dans `pg_stat_user_indexes`.

[5] https://dali.bo/j0_html

5.4 INDEX INUTILISÉ

C'est souvent tout à fait normal

- Utiliser l'index est-il rentable ?
- La requête est-elle compatible ?
- Bug de l'optimiseur : rare

C'est l'optimiseur SQL qui choisit si un index doit ou non être utilisé. Il est tout à fait possible que PostgreSQL décide qu'utiliser un index donné n'en vaut pas la peine par rapport à d'autres chemins. Il faut aussi savoir identifier les cas où l'index ne peut *pas* être utilisé.

L'optimiseur possède forcément quelques limitations. Certaines sont un compromis par rapport au temps que prendrait la recherche systématique de toutes les optimisations imaginables. Il y aussi le problème des estimations de volumétries, qui sont d'autant plus difficiles que la requête est complexe.

Quant à un vrai bug, si le cas peut être reproduit, il doit être remonté aux développeurs de PostgreSQL. D'expérience, c'est rarissime.

5.4.1 Index utilisable mais non utilisé

- L'optimiseur pense qu'il n'est pas rentable
  - sélectivité trop faible
  - meilleur chemin pour remplir d'autres critères
  - index redondant
  - *Index Only Scan* nécessite un `VACUUM` fréquent
- Les estimations de volumétries doivent être assez bonnes !
  - statistiques récentes, précises

Il existe plusieurs raisons pour que PostgreSQL néglige un index.

Sélectivité trop faible, trop de lignes :

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

Autrement dit, l'index n'est pas assez discriminant pour que ce soit la peine de faire des allers-retours entre lui et la table. Le seuil dépend entre autres des volumétries de la table et de l'index et du rapport entre les paramètres `random_page_cost` et `seq_page_cost` (respectivement 4 et 1 pour un disque dur classique peu rapide, et souvent 1 et 1 pour du SSD, voire moins).

Il y a un meilleur chemin :

Un index sur un champ n'est qu'un chemin parmi d'autres, en aucun cas une obligation, et une requête contient souvent plusieurs critères sur des tables différentes. Par exemple, un index sur un filtre peut être ignoré si un autre index permet d'éviter un tri coûteux, ou si l'optimiseur juge que faire une jointure avant de filtrer le résultat est plus performant.

Index redondant :

Il existe un autre index doublant la fonctionnalité de celui considéré. PostgreSQL favorise naturellement un index plus petit, plus rapide à parcourir. À l'inverse, un index plus complet peut favoriser plusieurs filtres, des tris, devenir couvrant…

VACUUM trop ancien :

Dans le cas précis des *Index Only Scan*, si la table n'a pas été récemment nettoyée, il y aura trop d'allers-retours avec la table pour vérifier les informations de visibilité (*heap fetches*). Un `VACUUM` permet de mettre à jour la *Visibility Map* pour éviter cela.

Statistiques périmées :

Il peut arriver que l'optimiseur se trompe quand il ignore un index. Des statistiques périmées sont une cause fréquente. Pour les rafraîchir :

```
ANALYZE (VERBOSE) nom_table;
```

Si cela résout le problème, ce peut être un indice que l'autovacuum ne passe pas assez souvent (voir `pg_stat_user_tables.last_autoanalyze`). Il faudra peut-être ajuster les paramètres `autovacuum_analyze_scale_factor` ou `autovacuum_analyze_threshold` sur les tables.

Statistiques pas assez fines :

Les statistiques sur les données peuvent être trop imprécises. Le défaut est un histogramme de 100 valeurs, basé sur 300 fois plus de lignes. Pour les grosses tables, augmenter l'échantillonnage sur les champs aux valeurs peu homogènes est possible :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500 ;
```

La valeur 500 n'est qu'un exemple. Monter beaucoup plus haut peut pénaliser les temps de planification. Ce sera d'autant plus vrai si on applique cette nouvelle valeur globalement, donc à tous les champs de toutes les tables (ce qui est certes le plus facile).

Estimations de volumétries trompeuses :

Par exemple, une clause `WHERE` sur deux colonnes corrélées (ville et code postal par exemple), mène à une sous-estimation de la volumétrie résultante par l'optimiseur, car celui-ci ignore le lien entre

les deux champs. Vous pouvez demander à PostgreSQL de calculer cette corrélation avec l'ordre `CREATE STATISTICS` (voir le module de formation J2[6] ou la documentation officielle[7]).

Compatibilité :

Il faut toujours s'assurer que la requête est écrite correctement et permet l'utilisation de l'index.

Un index peut être inutilisable à cause d'une fonction plus ou moins explicite, ou encore d'un mauvais typage. Il arrive que le critère de filtrage ne peut remonter sur la table indexée à cause d'un CTE matérialisé (explicitement ou non), d'un `DISTINCT`, ou d'une vue complexe.

Nous allons voir quelques problèmes classiques.

5.4.2 Index inutilisable à cause d'une fonction

- Pas le bon type (`CAST` plus ou moins explicite)

```
EXPLAIN SELECT * FROM clients WHERE client_id = 3::numeric;
```

- Utilisation de fonctions, comme :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

Voici quelques exemples d'index incompatible avec la clause `WHERE` :

Mauvais type :

Cela peut paraître contre-intuitif, mais certains transtypages ne permettent pas de garantir que les résultats d'un opérateur (par exemple l'égalité) seront les mêmes si les arguments sont convertis dans un type ou dans l'autre. Cela dépend des types et du sens de conversion. Dans les exemples suivants, le champ `client_id` est de type `bigint`. PostgreSQL réussit souvent à convertir, mais ce n'est pas toujours parfait.

```
EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = 3 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using clients_pkey on clients
   Index Cond: (client_id = 3)
```

```
EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = 3::numeric;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on clients
   Filter: ((client_id)::numeric = '3'::numeric)
```

[6] https://dali.bo/j2_html
[7] https://docs.postgresql.fr/current/sql-createstatistics.html

```
EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = 3::int;

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using clients_pkey on clients
   Index Cond: (client_id = 3)

EXPLAIN (COSTS OFF) SELECT * FROM clients WHERE client_id = '003';

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using clients_pkey on clients
   Index Cond: (client_id = '3'::bigint)
```

De même, les conversions entre `date` et `timestamp` / `timestamptz` se passent généralement bien.

Autres exemples :

- Dans une jointure, si les deux champs joints n'ont pas le même type, il est possible que de simples index ne soient pas utilisables, ou un seul d'entre eux. Il faudra corriger l'incohérence, ou créer des index fonctionnels incluant le transtypage.
- Un index B-tree sur un tableau ou un JSON ne peut servir pour une recherche sur un de ses éléments. Il faudra s'orienter vers un index plus spécialisé, par exemple GIN ou GiST.

Utilisation de fonction :

Si une fonction est appliquée sur la colonne à indexer, comme dans cet exemple classique :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

alors PostgreSQL n'utilisera pas l'index sur `ma_date`. Il faut réécrire la requête ainsi :

```
SELECT * FROM ma_table WHERE ma_date >='2014-01-01' AND ma_date<'2015-01-01' ;
```

Dans l'exemple suivant, on cherche les commandes dont la date tronquée au mois correspond au 1er janvier, c'est-à-dire aux commandes dont la date est entre le 1er et le 31 janvier. Pour un humain, la logique est évidente, mais l'optimiseur n'en a pas connaissance.

```
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_trunc('month', date_commande) = '2015-01-01';

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

Il faut plutôt écrire :

```
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_commande BETWEEN '2015-01-01' AND '2015-01-31' ;

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

Dans certains cas, la réécriture est impossible (fonction complexe, code non modifiable…). Nous verrons qu'un index fonctionnel peut parfois être la solution.

Ces exemples semblent évidents, mais il peut être plus compliqué de trouver dans l'urgence la cause du problème dans une grande requête d'un schéma mal connu.

5.4.3 Index inutilisable à cause d'un LIKE '…%'

```
SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';
```

- Solution :

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

Si vous avez un index « normal » sur une chaîne texte, certaines recherches de type `LIKE` n'utiliseront pas l'index. En effet, il faut bien garder à l'esprit qu'un index est basé sur un opérateur précis. Ceci est généralement indiqué correctement dans la documentation, mais pas forcément très intuitif.

Si un opérateur non supporté pour le critère de tri est utilisé, l'index ne servira à rien :

```
CREATE INDEX ON fournisseurs (commentaire);
EXPLAIN ANALYZE SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';

                           QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on fournisseurs  (cost=0.00..225.00 rows=1 width=45)
                           (actual time=0.045..1.477 rows=47 loops=1)
   Filter: (commentaire ~~ 'ipsum%'::text)
   Rows Removed by Filter: 9953
 Planning time: 0.085 ms
 Execution time: 1.509 ms
```

Nous verrons qu'il existe d'autre classes d'opérateurs, permettant d'indexer correctement la requête précédente, et que `varchar_pattern_ops` est l'opérateur permettant d'indexer la requête précédente.

5.4.4 Index inutilisable car invalide

– `CREATE INDEX … CONCURRENTLY` peut échouer

Dans le cas où un index a été construit avec la clause `CONCURRENTLY`, nous avons vu qu'il peut arriver que l'opération échoue et l'index existe mais reste invalide, et donc inutilisable. Le problème ne se pose pas pour un échec de `REINDEX … CONCURRENTLY`, car l'ancienne version de l'index est toujours là et utilisable.

5.5 INDEXATION B-TREE AVANCÉE

De nombreuses possibilités d'indexation avancée :

- Index partiels
- Index fonctionnels
- Index couvrants
- Classes d'opérateur

5.5.1 Index partiels

- N'indexe qu'une partie des données :

```
CREATE INDEX on evenements (type) WHERE  traite IS FALSE ;
```

- Ne sert que si la clause est logiquement équivalente !
  - ou partie de la clause (inégalités, `IN`)
- Intérêt : index beaucoup plus petit

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

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

```
CREATE TABLE evenements (
    id int primary key,
    traite bool NOT NULL,
    type text NOT NULL,
    payload text
);

-- 10 000 événements traités
INSERT INTO evenements (id, traite, type) (
    SELECT i,
        true,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
```

```
    FROM generate_series(1, 10000) as i);

-- et 10  non encore traités
INSERT INTO evenements (id, traite, type) (
    SELECT i,
        false,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
    FROM generate_series(10001, 10010) as i);

\d evenements

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

Typiquement, différents applicatifs vont être intéressés par des événements d'un certain type, mais les événements déjà traités ne sont quasiment jamais accédés, du moins via leur état (une requête portant sur `traite IS true` sera exceptionnelle et ramènera l'essentiel de la table : un index est inutile).

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

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

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

```
EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION' ;

                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on evenements  (cost=0.00..183.12 rows=50 width=69)
   Filter: (type = 'FACTURATION'::text)
```

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

```
EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION' AND NOT traite ;

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on evenements  (cost=8.22..54.62 rows=25 width=69)
   Recheck Cond: ((type = 'FACTURATION'::text) AND (NOT traite))
   ->  Bitmap Index Scan on index_partiel  (cost=0.00..8.21 rows=25 width=0)
         Index Cond: (type = 'FACTURATION'::text)
```

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

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

SELECT idxname, pg_size_pretty(pg_total_relation_size(idxname::text))
FROM (VALUES ('index_complet'), ('index_partiel')) as a(idxname);

    idxname    | pg_size_pretty
---------------+----------------
 index_complet | 88 kB
 index_partiel | 16 kB
```

Un index composé sur `(is_traite,type)` serait efficace, mais inutilement gros.

Clauses de requête et clause d'index :

Attention ! Les clauses de l'index et du `WHERE` doivent être **logiquement équivalentes** ! (et de préférence identiques)

Par exemple, dans les requêtes précédentes, un critère `traite IS FALSE` à la place de `NOT traite` n'utilise pas l'index (en effet, il ne s'agit pas du même critère à cause de `NULL` : `NULL = false` renvoie `NULL`, mais `NULL IS false` renvoie `false`).

Par contre, des conditions mathématiquement plus restreintes que l'index permettent son utilisation :

```
CREATE INDEX commandes_recentes_idx
ON commandes (client_id) WHERE date_commande > '2015-01-01' ;

EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE date_commande > '2016-01-01' AND client_id = 17 ;

                      QUERY PLAN
------------------------------------------------------
 Index Scan using commandes_recentes_idx on commandes
   Index Cond: (client_id = 17)
   Filter: (date_commande > '2016-01-01'::date)
```

Mais cet index partiel ne sera pas utilisé pour un critère précédant 2015.

De la même manière, si un index partiel contient une liste de valeurs, `IN ()` ou `NOT IN ()`, il est en principe utilisable :

```
CREATE INDEX commandes_1_3 ON commandes (numero_commande)
WHERE  mode_expedition IN (1,3);

EXPLAIN (COSTS OFF) SELECT * FROM commandes WHERE mode_expedition = 1 ;

                 QUERY PLAN
--------------------------------------------
 Index Scan using commandes_1_3 on commandes
   Filter: (mode_expedition = 1)

DROP INDEX commandes_1_3 ;
```

```
CREATE INDEX commandes_not34 ON commandes (numero_commande)
WHERE  mode_expedition NOT IN (3,4);

EXPLAIN (COSTS OFF) SELECT * FROM commandes WHERE mode_expedition = 1 ;

                  QUERY PLAN
-----------------------------------------------
 Index Scan using commandes_not34 on commandes
   Filter: (mode_expedition = 1)

DROP INDEX commandes_not34 ;
```

5.5.2 Index partiels : cas d'usage

- Données *chaudes* et *froides*
- Index dédié à une requête avec une condition fixe

Le cas typique d'utilisation d'un index partiel est celui de l'exemple précédent : une application avec des données *chaudes*, fréquemment accédées et traitées, et des données *froides*, qui sont plus destinées à de l'historisation ou de l'archivage. Par exemple, un système de vente en ligne aura probablement intérêt à disposer d'index sur les commandes dont l'état est différent de clôturé : en effet, un tel système effectuera probablement des requêtes fréquemment sur les commandes qui sont en cours de traitement, en attente d'expédition, en cours de livraison mais très peu sur des commandes déjà livrées, qui ne serviront alors plus qu'à de l'analyse statistique.

De manière générale, tout système est susceptible de bénéficier des index partiels s'il doit gérer des données à état dont seul un sous-ensemble de ces états est activement exploité par les requêtes à optimiser. Par exemple, toujours sur cette même table, des requêtes visant à faire des statistiques sur les expéditions pourraient tirer parti de cet index :

```
CREATE INDEX index_partiel_expes ON evenements (id) WHERE type = 'EXPEDITION' ;

EXPLAIN SELECT count(id) FROM evenements WHERE type = 'EXPEDITION' ;

                              QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=106.68..106.69 rows=1 width=8)
   ->  Index Only Scan using index_partiel_expes on evenements  (cost=0.28..98.34
↪  rows=3337 width=4)
```

Nous avons mentionné précédemment qu'un index est destiné à satisfaire une requête ou un ensemble de requêtes. Donc, si une requête présente fréquemment des critères de ce type :

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

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

```
WHERE une_autre_colonne = une_valeur_fixe
```

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

5.5.3 Index partiels : utilisation

- Éviter les index de type :

```
CREATE INDEX ON matable ( champ_filtre ) WHERE champ_filtre = …
```

- Préférer :

```
CREATE INDEX ON matable ( champ_resultat ) WHERE champ_filtre = …
```

En général, un index partiel doit indexer une colonne différente de celle qui est filtrée (et donc connue). Ainsi, dans l'exemple précédent, la colonne indexée (`type`) n'est pas celle de la clause `WHERE`. On pose un critère, mais on s'intéresse aux types d'événements ramenés. Un autre index partiel pourrait porter sur `id WHERE NOT traite` pour simplement récupérer une liste des identifiants non traités de tous types.

L'intérêt est d'obtenir un index très ciblé et compact, et aussi d'économiser la place disque et la charge CPU de maintenance. Il faut tout de même que les index partiels soient notablement plus petits que les index « génériques » (au moins de moitié). Avec des index partiels spécialisés, il est possible de « précalculer » certaines requêtes critiques en intégrant leurs critères de recherche exacts.

5.5.4 Index fonctionnels : principe

- Un index sur `a` est inutilisable pour :

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

- Indexer le résultat de la fonction :

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

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

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

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

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

5.5.5 Index fonctionnels : conditions

- Critère identique à la fonction dans l'index
- Fonction impérativement `IMMUTABLE` !
  - délicat avec les conversions de dates/heures
- Ne pas espérer d'*Index Only Scan*

Il est facile de créer involontairement des critères comportant des fonctions, notamment avec des conversions de type ou des manipulations de dates. Il a été vu plus haut qu'il vaut mieux placer la transformation du côté de la constante. Par exemple, la requête suivante retourne toutes les commandes de l'année 2011, mais la fonction `extract` est appliquée à la colonne `date_commande` (type `date`) et l'index est inutilisable.

L'optimiseur ne peut donc pas utiliser un index :

```
CREATE INDEX ON commandes (date_commande) ;

EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;

                            QUERY PLAN
------------------------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on commandes
         Filter: (EXTRACT(year FROM date_commande) = '2011'::numeric)
```

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

```
EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE date_commande BETWEEN '01-01-2011'::date AND '31-12-2011'::date;

                            QUERY PLAN
------------------------------------------------------------------------
 Index Scan using commandes_date_commande_idx on commandes
   Index Cond: ((date_commande >= '2011-01-01'::date) AND (date_commande <=
↪  '2011-12-31'::date))
```

C'est la solution la plus propre.

Mais dans d'autres cas, une telle réécriture de la requête sera impossible ou très délicate. On peut alors créer un index fonctionnel, dont la définition doit être **strictement** celle du `WHERE` :

```
CREATE INDEX annee_commandes_idx ON commandes( extract('year' from date_commande) ) ;

EXPLAIN (COSTS OFF) SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;

                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on commandes
   Recheck Cond: (EXTRACT(year FROM date_commande) = '2011'::numeric)
   ->  Bitmap Index Scan on annee_commandes_idx
         Index Cond: (EXTRACT(year FROM date_commande) = '2011'::numeric)
```

Ceci fonctionne si `date_commande` est de type `date` ou `timestamp without timezone`.

Fonction immutable :

Cependant, n'importe quelle fonction d'indexation n'est pas utilisable, ou pas pour tous les types. La fonction d'indexation doit être notée `IMMUTABLE` : cette propriété indique à PostgreSQL que la fonction retournera toujours le **même résultat** quand elle est appelée avec les **mêmes arguments**.

En d'autres termes : le résultat de la fonction ne doit dépendre :

- ni du contenu de la base (pas de `SELECT` donc) ;
- ni de la configuration, ni de l'environnement (variables d'environnement, paramètres de session, fuseau horaire, formatage…) ;
- ni du temps (`now()` ou `clock_timestamp()` sont interdits, et indirectement les calculs d'âge) ;
- ni d'une autre fonction non-déterministe (comme `random()`) ou plus généralement non immutable.

Sans ces restrictions, l'endroit dans lequel la donnée est insérée dans l'index serait potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Pour revenir à l'exemple précédent : pour calculer l'année, on peut aussi imaginer un index avec la fonction `to_char`, une autre fonction hélas fréquemment utilisée pour les conversions de date. Au moment de la création d'un tel index, PostgreSQL renvoie l'erreur suivante :

```
CREATE INDEX annee_commandes_idx2
ON commandes ((to_char(date_commande,'YYYY')::int));

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

En effet, `to_char()` n'est pas immutable, juste « stable » et cela dans toutes ses variantes :

```
magasin=# \df+ to_char
                                        Liste des fonctions
…  Nom   |…résultat|  Type de données des paramètres  |…|Volatibilité|…
+--------+---------+-----------------------------------+-+------------+-
…to_char | text    | bigint, text                      | | stable     |…
…to_char | text    | double precision, text            | | stable     |…
```

```
…to_char | text    | integer, text                    | | stable     |…
…to_char | text    | interval, text                   | | stable     |…
…to_char | text    | numeric, text                    | | stable     |…
…to_char | text    | real, text                       | | stable     |…
…to_char | text    | timestamp without time zone, text| | stable     |…
…to_char | text    | timestamp with time zone, text   | | stable     |…
(8 lignes)
```

La raison est que `to_date` accepte des paramètres de formatage qui dépendent de la session (nom du mois, virgule ou point décimal…). Ce n'est pas une très bonne fonction pour convertir une date ou heure en nombre.

La fonction `extract`, elle, est bien immutable quand il s'agit de convertir `commande.date_commande` de `date` vers une année, comme dans l'exemple plus haut.

```
 \sf extract (text, date)
CREATE OR REPLACE FUNCTION pg_catalog."extract"(text, date)
 RETURNS numeric
 LANGUAGE internal
 IMMUTABLE PARALLEL SAFE STRICT
AS $function$extract_date$function$
```

De même, `extract` est immutable avec une entrée de type `timestamp without time zone`.

Les choses se compliquent si l'on manipule des heures avec fuseau horaire. En effet, il est conseillé de toujours privilégier la variante `timestamp with time zone`. Cette fois, l'index fonctionnel basé avec `extract` va poser problème :

```
DROP INDEX annee_commandes_idx ;
-- Nouvelle table d'exemple avec date_commande comme timestamp with time zone
-- La conversion introduit implicitement le fuseau horaire de la session
CREATE TABLE commandes2 (LIKE commandes INCLUDING ALL);
ALTER TABLE  commandes2 ALTER COLUMN date_commande TYPE timestamp with time zone ;
INSERT INTO  commandes2 SELECT * FROM commandes ;
-- Reprise de l'index fonctionnel précédent
CREATE INDEX annee_commandes2_idx
ON commandes2(extract('year' from date_commande) ) ;

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

En effet la fonction `extract` n'est pas immutable pour le type `timestamp with time zone` :

```
magasin=# \sf extract (text, timestamp with time zone)
CREATE OR REPLACE FUNCTION pg_catalog."extract"(text, timestamp with time zone)
 RETURNS numeric
 LANGUAGE internal
 STABLE PARALLEL SAFE STRICT
AS $function$extract_timestamptz$function$
```

Pour certains *timestamps* autour du Nouvel An, l'année retournée dépend du fuseau horaire. Le problème se poserait bien sûr aussi si l'on extrayait les jours ou les mois.

Il est possible de « tricher » en figeant le fuseau horaire dans une fonction pour obtenir un type intermédiaire `timestamp without time zone`, qui ne posera pas de problème :

```
CREATE INDEX annee_commandes2_idx
ON commandes2(extract('year' from (
  date_commande AT TIME ZONE 'Europe/Paris' )::timestamp
  )) ;
```

Ce contournement impose de modifier le critère de la requête. Tant qu'on y est, il peut être plus clair d'enrober l'appel dans une fonction que l'on définira immutable.

```
CREATE OR REPLACE FUNCTION annee_paris (t timestamptz)
RETURNS int
AS $$
    SELECT extract ('year' FROM (t AT TIME ZONE 'Europe/Paris')::timestamp) ;
$$ LANGUAGE sql
IMMUTABLE ;

CREATE INDEX annee_commandes2_paris_idx ON commandes2 (annee_paris (date_commande));
VACUUM ANALYZE commandes2 ;

EXPLAIN (COSTS OFF)
SELECT * FROM commandes2
WHERE annee_paris (date_commande) = 2021 ;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using annee_commandes2_paris_idx on commandes2
   Index Cond: ((EXTRACT(year FROM (date_commande AT TIME ZONE
↪  'Europe/Paris'::text)))::integer = 2021)
```

Le nom de la fonction est aussi une indication pour les utilisateurs dans d'autres fuseaux.

Certes, on a ici modifié le code de la requête, mais il est parfois possible de contourner ce problème en passant par des vues qui masquent la fonction.

Signalons enfin la fonction `date_part` : c'est une alternative possible à `extract`, avec les mêmes soucis et contournement.

À partir de PostgreSQL 16, une autre possibilité existe avec `date_trunc` car la variante avec `timestamp without time zone` est devenue immutable :

```
CREATE INDEX annee_commandes2_paris_idx3
ON commandes2 ( (date_trunc ( 'year', date_commande, 'Europe/Paris')) );
ANALYZE commandes2 ;

EXPLAIN (COSTS OFF)
SELECT * FROM commandes2
WHERE date_trunc('year', date_commande, 'Europe/Paris') = '2021-01-01'::timestamptz;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using annee_commandes2_paris_idx3 on commandes2
   Index Cond: (date_trunc('year'::text, date_commande, 'Europe/Paris'::text) =
↪  '2021-01-01 00:00:00+01'::timestamp with time zone)
```

Index Only Scan :

Obtenir un *Index Only Scan* est une optimisation importante pour les requêtes critiques avec peu de champs sur la table. Hélas, en raison d'une limitation du planificateur, les index fonctionnels ne donnent pas lieu à un *Index Only Scan* :

```
EXPLAIN (COSTS OFF)
SELECT annee_paris (date_commande) FROM commandes2
WHERE annee_paris (date_commande) > 2021 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using annee_commandes2_paris_idx on commandes2
   Index Cond: ((EXTRACT(year FROM (date_commande AT TIME ZONE
↪  'Europe/Paris'::text)))::integer > 2021)
```

Plus insidieusement, le planificateur peut choisir un *Index Only Scan*… sur la colonne sur laquelle porte la fonction !

```
EXPLAIN SELECT count( annee_paris(date_commande) ) FROM commandes2 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
  Aggregate  (cost=28520.40..28520.41 rows=1 width=8)
   ->  Index Only Scan using commandes2_date_commande_idx on commandes2
↪  (cost=0.42..18520.41 rows=999999 width=8)
```

Ce qui entraîne au moins un gaspillage de CPU pour réexécuter les fonctions sur chaque ligne.

Sacrifier un peu d'espace disque pour une colonne générée et son index (non fonctionnel) peut s'avérer une solution :

```
-- Attention, cette commande réécrit la table
ALTER TABLE commandes2 ADD COLUMN annee_paris smallint
  GENERATED ALWAYS  AS ( annee_paris (date_commande) ) STORED ;
CREATE INDEX commandes2_annee_paris_idx ON commandes2 (annee_paris) ;
-- Prise en compte des statistiques et des lignes mortes sur la table réécrite
VACUUM ANALYZE commandes2;

EXPLAIN SELECT count( annee_paris ) FROM commandes2 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=14609.10..14609.11 rows=1 width=8)
   ->  Gather  (cost=14608.88..14609.09 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=13608.88..13608.89 rows=1 width=8)
               ->  Parallel Index Only Scan using commandes2_annee_paris_idx on
↪  commandes2  (cost=0.42..12567.20 rows=416672 width=2)
```

5.5.6 Index fonctionnels : maintenance

- Ne pas oublier `ANALYZE` après création d'un index fonctionnel
  - les statistiques peuvent même être l'intérêt majeur (<v14)
- La fonction ne doit jamais tomber en erreur
- Si modification de la fonction
  - réindexation

Statistiques :

Après la création de l'index fonctionnel, un `ANALYZE nom_table` est conseillé : en effet, l'optimiseur ne peut utiliser les statistiques déjà connues pour le résultat d'une fonction. Par contre, PostgreSQL peut créer des statistiques sur le résultat de la fonction pour chaque ligne. Ces statistiques seront visibles dans la vue système `pg_stats` (`tablename` contient le nom de l'index, et non celui de la table !).

Ces statistiques à jour sont d'ailleurs un des intêrêts de l'index fonctionnel, même si l'index lui-même est superflu. Dans ce cas, à partir de PostgreSQL 14, on pourra utiliser `CREATE STATISTICS` sur l'expression pour ne pas avoir à créer et maintenir un index entier.

Avertissements :

La fonction ne doit jamais tomber en erreur ! Il ne faut pas tester l'index qu'avec les données en place, mais aussi avec toutes celles susceptibles de se trouver dans le champ concerné. Sinon, il y aura des refus d'insertion ou de mise à jour. Des `ANALYZE` ou `VACUUM` pourraient aussi échouer, avec de gros problèmes sur le long terme.

Si le contenu de la fonction est modifié avec `CREATE OR REPLACE FUNCTION`, il faudra impérativement réindexer, car PostgreSQL ne le fera pas automatiquement. Sans cela, les résultats des requêtes différeront selon qu'elles utiliseront ou non l'index !

5.5.7 Index couvrants : principe

- But : obtenir un *Index Only Scan*

```
CREATE UNIQUE INDEX clients_idx1
ON clients (id_client) INCLUDE (nom_client) ;
```

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

```
SELECT id_client,nom_client FROM clients WHERE id_client > 100 ;
```

- …si l'index n'est pas trop gros
  - à comparer à un index multicolonne

5.5.7.1 Principe des index couvrants

Un index couvrant (*covering index*) cherche à favoriser le nœud d'accès le plus rapide, l'*Index Only Scan* : il contient non seulement les champs servant de critères de recherche, mais aussi tous les champs résultats. Ainsi, il n'y a plus besoin d'interroger la table.

Les index couvrants peuvent être explicitement déclarés avec la clause `INCLUDE` :

```
CREATE TABLE t (id int NOT NULL, valeur int) ;

INSERT INTO t SELECT i, i*50 FROM generate_series(1,1000000) i;

CREATE UNIQUE INDEX t_pk ON t (id) INCLUDE (valeur) ;

VACUUM t ;

EXPLAIN ANALYZE SELECT valeur FROM t WHERE id = 555555 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Index Only Scan using t_pk on t  (cost=0.42..1.44 rows=1 width=4)
                                  (actual time=0.034..0.035 rows=1 loops=1)
   Index Cond: (id = 555555)
   Heap Fetches: 0
 Planning Time: 0.084 ms
 Execution Time: 0.065 ms
```

Dans cet exemple, il n'y a pas eu d'accès à la table. L'index est unique mais contient aussi la colonne `valeur`.

Noter le `VACUUM`, nécessaire pour garantir que la *visibility map* de la table est à jour et permet ainsi un *Index Only Scan* sans aucun accès à la table (clause *Heap Fetches* à 0).

Par abus de langage, on peut dire d'un index multicolonne sans clause `INCLUDE` qu'il est « couvrant » s'il répond complètement à la requête.

Dans les versions antérieures à la 11, on émulait cette fonctionnalité en incluant les colonnes dans des index multicolonne :

```
CREATE INDEX t_idx ON t (id, valeur) ;
```

Cette technique reste tout à fait valable dans les versions suivantes, car l'index multicolonne (complètement trié) peut servir de manière optimale à d'autres requêtes. Il peut même être plus petit que celui utilisant `INCLUDE`.

Un intérêt de la clause `INCLUDE` est de se greffer sur des index uniques ou de clés et d'économiser un nouvel index et un peu de place. Accessoirement, il évite le tri des champs dans la clause `INCLUDE`.

5.5.7.2 Inconvénients & limitation des index couvrants

Il faut garder à l'esprit que l'ajout de colonnes à un index (couvrant ou multicolonne) augmente sa taille. Cela peut avoir un impact sur les performances des requêtes qui n'utilisent pas les colonnes supplémentaires. Il faut également être vigilant à ce que la taille des enregistrements avec les colonnes incluses ne dépassent pas 2,6 ko. Au-delà de cette valeur, les insertions ou mises à jour échouent.

Enfin, la déduplication (apparue en version 13) n'est pas active sur les index couvrants, ce qui a un impact supplémentaire sur la taille de l'index sur le disque et en cache. Ça n'a pas trop d'importance si l'index principal contient surtout des valeurs différentes, mais s'il y en a beaucoup moins que de lignes, il serait dommage de perdre l'intérêt de la déduplication. Là encore, le planificateur peut ignorer l'index s'il est trop gros. Il faut tester avec les données réelles, et comparer avec un index multicolonne (dédupliqué).

Les méthodes d'accès aux index doivent inclure le support de cette fonctionnalité. C'est le cas pour le B-tree ou le GiST, et pour le SP-GiST en version 14.

5.5.8 Classes d'opérateurs

- Un index utilise des opérateurs de comparaison
- Texte : différentes collations = différents tris… complexes
  - Index inutilisable sur :

```
WHERE col_varchar LIKE 'chaine%'
```

- Solution : opérateur `varchar_pattern_ops` :
  - force le tri caractère par caractère, sans la collation

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

- Plus généralement :
  - nombreux autres opérateurs pour d'autres types d'index

Un opérateur sert à indiquer à PostgreSQL comment il doit manipuler un certain type de données. Il y a beaucoup d'opérateurs par défaut, mais il est parfois possible d'en prendre un autre.

Pour l'indexation, il est notamment possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison.

Le cas d'utilisation le plus fréquent dans PostgreSQL est la comparaison de chaîne `LIKE 'chaine%'`. L'indexation texte « classique » utilise la collation par défaut de la base (en France, généralement `fr_FR.UTF-8` ou `en_US.UTF-8`) ou la collation de la colonne de la table si elle diffère. Cette collation contient des notions de tri. Les règles sont différentes pour chaque collation. Et ces règles sont complexes.

Par exemple, le **ß** allemand se place entre **ss** et **t** (et ce, même en français). En danois, le tri est très particulier car le **å** et le **aa** apparaissent après le **z**.

```
-- Cette collation doit exister sur le système
CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls ORDER BY x COLLATE "da_DK";
```

```
 x
----
 s
 ss
 ß
 t
 zz
 å
 aa
```

Il faut être conscient que cela a une influence sur le résultat d'un filtrage :

```
WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls
WHERE x > 'z' COLLATE "da_DK" ;

 x
----
 aa
 å
 zz
```

Il serait donc très complexe de réécrire le `LIKE` en un `BETWEEN`, comme le font habituellement tous les SGBD : `col_texte LIKE 'toto%'` peut être réécrit comme `coltexte >= 'toto' and  coltexte < 'totp'` en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur Unicode par exemple. Même si l'index est dans la bonne collation, il n'est pas facilement utilisable :

```
CREATE INDEX ON textes (livre) ;
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                                QUERY PLAN
--------------------------------------------------------------------------
 Gather  (cost=1000.00..525328.76 rows=75173 width=123)
   Workers Planned: 2
   ->  Parallel Seq Scan on textes  (cost=0.00..516811.46 rows=31322 width=123)
         Filter: (livre ~~ 'Les misérables%'::text)
```

La classe d'opérateurs `varchar_pattern_ops` sert à changer ce comportement :

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

Ce nouvel index est alors construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient. Il devient alors trivial pour l'optimiseur de faire la réécriture :

```
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using textes_livre_idx1 on textes  (cost=0.69..70406.87 rows=75173
↪  width=123)
   Index Cond: ((livre ~>=~ 'Les misérables'::text) AND (livre ~<~ 'Les
↪  misérablet'::text))
   Filter: (livre ~~ 'Les misérables%'::text)
```

Cela convient pour un `LIKE 'critère%'`, car le début est fixe, et l'ordre de tri n'influe pas sur le résultat. (Par contre cela ne permet toujours pas d'indexer `LIKE %critère%`.) Noter la clause `Filter` qui filtre en deuxième intention ce qui a pu être trouvé dans l'index.

Il existe quelques autres cas d'utilisation d'`opclass` alternatives, notamment pour utiliser d'autres types d'index que B-tree. Deux exemples :

- indexation d'un JSON (type `jsonb`) par un index GIN :

```
CREATE INDEX ON stock_jsonb USING gin (document_jsonb jsonb_path_ops);
```

- indexation de trigrammes de textes avec le module `pg_trgm` et des index GiST :

