Module J0

Introduction à EXPLAIN

25.06

Table des matières

Sur ce document . . . 1
Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 2
Forme de ce manuel . . . 2
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 3
Versions de PostgreSQL couvertes . . . 3

1/ Introduction aux plans d'exécution 5
1.1 Introduction . . . 6
1.1.1 Au menu . . . 6
1.1.2 Niveau SGBD . . . 7
1.2 Optimiseur . . . 9
1.2.1 Principe de l'optimiseur . . . 9
1.2.2 Exemple de requête et son résultat . . . 10
1.2.3 Décisions de l'optimiseur . . . 11
1.3 Mécanisme de calcul de coûts . . . 12
1.3.1 Statistiques . . . 12
1.3.2 Exemple - parcours d'index . . . 13
1.3.3 Exemple - parcours de table . . . 14
1.3.4 Exemple - parcours d'index forcé . . . 14
1.4 Qu'est-ce qu'un plan d'exécution ? . . . 16
1.4.1 Nœud d'exécution . . . 16
1.4.2 Récupérer un plan d'exécution . . . 17
1.4.3 Exemple de requête . . . 17
1.4.4 Plan pour cette requête . . . 17
1.4.5 Informations sur la ligne nœud . . . 18
1.4.6 Informations sur les lignes suivantes . . . 19
1.4.7 EXPLAIN (ANALYZE) . . . 19
1.4.8 EXPLAIN (ANALYZE, BUFFERS) . . . 20
1.4.9 EXPLAIN (ANALYZE, WAL) . . . 21
1.4.10 EXPLAIN (ANALYZE, SERIALIZE) . . . 22
1.4.11 EXPLAIN (GENERIC_PLAN) . . . 22
1.4.12 EXPLAIN (SETTINGS) . . . 23
1.4.13 Autres options . . . 24
1.4.14 Paramètre track_io_timing . . . 27
1.4.15 Détecter les problèmes . . . 28
1.5 Nœuds d'exécution les plus courants (introduction) . . . 29
1.5.1 Parcours . . . 29
1.5.2 Jointures . . . 30
1.5.3 Agrégats . . . 30

1.5.4 Opérations unitaires 31
1.6 Outils graphiques 32
1.6.1 pgAdmin 32
1.6.2 pgAdmin - copie d'écran 33
1.6.3 explain.depesz.com 33
1.6.4 explain.depesz.com - exemple 34
1.6.5 explain.dalibo.com 34
1.6.6 explain.dalibo.com - exemple 35
1.7 Conclusion 36
1.7.1 Questions 36
1.8 Quiz 37
1.9 Travaux pratiques 38
1.9.1 Manipuler explain 38
1.9.2 Manipuler explain (base magasin) 41
1.10 Travaux pratiques (solutions) 43
1.10.1 Manipuler explain 43
1.10.2 Manipuler explain (base magasin) 49

Les formations Dalibo **55**
Cursus des formations 55
Les livres blancs 56
Téléchargement gratuit 56

Sur ce document

Formation	Module J0
Titre	Introduction à EXPLAIN
Révision	25.06
PDF	https://dali.bo/j0_pdf
EPUB	https://dali.bo/j0_epub
HTML	https://dali.bo/j0_html
Slides	https://dali.bo/j0_slides
TP	https://dali.bo/j0_tp
TP (solutions)	https://dali.bo/j0_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, 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, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

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

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

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

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

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

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

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

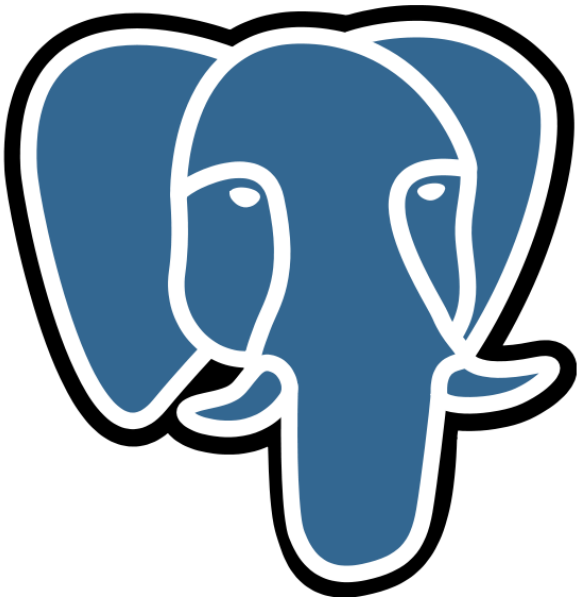
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ Introduction aux plans d’exécution



1.1 INTRODUCTION

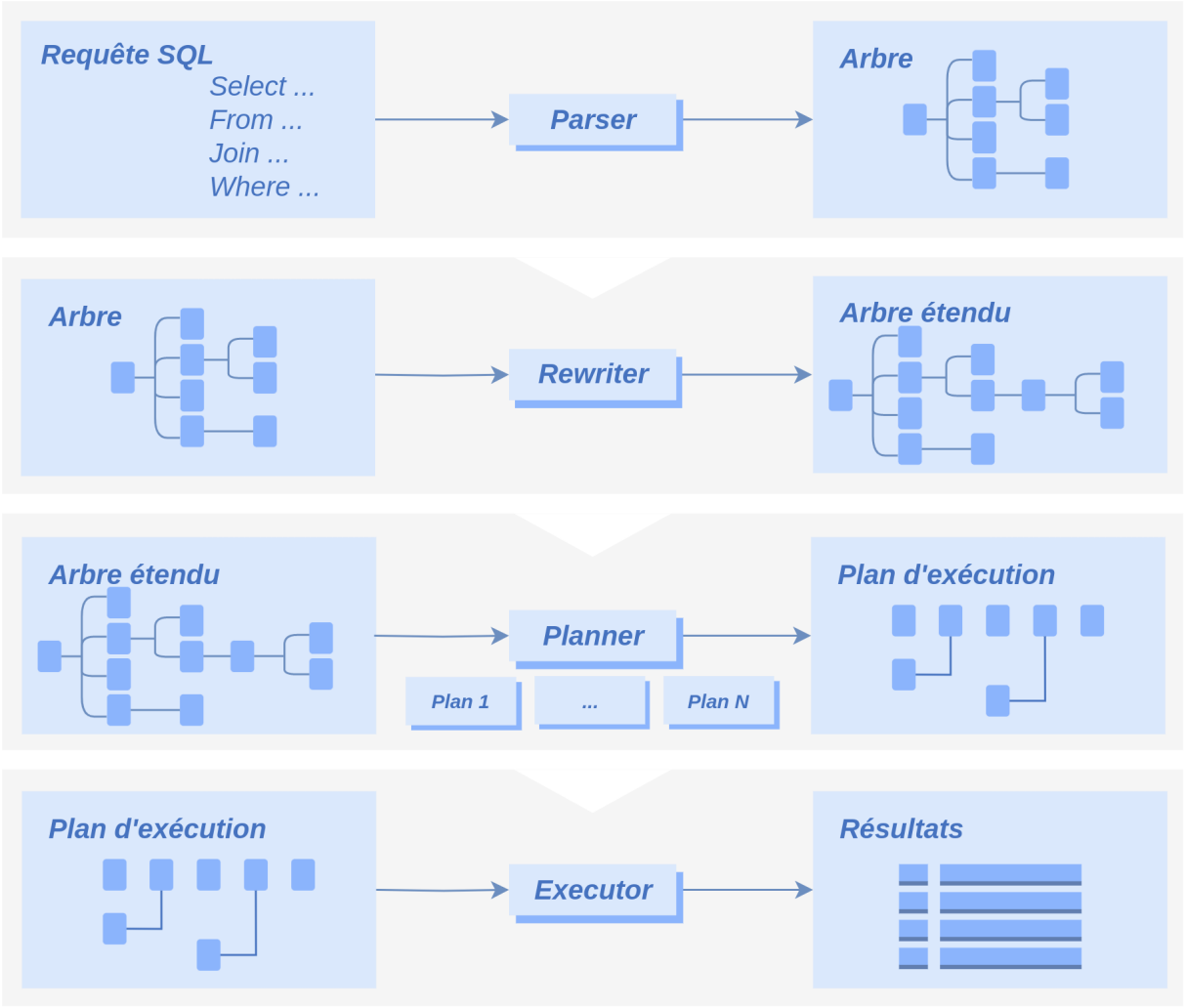
— Qu'est-ce qu'un plan d'exécution ?
— Quels outils peuvent aider

