Explain Cheat Sheet

Syntaxte

Syntaxe classique
```
EXPLAIN [ANALYSE] [VERBOSE] requête;
```
Nouvelle syntaxe (9.0 et plus)
```
EXPLAIN [(options, ...)] requête;
```
Avec les options *suivantes :*

`ANALYSE` *Exécute la commande et affiche les temps d'exécution*
`VERBOSE` *Affiche des informations supplémentaires*
`COSTS` *Affiche des informations détaillées sur les coûts*
`BUFFERS` *Affiche les informations sur le cache*
`FORMAT {TEXT | XML | JSON | YAML}` *Format de sortie*

Lire un plan de requête

Voici l'utilisation la plus simple d'`EXPLAIN`*:*
```
EXPLAIN SELECT * FROM tab1;
```
La commande renvoie le plan suivant :
```
Seq Scan on tab1 (cost=0.00..458.00 rows=10000 width=244)
```
Analyse :

Le moteur parcourt séquentiellement la table et renvoie toutes les lignes. Le coût total de l'opération est estimé à 458, le coût de traitement de la première ligne est de 0. Environ 10 000 lignes, de 244 octets chacune seront renvoyées.

Requête avec filtre

Ajoutons un premier filtre :
```
EXPLAIN SELECT * FROM tab1 WHERE col1 < 1000;
```
La commande renvoie le plan suivant :
```
Seq Scan on tab1 (cost=0.00..483.00 rows=7033 width=244)
Filter: (col1 < 1000)
```
Analyse :

Le moteur parcourt séquentiellement la table et renvoie les lignes validant le filtre. Notez l'augmentation du coût et la diminution du nombre de lignes estimées.

Filtres et index

Utilisons un filtre plus contraignant :
```
EXPLAIN SELECT * FROM tab1 WHERE col1 < 100;
```
La commande renvoie le plan suivant :
```
Bitmap Heap Scan on tab1 (cost=2.37..232.35 rows=106
width=244) Recheck Cond: (col1 < 100)
  -> Bitmap Index Scan on tab1_col1 (cost=0.00..2.37
rows=106 width=0) Index Cond: (col1 < 100)
```
Analyse :

Ce plan est un peu plus compliqué à lire car il est composé de deux instructions (plus une instruction est indentée, plus elle est exécutée tôt). Dans un premier temps l'index de la colonne *col1* est interrogé pour connaître les lignes validant le filtre, puis le second nœud récupère les pages sélectionnées, filtre les lignes et les renvoie (après les avoir triées dans leur ordre d'apparition en mémoire).

Filtres multiples

Ajoutons un premier filtre :
```
EXPLAIN SELECT * FROM tab1 WHERE col1 < 100 AND col2 > 9000;
```
La commande renvoie le plan suivant :
```
Bitmap Heap Scan on tab1 (cost=25.08..60.21 rows=10
width=244) Recheck Cond: ((col1 < 100) AND (col2 > 9000))
  -> BitmapAnd (cost=25.08..25.08 rows=10 width=0)
   -> Bitmap Index Scan on tab1_col1 (cost=0.00..5.04
rows=101 width=0) Index Cond: (col1 < 100)
   -> Bitmap Index Scan on tab1_col2 (cost=0.00..19.78
rows=999 width=0) Index Cond: (col2 > 9000)
```
Analyse :

Ici les deux colonnes servant de filtre possèdent un index. Le planificateur a donc choisi d'utiliser une combinaison binaire des indexes. la combinaison binaire d'indexes est un processus long à mettre en place (il est nécessaire d'ordonner les lignes en fonction de leur position sur le disque, mais cette opération permet un gain de temps substantiel si de nombreuses lignes sont à récupérer), le planificateur peut par conséquent opter pour une autre approche.

