Module J0

Introduction à EXPLAIN

22.09

Dalibo SCOP

https://dalibo.com/formations

Introduction à EXPLAIN

Module J0

TITRE : Introduction à EXPLAIN
SOUS-TITRE : Module J0

REVISION: 22.09
DATE: 02 septembre 2022

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, 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, 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.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

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

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

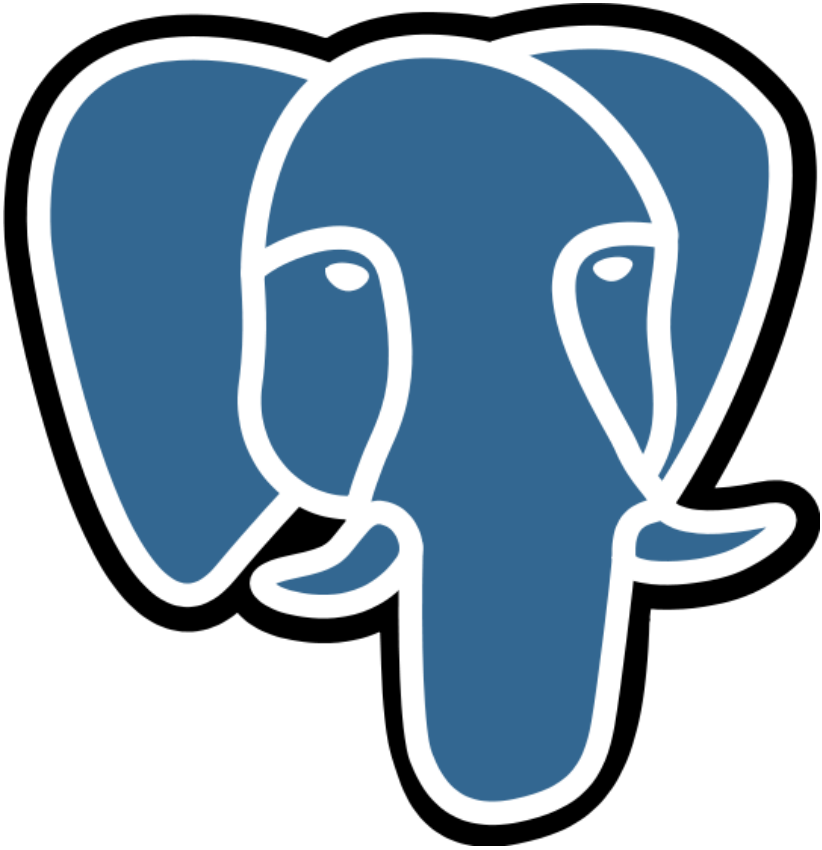
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

Table des Matières

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 rapide de l'optimiseur et des plans d'exécution. Il contient aussi 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

1.2 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.

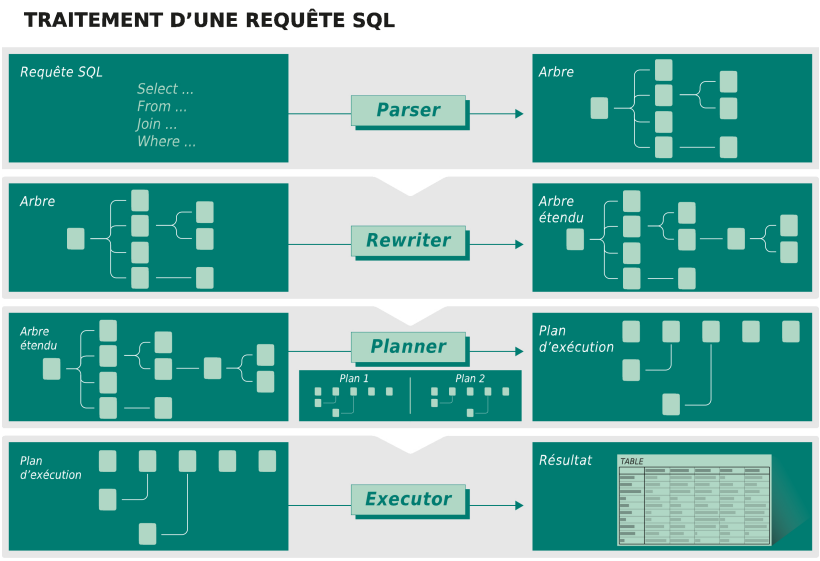
1.2.1 NIVEAU SYSTÈME

- 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.).