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.

1.1.1 Au menu

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

1.1.2 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.

1.2 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.

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.3 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.

1.3.1 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 !

En plus du coût unitaire de traitement d'une ligne, il faut connaître le nombre total de lignes à traiter pour calculer le coût total. L'optimiseur a donc besoin d'informations 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. Ces statistiques sur les données sont calculées lors de l'exécution de la commande SQL `ANALYZE`.

Le processus *autovacuum* permet de rafraîchir régulièrement les statistiques.

Le propriétaire de la table ou un superutilisateur peut également le faire manuellement. À partir de PostgreSQL 17, il est possible de créer un utilisateur dédié à la maintenance, qui, sans avoir de droits de lecture sur les tables, peut exécuter ces commandes. Cet utilisateur pourra intervenir soit sur toutes les tables s'il possède le rôle `pg_maintain`, soit sur des tables spécifiques si un `GRANT MAINTAIN` lui a été accordé.

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

1.3.2 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.

1.3.3 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).

1.3.4 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.

1.4 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 (en RAM ou sur disque).

Mais ce n'est qu'une des possibilités.

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.4.4 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`).

1.4.5 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.

1.4.6 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.

1.4.7 EXPLAIN (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).

1.4.8 EXPLAIN (ANALYZE, 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.

Il est conseillé de l'activer systématiquement. La quantité de blocs manipulés par une requête est souvent surprenante, et souvent la cause d'une mauvaise performance.

`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. `EXPLAIN (BUFFERS)` sans `ANALYZE` peut être utilisé, mais il ne montre que les blocs utilisés par la planification, plutôt que accédés à l'exécution. Ces blocs sont surtout des appels aux tables systèmes, et sont moins nombreux, voire absents, quand on appelle la requête une deuxième fois, ou plus, dans la même session.

1.4.9 EXPLAIN (ANALYZE, 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.) Il est conseillé de l'activer systématiquement, même pour les `SELECT`.

1.4.10 EXPLAIN (ANALYZE, SERIALIZE)

```
EXPLAIN (ANALYZE,VERBOSE,BUFFERS,SERIALIZE)
SELECT * FROM demotoast ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on public.demotoast  (cost=0.00..177.00 rows=1000 width=1196)
 ↪  (actual time=0.012..0.182 rows=1000 loops=1)
   Output: i, t, png, j
   Buffers: shared hit=167
 Query Identifier: 698565989149231362
 Planning Time: 0.055 ms
 Serialization: time=397.480 ms  output=307084kB  format=text
   Buffers: shared hit=14500
 Execution Time: 397.776 ms
```

— PostgreSQL 17

Cette option, apparue avec PostgreSQL 17, force la « sérialisation » du résultat de la requête, et met ainsi en ainsi en évidence sa participation dans le temps d'exécution total de la requête et la volumétrie de données à transférer. Sans cette option, `EXPLAIN (ANALYZE)` ne lit pas forcément toutes les données retournées, et affiche parfois un résultat plus optimiste que les requêtes réelles.

L'exemple montre l'impact de la lecture (pas forcément voulue) de la partie TOAST de la table, où sont relégués les gros champs textes ou binaires, à cause du `SELECT *`. Le coût réseau n'est toutefois pas inclus.

1.4.11 EXPLAIN (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.

1.4.12 EXPLAIN (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 la liste des paramètres influant sur la planification et qui ne sont pas à leur valeur par défaut pour la session ou la requête en cours. Il est conseillé de l'activer systématiquement. Elle est pratique quand il faut transmettre le plan à un collègue ou un prestataire qui n'a pas forcément accès à la machine et à sa configuration.

1.4.13 Autres options

- `VERBOSE`
  - affichage verbeux : schémas, colonnes, workers : conseillé
- `MEMORY`
  - mémoire utilisée pour le plan (v17)
- `COSTS OFF`
  - masquer les coûts
- `TIMING OFF`
  - désactiver le chronométrage & des informations vues/calculées par l'optimiseur
- `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 VERBOSE

N'hésitez pas à utiliser l'option `VERBOSE` pour 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 MEMORY

À partir de PostgreSQL 17, cette option affiche la mémoire consommée par le planificateur. Ce n'est utile que pour les requêtes compliquées. Cet exemple montre que même pour une requête très basique, un peu de mémoire est utilisée uniquement pour la planification :

```
EXPLAIN (MEMORY) SELECT relname FROM pg_class ;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on pg_class  (cost=0.00..27.27 rows=527 width=64)
 Planning:
   Memory: used=9kB  allocated=16kB
```

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 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/