```
CREATE INDEX ON livres USING gist (text_data gist_trgm_ops);
```

Pour plus de détails à ce sujet, se référer à la section correspondant aux classes d'opérateurs[8].

Ne mettez pas systématiquement `varchar_pattern_ops` dans tous les index de chaînes de caractère. Cet opérateur est adapté au `LIKE 'critère%` mais ne servira pas pour un tri sur la chaîne (`ORDER BY`). Selon les requêtes et volumétries, les deux index peuvent être nécessaires.

5.5.9 Conclusion

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

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

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

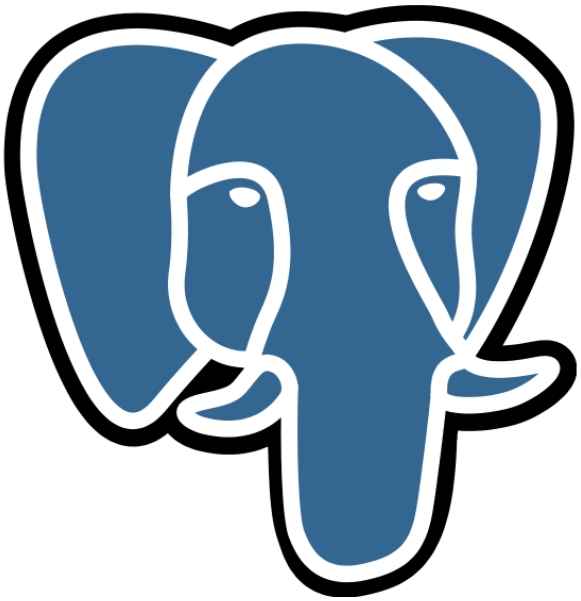
PostgreSQL fournit aussi d'autres types d'index moins utilisés, mais très précieux dans certaines situations : BRIN, GIN, GiST, etc.

[8] https://www.postgresql.org/docs/current/static/indexes-opclass.html

5.6 QUIZ

https://dali.bo/j4_quiz

6/ Comprendre EXPLAIN



6.1 INTRODUCTION

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

Face à un problème de performances, l'administrateur se retrouve assez rapidement face à une (ou plusieurs) requête(s). Une requête en soi représente très peu d'informations. Suivant la requête, des dizaines de plans peuvent être sélectionnés pour l'exécuter. Il est donc nécessaire de pouvoir trouver le plan d'exécution et de comprendre ce plan. Cela permet de mieux appréhender la requête et de mieux comprendre les pistes envisageables pour la corriger.

Ce qui suit se concentrera sur les plans d'exécution.

6.1.1 Au menu

- Exécution globale d'une requête
- Planificateur : utilité, statistiques et configuration
- `EXPLAIN`
- Nœuds d'un plan
- Outils

Nous ferons quelques rappels et approfondissements sur la façon dont une requête s'exécute globalement, et sur le planificateur : en quoi est-il utile, comment fonctionne-t-il, et comment le configurer.

Nous ferons un tour sur le fonctionnement de la commande `EXPLAIN` et les informations qu'elle fournit. Nous verrons aussi plus en détail l'ensemble des opérations utilisables par le planificateur, et comment celui-ci choisit un plan.

6.2 EXÉCUTION GLOBALE D'UNE REQUÊTE

- L'exécution peut se voir sur deux niveaux
  - niveau système
  - niveau SGBD
- De toute façon, composée de plusieurs étapes

L'exécution d'une requête peut se voir sur deux niveaux :

- ce que le système perçoit ;
- ce que le SGBD fait.

Une lenteur dans une requête peut se trouver dans l'un ou l'autre de ces niveaux.

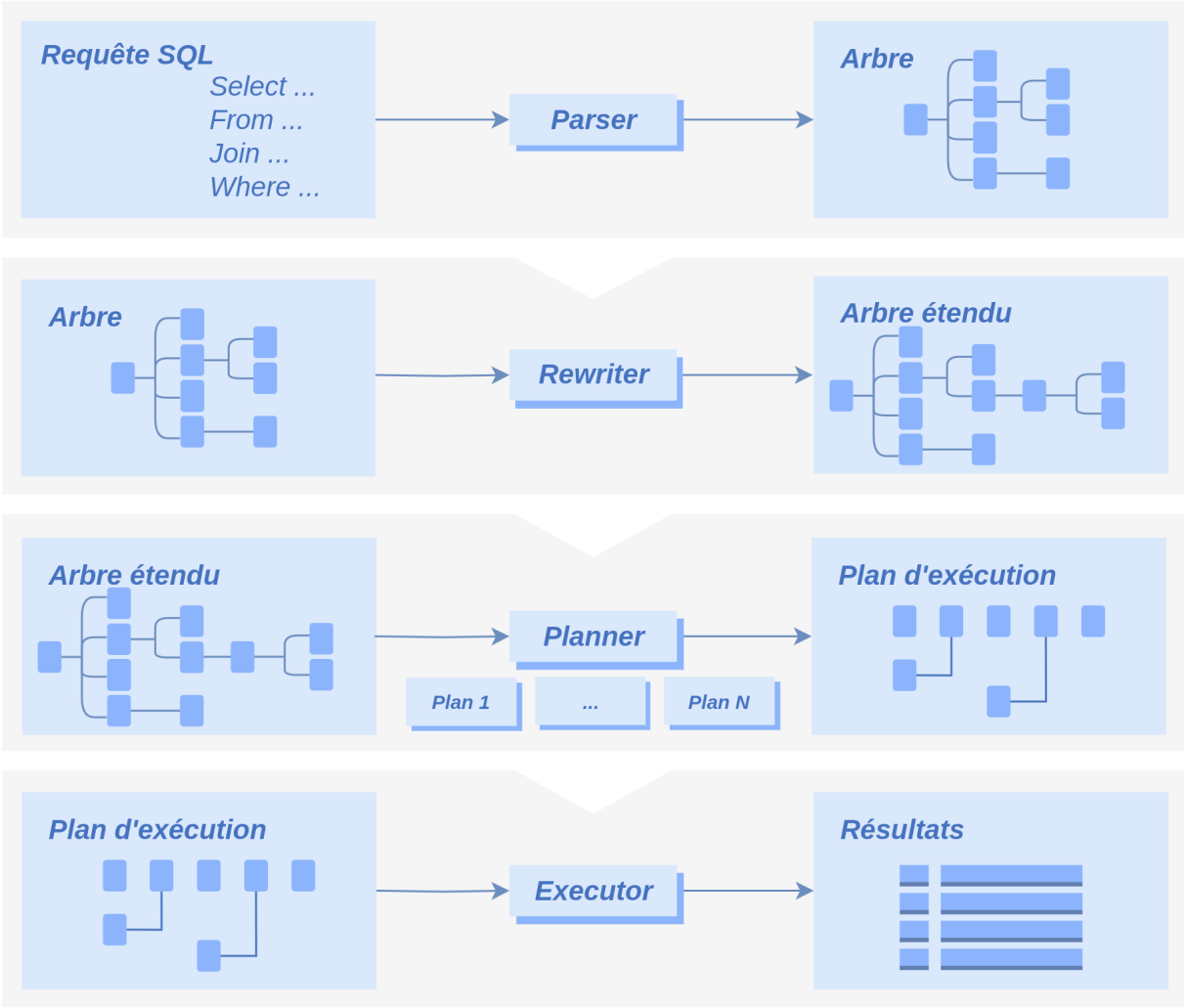
6.2.1 Niveau système

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

PostgreSQL est un système client-serveur. L'utilisateur se connecte via un outil (le client) à une base d'une instance PostgreSQL (le serveur). L'outil peut envoyer une requête au serveur, celui-ci l'exécute et finit par renvoyer les données résultant de la requête ou le statut de la requête.

Généralement, l'envoi de la requête est rapide. Par contre, la récupération des données peut poser problème si une grosse volumétrie est demandée sur un réseau à faible débit. 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.).

6.2.2 Traitement d'une requête



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

Tout d'abord, le *parser* va réaliser une analyse syntaxique de la requête.

Puis 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. Il en est de même pour une fonction SQL intégrable.

Ensuite, 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.

Enfin, l'*executer* exécute la requête.

Pour cela, il doit commencer par récupérer les verrous nécessaires sur les objets ciblés. Une fois les verrous récupérés, il exécute la requête.

Une fois la requête exécutée, il envoie les résultats à l'utilisateur.

Plusieurs goulets d'étranglement sont visibles ici. Les plus importants sont :

- la planification (à tel point qu'il est parfois préférable de ne générer qu'un sous-ensemble de plans, pour passer plus rapidement à la phase d'exécution) ;
- la récupération des verrous (une requête peut attendre plusieurs secondes, minutes, voire heures, avant de récupérer les verrous et exécuter réellement la requête) ;
- l'exécution de la requête ;
- l'envoi des résultats à l'utilisateur.

Il est possible de tracer l'exécution des différentes étapes grâce aux options `log_parser_stats`, `log_planner_stats` et `log_executor_stats`. Voici un exemple complet :

- Mise en place de la configuration sur la session :

```
SET log_parser_stats TO on;
SET log_planner_stats TO on;
SET log_executor_stats TO on;
SET client_min_messages TO log;
```

- Exécution de la requête :

```
SELECT fonction, COUNT(*) FROM employes_big GROUP BY fonction ORDER BY fonction;
```

- Trace du *parser* :

```
LOG:  PARSER STATISTICS
DÉTAIL : ! system usage stats:
!       0.000026 s user, 0.000017 s system, 0.000042 s elapsed
!       [0.013275 s user, 0.008850 s system total]
!       17152 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/3 [0/575] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
LOG:  PARSE ANALYSIS STATISTICS
DÉTAIL : ! system usage stats:
!       0.000396 s user, 0.000263 s system, 0.000660 s elapsed
!       [0.013714 s user, 0.009142 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/32 [0/607] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace du *rewriter* :

```
LOG:  REWRITER STATISTICS
DÉTAIL : ! system usage stats:
!       0.000010 s user, 0.000007 s system, 0.000016 s elapsed
!       [0.013747 s user, 0.009165 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/1 [0/608] page faults/reclaims, 0 [0] swaps
```

```
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace du *planner* :

```
DÉTAIL : ! system usage stats:
!       0.000255 s user, 0.000170 s system, 0.000426 s elapsed
!       [0.014021 s user, 0.009347 s system total]
!       19476 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/25 [0/633] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       0/0 [5/0] voluntary/involuntary context switches
```

- Trace de l'*executer* :

```
LOG:  EXECUTOR STATISTICS
DÉTAIL : ! system usage stats:
!       0.044788 s user, 0.004177 s system, 0.131354 s elapsed
!       [0.058917 s user, 0.013596 s system total]
!       46268 kB max resident size
!       0/0 [0/368] filesystem blocks in/out
!       0/468 [0/1124] page faults/reclaims, 0 [0] swaps
!       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!       4/16 [9/16] voluntary/involuntary context switches
```

- Résultat de la requête :

```
     fonction      | count
-------------------+--------
 Commercial        |      2
 Comptable         |      1
 Consultant        | 499005
 Développeur       |      2
 Directeur Général |      1
 Responsable       |      4
```

6.2.3 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.

6.3 QUELQUES DÉFINITIONS

- Prédicat
  - filtre de la clause `WHERE`
  - conditions de jointure
- Sélectivité
  - % de lignes retournées après application d'un prédicat
- Cardinalité
  - nombre de lignes d'une table
  - nombre de lignes retournées après filtrages

Un prédicat est une condition de filtrage présente dans la clause `WHERE` d'une requête. Par exemple `colonne = valeur`. On parle aussi de prédicats de jointure pour les conditions de jointures présentes dans la clause `WHERE` ou suivant la clause `ON` d'une jointure.

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 de la sortie d'un nœud. Elle représente aussi le nombre de lignes retournées par la lecture d'une table après application d'un ou plusieurs prédicats.

6.3.1 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)

Les deux volumétries différentes vont permettre de mettre en évidence certains effets.

6.3.2 Jeu de tests (schéma)

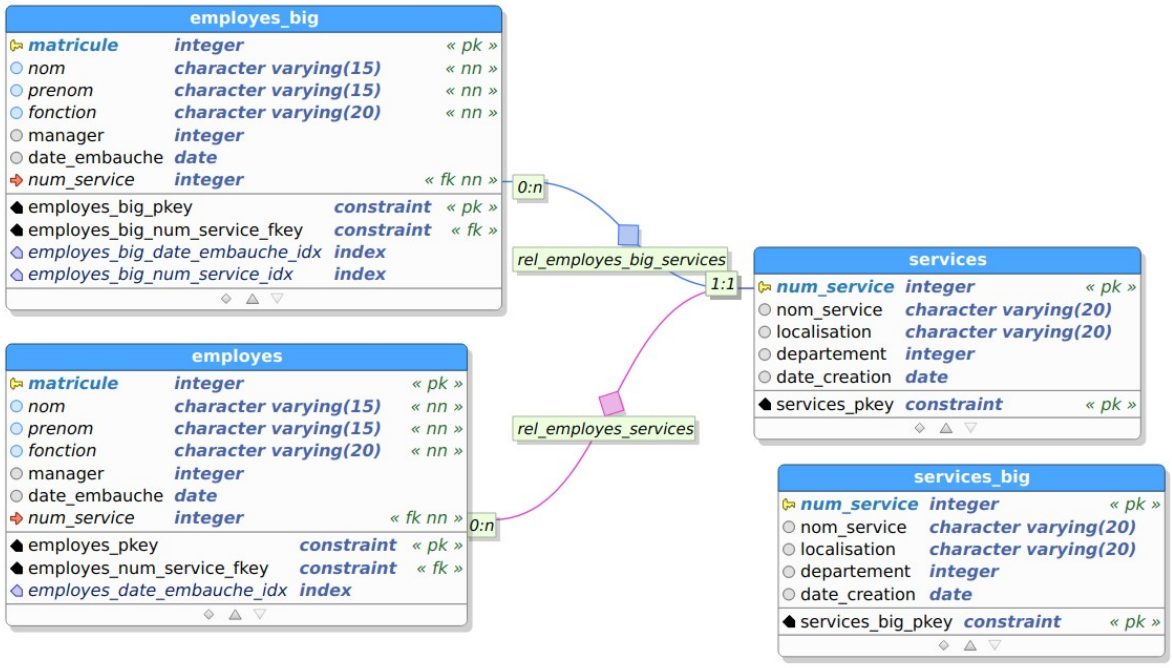


Figure 6/ .1: Tables employés & services

Les tables suivantes nous serviront d'exemple par la suite. Le script de création se télécharge et s'installe ainsi dans une nouvelle base **employes** :

```
curl -kL https://dali.bo/tp_employes_services -o employes_services.sql
createdb employes
psql employes < employes_services.sql
```

Les quelques tables occupent environ 80 Mo sur le disque.

```
-- 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;

-- 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;
```

6.3.3 Requête étudiée

```
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.

6.3.4 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.

6.4 PLANIFICATEUR

Rappels :

- SQL est un langage déclaratif
- Planificateur : trouver le meilleur plan
- Énumère tous les plans d'exécution possible
  - tous ou presque…
- Statistiques + configuration + règles → coût
- Coût le plus bas = meilleur plan

Le but du planificateur est assez simple. Pour une requête, il existe de nombreux plans d'exécution possibles. Il va donc tenter d'énumérer tous les plans d'exécution possibles ; même si leur nombre devient vite colossal dans une requête complexe : chaque table peut être accédée selon différents plans, selon l'un ou l'autre critère ou une combinaison, les algorithmes de jointure possibles sont multiples, etc.

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, à priori celui qui sera le plus rapide pour la requête demandée. (En toute rigueur, pour réduire le nombre de plans très voisins à étudier, des plans de coûts différents de 1% près peuvent être considérés équivalents[1], et le coût de démarrage peut alors les départager.)

Le coût d'un plan est une valeur calculée sans unité ni signification physique.

[1] https://doxygen.postgresql.org/pathnode_8c_source.html#l00151

6.4.1 Règles

- Règle 1 : récupérer le bon résultat
- Règle 2 : 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 : un résultat rapide mais faux n'a aucun intérêt ;
- 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 souvent privilégier les opérations séquentielles aux dépens des opérations aléatoires (qui demandent un déplacement de la tête de disque, opération la plus coûteuse sur les disques magnétiques).

6.4.2 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.

6.4.3 Optimisations

- À partir du modèle de données
  - suppression de jointures externes inutiles
- Transformation des sous-requêtes
  - certaines sous-requêtes transformées en jointures
  - ex : `critere IN (SELECT ...)`
- Appliquer les prédicats le plus tôt possible
  - réduit le jeu de données manipulé
  - CTE : barrière avant la v12 !
- 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. C'est généralement une bonne chose.

Mais en cas de problème, il est possible d'utiliser une CTE matérialisée (*Common Table Expression*, clause `WITH … AS MATERIALIZED (…)`) pour bloquer cette optimisation et forcer PostgreSQL à exécuter le contenu de la requête en premier[2]. En versions 12 et ultérieures, une CTE est par défaut non matérialisée et donc intégrée avec le reste de la requête (du moins dans les cas simples comme ci-dessus), comme une sous-requête. On retombe exactement sur le plan précédent :

```
-- v12 : CTE sans MATERIALIZED (comportement par défaut)
EXPLAIN
  WITH e AS ( SELECT * FROM employes WHERE num_service = 4 )
  SELECT MAX(date_embauche)
  FROM e
  WHERE e.date_embauche < '2006-01-01';

                                QUERY PLAN
--------------------------------------------------------------------------
 Aggregate  (cost=1.21..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))
```

Pour recréer la « barrière d'optimisation », il est nécessaire d'ajouter le mot-clé `MATERIALIZED` :

```
-- v12 : CTE avec MATERIALIZED
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)
```

La CTE est alors intégralement exécutée avec son filtre propre, avant que le deuxième filtre soit appliqué dans un autre nœud. Jusqu'en version 11 incluse, ce dernier comportement était celui par défaut, et les CTE étaient une source fréquente de problèmes de performances.

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;
```

[2] https://docs.postgresql.fr/current/queries-with.html#QUERIES-WITH-CTE-MATERIALIZATION

```
$BODY$
  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 20 sur ce jeu de données 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 flagrante. 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:
   Buffers: shared hit=61
 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.

6.4.4 Décisions

L'optimiseur doit choisir :

- Stratégie d'accès aux lignes
  - parcours de table, d'index, fonction, etc.
- Stratégie d'utilisation des jointures
  - ordre
  - ordre des tables jointes
  - type (*Nested Loop*, *Merge/Sort Join*, *Hash Join*…)
- Stratégie d'agrégation
  - brut, trié, haché
- En version parallélisée ?
- Tenir compte de la consommation mémoire

Pour exécuter une requête, le planificateur va utiliser des opérations. Pour lire des lignes, il peut :

- utiliser un parcours de table (*Seq Scan*) qui ne lit que celle-ci ;
- parcourir un index et revenir chercher les lignes dans la table (*Index Scan* ou *Bitmap Scan*) ;
- ou se contenter de l'index s'il suffit (*Index Only Scan*).

Il existe encore d'autres types de parcours. Les accès aux tables et index sont généralement les premières opérations utilisées.

Pour joindre les tables, l'ordre est très important pour essayer de réduire la masse des données manipulées. Les jointures se font toujours entre deux des tables impliquées, pas plus ; ou entre une table et le résultat d'un nœud, ou entre les résultats de deux nœuds.

Pour la jointure elle-même, il existe plusieurs méthodes différentes : boucles imbriquées (*Nested Loops*), hachage (*Hash Join*), tri-fusion (*Merge Join*)…

Il existe également plusieurs algorithmes d'agrégation des lignes. Un tri peut être nécessaire pour une jointure, une agrégation, ou pour un `ORDER BY`, et là encore il y a plusieurs algorithmes possibles. L'optimiseur peut aussi décider d'utiliser un index (déjà trié) pour éviter ce tri.

Certaines des opérations ci-dessus sont parallélisables. Certaines sont aussi susceptibles de consommer beaucoup de mémoire, l'optimiseur doit en tenir compte.

6.4.5 Parallélisation

- Processus supplémentaires pour certains nœuds
  - parer à la limitation par le CPU
- En lecture (sauf exceptions)
- Parcours séquentiel
- Jointures : *Nested Loop* / *Hash Join* / *Merge Join*
- Agrégats
- Parcours d'index (B-Tree uniquement)
- Création d'index B-Tree
- Certaines créations de table et vues matérialisées
- `DISTINCT` (v15)

Principe :

À partir d'une certaine quantité de données à traiter par un nœud, un ou plusieurs processus auxiliaires (*parallel workers*) apparaissent pour répartir la charge sur d'autres processeurs. Sans cela, une requête n'est traitée que par un seul processus sur un seul processeur.

Il ne s'agit pas de lire une table avec plusieurs processus mais de répartir le traitement des lignes. La parallélisation n'est donc utile que si le CPU est le facteur limitant. Par exemple, un simple `SELECT` sur une grosse table sans `WHERE` ne mènera pas à un parcours parallélisé.

La parallélisation concerne en premier lieu les parcours de tables (*Seq Scan*), les jointures (*Nested Loop*, *Hash Join*, *Merge Join*), ainsi que certaines fonctions d'agrégat (comme `min`, `max`, `avg`, `sum`, etc.) ; mais encore les parcours d'index B-Tree (*Index Scan*, *Index Only Scan* et *Bitmap Scan*) La parallélisation est en principe disponible pour les autres types d'index, mais ils n'en font pas usage pour l'instant.

La parallélisation ne concerne encore que les opérations en lecture. Il y a des exceptions, comme la création des index B-Tree de façon parallélisée. Certaines créations de table avec `CREATE TABLE … AS`, `SELECT … INTO` sont aussi parallélisables, ainsi que `CREATE MATERIALIZED VIEW`.

En version 15, il devient possible de paralléliser des clauses `DISTINCT`.

Paramétrage :

Le paramétrage s'est affiné au fil des versions. Les paramètres suivants sont valables à partir de la version 13.

Le paramètre `max_parallel_workers_per_gather` (2 par défaut) désigne le nombre de processus auxiliaires maximum d'un nœud d'une requête. `max_parallel_maintenance_workers` (2 par défaut)

est l'équivalent dans les opérations de maintenance (réindexation notamment). Trop de processus parallèles peuvent mener à une saturation de CPU ; l'exécuteur de PostgreSQL ne lancera donc pas plus de `max_parallel_workers` processus auxiliaires simultanés (8 par défaut), lui-même limité par `max_worker_processes` (8 par défaut). En pratique, on ajustera le nombre de *parallel workers* en fonction des CPU de la machine et de la charge attendue.

La mise en place de l'infrastructure de parallélisation a un coût, défini par `parallel_setup_cost` (1000 par défaut), et des tailles de table ou index minimales, en-dessous desquels la parallélisation n'est pas envisagée.

La plupart de ces paramètres peuvent être modifiés dans une sessions par `SET`.

6.4.6 Limites actuelles de la parallélisation

- Lourd à déclencher
- 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[3], notamment :

- elle est assez lourde à mettre en place, elle a donc un coût d'entrée qui la rend inutile quand il y a peu de lignes ;
- 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`, parallélisable à partir de la v11 ; ou le niveau d'isolation *serializable*: avant la v12, il ne permet aucune parallélisation.

[3] https://docs.postgresql.fr/current/when-can-parallel-query-be-used.html

6.5 MÉCANISME DE COÛTS & STATISTIQUES

- Modèle basé sur les coûts
  - quantifier la charge pour répondre à une requête
- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne issue d'une lecture de table ou d'index
  - appliquer un opérateur

L'optimiseur statistique de PostgreSQL utilise un modèle de calcul de coût. Les coûts calculés sont des indications arbitraires sur la charge nécessaire pour répondre à une requête. Chaque facteur de coût représente une unité de travail : lecture d'un bloc, manipulation des lignes en mémoire, application d'un opérateur sur des données.

6.5.1 Coûts unitaires

- Coûts à connaître :
  - accès au disque séquentiel / non séquentiel
  - traitement d'un enregistrement issu d'une table
  - traitement d'un enregistrement issu d'un index
  - application d'un opérateur
  - traitement d'un enregistrement dans un parcours parallélisé
  - mise en place d'un parcours parallélisé
  - mise en place du JIT, du parallélisme…
- Chaque coût = un paramètre
  - modifiable 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. Ces coûts sont arbitraires, à ne comparer qu'entre eux, et ne sont pas liés directement à des caractéristiques physiques du serveur.

- `seq_page_cost` (1 par défaut) 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` (4 par défaut) 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 pour un SSD ;
- `cpu_tuple_cost` (0,01 par défaut) représente le coût relatif de la manipulation d'une ligne en mémoire ;
- `cpu_index_tuple_cost` (0,005 par défaut) 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` (0,1 par défaut) indique le coût estimé du transfert d'une ligne d'un processus à un autre ;
- `parallel_setup_cost` (1000 par défaut) 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` (100 000 par défaut), `jit_inline_above_cost` (500 000 par défaut), `jit_optimize_above_cost` (500 000 par défaut) représentent les seuils d'activation de divers niveaux du JIT (*Just In Time* ou compilation à la volée des requêtes), outil 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, d'une carte RAID équipée d'un cache important ou de SSD. 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 modifiables dans une session. Ils peuvent être modifiés dynamiquement par l'application avec l'ordre `SET` pour des requêtes bien particulières, pour éviter de toucher au paramétrage général.

6.6 STATISTIQUES

- Combien de lignes va-t-on traiter ?
- Toutes les décisions du planificateur se basent sur les statistiques
  - le choix du parcours
  - comme le choix des jointures
- Mettre à jour les statistiques sur les données :
  - `ANALYZE`
- Sans bonnes statistiques, pas de bons plans !

Connaître le coût unitaire de traitement d'une ligne est une bonne chose, mais si on ne sait pas le nombre de lignes à traiter, on ne peut pas calculer le coût total. Le planificateur se base alors principalement sur les statistiques pour ses décisions. Avec ces informations et le paramétrage, l'optimiseur saura par exemple calculer le ratio d'un filtre et décider s'il faut passer par un index, ou calculer le ratio d'une jointure pour choisir la stratégie de jointure. 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 exécutée manuellement ou automatiquement (le démon autovacuum s'en occupe généralement, mais compléter avec une tâche planifiée avec cron ou les tâches planifiées sous Windows est possible). Nous allons voir comment les consulter.

6.6.1 Utilisation des statistiques

- Les statistiques indiquent :
  - la cardinalité d'un filtre → stratégie d'accès
  - la cardinalité d'une jointure → algorithme de jointure
  - la cardinalité d'un regroupement → 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 répar-

tition 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, le niveau de corrélation entre valeurs et place sur le disque, etc.

L'optimiseur peut donc déterminer la sélectivité d'un filtre (prédicat d'une clause `WHERE` ou une condition de jointure) et donc quelle sera la quantité de données récupérées par la lecture d'une table en utilisant le filtre évalué.

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 499 004 + 11 correspond bien aux 499 015 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.

6.6.2 Statistiques des tables et index

- Dans `pg_class`
  - `relpages` : taille
  - `reltuples` : lignes

L'optimiseur a besoin de deux données statistiques pour une table ou un index : sa taille physique et le nombre de lignes portées par l'objet.

Ces deux données statistiques sont stockées dans la table `pg_class`. La taille de la table ou de l'index est exprimée en nombre de blocs de 8 ko et stockée dans la colonne `relpages`. La cardinalité de la table ou de l'index, c'est-à-dire le nombre de lignes, est stockée dans la colonne `reltuples`.

L'optimiseur utilisera ces deux informations pour apprécier la cardinalité de la table en fonction de sa volumétrie courante en calculant sa densité estimée puis en utilisant cette densité multipliée par le nombre de blocs actuel de la table pour estimer le nombre de lignes réel de la table :

```
density = reltuples / relpages;
tuples = density * curpages;
```

6.6.3 Statistiques : mono-colonne

- Nombre de valeurs distinctes
- Nombre d'éléments qui n'ont pas de valeur (`NULL`)
- Largeur d'une colonne
- Distribution des données
  - tableau des valeurs les plus fréquentes
  - histogramme de répartition des valeurs

Au niveau d'une colonne, plusieurs données statistiques sont stockées :

- le nombre de valeurs distinctes ;
- le nombre d'éléments qui n'ont pas de valeur (`NULL`) ;
- la largeur moyenne des données portées par la colonne ;
- le facteur de corrélation entre l'ordre des données triées et la répartition physique des valeurs dans la table ;
- la distribution des données.

La distribution des données est représentée sous deux formes qui peuvent être complémentaires. Tout d'abord, un tableau de répartition permet de connaître les valeurs les plus fréquemment rencontrées et la fréquence d'apparition de ces valeurs. Un histogramme de distribution des valeurs rencontrées permet également de connaître la répartition des valeurs pour la colonne considérée.

6.6.4 Stockage des statistiques mono-colonne

- Stockage dans `pg_statistic`
  - préférer la vue `pg_stats`
- Une table nouvellement créée n'a pas de statistiques
- Utilisation :

```
SELECT * FROM pg_stats
  WHERE  schemaname = 'public'
  AND tablename     = 'employes'
  AND attname       = 'date_embauche' \gx
```

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

6.6.5 Vue pg_stats

```
-[ 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   | ¤
```

Ce qui prècède est le contenu de `pg_stats` pour la colonne `date_embauche` de la table `employes`.

Trois champs identifient cette 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 COLU`
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 `<`.

6.6.6 Statistiques : multicolonnes

- Pas par défaut
- `CREATE STATISTICS`
- Trois types de statistique
  - nombre de valeurs distinctes
  - dépendances fonctionnelles
  - liste MCV

Par défaut, la commande `ANALYZE` de PostgreSQL calcule des statistiques mono-colonnes uniquement. Elle peut aussi calculer certaines statistiques multicolonnes. En effet, les valeurs des colonnes ne sont pas indépendantes et peuvent varier ensemble.

Pour cela, il est nécessaire de créer un objet statistique avec l'ordre SQL `CREATE STATISTICS`. Cet objet indique les colonnes concernées ainsi que le type de statistique souhaité.

PostgreSQL supporte trois types de statistiques pour ces objets :

- `ndistinct` pour le nombre de valeurs distinctes sur ces colonnes ;
- `dependencies` pour les dépendances fonctionnelles ;
- `mcv` pour une liste des valeurs les plus fréquentes (depuis la version 12).

Dans tous les cas, cela peut permettre d'améliorer fortement les estimations de nombre de lignes, ce qui ne peut qu'amener de meilleurs plans d'exécution.

Prenons un exemple. On peut voir sur ces deux requêtes que les statistiques sont à jour :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris';

                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..786.05 rows=10013 width=28)
                     (actual time=0.019..4.773 rows=10001 loops=1)
   Filter: ((localisation)::text = 'Paris'::text)
   Rows Removed by Filter: 30003
 Planning time: 0.863 ms
 Execution time: 5.289 ms

EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE departement=75;

                              QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..786.05 rows=10013 width=28)
                     (actual time=0.020..7.013 rows=10001 loops=1)
   Filter: (departement = 75)
   Rows Removed by Filter: 30003
 Planning time: 0.219 ms
 Execution time: 7.785 ms
```

Cela fonctionne bien, *i.e.* l'estimation du nombre de lignes (10013) est très proche de la réalité (10001) dans le cas spécifique où le filtre se fait sur une seule colonne. Par contre, si le filtre se fait sur le lieu Paris *et* le département 75, l'estimation diffère d'un facteur 4, à 2506 lignes :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris'
  AND departement=75;

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=2506 width=28)
                    (actual time=0.032..7.081 rows=10001 loops=1)
   Filter: (((localisation)::text = 'Paris'::text) AND (departement = 75))
   Rows Removed by Filter: 30003
 Planning time: 0.257 ms
 Execution time: 7.767 ms
```

En fait, il y a une dépendance fonctionnelle entre ces deux colonnes (être dans le département 75 implique d'être à Paris), mais PostgreSQL ne le sait pas car ses statistiques sont mono-colonnes par défaut. Pour avoir des statistiques sur les deux colonnes, il faut créer un objet statistique dédié :

```
CREATE STATISTICS stat_services_big (dependencies)
  ON localisation, departement
  FROM services_big;
```

Après création de l'objet, il ne faut pas oublier de calculer les statistiques :

```
ANALYZE services_big;
```

Ceci fait, on peut de nouveau regarder les estimations :

```
EXPLAIN (ANALYZE)
  SELECT * FROM services_big
  WHERE localisation='Paris'
  AND departement=75;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=10038 width=28)
                      (actual time=0.008..6.249 rows=10001 loops=1)
   Filter: (((localisation)::text = 'Paris'::text) AND (departement = 75))
   Rows Removed by Filter: 30003
 Planning time: 0.121 ms
 Execution time: 6.849 ms
```

Cette fois, l'estimation (10038 lignes) est beaucoup plus proche de la réalité (10001). Cela ne change rien au plan choisi dans ce cas précis, mais dans certains cas la différence peut être énorme.

Maintenant, prenons le cas d'un regroupement :

```
EXPLAIN (ANALYZE)
  SELECT localisation, COUNT(*)
  FROM   services_big
  GROUP BY localisation ;
```

```
                               QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=886.06..886.10 rows=4 width=14)
          (actual time=12.925..12.926 rows=4 loops=1)
   Group Key: localisation
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=6)
                         (actual time=0.010..2.779 rows=40004 loops=1)
 Planning time: 0.162 ms
 Execution time: 13.033 ms
```

L'estimation du nombre de lignes pour un regroupement sur une colonne est très bonne.

À présent, testons avec un regroupement sur deux colonnes :

```
EXPLAIN (ANALYZE)
  SELECT localisation, departement, COUNT(*)
  FROM services_big
  GROUP BY localisation, departement;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=986.07..986.23 rows=16 width=18)
           (actual time=15.830..15.831 rows=4 loops=1)
   Group Key: localisation, departement
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=10)
                           (actual time=0.005..3.094 rows=40004 loops=1)
 Planning time: 0.102 ms
 Execution time: 15.860 ms
```

Là aussi, on constate un facteur d'échelle de 4 entre l'estimation (16 lignes) et la réalité (4). Et là aussi, un objet statistique peut fortement aider :

```
DROP STATISTICS IF EXISTS stat_services_big;

CREATE STATISTICS stat_services_big (dependencies,ndistinct)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;

EXPLAIN (ANALYZE)
  SELECT localisation, departement, COUNT(*)
  FROM services_big
  GROUP BY localisation, departement;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=986.07..986.11 rows=4 width=18)
          (actual time=14.351..14.352 rows=4 loops=1)
   Group Key: localisation, departement
   Batches: 1  Memory Usage: 24kB
   ->  Seq Scan on services_big  (cost=0.00..686.04 rows=40004 width=10)
                           (actual time=0.013..2.786 rows=40004 loops=1)
 Planning time: 0.305 ms
 Execution time: 14.413 ms
```

L'estimation est bien meilleure grâce aux statistiques spécifiques aux deux colonnes.

PostgreSQL 12 ajoute la méthode MCV (*most common values*) qui permet d'aller plus loin sur l'estimation du nombre de lignes. Notamment, elle permet de mieux estimer le nombre de lignes à partir d'un prédicat utilisant les opérations `<` et `>`. En voici un exemple :

```
DROP STATISTICS stat_services_big;

EXPLAIN (ANALYZE)
  SELECT *
  FROM services_big
  WHERE localisation='Paris'
  AND  departement > 74 ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
  Seq Scan on services_big  (cost=0.00..886.06 rows=2546 width=28)
                           (actual time=0.031..19.569 rows=10001 loops=1)
```

```
   Filter: ((departement > 74) AND ((localisation)::text = 'Paris'::text))
   Rows Removed by Filter: 30003
 Planning Time: 0.186 ms
 Execution Time: 21.403 ms
```

Il y a donc une erreur d'un facteur 4 (2 546 lignes estimées contre 10 001 réelles) que l'on peut corriger :

```
CREATE STATISTICS stat_services_big (mcv)
  ON localisation, departement
  FROM services_big;

ANALYZE services_big ;

EXPLAIN (ANALYZE)
  SELECT *
  FROM services_big
  WHERE localisation='Paris'
  AND  departement > 74;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on services_big  (cost=0.00..886.06 rows=10030 width=28)
                           (actual time=0.017..18.092 rows=10001 loops=1)
   Filter: ((departement > 74) AND ((localisation)::text = 'Paris'::text))
   Rows Removed by Filter: 30003
 Planning Time: 0.337 ms
 Execution Time: 18.907 ms
```

Une limitation existait avant PostgreSQL 13 : un seul objet statistique pouvait être utilisé par table et une requête ne pouvait utiliser qu'un seul objet statistique pour chaque table.

6.6.7 Statistiques sur les expressions

```
CREATE STATISTICS employe_big_extract
ON extract('year' FROM date_embauche) FROM employes_big;
```

- Résout le problème des statistiques difficiles à estimer
- Pas créées par défaut
- Ne pas oublier `ANALYZE`
- (Avant v14 : index fonctionnel nécessaire)

À partir de la version 14, il est possible de créer un objet statistique sur des expressions.

Les statistiques sur des expressions permettent de résoudre le problème des estimations sur les résultats de fonctions ou d'expressions. C'est un problème récurrent et impossible à résoudre sans statistiques dédiées.

On voit dans cet exemple que les statistiques pour l'expression `extract('year' from data_embauche)` sont erronées :

```
EXPLAIN SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..9552.15 rows=2495 width=40)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
       (cost=0.00..8302.65 rows=1040 width=40)
         Filter: (date_part('year'::text,
                 (date_embauche)::timestamp without time zone)
                 = '2006'::double precision)
```

L'ordre suivant crée des statistiques supplémentaires sur les résultats de l'expression. Les résultats sont calculés pour un échantilon des lignes et collectés en même temps ques les statistiques mono-colonnes habituelles. Il ne faut pas oublier de lancer manuellement la collecte la première fois :

```
CREATE STATISTICS employe_big_extract
    ON extract('year' FROM date_embauche) FROM employes_big;
ANALYZE employes_big;
```

Les estimations du plan sont désormais correctes :

```
EXPLAIN SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..12149.22 rows=498998 width=40)
   Filter: (EXTRACT(year FROM date_embauche) = '2006'::numeric)
```

Cet objet statistique apparaît dans `psql` dans un `\d employes_big` :

```
…
Objets statistiques :
    "public.employe_big_extract" ON EXTRACT(year FROM date_embauche) FROM
↪   employes_big
```

Avant la version 14, calculer ces statistiques est possible indirectement, en créant un index fonctionnel sur l'expression, ce qui provoque le calcul de statistiques dédiées. Or l'index fonctionnel n'a pas toujours d'intérêt : dans l'exemple précédent, presque toutes les dates d'embauche sont en 2006, et l'index ne serait donc pas utilisé.