1.2.2 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.

1.2.3 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.

1.3 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

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.

1.3.1 PRINCIPE

- Chargé de sélectionner le meilleur plan d'exécution
- Énumère tous les plans d'exécution
  - ou presque tous…
- Statistiques + configuration + règles => coût
- 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).

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.

1.3.2 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.

1.3.3 DÉCISIONS

- Stratégie d'accès aux lignes
  - parcours de table, d'index, de fonction, etc.
- Stratégie d'utilisation des jointures
  - ordre
  - type
- Stratégie d'agrégation
  - brut, trié, haché…

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

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.

1.4 MÉCANISME DE CALCUL DE COÛTS

- Modèle basé sur les coûts
  - quantifier la charge pour répondre à une requête
- Chaque opération a un coût :
  - lire un bloc selon sa position sur le disque
  - manipuler une ligne issue d'une lecture de table ou d'index
  - appliquer un opérateur

L'optimiseur statistique de PostgreSQL utilise un modèle de calcul de coût. Les coûts calculés sont des indications arbitraires 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.

1.4.1 COÛTS UNITAIRES

- Coûts à connaître :
  - accès au disque séquentiel / non séquentiel
  - traitement d'un enregistrement issu d'une table / d'un index
  - application d'un opérateur
  - traitement d'un enregistrement dans un parcours parallélisé
  - etc.
- 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` (par défaut 1) et `random_page_cost` (par défaut 4) représentent les coûts relatifs d'un accès séquentiel à un bloc sur le disque (c'est-à-dire lu en même temps que ses voisins dans la table) ou 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` (par défaut 0,01), `cpu_index_tuple_cost` (par défaut 0,005) et `cpu_operator_cost` représentent le coût relatif de la manipulation d'une ligne en mémoire, d'une donnée issue d'un index, ou de l'application d'un opérateur sur une donnée.

`parallel_setup_cost` (par défaut 1000) indique le coût de mise en place d'un parcours parallélisé, une procédure assez lourde qui ne se rentabilise pas pour les petites requêtes.

La compilation à la volée des requêtes (JIT ou *Just In Time*) est un processus d'optimisation coûteux. Les paramètres suivants permettent de définir à partir de quel coût les différents niveaux d'optimisation peuvent être déclenchés : `jit_above_cost` (par défaut 100 000), `jit_inline_above_cost` (par défaut 500 000), `jit_optimize_above_cost` (par défaut 500 000).

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 bons disques.

Pour des besoins particuliers, ces paramètres sont des paramètres de sessions. Ils peuvent être modifiés dynamiquement avec l'ordre `SET` au niveau de l'application en vue d'exécuter des requêtes bien particulières.

1.4.2 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`
- => Stratégie d'accès aux données
- Sans bonnes statistiques, pas de bons plans !

Le planificateur se base principalement sur la configuration des paramètres de coût, mais il a aussi besoin de statistiques sur les données pour prendre de bonnes décisions.

En effet, 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.

Avec ces informations et le paramétrage, l'optimiseur saura par exemple calculer le ratio

d'un filtre et s'il faut passer par un index, ou le ratio d'une jointure pour choisir la stratégie de jointure.

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 !

1.4.3 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 la colonne `c1` (de 1 à 1000). 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.

1.4.4 EXEMPLE - PARCOURS DE TABLE

```
UPDATE t1 SET c1=1;

ANALYZE t1;

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 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.

1.4.5 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)
SET enable_seqscan TO on;
```

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. Pour cela, 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.

1.5 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.

1.5.1 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.

1.5.2 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.

1.5.3 EXEMPLE DE REQUÊTES

```
DROP INDEX t1_c1_idx;
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.

1.5.4 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`).

1.5.5 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.

1.5.6 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.

1.5.7 OPTION ANALYZE

```
EXPLAIN (ANALYZE) 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).