1.4.14 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
```

— Tester la machine avec `pg_test_timing`

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).

Avant d'activer `track_io_timing`, vérifiez les capacités de la machine avec l'outil `pg_test_timing`.

pg_test_timing[2] est livré avec PostgreSQL et mesure les performances de l'horloge système. Si le temps de mesure renvoyé sur la deuxième ligne n'est que de quelques dizaines de nanosecondes, la machine est suffisamment rapide pour que `track_io_timing` renvoie des résultats précis et sans ralentir la requête. C'est le cas sur presque toutes les machines et systèmes d'exploitation actuels, mais il y a parfois des surprises, par exemple dans certaines machines virtuelles ou selon la source de

[2] https://docs.postgresql.fr/current/pgtesttiming.html

l'horloge système. Sinon, éviter d'activer `track_io_timing` sur un serveur de production. Sur une machine de test ou de formation, ce n'est pas un problème.

1.4.15 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.

1.5 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[3] et J6[4].

1.5.1 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 ;
- *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.

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

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

1.5.2 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.

1.5.3 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.

1.5.4 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*.

1.6 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.

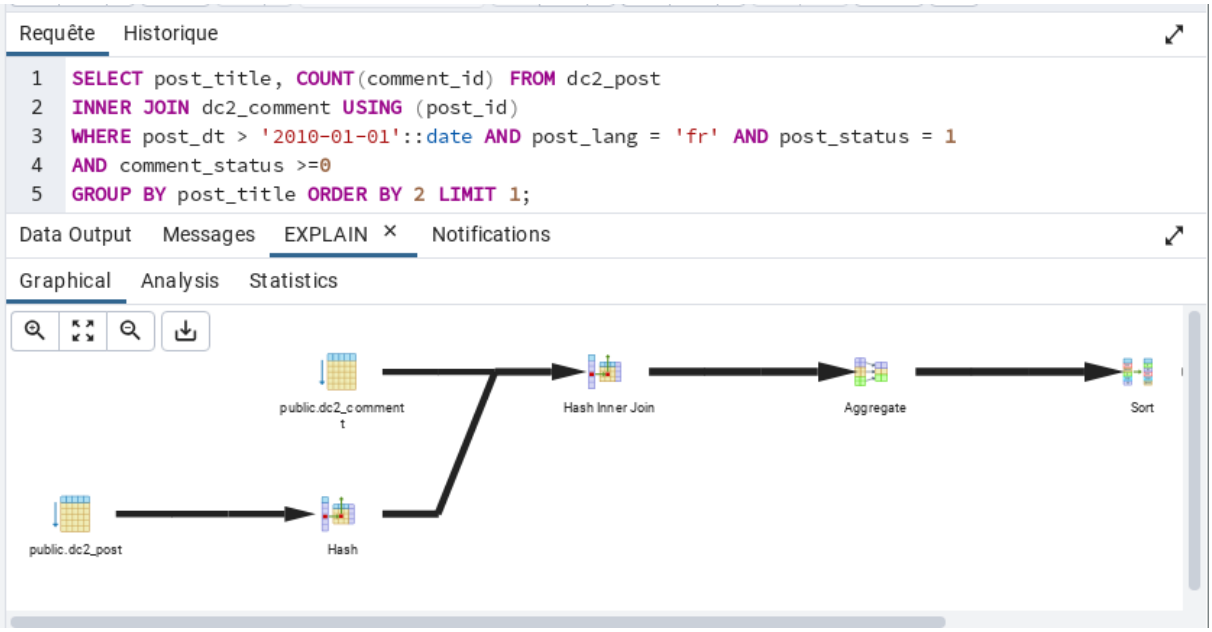
1.6.1 pgAdmin

— 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.

1.6.2 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.

1.6.3 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, explain.depesz.com[5].

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[6]

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

— la partie site web[7]

1.6.4 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[8].

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.

1.6.5 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

[7] https://gitlab.com/depesz/explain.depesz.com
[8] https://explain.depesz.com/

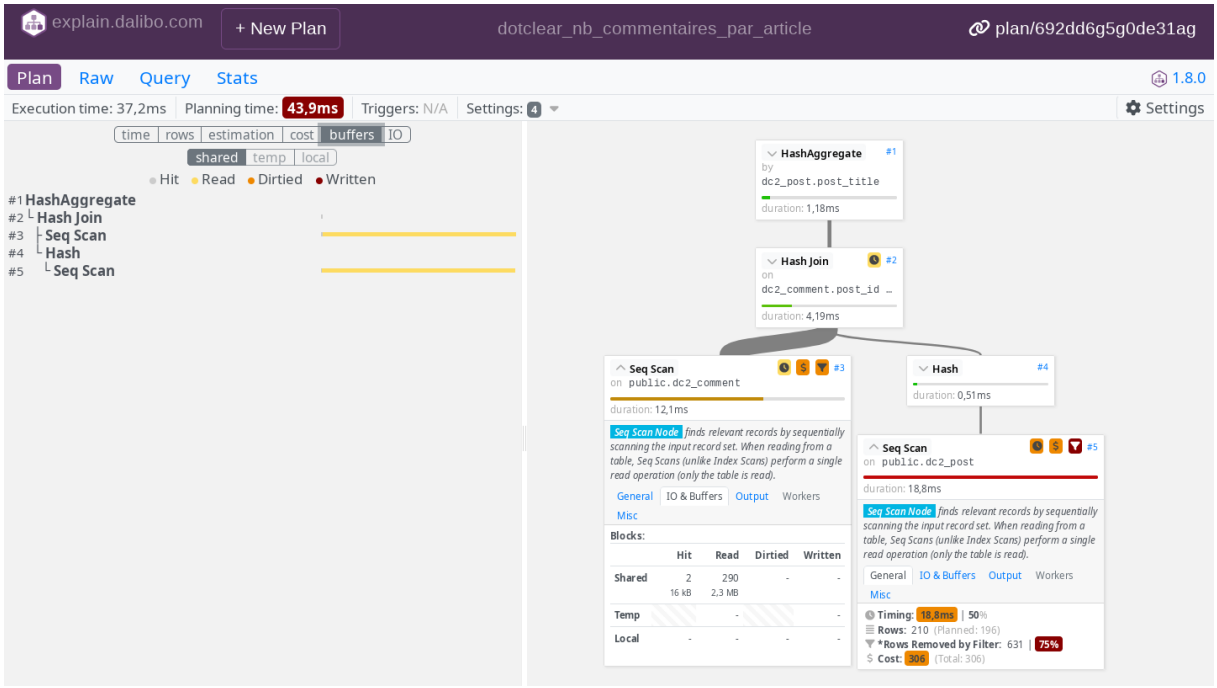
À l'origine, pev (*PostgreSQL Explain Visualizer*) est un outil libre[9] offrant un affichage graphique du plan d'exécution et pointant le nœud le plus coûteux, le plus long, le plus volumineux, etc. Utilisable en ligne[10], il n'est hélas plus maintenu depuis plusieurs années.

explain.dalibo.com[11] 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 locale de pev2[12], qui s'exécute uniquement dans votre navigateur.

Le code[13] est sous licence PostgreSQL. Techniquement, c'est un composant VueJS qui peut être intégré à vos propres outils.

1.6.6 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.

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

1.7 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.

1.7.1 Questions

N'hésitez pas, c'est le moment !

1.8 QUIZ

https://dali.bo/j0_quiz

1.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/j0_solutions.

Tous les TP se basent sur la configuration par défaut de PostgreSQL, sauf précision contraire.

1.9.1 Manipuler explain

But : Première manipulation d'EXPLAIN

Créer une base **machines** et y générer les données comme indiqué ci-dessous. L'exécution peut durer une minute ou deux suivant la machine.