6.6.8 Catalogues pour les statistiques étendues

Vues disponibles :

- `pg_stats_ext`
- `pg_stats_ext_exprs` (pour les expressions, v14)

Les statistiques étendues sont visibles dans des tables comme `pg_statistic_ext` et `pg_statistic_ext_data` (à partir de la version 12), mais il est plus aisé de passer par la vue `pg_stats_ext` :

```
SELECT * FROM pg_stats_ext \gx
```

```
-[ RECORD 1 ]----------+----------------------------------------
schemaname             | public
tablename              | employes_big
statistics_schemaname  | public
statistics_name        | employe_big_extract
statistics_owner       | postgres
attnames               |
exprs                  | {"EXTRACT(year FROM date_embauche)"}
kinds                  | {e}
inherited              | f
n_distinct             |
dependencies           |
most_common_vals       |
most_common_val_nulls  |
most_common_freqs      |
most_common_base_freqs |
-[ RECORD 2 ]----------+----------------------------------------
schemaname             | public
tablename              | services_big
statistics_schemaname  | public
statistics_name        | stat_services_big
statistics_owner       | postgres
attnames               | {localisation,departement}
exprs                  |
kinds                  | {m}
inherited              | f
n_distinct             |
dependencies           |
most_common_vals       | { {Paris,75},{Limoges,52},{Rennes,40},{Nantes,44} }
most_common_val_nulls  | { {f,f},{f,f},{f,f},{f,f} }
most_common_freqs      |
 ↪  {0.2512,0.2511666666666665,0.24886666666666668,0.24876666666666666}
most_common_base_freqs |
 ↪  {0.06310144,0.06308469444444444,0.061934617777777784,0.06188485444444444}
```

On voit qu'il n'y a pas d'informations détaillées sur les statistiques sur expression (première ligne). Elles sont visibles dans `pg_stats_ext_exprs` (à partir de la version 14) :

```
SELECT * FROM pg_stats_ext \gx
```

```
SELECT * FROM pg_stats_ext_exprs  \gx
-[ RECORD 1 ]----------+--------------------------------
schemaname             | public
tablename              | employes_big
statistics_schemaname  | public
statistics_name        | employe_big_extract
statistics_owner       | postgres
expr                   | EXTRACT(year FROM date_embauche)
inherited              | f
null_frac              | 0
avg_width              | 8
```

```
n_distinct             | 1
most_common_vals       | {2006}
most_common_freqs      | {1}
histogram_bounds       |
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

6.6.9 ANALYZE

- `ANALYZE [ VERBOSE ] [ table [ ( colonne [, ...] ) ] [, ...] ]`
  - sans argument : base entière
  - avec argument : table complète ou certaines colonnes
- Un échantillon de table → statistiques
- Table vide : conserve les anciennes statistiques
- Nouvelle table : valeur par défaut

`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 ou plusieurs arguments sont donnés, ils doivent correspondre au nom des tables à analyser (en les séparant par des virgules). 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 le catalogue système nommé `pg_statistics`, accessible par la vue `pg_stats`, comme vu précédemment.

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.
À partir de la version 14, lors de la planification, une table vide est bien considérée comme telle au niveau de son nombre de lignes, mais avec 10 blocs au minimum.
Pour les versions antérieures, une nouvelle table (nouvelle dans le sens `CREATE TABLE` mais aussi `VACUUM FULL` et `TRUNCATE`) n'est jamais considérée vide par l'optimiseur, qui utilise alors des valeurs par défaut dépendant de la largeur moyenne d'une ligne et d'un nombre arbitraire de blocs.

6.6.10 Fréquence d'analyse

- Dépend principalement de la fréquence des requêtes DML
- Autovacuum fait l'`ANALYZE` mais…
  - pas sur les tables temporaires
  - pas assez rapidement parfois
- Cron
  - `psql`
  - ou `vacuumdb --analyze-only`

Les statistiques doivent être mises à jour fréquemment. La fréquence exacte dépend surtout de la fréquence des requêtes d'insertion, de modification ou de suppression des lignes des tables. Néanmoins, un `ANALYZE` tous les jours semble un minimum, sauf cas spécifique.

L'exécution périodique peut se faire avec cron (ou les tâches planifiées sous Windows). Il n'existe pas d'outil PostgreSQL pour lancer un seul `ANALYZE`, mais l'outil `vacuumdb` a une option `--analyze-only`. Ces deux ordres sont équivalents :

```
vacuumdb --analyze-only -t matable -d mabase
```

```
psql -c "ANALYZE matable" -d mabase
```

Le démon `autovacuum` fait aussi des `ANALYZE`. La fréquence dépend de sa configuration. Cependant, il faut connaître deux particularités de cet outil :

- Ce démon a sa propre connexion à la base. Il ne peut donc pas voir les tables temporaires appartenant aux autres sessions. Il ne sera donc pas capable de mettre à jour leurs statistiques.
- Après une insertion ou une mise à jour massive, autovacuum ne va pas forcément lancer un `ANALYZE` immédiat. En effet, il ne cherche les tables à traiter que toutes les minutes (par défaut). Si, après la mise à jour massive, une requête est immédiatement exécutée, il y a de fortes chances qu'elle s'exécute avec des statistiques obsolètes. Il est préférable dans ce cas de lancer un `ANALYZE` manuel sur la ou les tables concernées juste après l'insertion ou la mise à jour massive. Pour des mises à jour plus régulières dans une grande table, il est assez fréquent qu'on doive réduire la valeur d'`autovacuum_analyze_scale_factor` (par défaut 10 % de la table doit être modifié pour déclencher automatiquement un `ANALYZE`).

6.6.11 Échantillon statistique

- `default_statistics_target` = 100
  - × 300 → 30 000 lignes au hasard
- Configurable par colonne

```
ALTER TABLE matable ALTER COLUMN nomchamp SET STATISTICS 300 ;
```

- Configurable par statistique étendue (v13+)

```
ALTER STATISTICS nom SET STATISTICS valeur ;
```

- `ANALYZE` ensuite
- Coût : temps de planification

Par défaut, un `ANALYZE` récupère 30 000 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` (100 par défaut). La taille de l'échantillon est de 300 fois `default_statistics_target`.

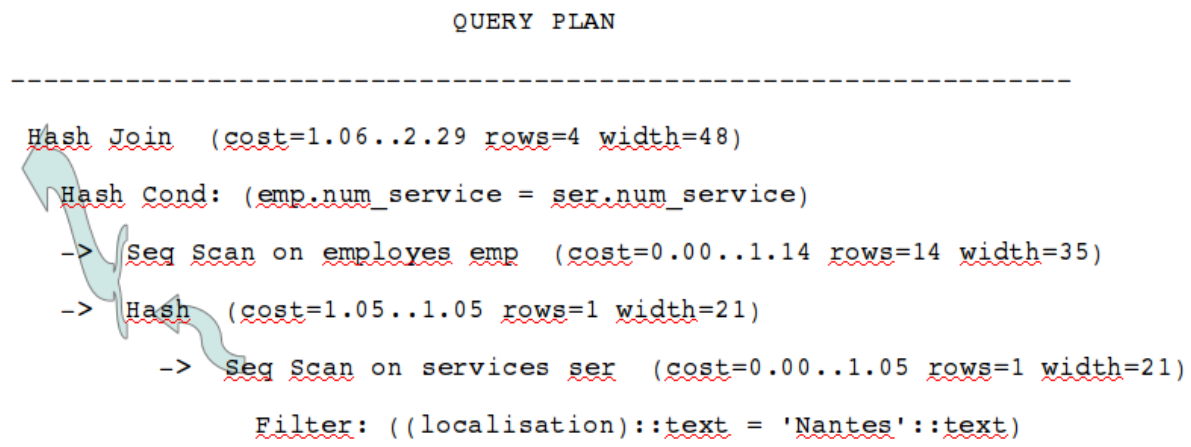
Si on l'augmente, les statistiques seront plus précises grâce à un échantillon plus important. Mais de ce fait, elles seront plus longues à calculer, prendront plus de place sur le disque et en RAM, et demanderont plus de travail au planificateur pour générer le plan optimal. Augmenter cette valeur n'a donc pas que des avantages : on évitera de dépasser 1000.

Il est possible de configurer ce paramètrage table par table et colonne par colonne :

```
ALTER TABLE nom_table ALTER nom_colonne SET STATISTICS valeur ;
```

Ne pas oublier de relancer un `ANALYZE nom_table ;` juste après.

6.7 LECTURE D'UN PLAN



Un plan d'exécution se lit en partant du nœud se trouvant le plus à droite et en remontant jusqu'au nœud final. Quand le plan contient plusieurs nœuds, le premier nœud exécuté est celui qui se trouve le plus à droite. Celui qui est le plus à gauche (la première ligne) est le dernier nœud exécuté. Tous les nœuds sont exécutés simultanément, et traitent les données dès qu'elles sont transmises par le nœud parent (le ou les nœuds justes 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 (elle dépend essentiellement de la taille de la table lue, mais aussi d'opération de filtrage). `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 :

```
SET enable_nestloop TO off;
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..2.34 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)

RESET enable_nestloop;
```

Ce plan débute en bas 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 (paramètre `seq_page_cost`) et à la manipulation des 4 lignes de la table `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 2,34. La jointure par hachage démarre réellement lorsque la lecture de la table `employes` commence. Cette lecture remontera 14 lignes, sans application de filtre. La totalité de la table est donc remontée et elle est très petite donc tient sur un seul bloc de 8 ko. Le coût d'accès total est donc facilement déduit à partir de cette information. À partir des sélectivités précédentes, l'optimiseur estime que la jointure ramènera 4 lignes au total.

6.7.1 Rappel des options d'EXPLAIN

- `ANALYZE` : exécution (danger !)
- `BUFFERS` : blocs *read*/*hit*/*written*/*dirtied*, *shared*/*local*/*temp*
- `GENERIC_PLAN` : plan générique (requête préparée, v16)
- `SETTINGS` : paramètres configurés pour l'optimisation
- `WAL` : nombre d'enregistrements et nombre d'octets écrits dans les journaux
- `COSTS` : par défaut
- `TIMING` : par défaut
- `VERBOSE` : colonnes considérées
- `SUMMARY` : temps de planification
- `FORMAT` : sortie en text, XML, JSON, YAML

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, ou au contraire de masquer des informations affichées par défaut.

Option ANALYZE

Le but de cette option est d'obtenir les informations sur l'exécution réelle de la requête.

Avec `ANALYZE`, la requête est réellement exécutée ! Attention donc aux `INSERT` / `UPDATE` / `DELETE`. N'oubliez pas non plus qu'un `SELECT` peut appeler des fonctions qui écrivent dans la base. Dans le doute, pensez à englober l'appel dans une transaction que vous annulerez après coup.

Voici un exemple utilisant cette option :

```
BEGIN;
EXPLAIN (ANALYZE) SELECT * FROM employes WHERE matricule < 100 ;
ROLLBACK;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                (actual time=0.004..0.005 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.027 ms
 Execution time: 0.013 ms
```

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 : comparer au nombre de la première parenthèse permet d'avoir une idée de la justesse des statistiques et de l'estimation ;
- `loops` est le nombre d'exécutions de ce nœud, car certains peuvent être répétés de nombreuses fois.

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 en pratique 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 :

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM employes WHERE matricule < 100;
```

```
                      QUERY PLAN
-------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                (actual time=0.002..0.004 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
   Buffers: shared hit=1
 Planning time: 0.024 ms
 Execution time: 0.011 ms
```

La nouvelle ligne est la ligne `Buffers`. `shared hit` indique un accès à une table ou index dans les *shared buffers* de PostgreSQL. Ces autres indications peuvent se rencontrer :

Informations	Type d'objet concerné	Explications
Shared hit	Table ou index permanent	Lecture d'un bloc dans le cache
Shared read	Table ou index permanent	Lecture d'un bloc hors du cache
Shared written	Table ou index permanent	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

Informations	Type d'objet concerné	Explications
Temp read	Tris et hachages	Lecture d'un bloc
Temp written	Tris et hachages	Écriture d'un bloc

`EXPLAIN (BUFFERS)`, sans `ANALYZE`, fonctionne certes à partir de PostgreSQL 13, mais n'affiche alors que les quelques blocs consommés par la planification.

Option GENERIC_PLAN

Cette option (à partir de PostgreSQL 16) sert quand on cherche le plan générique planifié pour une requête préparée (c'est-à-dire dont les paramètres seront fournis séparément).

```
EXPLAIN (GENERIC_PLAN, SUMMARY ON)
SELECT * FROM t1 WHERE c1 < $1 ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using t1_c1_idx on t1  (cost=0.15..14.98 rows=333 width=8)
   Index Cond: (c1 < $1)
 Planning Time: 0.195 ms
```

Option SETTINGS

Cette option permet d'obtenir les valeurs des paramètres spécifiques à l'optimisation de requêtes qui ne sont pas à leur valeur par défaut. Elle est désactivée par défaut.

```
EXPLAIN (SETTINGS) SELECT * FROM employes_big WHERE matricule=33;
```

```
                               QUERY PLAN
------------------------------------------------------------------------
Index Scan using employes_big_pkey on employes_big (cost=0.42..8.44 rows=1 width=41)
  Index Cond: (matricule = 33)
```

```
SET enable_indexscan TO off;
EXPLAIN (SETTINGS) SELECT * FROM employes_big WHERE matricule=33;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big  (cost=4.43..8.44 rows=1 width=41)
   Recheck Cond: (matricule = 33)
   ->  Bitmap Index Scan on employes_big_pkey  (cost=0.00..4.43 rows=1 width=0)
         Index Cond: (matricule = 33)
 Settings: enable_indexscan = 'off'
```

Option WAL

Cette option permet d'obtenir le nombre d'enregistrements et le nombre d'octets écrits dans les journaux de transactions. Elle apparaît avec PostgreSQL 13 et est désactivée par défaut.

```
CREATE TABLE t1 (id integer);
EXPLAIN (ANALYZE, WAL) INSERT INTO t1 SELECT generate_series(1, 1000) ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Insert on t1  (cost=0.00..15.02 rows=1000 width=12)
           (actual time=1.457..1.458 rows=0 loops=1)
   WAL: records=2009 bytes=123824
   ->  Subquery Scan on "*SELECT*"
       (cost=0.00..15.02 rows=1000 width=12)
       (actual time=0.003..0.146 rows=1000 loops=1)
         ->  ProjectSet  (cost=0.00..5.02 rows=1000 width=4)
                (actual time=0.002..0.068 rows=1000 loops=1)
               ->  Result  (cost=0.00..0.01 rows=1 width=0)
                  (actual time=0.001..0.001 rows=1 loops=1)
 Planning Time: 0.033 ms
 Execution Time: 1.479 ms
```

Option COSTS

Activée par défaut, l'option `COSTS` indique les estimations du planificateur. La désactiver permet de gagner un peu en lisibilité.

```
EXPLAIN (COSTS OFF) SELECT * FROM employes WHERE matricule < 100;
```

```
         QUERY PLAN
----------------------------
 Seq Scan on employes
   Filter: (matricule < 100)
```

```
EXPLAIN (COSTS ON) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
   Filter: (matricule < 100)
```

Option TIMING

Cette option n'est utilisable qu'avec l'option `ANALYZE` et est activée par défaut. Elle ajoute les informations sur les durées en milliseconde. 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 :

```
EXPLAIN (ANALYZE, TIMING ON) SELECT * FROM employes WHERE matricule < 100;
```

```
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
              (actual time=0.003..0.004 rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.022 ms
 Execution time: 0.010 ms
```

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM employes WHERE matricule < 100;
```

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

```
 Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
                                (actual rows=3 loops=1)
   Filter: (matricule < 100)
   Rows Removed by Filter: 11
 Planning time: 0.025 ms
 Execution time: 0.010 ms
```

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 nom du schéma, le nom de la fonction qualifié du nom du schéma, le nom du déclencheur (trigger), etc. Elle est désactivée par défaut.

```
EXPLAIN (VERBOSE) SELECT * FROM employes WHERE matricule < 100;

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on public.employes  (cost=0.00..1.18 rows=3 width=43)
   Output: matricule, nom, prenom, fonction, manager, date_embauche,
           num_service
   Filter: (employes.matricule < 100)
```

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, et permet d'afficher ou non le résumé final indiquant la durée de la planification et de l'exécution. Par défaut, un `EXPLAIN` simple n'affiche pas le résumé, mais un `EXPLAIN ANALYZE` le fait.

```
EXPLAIN SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)

EXPLAIN (SUMMARY ON) SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
 Planning time: 0.014 ms

EXPLAIN (ANALYZE) SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
              (actual time=0.002..0.003 rows=14 loops=1)
 Planning time: 0.013 ms
 Execution time: 0.009 ms

EXPLAIN (ANALYZE, SUMMARY OFF) SELECT * FROM employes;
```

```
                      QUERY PLAN
------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
               (actual time=0.002..0.003 rows=14 loops=1)
```

Option FORMAT

Par défaut, la sortie est sous forme d'un texte destiné à être lu par un humain, 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 :

```
EXPLAIN (FORMAT XML) SELECT * FROM employes WHERE matricule < 100;
```

```
                         QUERY PLAN
------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">+
   <Query>                                               +
     <Plan>                                              +
       <Node-Type>Seq Scan</Node-Type>                   +
       <Parallel-Aware>false</Parallel-Aware>            +
       <Relation-Name>employes</Relation-Name>           +
       <Alias>employes</Alias>                           +
       <Startup-Cost>0.00</Startup-Cost>                 +
       <Total-Cost>1.18</Total-Cost>                     +
       <Plan-Rows>3</Plan-Rows>                          +
       <Plan-Width>43</Plan-Width>                       +
       <Filter>(matricule &lt; 100)</Filter>             +
     </Plan>                                             +
   </Query>                                              +
 </explain>
(1 row)
```

Les signes `+` en fin de ligne indiquent un retour à la ligne lors de l'utilisation de l'outil `psql`. Il est possible de ne pas les afficher en configurant l'option `format` de `psql` à `unaligned`. Cela se fait ainsi :

```
\pset format unaligned
```

Ces formats semi-structurés sont utilisés principalement par des outils, car le contenu est plus facile à analyser, voire un peu plus complet.

6.7.2 Statistiques, cardinalités & 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. Cherchons comment calculer la cardinalité d'un filtre simple sur une table `employes` de 14 lignes, par exemple `WHERE num_service = 1`.

Ici, la valeur recherchée se trouve directement dans le tableau des valeurs les plus fréquentes (dans les champs `most_common_vals` et `most_common_freqs`) la cardinalité peut être calculée directement.

```
SELECT * FROM pg_stats
WHERE tablename = 'employes'
  AND attname   = 'num_service' \gx

-[ RECORD 1 ]----------+-----------------------------------------------
schemaname             | public
tablename              | employes
attname                | num_service
inherited              | f
null_frac              | 0
```

```
avg_width                 | 4
n_distinct                | -0.2857143
most_common_vals          | {4,3,2,1}
most_common_freqs         | {0.35714287,0.2857143,0.21428572,0.14285715}
histogram_bounds          | ¤
correlation               | 0.10769231
…
```

La requête suivante permet de récupérer la fréquence d'apparition de la valeur recherchée :

```
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 = 'employes'
           AND attname = 'num_service') get_mcv
 WHERE value = 1;
```

```
 tablename |   attname   | value |   freq
-----------+-------------+-------+----------
 employes  | num_service |     1 | 0.142857
```

Si l'on n'avait pas eu affaire à une des valeurs les plus fréquentes, il aurait fallu passer par l'histogramme des valeurs (`histogram_bounds`, ici vide car il y a trop peu de valeurs), pour calculer d'abord la sélectivité du filtre pour en déduire ensuite la cardinalité.

Une fois cette fréquence obtenue, l'optimiseur calcule la cardinalité du prédicat `WHERE num_service = 1` en la multipliant avec le nombre total de lignes de la table :

```
SELECT 0.142857 * reltuples AS cardinalite_predicat
  FROM pg_class
 WHERE relname = 'employes';
```

```
 cardinalite_predicat
----------------------
             1.999998
```

Le calcul est cohérent avec le plan d'exécution de la requête impliquant la lecture de `employes` sur laquelle on applique le prédicat évoqué plus haut :

```
EXPLAIN SELECT * FROM employes WHERE num_service = 1;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=2 width=43)
   Filter: (num_service = 1)
```

Calcul de coût

Notre table `employes` peuplée de 14 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 `employes` est calculé à partir de deux composantes. Tout d'abord, le nombre de pages (ou blocs) de la table permet de déduire le nombre de blocs à accéder pour lire la table intégralement. Le paramètre `seq_page_cost` (coût d'accès à un bloc dans un parcours complet) sera appliqué ensuite pour obtenir le coût de l'opération :

```
SELECT relname, relpages * current_setting('seq_page_cost')::float AS cout_acces
  FROM pg_class
 WHERE relname = 'employes';

 relname  | cout_acces
----------+------------
 employes |          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 multiplie donc par `cpu_tuple_cost` (0,01 par défaut) 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 = 'employes';

 relname  | cout
----------+------
 employes | 1.14
```

Le calcul est bon :

```
EXPLAIN SELECT * FROM employes;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
```

Avec un filtre dans la requête, les traitements seront plus lourds. Par exemple, en ajoutant le prédicat `WHERE date_embauche='2006-01-01'`, il faut non seulement extraire les lignes les unes après les autres, mais 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 = 'employes';

 relname  | cost
----------+-------
 employes | 1.175
```

Ce nombre se retrouve dans le plan, à l'arrondi près :

```
EXPLAIN SELECT * FROM employes WHERE date_embauche='2006-01-01';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=2 width=43)
   Filter: (date_embauche = '2006-01-01'::date)
```

Pour aller plus loin dans le calcul de sélectivité, de cardinalité et de coût, la documentation de PostgreSQL contient un exemple complet de calcul de sélectivité et indique les références des fichiers sources dans lesquels fouiller pour en savoir plus[4].

[4] https://docs.postgresql.fr/current/planner-stats-details.html

6.8 NŒUDS D'EXÉCUTION LES PLUS COURANTS

- Un plan est composé de nœuds
- qui produisent des données
- ou en consomment et en retournent
- Chaque nœud consomme les données produites par le(s) nœud(s) parent(s)
- Le nœud final retourne les données à l'utilisateur

Les plans sont extrêmement sensibles aux données elles-mêmes bien sûr, aux paramètres, aux tailles réelles des objets, à la version de PostgreSQL, à l'ordre des données dans la table, voire au moment du passage d'un `VACUUM`. Il n'est donc pas étonnant de trouver parfois des plans différents de ceux reproduits ici pour une même requête.

6.8.1 Nœuds de type parcours

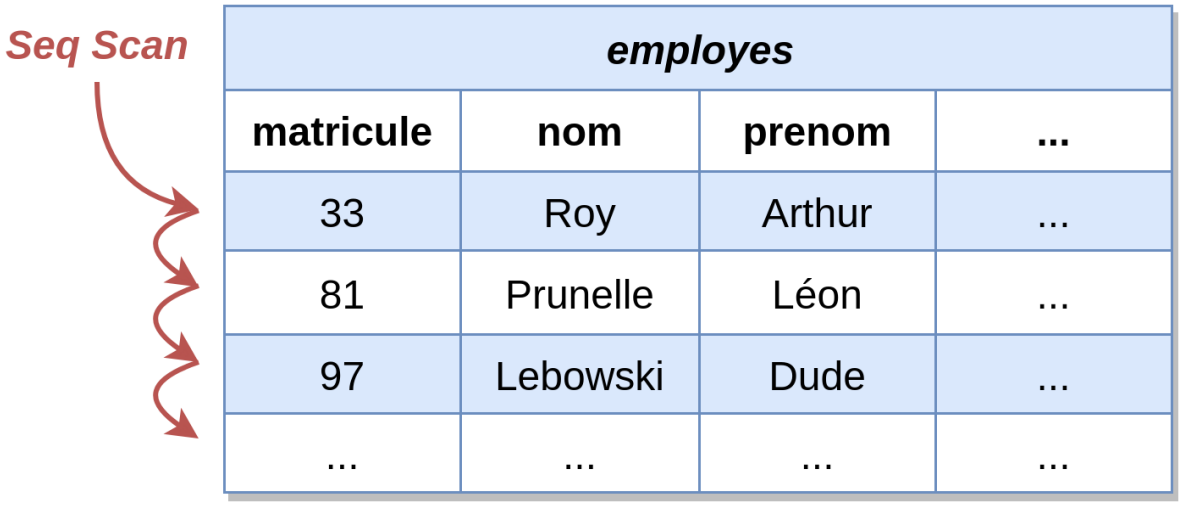
- Parcours de table
- Parcours d'index
- Autres parcours

Par parcours, on entend le renvoi d'un ensemble de lignes provenant soit d'un fichier, soit d'un traitement. Le fichier peut correspondre à une table ou à une vue matérialisée, et on parle dans ces deux cas d'un parcours de table. Le fichier peut aussi correspondre à un index, auquel cas on parle de parcours d'index. Un parcours peut être un traitement dans différents cas, principalement celui d'une procédure stockée.

6.8.2 Parcours de table

- *Seq Scan*
- *Parallel Seq Scan*

6.8.3 Parcours de table : Seq Scan



Seq Scan :

Les parcours de tables sont les principales opérations qui lisent les données des tables (normales, temporaires ou non journalisées) et des vues matérialisées. Elles ne prennent donc pas d'autre nœud en entrée et fournissent un ensemble de données en sortie. Cet ensemble peut être trié ou non, filtré ou non.

L'opération *Seq Scan* correspond à une lecture séquentielle d'une table, aussi appelée *Full table scan* sur d'autres SGBD. Il consiste à lire l'intégralité de la table, du premier bloc au dernier bloc. Une clause de filtrage peut être appliquée.

Ce nœud apparaît lorsque la requête nécessite de lire l'intégralité ou la majorité de la table :

```
EXPLAIN SELECT * FROM employes;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
```

Ce nœud peut également filtrer directement les données, la présence de la clause *Filter* montre le filtre appliqué par le nœud à la lecture des données :

```
EXPLAIN SELECT * FROM employes WHERE matricule=135;
```

```
                       QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.18 rows=1 width=43)
   Filter: (matricule = 135)
```

6.8.4 Parcours de table : paramètres

- *Seq Scan*
  - `seq_page_cost` (défaut : 1)
  - `cpu_tuple_cost` & `cpu_operator_cost`
  - `enable_seqscan`
- *Parallel Seq Scan*
  - `parallel_tuple_cost`, `min_parallel_table_scan_size`
  - et les autres paramètres de la parallélisation

Seq Scan :

Le coût d'un *Seq Scan* sera fonction du nombre de blocs à parcourir, et donc du paramètre `seq_page_cost`, ainsi que du nombre de lignes à décoder et, optionnellement, à filtrer. La valeur de `seq_page_cost`, par défaut à 1, est rarement modifiée ; elle est surtout importante comparée à `random_page_cost`.

Parallel Seq Scan :

Il est possible d'avoir un parcours parallélisé d'une table (*Parallel Seq Scan*). La parallélisation doit être activée comme décrit plus haut, notamment :

- les paramètres `max_parallel_workers_per_gather` et `max_parallel_workers` doivent être tous deux supérieurs à 0, ce qui est le cas par défaut ;
- la table à traiter doit avoir une taille minimale (`min_parallel_table_scan_size` est à 8 Mo) ; et plus elle sera grosse plus il y aura de *workers* ;

Pour que ce type de parcours soit valable, il faut que l'optimiseur soit persuadé que le problème sera le temps CPU et non la bande passante disque. Il y a cependant un coût pour chaque ligne (paramètre `parallel_tuple_cost`). Autrement dit, dans la majorité des cas, il faut un filtre sur une table importante pour que la parallélisation se déclenche.

Dans les exemples suivants, la parallélisation est activée :

```
SET max_parallel_workers_per_gather TO 5 ;   /*  défaut : 2 */

-- Plan d'exécution parallélisé
EXPLAIN SELECT * FROM employes_big WHERE num_service <> 4;

                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
         Filter: (num_service <> 4)
```

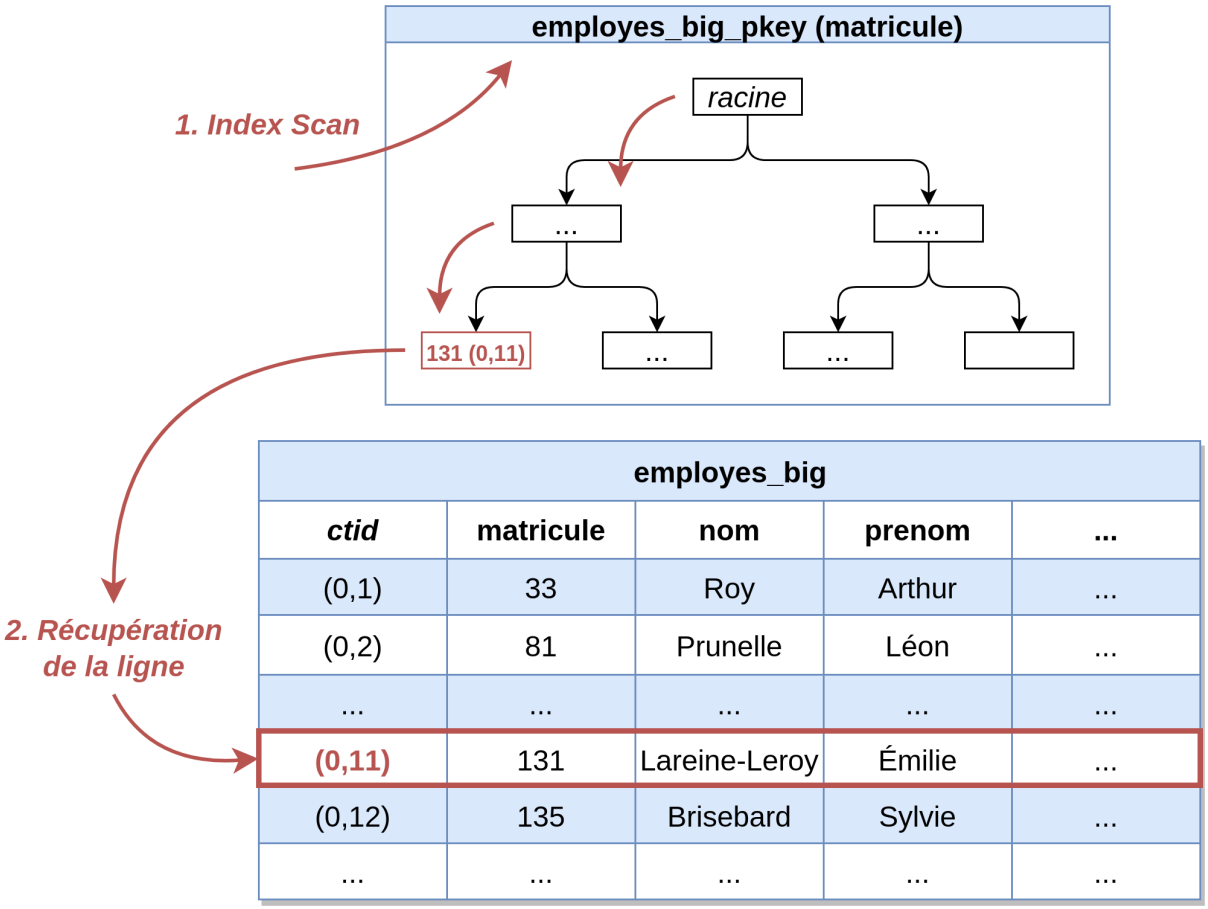
Ici, deux processus supplémentaires seront exécutés pour réaliser la requête. Dans le cas de ce type de parcours, chaque processus prend un bloc et traite toutes les lignes de ce bloc. Quand un processus a terminé de traiter son bloc, il regarde quel est le prochain bloc à traiter et le traite. (À partir de la version 14, il prend même un groupe de blocs pour profiter de la fonctionnalité de *read ahead* du noyau.)

6.8.5 Parcours d'index

- *Index Scan*
- *Index Only Scan*
- *Bitmap Index Scan*
- et leurs versions parallélisées (B-Tree)

PostgreSQL dispose de trois moyens d'accéder aux données à travers les index. Le plus connu est l'*Index Scan*, qui possède plusieurs variantes.

6.8.6 Index Scan



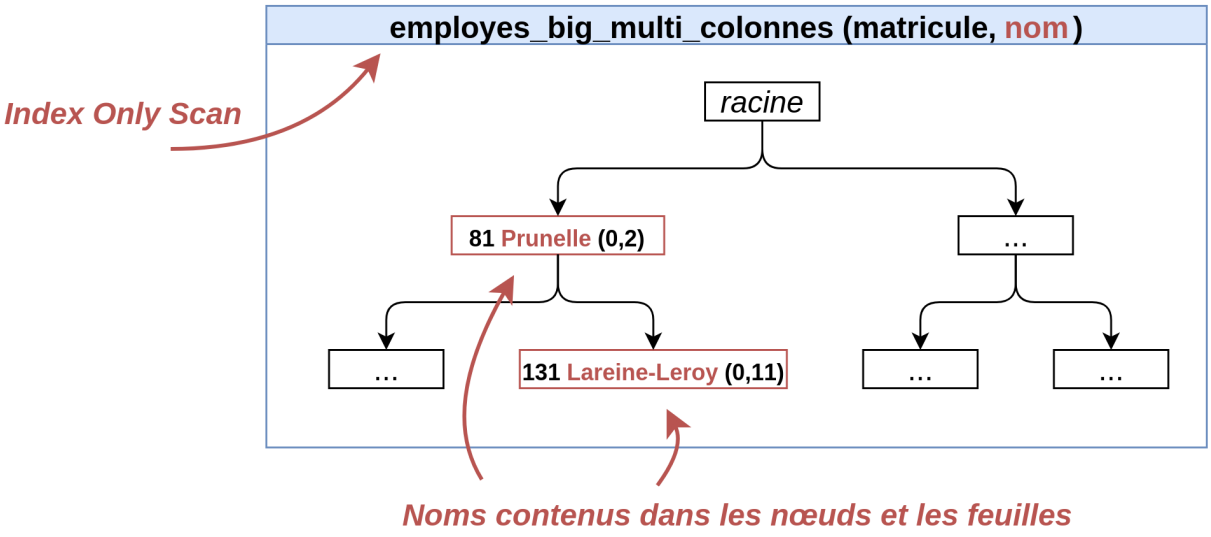
Le nœud *Index Scan* consiste à parcourir les blocs d'index jusqu'à trouver les pointeurs vers les blocs contenant les données. À chaque pointeur trouvé, PostgreSQL lit le bloc de la table pointée pour retrouver l'enregistrement et s'assurer notamment de sa visibilité pour la transaction en cours. De ce fait, il y a beaucoup d'accès non séquentiels pour lire l'index et la table.

```
EXPLAIN SELECT * FROM employes_big WHERE matricule = 132;
```

```
                        QUERY PLAN
-----------------------------------------------------
 Index Scan using employes_big_pkey on employes_big
   (cost=0.42..8.44 rows=1 width=41)
   Index Cond: (matricule = 132)
```

Ce type de nœud ne permet pas d'extraire directement les données à retourner depuis l'index, sans passer par la lecture des blocs correspondants de la table.

6.8.7 Index Only Scan : principe



6.8.8 Index Only Scan : utilité & limites

- Très performant
- Besoin d'un `VACUUM` récent
  - sinon trop de *Heap Fetches*
- Ne fonctionne pas pour les index fonctionnels
  - ajouter une colonne générée ?

Utilité :

Le nœud *Index Only Scan* est une version plus performante de l'*Index Scan*. Il est choisi si **toutes** les colonnes de la table concernées par la requête font partie de l'index :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT matricule
FROM   employes_big
WHERE  matricule < 100000 ;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Only Scan using employes_big_pkey on employes_big (actual time=0.019..8.195
↪  rows=99014 loops=1)
   Index Cond: (matricule < 100000)
   Heap Fetches: 0
   Buffers: shared hit=274
```

```
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.083 ms
 Execution Time: 11.123 ms
```

Il n'y a donc plus besoin d'accéder à la table et l'on économise de nombreux accès disque. C'est d'autant plus appréciable que les lignes sont nombreuses.

Par contre, si l'on ajoute le champ `nom` dans la requête, l'optimiseur se rabat sur un *Index Scan*. En effet, `nom` ne peut être lu que dans la table puisqu'il ne fait pas partie de l'index. Dans notre exemple avec de petites tables, le temps reste correct mais il est moins bon :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT matricule, nom
FROM employes_big
WHERE matricule < 100000 ;

                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using employes_big_pkey on employes_big (actual time=0.009..14.562
 ↪  rows=99014 loops=1)
   Index Cond: (matricule < 100000)
   Buffers: shared hit=1199
 Planning:
   Buffers: shared hit=60
 Planning Time: 0.181 ms
 Execution Time: 18.170 ms
```

Noter le nombre de blocs lus beaucoup plus important.

Index couvrants :

Il existe des index dits « couvrants », destinés à favoriser des *Index Only Scans*. Ils peuvent contenir des données en plus des champs indexés, même avec un index unique comme ici.

```
CREATE UNIQUE INDEX ON employes_big (matricule) INCLUDE (nom) ;
```

Cet index contient aussi `nom` et permet de revenir à de très bonnes performances :

```
EXPLAIN (COSTS OFF, ANALYZE,BUFFERS)
SELECT matricule, nom
FROM   employes_big
WHERE  matricule < 100000 ;

                          QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using employes_big_matricule_nom_idx on employes_big (actual
 ↪  time=0.010..5.769 rows=99014 loops=1)
   Index Cond: (matricule < 100000)
   Heap Fetches: 0
   Buffers: shared hit=383
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.064 ms
 Execution Time: 7.747 ms
```

L'expérience montre qu'un index classique multicolonne (sur `(matricule,nom)`) sera aussi efficace, voire plus, car il peut être plus petit (la clause `INCLUDE` inhibe la déduplication). La clause `INCLUDE` reste utile pour faire d'une clé primaire, unique ou étrangère, un index couvrant et économiser un peu de place disque.

Conditions pour obtenir un Index Only Scan :

- Toutes les colonnes de la table utilisées par la requête doivent figurer dans l'index, aussi bien celles retournées que celles servant aux calculs ou au filtrage. En pratique, cela réserve l'*Index Only Scan* au cas où peu de colonnes de la table sont concernées par la requête.
- Il est courant d'ajouter des index dédiés aux requêtes critiques n'utilisant que quelques colonnes. Mais plus le nombre de colonnes et la taille de l'index augmentent, moins PostgreSQL sera tenté par l'*Index Only Scan*.
- Pour que l'*Index Only Scan* soit réellement efficace, il faut que la table soit fréquemment traitée par des opérations `VACUUM`. En effet, les informations de visibilité des lignes ne sont pas stockées dans l'index. Pour savoir si la ligne trouvée dans l'index est visible ou pas par la session, il faut soit aller vérifier dans la table (et on en revient à un *Index Scan*), soit aller voir dans la *visibility map* de la table que le bloc ne contient pas de lignes potentiellement mortes, ce qui est beaucoup plus rapide. Le choix se fait à l'éxécution pour chaque ligne. Dans l'idéal, le plan indique qu'il n'y a jamais eu à aller dans la table (*heap*) ainsi :

```
Heap Fetches: 0
```

S'il y a un trop grand nombre de *Heap Fetches*, l'*Index Only Scan* perd tout son intérêt. Il peut suffire de rendre l'`autovacuum` beaucoup plus agressif sur la table.

Limite pour les index fonctionnels :

Le planificateur a du mal à définir un *Index Only Scan* pour un index fonctionnel. Il échoue par exemple ici :

```
CREATE INDEX employes_big_upper_idx ON employes_big (upper (nom)) ;
-- Créer les statistiques pour l'index fonctionnel
ANALYZE employes_big ;

EXPLAIN (COSTS OFF)
SELECT upper(nom) FROM employes_big WHERE upper(nom) = 'CRUCHOT' ;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Index Scan using employes_big_upper_idx on employes_big
   Index Cond: (upper((nom)::text) = 'CRUCHOT'::text)
```

L'index fonctionnel est bien utilisé mais ce n'est pas un *Index Only Scan*.

Dans certains cas critiques, il peut être intéressant de créer une colonne générée reprenant la fonction et de créer un index dessus, qui permettra un *Index Only Scan* :

```
-- Attention : réécriture de la table
ALTER TABLE employes_big
ADD COLUMN nom_maj text GENERATED ALWAYS AS ( upper(nom) ) STORED ;
```

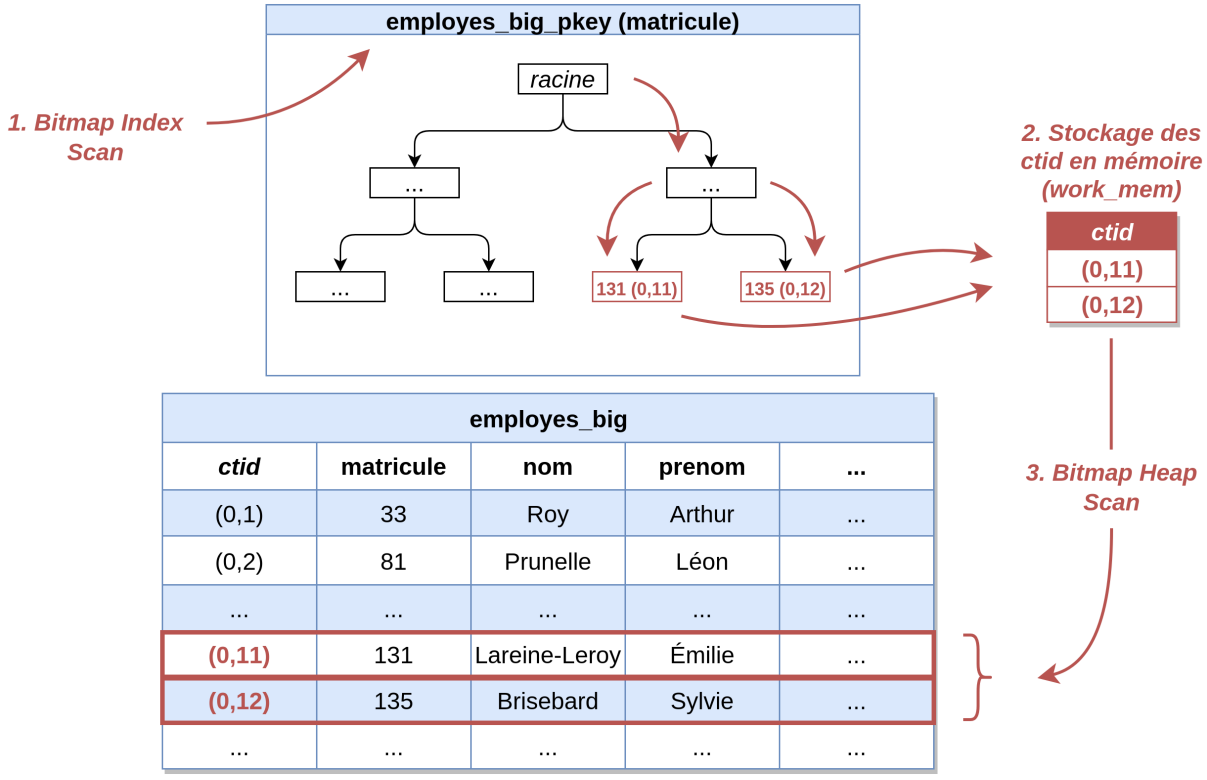
```
CREATE INDEX ON employes_big (nom_maj) ;
-- Créer les statistiques pour la nouvelle colonne
-- et éviter les Heap Fetches
VACUUM ANALYZE employes_big ;

EXPLAIN (COSTS OFF) SELECT nom_maj FROM employes_big WHERE nom_maj  = 'CRUCHOT' ;

                        QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using employes_big_nom_maj_idx on employes_big
   Index Cond: (nom_maj = 'CRUCHOT'::text)
```

La nouvelle colonne générée est bien sûr maintenue automatiquement si `nom` change et n'est pas modifiable directement. Mais sa création entraîne la réécriture de la table, qui d'ailleurs grossit un peu, et il faut adapter le code.

