

# Bases de Données: Introduction à l'optimisation

## Mini-Projet: optimisation

Asma BERRIRI

[asma.berriri@universite-paris-saclay.fr](mailto:asma.berriri@universite-paris-saclay.fr)

Page web du cours :

<https://sites.google.com/view/asmaberriri/home/aspects-systeme-base-de-donnees>

1. *Ce projet peut être réalisé en groupe*
2. *Chaque équipe doit rendre:*
  - *une archive nommée Lastname\_Optimisation, e.g. : Martin\_Optimisation (Martin est l'un des membres de l'équipe). Cette archive contient :*
    - (a) *un rapport détaillé (un fichier pdf) nommé Lastname\_Optimisation.pdf, e.g. : Martin\_Optimisation.pdf*
    - (b) *le/les fichier(s) source(s) sql, commenté(s)*
3. *Les réponses doivent être classées par section/numéro dans le rapport pdf, et dans les fichiers sql comme commentaires*
4. *Toutes les réponses devront être justifiées/illustrées. La qualité de la rédaction/présentation influera sensiblement sur la note*
5. *Il est conseillé de résoudre chaque exercice dans un fichier sql séparé:*
  - *Pour chaque exercice rédigez un fichier sql contenant les requêtes/instructions sql*
  - *pour chaque exercice, donnez une analyse/conclusion sur les plans d'exécution*
  - *Mettez tous les fichiers dans une archive (par exemple, zip, tar)*
6. *Votre rapport doit contenir les noms des membres de l'équipe*
7. *Les livrables doivent être envoyés à [asma.berriri@universite-paris-saclay.fr](mailto:asma.berriri@universite-paris-saclay.fr)*
8. *La date limite de soumission sera donnée par le professeur*

## Préambule

Les travaux de ce mini-projet consistent à comprendre des requêtes exécutées sur de véritables bases de données et à interpréter le plan d'exécution de ces requêtes.

- Vous pouvez installer et utiliser un SGBD PostgreSQL de votre choix. Par exemple: <https://www.postgresql.org/download/>

- Sinon, nous allons utiliser l'environnement en ligne offert par : <https://sqliteonline.com/>
  - ↪ Dans le menu de gauche, cliquez sur "PostgreSQL"
  - ↪ Ensuite Activez la fonction "click to connect"
  - ↪ Dans la zone centrale (interpreteur), vous pouvez saisir vos instructions et scripts SQL
  - ↪ Dans le menu en haut à gauche, vous avez le bouton "Run" vous permettant d'exécuter vos instructions/scripts SQL

- Exemple de commande permettant d'obtenir le plan d'exécution d'une requête

```
EXPLAIN SELECT ...;
```

C'est à vous de chercher à comprendre le plan renvoyé par cette commande. Regardez l'Annexe.

- La commande *ANALYZE* est utilisée pour calculer et stocker des informations statistiques sur les tables et index analysés. Les informations statistiques collectées seront stockées dans une table système et seront ensuite utilisées par l'optimiseur du SGBD pour déterminer le plan de requête le plus efficace. Vous pouvez taper la commande suivante pour voir quelques statistiques sur une table donnée

```
ANALYZE TableName;
```

- Commande permettant d'obtenir le plan d'exécution détaillé d'une requête:

```
Explain analyse  
SELECT ...;
```

- Cette page du manuel PostgreSQL vous aide à comprendre et interpréter vos plans : [Utiliser EXPLAIN](https://docs.postgresql.fr/17/using-explain.html) (URL: <https://docs.postgresql.fr/17/using-explain.html>)
- NB: Pour une visualisation des plans d'exécution sous forme arborescente, vous pouvez vous aider du lien suivant: <https://explain.dalibo.com/> ([PostgreSQLexecutionplanvisualizer](#))
- Des documents utiles- PostgreSQL:
  - [Memo-Syntaxe-PostgreSQL](#)
  - [Documentation-Officielle-PostgreSQL-Fr](#)

Dans chaque exercice, des requêtes SQL vous seront demandées, il faut les exécuter dans l'interpréteur. Vous devez générer les plans d'exécution associés et les inspecter. Des questions vous sont posées sur l'interprétation de ces plans.

## 1 Exercice 1

### 1.1 Création de la base de données SANS CONTRAINTES

Nous allons créer une base 'HOTELS'. Le schéma de la base ne comprend que deux tables:

↪ *OCCUPATION* décrit l'occupation des chambres d'un hôtel : ID de la chambre, jour de l'occupation de la chambre (date sous forme de chaîne de caractères), ID du client, le nombre de personnes dans la chambre et une indication (0 ou 1) sur la réservation préalable ou non de la chambre.

↪ *CLIENT* décrit les clients de l'hôtel : ID client, le titre (M., Mme. ou Mlle.), le nom et le prénom du client, ainsi que l'entreprise qu'il représente

Voici leur schéma:

```
CREATE TABLE CLIENT
(
  CLI_ID          INTEGER,
  TIT_CODE        VARCHAR(10),
  CLI_NOM         VARCHAR(30),
  CLI_PRENOM      VARCHAR(30),
  CLI_ENSEIGNE    VARCHAR(50)
);

CREATE TABLE OCCUPATION
(
  CHB_ID          INTEGER,
  JOUR            VARCHAR(10),
  CLI_ID          INTEGER,
  NB_PERS         INTEGER,
  RESERVE         INTEGER
);
```

1. Créer les tables CLIENT et OCCUPATION avec les instructions SQL données ci-dessus (sans les modifier)
2. Pour remplir ces deux tables de quelques données, téléchargez le script *fillBaseHotel.sql* disponible sur le site du cours, ouvrez-le dans l'interpréteur SQL et exécutez le
3. Exprimez les requêtes suivantes dans la fenêtre d'exécution et exécutez les.
  - (a) Le nom et prénom du client numéro 5
  - (b) Les jours où le client numéro 5 a occupé une chambre
  - (c) Les chambres occupées le 1999-01-22
  - (d) Le nom et prénom des clients ayant pris une chambre le 1999-01-22
4. Pour chacune des requêtes précédentes :
  - Obtenez le plan d'exécution retourné par le SGBD
  - **Interprétez et expliquez à chaque fois le plan d'exécution proposé par le système (types d'accès aux tables/index, types de jointure, coût du plan)** (la réponse doit être brève et concise)
  - Dessinez ce plan sous forme arborescente en utilisant la syntaxe vue en cours

## 1.2 Création d'index

1. En vous aidant de la documentation PostgreSQL, créez des index sur *CLIENT.CLI\_ID*, *OCCUPATION.CLI\_ID* et *OCCUPATION.JOUR*, en respectant le type d'index unique ou non-unique
2. Exécutez les mêmes requêtes de la section 1.1 et à nouveau
  - Obtenez le plan d'exécution retourné par le SGBD
  - **Interprétez et expliquez à chaque fois le plan d'exécution proposé par le système (types d'accès aux tables/index, types de jointure, coût du plan)** (la réponse doit être brève et concise). Ces plans changent-ils par rapport à la section précédente. Si le plan ne change pas, expliquez la raison la plus probable pour la non-utilisation de l'index.
  - Dessinez ces plans sous forme arborescente en utilisant la syntaxe vue en cours

## 1.3 Statistiques

1. Reprenez les requêtes de la section 1.2. Expliquez les plans qui changent (et la raison probable) après avoir recueilli des statistiques sur les tables. (La réponse doit être brève et concise)
- La commande suivante met à jour les statistiques

```
analyse CLIENT, OCCUPATION;
```

- La commande suivante permet de voir les statistiques à l'écran

```
analyse verbose CLIENT, OCCUPATION;
```

## NB

Le fichier sql à rendre pour cet exercice doit contenir pour chaque plan différent des sections 1.1, 1.2, 1.3, une réponse le représentant. Pour chacun des 12 cas (4 requêtes \* 3 contextes), donner l'explication du plan demandée dans votre rapport pdf.