```
curl -kL https://dali.bo/tp_machines_donnees -o machines_donnees.sql
createdb machines
psql machines < machines_donnees.sql
```

La base **machines** contiendra alors deux tables :

— `machines` est une liste de machines ;
— `donnees` contient des données horodatées de quelques capteurs de ces machines, entre janvier et août 2023.

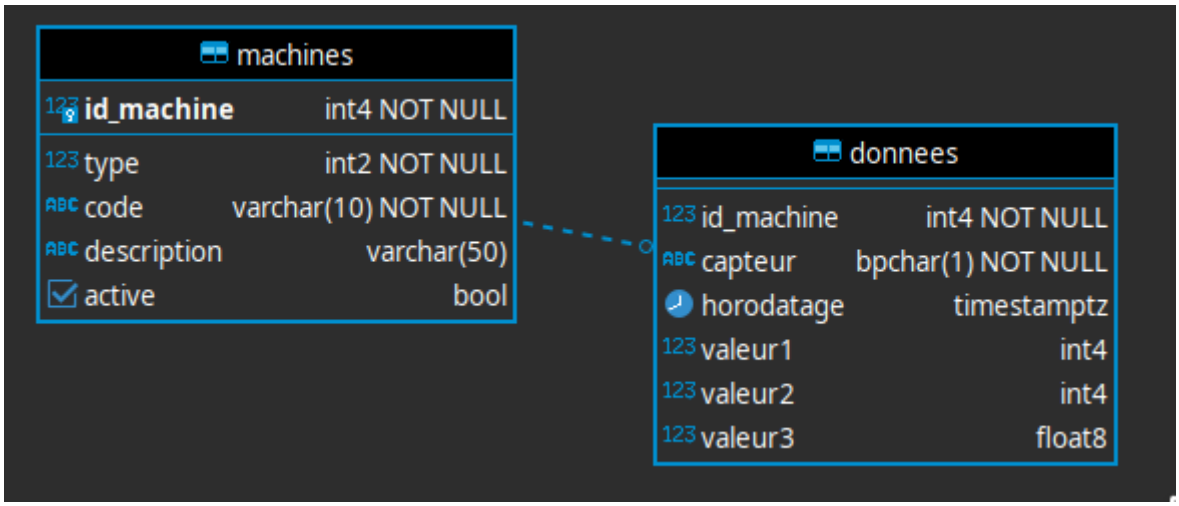


FIGURE 1/ .1 – Tables machines et donnees

Nettoyage et mise à jour des statistiques :

```
VACUUM ANALYZE machines,donnees;
```

Quelles sont les tailles des tables ?

Pour simplifier certains plans, désactivons le parallélisme et la compilation à la volée :

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

Requêtes sur les périodes :

Quel est le plan prévu pour récupérer les données du 31 janvier dans la table `donnees` ?

```
EXPLAIN
SELECT * FROM donnees
WHERE horodatage = '2023-01-31'::date ;
```

Quel est le plan prévu pour récupérer les données du mois de janvier dans la table `donnees` ?

```
EXPLAIN
SELECT * FROM donnees
WHERE horodatage BETWEEN '2023-01-01'::date AND '2023-01-31'::date ;
```

Quel est le plan prévu pour cette variante de la requête sur janvier ?

```
EXPLAIN
SELECT * FROM donnees
WHERE to_char (horodatage, 'YYYYMM') = '202301';
```

Pourquoi est-il différent ? Comparer avec le précédant en utilisant `EXPLAIN ANALYZE`.

Quel est le plan pour la même requête, cette fois sur deux mois ?

```
EXPLAIN
SELECT * FROM donnees
WHERE horodatage BETWEEN '2023-03-01'::date AND '2023-04-30'::date;
```

Relancer avec `EXPLAIN (ANALYZE)`.

Jointure :

Quel est le plan prévu pour cette jointure sur toutes les données d'une machine ?

```
EXPLAIN
SELECT *
FROM donnees INNER JOIN machines USING (id_machine)
WHERE machines.code = 'E4DA3B' AND type = 5;
```

Quel est le plan prévu pour la requête suivante, qui récupère toutes les données d'après juillet pour un type de machines donné ? Quelles en sont les 3 étapes ?

```
EXPLAIN
SELECT description, horodatage, valeur1
FROM donnees INNER JOIN machines USING (id_machine)
WHERE machines.type = 1
AND donnees.horodatage > '2023-07-01' ;
```

1.9.2 Manipuler explain (base magasin)

But : Manipuler explain.

— Créer une base de données nommée `magasin`.

— Importer le jeu de données d'exemple :

La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

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

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

— Le schéma à utiliser se nomme également `magasin`.
— Consulter les tables.

— Lancer un `ANALYZE` sur la base.

Le but est de chercher une personne nommée Moris Russel dans la table `contacts` par les champs `prenom` et `nom`.

— Quel est le plan qu'utilisera PostgreSQL pour le trouver ?
— À combien de résultats le planificateur s'attend-il ?

— Afficher le résultat.

— Quel est le plan réellement exécuté ?

— Rechercher la même personne par son `contact_id`.
— Quel est le plan ?

— La requête suivante recherche tous les fournisseurs résidents d'Hollywood.

```
SELECT c.nom, c.prenom FROM contacts c
INNER JOIN fournisseurs f
ON (f.contact_id = c.contact_id)
WHERE c.ville = 'Hollywood' ;
```

— Quel est le plan prévu ?
— Que donne-t-il à l'exécution ?

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

1.10.1 Manipuler explain

Créer une base **machines** et y générer les données comme indiqué ci-dessous. L'exécution peut durer une minute ou deux suivant la machine.