6.8.9 Bitmap Scan



Ce dernier parcours est particulièrement efficace pour des opérations de type *Range Scan*, c'est-à-dire où PostgreSQL doit retourner une plage de valeurs, ou pour combiner le résultat de la lecture de plusieurs index.

Contrairement à d'autres SGBD, un index *bitmap* de PostgreSQL n'a aucune existence sur disque : il est créé en mémoire lorsque son utilisation a un intérêt. Le but est de diminuer les déplacements de la tête de lecture en découplant le parcours de l'index du parcours de la table :

- lecture en un bloc de l'index ;
- lecture en un bloc de la partie intéressante de la table (dans l'ordre physique de la table, pas dans l'ordre logique de l'index).

```
SET enable_indexscan TO off ;

EXPLAIN
  SELECT * FROM employes_big WHERE matricule BETWEEN 200000 AND 300000;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big
   (cost=2108.46..8259.35 rows=99126 width=41)
   Recheck Cond: ((matricule >= 200000) AND (matricule <= 300000))
   ->  Bitmap Index Scan on employes_big_pkey*
       (cost=0.00..2083.68 rows=99126 width=0)
         Index Cond: ((matricule >= 200000) AND (matricule <= 300000))
```

```
RESET enable_indexscan;
```

Exemple de combinaison du résultat de la lecture de plusieurs index :

```
EXPLAIN
  SELECT * FROM employes_big
  WHERE matricule BETWEEN   1000 AND 100000
     OR matricule BETWEEN 200000 AND 300000;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on employes_big
   (cost=4265.09..12902.67 rows=178904 width=41)
   Recheck Cond: (((matricule >= 1000) AND (matricule <= 100000))
               OR ((matricule >= 200000) AND (matricule <= 300000)))
   ->  BitmapOr  (cost=4265.09..4265.09 rows=198679 width=0)
         ->  Bitmap Index Scan on employes_big_pkey
               (cost=0.00..2091.95 rows=99553 width=0)
               Index Cond: ((matricule >= 1000) AND (matricule <= 100000))
         ->  Bitmap Index Scan on employes_big_pkey
               (cost=0.00..2083.68 rows=99126 width=0)
               Index Cond: ((matricule >= 200000) AND (matricule <= 300000))
```

6.8.10 Parcours d'index : paramètres importants

- `random_page_cost` (4 ou moins ?)
- `cpu_index_tuple_cost`
- `effective_cache_size` (⅔ de la RAM ?)
- Selon le disque :
  - `effective_io_concurrency`
  - `maintenance_io_concurrency`
- `min_parallel_index_scan_size`
- `enable_indexscan`, `enable_indexonlyscan`, `enable_bitmapscan`

Index Scan :

L'*Index Scan* n'a d'intérêt que s'il y a très peu de lignes à récupérer, surtout si les disques sont mécaniques. Il faut donc que le filtre soit très sélectif.

Le rapport entre les paramètres `seq_page_cost` et `random_page_cost` est d'importance majeure dans le choix face à un *Seq Scan*. Plus il est proche de 1, plus les parcours d'index seront favorisés par rapport aux parcours de table.

Index Only Scan :

Il n'y a pas de paramètre dédié à ce parcours. S'il est possible, l'optimiseur le préfère à un *Index Scan*.

Bitmap Index Scan :

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[5], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. À l'inverse, il faut tenir compte du nombre de requêtes simultanées qui utiliseront ce nœud. Le défaut est seulement de 1, et la valeur maximale est 1000. Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[6].

Toujours à partir de la version 13, un nouveau paramètre apparaît : `maintenance_io_concurrency`. Il a le même but que `effective_io_concurrency`, mais pour les opérations de maintenance, non les requêtes. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Sa valeur par défaut est de 10, et il faut penser à le monter aussi si on adapte `effective_io_concurrency`.

[5] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[6] https://docs.postgresql.fr/13/release.html

Enfin, le paramètre `effective_cache_size` indique à PostgreSQL une estimation de la taille du cache disque du système (total du *shared buffers* et du cache du système). Une bonne pratique est de positionner ce paramètre à ⅔ 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 *Bitmap Index Scan*.

Parallélisation

Il est possible de paralléliser les parcours d'index. Cela donne donc les nœuds *Parallel Index Scan*, *Parallel Index Only Scan* et *Parallel Bitmap Heap Scan*. Cette infrastructure est actuellement uniquement utilisée pour les index B-Tree. Par contre, pour le *Bitmap Scan*, seul le parcours de la table est parallélisé. Un parcours parallélisé d'un index n'est considéré qu'à partir du moment où l'index a une taille supérieure à la valeur du paramètre `min_parallel_index_scan_size` (512 ko par défaut).

6.8.11 Autres parcours

- *Function Scan*
- *Values Scan*
- …et d'autres

On retrouve le nœud *Function Scan* lorsqu'une requête utilise directement le résultat d'une fonction, comme par exemple, dans des fonctions d'informations système de PostgreSQL :

```
EXPLAIN SELECT * from pg_get_keywords();
```

```
                            QUERY PLAN
------------------------------------------------------------------------
 Function Scan on pg_get_keywords  (cost=0.03..4.03 rows=400 width=65)
```

Il existe d'autres types de parcours, rarement rencontrés. Ils sont néanmoins détaillés en annexe[7].

[7] https://dali.bo/j6_html

6.8.12 Nœuds de jointure

- PostgreSQL implémente les 3 algorithmes de jointures habituels
  - *Nested Loop* : boucle imbriquée
  - *Hash Join* : hachage de la table interne
  - *Merge Join* : tri-fusion
- Parallélisation
- Pour `EXISTS`, `IN` et certaines jointures externes
  - *Hash Semi Join* & *Hash Anti Join*
- Paramètres :
  - `work_mem` (et `hash_mem_multiplier`)
  - `seq_page_cost` & `random_page_cost`.
  - `enable_nestloop`, `enable_hashjoin`, `enable_mergejoin`

Le choix du type de jointure dépend non seulement des données mises en œuvre, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment des paramètres `work_mem`, `hash_mem_multiplier`, `seq_page_cost` et `random_page_cost`.

Nested Loop :

La *Nested Loop* se retrouve principalement dans les jointures de petits ensembles de données. Dans l'exemple suivant, le critère sur `services` ramène très peu de lignes, il ne coûte pas grand-chose d'aller piocher à chaque fois dans l'index de `employes_big`.

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

```
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.42..10053.94 rows=124754 width=46)
   ->  Seq Scan on services ser  (cost=0.00..1.05 rows=1 width=21)
         Filter: ((localisation)::text = 'Nantes'::text)
   ->  Index Scan using employes_big_num_service_idx on employes_big emp
         (cost=0.42..7557.81 rows=249508 width=33)
         Index Cond: (num_service = ser.num_service)
```

Hash Join :

Le *Hash Join* se retrouve lorsque l'ensemble de la table interne est petit. L'optimiseur construit alors une table de hachage avec les valeurs de la ou les colonne(s) de jointure de la table interne. Il réalise ensuite un parcours de la table externe, et, pour chaque ligne de celle-ci, recherche des lignes cor-

respondantes dans la table de hachage, toujours en utilisant la ou les colonne(s) de jointure comme clé

```
EXPLAIN SELECT matricule, nom, prenom, nom_service, fonction, localisation
FROM   employes_big emp
JOIN   services ser ON (emp.num_service = ser.num_service);

                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=0.19..8154.54 rows=499015 width=45)
   Hash Cond: (emp.num_service = ser.num_service)
   ->  Seq Scan on employes_big emp  (cost=0.00..5456.55 rows=499015 width=32)
   ->  Hash  (cost=0.14..0.14 rows=4 width=21)
         ->  Seq Scan on services ser  (cost=0.00..0.14 rows=4 width=21)
```

Cette opération réclame de la mémoire de tri, visible avec `EXPLAIN (ANALYZE)` (dans le pire des cas, ce sera un fichier temporaire).

Merge Join :

La jointure par tri-fusion, ou *Merge Join*, prend deux ensembles de données triés en entrée et restitue l'ensemble de données après jointure. Cette jointure est assez lourde à initialiser si PostgreSQL ne peut pas utiliser d'index, mais elle a l'avantage de retourner les données triées directement :

```
EXPLAIN
  SELECT matricule, nom, prenom, nom_service, fonction, localisation
  FROM   employes_big emp
  JOIN   services_big ser ON (emp.num_service = ser.num_service)
  ORDER BY ser.num_service;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=0.82..20094.77 rows=499015 width=49)
   Merge Cond: (emp.num_service = ser.num_service)
   ->  Index Scan using employes_big_num_service_idx on employes_big emp
       (cost=0.42..13856.65 rows=499015 width=33)
   ->  Index Scan using services_big_pkey on services_big ser
       (cost=0.29..1337.35 rows=40004 width=20)
```

Il s'agit d'un algorithme de jointure particulièrement efficace pour traiter les volumes de données importants, surtout si les données sont pré-triées grâce à l'existence d'un index.

Hash Anti/Semi Join :

Les clauses `EXISTS` et `NOT EXISTS` mettent également en œuvre des algorithmes dérivés de semi et anti-jointures. En voici un exemple avec la clause `EXISTS` :

```
EXPLAIN
  SELECT *
  FROM services s
  WHERE EXISTS (SELECT 1
                FROM employes_big e
                WHERE e.date_embauche>s.date_creation
                  AND s.num_service = e.num_service) ;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Hash Semi Join  (cost=17841.84..19794.91 rows=1 width=25)
   Hash Cond: (s.num_service = e.num_service)
   Join Filter: (e.date_embauche > s.date_creation)
   ->  Seq Scan on services s  (cost=0.00..1.04 rows=4 width=25)
   ->  Hash  (cost=9654.15..9654.15 rows=499015 width=8)
         ->  Seq Scan on employes_big e
             (cost=0.00..9654.15 rows=499015 width=8)
```

Un plan sensiblement identique s'obtient avec `NOT EXISTS`. Le nœud *Hash Semi Join* est remplacé par *Hash Anti Join* :

```
EXPLAIN
  SELECT *
  FROM services s
  WHERE NOT EXISTS (SELECT 1
                    FROM employes_big e
                    WHERE e.date_embauche>s.date_creation
                      AND s.num_service = e.num_service);

                          QUERY PLAN
---------------------------------------------------------------
 Hash Anti Join  (cost=17841.84..19794.93 rows=3 width=25)
   Hash Cond: (s.num_service = e.num_service)
   Join Filter: (e.date_embauche > s.date_creation)
   ->  Seq Scan on services s  (cost=0.00..1.04 rows=4 width=25)
   ->  Hash  (cost=9654.15..9654.15 rows=499015 width=8)
         ->  Seq Scan on employes_big e
             (cost=0.00..9654.15 rows=499015 width=8)
```

Hash Join parallélisé :

Ces nœuds sont parallélisables. Pour les *Parallel Hash Join*, la table hachée est même commune pour les différents *workers* [8], et le calcul de celle-ci est réparti sur ceux-ci. Par contraste, pour les nœuds *Merge Join*, *Nested Loop* et *Hash Join* (non complètement parallélisé), seul le parcours de la table externe peut être parallélisé, tandis que la table interne est parcourue (voire hachée) entièrement par chaque worker.

Exemple (testé en version 16.1) :

```
CREATE TABLE foo(id serial, a int);
CREATE TABLE bar(id serial, foo_a int, b int);
INSERT INTO foo(a) SELECT i*2 FROM generate_series(1,1000000) i;
INSERT INTO bar(foo_a, b) SELECT i*2, i%7 FROM generate_series(1,100) i;
VACUUM ANALYZE foo, bar;

EXPLAIN (ANALYZE, VERBOSE, COSTS OFF)
SELECT foo.a, bar.b FROM foo JOIN bar ON (foo.a = bar.foo_a) WHERE a % 3 = 0;

                                QUERY PLAN
---------------------------------------------------------------------------------
 Gather (actual time=0.192..21.305 rows=33 loops=1)
   Output: foo.a, bar.b
```

[8] https://write-skew.blogspot.com/2018/01/parallel-hash-for-postgresql.html

```
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join (actual time=10.757..16.903 rows=11 loops=3)
         Output: foo.a, bar.b
         Hash Cond: (foo.a = bar.foo_a)
         Worker 0:  actual time=16.118..16.120 rows=0 loops=1
         Worker 1:  actual time=16.132..16.134 rows=0 loops=1
         ->  Parallel Seq Scan on public.foo (actual time=0.009..12.961 rows=111111
↪  loops=3)
               Output: foo.id, foo.a
               Filter: ((foo.a % 3) = 0)
               Rows Removed by Filter: 222222
               Worker 0:  actual time=0.011..12.373 rows=102953 loops=1
               Worker 1:  actual time=0.011..12.440 rows=102152 loops=1
         ->  Hash (actual time=0.022..0.023 rows=100 loops=3)
               Output: bar.b, bar.foo_a
               Buckets: 1024  Batches: 1  Memory Usage: 12kB
               Worker 0:  actual time=0.027..0.027 rows=100 loops=1
               Worker 1:  actual time=0.026..0.026 rows=100 loops=1
               ->  Seq Scan on public.bar (actual time=0.008..0.013 rows=100 loops=3)
                     Output: bar.b, bar.foo_a
                     Worker 0:  actual time=0.012..0.017 rows=100 loops=1
                     Worker 1:  actual time=0.011..0.016 rows=100 loops=1
 Planning Time: 0.116 ms
 Execution Time: 21.321 ms
```

Dans ce plan, la table externe `foo` est lue de manière parallélisée, les trois processus se partageant son contenu. Chacun a sa version de la toute petite table interne `bar`, qui est hachée trois fois (les deux *workers* et le processus principal lisent les 100 lignes).

Si `bar` est beaucoup plus grosse que `foo`, le plan bascule sur un *Parallel Hash Join* dont `bar` est cette fois la table externe :

```
INSERT INTO bar(foo_a, b) SELECT i*2, i%7 FROM generate_series(1,300000) i;
VACUUM ANALYZE bar;

EXPLAIN (ANALYZE, VERBOSE, COSTS OFF)
SELECT foo.a, bar.b FROM foo JOIN bar ON (foo.a = bar.foo_a) WHERE a % 3 = 0;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather (actual time=69.490..95.741 rows=100033 loops=1)
   Output: foo.a, bar.b
   Workers Planned: 1
   Workers Launched: 1
   ->  Parallel Hash Join (actual time=66.408..84.866 rows=50016 loops=2)
         Output: foo.a, bar.b
         Hash Cond: (bar.foo_a = foo.a)
         Worker 0:  actual time=63.450..83.008 rows=52081 loops=1
         ->  Parallel Seq Scan on public.bar (actual time=0.002..5.332 rows=150050
↪  loops=2)
               Output: bar.id, bar.foo_a, bar.b
               Worker 0:  actual time=0.002..5.448 rows=148400 loops=1
         ->  Parallel Hash (actual time=49.467..49.468 rows=166666 loops=2)
               Output: foo.a
```

```
              Buckets: 262144 (originally 8192)  Batches: 4 (originally 1)  Memory
↪  Usage: 5344kB
              Worker 0:  actual time=48.381..48.382 rows=158431 loops=1
              ->  Parallel Seq Scan on public.foo (actual time=0.007..21.265
↪  rows=166666 loops=2)
                    Output: foo.a
                    Filter: ((foo.a % 3) = 0)
                    Rows Removed by Filter: 333334
                    Worker 0:  actual time=0.009..20.909 rows=158431 loops=1
Planning Time: 0.083 ms
Execution Time: 97.567 ms
```

Le *Hash* de `foo` se fait par deux processus qui ne traitent cette fois que la moitié du million de lignes, en en filtrant les ⅔ (la dernière ligne indique quasiment le tiers de 500 000). Le coût de démarrage de ce *hash* parallélisé est cependant assez lourd (la jointure ne commence qu'après 66 ms). Pour la même requête, PostgreSQL 10, qui ne connaît pas le *Parallel Hash Join*, procédait à un *Hash Join* classique et prenait 50 % plus longtemps. Précisons encore qu'augmenter `work_mem` ne change pas le plan, mais permet cas d'accélérer un peu les hachages (réduction du nombre de *batches*).

6.8.13 Nœuds de tris et de regroupements

- Deux nœuds de tri :
  - *Sort*
  - *Incremental Sort*
- Regroupement/agrégation :
  - *Aggregate*
  - *Hash Aggregate*
  - *Group Aggregate*
  - *Mixed Aggregate*
  - *Partial/Finalize Aggregate*
- Paramètres :
  - `enable_hashagg`
  - `work_mem` & `hash_mem_multiplier` (v13)

Pour réaliser un tri, PostgreSQL dispose de deux nœuds : *Sort* et *Incremental Sort*. Leur efficacité va dépendre du paramètre `work_mem` qui va définir la quantité de mémoire que PostgreSQL pourra utiliser pour un tri.

Sort :

```
EXPLAIN (ANALYZE)
  SELECT * FROM employes ORDER BY fonction;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Sort  (cost=1.41..1.44 rows=14 width=43)
       (actual time=0.013..0.014 rows=14 loops=1)
   Sort Key: fonction
   Sort Method: quicksort  Memory: 26kB
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=43)
                     (actual time=0.003..0.004 rows=14 loops=1)
 Planning time: 0.021 ms
 Execution time: 0.021 ms
```

Si le tri ne tient pas en mémoire, l'algorithme de tri gère automatiquement le débordement sur disque (26 Mo ici) :

```
EXPLAIN (ANALYZE)
  SELECT * FROM employes_big ORDER BY fonction;

                                QUERY PLAN
-------------------------------------------------------------------------
 Sort  (cost=70529.24..71776.77 rows=499015 width=40)
   (actual time=252.827..298.948 rows=499015 loops=1)
   Sort Key: fonction
   Sort Method: external sort  Disk: 26368kB
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=40)
                           (actual time=0.003..29.012 rows=499015 loops=1)
 Planning time: 0.021 ms
 Execution time: 319.283 ms
```

Cependant, un tri est coûteux. Donc si un index existe sur le champ de tri, PostgreSQL aura tendance à l'utiliser. Les données sont déjà triées dans l'index, il suffit de le parcourir pour lire les lignes dans l'ordre :

```
EXPLAIN SELECT * FROM employes_big  ORDER BY matricule;

                   QUERY PLAN
-------------------------------------------------
 Index Scan using employes_pkey on employes
   (cost=0.42..17636.65 rows=499015 width=41)
```

Et ce, dans n'importe quel ordre de tri :

```
EXPLAIN SELECT * FROM employes_big ORDER BY matricule DESC;

                     QUERY PLAN
-----------------------------------------------------
 Index Scan Backward using employes_pkey on employes
   (cost=0.42..17636.65 rows=499015 width=41)
```

Le choix du type d'opération de regroupement dépend non seulement des données mises en œuvres, mais elle dépend également beaucoup du paramétrage de PostgreSQL, notamment du paramètre `work_mem`.

Comme vu précédemment, PostgreSQL sait utiliser un index pour trier les données. Cependant, dans certains cas, il ne sait pas utiliser l'index alors qu'il pourrait le faire. Prenons un exemple.

Voici un jeu de données contenant une table à trois colonnes, et un index sur une colonne :

```
DROP TABLE IF exists t1;
CREATE TABLE t1 (c1 integer, c2 integer, c3 integer);
INSERT INTO t1 SELECT i, i+1, i+2 FROM generate_series(1, 10000000) AS i;
CREATE INDEX t1_c2_idx ON t1(c2);
VACUUM ANALYZE t1;
```

PostgreSQL sait utiliser l'index pour trier les données. Par exemple, voici le plan d'exécution pour un tri sur la colonne `c2` (colonne indexée au niveau de l'index `t1_c2_idx`) :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using t1_c2_idx on t1  (cost=0.43..313749.06 rows=10000175 width=12)
                                (actual time=0.016..1271.115 rows=10000000 loops=1)
   Buffers: shared hit=81380
 Planning Time: 0.173 ms
 Execution Time: 1611.868 ms
```

Par contre, si on essaie de trier par rapport aux colonnes `c2` et `c3`, les versions 12 et antérieures ne savent pas utiliser l'index, comme le montre ce plan d'exécution :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Gather Merge  (cost=697287.64..1669594.86 rows=8333480 width=12)
           (actual time=1331.307..3262.511 rows=10000000 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=54149, temp read=55068 written=55246
   ->  Sort  (cost=696287.62..706704.47 rows=4166740 width=12)
         (actual time=1326.112..1766.809 rows=3333333 loops=3)
         Sort Key: c2, c3
         Sort Method: external merge  Disk: 61888kB
         Worker 0:  Sort Method: external merge  Disk: 61392kB
         Worker 1:  Sort Method: external merge  Disk: 92168kB
         Buffers: shared hit=54149, temp read=55068 written=55246
         ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                              (actual time=0.015..337.901 rows=3333333 loops=3)
               Buffers: shared hit=54055
 Planning Time: 0.068 ms
 Execution Time: 3716.541 ms
```

Comme PostgreSQL ne sait pas utiliser un index pour réaliser ce tri, il passe par un parcours de table (parallélisé dans le cas présent), puis effectue le tri, ce qui prend beaucoup de temps, encore plus s'il faut déborder sur disque. La durée d'exécution a plus que doublé.

Incremental Sort :

La version 13 est beaucoup plus maline à cet égard. Elle est capable d'utiliser l'index pour faire un premier tri des données (sur la colonne `c2` d'après notre exemple), puis elle trie les données du résultat par rapport à la colonne `c3` :

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

```
 Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
             (actual time=0.082..2427.099 rows=10000000 loops=1)
   Sort Key: c2, c3
   Presorted Key: c2
   Full-sort Groups: 312500  Sort Method: quicksort  Average Memory: 26kB  Peak
↪  Memory: 26kB
   Buffers: shared hit=81387
   ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000
↪  width=12)
                                  (actual time=0.007..1263.517 rows=10000000 loops=1)
         Buffers: shared hit=81380
 Planning Time: 0.059 ms
 Execution Time: 2766.530 ms
```

La requête en version 12 prenait 3,7 secondes. La version 13 n'en prend que 2,7 secondes. On remarque un nouveau type de nœud, le *Incremental Sort*, qui s'occupe de re-trier les données après un renvoi de données triées, grâce au parcours d'index. Le plan distingue bien la clé déjà triée (`c2`) et celles à trier (`c2, c3`).

L'apport en performance est déjà très intéressant, mais il devient remarquable si on utilise une clause `LIMIT`. Voici le résultat en version 12 :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3 LIMIT 10;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=186764.17..186765.34 rows=10 width=12)
      (actual time=718.576..724.791 rows=10 loops=1)
   Buffers: shared hit=54149
   ->  Gather Merge  (cost=186764.17..1159071.39 rows=8333480 width=12)
                         (actual time=718.575..724.788 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=54149
         ->  Sort  (cost=185764.15..196181.00 rows=4166740 width=12)
                   (actual time=716.606..716.608 rows=10 loops=3)
               Sort Key: c2, c3
               Sort Method: top-N heapsort  Memory: 25kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 25kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
               Buffers: shared hit=54149
               ->  Parallel Seq Scan on t1  (cost=0.00..95722.40 rows=4166740 width=12)
                                    (actual time=0.010..347.085 rows=3333333 loops=3)
                     Buffers: shared hit=54055
 Planning Time: 0.044 ms
 Execution Time: 724.818 ms
```

Et celui en version 13 :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.48..1.24 rows=10 width=12) (actual time=0.027..0.029 rows=10 loops=1)
   Buffers: shared hit=4
   ->  Incremental Sort  (cost=0.48..763746.44 rows=10000000 width=12)
                            (actual time=0.027..0.027 rows=10 loops=1)
```

```
        Sort Key: c2, c3
        Presorted Key: c2
        Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 25kB  Peak
↪  Memory: 25kB
        Buffers: shared hit=4
        ->  Index Scan using t1_c2_idx on t1  (cost=0.43..313746.43 rows=10000000
↪  width=12)
                                          (actual time=0.012..0.014 rows=11 loops=1)
              Buffers: shared hit=4
Planning Time: 0.052 ms
Execution Time: 0.038 ms
```

La requête passe donc de 724 ms à 0,029 ms.

En version 16, l'*Incremental Sort* peut aussi servir à accélérer les `DISTINCT` :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT DISTINCT c2,c1,c3 FROM t1;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Unique (actual time=39.208..2479.229 rows=10000000 loops=1)
   Buffers: shared read=81380
   ->  Incremental Sort (actual time=39.206..1841.165 rows=10000000 loops=1)
         Sort Key: c2, c1, c3
         Presorted Key: c2
         Full-sort Groups: 312500  Sort Method: quicksort  Average Memory: 26kB  Peak
↪  Memory: 26kB
         Buffers: shared read=81380
         ->  Index Scan using t1_c2_idx on t1 (actual time=39.182..949.921
↪  rows=10000000 loops=1)
               Buffers: shared read=81380
 Planning:
   Buffers: shared read=3
 Planning Time: 0.274 ms
 Execution Time: 2679.447 ms
```

Cela devrait accélérer de nombreuses requêtes qui possèdent abusivement une clause `DISTINCT` ajoutée par un ETL, si le premier champ du `DISTINCT` est par chance indexé, comme ici :

```
EXPLAIN (COSTS OFF) SELECT DISTINCT date_commande, *  FROM commandes ;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Unique
   ->  Incremental Sort
         Sort Key: date_commande, numero_commande, client_id, mode_expedition,
↪  commentaire
         Presorted Key: date_commande
         ->  Index Scan using commandes_date_commande_idx on commandes
```

Aggregate :

Concernant les opérations d'agrégations, on retrouve un nœud de type *Aggregate* lorsque la requête réalise une opération d'agrégation simple, sans regroupement :

```
EXPLAIN SELECT count(*) FROM employes;

                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=1.18..1.19 rows=1 width=8)
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=0)
```

Hash Aggregate :

Si l'optimiseur estime que l'opération d'agrégation tient en mémoire (paramètre `work_mem`), il va utiliser un nœud de type *HashAggregate* :

```
EXPLAIN SELECT fonction, count(*) FROM employes GROUP BY fonction;

                           QUERY PLAN
-----------------------------------------------------------------
 HashAggregate  (cost=1.21..1.27 rows=6 width=20)
   Group Key: fonction
   ->  Seq Scan on employes  (cost=0.00..1.14 rows=14 width=12)
```

Avant la version 13, l'inconvénient de ce nœud est que sa consommation mémoire n'est pas limitée par `work_mem`, il continuera malgré tout à allouer de la mémoire. Dans certains cas, heureusement très rares, l'optimiseur peut se tromper suffisamment pour qu'un nœud *HashAggregate* consomme plusieurs gigaoctets de mémoire et sature ainsi la mémoire du serveur.

La version 13 améliore cela en utilisant le disque à partir du moment où la mémoire nécessaire dépasse la multiplication de la valeur du paramètre `work_mem` et celle du paramètre `hash_mem_multiplier` (2 par défaut à partir de la version 15, 1 auparavant). La requête sera plus lente, mais la mémoire ne sera pas saturée.

Group Aggregate :

Lorsque l'optimiseur estime que le volume de données à traiter ne tient pas dans `work_mem` ou quand il peut accéder aux données pré-triées, il utilise plutôt l'algorithme *GroupAggregate* :

```
EXPLAIN SELECT matricule, count(*) FROM employes_big GROUP BY matricule;

                          QUERY PLAN
---------------------------------------------------------------
 GroupAggregate  (cost=0.42..20454.87 rows=499015 width=12)
   Group Key: matricule
   Planned Partitions: 16
   ->  Index Only Scan using employes_big_pkey on employes_big
       (cost=0.42..12969.65 rows=499015 width=4)
```

Mixed Aggregate :

Le *Mixed Aggregate* est très efficace pour les clauses `GROUP BY GROUPING SETS` ou `GROUP BY CUBE` grâce à l'utilisation de *hashs* :

```
EXPLAIN (ANALYZE,BUFFERS)
  SELECT manager, fonction, num_service, COUNT(*)
  FROM employes_big
  GROUP BY CUBE(manager,fonction,num_service) ;

                          QUERY PLAN
-------------------------------------------------------------------
 MixedAggregate  (cost=0.00..34605.17 rows=27 width=27)
        (actual time=581.562..581.573 rows=51 loops=1)
   Hash Key: manager, fonction, num_service
   Hash Key: manager, fonction
   Hash Key: manager
   Hash Key: fonction, num_service
   Hash Key: fonction
   Hash Key: num_service, manager
   Hash Key: num_service
   Group Key: ()
   Batches: 1  Memory Usage: 96kB
   Buffers: shared hit=4664
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=19)
                        (actual time=0.015..35.840 rows=499015 loops=1)
         Buffers: shared hit=4664
 Planning time: 0.223 ms
 Execution time: 581.671 ms
```

(Comparer avec le plan et le temps obtenus auparavant, que l'on peut retrouver avec `SET enable_hashagg TO off;`)

Le calcul d'un agrégat peut être parallélisé. Dans ce cas, deux nœuds sont utilisés : un pour le calcul partiel de chaque processus (*Partial Aggregate*), et un pour le calcul final (*Finalize Aggregate*). Voici un exemple de plan :

```
EXPLAIN (ANALYZE,COSTS OFF)
  SELECT date_embauche, count(*), min(date_embauche), max(date_embauche)
  FROM employes_big
  GROUP BY date_embauche;

                               QUERY PLAN
-----------------------------------------------------------------------
 Finalize GroupAggregate (actual time=92.736..92.740 rows=7 loops=1)
   Group Key: date_embauche
   ->  Sort (actual time=92.732..92.732 rows=9 loops=1)
         Sort Key: date_embauche
         Sort Method: quicksort  Memory: 25kB
         ->  Gather (actual time=92.664..92.673 rows=9 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial HashAggregate
                   (actual time=89.531..89.532 rows=3 loops=3)
                     Group Key: date_embauche
                     ->  Parallel Seq Scan on employes_big
                         (actual time=0.011..35.801 rows=166338 loops=3)
 Planning time: 0.127 ms
 Execution time: 95.601 ms
```

6.8.14 Les autres nœuds

- *Limit*
- *Unique* (`DISTINCT`)
- *Append* (`UNION ALL`), *Except*, *Intersect*
- *Gather* (parallélisme)
- *InitPlan*, *Subplan*, etc.
- *Memoize* (14+)

Limit :

On rencontre le nœud `Limit` lorsqu'on limite le résultat avec l'ordre `LIMIT` :

```
EXPLAIN SELECT * FROM employes_big LIMIT 1;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.00..0.02 rows=1 width=40)
   ->  Seq Scan on employes_big  (cost=0.00..9654.15 rows=499015 width=40)
```

Le nœud *Sort* utilisera dans ce cas une méthode de tri appelée *top-N heapsort* qui permet d'optimiser le tri pour retourner les n premières lignes :

```
EXPLAIN ANALYZE
  SELECT * FROM employes_big ORDER BY fonction LIMIT 5;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Limit  (cost=17942.61..17942.62 rows=5 width=40)
      (actual time=80.359..80.360 rows=5 loops=1)
   ->  Sort  (cost=17942.61..19190.15 rows=499015 width=40)
                (actual time=80.358..80.359 rows=5 loops=1)
         Sort Key: fonction
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on employes_big
             (cost=0.00..9654.15 rows=499015 width=40)
             (actual time=0.005..27.506 rows=499015 loops=1)
 Planning time: 0.035 ms
 Execution time: 80.375 ms
```

Unique :

On retrouve le nœud *Unique* lorsque l'on utilise `DISTINCT` pour dédoublonner le résultat d'une requête :

```
EXPLAIN SELECT DISTINCT matricule FROM employes_big;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Unique  (cost=0.42..14217.19 rows=499015 width=4)
   ->  Index Only Scan using employes_big_pkey on employes_big
       (cost=0.42..12969.65 rows=499015 width=4)
```

On le verra plus loin, il est souvent plus efficace d'utiliser `GROUP BY` pour dédoublonner les résultats d'une requête.

Append, Except, Intersect :

Les nœuds *Append*, *Except* et *Intersect* se rencontrent avec les opérateurs ensemblistes `UNION`, `EXCEPT` et `INTERSECT`. Par exemple, avec `UNION ALL` :

```
EXPLAIN
  SELECT * FROM employes
  WHERE num_service = 2
  UNION ALL
  SELECT * FROM employes
  WHERE num_service = 4;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Append  (cost=0.00..2.43 rows=8 width=43)
   ->  Seq Scan on employes  (cost=0.00..1.18 rows=3 width=43)
         Filter: (num_service = 2)
   ->  Seq Scan on employes employes_1  (cost=0.00..1.18 rows=5 width=43)
         Filter: (num_service = 4)
```

InitPlan :

Le nœud *InitPlan* apparaît lorsque PostgreSQL a besoin d'exécuter une première sous-requête pour ensuite exécuter le reste de la requête. Il est assez rare :

```
EXPLAIN
  SELECT *,
    (SELECT nom_service FROM services WHERE num_service=1)
  FROM employes WHERE num_service = 1;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on employes  (cost=1.05..2.23 rows=2 width=101)
   Filter: (num_service = 1)
   InitPlan 1 (returns $0)
     ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=10)
           Filter: (num_service = 1)
```

SubPlan :

Le nœud *SubPlan* est utilisé lorsque PostgreSQL a besoin d'exécuter une sous-requête pour filtrer les données :

```
EXPLAIN
  SELECT * FROM employes
    WHERE num_service NOT IN (SELECT num_service FROM services
                              WHERE nom_service = 'Consultants');
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on employes  (cost=1.05..2.23 rows=7 width=43)
   Filter: (NOT (hashed SubPlan 1))
```

```
SubPlan 1
  ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=4)
        Filter: ((nom_service)::text = 'Consultants'::text)
```

Gather :

Le nœud *Gather* n'apparaît que s'il y a du parallélisme. Il est utilisé comme nœud de rassemblement des données.

Memoize :

Apparu avec PostgreSQL 14, le nœud *Memoize* est un cache de résultat qui permet d'optimiser les performances d'autres nœuds en mémorisant des données qui risquent d'être accédées plusieurs fois de suite. Pour le moment, ce nœud n'est utilisable que pour les données de l'ensemble interne d'un *Nested Loop*.

D'autres types de nœuds peuvent également être trouvés dans les plans d'exécution. L'annexe décrit tous ces nœuds en détail.

6.9 PROBLÈMES LES PLUS COURANTS

- L'optimiseur se trompe parfois
  - mauvaises statistiques
  - écriture particulière de la requête
  - problèmes connus de l'optimiseur

L'optimiseur de PostgreSQL est sans doute la partie la plus complexe de PostgreSQL. Il se trompe rarement, mais certains facteurs peuvent entraîner des temps d'exécution très lents, voire catastrophiques de certaines requêtes.

6.9.1 Statistiques pas à jour

Les statistiques sont-elles à jour ?

- Traitement lourd
  - faire tout de suite `ANALYZE`
- Table trop grosse
  - régler l'échantillonnage
  - régler l'autovacuum sur cette table
- Retard de mise à jour suite à crash ou restauration

Il est fréquent que les statistiques ne soient pas à jour. Il peut y avoir plusieurs causes.

Gros chargement :

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. 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. À fortiori si le traitement complet est imbriqué dans une seule transaction.

On utilisera l'ordre `ANALYZE table` pour déclencher explicitement la collecte des statistiques après un tel traitement. Notamment, 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;
```

Volumétrie importante :

Un autre problème qui peut se poser avec les statistiques concerne les tables de très forte volumétrie. Dans certains cas, l'échantillon de données ramené par `ANALYZE` n'est pas assez précis pour donner à l'optimiseur de PostgreSQL une vision suffisamment juste des données. Il choisira alors de mauvais plans d'exécution.

Il est possible d'augmenter la taille de l'échantillon de données analysées, ainsi que la précision des statistiques, pour les colonnes où cela est important, à l'aide de cet ordre vu plus haut :

```
ALTER TABLE nom_table ALTER nom_colonne SET STATISTICS valeur;
```

Autre problème : l'autovacuum se base par défaut sur la proportion de lignes modifiées par rapport à celles existantes pour savoir s'il doit déclencher un `ANALYZE` (10 % par défaut). Au fil du temps, beaucoup de tables grossissent en accumulant des lignes statiques. À volume d'activité constante, les lignes actives représentent alors une proportion de plus en plus faible et l'autovacuum se déclenche de moins en moins souvent. Il est courant de descendre la valeur de `autovacuum_analyze_scale_factor` pour compenser. On peut aussi chercher à isoler les données statiques dans leur partition.

Perte des statistiques d'activité après un arrêt brutal :

Le fonctionnement du collecteur des statistiques d'activité implique qu'un arrêt brutal de PostgreSQL les réinitialise. (Il s'agit des statistiques sur les lignes insérées ou modifiées, que l'on trouve notamment dans `pg_stat_user_tables`, pas des statistiques sur les données, qui sont bien préservées.) Elles ne sont pas directement utilisées par le planificateur, mais cette réinitialisation peut mener à un retard dans l'activité de l'autovacuum et la mise à jour des statistiques des données. Après un plantage, un arrêt en mode immédiat ou une restauration physique, il est donc conseillé de relancer un `ANALYZE` général (et même un `VACUUM` ensuite si possible.)