## 2 Exercice 2

Dans cet exercice, nous allons toujours utiliser le SGBD PostgreSQL.

On considère le schéma conceptuel de la figure 1

On souhaite prévoir, avant sa mise en production, le comportement des requêtes sur cette base de données.

### 2.1 Création de la base des commandes

1. Exécutez le script suivant qui implémente ce schéma dans une BD relationnelle



Page 5

```

DateCom      date,
NumC         integer,
Quantite     integer,
primary key (NumP, DateCom, NumC)
);

```

La base de données se trouve alors dans le schéma "optimisation" de votre base de données

2. Exécutez le script suivant qui implémente une génération aléatoire de tuples afin de tester les performances des charges de travail future

```

drop procedure if exists rempli_tables;
create procedure rempli_tables(nbtuples integer)
as $$
declare
k
un_numP      integer;
compt        integer;
une_date     date;
curs_com     cursor for select DateCom, numc
from optimisation.commandes;
begin

INSERT INTO optimisation.clients (numc , nomc , adressec)
SELECT g, 'nomc_' || ((random() * nbtuples * 4 / 5)::integer)::varchar, md5(g::text)
||md5((2*g)::text)||md5((3*g)::text)
FROM generate_series(0,nbtuples-1) as g;

INSERT INTO optimisation.produits (numP , nomp , descriptif)
SELECT g, 'nomp_' || ((random() * nbtuples * 4 / 5)::integer)::varchar, md5(g::text)
||md5((2*g)::text)||md5((3*g)::text)
FROM generate_series(0,nbtuples-1) as g;

for g in 1..4*(nbtuples-1) loop
une_date = CURRENT_TIMESTAMP - ((random() * 3650)::integer || 'day')::
interval;
un_numP=(random()*(nbtuples-1))::integer;
insert into optimisation.commandes (DateCom, numc, commentaire)
select une_date, un_numP, md5(g::text)||md5((2*g)::text)||md5((3*
g)::text) on conflict do nothing;
end loop;

for t in curs_com
loop
k=(random() *6)::integer;
for i in 1..k loop
un_numP = (random()*(nbtuples-1))::integer;
insert into optimisation.concerne (NumP,datecom,numc,
quantite)
values (un_numP,t.Datecom,t.numc,(random()*100)::integer)
on conflict do nothing;
if i>3 then
une_date = t.datecom+((random() * 30)::integer || '
days')::interval;
insert into optimisation.livraisons (DateLiv,
DateCom, numC, prestataire)
values(une_date,t.datecom,t.numc,'prest'||((random
()*20)::integer)::varchar) on conflict do
nothing;
end if;
end loop;
end loop;

```

```
end;
$$ language plpgsql;
```

Ce script devrait vous créer une procédure stockée appelée *rempli\_tables* que vous voyez dans le menu de gauche sous l'onglet *Procedure*.

3. Exécutez le script suivant qui appelle votre procédure

```
do $$
begin
    call rempli_tables(500);
end
$$;
```

Observez les tables créées, et notamment leurs différentes tailles.

4. Ajouter les contraintes en exécutant une à une les instructions suivantes

```
alter table optimisation.commandes add CONSTRAINT fk_commande_client foreign key (numc)
references optimisation.clients(NumC) on delete cascade not valid;
```

```
alter table optimisation.livraisons add constraint fk_livraison_commande foreign key (
    DateCom, NumC) references optimisation.commandes(DateCom, NumC) on delete cascade not
valid;
```

```
alter table optimisation.concerne add constraint fk_concerne_commande foreign key (DateCom
, NumC) references optimisation.commandes(DateCom, NumC) on delete cascade not valid;
```

```
alter table optimisation.concerne add constraint fk_concerne_produit foreign key (NumP)
references optimisation.produits(NumP) not valid;
```

5. Lancez la commande suivante pour mettre à jour les statistiques

```
analyse optimisation.clients, optimisation.commandes, optimisation.produits, optimisation.
concerne, optimisation.livraisons;
```

6. Lancez la commande suivante pour voir les statistiques à l'écran

```
analyse verbose optimisation.clients, optimisation.commandes, optimisation.produits,
optimisation.concerne, optimisation.livraisons;
```

## 2.2 Plans d'exécution

Pour chacune des requêtes (*REQi*) ci-dessous, rédigez une requête SQL qui répond à la question, et demandez à PostgreSQL une analyse de la requête en vous aidant de la syntaxe suivante:

```
EXPLAIN (analyse, buffers)
select ...
from ...
etc...
```

La commande *EXPLAIN* a pour effet de vous afficher le plan d'exécution choisi par l'optimiseur de requêtes. L'option "analyse" entre parenthèses va entraîner l'exécution effective de la requête, ce qui permet de comparer le coût du plan avec le temps réel, ainsi que le nombre réel de tuples générés à chaque étape. Enfin, "buffers" permet d'afficher le nombre de blocs lus, soit dans la mémoire cache (shared hits) soit sur le disque (shared read).

- Pour chaque requête (*REQi*) effectuée, faites des observations et des analyses dans votre rapport (et en commentaire dans votre fichier sql) en vous aidant des critères suivants:

- ↪ Le coût total estimé pour le plan d'exécution et le temps réel de l'exécution
- ↪ Le nombre de tuples estimé en sortie et la taille en octets de chaque tuple
- ↪ Le nombre de blocs lus
- ↪ L'accès utilisés vers les tables
- ↪ Les algorithmes utilisés

1. (*REQ1*) Listez toutes les informations sur les produits
2. (*REQ2*) Listez le numéro et le nom de tous les produits
3. (*REQ3*) Idem, en ajoutant la clause "distinct"
4. (*REQ4*) Même question en ordonnant le résultat selon le nom des produits
5. (*REQ5*) Listez les produits dont le nom est 'nomp\_327'
6. (*REQ6*) Donnez le nombre de commandes par client (numc, nombre)
7. (*REQ7*) Donnez les infos sur chaque commande, sous la forme ('nom du client', 'date de la commande'). Quelle est la méthode de jointure utilisée ? Pourquoi les relations sont elles considérées dans cet ordre dans le plan d'exécution ? Pourquoi la table commande n'est pas utilisée en fait ? Vous pouvez écrire différentes variations syntaxiques de la jointure pour observer ce qui se passe

On souhaite maintenant afficher les produits commandés par les clients. Soit la requête (*REQ8*) suivante:

```
----- REQ8 -----
explain -- Attention, pas de mot clé "analyse" ici car la requête ne termine pas !
select distinct nomc, nomp
from optimisation.clients,optimisation.concerne, optimisation.produits, optimisation.commandes,
     optimisation.livraisons
where  optimisation.clients.numc = optimisation.commandes.numc
       and optimisation.commandes.datecom = optimisation.concerne.datecom
       and optimisation.commandes.numc=optimisation.concerne.numc
       and optimisation.produits.nump=optimisation.concerne.nump
```