```
curl -kL https://dali.bo/tp_machines_donnees -o machines_donnees.sql
createdb machines
psql machines < machines_donnees.sql
```

La base **machines** contiendra alors deux tables :

— `machines` est une liste de machines ;
— `donnees` contient des données horodatées de quelques capteurs de ces machines, entre janvier et août 2023.

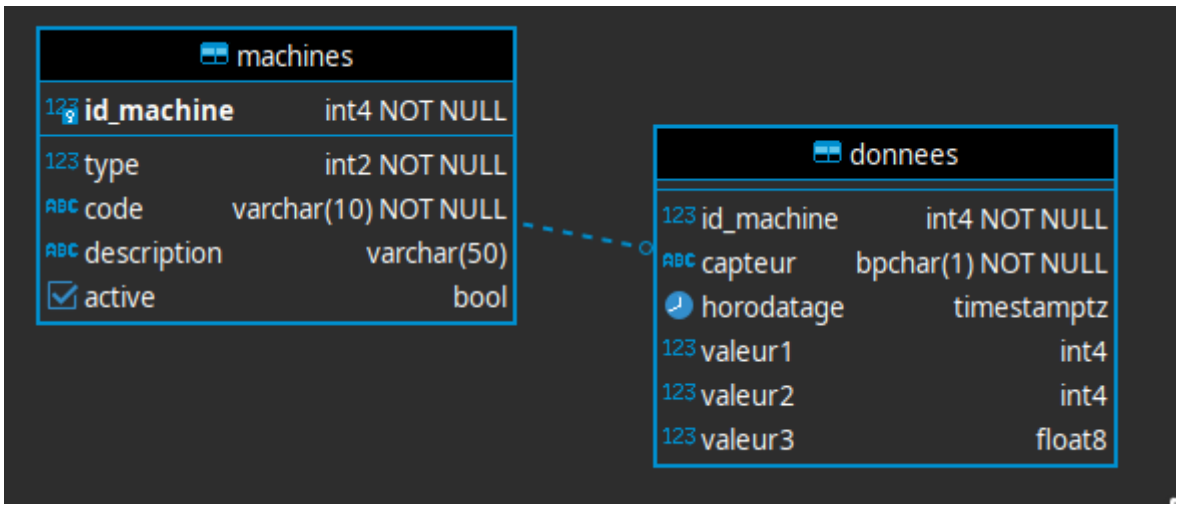


FIGURE 1/ .2 – Tables machines et donnees

Nettoyage et mise à jour des statistiques :

```
VACUUM ANALYZE machines,donnees;
```

Cette opération est à faire systématiquement sur des tables récentes, ou au moindre doute. L'autovacuum n'est parfois pas assez rapide pour effectuer ces opérations.

Quelles sont les tailles des tables ?

Sous `psql` :

```
=# \dt+
                                                 Liste des relations
 Schéma |       Nom       | Type  | Propriétaire |  …  | Taille  | …
--------+-----------------+-------+--------------+-----+---------+--
 public | donnees         | table | postgres     |     | 284 MB  |
 public | machines        | table | postgres     |     | 112 kB  |
```

Quant aux nombres de lignes :

```
SELECT count (*) FROM machines ;

 count
-------
  1000

SELECT count (*) FROM donnees ;

  count
---------
 4950225
```

Tout plan d'exécution dépend de la configuration de PostgreSQL. Sauf précision contraire, nous partons toujours de la configuration par défaut.

Pour simplifier certains plans, désactivons le parallélisme et la compilation à la volée :

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

Requêtes sur les périodes :

Quel est le plan prévu pour récupérer les données du 31 janvier dans la table `donnees` ?

```
EXPLAIN
SELECT * FROM donnees
WHERE horodatage = '2023-01-31'::date ;
```

Le plan prévu est :

```
                                QUERY PLAN
---------------------------------------------------------------------------
Index Scan using donnees_horodatage_idx on donnees (cost=0.43..8.64 rows=12 width=30)
  Index Cond: (horodatage = '2023-01-31'::date)
```

Il existe un index sur le critère, il est naturel qu'il soit utilisé.

Quel est le plan prévu pour récupérer les données du mois de janvier dans la table `donnees` ?

```
EXPLAIN
```

```
SELECT * FROM donnees
WHERE horodatage BETWEEN '2023-01-01'::date AND '2023-01-31'::date ;
```

Le plan prévu est le même, au critère près :

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using donnees_horodatage_idx on donnees  (cost=0.43..933.87 rows=28722
↪  width=30)
   Index Cond: ((horodatage >= '2023-01-01'::date) AND (horodatage <=
↪  '2023-01-31'::date))
```

Noter la réécriture du `BETWEEN` sous forme d'inégalités.

> Quel est le plan prévu pour cette variante de la requête sur janvier ?
>
> ```
> EXPLAIN
> SELECT * FROM donnees
> WHERE to_char (horodatage, 'YYYYMM') = '202301';
> ```
>
> Pourquoi est-il différent ? Comparer avec le précédant en utilisant `EXPLAIN ANALYZE`.

Le plan cette fois est un parcours de table. L'index est ignoré, toute la table est lue :

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on donnees  (cost=0.00..110652.25 rows=24751 width=30)
   Filter: (to_char(horodatage, 'YYYYMM'::text) = '202301'::text)
```

Si le parallélisme est activé, il existe une variante parallélisée de ce plan :

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..70812.96 rows=24751 width=30)
   Workers Planned: 2
   ->  Parallel Seq Scan on donnees  (cost=0.00..67337.86 rows=10313 width=30)
         Filter: (to_char(horodatage, 'YYYYMM'::text) = '202301'::text)
```

La raison du changement de plan est le changement du critère. C'est évident pour un humain, mais PostgreSQL ne fait pas l'équivalence entre les deux formulations du critère sur le mois de janvier. Or il n'y a pas d'index sur la fonction `to_char(horodatage, 'YYYYMM')` (il serait possible d'en créer un).

Si l'on compare les deux plans en les exécutant réellement, avec `EXPLAIN (ANALYZE)`, on obtient pour la variante avec `BETWEEN` :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM donnees
WHERE horodatage BETWEEN '2023-01-01'::date AND '2023-01-31'::date ;

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

```
 Index Scan using donnees_horodatage_idx on donnees  (cost=0.43..933.75 rows=28716
↪  width=30) (actual time=0.060..9.405 rows=19600 loops=1)
   Index Cond: ((horodatage >= '2023-01-01'::date) AND (horodatage <=
↪  '2023-01-31'::date))
   Buffers: shared hit=6 read=191
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.072 ms
 Execution Time: 10.472 ms
```

et pour la variante avec `to_char` :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM donnees
WHERE to_char (horodatage, 'YYYYMM') = '202301';
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on donnees  (cost=0.00..110652.25 rows=24751 width=30) (actual
↪  time=0.013..1503.631 rows=19600 loops=1)
   Filter: (to_char(horodatage, 'YYYYMM'::text) = '202301'::text)
   Rows Removed by Filter: 4930625
   Buffers: shared hit=16063 read=20336
 Planning Time: 0.025 ms
 Execution Time: 1504.379 ms
```

La dernière ligne indique 10 ms pour la variante avec `BETWEEN` contre 1,5 s pour la variante avec `to_char` : l'utilisation de l'index est nettement plus intéressante que le parcours complet de la table. Le plan indique aussi que beaucoup plus de blocs (*buffers*) ont été lus.

> Quel est le plan pour la même requête, cette fois sur deux mois ?
>
> ```
> EXPLAIN
> SELECT * FROM donnees
> WHERE horodatage BETWEEN '2023-03-01'::date AND '2023-04-30'::date;
> ```
>
> Relancer avec `EXPLAIN (ANALYZE)`.

On s'attend au même plan que pour la recherche sur janvier, mais PostgreSQL prévoit cette fois un parcours complet :

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on donnees  (cost=0.00..110652.25 rows=4184350 width=30)
   Filter: ((horodatage >= '2023-03-01'::date) AND (horodatage <=
↪  '2023-04-30'::date))
```

En effet, il y a beaucoup plus de lignes à récupérer sur mars-avril qu'en janvier. La mention `rows` indique l'estimation des lignes ramenées et indique 4,2 millions de lignes sur les 4,9 de la table ! Le plus efficace est donc de lire directement la table. Les statistiques permettent donc à PostgreSQL de changer de stratégie suivant les volumétries attendues.