6.9.2 Colonnes corrélées

```
SELECT * FROM corr1 WHERE c1=1 AND c2=1
```

- Si `c1 = 1` pour 20 % des lignes
- et `c2 = 1` pour 10 % des lignes
- Alors le planificateur calcule : 2 % des lignes (20 % × 10 %)
    - Mais en réalité ?
- Pour corriger :

```
CREATE STATISTICS corr1_c1_c2 ON c1,c2 FROM corr1 ;
```

PostgreSQL conserve des statistiques par colonne simple. Mais dans la vie, les valeurs ne sont pas indépendantes. Dans cet exemple, les calculs d'estimation du résultat seront mauvais :

```
CREATE TABLE corr1 AS
SELECT mod(i,5) AS c1 ,mod(i,10) AS c2, i FROM generate_series (1,100000) i;
CREATE INDEX ON corr1 (c1,c2) ;
SELECT c1,c2, count(*) FROM corr1 GROUP BY 1,2 ORDER BY 1,2;
```

```
 c1 | c2 | count
----+----+-------
  0 |  0 | 10000
  0 |  5 | 10000
  1 |  1 | 10000
  1 |  6 | 10000
  2 |  2 | 10000
…
(10 lignes)
```

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`. Faute de mieux, il multiplie les deux estimations et obtient 2 % (20 % × 10 %), soit environ 2000 lignes, ce qui est faux :

```
ANALYZE corr1 ;
EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 1 ;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=29.40..636.28 rows=2059 width=12)
                     (actual time=0.653..3.034 rows=10000 loops=1)
   Recheck Cond: ((c1 = 1) AND (c2 = 1))
   Heap Blocks: exact=541
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..28.88 rows=2059 width=0)
                                    (actual time=0.480..0.481 rows=10000 loops=1)
         Index Cond: ((c1 = 1) AND (c2 = 1))
```

Pour corriger cela, il faut générer des statistiques sur plusieurs colonnes spécifiques. Ce n'est pas automatique, il faut créer un objet statistique avec l'ordre `CREATE STATISTICS`.

```
CREATE STATISTICS corr1_c1_c2 ON c1,c2 FROM corr1 ;
ANALYZE corr1 ;    /* ne pas oublier */

EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 1 ;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on corr1  (cost=139.85..867.39 rows=10103 width=12)
                      (actual time=0.748..3.505 rows=10000 loops=1)
   Recheck Cond: ((c1 = 1) AND (c2 = 1))
   Heap Blocks: exact=541
   ->  Bitmap Index Scan on corr1_c1_c2_idx  (cost=0.00..137.32 rows=10103 width=0)
                                     (actual time=0.563..0.564 rows=10000 loops=1)
         Index Cond: ((c1 = 1) AND (c2 = 1))
```

Dans ce cas précis, de meilleures statistiques ne changent pas le plan. Par contre, avec le critère `c1 = 1 AND c2 = 2` (qui ne renvoie rien), les meilleures statistique permettent de basculer du même *Bitmap Scan* que ci-dessus à un *Index Scan* plus léger :

```
EXPLAIN (ANALYZE, SUMMARY OFF)
SELECT * FROM corr1 WHERE c1 = 1 AND c2 = 2 ;

                            QUERY PLAN
------------------------------------------------------------------
 Index Scan using corr1_c1_c2_idx on corr1  (cost=0.29..8.31 rows=1 width=12)
                                    (actual time=0.010..0.011 rows=0 loops=1)
   Index Cond: ((c1 = 1) AND (c2 = 2))
```

6.9.3 La jointure de trop

- PostgreSQL choisit l'ordre des jointures
  - uniquement pour les X premières tables
  - où X = `join_collapse_limit` (défaut : 8)
- Les jointures supplémentaires sont ajoutées *après*
- … d'où plans non optimaux
- → augmenter `join_collapse_limit` si nécessaire (12-15)
  - ainsi que `from_collapse_limit`

Voici un exemple complet de ce problème. Disons que `join_collapse_limit` est configuré à 2 (le défaut est en réalité 8).

```
SET join_collapse_limit TO 2 ;
```

Nous allons déjà créer deux tables et les peupler avec 1 million de lignes chacune :

```
CREATE TABLE t1 (id integer);
INSERT INTO t1 SELECT generate_series(1, 1000000);
CREATE TABLE t2 (id integer);
INSERT INTO t2 SELECT generate_series(1, 1000000);
ANALYZE;
```

Maintenant, nous allons demander le plan d'exécution pour une jointure entre les deux tables :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=30832.00..70728.00 rows=1000000 width=8)
            (actual time=2355.012..6141.672 rows=1000000 loops=1)
   Hash Cond: (t1.id = t2.id)
   ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000 width=4)
                       (actual time=0.012..1137.629 rows=1000000 loops=1)
   ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
```

```
            (actual time=2354.750..2354.753 rows=1000000 loops=1)
         Buckets: 131072  Batches: 16  Memory Usage: 3227kB
         ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=4)
                             (actual time=0.008..1144.492 rows=1000000 loops=1)
 Planning Time: 0.095 ms
 Execution Time: 7246.491 ms
```

PostgreSQL choisit de lire la table `t2`, de remplir une table de hachage avec le résultat de cette lecture, puis de parcourir la table `t1`, et enfin de tester la condition de jointure grâce à la table de hachage.

Ajoutons maintenant une troisième table, sans données cette fois :

```
CREATE TABLE t3 (id integer);
```

Et ajoutons une jointure à la requête précédente. Cela nous donne cette requête :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id
  JOIN t3 ON t2.id=t3.id;
```

Son plan d'exécution, avec la configuration par défaut de PostgreSQL, sauf le `join_collapse_limit` à 2, est :

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=77972.88..80334.59 rows=2550 width=12)
         (actual time=2902.385..2913.956 rows=0 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Merge Join  (cost=76972.88..79079.59 rows=1062 width=12)
                   (actual time=2894.440..2894.615 rows=0 loops=3)
         Merge Cond: (t1.id = t3.id)
         ->  Sort  (cost=76793.10..77834.76 rows=416667 width=8)
                   (actual time=2894.405..2894.572 rows=1 loops=3)
               Sort Key: t1.id
               Sort Method: external merge  Disk: 5912kB
               Worker 0:  Sort Method: external merge  Disk: 5960kB
               Worker 1:  Sort Method: external merge  Disk: 5848kB
               ->  Parallel Hash Join  (cost=15428.00..32202.28 rows=416667 width=8)
                                       (actual time=1892.071..2400.515 rows=333333 loops=3)
                     Hash Cond: (t1.id = t2.id)
                     ->  Parallel Seq Scan on t1  (cost=0.00..8591.67 rows=416667
↪  width=4)
                                                  (actual time=0.007..465.746 rows=333333
↪  loops=3)
                     ->  Parallel Hash  (cost=8591.67..8591.67 rows=416667 width=4)
                                        (actual time=950.509..950.514 rows=333333 loops=3)
                           Buckets: 131072  Batches: 16  Memory Usage: 3520kB
                           ->  Parallel Seq Scan on t2  (cost=0.00..8591.67
↪  rows=416667 width=4)
                                                        (actual time=0.017..471.653 rows=333333
↪  loops=3)
         ->  Sort  (cost=179.78..186.16 rows=2550 width=4)
                   (actual time=0.028..0.032 rows=0 loops=3)
```

```
               Sort Key: t3.id
               Sort Method: quicksort  Memory: 25kB
               Worker 0:  Sort Method: quicksort  Memory: 25kB
               Worker 1:  Sort Method: quicksort  Memory: 25kB
               ->  Seq Scan on t3  (cost=0.00..35.50 rows=2550 width=4)
                                   (actual time=0.019..0.020 rows=0 loops=3)
 Planning Time: 0.120 ms
 Execution Time: 2914.661 ms
```

En effet, dans ce cas, PostgreSQL va trier les jointures sur les 2 premières tables (soit `t1` et `t2`), et il ajoutera ensuite les autres jointures dans l'ordre indiqué par la requête. Donc, ici, il joint `t1` et `t2`, puis le résultat avec `t3`, ce qui nous donne une requête exécutée en un peu moins de 3 secondes. C'est beaucoup quand on considère que la table `t3` est vide et que le résultat sera forcément vide lui aussi (l'optimiseur a certes estimé trouver 2550 lignes dans `t3`, mais cela reste très faible par rapport aux autres tables).

Maintenant, voici le plan d'exécution pour la même requête avec un `join_collapse_limit` à 3 :

```
EXPLAIN (ANALYZE)
  SELECT * FROM t1
  JOIN t2 ON t1.id=t2.id
  JOIN t3 ON t2.id=t3.id ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=35861.44..46281.24 rows=2550 width=12)
         (actual time=14.943..15.617 rows=0 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join  (cost=34861.44..45026.24 rows=1062 width=12)
                  (actual time=0.119..0.134 rows=0 loops=3)
         Hash Cond: (t2.id = t1.id)
         ->  Parallel Seq Scan on t2  (cost=0.00..8591.67 rows=416667 width=4)
                                      (actual time=0.010..0.011 rows=1 loops=3)
         ->  Hash  (cost=34829.56..34829.56 rows=2550 width=8)
                   (actual time=0.011..0.018 rows=0 loops=3)
               Buckets: 4096  Batches: 1  Memory Usage: 32kB
               ->  Hash Join  (cost=30832.00..34829.56 rows=2550 width=8)
                              (actual time=0.008..0.013 rows=0 loops=3)
                     Hash Cond: (t3.id = t1.id)
                     ->  Seq Scan on t3  (cost=0.00..35.50 rows=2550 width=4)
                                         (actual time=0.006..0.007 rows=0 loops=3)
                     ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=4)
                               (never executed)
                           ->  Seq Scan on t1  (cost=0.00..14425.00 rows=1000000
↪  width=4)
                                               (never executed)
 Planning Time: 0.331 ms
 Execution Time: 15.662 ms
```

Déjà, on voit que la planification a pris plus de temps. La durée reste très basse (0,3 milliseconde) ceci dit.

Cette fois, PostgreSQL commence par joindre `t3` à `t1`. Comme `t3` ne contient aucune ligne, `t1`

n'est même pas parcourue (texte `never executed`) et le résultat de cette première jointure renvoie 0 lignes. De ce fait, la création de la table de hachage est très rapide. La table de hachage étant vide, le parcours de `t2` est abandonné après la première ligne lue. Cela nous donne une requête exécutée en 15 millisecondes.

Une configuration adéquate de `join_collapse_limit` est donc essentielle pour de bonnes performances, notamment sur les requêtes réalisant un grand nombre de jointures.

Il est courant de monter `join_collapse_limit` à 12 si l'on a des requêtes avec autant de tables (y compris celles des vues).

Il existe un paramètre très voisin, `from_collapse_limit`, qui définit à quelle profondeur « aplatir » les sous-requêtes. On le monte à la même valeur que `join_collapse_limit`.

Comme le temps de planification augmente très vite avec le nombre de tables, il vaut mieux ne pas monter `join_collapse_limit` beaucoup plus haut sans tester que ce n'est pas contre-productif. Dans la session concernée, il reste possible de définir :

```
SET join_collapse_limit = … ;
SET from_collapse_limit = … ;
```

À l'inverse, la valeur 1 permet de forcer les jointures dans l'ordre de la clause `FROM`, ce qui est à réserver aux cas désespérés.

Au-delà de 12 tables intervient encore un autre mécanisme, l'optimiseur génétique (GEQO[9]). Pour limiter le nombre de plans étudiés, seul un échantillonnage aléatoire est testé puis recombiné.

6.9.4 Prédicats et statistiques

```
SELECT *
FROM employes_big
WHERE extract('year' from date_embauche) = 2006 ;
```

- L'optimiseur n'a pas de statistiques sur le résultat de la fonction `extract`
- Il estime la sélectivité du prédicat à 0,5 % …
- `CREATE STATISTIC` (v14)

Lorsque les valeurs des colonnes sont transformées par un calcul ou par une fonction, l'optimiseur n'a aucun moyen de 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 alors que la requête va ramener 2495 lignes :

[9] https://docs.postgresql.fr/current/geqo-pg-intro.html

```
EXPLAIN
  SELECT * FROM employes_big
  WHERE extract('year' from date_embauche) = 2006;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Gather  (cost=1000.00..9552.15 rows=2495 width=40)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
       (cost=0.00..8302.65 rows=1040 width=40)
         Filter: (date_part('year'::text,
                 (date_embauche)::timestamp without time zone)
                 = '2006'::double precision)
```

Ces 2495 lignes correspondent à 0,5 % de la table `employes_big`.

Nous avons vu qu'il est préférable de réécrire la requête de manière à pouvoir utiliser les statistiques existantes sur la colonne, mais ce n'est pas toujours aisé ou même possible.

Dans ce cas, on peut se rabattre sur l'ordre `CREATE STATISTICS`. Nous avons vu plus haut qu'il permet de calculer des statistiques sur des résultats d'expressions (ne pas oublier `ANALYZE`).

```
CREATE STATISTICS employe_big_extract
    ON extract('year' from date_embauche) FROM employes_big;
ANALYZE employes_big;
```

Les estimations du plan sont désormais correctes :

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on employes_big  (cost=0.00..12149.22 rows=498998 width=40)
   Filter: (EXTRACT(year FROM date_embauche) = '2006'::numeric)
```

Avant PostgreSQL 14, il est nécessaire de créer un index fonctionnel sur l'expression pour que des statistiques soient calculées.

6.9.5 Problème avec LIKE

```
SELECT * FROM t1 WHERE c2 LIKE 'x%';
```

- PostgreSQL peut utiliser un index dans ce cas
- **MAIS** si l'encodage n'est pas C
  - déclarer l'index avec une classe d'opérateur
  - `varchar_pattern_ops` / `text_pattern_ops`, etc.

```
CREATE INDEX ON matable (champ_texte varchar_pattern_ops);
```

- Outils pour `LIKE '%mot%'` :
  - `pg_trgm`,
  - *Full Text Search*

Il existe cependant une spécificité à PostgreSQL : dans le cas d'une recherche avec préfixe, il peut utiliser directement un index sur la colonne si l'encodage est « C ». Or le collationnement par défaut d'une base est presque toujours `en_US.UTF-8` ou `fr_FR.UTF-8`, selon les choix à l'installation de l'OS ou de PostgreSQL :

```
\l
                      Liste des bases de données
    Nom    | Propriétaire | Encodage | Collationnement | Type caract. | …
-----------+--------------+----------+-----------------+--------------+---
 pgbench   | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
 postgres  | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
 template0 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  | …
 template1 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  | …
 textes_10 | postgres     | UTF8     | en_US.UTF-8     | en_US.UTF-8  |
```

Il faut alors 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);
```

Ce n'est qu'à cette condition qu'un `LIKE 'mot%'` pourra utiliser l'index. Par contre, l'opérateur `varchar_pattern_ops` ne permet pas de trier (`ORDER BY` notamment), faute de collation, il faudra donc peut-être indexer deux fois la colonne.

Un encodage C (purement anglophone) ne nécessite pas l'ajout d'une classe d'opérateurs `varchar_pattern_ops`.

Pour les recherches à l'intérieur d'un texte (`LIKE '%mot%'`), il existe deux autres options :

- `pg_trgm` est une extension permettant de faire des recherches de type par trigramme et un index GIN ou GiST ;
- la *Full Text Search* est une fonctionnalité extrêmement puissante, mais avec une syntaxe différente.

6.9.6 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 (2 312 835 ms), pour ne supprimer aucune ligne. En fait, c'est la vérification de la contrainte `fk_nonbatia21descrsuf_lota30descrlot` qui prend pratiquement tout le temps. C'est d'ailleurs pour cette raison qu'il est recommandé de positionner des index sur les clés étrangères, car cet index permet d'accélérer la recherche liée à la contrainte.

Attention donc aux contraintes de clés étrangères pour les instructions DML !

6.9.7 Dédoublonnage

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

- `DISTINCT` est souvent utilisé pour dédoublonner les lignes
  - souvent utilisé de manière abusive
  - tri !!
  - barrière à l'optimisation
- Penser à :
  - `DISTINCT ON`
  - `GROUP BY`
- Une clé primaire permet de dédoublonner efficacement

Un `DISTINCT` est une opération coûteuse à cause du tri nécessaire. Il est fréquent de le voir ajouté abusivement, « par prudence » ou pour compenser un bug de jointure. De plus il constitue une « barrière à l'optimisation » s'il s'agit d'une partie de requête.

Si le résultat contient telles quelles les clés primaires de toutes les tables jointes, le `DISTINCT` est mathématiquement inutile ! PostgreSQL ne sait malheureusement pas repérer tout seul ce genre de cas.

Quand le dédoublonnage est justifié, il faut savoir qu'il y a deux alternatives principales au `DISTINCT`. Leurs efficacités relatives sont très dépendantes du paramétrage mémoire (`work_mem`) ou des volumétries, ou encore de la présence d'index permettant d'éviter le tri.

- Un `GROUP BY` des colonnes retournées est fastidieux à coder, mais donne parfois un plan efficace. Cette astuce est plus fréquemment utile avant PostgreSQL 13.
- Une autre possibilité est d'utiliser la syntaxe `DISTINCT ON (champs)`, qui renvoie la première ligne rencontrée sur une clé fournie (documentation[10]).

Exemples (sous PostgreSQL 15.2, configuration par défaut sur une petite configuration, cache chaud) :

Il s'agit ici d'afficher la liste des membres des différents services.

- Plan avec `DISTINCT` : notez le tri sur disque.

[10] https://docs.postgresql.fr/current/sql-select.html#SQL-DISTINCT

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT DISTINCT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Unique (actual time=2930.441..4765.048 rows=499015 loops=1)
   ->  Sort (actual time=2930.435..3351.819 rows=499015 loops=1)
         Sort Key: employes_big.matricule, employes_big.nom, employes_big.prenom,
↪  employes_big.fonction, employes_big.manager, employes_big.date_embauche,
↪  employes_big.num_service, services.nom_service, services.localisation,
↪  services.departement
         Sort Method: external merge  Disk: 38112kB
         ->  Hash Join (actual time=0.085..1263.867 rows=499015 loops=1)
               Hash Cond: (employes_big.num_service = services.num_service)
               ->  Seq Scan on employes_big (actual time=0.030..273.710 rows=499015
↪  loops=1)
               ->  Hash (actual time=0.032..0.035 rows=4 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on services (actual time=0.014..0.020 rows=4 loops=1)
 Planning Time: 0.973 ms
 Execution Time: 4938.634 ms
```

- Réécriture avec `GROUP BY` : il n'y a pas de gain en temps dans ce cas précis, mais il n'y a plus de tri sur disque, car l'index sur la clé primaire est utilisé. Noter que PostgreSQL est assez malin pour repérer les clés primaire (ici `matricule` et `num_service`). Il évite alors d'inclure dans les données à regrouper ces clés, et tous les champs de la première table.

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service)
GROUP BY
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement ;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Group (actual time=0.409..5067.075 rows=499015 loops=1)
   Group Key: employes_big.matricule, services.nom_service, services.localisation,
↪  services.departement
   ->  Incremental Sort (actual time=0.405..3925.924 rows=499015 loops=1)
         Sort Key: employes_big.matricule, services.nom_service,
↪  services.localisation, services.departement
         Presorted Key: employes_big.matricule
         Full-sort Groups: 15595  Sort Method: quicksort  Average Memory: 28kB  Peak
↪  Memory: 28kB
         ->  Nested Loop (actual time=0.092..2762.395 rows=499015 loops=1)
```

```
               ->  Index Scan using employes_big_pkey on employes_big (actual
↪  time=0.050..861.828 rows=499015 loops=1)
               ->  Memoize (actual time=0.001..0.001 rows=1 loops=499015)
                     Cache Key: employes_big.num_service
                     Cache Mode: logical
                     Hits: 499011  Misses: 4  Evictions: 0  Overflows: 0  Memory
↪  Usage: 1kB
                     ->  Index Scan using services_pkey on services (actual
↪  time=0.012..0.012 rows=1 loops=4)
                           Index Cond: (num_service = employes_big.num_service)
 Planning Time: 0.900 ms
 Execution Time: 5190.287 ms
```

- Si l'on monte `work_mem` de 4 à 100 Mo, les deux versions basculent sur ce plan, ici plus efficace, qui n'utilise plus l'index, mais ne trie qu'en mémoire, avec la même astuce que ci-dessus.

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 HashAggregate (actual time=3122.612..3849.449 rows=499015 loops=1)
   Group Key: employes_big.matricule, services.nom_service, services.localisation,
↪  services.departement
   Batches: 1  Memory Usage: 98321kB
   ->  Hash Join (actual time=0.136..1354.195 rows=499015 loops=1)
         Hash Cond: (employes_big.num_service = services.num_service)
         ->  Seq Scan on employes_big (actual time=0.050..322.423 rows=499015 loops=1)
         ->  Hash (actual time=0.042..0.046 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on services (actual time=0.020..0.026 rows=4 loops=1)
 Planning Time: 0.967 ms
 Execution Time: 3970.353 ms
```

- La technique la plus propre consiste à indiquer quel est le critère fonctionnel pour dédupliquer. Ici, la structure des tables impose qu'il n'y ait qu'un service par matricule, ce n'est pas évident en regardant la requête. Les index suffisent à ramener une ligne de `service` pour chacune d'`employes_big`.

```
RESET work_mem ;

EXPLAIN (COSTS OFF, ANALYZE)
SELECT DISTINCT ON (matricule)
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Unique (actual time=0.093..3812.414 rows=499015 loops=1)
   ->  Nested Loop (actual time=0.090..2741.919 rows=499015 loops=1)
         ->  Index Scan using employes_big_pkey on employes_big (actual
↪  time=0.049..847.356 rows=499015 loops=1)
         ->  Memoize (actual time=0.001..0.001 rows=1 loops=499015)
               Cache Key: employes_big.num_service
               Cache Mode: logical
```

```
              Hits: 499011  Misses: 4  Evictions: 0  Overflows: 0  Memory Usage: 1kB
               ->  Index Scan using services_pkey on services (actual
↪  time=0.012..0.012 rows=1 loops=4)
                     Index Cond: (num_service = employes_big.num_service)
 Planning Time: 0.711 ms
 Execution Time: 3982.201 ms
```

- Le plus propre et performant reste tout de même de remarquer que les deux clés primaires sont dans le résultat, et que le `DISTINCT` est inutile. La jointure peut se faire de manière plus classique.

```
EXPLAIN (COSTS OFF, ANALYZE)
SELECT
       matricule,
       nom, prenom, fonction, manager, date_embauche,
       num_service, nom_service, localisation, departement
FROM employes_big
JOIN services USING (num_service) ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join (actual time=0.083..1014.796 rows=499015 loops=1)
   Hash Cond: (employes_big.num_service = services.num_service)
   ->  Seq Scan on employes_big (actual time=0.027..214.360 rows=499015 loops=1)
   ->  Hash (actual time=0.032..0.036 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on services (actual time=0.013..0.019 rows=4 loops=1)
 Planning Time: 0.719 ms
 Execution Time: 1117.126 ms
```

Si les `DISTINCT` sont courants et critiques dans votre application, notez que le nœud est parallélisable depuis PostgreSQL 15.

6.9.8 Index inutilisés

- Statistiques pas à jour/peu précises/oubliées
- Trop de lignes retournées
- Ordre des colonnes de l'index (B-tree)
- Index trop gros
- Prédicat avec transformation

```
WHERE col1 + 2 > 5  →  WHERE col1 > 5 - 2
```

- Opérateur non supporté par l'index

```
WHERE col1 <> 'valeur';
```

- Paramètres
  - `random_page_cost`
  - `effective_cache_size`

Il est relativement fréquent de créer soigneusement un index, et que PostgreSQL ne daigne pas l'utiliser. Il peut y avoir plusieurs raisons à cela.

Problème de statistiques :

Les statistiques de la table peuvent être périmées ou imprécises, pour les causes vues plus haut.

Un index fonctionnel possède ses propres statistiques : il faut donc penser à lancer `ANALYZE` après sa création. De même après un `CREATE STATISTICS`.

Nombre de lignes trouvées dans l'index :

Il faut se rappeler que PostgreSQL aura tendance à ne pas utiliser un index s'il doit chercher trop de lignes (ou plutôt de blocs), dans l'index comme dans la table ensuite. Il sera par contre tenté si cet index permet d'éviter des tris ou s'il est couvrant, et pas trop gros. La dispersion des lignes rencontrées dans la table est un facteur également pris en compte.

Colonnes d'un index B-tree :

Un index B-tree multicolonne est inutilisable, en tout cas beaucoup moins performant, si les premiers champs ne sont pas fournis. L'ordre des colonnes a son importance.

Taille d'un index :

PostgreSQL tient compte de la taille des index. Un petit index peut être préféré à un index multicolonne auquel on a ajouté trop de champs pour qu'il soit couvrant.

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 assez représentatif et démontre bien le problème :

```
SELECT * FROM employes WHERE date_embauche + interval '1 month' = '2006-01-01';
```

Avec une telle construction, l'optimiseur ne saura pas tirer partie d'un quelconque index, à moins d'avoir créé un index fonctionnel sur `date_embauche + interval '1 month'`, mais cet index est largement contre-productif par rapport à une réécriture de la requête.

Ce genre de problème se rencontre plus souvent avec des prédicats sur des dates :

```
SELECT * FROM employes WHERE date_trunc('month', date_embauche) = 12;
```

ou encore plus fréquemment rencontré :

```
SELECT * FROM employes WHERE extract('year' from date_embauche) = 2006;
SELECT * FROM employes WHERE upper(prenom) = 'GASTON';
```

Opérateurs non-supportés :

Les index B-tree supportent la plupart des opérateurs généraux sur les variables scalaires (entiers, chaînes, dates, mais pas les types composés comme les 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 :

```
EXPLAIN SELECT * FROM employes_big WHERE num_service<>4;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..8264.74 rows=17 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=7 width=41)
         Filter: (num_service <> 4)
```

La création d'un index partiel permet d'en tirer partie :

```
CREATE INDEX ON employes_big(num_service) WHERE num_service<>4;

EXPLAIN SELECT * FROM employes_big WHERE num_service<>4;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using employes_big_num_service_idx1 on employes_big
   (cost=0.14..12.35 rows=17 width=40)
```

Paramétrage de PostgreSQL

Plusieurs paramètres de PostgreSQL influencent l'optimiseur sur l'utilisation 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 sont 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 voire 2 ou 1.

Pour aller plus loin, n'hésitez pas à consulter cet article de blog[11]

6.9.9 Écriture du SQL

- `NOT IN` avec une sous-requête
  - remplacer par `NOT EXISTS`
- `UNION` entraîne un tri systématique
  - préférer `UNION ALL`
- 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 performante lorsqu'elle est utilisée avec une sous-requête. L'optimiseur ne parvient pas à exécuter ce type de requête efficacement.

```
SELECT *
  FROM services
 WHERE num_service NOT IN (SELECT num_service FROM employes_big);
```

Il est nécessaire de la réécrire avec la clause `NOT EXISTS`, par exemple :

```
SELECT *
  FROM services s
 WHERE NOT EXISTS (SELECT 1
                   FROM employes_big e
                   WHERE s.num_service = e.num_service);
```

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

6.9.10 Absence de hints

- Certains regrettent l'absence de *hints*
- C'est la politique du projet :
  - vouloir ne signifie pas avoir besoin
  - PostgreSQL est un projet libre qui a le luxe de se défaire de la pression du marché
  - cela permet d'être plus facilement et rapidement mis au courant des problèmes de l'optimiseur
- Ne pensez pas être plus intelligent que le planificateur
- Mais il ne peut faire qu'avec ce qu'il a

L'absence de la notion de *hints*, qui permettent au DBA de forcer l'optimiseur à choisir des plans d'exécution jugés pourtant trop coûteux, est voulue. Elle a même été intégrée dans la liste des fonctionnalités dont la communauté ne voulait pas (« *Features We Do Not Want*[12] »).

L'absence des *hints* est très bien expliquée dans un billet de Josh Berkus, ancien membre de la Core Team de PostgreSQL[13] :

Le fait que certains DBA demandent cette fonctionnalité ne veut pas dire qu'ils ont réellement besoin de cette fonctionnalité. Parfois ce sont de mauvaises habitudes d'une époque révolue, où les optimiseurs étaient parfaitement stupides. Ajoutons à cela que les SGBD courants étant des projets commerciaux, ils sont forcément plus poussés à accéder aux demandes des clients, même si ces demandes ne se justifient pas, ou sont le résultat de pressions de pur court terme. Le fait que PostgreSQL soit un projet libre permet justement aux développeurs du projet de choisir les fonctionnalités implémentées suivant leurs idées, et non pas la pression du marché.

Selon le wiki sur le sujet[14], l'avis de la communauté PostgreSQL est que les *hints*, du moins tels qu'ils sont implémentés ailleurs, mènent à une plus grande complexité du code applicatif, donc à des problèmes de maintenabilité, interfèrent avec les mises à jour, risquent d'être contre-productifs au fur et à mesure que vos tables grossissent, et sont généralement inutiles. Sur le long terme, il vaut mieux rapporter un problème rencontré avec l'optimiseur pour qu'il soit définitivement corrigé. L'absence de *hints* permet d'être plus facilement et rapidement mis au courant des problèmes de l'optimiseur. Sur le long terme, cela est meilleur pour le projet comme pour les utilisateurs. Cela a notamment mené à améliorer l'optimiseur et le recueil des statistiques.

L'accumulation de *hints* dans un système a tendance à poser problème lors de l'évolution des besoins, de la volumétrie ou après des mises à jour. Si le plan d'exécution généré n'est pas optimal, il est préférable de chercher à comprendre d'où vient l'erreur. Il est rare que l'optimiseur se trompe : en général c'est lui qui a raison. Mais il ne peut faire qu'avec les statistiques à sa disposition, le modèle qu'il voit,

[12] https://wiki.postgresql.org/wiki/Todo#Features_We_Do_Not_Want
[13] https://it.toolbox.com/blogs/josh-berkus/why-postgresql-doesnt-have-query-hints-020411
[14] https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion

les index que vous avez créés. Nous avons vu dans ce module quelles pouvaient être les causes entraînant des erreurs de plan :

- mauvaise écriture de requête ;
- modèle de données pas optimal ;
- manque d'index adéquats (et PostgreSQL en possède une grande variété) ;
- statistiques pas à jour ;
- statistiques pas assez fines ;
- colonnes corrélées ;
- paramétrage de la mémoire ;
- paramétrage de la profondeur de recherche de l'optimiseur ;
- …

Ajoutons qu'il existe des outils comme PoWA[15] pour vous aider à optimiser des requêtes.

[15] https://powa.readthedocs.io/en/latest/

6.10 OUTILS D'OPTIMISATION

- auto_explain
- plantuner
- HypoPG

6.10.1 auto_explain

- Tracer les plans des requêtes lentes automatiquement
- Contrib officielle
- Mise en place globale (traces) :
  - globale :

```
shared_preload_libraries='auto_explain'   -- redémarrage !
ALTER DATABASE erp SET auto_explain.log_min_duration = '3s' ;
```

  - session :

```
LOAD 'auto_explain' ;
SET auto_explain.log_analyze TO true;
```

L'outil `auto_explain` est habituellement activé quand on a le sentiment qu'une requête devient subitement lente à certains moments, et qu'on suspecte que son plan diffère entre deux exécutions. Elle permet de tracer dans les journaux applicatifs, voire dans la console, le plan de la requête dès qu'elle dépasse une durée configurée.

C'est une « contrib » officielle de PostgreSQL (et non une extension). Tracer systématiquement le plan d'exécution d'une requête souvent répétée prend de la place, et est assez coûteux. C'est donc un outil à utiliser parcimonieusement. En général on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre `auto_explain.log_min_duration`. Par défaut, ce paramètre vaut -1 pour ne tracer aucun plan.

Comme dans un `EXPLAIN` classique, on peut activer les options (par exemple `ANALYZE` ou `TIMING` avec, respectivement, un `SET auto_explain.log_analyze TO true;` ou un `SET auto_explain.log_timing TO true;`) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'`EXPLAIN`, notamment : `auto_explain.log_buffers`, `auto_explain.log_settings`.

Quant à `auto_explain.sample_rate`, il permet de ne tracer qu'un échantillon des requêtes (voir la documentation[16]).

Pour utiliser `auto_explain` globalement, il faut charger la bibliothèque au démarrage dans le fichier `postgresql.conf` via le paramètre `shared_preload_libraries`.

```
shared_preload_libraries='auto_explain'
```

Après un redémarrage de l'instance, il est possible de configurer les paramètres de capture des plans d'exécution par base de données. Dans l'exemple ci-dessous, l'ensemble des requêtes sont tracées sur la base de données `bench`, qui est utilisée par `pgbench`.

```
ALTER DATABASE bench SET auto_explain.log_min_duration = '0';
ALTER DATABASE bench SET auto_explain.log_analyze = true;
```

Attention, l'activation des traces complètes sur une base de données avec un fort volume de requêtes peut être très coûteux.

Un benchmark `pgbench` est lancé sur la base de données `bench` avec 1 client qui exécute 1 transaction par seconde pendant 20 secondes :

```
pgbench -c1 -R1 -T20 bench
```

Les plans d'exécution de l'ensemble les requêtes exécutées par `pgbench` sont alors tracés dans les traces de l'instance.

```
2021-07-01 13:12:55.790 CEST [1705] LOG:  duration: 0.041 ms  plan:
    Query Text: SELECT abalance FROM pgbench_accounts WHERE aid = 416925;
    Index Scan using pgbench_accounts_pkey on pgbench_accounts
        (cost=0.42..8.44 rows=1 width=4) (actual time=0.030..0.032 rows=1 loops=1)
        Index Cond: (aid = 416925)
2021-07-01 13:12:55.791 CEST [1705] LOG:  duration: 0.123 ms  plan:
    Query Text: UPDATE pgbench_tellers SET tbalance = tbalance + -3201 WHERE tid = 19;
    Update on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                               (actual time=0.120..0.121 rows=0 loops=1)
        ->  Seq Scan on pgbench_tellers  (cost=0.00..2.25 rows=1 width=358)
                                         (actual time=0.040..0.058 rows=1 loops=1)
            Filter: (tid = 19)
            Rows Removed by Filter: 99
2021-07-01 13:12:55.797 CEST [1705] LOG:  duration: 0.116 ms  plan:
    Query Text: UPDATE pgbench_branches SET bbalance = bbalance + -3201 WHERE bid = 5;
    Update on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.112..0.114 rows=0 loops=1)
        ->  Seq Scan on pgbench_branches  (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.036..0.038 rows=1 loops=1)
            Filter: (bid = 5)
            Rows Removed by Filter: 9
[...]
```

[16] https://docs.postgresql.fr/current/auto-explain.html

Pour utiliser `auto_explain` uniquement dans la session en cours, il faut penser à descendre au niveau de message `LOG` (défaut de `auto_explain`). On procède ainsi :

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0;
SET auto_explain.log_analyze = true;
SET client_min_messages to log;
SELECT count(*)
   FROM pg_class, pg_index
   WHERE oid = indrelid AND indisunique;

LOG:  duration: 1.273 ms  plan:
Query Text: SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
Aggregate  (cost=38.50..38.51 rows=1 width=8)
    (actual time=1.247..1.248 rows=1 loops=1)
  ->  Hash Join  (cost=29.05..38.00 rows=201 width=0)
         (actual time=0.847..1.188 rows=198 loops=1)
        Hash Cond: (pg_index.indrelid = pg_class.oid)
        ->  Seq Scan on pg_index  (cost=0.00..8.42 rows=201 width=4)
                      (actual time=0.028..0.188 rows=198 loops=1)
              Filter: indisunique
              Rows Removed by Filter: 44
        ->  Hash  (cost=21.80..21.80 rows=580 width=4)
           (actual time=0.726..0.727 rows=579 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 29kB
              ->  Seq Scan on pg_class  (cost=0.00..21.80 rows=580 width=4)
                            (actual time=0.016..0.373 rows=579 loops=1)
 count
-------
   198
```

`auto_explain` est aussi un moyen de suivre les plans au sein de fonctions. Par défaut, un plan n'indique les compteurs de blocs *hit*, *read*, *temp*… que de l'appel global à la fonction.

Une fonction simple en PL/pgSQL est définie pour récupérer le solde le plus élevé dans la table `pgbench_accounts` :

```
CREATE OR REPLACE function f_max_balance() RETURNS int AS $$
    DECLARE
        acct_balance int;
    BEGIN
        SELECT max(abalance)
        INTO acct_balance
        FROM pgbench_accounts;
        RETURN acct_balance;
    END;
$$ LANGUAGE plpgsql ;
```

Un simple `EXPLAIN ANALYZE` de l'appel de la fonction ne permet pas d'obtenir le plan de la requête `SELECT max(abalance) FROM pgbench_accounts` contenue dans la fonction :

```
EXPLAIN (ANALYZE,VERBOSE) SELECT f_max_balance();
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=4) (actual time=49.214..49.216 rows=1 loops=1)
   Output: f_max_balance()
 Planning Time: 0.149 ms
 Execution Time: 49.326 ms
```

Par défaut, `auto_explain` ne va pas capturer plus d'information que la commande `EXPLAIN ANALYZE`. Le fichier log de l'instance capture le même plan lorsque la fonction est exécutée.

```
2021-07-01 15:39:05.967 CEST [2768] LOG:  duration: 42.937 ms  plan:
    Query Text: select f_max_balance();
    Result  (cost=0.00..0.26 rows=1 width=4)
            (actual time=42.927..42.928 rows=1 loops=1)
```

Il est cependant possible d'activer le paramètre `log_nested_statements` avant l'appel de la fonction, de préférence uniquement dans la ou les sessions concernées :

```
\c bench
SET auto_explain.log_nested_statements = true;
SELECT f_max_balance();
```

Le plan d'exécution de la requête SQL est alors visible dans les traces de l'instance :

```
2021-07-01 14:58:40.189 CEST [2202] LOG:  duration: 58.938 ms  plan:
Query Text: select max(abalance)
            from pgbench_accounts
Finalize Aggregate
(cost=22632.85..22632.86 rows=1 width=4)
(actual time=58.252..58.935 rows=1 loops=1)
    ->  Gather
        (cost=22632.64..22632.85 rows=2 width=4)
        (actual time=57.856..58.928 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate
                (cost=21632.64..21632.65 rows=1 width=4)
                (actual time=51.846..51.847 rows=1 loops=3)
            ->  Parallel Seq Scan on pgbench_accounts
                    (cost=0.00..20589.51 rows=417251 width=4)
                    (actual time=0.014..29.379 rows=333333 loops=3)
```

pgBadger est capable de lire les plans tracés par `auto_explain`, de les intégrer à son rapport et d'inclure un lien vers depesz.com[17] pour une version plus lisible.

[17] https://explain.depesz.com/

6.10.2 Extension plantuner

- Pour :
  - interdire certains index
  - forcer à zéro les statistiques d'une table vide
- Intéressant en développement pour tester les plans
  - pas en production !

Cette extension est disponible à cette adresse[18] (le miroir GitHub ne semble pas maintenu). Oleg Bartunov, l'un de ses auteurs, a publié en 2018 un article intéressant[19] sur son utilisation.

Il faudra récupérer le source et le compiler. La configuration est basée sur trois paramètres :

- `plantuner.enable_index` pour préciser les index à activer ;
- `plantuner.disable_index` pour préciser les index à désactiver ;
- `plantuner.fix_empty_table` pour forcer à zéro les statistiques des tables de 0 bloc.

Ils sont configurables à chaud, comme le montre l'exemple suivant :

```
LOAD 'plantuner';
EXPLAIN (COSTS OFF)
  SELECT * FROM employes_big WHERE date_embauche='1000-01-01';

                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
   Index Cond: (date_embauche = '1000-01-01'::date)

SET plantuner.disable_index='employes_big_date_embauche_idx';

EXPLAIN (COSTS OFF)
  SELECT * FROM employes_big WHERE date_embauche='1000-01-01';

                      QUERY PLAN
------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big
         Filter: (date_embauche = '1000-01-01'::date)
```

Un des intérêts de cette extension est de pouvoir interdire l'utilisation d'un index, afin de pouvoir ensuite le supprimer de manière transparente, c'est-à-dire sans bloquer aucune requête applicative.

[18] http://www.sai.msu.su/~megera/wiki/plantuner
[19] https://obartunov.livejournal.com/197604.html

Cependant, généralement, cette extension a sa place sur un serveur de développement pour bien comprendre les choix de planification, pas sur un serveur de production. En tout cas, pas dans le but de tromper le planificateur.

Comme avec toute extension en C, un bug est susceptible de provoquer un plantage complet du serveur.

6.10.3 Extension pg_plan_hint

- Pour :
  - forcer l'utilisation d'un nœud entre deux tables
  - imposer une valeur de paramètre
  - appliquer automatiquement ces *hints* à des requêtes

Cette extension existe depuis longtemps. Elle doit être compilée et installée depuis le dépôt Github[20].

La documentation[21] en anglais peut être complétée par la version japonaise[22] plus à jour, ou cet article[23].

Comme avec toute extension en C, un bug est susceptible de provoquer un plantage complet du serveur !

6.10.4 Extension HypoPG

- Extension PostgreSQL
- Création d'index hypothétiques pour tester leur intérêt
  - avant de les créer pour de vrai
- Limitations : surtout B-Tree, statistiques

[20] https://github.com/ossc-db/pg_hint_plan
[21] http://pghintplan.osdn.jp/pg_hint_plan.html
[22] http://pghintplan.osdn.jp/pg_hint_plan-ja.html
[23] https://docs.yugabyte.com/latest/explore/query-1-performance/pg-hint-plan/

Cette extension est disponible sur GitHub[24] et dans les paquets du PGDG. Il existe trois fonctions principales et une vue :

- `hypopg_create_index()` pour créer un index hypothétique ;
- `hypopg_drop_index()` pour supprimer un index hypothétique particulier ou `hypopg_reset()` pour tous les supprimer ;
- `hypopg_list_indexes` pour les lister.

Un index hypothétique n'existe que dans la session, ni en mémoire ni sur le disque, mais le planificateur le prendra en compte dans un `EXPLAIN` simple (évidemment pas un `EXPLAIN ANALYZE`). En quittant la session, tous les index hypothétiques restants et créés sur cette session sont supprimés.

L'exemple suivant est basé sur la base dont le script peut être téléchargé sur https://dali.bo/tp_employes_services.