8. Etudiez le plan d'exécution de la requête (*REQ8*), observez le coût et le volume de tuples attendu, commentez

## 2.3 Optimisation par des index

Commande sous PostgreSQL pour lister les index dans la base 'optimisation'

```
SELECT
    tablename,
    indexname,
    indexdef
FROM
    pg_indexes
HERE
```



```
    schemaname = 'optimisation'
ORDER BY
    tablename,
    indexname;
```

1. Soient les trois requêtes suivantes :
2. *NB : les valeurs 109000 de numC et 2020-03-31 de datecom sont un exemple à suivre. Ces valeurs, étant été générée aléatoirement par le script donné précédemment, les valeurs dans votre base seront différentes. C'est à vous de choisir ces valeurs de telle sorte à ce que les requêtes qui suivent vous donnent des résultats.*

- Requête-1:

```
Select *
from optimisation.commandes
where numC='109000' -- valeur à modifier selon votre base
and      datecom='2020-03-31'; -- valeur à modifier selon votre base
```

- Requête-2:

```
Select *
from optimisation.commandes
where datecom='2020-03-31';
```

- Requête-3:

```
Select *
from optimisation.commandes
where numC='109000';
```

Pour chacune des trois requêtes

- (a) Donnez le plan d'exécution PEP retourné par PostgreSQL
- (b) Observez l'utilisation de l'index de clé primaires, et trouvez dans la documentation de PostgreSQL à quoi correspondent les chemins d'accès
- (c) Dessinez ce plan sous forme arborescente en utilisant la syntaxe vue en cours

En conclusion de cette question, que faire si dans les requêtes sur la relation "commande", la recherche par numéro de client est beaucoup plus fréquente que la recherche par date de commande ?

3. Soit la requête *ReqA* suivante

```
--ReqA-- Noms de produits commandés par les clients qui portent le nom : 'nomc_1287'--
Select nomp
from optimisation.produits p
    join optimisation.concerne co using(nump)
    join optimisation.clients c using(numc)
where nomc='nomc_1287';
```

- (a) Donnez le plan d'exécution PEP retourné par PostgreSQL
- (b) Dessinez ce plan sous forme arborescente en utilisant la syntaxe vue en cours, et en annotant avec les algorithmes et méthodes d'accès aux tables

- (c) Optimisez par deux index la requête *ReqA*. N'oubliez pas de reporter les commandes de création de vos index dans votre rapport
- Reférez vous à la documentation de PostgreSQL et à la syntaxe suivante pour créer des index

```
create index ... ;  
create index ... ;
```

- (d) Donnez le plan d'exécution PEP retourné par PostgreSQL après la création des index et comparer avec le plan précédent

A la fin de cette question, supprimez les index créés (Reférez vous à la documentation de PostgreSQL)

4. Comparer les plans d'exécution des requêtes ci-dessous

- Requête-A:

```
Select numc  
from optimisation.clients  
where numc not in  
      (select numc from optimisation.livraisons);
```

- Requête-B:

```
Select numc  
from optimisation.clients  
except  
select numc  
from optimisation.livraisons;
```

- Requête-C:

```
Select numc  
from optimisation.clients  
where not exists  
      (select numc from optimisation.livraisons  
       where optimisation.livraisons.numc=optimisation.clients.numc);
```

5. Soit la requête suivante, qui est souvent utilisée pour éviter de manquer des recherches pour un problème de casse.

```
-- Recherche et affichage des clients avec passage en majuscules  
select upper(numc), adressec  
from clients  
where upper(numc)='NOMC_1725';
```

- (a) Donnez le plan d'exécution de cette requête en utilisant cette syntaxe:

```
Explain analyse  
requête à expliciter>
```

- (b) Dessinez ce plan PEP sous forme arborescente (syntaxe vu en cours) en l'annotant et expliquez-le
- (c) Suggérez l'ajout d'un index pour optimiser la requête. N'oubliez pas de reporter la commande de création de votre index dans votre rapport

- (d) Redonnez le PEP correspondant après la création de votre index et expliquez

A la fin de cette question, supprimez les index créés (Reférez vous à la documentation de PostgreSQL)

6. Soit maintenant la requête suivante

```
SELECT COUNT(*)
FROM commandes
WHERE EXTRACT(YEAR FROM datecom) = 2017;
```

- (a) Donnez le plan d'exécution de cette requête en utilisant la syntaxe:

```
Explain analyse
equête à expliciter>
```

- (b) Dessinez ce plan PEP sous forme arborescente (syntaxe vu en cours) en l'annotant et expliquez-le en indiquant notamment les chemins d'accès aux tables et les algorithmes ...
- (c) La clé primaire est-elle utilisée ? Pourquoi ?
- (d) Plutôt que de créer un nouvel index coûteux, proposez une ré-écriture de la requête qui utilisera l'index existant via la clé primaire.

### 3 Exercice 3: Interprétation de PEP Oracle

Nous allons maintenant regarder de près quelques plans de l'optimiseur du SGBD Oracle (sans les exécuter).

Soit le schéma suivant :

```
CREATE TABLE Artiste (
  ID-artiste NUMBER(4),
  Nom VARCHAR2(32),
  Adresse VARCHAR2(32) );

CREATE TABLE Film (
  ID-film NUMBER(4),
  Titre VARCHAR2(32),
  Année NUMBER(4),
  ID-réalisateur NUMBER(4) );

CREATE TABLE Joue (
  ID-artiste NUMBER(4),
  ID-film NUMBER(4) );
```

1. Donner l'ordre SQL pour la requête : Afficher le nom des acteurs et le titre des films où ils ont joué
2. Donner l'expression algébrique correspondante (PEL)
3. Dans chacun des cas suivants, le plan d'exécution PEP retourné par Oracle vous est donné
  - (a) 1er cas : il n'existe que deux index, un sur FILM(*ID-réalisateur*), et un sur ARTISTE(*ID-artiste*)

```

0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 NESTED LOOPS
        4 TABLE ACCESS FULL JOUE
        5 TABLE ACCESS BY ROWID ARTISTE
          6 INDEX UNIQUE SCAN ARTISTE_IDX
      7 SORT JOIN
        8 TABLE ACCESS FULL

```

(b) 2ème cas: un index sur FILM( $ID - Film$ ), et un sur JOUE( $ID - Artiste$ )

```

0 SELECT STATEMENT
  1 NESTED LOOPS
    2 NESTED LOOPS
      3 TABLE ACCESS FULL ARTISTE
      4 TABLE ACCESS BY ROWID JOUE
        5 INDEX RANGE SCAN JOUE_ARTISTE
      6 TABLE ACCESS BY ROWID FILM
        7 INDEX UNIQUE SCAN FILM_IDX

```

(c) 3ème cas: un index sur FILM( $ID - Film$ ), et un sur JOUE( $ID - Film$ )

```

0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 NESTED LOOPS
        4 TABLE ACCESS FULL JOUE
        5 TABLE ACCESS BY ROWID FILM
          6 INDEX UNIQUE SCAN FILM_IDX
      7 SORT JOIN
        8 TABLE ACCESS FULL ARTISTE