1.5.8 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.

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`).

`BUFFERS` n'affiche que des données réelles, pas des estimations, et implique d'utiliser aussi `ANALYZE`. (À partir de la version 13, `EXPLAIN (BUFFERS)` fonctionne, mais n'affiche que les quelques blocs utilisés par la planification, pas tous ceux auxquels la requête accéderaient vraiment.)

1.5.9 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=0.15..37.65 rows=9 width=8)
   Filter: (c2 < 10)
 Settings: enable_seqscan = 'off', work_mem = '100MB'
RESET enable_seqscan;
RESET work_mem;
```

Désactivée par défaut, cette option permet d'obtenir les valeurs des paramètres spécifiques à l'optimisation de requêtes qui ne sont pas à leur valeur par défaut.

1.5.10 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`, cette option permet d'obtenir le nombre d'enregistrements et le nombre d'octets écrits dans les journaux de transactions.

1.5.11 AUTRES OPTIONS

- `COSTS` : affichage des coûts
- `TIMING` : activation du chronométrage et affichage des informations vues/calculées par l'optimiseur
- `VERBOSE` : affichage verbeux (schémas, colonnes, workers)
- `SUMMARY` : affichage du temps de planification et exécution (si applicable)
- `FORMAT` : format de sortie (texte, XML, JSON, 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

Cette option apparaît en version 10. Elle permet d'afficher ou non le résumé final indiquant la durée de la planification et de l'exécution. Un `EXPLAIN` simple n'affiche pas le résumé par défaut. Par contre, un `EXPLAIN ANALYZE` l'affiche par défaut.

```
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 XML, JSON et YAML. Voici ce que donne la commande `EXPLAIN` avec le format XML :