```
CREATE EXTENSION hypopg;

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                                QUERY PLAN
--------------------------------------------------------------------------------
 Gather  (cost=1000.00..8263.14 rows=1 width=41)
   Workers Planned: 2
   ->  Parallel Seq Scan on employes_big  (cost=0.00..7263.04 rows=1 width=41)
         Filter: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_create_index('CREATE INDEX ON employes_big(prenom)');

 indexrelid |            indexname
------------+---------------------------------
      24591 | <24591>btree_employes_big_prenom

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using <24591>btree_employes_big_prenom on employes_big
                              (cost=0.05..4.07 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)

SELECT * FROM hypopg_list_indexes;

 indexrelid |            indexname             | nspname |   relname    | amname
------------+----------------------------------+---------+--------------+--------
      24591 | <24591>btree_employes_big_prenom | public  | employes_big | btree

SELECT * FROM hypopg_reset();

 hypopg_reset
--------------

(1 row)
```

[24] https://github.com/HypoPG/hypopg

```
CREATE INDEX ON employes_big(prenom);

EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using employes_big_prenom_idx on employes_big
                        (cost=0.42..4.44 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
```

Le cas idéal d'utilisation est l'index B-Tree sur une colonne. Un index fonctionnel est possible, mais, faute de statistiques disponibles avant la création réelle de l'index, les estimations peuvent être fausses. Les autres types d'index sont moins bien ou non supportées.

6.11 CONCLUSION

- Planificateur très avancé
- Ne pensez pas être plus intelligent que lui
- Il faut bien comprendre son fonctionnement

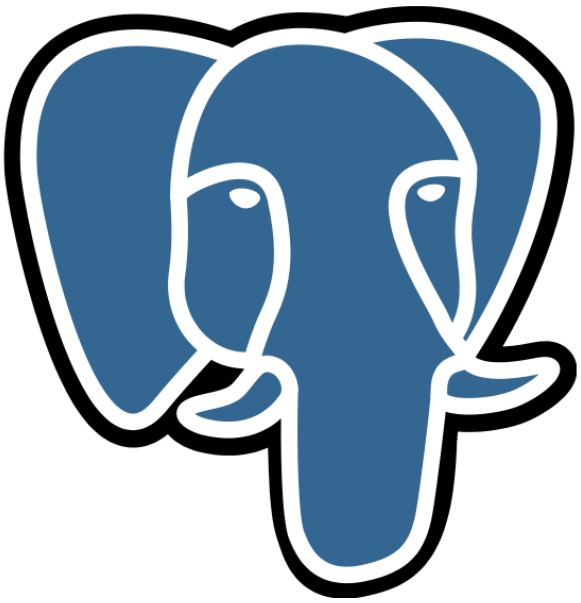
6.11.1 Questions

N'hésitez pas, c'est le moment !

6.12 QUIZ

https://dali.bo/j2_quiz

7/ SQL : Ce qu’il ne faut pas faire



7.1 DES MAUVAISES PRATIQUES

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

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

7.2 PROBLÈMES DE MODÉLISATION

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

7.2.1 Que veut dire « relationnel » ?

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

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

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

7.2.2 Quelques rappels sur le modèle relationnel

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

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

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

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

7.2.3 Formes normales

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

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

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

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

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

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

7.3 ATOMICITÉ

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

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

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

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

7.3.1 Atomicité - mauvais exemple

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

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

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

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

ce qui sera évidemment très inefficace.

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

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

7.3.2 Atomicité - propositions

- Champs dédiés :

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

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

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

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

- une contrainte simple :

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

- un type « énumération[2] » :

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

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

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

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

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

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

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

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

7.4 CONTRAINTES ABSENTE

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

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

L'absence de contraintes a souvent des conséquences catastrophiques.

7.4.1 Conséquences de l'absence de contraintes

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

De plus, l'absence de contraintes va également entraîner des problèmes d'intégrité des données. Il est par exemple très compliqué de se prémunir efficacement contre une *race condition*[3] en l'absence de clé étrangère.

Imaginez le scénario suivant :

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

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

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

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

Lorsque ces problèmes d'intégrité seront détectés, il s'en suivra également la création de procédures de vérification de cohérence des données qui vont aussi alourdir les développements, entraînant ainsi un travail supplémentaire pour trouver et corriger les incohérences. Ce qui a été gagné d'un côté est perdu de l'autre, mais sous une forme différente.

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

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

7.4.2 Suspension des contraintes le temps d'une transaction

- Solution :
  - contraintes `DEFERRABLE` !

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

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

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

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

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

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

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

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

```
BEGIN ;
SET CONSTRAINTS ALL DEFERRED ;

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

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

COMMIT;
```

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

7.5 STOCKAGE ENTITÉ-CLÉ-VALEUR

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

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

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

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

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

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

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

 id
----
  1

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

Un tel modèle peut sembler souple mais pose plusieurs problèmes. Le premier concerne l'intégrité des données. Il n'est pas possible de garantir la présence d'un attribut comme on le ferait avec une

contrainte `NOT NULL`. Si l'on souhaite stocker des données dans un autre format qu'une chaîne de caractère, pour bénéficier des contrôles de la base de données sur ce type, la seule solution est de créer autant de colonnes d'attributs qu'il y a de types de données à représenter. Ces colonnes ne permettront pas d'utiliser des contraintes `CHECK` pour garantir la cohérence des valeurs stockées avec ce qui est attendu, car les attributs peuvent stocker n'importe quelle donnée.

7.5.1 Stockage Entité-Clé-Valeur : exemple

Comment lister tous les DBA ?

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

7.5.2 Stockage Entité-Clé-Valeur : requête associée

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

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

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

7.5.3 Stockage Entité-Clé-Valeur, hstore, JSON

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

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

Dans d'autres cas où le nombre de champs est *vraiment* élevé et variable, il vaut mieux utiliser un type de données de PostgreSQL qui est approprié, comme `hstore` qui permet de stocker des données sous la forme `clé->valeur`. On conserve ainsi l'intégrité des données (on n'a qu'une ligne par personne), on évite de très nombreuses jointures source d'erreurs et de ralentissements, et même de la place disque.

De plus, ce type de données peut être indexé pour garantir de bons temps de réponses des requêtes qui nécessitent des recherches sur certaines clés ou certaines valeurs.

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

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

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

SELECT * FROM personnes;

 id |              attributs
----+--------------------------------------
  1 | "nom"=>"Prunelle", "prenom"=>"Léon"
  2 | "nom"=>"Lagaffe", "prenom"=>"Gaston"
  3 | "nom"=>"DeMaesmaker"

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

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

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

 id |     nom
----+-------------
```

```
1 | Prunelle
2 | Lagaffe
3 | DeMaesmaker
```

Le principe du JSON est similaire.

7.6 ATTRIBUTS MULTICOLONNES

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

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

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

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

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

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

```
CREATE TABLE personnes (
```

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

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

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

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

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

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

SELECT *
  FROM personnes;

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

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

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

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

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

 per_id |   nom    | pnom |   numero
--------+---------+------+------------
      1 | Simpson | Omer | 0607080910
      1 | Simpson | Omer | 0102030420
```

7.7 NOMBREUSES LIGNES DE PEU DE COLONNES

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

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

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

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

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

SELECT pg_size_pretty(pg_relation_size ('valeurs_capteur')) ;

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

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

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

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

SELECT pg_size_pretty(pg_relation_size ('valeurs_capteur_2'));

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

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

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

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

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

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

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

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

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

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

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

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

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

7.8 TABLES AUX TRÈS NOMBREUSES COLONNES

Tables à plusieurs dizaines, voire centaines de colonnes :

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

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

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

7.9 CHOIX D'UN TYPE NUMÉRIQUE

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

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

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

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

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

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

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

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

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

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

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

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

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

Pour plus de détails, voir la documentation officielle :

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

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

7.10 COLONNE DE TYPE VARIABLE

Plus rarement, on rencontre aussi :

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

On rencontre parfois ce genre de choses :

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

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

Cela pose un gros problème de modélisation : la colonne a un type de contenu qui dépend de l'information qu'elle contient. On va aussi avoir un problème de performance en joignant cette chaîne à la clé numérique de la table `tournée`. Le moteur n'aura que deux choix : convertir la chaîne en numérique, avec une exception à la clé en essayant de convertir « ANNULÉE », ou bien (ce qu'il fera) convertir le numérique de la table `tournee` en chaîne. Cette dernière méthode rendra l'accès à l'identifiant de tournée par index impossible. D'où un parcours complet (*Seq Scan*) de la table `tournée` à chaque accès et des performances qui décroissent au fur et à mesure que la table grossit.

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

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

7.11 PROBLÈMES COURANTS D'ÉCRITURE DE REQUÊTES

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

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

7.12 NULL

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

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

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

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

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

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

7.13 ORDRE IMPLICITE DES COLONNES

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

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

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

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

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

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

Exemples

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

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

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

```
SELECT * FROM insere ;

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

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

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

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

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

SELECT * FROM insere ;

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

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

7.14 CODE SPAGHETTI

Le problème est similaire à tout autre langage :

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

Un exemple (sous Oracle) :

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

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

```
    GrilleDePrix.datem                              AS GrilleDePrix_34_1,
    GrilleDePrix.libelle                            AS GrilleDePrix_34_3,
    FicheAgreage.code                               AS FicheAgreage_38_0,
    Codelec.iddp_periact                            AS Codelec_40_1,
    Codelec.libelle                                 AS Codelec_40_2,
    Codelec.niveau1                                 AS Codelec_40_3,
    Codelec.niveau2                                 AS Codelec_40_4,
    Codelec.niveau3                                 AS Codelec_40_5,
    Codelec.niveau4                                 AS Codelec_40_6,
    PerimetreActivite.code                          AS PerimetreActivite_41_0,
    DonneesPersonnalisablesCodelec.gestionreftech AS DonneesPersonnalisab_42_0,
    ClassificationArticleInterne.id                 AS ClassificationArticl_43_0,
    ClassificationArticleInterne.niveau1            AS ClassificationArticl_43_2,
    DossierCommercial.id                            AS DossierCommercial_52_0,
    DossierCommercial.codefourndc                   AS DossierCommercial_52_1,
    DossierCommercial.anneedc                       AS DossierCommercial_52_3,
    DossierCommercial.codeclassdc                   AS DossierCommercial_52_4,
    DossierCommercial.numversiondc                  AS DossierCommercial_52_5,
    DossierCommercial.indice                        AS DossierCommercial_52_6,
    DossierCommercial.code_ss_classement            AS DossierCommercial_52_7,
    OrigineNegociation.code                         AS OrigineNegociation_53_0,
    MotifBlocageInformation.libellelong             AS MotifBlocageInformat_54_3,
    ArbreLogistique.id                              AS ArbreLogistique_63_1,
    ArbreLogistique.codesap                         AS ArbreLogistique_63_5,
    Fournisseur.code                                AS Fournisseur_66_0,
    Fournisseur.nom                                 AS Fournisseur_66_2,
    Filiere.code                                    AS Filiere_67_0,
    Filiere.nom                                     AS Filiere_67_2,
    ValorisationAchat.val_ach_patc                  AS Valorisation_74_3,
    LienPrixVente.code                              AS LienPrixVente_76_0,
    LienPrixVente.datem                             AS LienPrixVente_76_1,
    LienGratuite.code                               AS LienGratuite_78_0,
    LienGratuite.datem                              AS LienGratuite_78_1,
    LienCoordonnable.code                           AS LienCoordonnable_79_0,
    LienCoordonnable.datem                          AS LienCoordonnable_79_1,
    LienStatistique.code                            AS LienStatistique_81_0,
    LienStatistique.datem                           AS LienStatistique_81_1
FROM   da_article Article
       join (SELECT idarticle,
                    poids,
                    ROW_NUMBER()
                      over (
                        PARTITION BY RNA.id
                        ORDER BY INNERSEARCH.poids) RN,
                    titre,
                    nom,
                    prenom
             FROM   da_article RNA
                    join (SELECT idarticle,
                        pkg_db_indexation.CALCULPOIDSMOTS(chaine,
                            'foire vins%') AS POIDS,
                        DECODE(index_clerecherche, 'Piste.titre', chaine,
                                                   '')               AS TITRE,
                        DECODE(index_clerecherche, 'Artiste.nom_prenom',
                            SUBSTR(chaine, 0, INSTR(chaine, '_') - 1),
                                                   '')               AS NOM,
```

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

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

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

```
left join dp_periact PerimetreActivite
       ON PerimetreActivite.id = Codelec.iddp_periact
left join dp_perscod DonneesPersonnalisablesCodelec
       ON Codelec.id = DonneesPersonnalisablesCodelec.iddp_codelec
          AND DonneesPersonnalisablesCodelec.db_suplog = 0
          AND DonneesPersonnalisablesCodelec.iddb_sitecl = 1012124
left join dp_clsart ClassificationArticleInterne
       ON DonneesPersonnalisablesCodelec.iddp_clsart =
         ClassificationArticleInterne.id
left join da_artdeno ArticleDenormalise
       ON Article.id = ArticleDenormalise.idda_article
left join df_clasmnt ClassementFournisseur
       ON ArticleDenormalise.iddf_clasmnt = ClassementFournisseur.id
left join tr_dosclas DossierDeClassement
       ON ClassementFournisseur.id = DossierDeClassement.iddf_clasmnt
          AND DossierDeClassement.date_deb <= '2013-09-27'
          AND COALESCE(DossierDeClassement.date_fin,
             TO_DATE('31129999', 'DDMMYYYY')) >= '2013-09-27'
left join tr_doscomm DossierCommercial
       ON DossierDeClassement.idtr_doscomm = DossierCommercial.id
left join dp_valdico OrigineNegociation
       ON DossierCommercial.iddp_dossref = OrigineNegociation.id
left join dp_motbloc MotifBlocageInformation
       ON MotifBlocageInformation.id = ArticleDenormalise.idda_motinf
left join da_arblogi ArbreLogistique
       ON Article.id = ArbreLogistique.idda_article
          AND ArbreLogistique.princ = 1
          AND ArbreLogistique.db_suplog = 0
left join df_filiere Filiere
       ON ArbreLogistique.iddf_filiere = Filiere.id
left join df_fourn Fournisseur
       ON Filiere.iddf_fourn = Fournisseur.id
left join od_dosal dossierALValo
       ON dossierALValo.idda_arblogi = ArbreLogistique.id
          AND dossierALValo.idod_dossier IS NULL
left join tt_val_dal valoDossier
       ON valoDossier.idod_dosal = dossierALValo.id
          AND valoDossier.estarecalculer = 0
left join tt_valo ValorisationAchat
       ON ValorisationAchat.idtt_val_dal = valoDossier.id
          AND ValorisationAchat.date_modif_retro IS NULL
          AND ValorisationAchat.date_debut_achat <= '2013-09-27'
          AND COALESCE(ValorisationAchat.date_fin_achat,
             TO_DATE('31129999', 'DDMMYYYY')) >= '2013-09-27'
          AND ValorisationAchat.val_ach_pab IS NOT NULL
left join da_lienart assoALPXVT
       ON assoALPXVT.idda_article = Article.id
          AND assoALPXVT.iddp_typlien = 14893
left join da_lien LienPrixVente
       ON LienPrixVente.id = assoALPXVT.idda_lien
left join da_lienart assoALGRAT
       ON assoALGRAT.idda_article = Article.id
          AND assoALGRAT.iddp_typlien = 14894
left join da_lien LienGratuite
       ON LienGratuite.id = assoALGRAT.idda_lien
left join da_lienart assoALCOOR
```

```
               ON assoALCOOR.idda_article = Article.id
                  AND assoALCOOR.iddp_typlien = 14899
          left join da_lien LienCoordonnable
                 ON LienCoordonnable.id = assoALCOOR.idda_lien
          left join da_lienal assoALSTAT
                 ON assoALSTAT.idda_arblogi = ArbreLogistique.id
                  AND assoALSTAT.iddp_typlien = 14897
          left join da_lien LienStatistique
                 ON LienStatistique.id = assoALSTAT.idda_lien WHERE
SEARCHMC.rn = 1
   AND ( ValorisationAchat.id IS NULL
          OR ValorisationAchat.date_debut_achat = (
                 SELECT MAX(VALMAX.date_debut_achat)
                 FROM   tt_valo VALMAX
                 WHERE  VALMAX.idtt_val_dal = ValorisationAchat.idtt_val_dal
                    AND VALMAX.date_modif_retro IS NULL
                    AND VALMAX.val_ach_pab IS NOT NULL
                    AND VALMAX.date_debut_achat <= '2013-09-27') )
   AND ( Article.id IN (SELECT A.id
                        FROM   da_article A
                               join du_ucutiar AssoUcUtiAr
                                 ON AssoUcUtiAr.idda_article = A.id
                               join du_asucuti AssoUcUti
                                 ON AssoUcUti.id = AssoUcUtiAr.iddu_asucuti
                        WHERE  ( AssoUcUti.iddu_uti IN ( 90000000000022 ) )
                           AND a.iddb_sitecl = 1012124) )
   AND Article.db_suplog = 0
ORDER  BY SEARCHMC.poids ASC
```

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

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

7.15 RECHERCHE TEXTUELLE

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

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

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

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

Exemples

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

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

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

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

CREATE EXTENSION pg_trgm;
```

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

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

                                QUERY PLAN

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

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

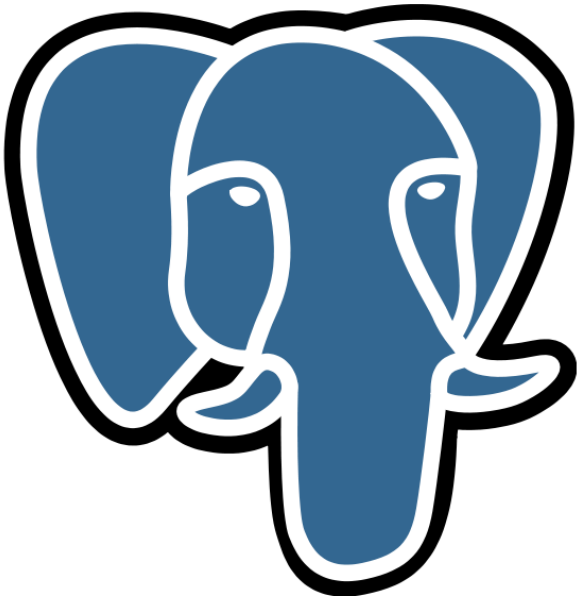
7.16 CONCLUSION

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

7.17 QUIZ

https://dali.bo/s8_quiz

8/ PL/pgSQL : les bases



8.1 PRÉAMBULE

- Vous apprendrez :
  - à choisir si vous voulez écrire du PL
  - à choisir votre langage PL
  - les principes généraux des langages PL autres que PL/pgSQL
  - les bases de PL/pgSQL

Ce module présente la programmation PL/pgSQL. Il commence par décrire les routines stockées et les différents langages disponibles. Puis il aborde les bases du langage PL/pgSQL, autrement dit :

- comment installer PL/pgSQL dans une base PostgreSQL ;
- comment créer un squelette de fonction ;
- comment déclarer des variables ;
- comment utiliser les instructions de base du langage ;
- comment créer et manipuler des structures ;
- comment passer une valeur de retour de la fonction à l'appelant.

8.1.1 Au menu

- Présentation du PL et des principes
- Présentations de PL/pgSQL et des autres langages PL
- Installation d'un langage PL
- Détails sur PL/pgSQL

8.1.2 Objectifs

- Comprendre les cas d'utilisation d'une routine PL/pgSQL
- Choisir son langage PL en connaissance de cause
- Comprendre la différence entre PL/pgSQL et les autres langages PL
- Écrire une routine simple en PL/pgSQL
  - et même plus complexe

8.2 INTRODUCTION

8.2.1 Qu'est-ce qu'un PL ?

- PL = *Procedural Language*
- 3 langages activés par défaut :
  - C
  - SQL
  - PL/pgSQL

PL est l'acronyme de « Procedural Languages ». En dehors du C et du SQL, tous les langages acceptés par PostgreSQL sont des PL.

Par défaut, trois langages sont installés et activés : C, SQL et PL/pgSQL.

8.2.2 Quels langages PL sont disponibles ?

- Installé par défaut :
  - PL/pgSQL
- Intégrés au projet :
  - PL/Perl
  - PL/Python
  - PL/Tcl
- Extensions tierces :
  - PL/java, PL/R, PL/v8 (Javascript), PL/sh …
  - extensible à volonté

Les quatre langages PL supportés nativement (en plus du C et du SQL bien sûr) sont décrits en détail dans la documentation officielle :

- PL/PgSQL[1] est intégré par défaut dans toute nouvelle base (de par sa présence dans la base modèle **template1**) ;
- PL/Tcl[2] (existe en version *trusted* et *untrusted*) ;

[1] https://docs.postgresql.fr/current/plpgsql.html
[2] https://docs.postgresql.fr/current/pltcl.html

- PL/Perl[3] (existe en version *trusted* et *untrusted*) ;
- PL/Python[4] (uniquement en version *untrusted*).

D'autres langages PL sont accessibles en tant qu'extensions tierces. Les plus stables sont mentionnés dans la documentation[5], comme PL/Java[6] ou PL/R[7]. Ils réclament généralement d'installer les bibliothèques du langage sur le serveur.

Une liste plus large est par ailleurs disponible sur le wiki PostgreSQL[8], Il en ressort qu'au moins 16 langages sont disponibles, dont 10 installables en production. De plus, il est possible d'en ajouter d'autres, comme décrit dans la documentation[9].

8.2.3 Langages *trusted* vs *untrusted*

- *Trusted* = langage de confiance :
  - ne permet que l'accès à la base de données
  - donc pas aux systèmes de fichiers, aux sockets réseaux, etc.
  - SQL, PL/pgSQL, PL/Perl, PL/Tcl
- *Untrusted*:
  - PL/Python, C…
  - PL/TclU, PL/PerlU

Les langages de confiance ne peuvent accéder qu'à la base de données. Ils ne peuvent pas accéder aux autres bases, aux systèmes de fichiers, au réseau, etc. Ils sont donc confinés, ce qui les rend moins facilement utilisables pour compromettre le système. PL/pgSQL est l'exemple typique. Mais de ce fait, ils offrent moins de possibilités que les autres langages.

Seuls les superutilisateurs peuvent créer une routine dans un langage *untrusted*. Par contre, ils peuvent ensuite donner les droits d'exécution à ces routines aux autres rôles dans la base :

```
GRANT EXECUTE ON FUNCTION nom_fonction TO un_role ;
```

[3] https://docs.postgresql.fr/current/plperl.html
[4] https://docs.postgresql.fr/current/plpython.html
[5] https://docs.postgresql.fr/current/external-pl.html
[6] https://tada.github.io/pljava/
[7] https://github.com/postgres-plr/plr
[8] https://wiki.postgresql.org/wiki/PL_Matrix
[9] https://docs.postgresql.fr/current/plhandler.html

8.2.4 Les langages PL de PostgreSQL

- Les langages PL fournissent :
  - des fonctionnalités procédurales dans un univers relationnel
  - des fonctionnalités avancées du langage PL choisi
  - des performances de traitement souvent supérieures à celles du même code côté client

La question se pose souvent de placer la logique applicative du côté de la base, dans un langage PL, ou des clients. Il peut y avoir de nombreuses raisons en faveur de la première option. Simplifier et centraliser des traitements clients directement dans la base est l'argument le plus fréquent. Par exemple, une insertion complexe dans plusieurs tables, avec mise en place d'identifiants pour liens entre ces tables, peut évidemment être écrite côté client. Il est quelquefois plus pratique de l'écrire sous forme de PL. Les avantages sont :

Centralisation du code :

Si plusieurs applications ont potentiellement besoin d'opérer un même traitement, à fortiori dans des langages différents, porter cette logique dans la base réduit d'autant les risques de *bugs* et facilite la maintenance.

Une règle peut être que tout ce qui a trait à l'intégrité des données devrait être exécuté au niveau de la base.

Performances :

Le code s'exécute localement, directement dans le moteur de la base. Il n'y a donc pas tous les changements de contexte et échanges de messages réseaux dus à l'exécution de nombreux ordres SQL consécutifs. L'impact de la latence due au trafic réseau de la base au client est souvent sous-estimée.

Les langages PL permettent aussi d'accéder à leurs bibliothèques spécifiques (extrêmement nombreuses en python ou perl, entre autres).

Une fonction en PL peut également servir à l'indexation des données. Cela est impossible si elle se calcule sur une autre machine.

Simplicité :

Suivant le besoin, un langage PL peut être bien plus pratique que le langage client.

Il est par exemple très simple d'écrire un traitement d'insertion/mise à jour en PL/pgSQL, le langage étant créé pour simplifier ce genre de traitements, et la gestion des exceptions pouvant s'y produire. Si vous avez besoin de réaliser du traitement de chaîne puissant, ou de la manipulation de fichiers, PL/Perl ou PL/Python seront probablement des options plus intéressantes car plus performantes, là aussi utilisables dans la base.

La grande variété des différents langages PL supportés par PostgreSQL permet normalement d'en trouver un correspondant aux besoins et aux langages déjà maîtrisés dans l'entreprise.

Les langages PL permettent donc de rajouter une couche d'abstraction et d'effectuer des traitements avancés directement en base.

8.2.5 Intérêts de PL/pgSQL en particulier

- Inspiré de l'ADA, proche du Pascal
- Ajout de structures de contrôle au langage SQL
- **Dédié au traitement des données et au SQL**
- Peut effectuer des traitements complexes
- Hérite de tous les types, fonctions et opérateurs définis par les utilisateurs
- *Trusted*
- Facile à utiliser

Le langage étant assez ancien, proche du Pascal et de l'ADA, sa syntaxe ne choquera personne. Elle est d'ailleurs très proche de celle du PLSQL d'Oracle.

Le PL/pgSQL permet d'écrire des requêtes directement dans le code PL sans déclaration préalable, sans appel à des méthodes complexes, ni rien de cette sorte. Le code SQL est mélangé naturellement au code PL, et on a donc un sur-ensemble procédural de SQL.

PL/pgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux rajoutés par l'utilisateur. Il peut les manipuler de façon transparente.

PL/pgSQL est *trusted*. Tous les utilisateurs peuvent donc créer des routines dans ce langage (par défaut). Vous pouvez toujours soit supprimer le langage, soit retirer les droits à un utilisateur sur ce langage (via la commande SQL `REVOKE`).

PL/pgSQL est donc raisonnablement facile à utiliser : il y a peu de complications, peu de pièges, et il dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

8.2.6 Les autres langages PL ont toujours leur intérêt

- Avantages des autres langages PL par rapport à PL/pgSQL :
  - beaucoup plus de possibilités
  - souvent plus performants pour la résolution de certains problèmes
- Mais :
  - pas spécialisés dans le traitement de requêtes
  - types différents
  - interpréteur séparé

Les langages PL « autres », comme PL/perl[10] et PL/Python (les deux plus utilisés après PL/pgSQL), sont bien plus évolués que PL/PgSQL. Par exemple, ils sont bien plus efficaces en matière de traitement de chaînes de caractères, possèdent des structures avancées comme des tables de hachage, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leur version *untrusted*, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un service web par exemple, ou d'écrire des données dans un fichier externe.

Il existe des langages PL spécialisés. Le plus emblématique d'entre eux est PL/R[11]. R est un langage utilisé par les statisticiens pour manipuler de gros jeux de données. PL/R permet donc d'effectuer ces traitements R directement en base, traitements qui seraient très pénibles à écrire dans d'autres langages, et avec une latence dans le transfert des données.

Il existe aussi un langage qui est, du moins sur le papier, plus rapide que tous les langages cités précédemment : vous pouvez écrire des procédures stockées en C[12], directement. Elles seront compilées à l'extérieur de PostgreSQL, en respectant un certain formalisme, puis seront chargées en indiquant la bibliothèque C qui les contient et leurs paramètres et types de retour.

Mais attention : toute erreur dans le code C est susceptible d'accéder à toute la mémoire visible par le processus PostgreSQL qui l'exécute, et donc de corrompre les données. Il est donc conseillé de ne faire ceci qu'en dernière extrémité.

Le gros défaut est simple et commun à tous ces langages : ils ne sont pas spécialement conçus pour s'exécuter en tant que langage de procédures stockées. Ce que vous utilisez quand vous écrivez du PL/Perl est donc du code Perl, avec quelques fonctions supplémentaires (préfixées par `spi`) pour accéder à la base de données ; de même en C. L'accès aux données est assez fastidieux au niveau syntaxique, comparé à PL/pgSQL.

Un autre problème des langages PL (autre que C et PL/pgSQL), est que ces langages n'ont pas les mêmes types natifs que PostgreSQL, et s'exécutent dans un interpréteur relativement séparé. Les performances sont donc moindres que PL/pgSQL et C, pour les traitements dont le plus consommateur est l'accès aux données. Souvent, le temps de traitement dans un de ces langages plus évolués est tout de même meilleur grâce au temps gagné par les autres fonctionnalités (la possibilité d'utiliser un cache, ou une table de hachage par exemple).

[10] https://docs.postgresql.fr/current/plperl-builtins.html
[11] https://github.com/postgres-plr/plr/blob/master/userguide.md
[12] https://docs.postgresql.fr/current/xfunc-c.html

8.2.7 Routines / Procédures stockées / Fonctions

- **Procédure** stockée
  - pas de retour
  - contrôle transactionnel : `COMMIT` / `ROLLBACK`
- **Fonction**
  - peut renvoyer des données (même des lignes)
  - utilisable dans un `SELECT`
  - peut être de type `TRIGGER`, agrégat, fenêtrage
- **Routine**
  - procédure ou fonction

Les programmes écrits à l'aide des langages PL sont habituellement enregistrés sous forme de « routines » :

- procédures ;
- fonctions ;
- fonctions *trigger* ;
- fonctions d'agrégat ;
- fonctions de fenêtrage (*window functions*).

Le code source de ces objets est stocké dans la table `pg_proc` du catalogue.

Les procédures, apparues avec PostgreSQL 11, sont très similaires aux fonctions. Les principales différences entre les deux sont :

- Les fonctions doivent déclarer des arguments en sortie (`RETURNS` ou arguments `OUT`). Elles peuvent renvoyer n'importe quel type de donnée, ou des ensembles de lignes. Il est possible d'utiliser `void` pour une fonction sans argument de sortie ; c'était d'ailleurs la méthode utilisée pour émuler le comportement d'une procédure avant leur introduction avec PostgreSQL 11. Les procédures n'ont pas de code retour (on peut cependant utiliser des paramètres `OUT` ou `INOUT`).
- Les procédures offrent le support du contrôle transactionnel, c'est-à-dire la capacité de valider (`COMMIT`) ou annuler (`ROLLBACK`) les modifications effectuées jusqu'à ce point par la procédure. L'intégralité d'une fonction s'effectue dans la transaction appelante.
- Les procédures sont appelées exclusivement par la commande SQL `CALL` ; les fonctions peuvent être appelées dans la plupart des ordres DML/DQL (notamment `SELECT`), mais pas par `CALL`.
- Les fonctions peuvent être déclarées de telle manière qu'elles peuvent être utilisées dans des rôles spécifiques (*trigger*, agrégat ou fonction de fenêtrage).

8.3 INSTALLATION

8.3.1 Installation des binaires nécessaires

- SQL, C et PL/pgSQL
  - compilés et installés par défaut
- Paquets du PGDG pour la plupart des langages :

```
yum|dnf install postgresql16-plperl
apt     install postgresql-plpython3-16
```

- Autres langages :
  - à compiler soi-même

Pour savoir si PL/Perl ou PL/Python a été compilé, on peut demander à `pg_config` :

```
pg_config --configure
'--prefix=/usr/local/pgsql-10_icu' '--enable-thread-safety'
'--with-openssl' '--with-libxml' '--enable-nls' '--with-perl' '--enable-debug'
'ICU_CFLAGS=-I/usr/local/include/unicode/'
'ICU_LIBS=-L/usr/local/lib -licui18n -licuuc -licudata' '--with-icu'
```

Si besoin, les emplacements exacts d'installation des bibliothèques peuvent être récupérés à l'aide des options `--libdir` et `--pkglibdir` de `pg_config`.

Cependant, dans les paquets fournis par le PGDG, il faudra installer explicitement le paquet dédié à `plperl` pour la version majeure de PostgreSQL concernée. Pour PostgreSQL 16, les paquets sont `postgresql16-plperl` (depuis yum.postgresql.org) ou `postgresql-plperl-16` (depuis apt.postgresql.org). De même pour Python 3 (paquets `postgresql14-plpython3` ou `postgresql-plython3-14`).

Les bibliothèques `plperl.so`, `plpython3.so` ou `plpgsql.so` contiennent les fonctions qui permettent l'utilisation de chaque langage. La bibliothèque nécessaire est chargée par le moteur à la première utilisation d'une procédure utilisant ce langage.

La plupart des langages intéressants sont disponibles sous forme de paquets. Des versions très récentes, ou des langages plus exotiques, peuvent nécessiter une compilation de l'extension.

8.3.2 Activer un langage

Activer un langage passe par la création de l'extension :

```
CREATE EXTENSION plperl ;      -- pour tous
-- versions untrusted
CREATE EXTENSION plperlu ;     -- pour le superutilisateur
CREATE EXTENSION plpython3u ;
```

- Liste : `\dL` ou `pg_language`

Le langage est activé uniquement dans la base dans laquelle la commande est lancée. Il faudra donc répéter le `CREATE EXTENSION` dans chaque base au besoin (noter qu'activer un langage dans la base modèle `template1` l'activera aussi pour toutes les bases créées par la suite, comme c'est déjà le cas pour le PL/pgSQL).

Pour voir les langages activés, utiliser la commande `\dL` qui reprend le contenu de la table système `pg_language` :

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
CREATE EXTENSION plsh ;
CREATE EXTENSION plr;
```

```
postgres=# \dL
                        Liste des langages
    Nom     | … | De confiance |                Description
------------+---+--------------+-------------------------------------------
 plperl     | … | t            | PL/PerlU untrusted procedural language
 plpgsql    | … | t            | PL/pgSQL procedural language
 plpython3u | … | f            | PL/Python3U untrusted procedural language
 plr        | … | f            |
 plsh       | … | f            | PL/sh procedural language
```

Noter la distinction entre les langages *trusted* (de confiance) et *untrusted*. Si un langage est *trusted*, tous les utilisateurs peuvent créer des procédures dans ce langage sans danger. Sinon seuls les superutilisateurs le peuvent.

Il existe par exemple deux variantes de PL/Perl : PL/Perl et PL/PerlU. La seconde est la variante *untrusted* et est un Perl « complet ». La version *trusted* n'a pas le droit d'ouvrir des fichiers, des sockets, ou autres appels systèmes qui seraient dangereux.

SQL, PL/pgSQL, PL/Tcl, PL/Perl (mais pas PL/Python) sont *trusted* et les utilisateurs peuvent les utiliser à volonté.

C, PL/TclU, PL/PerlU, et PL/Python3U sont *untrusted*. Un superutilisateur doit alors écrire les fonctions et procédures et opérer des `GRANT EXECUTE` aux utilisateurs.

8.4 EXEMPLES DE FONCTIONS & PROCÉDURES

8.4.1 Fonction PL/pgSQL simple

Une fonction simple en PL/pgSQL :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS '
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2;
  RETURN resultat;
END ' ;
```

```
SELECT addition (1,2);

 addition
----------
        3
```

8.4.2 Exemple de fonction SQL

Même fonction en SQL pur :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE sql
IMMUTABLE
AS '     SELECT entier1 + entier2 ; ' ;
```

- Intérêt : *inlining* & planification
- Syntaxe allégée (v14+) :

```
CREATE OR REPLACE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE sql
IMMUTABLE
RETURN entier1 + entier2 ;
```

Les fonctions simples peuvent être écrites en SQL pur. La syntaxe est plus claire, mais bien plus limitée qu'en PL/pgSQL (ni boucles, ni conditions, ni exceptions notamment).

À partir de PostgreSQL 14, il est possible de se passer des guillemets encadrants, pour les fonctions SQL uniquement. La même fonction devient donc :

```
CREATE OR REPLACE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE sql
IMMUTABLE
RETURN entier1 + entier2 ;
```

Cette nouvelle écriture respecte mieux le standard SQL. Surtout, elle autorise un *parsing* et une vérification des objets impliqués dès la déclaration, et non à l'utilisation. Les dépendances entre fonctions et objets utilisés sont aussi mieux tracées.

L'avantage principal des fonctions en pur SQL est, si elles sont assez simples, leur intégration lors de la réécriture interne de la requête (*inlining*) : elles ne sont donc pas pour l'optimiseur des « boîtes noires ». À l'inverse, l'optimiseur ne sait rien du contenu d'une fonction PL/pgSQL.