```

- Analysez les PEP retourné par Oracle en répondant aux questions suivantes:
  - Pour le 1er cas (a), le parcours séquentiel est fait sur quelle(s) table(s)? Quel(s) index sont utilisés pour la jointure? Avec quel autre table on aurait pu faire le parcours séquentiel? L'index sur  $ID - realisateur$  est-il utilisable pour la jointure? Pourquoi?
  - Pour le 2ème cas (b), pourquoi la table (ARTISTE) est elle choisi pour effectuer le parcours séquentiel initial dans la jointure?
  - Pour le 3ème cas (c), les index existant ont servi à la jointure entre quelles tables? Pourrait on inverser l'ordre dans cette jointure? Quel algorithme est utilisé pour la seconde jointure?
- Pour chacun des cas (a), (b) et (c):
  - Donnez le PEP (sous forme arborescente) correspondant au plan retourné par Oracle, en l'annotant par les algorithmes de jointure et les méthodes d'accès aux tables utilisés

## A Annexes:

Utiliser **EXPLAIN** (URL: <https://docs.postgresql.fr/17/using-explain.html>)

17 (1154 pages) 

Documentation PostgreSQL 17.2 » Langage SQL » Conseils sur les performances » Utiliser EXPLAIN

[Précédent](#)
[Chapitre 14. Conseils sur les performances](#)
[Niveau supérieur](#)
[Sommaire](#)
[Suivant](#)
[14.2. Statistiques utilisées par le planificateur](#)

## 14.1. Utiliser EXPLAIN

### 14.1.1. Concepts d'EXPLAIN

#### 14.1.2. EXPLAIN ANALYZE

#### 14.1.3. Avertissements

PostgreSQL réalise un *plan de requête* pour chaque requête qu'il reçoit. Choisir le bon plan correspondant à la structure de la requête et aux propriétés des données est absolument critique pour de bonnes performances, donc le système inclut un *planificateur* ou *optimiseur* complexe qui tente de choisir les bons plans. Vous pouvez utiliser la commande `EXPLAIN` pour voir quel plan de requête le planificateur crée pour une requête particulière. La lecture du plan est un art qui requiert de l'expérience pour le maîtriser, mais cette section essaie de couvrir les bases.

Les exemples dans cette section sont tirés de la base de données pour les tests de régression après avoir effectué un `VACUUM ANALYZE`, avec les sources de la version de développement v17. Vous devriez obtenir des résultats similaires si vous essayez les exemples vous-même, mais vos estimations de coût et de nombre de lignes pourraient légèrement varier parce que les statistiques d'`ANALYZE` sont basées sur des échantillons aléatoires, et parce que les coûts sont dépendants de la plateforme utilisée.

Les exemples utilisent le format de sortie par défaut (« text ») d'`EXPLAIN`, qui est compact et pratique pour la lecture. Si vous voulez utiliser la sortie d'`EXPLAIN` avec un programme pour une analyse ultérieure, vous devriez utiliser un des formats de sortie au format machine (XML, JSON ou YAML) à la place.

### 14.1.1. Concepts d'EXPLAIN

La structure d'un plan de requête est un arbre de *nœuds de plan*. Les nœuds de bas niveau sont les nœuds de parcours : ils renvoient les lignes brutes d'une table. Il existe différents types de nœuds de parcours pour les différentes méthodes d'accès aux tables : parcours séquentiel, parcours d'index et parcours d'index bitmap. Il y a également des ensembles de lignes qui ne proviennent pas de tables, avec par exemple des clauses `VALUES` ainsi que des fonctions renvoyant des ensembles dans un `FROM`, qui ont leurs propres types de nœuds de parcours. Si la requête requiert des jointures, agrégations, tris ou d'autres opérations sur les lignes brutes, ce seront des nœuds supplémentaires au-dessus des nœuds de parcours pour réaliser ces opérations. Encore une fois, il existe plus d'une façon de réaliser ces opérations, donc différents types de nœuds peuvent aussi apparaître ici. La sortie d'`EXPLAIN` comprend une ligne pour chaque nœud dans l'arbre du plan, montrant le type de nœud basique avec les estimations de coût que le planificateur a faites pour l'exécution de ce nœud du plan. Des lignes supplémentaires peuvent apparaître, indentées par rapport à la ligne de résumé du nœud, pour montrer les propriétés supplémentaires du nœud. La première ligne (le nœud tout en haut) comprend le coût d'exécution total estimé pour le plan ; c'est ce nombre que le planificateur cherche à minimiser.

Voici un exemple trivial, juste pour montrer à quoi ressemble l'affichage.

```
EXPLAIN SELECT * FROM tenk1;

              QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

Puisque la requête n'a pas de clause `WHERE`, il faut parcourir toutes les lignes de la table, c'est pourquoi le planificateur a choisi d'utiliser un plan avec un simple parcours séquentiel. Les nombres affichés entre parenthèses sont (de gauche à droite) :

- Coût estimé du lancement. Cela correspond au temps passé avant que l'affichage de la sortie ne commence, par exemple le temps de faire un tri dans un nœud de tri ;
- Coût total estimé. Cela suppose que le nœud du plan d'exécution est exécuté entièrement, c'est-à-dire que toutes les lignes disponibles sont récupérées. En pratique, un nœud parent peut arrêter la récupération de toutes les lignes disponibles avant la fin (voir l'exemple `LIMIT` ci-dessous) ;
- Nombre de lignes estimé en sortie par ce nœud de plan. Encore une fois, on suppose que le nœud est exécuté entièrement.
- Largeur moyenne estimée (en octets) des lignes en sortie par ce nœud du plan.

Les coûts sont mesurés en unités arbitraires déterminées par les paramètres de coût du planificateur (voir [Section 19.7.2](#)). La pratique habituelle est de mesurer les coûts en unité de récupération de pages disque ; autrement dit, `seq_page_cost` est initialisé à 1.0 par convention et les autres paramètres de coût sont relatifs à cette valeur. Les exemples de cette section sont exécutés avec les paramètres de coût par défaut.

Il est important de comprendre que le coût d'un nœud de haut niveau inclut le coût de tous les nœuds fils. Il est aussi important de réaliser que le coût reflète seulement les éléments d'importance pour le planificateur. En particulier, le coût ne considère pas le temps passé à convertir les valeurs en entrée vers le format texte ou à les transmettre au client. Les deux pourraient être des facteurs importants du temps réel passé ; mais l'optimiseur ignore ces coûts parce qu'il ne peut pas les changer sans modifier le plan. (Chaque plan correct renverra le même ensemble de ligne.)

La valeur `rows` est un peu difficile car il ne s'agit pas du nombre de lignes traitées ou parcourues par le plan de nœuds, mais plutôt le nombre émis par le nœud. C'est habituellement moins, reflétant la sélectivité estimée des conditions de la clause `WHERE` qui sont appliquées au nœud. Idéalement, les estimations des lignes de haut niveau seront une approximation des nombres de lignes déjà renvoyées, mises à jour, supprimées par la requête.

Quand un `UPDATE`, un `DELETE` ou un `MERGE` affecte une table partitionnée ou une hiérarchie d'héritage, la sortie pourrait ressembler à ceci :

```
+EXPLAIN UPDATE gtest_parent SET f1 = CURRENT_DATE WHERE f2 = 101;

              QUERY PLAN
-----
Update on gtest_parent  (cost=0.00..3.06 rows=0 width=0)
  Update on gtest_child gtest_parent_1
  Update on gtest_child2 gtest_parent_2
  Update on gtest_child3 gtest_parent_3
```