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

                          QUERY PLAN
------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">  +
   <Query>                                                 +
     <Plan>                                                +
       <Node-Type>Sort</Node-Type>                         +
       <Parallel-Aware>false</Parallel-Aware>              +
       <Startup-Cost>21.64</Startup-Cost>                  +
       <Total-Cost>21.67</Total-Cost>                      +
       <Plan-Rows>9</Plan-Rows>                            +
       <Plan-Width>8</Plan-Width>                          +
       <Sort-Key>                                          +
         <Item>c1</Item>                                   +
       </Sort-Key>                                         +
       <Plans>                                             +
         <Plan>                                            +
           <Node-Type>Seq Scan</Node-Type>                 +
           <Parent-Relationship>Outer</Parent-Relationship>+
           <Parallel-Aware>false</Parallel-Aware>          +
           <Relation-Name>t1</Relation-Name>               +
           <Alias>t1</Alias>                               +
           <Startup-Cost>0.00</Startup-Cost>               +
           <Total-Cost>21.50</Total-Cost>                  +
           <Plan-Rows>9</Plan-Rows>                        +
           <Plan-Width>8</Plan-Width>                      +
           <Filter>(c2 &lt; 10)</Filter>                   +
         </Plan>                                           +
       </Plans>                                            +
     </Plan>                                               +
   </Query>                                                +
 </explain>
(1 row)
```

Les formats semi-structurés sont utilisés principalement par des outils, car le contenu est plus facile à analyser, et même un peu plus complet.

1.5.12 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)
   Sort Key: c1
   Sort Method: quicksort  Memory: 26kB
   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 opération permet de vérifier si le temps d'exécution de la requête est passé surtout sur la demande de blocs au système d'exploitation (donc extérieur à PostgreSQL) ou à l'exécution même de la requête (donc interne à PostgreSQL).

1.5.13 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.

1.6 NŒUDS D'EXÉCUTION LES PLUS COURANTS

- 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 autres formations proposées par Dalibo.

1.6.1 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 objets peuvent être parcourus. Chacun va disposer d'un ou 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 colonnes en sortie correspondent aux colonnes de l'index (ce qui permet d'éviter une lecture de certains blocs 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*).

1.6.2 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.

1.6.3 AGRÉGATS

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

1.6.4 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 renvoie 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*.

1.7 OUTILS GRAPHIQUES

- pgAdmin
- OmniDB
- 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.

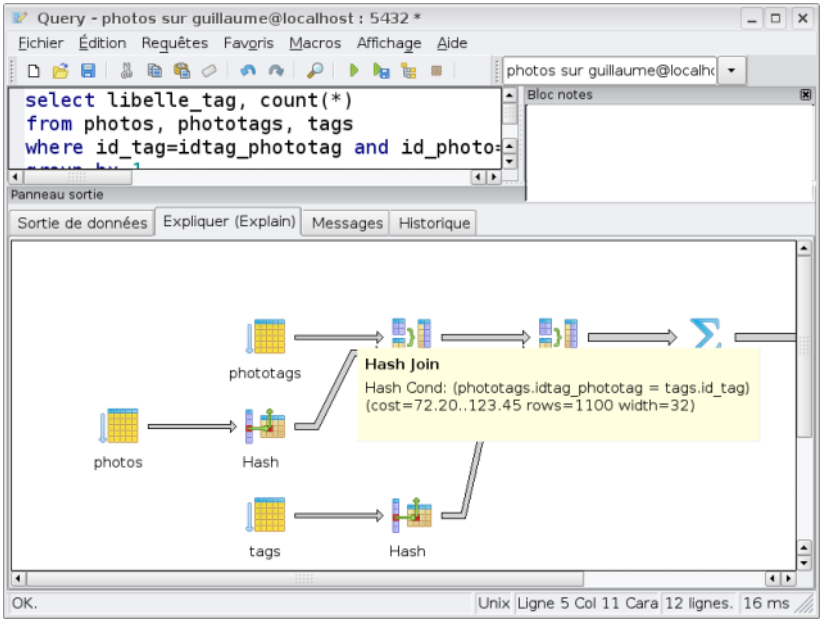
1.7.1 PGADMIN

- 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.

1.7.2 PGADMIN - COPIE D'ÉCRAN



Voici un exemple d'un EXPLAIN graphique réalisé par pgAdmin 3. En passant la souris sur les nœuds, un message affiche les informations statistiques sur le nœud.