Dans l'exemple suivant, la fonction sert de filtre à la requête. Comme elle est en pur SQL, elle permet d'utiliser l'index sur la colonne `date_embauche` de la table `employes_big` :

```
CREATE FUNCTION employe_eligible_prime_sql (service int, date_embauche date)
RETURNS boolean
LANGUAGE sql
AS $$
  SELECT ( service !=3 AND date_embauche < '2003-01-01')  ;
$$ ;

EXPLAIN (ANALYZE) SELECT matricule, num_service, nom, prenom
FROM    employes_big
WHERE   employe_eligible_prime_sql (num_service, date_embauche) ;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
       (cost=0.42..1.54 rows=1 width=22) (actual time=0.008..0.009 rows=1 loops=1)
   Index Cond: (date_embauche < '2003-01-01'::date)
   Filter: (num_service <> 3)
   Rows Removed by Filter: 1
 Planning Time: 0.102 ms
 Execution Time: 0.029 ms
```

Avec une version de la même fonction en PL/pgSQL, le planificateur ne voit pas le critère indexé. Il n'a pas d'autre choix que de lire toute la table et d'appeler la fonction pour chaque ligne, ce qui est bien sûr plus lent :

```
CREATE FUNCTION employe_eligible_prime_pl (service int, date_embauche date)
RETURNS boolean
LANGUAGE plpgsql AS $$
BEGIN
 RETURN ( service !=3 AND date_embauche < '2003-01-01') ;
END ;
$$ ;

EXPLAIN (ANALYZE) SELECT matricule, num_service, nom, prenom
FROM employes_big
WHERE  employe_eligible_prime_pl (num_service, date_embauche) ;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on employes_big   (cost=0.00..134407.90 rows=166338 width=22)
                            (actual time=0.069..269.121 rows=1 loops=1)
   Filter: employe_eligible_prime_pl(num_service, date_embauche)
   Rows Removed by Filter: 499014
 Planning Time: 0.038 ms
 Execution Time: 269.157 ms
```

Le wiki[13] décrit les conditions pour que l'*inlining* des fonctions SQL fonctionne : obligation d'un seul `SELECT`, interdiction de certains fonctionnalités…

8.4.3 Exemple de fonction PL/pgSQL utilisant la base

```
CREATE OR REPLACE FUNCTION nb_lignes_table (sch text, tbl text)
 RETURNS bigint
 STABLE
AS '
DECLARE     n bigint ;
BEGIN
    SELECT n_live_tup
    INTO n
    FROM pg_stat_user_tables
    WHERE schemaname = sch AND relname = tbl ;
    RETURN n ;
END ; '
LANGUAGE plpgsql ;
```

Dans cet exemple, on récupère l'estimation du nombre de lignes actives d'une table passée en paramètres.

L'intérêt majeur du PL/pgSQL et du SQL sur les autres langages est la facilité d'accès aux données. Ici, un simple `SELECT <champ> INTO <variable>` suffit à récupérer une valeur depuis une table dans une variable.

```
SELECT nb_lignes_table ('public', 'pgbench_accounts');
```

```
 nb_lignes_table
-----------------
        10000000
```

[13] https://wiki.postgresql.org/wiki/Inlining_of_SQL_functions

8.4.4 Exemple de fonction PL/Perl complexe

- Permet d'insérer une facture associée à un client
- Si le client n'existe pas, une entrée est créée
- Utilisation fréquente de `spi_exec`

Voici l'exemple de la fonction :

```
CREATE OR REPLACE FUNCTION
    public.demo_insert_perl(nom_client text, titre_facture text)
 RETURNS integer
 LANGUAGE plperl
 STRICT
AS $function$
  use strict;
  my ($nom_client, $titre_facture)=@_;
  my $rv;
  my $id_facture;
  my $id_client;

  # Le client existe t'il ?
  $rv = spi_exec_query('SELECT id_client FROM mes_clients WHERE nom_client = '
    . quote_literal($nom_client)
  );
  # Sinon on le crée :
  if ($rv->{processed} == 0)
  {
    $rv = spi_exec_query('INSERT INTO mes_clients (nom_client) VALUES ('
        . quote_literal($nom_client) . ') RETURNING id_client'
    );
  }
  # Dans les deux cas, l'id client est dans $rv :
  $id_client=$rv->{rows}[0]->{id_client};

  # Insérons maintenant la facture
  $rv = spi_exec_query(
    'INSERT INTO mes_factures (titre_facture, id_client) VALUES ('
    . quote_literal($titre_facture) . ", $id_client ) RETURNING id_facture"
  );

  $id_facture = $rv->{rows}[0]->{id_facture};

  return $id_facture;
$function$ ;
```

Cette fonction n'est pas parfaite, elle ne protège pas de tout. Il est tout à fait possible d'avoir une insertion concurrente entre le `SELECT` et le `INSERT` par exemple.

Il est clair que l'accès aux données est malaisé en PL/Perl, comme dans la plupart des langages, puisqu'ils ne sont pas prévus spécifiquement pour cette tâche. Par contre, on dispose de toute la

puissance de Perl pour les traitements de chaîne, les appels système…

PL/Perl, c'est :

- Perl, moins les fonctions pouvant accéder à autre chose qu'à PostgreSQL (il faut utiliser PL/PerlU pour passer outre cette limitation) ;
- un bloc de code anonyme appelé par PostgreSQL ;
- des fonctions d'accès à la base, `spi_*`

8.4.5 Exemple de fonction PL/pgSQL complexe

- Même fonction en PL/pgSQL que précédemment
- L'accès aux données est simple et naturel
- Les types de données SQL sont natifs
- La capacité de traitement est limitée par le langage
- **Attention** au nommage des variables et paramètres

Pour éviter les conflits avec les objets de la base, il est conseillé de préfixer les variables.

```
CREATE OR REPLACE FUNCTION
public.demo_insert_plpgsql(p_nom_client text, p_titre_facture text)
 RETURNS integer
 LANGUAGE plpgsql
 STRICT
AS $function$
DECLARE
  v_id_facture int;
  v_id_client int;
BEGIN
  -- Le client existe t'il ?
  SELECT id_client
  INTO v_id_client
  FROM mes_clients
  WHERE nom_client = p_nom_client;

  -- Sinon on le crée :
  IF NOT FOUND THEN
    INSERT INTO mes_clients (nom_client)
    VALUES (p_nom_client)
    RETURNING id_client INTO v_id_client;
  END IF;

  -- Dans les deux cas, l'id client est maintenant dans v_id_client

  -- Insérons maintenant la facture
  INSERT INTO mes_factures (titre_facture, id_client)
  VALUES (p_titre_facture, v_id_client)
  RETURNING id_facture INTO v_id_facture;
```

```
  return v_id_facture;
END;
$function$ ;
```

8.4.6 Exemple de procédure

```
CREATE OR REPLACE PROCEDURE vide_tables (dry_run BOOLEAN)
AS '
BEGIN
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
    IF dry_run THEN
        ROLLBACK ;
    END IF ;
END ;
 ' LANGUAGE plpgsql ;
```

Cette procédure tronque des tables de la base d'exemple **pgbench**, et annule si `dry_run` est vrai.

Les procédures sont récentes dans PostgreSQL (à partir de la version 11). Elles sont à utiliser quand on n'attend pas de résultat en retour. Surtout, elles permettent de gérer les transactions (`COMMIT`, `ROLLBACK`), ce qui ne peut se faire dans des fonctions, même si celles-ci peuvent modifier les données.

Une procédure ne peut utiliser le contrôle transactionnel que si elle est appelée en dehors de toute transaction.

Comme pour les fonctions, il est possible d'utiliser le SQL pur dans les cas les plus simples, sans contrôle transactionnel notamment :

```
CREATE OR REPLACE PROCEDURE vide_tables ()
AS '
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
' LANGUAGE sql;
```

Toujours pour les procédures en SQL, il existe une variante sans guillemets, à partir de PostgreSQL 14, mais qui ne supporte pas tous les ordres. Comme pour les fonctions, l'intérêt est la prise en compte des dépendances entre objets et procédures.

```
CREATE OR REPLACE PROCEDURE vide_tables ()
BEGIN ATOMIC
    DELETE FROM pgbench_history ;
    DELETE FROM pgbench_accounts ;
    DELETE FROM pgbench_tellers ;
    DELETE FROM pgbench_branches ;
END ;
```

8.4.7 Exemple de bloc anonyme en PL/pgSQL

- Bloc procédural anonyme en PL/pgSQL :

```
DO $$
DECLARE r record;
BEGIN
    FOR r IN (SELECT schemaname, relname
              FROM pg_stat_user_tables
              WHERE coalesce(last_analyze, last_autoanalyze) IS NULL
              ) LOOP
        RAISE NOTICE 'Analyze %.%', r.schemaname, r.relname ;
        EXECUTE 'ANALYZE ' || quote_ident(r.schemaname)
                           || '.' || quote_ident(r.relname) ;
    END LOOP;
END$$;
```

Les blocs anonymes sont utiles pour des petits scripts ponctuels qui nécessitent des boucles ou du conditionnel, voire du transactionnel, sans avoir à créer une fonction ou une procédure. Ils ne renvoient rien. Ils sont habituellement en PL/pgSQL mais tout langage procédural installé est possible.

L'exemple ci-dessus lance un `ANALYZE` sur toutes les tables où les statistiques n'ont pas été calculées d'après la vue système, et donne aussi un exemple de SQL dynamique. Le résultat est par exemple :

```
NOTICE:  Analyze public.pgbench_history
NOTICE:  Analyze public.pgbench_tellers
NOTICE:  Analyze public.pgbench_accounts
NOTICE:  Analyze public.pgbench_branches
DO
Temps : 141,208 ms
```

(Pour ce genre de SQL dynamique, si l'on est sous `psql` , il est souvent plus pratique d'utiliser `\gexec` [14].)

Noter que les ordres constituent une transaction unique, à moins de rajouter des `COMMIT` ou `ROLLBACK` explicitement (ce n'est autorisé qu'à partir de la version 11).

[14] https://docs.postgresql.fr/current/app-psql.html#R2-APP-PSQL-4

8.5 UTILISER UNE FONCTION OU UNE PROCÉDURE

8.5.1 Invocation d'une fonction ou procédure

- Appeler une procédure : ordre spécifique `CALL`

```
CALL ma_procedure('arg1');
```

- Appeler une fonction : dans une requête

```
SELECT ma_fonction('arg1', 'arg2') ;

SELECT * FROM ma_fonction('arg1', 'arg2') ;

INSERT INTO matable
SELECT ma_fonction( champ1, champ2 )   FROM ma_table2 ;

CALL ma_procedure( mafonction() );

CREATE INDEX ON ma_table ( ma_fonction(ma_colonne) );
```

Demander l'exécution d'une procédure se fait en utilisant un ordre SQL spécifique : `CALL`[15]. Il suffit de fournir les paramètres. Il n'y a pas de code retour.

Les fonctions ne sont quant à elles pas directement compatibles avec la commande `CALL`, il faut les invoquer dans le contexte d'une commande SQL. Elles sont le plus couramment appelées depuis des commandes de type DML (`SELECT`, `INSERT`, etc.), mais on peut aussi les trouver dans d'autres commandes.

Voici quelques exemples :

- dans un `SELECT` (la fonction ne doit renvoyer qu'une seule ligne) :

```
SELECT ma_fonction('arg1', 'arg2');
```

- dans un `SELECT`, en passant en argument les valeurs d'une colonne d'une table :

```
SELECT ma_fonction(ma_colonne) FROM ma_table;
```

- dans le `FROM` d'un `SELECT`, la fonction renvoit ici généralement plusieurs lignes (`SETOF`), et un résultat de type `RECORD` :

```
SELECT result FROM ma_fonction() AS f(result);
```

- dans un `INSERT` pour générer la valeur à insérer :

[15] https://docs.postgresql.fr/current/sql-call.html

```
INSERT INTO ma_table(ma_colonne) VALUES ( ma_fonction() );
```

- dans une création d'index (index fonctionnel, la fonction sera réellement appelée lors des mises à jour de l'index… attention la fonction doit être déclarée « immutable ») :

```
CREATE INDEX ON ma_table ( ma_fonction(ma_colonne) );
```

- appel d'une fonction en paramètre d'une autre fonction ou d'une procédure, par exemple ici le résultat de la fonction `ma_fonction()` (qui doit renvoyer une seule ligne) est passé en argument d'entrée de la procédure `ma_procedure()` :

```
CALL ma_procedure( ma_fonction() );
```

Par ailleurs, certaines fonctions sont spécialisées et ne peuvent être invoquées que dans le contexte pour lequel elles ont été conçues (fonctions trigger, d'agrégat, de fenêtrage, etc.).

8.6 CONTRÔLE TRANSACTIONNEL DANS LES PROCÉDURES

- `COMMIT` et `ROLLBACK` : possibles dans les procédures
- Pas de `BEGIN`
  - automatique après la fin d'une transaction dans le code
- Un seul niveau de transaction
  - pas de sous-transactions
  - pas d'appel depuis une transaction
- Incompatible avec une clause `EXCEPTION`

Une procédure peut contenir des ordres `COMMIT` ou `ROLLBACK` pour du contrôle transactionnel. (À l'inverse une fonction est une transaction unique, ou opère dans une transaction.)

Voici un exemple validant ou annulant une insertion suivant que le nombre est pair ou impair :

```
CREATE TABLE test1 (a int) ;

CREATE OR REPLACE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
  FOR i IN 0..5 LOOP
    INSERT INTO test1 (a) VALUES (i);
    IF i % 2 = 0 THEN
      COMMIT;
    ELSE
      ROLLBACK;
    END IF;
  END LOOP;
END
$$;

CALL transaction_test1();

SELECT * FROM test1;
 a | b
---+---
 0 |
 2 |
 4 |
```

Une exemple plus fréquemment utilisé est celui d'une procédure effectuant un traitement de modification des données par lots, et donc faisant un `COMMIT` à intervalle régulier.

Noter qu'il n'y a pas de `BEGIN` explicite dans la gestion des transactions. Après un `COMMIT` ou un `ROLLBACK`, un `BEGIN` est immédiatement exécuté.

On ne peut pas imbriquer des transactions, car PostgreSQL ne connaît pas les sous-transactions :

```
BEGIN ; CALL transaction_test1() ;

ERROR:  invalid transaction termination
CONTEXTE : PL/pgSQL function transaction_test1() line 6 at COMMIT
```

On ne peut pas utiliser en même temps une clause `EXCEPTION` et le contrôle transactionnel :

```
DO LANGUAGE plpgsql $$
BEGIN
  BEGIN
    INSERT INTO test1 (a) VALUES (1);
  COMMIT;
  INSERT INTO test1 (a) VALUES (1/0);
COMMIT;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'caught division_by_zero';
  END;
END;
$$;

ERREUR:  cannot commit while a subtransaction is active
CONTEXTE : fonction PL/pgSQL inline_code_block, ligne 5 à COMMIT
```

8.7 CRÉATION ET MAINTENANCE DES FONCTIONS ET PROCÉDURES

8.7.1 Création

- `CREATE FUNCTION`
- `CREATE PROCEDURE`

Voici la syntaxe complète pour une fonction d'après la documentation[16] :

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, …] ] )
     [ RETURNS rettype
       | RETURNS TABLE ( column_name column_type [, …] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, … ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } …
```

Voici la syntaxe complète pour une procédure d'après la documentation[17] :

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, …] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, … ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } …
```

Noter qu'il n'y a pas de langage par défaut. Il est donc nécessaire de le spécifier à chaque création d'une routine comme dans les exemples ci-dessous.

[16] https://www.postgresql.org/docs/current/sql-createfunction.html
[17] https://www.postgresql.org/docs/current/sql-createprocedure.html

8.7.2 Structure d'une routine PL/pgSQL

- Reprenons le code montré plus haut :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS '
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2 ;
  RETURN resultat ;
END';
```

Le langage PL/pgSQL n'est pas sensible à la casse, tout comme SQL (sauf les noms des objets ou variables, si vous les mettez entre des guillemets doubles). L'opérateur de comparaison est `=`, l'opérateur d'affectation `:=`

8.7.3 Structure d'une routine PL/pgSQL (suite)

- `DECLARE`
  - déclaration des variables locales
- `BEGIN`
  - début du code de la routine
- `END`
  - la fin
- Instructions séparées par des points-virgules
- Commentaires commençant par `--` ou compris entre `/*` et `*/`

Une routine est composée d'un bloc de déclaration des variables locales et d'un bloc de code. Le bloc de déclaration commence par le mot clé `DECLARE` et se termine avec le mot clé `BEGIN`. Ce mot clé est celui qui débute le bloc de code. La fin est indiquée par le mot clé `END`.

Toutes les instructions se terminent avec des points-virgules. Attention, `DECLARE`, `BEGIN` et `END` ne sont pas des instructions.

Il est possible d'ajouter des commentaires. `--` indique le début d'un commentaire qui se terminera en fin de ligne. Pour être plus précis dans la délimitation, il est aussi possible d'utiliser la notation C : `/*` est le début d'un commentaire et `*/` la fin.

8.7.4 Blocs nommés

- Labels de bloc possibles
- Plusieurs blocs d'exception possibles dans une routine
- Permet de préfixer des variables avec le label du bloc
- De donner un label à une boucle itérative
- Et de préciser de quelle boucle on veut sortir, quand plusieurs d'entre elles sont imbriquées

Indiquer le nom d'un label ainsi :

```
<<mon_label>>
-- le code (blocs DECLARE, BEGIN-END, et EXCEPTION)
```

ou bien (pour une boucle)

```
[ <<mon_label>> ]
LOOP
    ordres …
END LOOP [ mon_label ];
```

Bien sûr, il est aussi possible d'utiliser des labels pour des boucles `FOR`, `WHILE`, `FOREACH`.

On sort d'un bloc ou d'une boucle avec la commande `EXIT`, on peut aussi utiliser `CONTINUE` pour passer à l'exécution suivante d'une boucle sans terminer l'itération courante.

Par exemple :

```
EXIT [mon_label] WHEN compteur > 1;
```

8.7.5 Modification du code d'une routine

- `CREATE OR REPLACE FUNCTION`
- `CREATE OR REPLACE PROCEDURE`
- Une routine est définie par son nom et ses arguments
- Si type de retour différent, la fonction doit d'abord être supprimée puis recréée

Une routine est surchargeable. La seule façon de les différencier est de prendre en compte les arguments (nombre et type). Les noms des arguments peuvent être indiqués mais ils seront ignorés.

Deux routines identiques aux arguments près (on parle de prototype) ne sont pas identiques, mais bien deux routines distinctes.

`CREATE OR REPLACE` a principalement pour but de modifier le code d'une routine, mais il est aussi possible de modifier les méta-données.

8.7.6 Modification des méta-données d'une routine

- `ALTER FUNCTION` / `ALTER PROCEDURE`
- Une routine est définie par son nom et ses arguments
- Permet de modifier nom, propriétaire, schéma et autres options

Toutes les méta-données discutées plus haut sont modifiables avec un `ALTER`.

8.7.7 Suppression d'une routine

- Une routine est définie par son nom et ses arguments :

```
DROP FUNCTION addition (integer, integer) ;

DROP PROCEDURE public.vide_tables (boolean);
DROP PROCEDURE public.vide_tables ();
```

La suppression se fait avec l'ordre `DROP`.

Une fonction pouvant exister en plusieurs exemplaires, avec le même nom et des arguments de type différents, il faudra parfois parfois préciser ces derniers.

8.7.8 Utilisation des guillemets

- Les guillemets deviennent très rapidement pénibles
  - préférer `$$`
  - ou `$fonction$`, `$toto$`…

Définir une fonction entre guillemets simples (`'`) devient très pénible dès que la fonction doit en contenir parce qu'elle contient elle-même des chaînes de caractères. PostgreSQL permet de remplacer les guillemets par `$$`, ou tout mot encadré de `$`.

Par exemple, on peut reprendre la syntaxe de déclaration de la fonction `addition()` précédente en utilisant cette méthode :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS $ma_fonction_addition$
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2;
  RETURN resultat;
END
$ma_fonction_addition$;
```

Ce peut être utile aussi dans tout code réalisant une concaténation de chaînes de caractères contenant des guillemets. La syntaxe traditionnelle impose de les multiplier pour les protéger, et le code devient difficile à lire. :

```
requete := requete || '' AND vin LIKE ''''bordeaux%'''' AND xyz ''
```

En voilà une simplification grâce aux dollars :

```
requete := requete || $sql$ AND vin LIKE 'bordeaux%' AND xyz $sql$
```

Si vous avez besoin de mettre entre guillemets du texte qui inclut `$$`, vous pouvez utiliser `$Q$`, et ainsi de suite. Le plus simple étant de définir un marqueur de fin de routine plus complexe, par exemple incluant le nom de la fonction.

8.8 PARAMÈTRES ET RETOUR DES FONCTIONS ET PROCÉDURES

8.8.1 Version minimaliste

```
CREATE FUNCTION fonction (entier integer, texte text)
RETURNS int  AS …
```

Ceci une forme de fonction très simple (et très courante) : deux paramètres en entrée (implicitement en entrée seulement), et une valeur en retour.

Dans le corps de la fonction, il est aussi possible d'utiliser une notation numérotée au lieu des noms de paramètre : le premier argument a pour nom `$1`, le deuxième `$2`, etc. C'est à éviter.

Tous les types sont utilisables, y compris les types définis par l'utilisateur. En dehors des types natifs de PostgreSQL, PL/pgSQL ajoute des types de paramètres spécifiques pour faciliter l'écriture des routines.

8.8.2 Paramètres IN, OUT, INOUT & retour

```
CREATE FUNCTION cree_utilisateur (
  nom text,                  -- IN
  type_id int DEFAULT 0      -- IN
) RETURNS id_utilisateur int  AS …

CREATE FUNCTION explose_date (
  IN  d date,
  OUT jour int, OUT mois int, OUT annee int
 ) AS …
```

- `VARIADIC` : nombre variable

Si le mode d'un argument est omis, `IN` est la valeur implicite : la valeur en entrée ne sera pas modifiée par la fonction.

Un paramètre `OUT` sera modifié. S'il s'agit d'une variable d'un bloc PL appelant, sa valeur sera modifiée. Un paramètre `INOUT` est un paramètre en entrée qui peut être également modifié. (Jusque PostgreSQL 13 inclus, les procédures ne supportent pas les arguments `OUT`, seulement `IN` et `INOUT`.)

Dans le corps d'une fonction, `RETURN` est inutile avec des paramètres `OUT` parce que c'est la valeur des paramètres `OUT` à la fin de la fonction qui est retournée, comme dans l'exemple plus bas.

L'option `VARIADIC` permet de définir une fonction avec un nombre d'arguments libres à condition de respecter le type de l'argument (comme `printf` en C par exemple). Seul un argument `OUT` peut suivre un argument `VARIADIC` : l'argument `VARIADIC` doit être le dernier de la liste des paramètres en entrée puisque tous les paramètres en entrée suivant seront considérées comme faisant partie du tableau variadic. Seuls les arguments `IN` et `VARIADIC` sont utilisables avec une fonction déclarée comme renvoyant une table (clause `RETURNS TABLE`, voir plus loin).

La clause `DEFAULT` permet de rendre les paramètres optionnels. Après le premier paramètre ayant une valeur par défaut, tous les paramètres qui suivent doivent aussi avoir une valeur par défaut. Pour rendre le paramètre optionnel, il doit être le dernier argument ou alors les paramètres suivants doivent aussi avoir une valeur par défaut.

8.8.3 Type en retour : 1 valeur simple

- Fonctions uniquement

```
RETURNS type      -- int, text, etc
```

- Tous les types de base & utilisateur
- Rien : `void`

Le type de retour (clause `RETURNS` dans l'entête) est obligatoire pour les fonctions et interdit pour les procédures.

Avant la version 11, il n'était pas possible de créer une procédure, mais il était possible de créer une fonction se comportant globalement comme une procédure en utilisant le type de retour `void`.

Des exemples plus haut utilisent des types simples, mais tous ceux de PostgreSQL ou les types créés par l'utilisateur sont utilisables.

Depuis le corps de la fonction, le résultat est renvoyé par un appel à `RETURN` (PL/pgSQL) ou `SELECT` (SQL).

8.8.4 Type en retour : 1 ligne, plusieurs champs (exemple)

Comment obtenir ceci ?

```
SELECT * FROM explose_date ('31-12-2020');

 jour | mois | annee
------+------+-------
   31 |    0 |  2020
```

8.8.5 Type en retour : 1 ligne, plusieurs champs

3 options :

- Type composé dédié

```
CREATE TYPE ma_structure AS ( … ) ;
CREATE FUNCTION …  RETURNS ma_structure ;
```

- Paramètres `OUT`

```
CREATE FUNCTION explose_date (IN d date,
                              OUT jour int, OUT mois int, OUT annee int) AS …
```

- `RETURNS TABLE`

```
CREATE FUNCTION explose_date_table (d date)
RETURNS TABLE  (jour integer, mois integer, annee integer) AS…
```

S'il y a besoin de renvoyer plusieurs valeurs à la fois, une première possibilité est de renvoyer un type composé défini auparavant.

Une alternative très courante est d'utiliser plusieurs paramètres `OUT` (et pas de clause `RETURN` dans l'entête) pour obtenir un enregistrement composite :

```
CREATE OR REPLACE FUNCTION explose_date
             (IN d date, OUT jour int, OUT mois int, OUT annee int)
AS $$
SELECT extract (day FROM d)::int,
               extract(month FROM d)::int, extract (year FROM d)::int
$$
LANGUAGE sql;

SELECT * FROM explose_date ('31-12-2020');

 jour | mois | annee
------+------+-------
   31 |    0 |  2020
```

(Noter que l'exemple ci-dessus est en simple SQL.)

La clause `TABLE` est une autre alternative, sans doute plus claire. Cet exemple devient alors, toujours en pur SQL :

```
CREATE OR REPLACE FUNCTION explose_date_table (d date)
RETURNS TABLE (jour integer, mois integer, annee integer)
LANGUAGE sql
AS $$
  SELECT  extract (day FROM d)::int,
                 extract(month FROM d)::int, extract (year FROM d)::int ;
$$ ;
```

8.8.6 Retour multiligne

- 1 seul champ ou plusieurs ?

```
RETURNS SETOF type   -- int, text, type personnalisé

RETURNS TABLE ( col1 type, col2 type … )
```

- Ligne à ligne ou en bloc ?

```
RETURN NEXT …

RETURN QUERY   SELECT …
RETURN QUERY   EXECUTE …
```

- Le résultat est stocké **puis** envoyé

RETURNS SETOF :

Pour renvoyer plusieurs lignes, la première possibilité est de déclarer un type de retour `SETOF`. Cet exemple utilise `RETURN NEXT` pour renvoyer les lignes une à une :

```
CREATE OR REPLACE FUNCTION liste_entiers_setof (limite int)
 RETURNS SETOF integer
 LANGUAGE plpgsql
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    RETURN NEXT i;
  END LOOP;
END
$$ ;

SELECT * FROM liste_entiers_setof (3) ;

 liste_entiers_setof
---------------------
                   1
                   2
                   3
```

Renvoyer une structure existante :

S'il y a plusieurs champs à renvoyer, une possibilité est d'utiliser un type dédié (composé), qu'il faudra cependant créer auparavant. L'exemple suivant utilise aussi un `RETURN QUERY` pour éviter d'itérer sur toutes les lignes du résultat :

```
CREATE TYPE pgt AS (schemaname text, tablename text) ;

CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
```

```
RETURNS SETOF pgt
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
               FROM pg_tables WHERE tableowner=p_owner
               ORDER BY tablename ;
END $$ ;

SELECT * FROM tables_by_owner ('pgbench');

 schemaname |    tablename
------------+------------------
 public     | pgbench_accounts
 public     | pgbench_branches
 public     | pgbench_history
 public     | pgbench_tellers
```

Si l'on veut renvoyer une structure correspondant exactement à une table ou vue, la syntaxe est très simple (il n'y a même pas besoin de `%ROWTYPE`) :

```
CREATE OR REPLACE FUNCTION  tables_jamais_analyzees ()
RETURNS SETOF pg_stat_user_tables
LANGUAGE sql
AS $$
  SELECT * FROM pg_stat_user_tables
  WHERE coalesce(last_analyze, last_autoanalyze) IS NULL ;
$$ ;

SELECT * FROM tables_jamais_analyzees() \gx

-[ RECORD 1 ]-------+------------------------------
relid               | 414453
schemaname          | public
relname             | table_nouvelle
…
n_mod_since_analyze | 10
n_ins_since_vacuum  | 10
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    |
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
-[ RECORD 2 ]-------+------------------------------
…
```

NB : attention de ne pas oublier le `SETOF`, sinon une seule ligne sera retournée.

RETURNS TABLE :

On a vu que la clause `TABLE` permet de renvoyer plusieurs champs. Or, elle implique aussi `SETOF`, et les deux exemples ci-dessus peuvent devenir :

```
CREATE OR REPLACE FUNCTION liste_entiers_table (limite int)
RETURNS TABLE (j int)
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    j = i ;
    RETURN NEXT ;  -- renvoie la valeur de j en cours
  END LOOP;
END $$ LANGUAGE plpgsql;

SELECT * FROM liste_entiers_table (3) ;
 j
 1
 2
 3
```

(Noter ici que le nom du champ retourné dépend du nom de la variable utilisée, et n'est pas forcément le nom de la fonction. En effet, chaque appel à `RETURN NEXT` retourne un enregistrement composé d'une copie de toutes les variables, au moment de l'appel à `RETURN NEXT`.)

```
DROP FUNCTION tables_by_owner ;
CREATE FUNCTION tables_by_owner (p_owner text)
RETURNS TABLE (schemaname text, tablename text)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT t.schemaname::text, t.tablename::text
               FROM pg_tables t WHERE tableowner=p_owner
               ORDER BY t.tablename ;
END $$ ;
```

Si `RETURNS TABLE` est peut-être le plus souple et le plus clair, le choix entre toutes ces méthodes est affaire de goût, ou de compatibilité avec du code ancien ou converti d'un produit concurrent.

Renvoyer le résultat d'une requête :

Les exemples ci-dessus utilisent `RETURN NEXT` (pour du ligne à ligne) ou `RETURN QUERY` (pour envoyer directement le résultat d'une requête).

La variante `RETURN QUERY EXECUTE …` est destinée à des requêtes en SQL dynamique (voir plus loin).

Quand plusieurs lignes sont renvoyées, tout est conservé en mémoire jusqu'à la fin de la fonction. S'il y en a beaucoup, cela peut poser des problèmes de latence, voire de mémoire. Le paramètre `work_mem` permet de définir la mémoire utilisée avant de basculer sur un fichier temporaire, qui a bien sûr un impact sur les performances.

Appel de fonction :

En général, l'appel se fait ainsi pour obtenir des lignes :

```
SELECT * FROM ma_fonction();
```

Une alternative est d'utiliser :

```
SELECT ma_fonction();
```

pour récupérer un résultat d'une seule colonne, scalaire, type composite ou `RECORD` suivant la fonction.

Cette différence concerne aussi les fonctions système :

```
SELECT * FROM pg_control_system () ;
```

```
 pg_control_version | catalog_version_no |  system_identifier  | pg_control_…
--------------------+--------------------+---------------------+-------------
               1201 |          201909212 | 6744959735975969621 | 2021-09-17 …
(1 ligne)
```

```
SELECT pg_control_system () ;
```

```
                      pg_control_system
-------------------------------------------------------------
 (1201,201909212,6744959735975969621,"2021-09-17 18:24:05+02")
 (1 ligne)
```

8.8.7 Gestion des valeurs NULL

Comment gérer les paramètres à `NULL` ?

- `STRICT` :
  - 1 paramètre `NULL` : retourne `NULL` immédiatement
- Défaut :
  - gestion par la fonction

Si une fonction est définie comme `STRICT` et qu'un des arguments d'entrée est `NULL`, PostgreSQL n'exécute même pas la fonction et utilise `NULL` comme résultat.

Dans la logique relationnelle, `NULL` signifie « la valeur est inconnue ». La plupart du temps, il est logique qu'une fonction ayant un paramètre à une valeur inconnue retourne aussi une valeur inconnue, ce qui fait que cette optimisation est très souvent pertinente.

On gagne à la fois en temps d'exécution, mais aussi en simplicité du code (il n'y a pas à gérer les cas `NULL` pour une fonction dans laquelle `NULL` ne doit jamais être injecté).

Dans la définition d'une fonction, les options sont `STRICT` ou son synonyme `RETURNS NULL ON NULL INPUT`, ou le défaut implicite `CALLED ON NULL INPUT`.

8.9 VARIABLES EN PL/PGSQL

8.9.1 Clause DECLARE

- Dans le source, partie `DECLARE` :

```
DECLARE
  i  integer;
  j  integer := 5;
  k  integer NOT NULL DEFAULT 1;
  ch text    COLLATE "fr_FR";
```

- Blocs `DECLARE` / `BEGIN` / `END` imbriqués possible
  - restriction de scope de variable

En PL/pgSQL, pour utiliser une variable dans le corps de la routine (entre le `BEGIN` et le `END`), il est obligatoire de l'avoir déclarée précédemment :

- soit dans la liste des arguments (`IN` , `INOUT` ou `OUT`) ;
- soit dans la section `DECLARE` .

La déclaration doit impérativement préciser le nom et le type de la variable.

En option, il est également possible de préciser :

- sa valeur initiale (si rien n'est précisé, ce sera `NULL` par défaut) :

  ```
  answer integer := 42;
  ```

- sa valeur par défaut, si on veut autre chose que `NULL` :

  ```
  answer integer DEFAULT 42;
  ```

- une contrainte `NOT NULL` (dans ce cas, il faut impérativement un défaut différent de `NULL` , et toute éventuelle affectation ultérieure de `NULL` à la variable provoquera une erreur) :

  ```
  answer integer NOT NULL DEFAULT 42;
  ```

- le collationnement à utiliser, pour les variables de type chaîne de caractères :

  ```
  question text COLLATE "en_GB";
  ```

Pour les fonctions complexes, avec plusieurs niveaux de boucle par exemple, il est possible d'imbriquer les blocs `DECLARE` / `BEGIN` / `END` en y déclarant des variables locales à ce bloc. Si une variable est par erreur utilisée hors du *scope* prévu, une erreur surviendra.

8.9.2 Constantes

- Clause supplémentaire `CONSTANT` :

```
DECLARE
  eur_to_frf   CONSTANT numeric := 6.55957 ;
  societe_nom  CONSTANT text    := 'Dalibo SARL';
```

L'option `CONSTANT` permet de définir une variable pour laquelle il sera alors impossible d'assigner une valeur dans le reste de la routine.

8.9.3 Types de variables

- Récupérer le type d'une autre variable avec `%TYPE` :

```
quantite    integer ;
total       quantite%TYPE ;
```

- Récupérer le type de la colonne d'une table :

```
quantite    ma_table.ma_colonne%TYPE ;
```

Cela permet d'écrire des routines plus génériques.

8.9.4 Type ROW - 1

- Pour renvoyer plusieurs valeurs à partir d'une fonction
- Utiliser un type composite :

```
CREATE TYPE ma_structure AS (
    un_entier integer,
    une_chaine text,
    …);

CREATE FUNCTION ma_fonction () RETURNS ma_structure …;
```

8.9.5 Type ROW - 2

- Utiliser le type composite défini par la ligne d'une table

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
  ligne ma_table%ROWTYPE;
…
$$
```

L'utilisation de `%ROWTYPE` permet de définir une variable qui contient la structure d'un enregistrement de la table spécifiée. `%ROWTYPE` n'est pas obligatoire, il est néanmoins préférable d'utiliser cette forme, bien plus portable. En effet, dans PostgreSQL, toute création de table crée un type associé de même nom, le seul nom de la table est donc suffisant.

8.9.6 Type RECORD

- `RECORD` identique au type `ROW`
  - …sauf que son type n'est connu que lors de son affectation
- `RECORD` peut changer de type au cours de l'exécution de la routine
- Curseur et boucle sur une requête

`RECORD` est beaucoup utilisé pour manipuler des curseurs, ou dans des boucles `FOR … LOOP` : cela évite de devoir se préoccuper de déclarer un type correspondant exactement aux colonnes de la requête associée à chaque curseur.

8.9.7 Type RECORD : exemple

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
  ligne RECORD;
BEGIN
  -- récupération de la 1è ligne uniquement
  SELECT * INTO ligne FROM ma_première_table;
  -- ou : traitement ligne à ligne
  FOR ligne IN SELECT * FROM ma_deuxième_table  LOOP
    …
  END LOOP ;
  RETURN … ;
END $$ ;
```

Dans ces exemples, on récupère la première ligne de la fonction avec `SELECT … INTO`, puis on ouvre un curseur implicite pour balayer chaque ligne obtenue d'une deuxième table. Le type `RECORD` permet de ne pas déclarer une nouvelle variable de type ligne.

8.10 EXÉCUTION DE REQUÊTE DANS UN BLOC PL/PGSQL

8.10.1 Requête dans un bloc PL/pgSQL

- Toutes opérations sur la base de données
- Et calculs, comparaisons, etc.
- Toute expression écrite en PL/pgSQL sera passée à `SELECT` pour interprétation par le moteur
- `PREPARE` implicite, avec cache

Par expression, on entend par exemple des choses comme :

```
IF myvar > 0 THEN
    myvar2 := 1 / myvar;
END IF;
```

Dans ce cas, l'expression `myvar > 0` sera préparée par le moteur de la façon suivante :

```
PREPARE statement_name(integer, integer) AS SELECT $1 > $2;
```

Puis cette requête préparée sera exécutée en lui passant en paramètre la valeur de `myvar` et la constante `0`.

Si `myvar` est supérieur à `0`, il en sera ensuite de même pour l'instruction suivante :

```
PREPARE statement_name(integer, integer) AS SELECT $1 / $2;
```

Comme toute requête préparée, son plan sera mis en cache.

Pour les détails, voir les dessous de PL/pgSQL[18].

8.10.2 Affectation d'une valeur à une variable

- Utiliser l'opérateur `:=` :

```
un_entier := 5;
```

- Utiliser `SELECT INTO` :

```
SELECT 5 INTO un_entier;
```

[18] https://docs.postgresql.fr/current/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING

Privilégiez la première écriture pour la lisibilité, la seconde écriture est moins claire et n'apporte rien puisqu'il s'agit ici d'une affectation de constante.

À noter que l'écriture suivante est également possible pour une affectation :

```
ma_variable := une_colonne FROM ma_table WHERE id = 5;
```

Cette méthode profite du fait que toutes les expressions du code PL/pgSQL vont être passées au moteur SQL de PostgreSQL dans un `SELECT` pour être résolues. Cela va fonctionner, mais c'est très peu lisible, et donc non recommandé.

8.10.3 Exécution d'une requête

- Affectation de la ligne :

```
SELECT *
INTO ma_variable_ligne  -- type ROW ou RECORD
FROM …;
```

- `INTO STRICT` pour garantir unicité
  - `INTO` seul : juste 1è ligne !
- Plus d'un enregistrement :
  - écrire une boucle
- Ordre statique :
  - colonnes, clause `WHERE`, tables figées

Récupérer une ligne de résultat d'une requête dans une ligne de type `ROW` ou `RECORD` se fait avec `SELECT … INTO`. La première ligne est récupérée. Généralement on préférera utiliser `INTO STRICT` pour lever une de ces erreurs si la requête renvoie zéro ou plusieurs lignes :

```
ERROR:  query returned no rows
ERROR:  query returned more than one row
```

Dans le cas du type `ROW`, la définition de la ligne doit correspondre parfaitement à la définition de la ligne renvoyée. Utiliser un type `RECORD` permet d'éviter ce type de problème. La variable obtient directement le type `ROW` de la ligne renvoyée.

Il est possible d'utiliser `SELECT INTO` avec une simple variable si l'on n'a qu'un champ d'une ligne à récupérer.