```

-> Append (cost=0.00..3.06 rows=3 width=14)
    -> Seq Scan on gtest_child gtest_parent_1 (cost=0.00..1.01 rows=1 width=14)
        Filter: (f2 = 101)
    -> Seq Scan on gtest_child2 gtest_parent_2 (cost=0.00..1.01 rows=1 width=14)
        Filter: (f2 = 101)
    -> Seq Scan on gtest_child3 gtest_parent_3 (cost=0.00..1.01 rows=1 width=14)
        Filter: (f2 = 101)

```

Dans cet exemple, le nœud Update doit prendre en compte les trois tables filles mais pas la table partitionnée mentionnée à l'origine (tout simplement parce qu'elle ne contient aucune données). Donc il y a trois sous-plans de parcours en entrée, un par table. Pour plus de clarté, le nœud Update est annoté pour afficher les tables cibles spécifiques à mettre à jour, dans le même ordre que les sous-plans correspondants.

Le Temps de planification (Planning time) affiché est le temps qu'il a fallu pour générer le plan d'exécution de la requête analysée et pour l'optimiser. Cela n'inclut ni le temps de réécriture ni le temps d'analyse.

Pour revenir à notre exemple :

```

EXPLAIN SELECT * FROM tenk1;

               QUERY PLAN
-----
Seq Scan on tenk1 (cost=0.00..445.00 rows=10000 width=244)

```

Ces nombres sont directement dérivés. Si vous faites :

```

SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

```

vous trouverez que `tenk1` a 345 blocs disque et 10000 lignes. Le coût estimé est calculé avec (nombre de blocs lus \* `seq_page_cost`) + (lignes parcourues \* `cpu_tuple_cost`). Par défaut, `seq_page_cost` vaut 1,0 et `cpu_tuple_cost` vaut 0,01. Donc le coût estimé est de (345 \* 1,0) + (10000 \* 0,01), soit 445.

Maintenant, modifions la requête originale pour ajouter une condition `WHERE` :

```

EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

               QUERY PLAN
-----
Seq Scan on tenk1 (cost=0.00..470.00 rows=7000 width=244)
  Filter: (unique1 < 7000)

```

Notez que l'affichage d'`EXPLAIN` montre la clause `WHERE` appliquée comme une condition de « filtre » rattachée au nœud de parcours séquentiel ; ceci signifie que le nœud de plan vérifie la condition pour chaque ligne qu'il parcourt et ne conserve que celles qui satisfont la condition. L'estimation des lignes en sortie a baissé à cause de la clause `WHERE`. Néanmoins, le parcours devra toujours visiter les 10000 lignes, donc le coût n'a pas baissé ; en fait, il a un peu augmenté (par 10000 \* `cpu_operator_cost` pour être exact) dans le but de refléter le temps CPU supplémentaire dépensé pour vérifier la condition `WHERE`.

Le nombre réel de lignes que cette requête sélectionnera est 7000, mais l'estimation `rows` est approximative. Si vous tentez de dupliquer cette expérience, vous obtiendrez probablement une estimation légèrement différente ; de plus, elle changera après chaque commande `ANALYZE` parce que les statistiques produites par `ANALYZE` sont prises à partir d'un extrait au hasard de la table.

Maintenant, rendons la condition plus restrictive :

```

EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

               QUERY PLAN
-----
Bitmap Heap Scan on tenk1 (cost=5.06..224.98 rows=100 width=244)
  Recheck Cond: (unique1 < 100)
-> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=100 width=0)
    Index Cond: (unique1 < 100)

```

Ici, le planificateur a décidé d'utiliser un plan en deux étapes : le nœud en bas du plan visite un index pour trouver l'emplacement des lignes correspondant à la condition de l'index, puis le nœud du plan du dessus récupère réellement ces lignes de la table. Récupérer séparément les lignes est bien plus coûteux que de les lire séquentiellement, mais comme toutes les pages de la table n'ont pas à être visitées, cela revient toujours moins cher qu'un parcours séquentiel (la raison de l'utilisation d'un plan à deux niveaux est que le nœud du plan du dessus trie les emplacements des lignes identifiées par l'index dans l'ordre physique avant de les lire pour minimiser les coûts des récupérations séparées. Le « bitmap » mentionné dans les noms de nœuds est le mécanisme qui s'occupe du tri).

Maintenant, ajoutons une autre condition à la clause `WHERE` :

```

EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

               QUERY PLAN
-----
Bitmap Heap Scan on tenk1 (cost=5.04..225.20 rows=1 width=244)
  Recheck Cond: (unique1 < 100)
  Filter: (stringu1 = 'xxx':name)
-> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=100 width=0)
    Index Cond: (unique1 < 100)

```

L'ajout de la condition `stringu1 = 'xxx'` réduit l'estimation du nombre de lignes renvoyées, mais pas son coût, car il faut toujours parcourir le même ensemble de lignes. C'est dû fait que la clause `stringu1` ne peut être appliquée comme une condition d'index car l'index ne porte que sur la colonne `unique1`. À la place, un filtre a été appliqué sur les lignes récupérées en utilisant l'index. C'est pourquoi le coût a légèrement augmenté pour refléter la vérification supplémentaire.

Dans certains cas, le planificateur préférera un plan « simple » d'index :

```

EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

```

```

QUERY PLAN
-----
Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
    Index Cond: (unique1 = 42)

```

Dans ce type de plan, les lignes de la table sont récupérées dans l'ordre de l'index, ce qui les rend encore plus coûteuses à récupérer, mais il y en a tellement peu que le coût supplémentaire pour trier l'ordre des lignes n'est pas rentable. Vous verrez principalement ce type de plan pour les requêtes qui ne récupèrent qu'une seule ligne, ou pour les requêtes qui ont une condition `ORDER BY` qui correspond à l'ordre de l'index, car cela ne nécessite aucune étape supplémentaire pour satisfaire l'`ORDER BY`. Dans cet exemple, ajouter `ORDER BY unique1` ferait que l'optimiseur utilise le même plan parce que l'index fournit déjà implicitement le tri requis.

L'optimiseur peut exécuter une clause `ORDER BY` de plusieurs façons. L'exemple ci-dessus montre qu'une clause de tri peut être effectué implicitement. L'optimiseur peut aussi ajouter une étape de tri (`Sort`) explicite :

```

EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;
               QUERY PLAN
-----
Sort  (cost=1109.39..1134.39 rows=10000 width=244)
    Sort Key: unique1
    -> Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)

```

Si une partie du plan garantit un ordre sur un préfixe des clés de tri requises, alors l'optimiseur peut décider à la place d'utiliser une étape de tri incrémental (`Incremental Sort`) :

```

EXPLAIN SELECT * FROM tenk1 ORDER BY hundred, ten LIMIT 100;
               QUERY PLAN
-----
Limit  (cost=19.35..39.49 rows=100 width=244)
    -> Incremental Sort  (cost=19.35..2033.39 rows=10000 width=244)
        Sort Key: hundred, ten
        Presorted Key: hundred
        -> Index Scan using tenk1_hundred on tenk1  (cost=0.29..1574.20 rows=10000 width=244)

```

Comparé aux tris habituels, le tri incrémental permet de renvoyer les lignes avant que l'ensemble du résultat ne soit trié, ce qui permet en particulier des optimisations avec les requêtes utilisant la clause `LIMIT`. Il peut aussi réduire l'utilisation de la mémoire et la probabilité d'envoyer des tris sur disque, mais cela a un coût : une surcharge pour répartir l'ensemble de lignes résultats dans plusieurs groupes de tri.

S'il y a des index sur plusieurs colonnes référencées dans la clause `WHERE`, le planificateur pourrait choisir d'utiliser une combinaison binaire (`AND` et `OR`) des index :