Une exécution réelle indique que cette estimation est bonne, et dure logiquement à peu près aussi longtemps que le parcours complet ci-dessus :

```
EXPLAIN (ANALYZE)
SELECT * FROM donnees
WHERE horodatage BETWEEN '2023-03-01'::date AND '2023-04-30'::date ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on donnees  (cost=0.00..110652.25 rows=4184350 width=30) (actual
↪  time=160.385..1255.020 rows=4182160 loops=1)
   Filter: ((horodatage >= '2023-03-01'::date) AND (horodatage <=
↪  '2023-04-30'::date))
   Rows Removed by Filter: 768065
 Planning Time: 0.470 ms
 Execution Time: 1378.383 ms
```

Jointure :

> Quel est le plan prévu pour cette jointure sur toutes les données d'une machine ?
>
> ```
> EXPLAIN
> SELECT *
> FROM donnees INNER JOIN machines USING (id_machine)
> WHERE machines.code = 'E4DA3B' AND type = 5;
> ```

Le plan est :

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.71..5033.11 rows=4950 width=75)
   ->  Index Scan using machines_type_code_key on machines  (cost=0.28..8.29 rows=1
↪  width=49)
         Index Cond: ((type = 5) AND ((code)::text = 'E4DA3B'::text))
   ->  Index Scan using donnees_id_machine_idx on donnees  (cost=0.43..4124.77
↪  rows=90004 width=30)
         Index Cond: (id_machine = machines.id_machine)
```

Il s'agit :

- d'un accès à `machines` par l'index sur `machines (type, code)` (cet index marque l'unicité) ;
- suivi d'un accès à `donnees`, toujours par l'index sur le champ indexé `id_machine`.

> Quel est le plan prévu pour la requête suivante, qui récupère toutes les données d'après juillet pour un type de machines donné ? Quelles en sont les 3 étapes ?
>
> ```
> EXPLAIN
> SELECT description, horodatage, valeur1
> FROM donnees INNER JOIN machines USING (id_machine)
> WHERE machines.type = 1
> AND donnees.horodatage > '2023-07-01' ;
> ```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
Hash Join  (cost=30.67..8380.56 rows=138788 width=47)
   Hash Cond: (donnees.id_machine = machines.id_machine)
   ->  Index Scan using donnees_horodatage_idx on donnees  (cost=0.43..7671.54
↪  rows=257492 width=16)
```

```
      Index Cond: (horodatage > '2023-07-01 00:00:00+02'::timestamp with time zone)
->  Hash  (cost=23.50..23.50 rows=539 width=39)
      ->  Seq Scan on machines  (cost=0.00..23.50 rows=539 width=39)
            Filter: (type = 1)
```

Il s'agit ici d'une jointure en *hash join*, courante dans les jointures brassant beaucoup de lignes.

PostgreSQL commence par un parcours complet de `machines` (`type = 1` concerne la plupart des machines). Puis il crée une « table de hachage » à partir des `id_machine` des lignes résultantes. Il parcoure `donnees` en se basant sur l'index sur la date. Les lignes résultantes seront comparées au contenu de la table de hachage pour savoir s'il faut garder les valeurs.

1.10.2 Manipuler explain (base magasin)

> — Créer une base de données nommée `magasin`.

Si l'on est connecté à la base, en tant que superutilisateur **postgres** :

```
CREATE DATABASE magasin;
```

Alternativement, depuis le shell, en tant qu'utilisateur système **postgres** :

```
postgres$ createdb --echo  magasin
```

```
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE magasin;
```

> — Importer le jeu de données d'exemple :

La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

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

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

> — Le schéma à utiliser se nomme également `magasin`.
> — Consulter les tables.

Le schéma par défaut `public` ne contient effectivement aucune table intéressante.

```
\dn
     Liste des schémas
     Nom     | Propriétaire
-------------+--------------
 facturation | postgres
 magasin     | postgres
 public      | postgres
```

```
SET search_path to magasin ;
```

```
 \dt+
                                    Liste des relations
 Schéma  |         Nom          | Type  | Propriétaire | Persistence |   Taille   | D…
---------+----------------------+-------+--------------+-------------+------------+-
 ↪  --
```

```
 magasin | clients              | table | postgres | permanent | 8248 kB    |
 magasin | commandes             | table | postgres | permanent | 79 MB      |
 magasin | conditions_reglement  | table | postgres | permanent | 16 kB      |
 magasin | contacts              | table | postgres | permanent | 24 MB      |
 magasin | etats_retour          | table | postgres | permanent | 16 kB      |
 magasin | fournisseurs          | table | postgres | permanent | 840 kB     |
 magasin | lignes_commandes      | table | postgres | permanent | 330 MB     |
 magasin | lots                  | table | postgres | permanent | 74 MB      |
 magasin | modes_expedition      | table | postgres | permanent | 16 kB      |
 magasin | modes_reglement       | table | postgres | permanent | 16 kB      |
 magasin | numeros_sequence      | table | postgres | permanent | 16 kB      |
 magasin | pays                  | table | postgres | permanent | 16 kB      |
 magasin | pays_transporteurs    | table | postgres | permanent | 8192 bytes |
 magasin | produit_fournisseurs  | table | postgres | permanent | 216 kB     |
 magasin | produits              | table | postgres | permanent | 488 kB     |
 magasin | regions               | table | postgres | permanent | 16 kB      |
 magasin | transporteurs         | table | postgres | permanent | 16 kB      |
 magasin | types_clients         | table | postgres | permanent | 16 kB      |
```

Conseils pour la suite :

— Préciser `\timing on` dans `psql` pour afficher les temps d'exécution de la recherche.
— Pour rendre les plans plus lisibles, désactiver le JIT et le parallélisme :

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

— Lancer un `ANALYZE` sur la base.

```
ANALYZE ;
```

Le but est de chercher une personne nommée Moris Russel dans la table `contacts` par les champs `prenom` et `nom`.

— Quel est le plan qu'utilisera PostgreSQL pour le trouver ?
— À combien de résultats le planificateur s'attend-il ?

```
 EXPLAIN SELECT * FROM contacts WHERE nom ='Russel'  AND prenom = 'Moris' ;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on contacts  (cost=0.00..4693.07 rows=1 width=298)
   Filter: (((nom)::text = 'Russel'::text) AND ((prenom)::text = 'Moris'::text))
```

La table sera entièrement parcourue (*Seq Scan*). PostgreSQL pense qu'il trouvera une ligne.