pgAdmin 4 dispose aussi d'un tel système.

1.7.3 OMNIDB

- Vision graphique d'un EXPLAIN
- Un affichage textuel amélioré
- Deux affichages graphiques sous forme d'arbre
  - pas de détail de chaque nœud
  - taille du nœud dépendante du coût

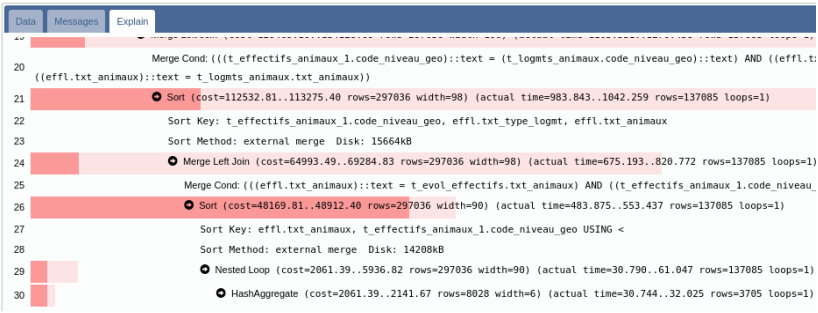
OmniDB est un outil d'administration pour PostgreSQL assez récent. Il propose un affichage graphique de l'EXPLAIN. Cet affichage est intéressant car il montre simplement l'ordre dans lequel les opérations sont effectuées. Chaque nœud est représenté par un élément de l'arbre. Les flèches entre chaque nœud indiquent où sont envoyés les flux de

données, la taille du nœud dépendant de son coût.

Malheureusement, pour l'instant, les statistiques ne sont pas affichées.

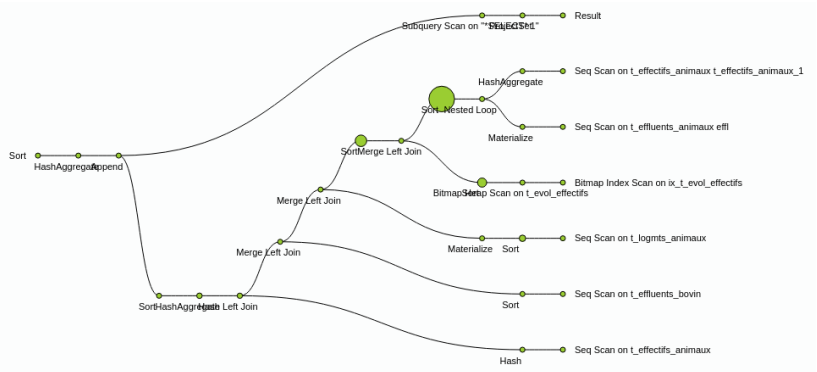
L'affichage texte a un petit plus (une colorisation) permettant de trouver rapidement le nœud le plus coûteux.

1.7.4 OMNIDB - TABLEAU



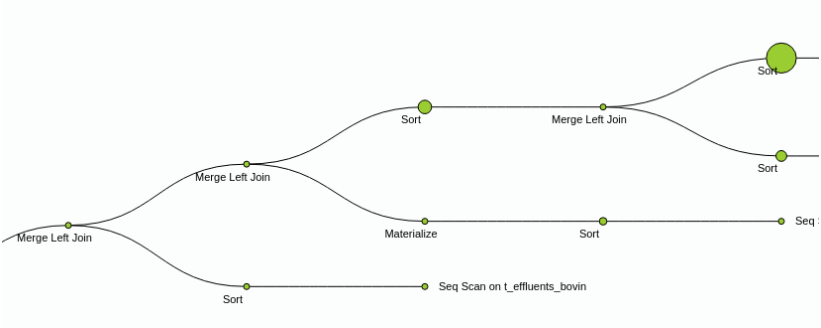
Voici un exemple d'un `EXPLAIN` texte réalisé par OmniDB. Le fond coloré permet de déduire rapidement que le nœud *Sort* est le plus coûteux.

1.7.5 OMNIDB - GRAPHE 1



Ceci est le premier affichage graphique disponible avec OmniDB. Il s'agit d'une représentation sous la forme d'un arbre, un peu comme pgAdmin, les icônes en moins, la taille du nœud en plus.

1.7.6 OMNIDB - GRAPHE 2



Même affichage mais plan différent.

1.7.7 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/

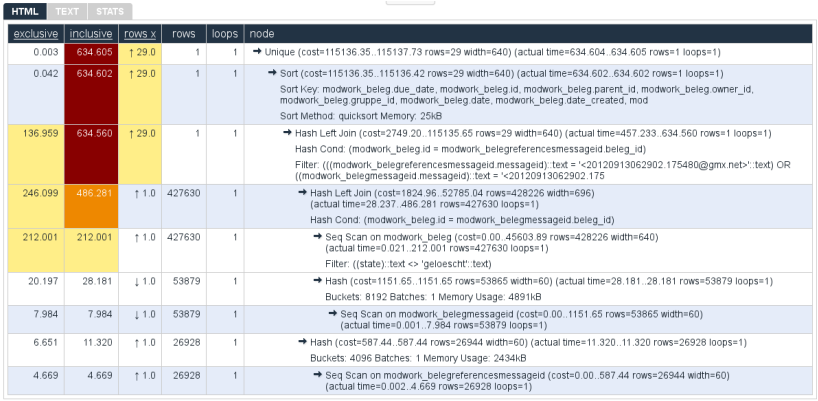
Il suffit d'aller sur ce site, de coller le résultat d'un `EXPLAIN ANALYZE`, et le site affichera le plan d'exécution avec des codes couleurs pour bien distinguer les nœuds performants des autres.

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[2]
- la partie site web[3]

1.7.8 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[4] .

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.

---

[2] https://gitlab.com/depesz/Pg--Explain
[3] https://gitlab.com/depesz/explain.depesz.com
[4] https://explain.depesz.com/

1.7.9 EXPLAIN.DALIBO.COM

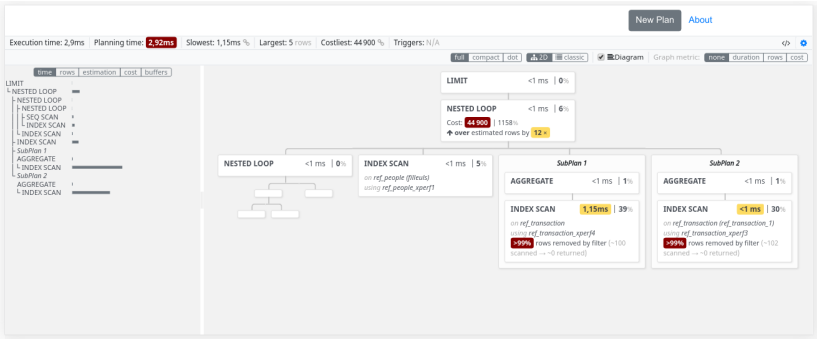
- Reprise de **pev** d'Alex Tatiyants, par Pierre Giraud (Dalibo)
- Page web avec affichage graphique d'un `EXPLAIN [ANALYZE]`
- Repérage des nœuds longs, lourds…
- Affichage flexible
- explain.dalibo.com
- Installable en local

À l'origine, pev (*PostgreSQL Explain Visualizer*) est un outil libre[5] 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[6] , il n'est hélas plus maintenu depuis plusieurs années.

explain.dalibo.com[7] 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[8] .

Le code[9] est sous licence PostgreSQL. Techniquement, c'est un composant VueJS qui peut être intégré à vos propres outils.

1.7.10 EXPLAIN.DALIBO.COM - EXEMPLE



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

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.

1.8 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.

1.8.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.9 QUIZ

https://dali.bo/j0_quiz

1.10 TRAVAUX PRATIQUES

But : Manipuler explain.

> Créer une base de données nommée `magasin`.

> Importer le jeu de données d'exemple :

La base **magasin** peut être téléchargée depuis https://dali.bo/tp_magasin (dump de 96 Mo, pour 667 Mo sur le disque au final). Elle s'importe de manière très classique (une erreur sur le schéma **public** déjà présent est normale), ici dans une base nommée aussi `magasin` :