Cette fonction compte les tables, et en trace la liste (les tables ne font pas partie du résultat) :

```
CREATE OR REPLACE FUNCTION compte_tables () RETURNS int LANGUAGE plpgsql AS $$
DECLARE
  n int ;
  t RECORD ;
BEGIN
  SELECT count(*) INTO STRICT n
  FROM pg_tables ;

  FOR t IN SELECT * FROM pg_tables LOOP
    RAISE NOTICE 'Table %.%', t.schemaname, t.tablename;
  END LOOP ;

  RETURN n ;
END ;
$$ ;

# SELECT compte_tables ();

NOTICE:  Table pg_catalog.pg_foreign_server
NOTICE:  Table pg_catalog.pg_type
…
NOTICE:  Table public.pgbench_accounts
NOTICE:  Table public.pgbench_branches
NOTICE:  Table public.pgbench_tellers
NOTICE:  Table public.pgbench_history
 compte_tables
---------------
           186
(1 ligne)
```

8.10.4 Exécution d'une requête sans besoin du résultat

- `PERFORM` : résultat ignoré

```
PERFORM * FROM ma_table WHERE une_colonne>0 ;
PERFORM mafonction (argument1) ;
```

- Variable `FOUND`
  - si une ligne est affectée par l'instruction
- Nombre de lignes :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

On peut déterminer qu'aucune ligne n'a été trouvée par la requête en utilisant la variable `FOUND` :

```
PERFORM * FROM ma_table WHERE une_colonne>0;
IF NOT FOUND THEN
```

```
…
END IF;
```

Pour appeler une fonction, il suffit d'utiliser `PERFORM` de la manière suivante :

```
PERFORM mafonction(argument1);
```

Pour récupérer le nombre de lignes affectées par l'instruction exécutée, il faut récupérer la variable de diagnostic `ROW_COUNT` :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

Il est à noter que le `ROW_COUNT` récupéré ainsi s'applique à l'ordre SQL précédent, quel qu'il soit :

- `PERFORM` ;
- `EXECUTE` ;
- ou même à un ordre statique directement dans le code PL/pgSQL.

8.11 SQL DYNAMIQUE

8.11.1 EXECUTE d'une requête

```
EXECUTE 'chaine' [INTO [STRICT] cible] [USING (paramètres)] ;
```

- Exécute la requête dans `chaine`
- `chaine` peut être construite à partir d'autres variables
- `cible` : résultat (une seule ligne)

`EXECUTE` dans un bloc PL/pgSQL permet notamment du SQL dynamique : l'ordre peut être construit dans une variable.

8.11.2 EXECUTE & requête dynamique : injection SQL

Si `nom` vaut : « `'Robert' ; DROP TABLE eleves ;` »
que renvoie ceci ?

```
EXECUTE 'SELECT * FROM eleves WHERE nom = '|| nom ;
```

Un danger du SQL dynamique est de faire aveuglément confiance aux valeurs des variables en construisant un ordre SQL :

```
CREATE TEMP TABLE eleves (nom text, id int) ;
INSERT INTO eleves VALUES ('Robert', 0) ;

-- Mise à jour d'un ID
DO $f$
DECLARE
    nom text := $$'Robert' ; DROP TABLE eleves;$$  ;
    id  int ;
BEGIN
RAISE NOTICE 'A exécuter : %','SELECT * FROM eleves WHERE nom = '|| nom ;
EXECUTE 'UPDATE eleves SET id = 327 WHERE nom = '|| nom ;
END ;
$f$ LANGUAGE plpgsql ;
```

```
NOTICE:  A exécuter : SELECT * FROM eleves WHERE nom = 'Robert' ; DROP TABLE eleves;

\d+ eleves
Aucune relation nommée « eleves » n'a été trouvée.
```

Cet exemple est directement inspiré d'un dessin très connu de XKCD[19].

[19] https://xkcd.com/327/

Dans la pratique, la variable `nom` (entrée ici en dur) proviendra par exemple d'un site web, et donc contient potentiellement des caractères terminant la requête dynamique et en insérant une autre, potentiellement destructrice.

Moins grave, une erreur peut être levée à cause d'une apostrophe (*quote*) dans une chaîne texte. Il existe effectivement des gens avec une apostrophe dans le nom.

Ce qui suit concerne le SQL dynamique dans des routines PL/pgSQL, mais le principe concerne tous les langages et clients, y compris `psql` et sa méta-commande `\gexec`[20]. En SQL pur, la protection contre les injections SQL est un argument pour utiliser les requêtes préparées[21], dont l'ordre `EXECUTE` diffère de celui-ci du PL/pgSQL ci-dessous.

8.11.3 EXECUTE & requête dynamique : 3 possibilités

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(nom_colonne)
    || ' = '
    || quote_literal(nouvelle_valeur)
    || ' WHERE cle = '
    || quote_literal(valeur_cle) ;

EXECUTE format('UPDATE matable SET %I = %L '
   'WHERE clef = %L', nom_colonne, nouvelle_valeur, valeur_clef);

EXECUTE format('UPDATE table SET %I = $1 '
   'WHERE clef = $2', nom_colonne) USING nouvelle_valeur, valeur_clef;
```

Les trois exemples précédents sont équivalents.

Le premier est le plus simple au premier abord. Il utilise `quote_ident` et `quote_literal` pour protéger des injections SQL[22] (voir plus loin).

Le second est plus lisible grâce à la fonction de formatage `format`[23] qui évite ces concaténations et appelle implicitement les fonctions `quote_%` Si un paramètre ne peut pas prendre la valeur NULL, utiliser `%L` (équivalent de `quote_nullable`) et non `%I` (équivalent de `quote_ident`).

La troisième alternative avec `USING` et les paramètres numériques `$1` et `$2` est considérée comme la plus performante. (Voir les détails dans la documentation[24]).

[20] https://docs.postgresql.fr/current/app-psql.html#APP-PSQL-META-COMMANDS
[21] https://docs.postgresql.fr/current/sql-prepare.html
[22] https://fr.wikipedia.org/wiki/Injection_SQL
[23] https://docs.postgresql.fr/current/functions-string.html#FUNCTIONS-STRING-FORMAT
[24] https://docs.postgresql.fr/current/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE

L'exemple complet suivant tiré de la documentation officielle[25] utilise `EXECUTE` pour rafraîchir des vues matérialisées en masse.

```
CREATE FUNCTION rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Rafraîchissement de toutes les vues matérialisées…';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP
        -- Maintenant "mviews" contient un enregistrement
        -- avec les informations sur la vue matérialisé
        RAISE NOTICE 'Rafraichissement de la vue matérialisée %.% (owner: %)…',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I',
                       mviews.mv_schema, mviews.mv_name) ;
    END LOOP;

    RAISE NOTICE 'Fin du rafraîchissement';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

8.11.4 EXECUTE & requête dynamique (suite)

```
EXECUTE 'chaine' [INTO STRICT cible] [USING (paramètres)] ;
```

- `STRICT` : **1** résultat
  - sinon `NO_DATA_FOUND` ou `TOO_MANY_ROWS`
- Sans `STRICT` :
  - 1ère ligne ou `NO_DATA_FOUND`
- Nombre de lignes :
  - `GET DIAGNOSTICS integer_var = ROW_COUNT`

[25] https://www.postgresql.org/docs/current/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE

De la même manière que pour `SELECT … INTO`, utiliser `STRICT` permet de garantir qu'il y a exactement une valeur comme résultat de `EXECUTE`, ou alors une erreur sera levée.

Nous verrons plus loin comment traiter les exceptions.

8.11.5 Outils pour construire une requête dynamique

- `quote_ident ()`
  - pour mettre entre guillemets un identifiant d'un objet PostgreSQL (table, colonne, etc.)
- `quote_literal ()`
  - pour mettre entre guillemets une valeur (chaîne de caractères)
- `quote_nullable ()`
  - pour mettre entre guillemets une valeur (chaîne de caractères), sauf NULL qui sera alors renvoyé sans les guillemets
- `||` : concaténer
- Ou fonction `format(…)`, équivalent de `sprintf` en C

La fonction `format` est l'équivalent de la fonction `sprintf` en C : elle formate une chaîne en fonction d'un patron et de valeurs à appliquer à ses paramètres et la retourne. Les types de paramètre reconnus par `format` sont :

- `%I` : est remplacé par un identifiant d'objet. C'est l'équivalent de la fonction `quote_ident`. L'objet en question est entouré de guillemets doubles si nécessaire ;
- `%L` : est remplacé par une valeur littérale. C'est l'équivalent de la fonction `quote_literal`. Des guillemets simples sont ajoutés à la valeur et celle-ci est correctement échappée si nécessaire ;
- `%s` : est remplacé par la valeur donnée sans autre forme de transformation ;
- `%%` : est remplacé par un simple `%`.

Voici un exemple d'utilisation de cette fonction, utilisant des paramètres positionnels :

```
SELECT format(
    'SELECT %I FROM %I WHERE %1$I=%3$L',
    'MaColonne',
    'ma_table',
    $$l'été$$
);
```

```
                          format
-----------------------------------------------------------
 SELECT "MaColonne" FROM ma_table WHERE "MaColonne"='l''été'
```

8.12 STRUCTURES DE CONTRÔLE EN PL/PGSQL

- But du PL : les traitements procéduraux

8.12.1 Tests conditionnels - 2

Exemple :

```
IF nombre = 0 THEN
  resultat := 'zero';
ELSEIF nombre > 0 THEN
   resultat := 'positif';
ELSEIF nombre < 0 THEN
   resultat := 'négatif';
ELSE
   resultat := 'indéterminé';
END IF;
```

8.12.2 Tests conditionnels : CASE

```
    CASE nombre
    WHEN nombre = 0  THEN 'zéro'
    WHEN variable > 0  THEN 'positif'
    WHEN variable < 0  THEN 'négatif'
    ELSE 'indéterminé'
    END CASE
```

ou :

```
CASE current_setting ('server_version_num')::int/10000
    WHEN 8,9,10,11      THEN RAISE NOTICE 'Version non supportée !!' ;
    WHEN 12,13,14,15,16 THEN RAISE NOTICE 'Version supportée' ;
    ELSE                     RAISE NOTICE 'Version inconnue (fin 2023)' ;
END CASE ;
```

L'instruction `CASE WHEN` est proche de l'expression `CASE`[26] des requêtes SQL dans son principe (à part qu'elle se clôt par `END` en SQL, et `END CASE` en PL/pgSQL).

Elle est parfois plus légère à lire que des `IF` imbriqués.

[26] https://docs.postgresql.fr/current/functions-conditional.html#FUNCTIONS-CASE

Exemple complet :

```
DO $$
BEGIN
CASE current_setting ('server_version_num')::int/10000
    WHEN 8,9,10,11      THEN RAISE NOTICE 'Version non supportée !!' ;
    WHEN 12,13,14,15,16 THEN RAISE NOTICE 'Version supportée' ;
    ELSE                 RAISE NOTICE 'Version inconnue (fin 2023)' ;
END CASE ;
END ;
$$ LANGUAGE plpgsql ;
```

8.12.3 Boucle LOOP/EXIT/CONTINUE : syntaxe

- Boucle :
  - `LOOP` / `END LOOP`
  - label possible
- En sortir :
  - `EXIT [label] [WHEN expression_booléenne]`
- Commencer une nouvelle itération de la boucle
  - `CONTINUE [label] [WHEN expression_booléenne]`

Des boucles simples s'effectuent avec `LOOP` / `END LOOP`.

Pour les détails, voir la documentation officielle[27].

8.12.4 Boucle LOOP/EXIT/CONTINUE : exemple

```
LOOP
  resultat := resultat + 1;
  EXIT WHEN resultat > 100;
  CONTINUE WHEN resultat < 50;
  resultat := resultat + 1;
END LOOP;
```

Cette boucle incrémente le résultat de 1 à chaque itération tant que la valeur du résultat est inférieure

[27] https://docs.postgresql.fr/current/plpgsql-control-structures.html#PLPGSQL-CONTROL-STRUCTURES-LOOPS

à 50. Ensuite, le résultat est incrémenté de 1 à deux reprises pour chaque tour de boucle. On incrémente donc de 2 par tour de boucle. Arrivée à 100, la procédure sort de la boucle.

8.12.5 Boucle WHILE

```
WHILE condition LOOP
  …
END LOOP;
```

- Boucle jusqu'à ce que la condition soit fausse
- Label possible

8.12.6 Boucle FOR : syntaxe

```
FOR variable in [REVERSE] entier1..entier2 [BY incrément]
LOOP
  …
END LOOP;
```

- `variable` va obtenir les différentes valeurs entre entier1 et entier2
- Label possible

La boucle `FOR` n'a pas d'originalité par rapport à d'autres langages.

L'option `BY` permet d'augmenter l'incrémentation :

```
FOR variable in 1..10 BY 5…
```

L'option `REVERSE` permet de faire défiler les valeurs en ordre inverse :

```
FOR variable in REVERSE 10..1 …
```

8.12.7 Boucle FOR … IN … LOOP : parcours de résultat de requête

```
FOR ligne IN ( SELECT * FROM ma_table ) LOOP
  …
END LOOP;
```

- Pour boucler dans les lignes résultats d'une requête
- `ligne` de type `RECORD`, `ROW`, ou liste de variables séparées par des virgules
- Utilise un curseur en interne
- Label possible

Cette syntaxe très pratique permet de parcourir les lignes résultant d'une requête sans avoir besoin de créer et parcourir un curseur. Souvent on utilisera une variable de type `ROW` ou `RECORD` (comme dans l'exemple de la fonction `rafraichir_vuemat` plus haut), mais l'utilisation directe de variables (déclarées préalablement) est possible :

```
FOR a, b, c, d IN
  (SELECT col_a, col_b, col_c, col_d FROM ma_table)
LOOP
  -- instructions utilisant ces variables
  …
END LOOP;
```

Attention de ne pas utiliser les variables en question hors de la boucle, elles auront gardé la valeur acquise dans la dernière itération.

8.12.8 Boucle FOREACH

```
FOREACH variable [SLICE n] IN ARRAY expression LOOP
    …
END LOOP ;
```

- Pour boucler sur les éléments d'un tableau
- `variable` va obtenir les différentes valeurs du tableau retourné par `expression`
- `SLICE` permet de jouer sur le nombre de dimensions du tableau à passer à la variable
- Label possible

Voici deux exemples permettant d'illustrer l'utilité de `SLICE` :

- sans `SLICE` :

```
DO $$
DECLARE a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int;
BEGIN
  FOREACH b IN ARRAY a LOOP
  RAISE INFO 'var: %', b;
END LOOP;
END $$ ;
```

```
INFO:  var: 1
INFO:  var: 2
INFO:  var: 3
INFO:  var: 4
INFO:  var: 5
INFO:  var: 6
```

- avec `SLICE` :

```
DO $$
DECLARE a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int[];
BEGIN
  FOREACH b SLICE 1 IN ARRAY a LOOP
  RAISE INFO 'var: %', b;
END LOOP;
END $$;
```

```
INFO:  var: {1,2}
INFO:  var: {3,4}
INFO:  var: {5,6}
```

et avec `SLICE 2`, on obtient :

```
INFO:  var: {{1,2},{3,4},{5,6}}
```

8.13 AUTRES PROPRIÉTÉS DES FONCTIONS

- Sécurité
- Optimisations
- Parallélisation

8.13.1 Politique de sécurité

- `SECURITY INVOKER` : défaut
- `SECURITY DEFINER`
  - « sudo de la base de données »
  - potentiellement dangereux
  - ne pas laisser à **public** !

Une fonction `SECURITY INVOKER` s'exécute avec les droits de l'appelant. C'est le mode par défaut.

Une fonction `SECURITY DEFINER` s'exécute avec les droits du créateur. Cela permet, au travers d'une fonction, de permettre à un utilisateur d'outrepasser ses droits de façon contrôlée. C'est l'équivalent du `sudo` d'Unix.

Bien sûr, une fonction `SECURITY DEFINER` doit faire l'objet d'encore plus d'attention qu'une fonction normale. Elle peut facilement constituer un trou béant dans la sécurité de votre base. C'est encore plus important si le propriétaire de la fonction est un superutilisateur, car celui-ci a la possibilité d'accéder aux fichiers de PostgreSQL et au système d'exploitation.

Plusieurs points importants sont à noter pour `SECURITY DEFINER` :

- Par défaut, toute fonction créée dans **public** est exécutable par le rôle **public**. La première chose à faire est donc de révoquer ce droit. Mieux : créer la fonction dans un schéma séparé est recommandé pour gérer plus finalement les accès.
- Il faut se protéger des variables de session qui pourraient être utilisées pour modifier le comportement de la fonction, en particulier le *search_path* (qui pourrait faire pointer vers des tables de même nom dans un autre schéma). Il doit donc **impérativement** être positionné en dur dans cette fonction (soit d'emblée, avec un `SET` en début de fonction, soit en positionnant un `SET` dans le `CREATE FUNCTION`) ; et/ou les fonctions doivent préciser systématiquement le schéma dans les appels de tables (`SELECT … FROM nomschema.nomtable …`).

Exemple d'une fonction en `SECURITY DEFINER` avec un *search path* sécurisé :

```
\c pgbench pgbench

-- A exécuter en tant que pgbench, propriétaire de la base pgbench
CREATE SCHEMA pgbench_util ;

CREATE OR REPLACE FUNCTION pgbench_util.accounts_balance (pbid integer)
 RETURNS integer
 LANGUAGE sql
 IMMUTABLE PARALLEL SAFE
 SECURITY DEFINER
 SET search_path TO ''   -- précaution supplémentaire
AS $function$
  SELECT bbalance FROM public.pgbench_branches br WHERE br.bid = pbid ;
$function$ ;

GRANT USAGE ON SCHEMA pgbench_util TO lecteur ;
GRANT EXECUTE ON FUNCTION pgbench_util.accounts_balance TO lecteur ;
```

L'utilisateur **lecteur** peut bien lire le résultat de la fonction sans accès à la table :

```
\c pgbench lecteur

SELECT pgbench_util.accounts_balance (5) ;

 accounts_balance
------------------
                0
```

Exemple de fonction laxiste et d'attaque :

```
-- Exemple sur une base pgbench, appartenant à pgbench
-- créée par exemple ainsi :
-- createdb pgbench -O pgbench
-- pgbench -U pgbench -i -s 1  pgbench
-- Deux utilisateurs :
--   pgbench
```

```
--   attaquant qui a son propre schéma

\set timing off
\set ECHO all
\set ON_ERROR_STOP 1

\c pgbench pgbench

-- Fonction non sécurisée fournie par l'utilisateur pgbench
-- à tout le monde par public
CREATE OR REPLACE FUNCTION public.accounts_balance_insecure(pbid integer)
 RETURNS integer
 LANGUAGE plpgsql
 IMMUTABLE PARALLEL SAFE
 SECURITY DEFINER
 -- oublié : SET search_path TO ''
AS $function$ BEGIN
  RETURN bbalance FROM /* pas de schéma */ pgbench_branches br
                  WHERE br.bid = pbid ;
END $function$ ;

-- Droits trop ouverts
GRANT EXECUTE ON FUNCTION accounts_balance_insecure TO public ;

-- Résultat normal : renvoie 0
SELECT * FROM accounts_balance_insecure (1) ;

-- Création d'un utilisateur avec droit d'écrire dans un schéma
\c pgbench postgres

DROP SCHEMA IF EXISTS piege CASCADE ;
--DROP ROLE attaquant ;

CREATE ROLE attaquant LOGIN ;   -- pg_hba.conf laissé en exercice au lecteur

-- Il faut que l'attaquant ait un schéma où écrire,
-- et puisse donner l'accès à la victime.
-- Le schéma public convient parfaitement pour cela avant PostgreSQL 15…
CREATE SCHEMA piege ;
GRANT ALL ON SCHEMA piege TO attaquant WITH GRANT OPTION ;

\c pgbench attaquant

\conninfo

-- Résultat normal (accès peut-être indu mais pour le moment sans danger)
SELECT * FROM accounts_balance_insecure (1) ;

-- L'attaquant peut voir la fonction et étudier comment la détourner
\sf accounts_balance_insecure

-- Fonction que l'attaquant veut faire exécuter à pgbench
CREATE FUNCTION piege.lit_donnees_cachees ()
RETURNS TABLE (bid int, bbalance int)
LANGUAGE plpgsql
AS $$
```

```
DECLARE
    n int ;
BEGIN
    -- affichage de l'utilisateur pgbench
    RAISE NOTICE 'Entrée dans fonction piégée en tant que %', current_user ;
    -- copie de données non autorisées dans le schéma de l'attaquant
    CREATE TABLE piege.donnees_piratees AS SELECT * FROM pgbench_tellers ;
    GRANT ALL ON piege.donnees_piratees TO attaquant ;
    -- destruction de données…
    DROP TABLE IF EXISTS pgbench_history ;
    -- sortie propre impérative pour éviter le rollback
    RETURN QUERY SELECT 666 AS bid, 42 AS bbalance ;
END ;
$$ ;

-- Vue d'enrobage pour « masquer » la vraie table de même nom
CREATE OR REPLACE VIEW piege.pgbench_branches AS
SELECT * FROM piege.lit_donnees_cachees () ;

-- Donner les droits au compte attaqué sur les objets
-- de l'attaquant
GRANT USAGE,CREATE ON SCHEMA piege TO pgbench ;
GRANT ALL ON piege.pgbench_branches TO pgbench ;
GRANT ALL ON FUNCTION piege.lit_donnees_cachees TO pgbench ;

-- Détournement du chemin d'accès
SET search_path TO piege,public ;
-- Attaque
SELECT * FROM accounts_balance_insecure (666) ;

-- Lecture des données piratées
SELECT COUNT (*) as nb_lignes_recuperees FROM piege.donnees_piratees ;
```

8.13.2 Optimisation des fonctions

- Fonctions uniquement
- À destination de l'optimiseur
- `COST cout_execution`
  - coût estimé pour l'exécution de la fonction
- `ROWS nb_lignes_resultat`
  - nombre estimé de lignes que la fonction renvoie

`COST` est un coût représenté en unité de `cpu_operator_cost` (100 par défaut).

`ROWS` vaut par défaut 1000 pour les fonctions `SETOF` ou `TABLE`, et 1 pour les autres.

Ces deux paramètres ne modifient pas le comportement de la fonction. Ils ne servent que pour aider l'optimiseur de requête à estimer le coût d'appel à la fonction, afin de savoir, si plusieurs plans sont possibles, lequel est le moins coûteux par rapport au nombre d'appels de la fonction et au nombre d'enregistrements qu'elle retourne.

8.13.3 Parallélisation

- Fonctions uniquement
- La fonction peut-elle être exécutée en parallèle ?
  - `PARALLEL UNSAFE` (défaut)
  - `PARALLEL RESTRICTED`
  - `PARALLEL SAFE`

`PARALLEL UNSAFE` indique que la fonction ne peut pas être exécutée dans le mode parallèle. La présence d'une fonction de ce type dans une requête SQL force un plan d'exécution en série. C'est la valeur par défaut.

Une fonction est non parallélisable si elle modifie l'état d'une base ou si elle fait des changements sur la transaction.

`PARALLEL RESTRICTED` indique que la fonction peut être exécutée en mode parallèle mais l'exécution est restreinte au processus principal d'exécution.

Une fonction peut être déclarée comme restreinte si elle accède aux tables temporaires, à l'état de connexion des clients, aux curseurs, aux requêtes préparées.

`PARALLEL SAFE` indique que la fonction s'exécute correctement dans le mode parallèle sans restriction.

En général, si une fonction est marquée sûre ou restreinte à la parallélisation alors qu'elle ne l'est pas, elle pourrait renvoyer des erreurs ou fournir de mauvaises réponses lorsqu'elle est utilisée dans une requête parallèle.

En cas de doute, les fonctions doivent être marquées comme `UNSAFE`, ce qui correspond à la valeur par défaut.

8.14 UTILISATION DE FONCTIONS DANS LES INDEX

- Fonctions uniquement !
- `IMMUTABLE | STABLE | VOLATILE`
- Ce mode précise la « volatilité » de la fonction.
- Permet de réduire le nombre d'appels
- Index : fonctions immutables uniquement (sinon problèmes !)

On peut indiquer à PostgreSQL le niveau de volatilité (ou de stabilité) d'une fonction. Ceci permet d'aider PostgreSQL à optimiser les requêtes utilisant ces fonctions, mais aussi d'interdire leur utilisation dans certains contextes.

Une fonction est « **immutable** » si son exécution ne dépend que de ses paramètres. Elle ne doit donc dépendre ni du contenu de la base (pas de `SELECT`, ni de modification de donnée de quelque sorte), ni d'**aucun** autre élément qui ne soit pas un de ses paramètres. Les fonctions arithmétiques simples (`+`, `*`, `abs`…) sont immutables.

À l'inverse, `now()` n'est évidemment pas immutable. Une fonction sélectionnant des données d'une table non plus. `to_char()` n'est pas non plus immutable, car son comportement dépend des paramètres de session, par exemple `to_char(timestamp with time zone, text)` dépend du paramètre de session `timezone`…

Une fonction est « **stable** » si son exécution donne toujours le même résultat sur toute la durée d'un ordre SQL, pour les mêmes paramètres en entrée. Cela signifie que la fonction ne modifie pas les données de la base. Une fonction n'exécutant que des `SELECT` sur des tables (pas des fonctions !) sera stable. `to_char()` est stable. L'optimiseur peut réduire ainsi le nombre d'appels sans que ce soit en pratique toujours le cas.

Une fonction est « **volatile** » dans tous les autres cas. `random()` est volatile. Une fonction volatile peut même modifier les donneés. Une fonction non déclarée comme stable ou immutable est volatile par défaut.

La volatilité des fonctions intégrées à PostgreSQL est déjà définie. C'est au développeur de préciser la volatilité des fonctions qu'il écrit. Ce n'est pas forcément évident. Une erreur peut poser des problèmes quand le plan est mis en cache, ou, on le verra, dans des index.

Quelle importance cela a-t-il ?

Prenons une table d'exemple sur les heures de l'année 2020 :

```
-- Une ligne par heure dans l année, 8784 lignes
CREATE TABLE heures
AS
SELECT i, '2020-01-01 00:00:00+01:00'::timestamptz + i * interval '1 hour'  AS t
FROM generate_series (1,366*24) i;
```

Définissons une fonction un peu naïve ramenant le premier jour du mois, volatile faute de mieux :

```
CREATE OR REPLACE FUNCTION premierjourdumois(t timestamptz)
RETURNS timestamptz
LANGUAGE plpgsql
VOLATILE
AS $$
BEGIN
    RAISE notice 'appel premierjourdumois' ; -- trace des appels
    RETURN date_trunc ('month', t);
END $$ ;
```

Demandons juste le plan d'un appel ne portant que sur le dernier jour :

```
EXPLAIN SELECT * FROM heures
WHERE t > premierjourdumois('2020-12-31 00:00:00+02:00'::timestamptz)
LIMIT 10 ;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
Limit  (cost=0.00..8.04 rows=10 width=12)
 ->  Seq Scan on heures  (cost=0.00..2353.80 rows=2928 width=12)
       Filter: (t > premierjourdumois(
                    '2020-12-30 23:00:00+01'::timestamp with time zone))
```

Le nombre de lignes attendues (2928) est le tiers de la table, alors que nous ne demandons que le dernier mois. Il s'agit de l'estimation forfaitaire que PostgreSQL utilise faute d'informations sur ce que va retourner la fonction.

Demander à voir le résultat mène à l'affichage de milliers de `NOTICE` : la fonction est appelée à chaque ligne pour calculer s'il faut filtrer la valeur. En effet, une fonction volatile sera systématiquement exécutée à chaque appel, et, selon le plan, ce peut être pour chaque ligne parcourue !

Cependant notre fonction ne fait que des calculs à partir du paramètre, sans effet de bord. Déclarons-la donc stable :

```
ALTER FUNCTION premierjourdumois(timestamp with time zone) STABLE ;
```

Une fonction stable peut en théorie être remplacée par son résultat pendant l'exécution de la requête. Mais c'est impossible de le faire plus tôt, car on ne sait pas forcément dans quel contexte la fonction va être appelée (par exemple, en cas de requête préparée, les paramètres de la session ou les données de la base peuvent même changer entre la planification et l'exécution).

Dans notre cas, le même `EXPLAIN` simple mène à ceci :

```
NOTICE:  appel premierjourdumois
                                  QUERY PLAN
------------------------------------------------------------------------------
Limit  (cost=0.00..32.60 rows=10 width=12)
 ->  Seq Scan on heures  (cost=0.00..2347.50 rows=720 width=12)
       Filter: (t > premierjourdumois(
                    '2020-12-30 23:00:00+01'::timestamp with time zone))
```

Comme il s'agit d'un simple `EXPLAIN`, la requête n'est pas exécutée. Or le message `NOTICE` est renvoyé : la fonction est donc exécutée pour une simple planification. Un appel unique suffit, puisque

la valeur d'une fonction stable ne change pas pendant toute la durée de la requête pour les mêmes paramètres (ici une constante). Cet appel permet d'affiner la volumétrie des valeurs attendues, ce qui peut avoir un impact énorme.

Cependant, à l'exécution, les `NOTICE` apparaîtront pour indiquer que la fonction est à nouveau appelée à chaque ligne. Pour qu'un seul appel soit effectué pour toute la requête, il faudrait déclarer la fonction comme immutable, ce qui serait faux, puisqu'elle dépend implicitement du fuseau horaire.

Dans l'idéal, une fonction immutable peut être remplacée par son résultat avant même la planification d'une requête l'utilisant. C'est le cas avec les calculs arithmétiques par exemple :

```
EXPLAIN SELECT * FROM heures
WHERE i > abs(364*24) AND t > '2020-06-01'::date + interval '57 hours' ;
```

La valeur est substituée très tôt, ce qui permet de les comparer aux statistiques :

```
Seq Scan on heures  (cost=0.00..179.40 rows=13 width=12)
  Filter: ((i > 8736) AND (t > '2020-06-03 09:00:00'::timestamp without time zone))
```

Pour forcer un appel unique quand on sait que la fonction renverra une constante, du moins le temps de la requête, même si elle est volatile, une astuce est de signifier à l'optimiseur qu'il n'y aura qu'une seule valeur de comparaison, même si on ne sait pas laquelle :

```
EXPLAIN (ANALYZE) SELECT * FROM heures
WHERE t > (SELECT premierjourdumois('2020-12-31 00:00:00+02:00'::timestamptz)) ;

NOTICE:  appel premierjourdumois
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on heures  (cost=0.26..157.76 rows=2920 width=12)
                     (actual time=1.090..1.206 rows=721 loops=1)
   Filter: (t > $0)
   Rows Removed by Filter: 8039
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.26 rows=1 width=8)
                 (actual time=0.138..0.139 rows=1 loops=1)
 Planning Time: 0.058 ms
 Execution Time: 1.328 ms
```

On note qu'il n'y a qu'un appel. On comprend donc l'intérêt de se poser la question à l'écriture de chaque fonction.

La volatilité est encore plus importante quand il s'agit de créer des fonctions sur index :

```
CREATE INDEX ON heures (premierjourdumois( t )) ;

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

Ceci n'est possible que si la fonction est immutable. En effet, si le résultat de la fonction dépend de l'état de la base ou d'autres paramètres, la fonction exécutée au moment de la création de la clé d'index pourrait ne plus retourner le même résultat quand viendra le moment de l'interroger. PostgreSQL n'acceptera donc que les fonctions immutables dans la déclaration des index fonctionnels.

Déclarer hâtivement une fonction comme immutable juste pour pouvoir l'utiliser dans un index est dangereux : en cas d'erreur, les résultats d'une requête peuvent alors dépendre du plan d'exécution, selon que les index seront utilisés ou pas !

Cela est particulièrement fréquent quand les fuseaux horaires ou les dictionnaires sont impliqués. Vérifiez bien que vous n'utilisez que des fonctions immutables dans les index fonctionnels, les pièges sont nombreux.

Par exemple, si l'on veut une version immutable de la fonction précédente, il faut fixer le fuseau horaire dans l'appel à `date_trunc`. En effet, on peut voir avec `df+ date_trunc` que la seule version immutable de `date_trunc` n'accepte que des `timestamp` (sans fuseau), et en renvoie un. Notre fonction devient donc :

```
CREATE OR REPLACE FUNCTION premierjourdumois_utc(t timestamptz)
RETURNS timestamptz
LANGUAGE plpgsql
IMMUTABLE
AS $$
DECLARE
    jour1   timestamp ; --sans TZ
BEGIN
    jour1 := date_trunc ('month', (t at time zone 'UTC')::timestamp) ;
    RETURN jour1 AT TIME ZONE 'UTC';
END $$ ;
```

Testons avec une date dans les dernières heures de septembre en Alaska, qui correspond au tout début d'octobre en temps universel, et par exemple aussi au Japon :

```
\x

SET timezone TO 'US/Alaska';

SELECT d,
       d AT TIME ZONE 'UTC' AS d_en_utc,
       premierjourdumois_utc (d),
       premierjourdumois_utc (d) AT TIME ZONE 'UTC' as pjm_en_utc
FROM (SELECT '2020-09-30 18:00:00-08'::timestamptz AS d) x;

-[ RECORD 1 ]---------+----------------------
d                     | 2020-09-30 18:00:00-08
d_en_utc              | 2020-10-01 02:00:00
premierjourdumois_utc | 2020-09-30 16:00:00-08
pjm_en_utc            | 2020-10-01 00:00:00

SET timezone TO 'Japan';

SELECT d,
       d AT TIME ZONE 'UTC' AS d_en_utc,
       premierjourdumois_utc (d),
       premierjourdumois_utc (d) AT TIME ZONE 'UTC' as pjm_en_utc
FROM (SELECT '2020-09-30 18:00:00-08'::timestamptz AS d) x;

-[ RECORD 1 ]---------+----------------------
d                     | 2020-10-01 11:00:00+09
```

```
d_en_utc               | 2020-10-01 02:00:00
premierjourdumois_utc | 2020-10-01 09:00:00+09
pjm_en_utc             | 2020-10-01 00:00:00
```

Malgré les différences d'affichage dues au fuseau horaire, c'est bien le même moment (la première seconde d'octobre en temps universel) qui est retourné par la fonction.

Pour une fonction aussi simple, la version SQL est même préférable :

```
CREATE OR REPLACE FUNCTION premierjourdumois_utc(t timestamptz)
RETURNS timestamptz
LANGUAGE sql
IMMUTABLE
AS $$
    SELECT (date_trunc ('month',
                        (t at time zone 'UTC')::timestamp
                       )
           ) AT TIME ZONE 'UTC';
$$ ;
```

Enfin, la volatilité a également son importance lors d'autres opérations d'optimisation, comme l'exclusion de partitions. Seules les fonctions immutables sont compatibles avec le *partition pruning* effectué à la planification, mais les fonctions stable sont éligibles au *dynamic partition pruning* (à l'exécution) apparu avec PostgreSQL 11.

8.15 CONCLUSION

- Grand nombre de structure de contrôle (test, boucle, etc.)
- Facile à utiliser et à comprendre

8.15.1 Pour aller plus loin

- Documentation officielle
  - « Chapitre 40. PL/pgSQL - Langage de procédures SQL »
- Module de formation Dalibo P2[28]
  - variadic, routines polymorphes
  - triggers, tables de transition
  - curseurs
  - gestion des erreurs
  - sécurité
  - optimisation

La documentation officielle sur le langage PL/pgSQL peut être consultée en français à cette adresse[29].

8.15.2 Questions

```
FOR q IN (SELECT * FROM questions ) LOOP

  répondre (q) ;

END LOOP ;
```

[29] https://docs.postgresql.fr/current/plpgsql.html

8.16 QUIZ

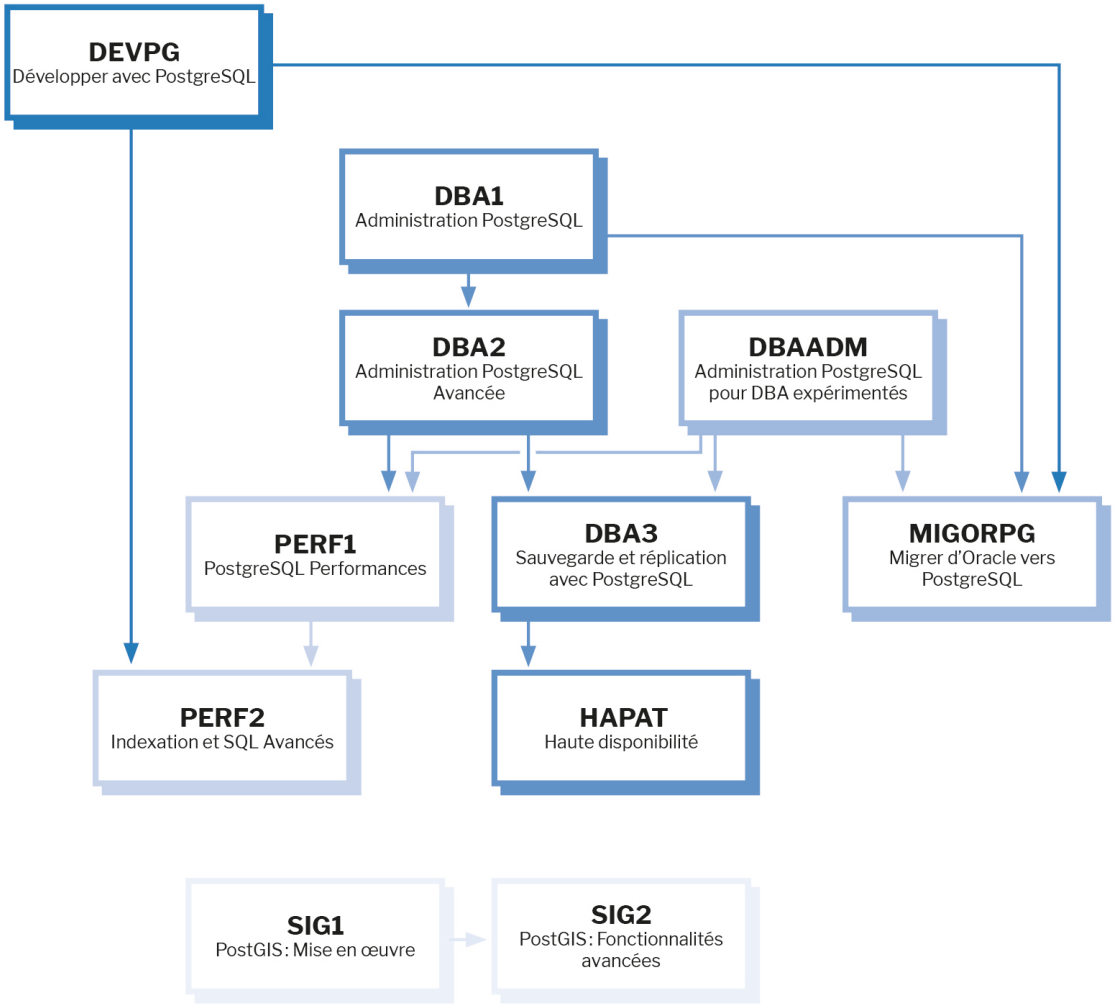
https://dali.bo/p1_quiz

Les formations Dalibo

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

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

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

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

Téléchargement gratuit

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