— Afficher le résultat.

```
 SELECT * FROM contacts WHERE nom ='Russel' AND prenom = 'Moris' ;
```

```
-[ RECORD 1 ]----------------------------------------
contact_id  | 26452
login       | Russel_Moris
passwd      | 9f81a90c36dd3c60ff06f3c800ae4c1b
email       | ubaldo@hagenes-kulas-and-oberbrunner.mo
```

```
nom         | Russel
prenom      | Moris
adresse1    | 02868 Norris Greens
adresse2    | ¤
code_postal | 62151
ville       | Laguna Beach
code_pays   | CA
telephone   | {"+(05) 4.45.08.11.03"}

Temps : 34,091 ms
```

La requête envoie bien une ligne, et l'obtenir a pris 34 ms sur cette machine avec SSD.

> — Quel est le plan réellement exécuté ?

Il faut relancer la requête :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM contacts
  WHERE nom ='Russel' AND prenom = 'Moris' ;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on contacts  (cost=0.00..4693.07 rows=1 width=297)
                       (actual time=3.328..16.789 rows=1 loops=1)
   Filter: (((nom)::text = 'Russel'::text) AND ((prenom)::text = 'Moris'::text))
   Rows Removed by Filter: 110004
   Buffers: shared hit=3043
 Planning Time: 0.052 ms
 Execution Time: 16.848 ms
```

PostgreSQL a à nouveau récupéré une ligne. Ici, cela n'a pris que 17 ms.

La table a été parcourue entièrement, et 110 004 lignes ont été rejetées. La ligne *shared hit* indique que 3043 blocs de 8 ko ont été lus dans le cache de PostgreSQL. La requête précédente a apparemment suffi à charger la table entière en cache (il n'y a pas de *shared read*).

> — Rechercher la même personne par son `contact_id`.
> — Quel est le plan ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM contacts WHERE contact_id = 26452 ;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using contacts_pkey on contacts (cost=0.42..8.44 rows=1 width=297)
                                    (actual time=0.057..0.058 rows=1 loops=1)
   Index Cond: (contact_id = 26452)
   Buffers: shared hit=4 read=3
 Planning:
   Buffers: shared hit=6 read=3
 Planning Time: 0.137 ms
 Execution Time: 0.081 ms
```

PostgreSQL estime correctement trouver une ligne. Cette fois, il s'agit d'un *Index Scan*, en l'occurrence sur l'index de la clé primaire. Le résultat est bien plus rapide : 137 µs pour planifier, 81 µs pour exécuter.

Les blocs lus se répartissent entre *read* et *hit* : une partie était en mémoire, notamment ceux liés à la table, puisque la table aussi a été interrogée (l'index ne contient que les données de `contact_id`) ; mais l'index n'était pas en mémoire.

— La requête suivante recherche tous les fournisseurs résidents d'Hollywood.

```
SELECT c.nom, c.prenom FROM contacts c
INNER JOIN fournisseurs f
ON (f.contact_id = c.contact_id)
WHERE c.ville = 'Hollywood' ;
```

— Quel est le plan prévu ?
— Que donne-t-il à l'exécution ?

Le plan simplifié est :

```
EXPLAIN (COSTS OFF)
SELECT c.nom, c.prenom
FROM contacts c INNER JOIN fournisseurs f ON (f.contact_id = c.contact_id)
WHERE c.ville = 'Hollywood' ;
```

```
                     QUERY PLAN
-----------------------------------------------------
 Merge Join
   Merge Cond: (c.contact_id = f.contact_id)
   ->  Index Scan using contacts_pkey on contacts c
         Filter: ((ville)::text = 'Hollywood'::text)
   ->  Sort
         Sort Key: f.contact_id
         ->  Seq Scan on fournisseurs f
```

Il consiste à parcourir intégralement la table `fournisseurs` (*Seq Scan*), à trier sa colonne `contact_id`, et à effectuer une jointure de type *Merge Join* avec la clé primaire de la table `contacts`. En effet, un *Merge Join* s'effectue entre deux ensembles triés : l'index l'est déjà, mais `fournisseurs.contact_id` ne l'est pas.

Noter qu'aucune donnée n'est récupérée de `fournisseurs`. Il est pourtant nécessaire de la joindre à `contacts` car de nombreux contacts ne sont *pas* des fournisseurs.

Exécutée, cette requête renvoie le plan suivant :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT c.nom, c.prenom FROM contacts c
INNER JOIN fournisseurs f ON (f.contact_id = c.contact_id)
WHERE c.ville = 'Hollywood' ;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=864.82..1469.89 rows=31 width=14)
             (actual time=5.079..11.063 rows=32 loops=1)
   Merge Cond: (c.contact_id = f.contact_id)
   Buffers: shared hit=7 read=464
   ->  Index Scan using contacts_pkey on contacts c
                                  (cost=0.42..6191.54 rows=346 width=22)
```