```

EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
               QUERY PLAN
-----
Bitmap Heap Scan on tenk1  (cost=25.07..60.11 rows=10 width=244)
    Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
    -> BitmapAnd  (cost=25.07..25.07 rows=10 width=0)
        -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
            Index Cond: (unique1 < 100)
        -> Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
            Index Cond: (unique2 > 9000)

```

Mais ceci requiert de visiter plusieurs index, donc ce n'est pas nécessairement un gain comparé à l'utilisation d'un seul index et au traitement de l'autre condition par un filtre. Si vous variez les échelles de valeurs impliquées, vous vous apercevrez que le plan change en accord.

Voici un exemple montrant les effets d'un `LIMIT` :

```

EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
               QUERY PLAN
-----
Limit  (cost=0.29..14.28 rows=2 width=244)
    -> Index Scan using tenk1_unique2 on tenk1  (cost=0.29..70.27 rows=10 width=244)
        Index Cond: (unique2 > 9000)
        Filter: (unique1 < 100)

```

C'est la même requête qu'au-dessus, mais avec l'ajout de `LIMIT`, ce qui fait que toutes les lignes ne seront pas récupérées, et donc que le planificateur change sa façon de procéder. Notez que le coût total ainsi que le nombre de lignes du nœud de parcours d'index sont affichés comme si le nœud devait être exécuté entièrement. Cependant, le nœud `Limit` s'attend à s'arrêter après avoir récupéré seulement un cinquième de ces lignes, c'est pourquoi son coût total n'est qu'un cinquième du coût précédent, ce qui est le vrai coût estimé de la requête. Ce plan est préférable à l'ajout d'un nœud `Limit` au plan précédent, car le `Limit` ne pourrait pas empêcher le coût de départ du parcours d'index `Bitmap`, ce qui augmenterait le coût d'environ 25 unités avec cette approche.

Maintenant, essayons de joindre deux tables, en utilisant les colonnes dont nous avons discuté :

```

EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;
               QUERY PLAN
-----
Nested Loop  (cost=4.65..118.50 rows=10 width=488)
    -> Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10 width=244)
        Recheck Cond: (unique1 < 10)
        -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
            Index Cond: (unique1 < 10)

```

```
-> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..7.90 rows=1 width=244)
    Index Cond: (unique2 = t1.unique2)
```

Dans ce plan, nous avons un nœud de jointure en boucle imbriquée sur deux parcours de tables en entrée. L'indentation des lignes de sommaire des nœuds reflète la structure en arbre du plan. Le premier nœud, ou nœud « externe », utilise le même parcours de bitmap que celui vu précédemment, et donc ses coût et nombre de lignes sont les mêmes que ce que l'on aurait obtenu avec `SELECT ... WHERE unique1 < 10`, car la même clause `WHERE unique1 < 10` est appliquée à ce nœud. La clause `t1.unique2 = t2.unique2` n'a pas encore d'intérêt, elle n'affecte donc pas le nombre de lignes du parcours externe. Le nœud de jointure en boucle imbriquée s'exécute sur le deuxième nœud, ou nœud « interne », pour chaque ligne obtenue du nœud externe. Les valeurs de colonne de la ligne externe courante peuvent être utilisées dans le parcours interne ; ici, la valeur `t1.unique2` de la ligne externe est disponible, et on peut obtenir un plan et un coût similaires à ce que l'on a vu plus haut pour le cas simple `SELECT ... WHERE t2.unique2 = constant`. (Le coût estimé est ici un peu plus faible que celui vu précédemment, en prévision de la mise en cache des données durant les parcours d'index répétés sur `t2`.) Les coûts du nœud correspondant à la boucle sont ensuite initialisés sur la base du coût du parcours externe, avec une répétition du parcours interne pour chaque ligne externe (ici  $10 * 7,90$ ), plus un petit temps CPU pour traiter la jointure.

Dans cet exemple, le nombre de lignes en sortie de la jointure est identique au nombre de lignes des deux parcours, mais ce n'est pas vrai en règle générale car vous pouvez avoir des clauses `WHERE` mentionnant les deux tables et qui, donc, peuvent seulement être appliquées au point de jointure, et non pas aux parcours d'index. Voici un exemple :

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

               QUERY PLAN
-----
Nested Loop  (cost=4.65..49.36 rows=33 width=488)
  Join Filter: (t1.hundred < t2.hundred)
-> Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10 width=244)
    Recheck Cond: (unique1 < 10)
-> Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
    Index Cond: (unique1 < 10)
-> Materialize  (cost=0.29..8.51 rows=10 width=244)
    -> Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10 width=244)
        Index Cond: (unique2 < 10)
```

La condition `t1.hundred < t2.hundred` ne peut être testée dans l'index `tenk2_unique2`, elle est donc appliquée au nœud de jointure. Cela réduit l'estimation du nombre de lignes dans le nœud de jointure, mais ne change aucun parcours d'entrée.

Notez qu'ici le planificateur a choisi de matérialiser la relation interne de la jointure en plaçant un nœud `Materialize` au-dessus. Cela signifie que le parcours d'index de `t2` ne sera réalisé qu'une seule fois, même si le nœud de jointure par boucle imbriquée va lire dix fois les données, une fois par ligne de la relation externe. Le nœud `Materialize` conserve les données en mémoire lors de leur première lecture, puis renvoie les données depuis la mémoire à chaque lecture supplémentaire.

Quand vous utilisez des jointures externes, vous pouvez voir des nœuds de plan de jointure avec à la fois des conditions « Join Filter » et « Filter » simples attachées. Les conditions « Join Filter » viennent des clauses de jointures externes `ON`, pour qu'une ligne ne satisfaisant pas la condition « Join Filter » puisse toujours être récupérée comme une ligne `NULL`. Mais une condition « Filter » simple est appliquée après la règle de jointure externe et supprime donc les lignes de manière inconditionnelle. Dans une jointure interne, il n'y a pas de différence sémantique entre ces types de filtres.

Si nous changeons un peu la sélectivité de la requête, on pourrait obtenir un plan de jointure très différent :

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

               QUERY PLAN
-----
Hash Join  (cost=226.23..709.73 rows=100 width=488)
  Hash Cond: (t2.unique2 = t1.unique2)
-> Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
-> Hash  (cost=224.98..224.98 rows=100 width=244)
    -> Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98 rows=100 width=244)
        Recheck Cond: (unique1 < 100)
    -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
        Index Cond: (unique1 < 100)
```

Ici, le planificateur a choisi d'utiliser une jointure de hachage, dans laquelle les lignes d'une table sont entrées dans une table de hachage en mémoire, après quoi l'autre table est parcourue et la table de hachage est sondée pour faire correspondre chaque ligne. Notez encore une fois comment l'indentation reflète la structure du plan : le parcours d'index bitmap sur `tenk1` est l'entrée du nœud de hachage, qui construit la table de hachage. C'est alors retourné au nœud de jointure de hachage, qui lit les lignes depuis le plan du fils externe et cherche dans la table de hachage pour chaque ligne.

Un autre type de jointure possible est la jointure d'assemblage, illustrée ici :

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

               QUERY PLAN
-----
Merge Join  (cost=0.56..233.49 rows=10 width=488)
  Merge Cond: (t1.unique2 = t2.unique2)
-> Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..643.28 rows=100 width=244)
    Filter: (unique1 < 100)
-> Index Scan using onek_unique2 on onek t2  (cost=0.28..166.28 rows=1000 width=244)
```

La jointure d'assemblage nécessite que les données en entrée soient triées sur la clé de jointure. Dans cet exemple, chaque entrée est triée en utilisant un parcours d'index pour visiter les lignes dans le bon ordre ; un parcours séquentiel suivi d'un tri pourraient aussi être utilisés. (Un parcours séquentiel suivi d'un tri bat fréquemment un parcours d'index pour trier de nombreuses lignes, du fait des accès disques non séquentiels requis par le parcours d'index.)