```
pg_restore -d magasin  magasin.dump
```

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

> 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 ?

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

> 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** peut être téléchargée depuis https://dali.bo/tp_magasin (dump de 96 Mo, pour 667 Mo sur le disque au final). Elle s'importe de manière très classique (une erreur sur le schéma **public** déjà présent est normale), ici dans une base nommée aussi `magasin` :

```
pg_restore -d magasin  magasin.dump
```

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

> 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
(6 lignes)

Temps : 17,247 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
(7 lignes)

Temps : 0,624 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
(7 lignes)
```

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
(17 lignes)

Temps : 11,754 ms
```

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

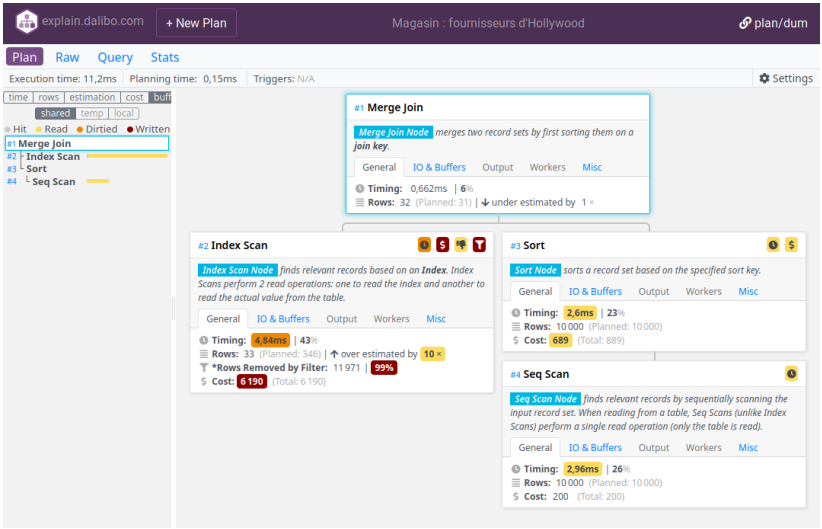


Figure 1: Plan d'exécution

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.

NOTES

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **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

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.