```
                                    (actual time=0.029..4.842 rows=33 loops=1)
        Filter: ((ville)::text = 'Hollywood'::text)
        Rows Removed by Filter: 11971
        Buffers: shared hit=7 read=364
  ->  Sort  (cost=864.39..889.39 rows=10000 width=8)
            (actual time=5.044..5.559 rows=10000 loops=1)
        Sort Key: f.contact_id
        Sort Method: quicksort  Memory: 853kB
        Buffers: shared read=100
        ->  Seq Scan on fournisseurs f (cost=0.00..200.00 rows=10000 width=8)
                                       (actual time=0.490..2.960 rows=10000 loops=1)
              Buffers: shared read=100
Planning:
  Buffers: shared hit=4
Planning Time: 0.150 ms
Execution Time: 11.174 ms
```

Ce plan est visible graphiquement sur https://explain.dalibo.com/plan/dum :

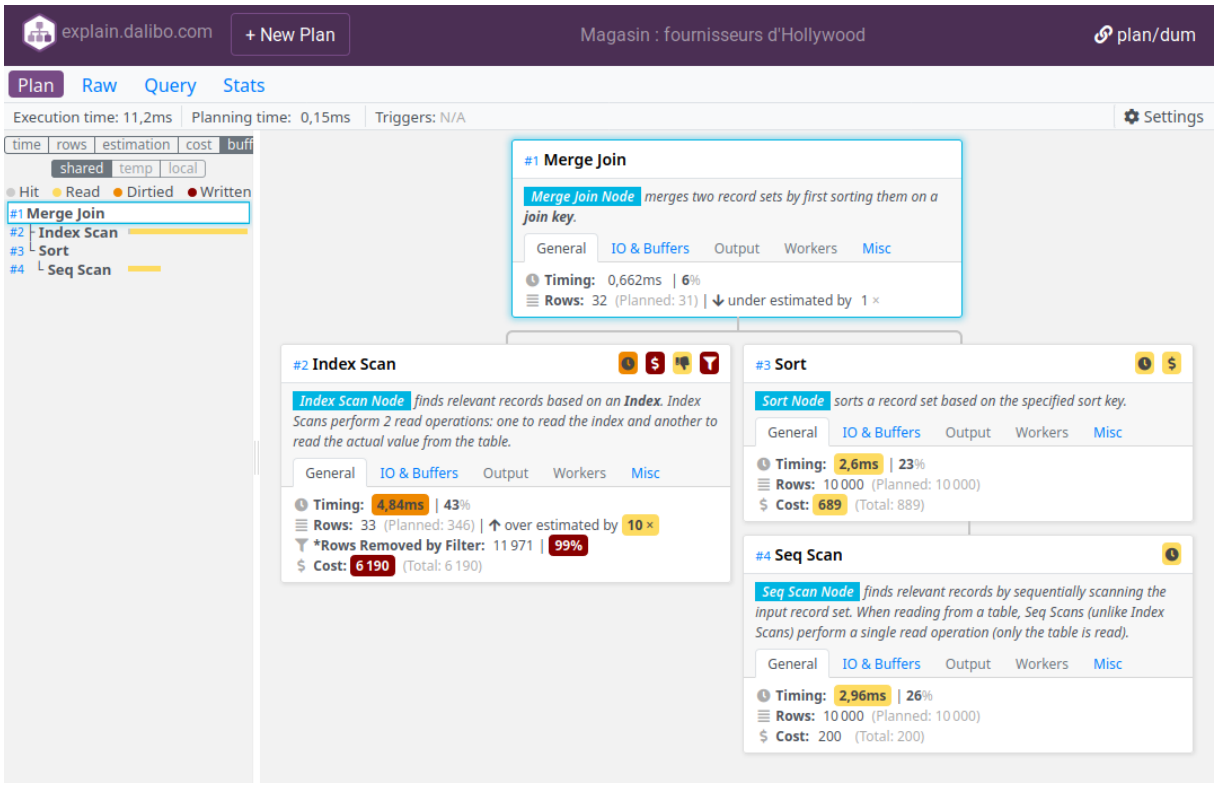


FIGURE 1/ .3 – Plan d'exécution

Le *Seq Scan* sur `fournisseurs` lit 10 000 lignes (100 blocs, hors du cache), ce qui était prévu. Cela prend 2,96 ms. Le nœud *Sort* trie les `contact_id` et consomme 853 ko en mémoire. Il renvoie bien sûr aussi 10 000 lignes, et il commence à le faire au bout de 5,04 ms.

La jointure peut commencer. Il s'agit de parcourir simultanément l'ensemble que l'on vient de trier

d'une part, et l'index `contacts_pkey` d'autre part. À cette occasion, le nœud *Index Scan* va filtrer les lignes récupérées en comparant à la valeur de `ville`, et en exclue 11 971. Au final, le parcours de l'index sur `contacts` renvoie 33 lignes, et non les 346 estimées au départ (valeur dérivée de l'estimation du nombre de lignes où la ville est « Hollywood »). Si l'on regarde les coûts calculés, c'est cette étape qui est la plus lourde (6191).

En haut, on peut lire qu'au total 464 blocs ont été lus hors du cache, et 7 dedans. Ces valeurs varient bien sûr en fonction de l'activité précédente sur la base. Au final, 32 lignes sont retournées, ce qui était attendu.

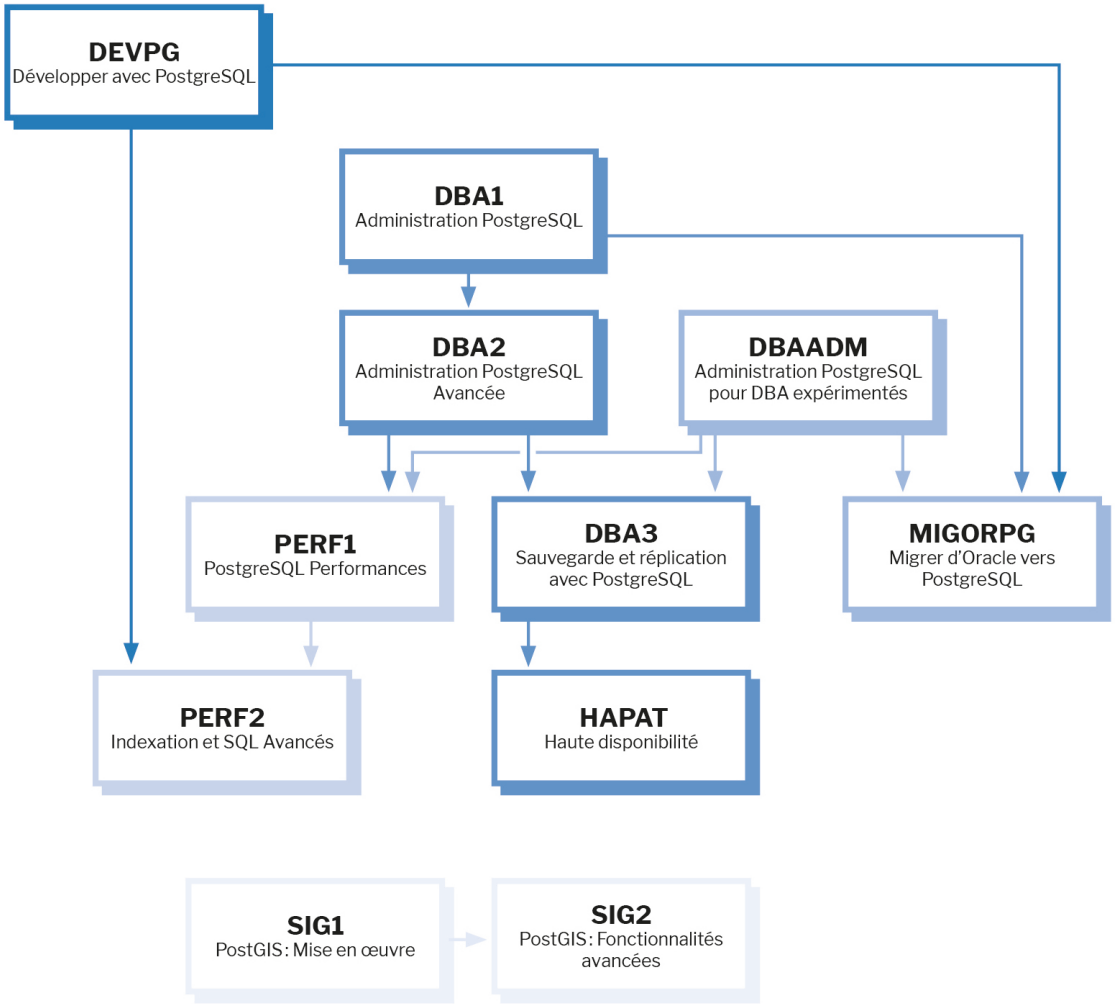
Le temps écoulé est de 11,17 ms. La majorité de ce temps s'est déroulé pendant le *Merge Join* (11,0-5,0 = 6 ms), dont l'essentiel est constitué par le parcours de l'index.

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.