Une façon de chercher des plans différents est de forcer le planificateur à oublier certaines stratégies qu'il aurait trouvées moins coûteuses en utilisant les options d'activation (enable)/désactivation (disable) décrites dans la [Section 19.7.1](#) (c'est un outil complexe, mais utile ; voir aussi la [Section 14.3](#)). Par exemple, si nous n'étions pas convaincus que la jointure d'assemblage soit le meilleur type de jointure dans l'exemple précédent, nous pourrions essayer

```
SET enable_mergejoin = off;
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

               QUERY PLAN
-----
Hash Join  (cost=226.23..344.08 rows=10 width=488)
  Hash Cond: (t2.unique2 = t1.unique2)
    -> Seq Scan on onek t2  (cost=0.00..114.00 rows=1000 width=244)
    -> Hash  (cost=224.98..224.98 rows=100 width=244)
          -> Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98 rows=100 width=244)
              Recheck Cond: (unique1 < 100)
              -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
                  Index Cond: (unique1 < 100)
```

ce qui montre que le planificateur pense que la jointure par hachage serait pratiquement 50% plus coûteuse qu'une jointure par assemblage dans ce cas. Bien sûr, la question suivante est de savoir s'il a raison sur ce point. Nous pourrions vérifier cela en utilisant `EXPLAIN ANALYZE`, comme expliqué [ci-dessous](#).

Certains plans de requête impliquent des *sous-plans*, qui viennent des sous-`SELECT` de la requête originale. De telles requêtes peuvent parfois être transformés en jointures ordinaires mais, quand ce n'est pas possible, nous obtenons des plans de ce type :

```
EXPLAIN VERBOSE SELECT unique1
FROM tenk1 t
WHERE t.ten < ALL (SELECT o.ten FROM onek o WHERE o.four = t.four);

               QUERY PLAN
-----
Seq Scan on public.tenk1 t  (cost=0.00..586095.00 rows=5000 width=4)
  Output: t.unique1
  Filter: (ALL (t.ten < (SubPlan 1).col1))
  SubPlan 1
    -> Seq Scan on public.onek o  (cost=0.00..116.50 rows=250 width=4)
        Output: o.ten
        Filter: (o.four = t.four)
```

Cet exemple assez artificiel sert à illustrer un ensemble de points : les valeurs au niveau du plan externe sont passées au sous-plan (ici, il s'agit de `t.four`) et les résultats de la sous-requête sont disponibles sur le plan externe. Ces valeurs de résultat sont affichées par `EXPLAIN` avec des notations comme **(nom\_sousplan).colN**, qui fait référence à la *N*-ième colonne en sortie du sous-`SELECT`.

Dans l'exemple ci-dessus, l'opérateur `ALL` exécute le sous-plan pour chaque ligne de la requête externe (ce qui explique le coût estimé très fort). Certaines requêtes peuvent utiliser un *sous-plan haché* pour éviter cela :

```
EXPLAIN SELECT *
FROM tenk1 t
WHERE t.unique1 NOT IN (SELECT o.unique1 FROM onek o);

               QUERY PLAN
-----
Seq Scan on tenk1 t  (cost=61.77..531.77 rows=5000 width=244)
  Filter: (NOT (ANY (unique1 = (hashed SubPlan 1).col1)))
  SubPlan 1
    -> Index Only Scan using onek_unique1 on onek o  (cost=0.28..59.27 rows=1000 width=4)
(4 rows)
```

Ici, le sous-plan est lancé une seule fois et le résultat est chargé dans une table de hachage en mémoire, qui est ensuite interrogée par l'opérateur externe `ANY`. Ceci nécessite que le sous-`SELECT` ne référence pas de variables de la requête externe, et que l'opérateur de comparaison `ANY` puisse utiliser le hachage.

Si, en plus de ne pas référencer les variables de la requête externe, le sous-`SELECT` ne peut pas renvoyer plus d'une ligne, il pourrait être implémenté à la place comme un *initplan* :

```
EXPLAIN VERBOSE SELECT unique1
FROM tenk1 t1 WHERE t1.ten = (SELECT (random() * 10)::integer);

               QUERY PLAN
-----
Seq Scan on public.tenk1 t1  (cost=0.02..470.02 rows=1000 width=4)
  Output: t1.unique1
  Filter: (t1.ten = (InitPlan 1).col1)
  InitPlan 1
    -> Result  (cost=0.00..0.02 rows=1 width=4)
        Output: ((random() * '10'::double precision)::integer)
```

Un *initplan* est exécuté une seule fois par exécution du plan externe, et les résultats sont sauvegardés pour une ré-utilisation par les lignes suivantes du plan externe. Donc dans cet exemple, `random()` est évalué seulement une fois et toutes les valeurs de `t1.ten` sont comparées au même entier choisi au hasard. C'est très différent de ce qui serait survenu sans la construction du sous-`SELECT`.

### 14.1.2. `EXPLAIN ANALYZE`

Il est possible de vérifier l'exactitude des estimations du planificateur en utilisant l'option `ANALYZE` de `EXPLAIN`. Avec cette option, `EXPLAIN` exécute vraiment la requête, puis affiche le vrai nombre de lignes et les vrais temps passés dans chaque nœud, avec ceux estimés par un simple `EXPLAIN`. Par exemple, nous pourrions avoir un résultat tel que :

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;
```

#### QUERY PLAN

```
Nested Loop (cost=4.65..118.50 rows=10 width=488) (actual time=0.017..0.051 rows=10 loops=1)
-> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.38 rows=10 width=244) (actual time=0.009..0.017 rows=10 loops=1)
    Recheck Cond: (unique1 < 10)
    Heap Blocks: exact=10
-> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.36 rows=10 width=0) (actual time=0.004..0.004 rows=10 loops=1)
    Index Cond: (unique1 < 10)
-> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..7.90 rows=1 width=244) (actual time=0.003..0.003 rows=1 loops=1)
    Index Cond: (unique2 = t1.unique2)
Planning Time: 0.485 ms
Execution Time: 0.073 ms
```

Notes que les valeurs « temps réel » sont en millisecondes alors que les estimations de « coût » sont exprimées dans des unités arbitraires ; il y a donc peu de chances qu'elles correspondent. L'information qu'il faut généralement rechercher est si le nombre de lignes estimées est raisonnablement proche de la réalité. Dans cet exemple, les estimations étaient toutes rigoureusement exactes, mais c'est en pratique plutôt inhabituel.

Dans certains plans de requête, il est possible qu'un nœud de sous-plan soit exécuté plus d'une fois. Par exemple, le parcours d'index interne est exécuté une fois par ligne externe dans le plan de boucle imbriquée ci-dessus. Dans de tels cas, la valeur `loops` renvoie le nombre total d'exécutions du nœud, et le temps réel et les valeurs des lignes affichées sont une moyenne par exécution. Ceci est fait pour que les nombres soient comparables avec la façon dont les estimations de coûts sont affichées. Multipliez par la valeur de `loops` pour obtenir le temps total réellement passé dans le nœud. Dans l'exemple précédent, le parcours d'index sur `tenk2` a pris un total de 0,030 milliseconde.

Dans certains cas, `EXPLAIN ANALYZE` affiche des statistiques d'exécution supplémentaires après le temps et le nombre de lignes de l'exécution d'un nœud du plan. Par exemple, les nœuds de tri et de hachage fournissent des informations supplémentaires :

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;
```