Modifions légèrement la requête :
```
EXPLAIN SELECT * FROM tab1 WHERE col1 < 100 AND col2 > 9000
LIMIT 2;
```
La commande renvoie le plan suivant :
```
Limit (cost=0.29..14.48 rows=2 width=244)
  -> Index Scan using tab1_col2 on tab1 (cost=0.29..71.27
rows=10 width=244) Index Cond: (col2 > 9000) Filter: (col1 <
100)
```
Analyse :

Cette requête montre bien à quel point un petit changement (ajout d'une clause *limit*) peut totalement changer un plan de requête. Le planificateur à choisi d'effectuer une recherche sur l'index de la colonne *col2* puis de filtrer les lignes sélectionnées avec la colonne *col1* (La particularité du nœud `Index Scan` est que, contrairement à nœud `Bitmap Heap Scan`, les lignes ne sont pas triées en fonction de leur position sur le disque). Cette nouvelle approche permet de diminuer le coût de la requête en supprimant l'interrogation du second index et ce malgré le surcoût de récupération des lignes induit par le nœud `Index Scan`.

Jointure et filtres

Ajoutons un premier filtre :
```
EXPLAIN SELECT * FROM tab1 AS t1, tab2 AS t2
WHERE t1.col1 < 10 AND t1.col2 = t2.col2;
```
La commande renvoie le plan suivant :
```
Nested Loop (cost=4.65..118.62 rows=10 width=488)
  -> Bitmap Heap Scan on tab1 t1 (cost=4.36..39.47 rows=10
width=244) Recheck Cond: (col1 < 10)
    -> Bitmap Index Scan on tab1_col1 (cost=0.00..4.36
rows=10 width=0) Index Cond: (col1 < 10)
  -> Index Scan using tab2_unique2 on tab2 t2
(cost=0.29..7.91 rows=1 width=244) Index Cond: (col2 =
t1.col2)
```
Analyse :

L'ajout d'une jointure modifie grandement le plan précédent. La première étape est une nouvelle fois la sélection par index des lignes validant la condition `col1 < 10`. Comme précédemment les lignes concernées sont récupérées par le nœud `Bitmap Heap Scan`, après une nouvelle vérification de la condition. Le dernier nœud extrait les lignes de la table *tab2* validant la condition de jointure. Finalement les lignes sont jointes dans le noeud `Nested Loop` (il s'agit d'une boucle imbriquée parcourant les deux ensembles de lignes).

Utilisation d'`ANALYSE`

Reprenons la première requête et ajoutons l'option `analyse`*:*
```
EXPLAIN ANALYSE SELECT * FROM tab1;
```
La commande renvoie le plan suivant :
```
Seq Scan on tab1 (cost=0.00..458.00 rows=10000 width=244)
(actual time=0.05..0.8 rows=9500 loops=1)
Planning time: 0.181 ms
Execution time: 0.80 ms
```
Analyse :

Aux informations déjà vues précédemment s'ajoutent, la durée de la planification, le temps d'exécution pour la première, pour tous les lignes et total, le nombre de lignes réellement retournées et le nombre d'exécutions de ce nœud.

Liste des nœuds

Seuls les nœuds les plus courants sont présentés.

Nœuds de parcours

`Seq Scan` *Parcours séquentiel de la table*
`Index Scan` *Recherche par index*
`Bitmap Heap Scan` *Recherche par index, algorithme alternatif*
`Index Only Scan` *Uniquement pour les* index couvrants

Nœuds de jointure (liste exhaustive)

`Nested Loop` *Boucles imbriquées*
`Merge Join` *Tri suivit d'une fusion*
`Hash Join` *Ne fonctionne qu'avec les égalités*

Nœuds d'agrégation

`Aggregate` *Tri puis agrégation*
`GroupAggregate` *Nécessite un pré-tri*
`HashAggregate` *Hachage des données et regroupement*

Pour aller plus loin

Outils d'analyse

Un outil de visualisation d'explain https ://explain.depesz.com/
Un autre outil du même type http ://tatiyants.com/pev/#/plans

Documentation

Use the index, Luke http ://use-the-index-luke.com/fr/sql/plans-dexecution/postgresql/operations
Comprendre *explain*
https ://www.dalibo.org/_media/comprendre_explain.pdf
Et bien entendu la documentation officielle
https ://docs.postgresql.fr/9.5/performance-tips.html#using-explain

https ://github.com/MBunel/CheatSheets
Réalisé pour la version 9.5 de PostgreSQL
NB : Dans un souci de concision ce document ne respecte pas l'indentation traditionnelle.