#### QUERY PLAN

```
Sort (cost=713.05..713.30 rows=100 width=488) (actual time=2.995..3.002 rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort Memory: 74kB
-> Hash Join (cost=226.23..709.73 rows=100 width=488) (actual time=0.515..2.920 rows=100 loops=1)
    Hash Cond: (t2.unique2 = t1.unique2)
-> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244) (actual time=0.026..1.790 rows=10000 loops=1)
-> Hash (cost=224.98..224.98 rows=100 width=244) (actual time=0.476..0.477 rows=100 loops=1)
    Buckets: 1024 Batches: 1 Memory Usage: 35kB
-> Bitmap Heap Scan on tenk1 t1 (cost=5.06..224.98 rows=100 width=244) (actual time=0.030..0.450 rows=100 loops=1)
    Recheck Cond: (unique1 < 100)
    Heap Blocks: exact=90
-> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=100 width=0) (actual time=0.013..0.013 rows=100 loops=1)
    Index Cond: (unique1 < 100)
Planning Time: 0.187 ms
Execution Time: 3.036 ms
```

Le nœud de tri donne la méthode de tri utilisée (en particulier, si le tri s'est effectué en mémoire ou sur disque) ainsi que la quantité de mémoire ou d'espace disque requis. Le nœud de hachage montre le nombre de paquets de hachage, le nombre de lots ainsi la quantité maximale de mémoire utilisée pour la table de hachage (si le nombre de lots est supérieur à un, il y aura également l'utilisation de l'espace disque impliqué, mais cela n'est pas montré dans cet exemple).

Un autre type d'information supplémentaire est le nombre de lignes supprimées par une condition de filtrage :

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;
```

#### QUERY PLAN

```
Seq Scan on tenk1 (cost=0.00..470.00 rows=7000 width=244) (actual time=0.030..1.995 rows=7000 loops=1)
  Filter: (ten < 7)
  Rows Removed by Filter: 3000
Planning Time: 0.102 ms
Execution Time: 2.145 ms
```

Ces nombres peuvent être particulièrement précieux pour les conditions de filtres appliquées aux nœuds de jointure. La ligne « Rows Removed » n'apparaît que si au moins une ligne parcourue, ou une ligne potentiellement appariée dans le cas d'un nœud de jointure, est rejetée par la condition de filtre.

Un cas similaire aux conditions de filtre apparaît avec des parcours d'index « avec perte ». Par exemple, regardez cette recherche de polygone contenant un point spécifique :

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';
```

#### QUERY PLAN

```
Seq Scan on polygon_tbl (cost=0.00..1.09 rows=1 width=85) (actual time=0.023..0.023 rows=0 loops=1)
  Filter: (f1 @> '((0.5,2))'::polygon)
  Rows Removed by Filter: 7
Planning Time: 0.039 ms
Execution Time: 0.033 ms
```

Le planificateur pense (plutôt correctement) que cette table d'échantillon est trop petite pour s'embêter avec un parcours d'index, et utilise donc un parcours séquentiel dans lequel toutes les lignes sont rejetées par la condition de filtre. Mais si nous forçons l'utilisation d'un parcours d'index, nous voyons :

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';
```

```
QUERY PLAN
-----
Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=85) (actual time=0.074..0.074 rows=0 loops=1)
  Index Cond: (f1 @> '((0.5,2))'::polygon)
  Rows Removed by Index Recheck: 1
  Planning Time: 0.039 ms
  Execution Time: 0.098 ms
```

L'index retourne une ligne candidate, qui est ensuite rejetée par une deuxième vérification de la condition de l'index. Cela arrive car un index GiST est « avec perte » pour les tests de contenance de polygone : il retourne en fait les lignes pour lesquelles les polygones chevauchent la cible, ce qui nécessite après coup un test de contenance exacte sur ces lignes.

EXPLAIN a une option BUFFERS qui peut être utilisée avec ANALYZE pour obtenir encore plus de statistiques d'exécution:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
```

```
QUERY PLAN
-----
Bitmap Heap Scan on tenk1  (cost=25.07..60.11 rows=10 width=244) (actual time=0.105..0.114 rows=10 loops=1)
  Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
  Heap Blocks: exact=10
  Buffers: shared hit=14 read=3
  -> BitmapAnd (cost=25.07..25.07 rows=10 width=0) (actual time=0.100..0.101 rows=0 loops=1)
    Buffers: shared hit=4 read=3
    -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0) (actual time=0.027..0.027 rows=100 loops=1)
      Index Cond: (unique1 < 100)
      Buffers: shared hit=2
    -> Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0) (actual time=0.070..0.070 rows=999 loops=1)
      Index Cond: (unique2 > 9000)
      Buffers: shared hit=2 read=3
  Planning:
    Buffers: shared hit=3
  Planning Time: 0.162 ms
  Execution Time: 0.143 ms
```

Les nombres fournis par BUFFERS aident à identifier les parties de la requête les plus intensives en termes d'entrées sorties.

Il faut garder en tête que comme EXPLAIN ANALYZE exécute vraiment la requête, tous les effets secondaires se produiront comme d'habitude, même si, quel que soit l'affichage de la requête, il est remplacé par la sortie des données d'EXPLAIN. Si vous voulez analyser une requête modifiant les données sans changer les données en table, vous pouvez annuler les modifications après, par exemple :

```
BEGIN;
EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;
```

```
QUERY PLAN
-----
Update on tenk1  (cost=5.06..225.23 rows=0 width=0) (actual time=1.634..1.635 rows=0 loops=1)
  -> Bitmap Heap Scan on tenk1  (cost=5.06..225.23 rows=100 width=10) (actual time=0.065..0.141 rows=100 loops=1)
    Recheck Cond: (unique1 < 100)
    Heap Blocks: exact=90
    -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0) (actual time=0.031..0.031 rows=100 loops=1)
      Index Cond: (unique1 < 100)
  Planning Time: 0.151 ms
  Execution Time: 1.856 ms

ROLLBACK;
```

Comme vous pouvez le voir dans cet exemple, quand la requête contient une commande INSERT, UPDATE, DELETE ou MERGE, l'application des changements est faite au niveau du nœud principal Insert, Update, Delete ou Merge du plan. Les nœuds du plan sous celui-ci effectuent le travail de recherche des anciennes lignes et/ou le calcul des nouvelles données. Ainsi, au-dessus, on peut voir les mêmes tris de parcours de bitmap déjà vus précédemment, et leur sortie est envoyée à un nœud de mise à jour qui stocke les lignes modifiées. Il est intéressant de noter que bien que le nœud de modification de données puisse prendre une part considérable sur le temps d'exécution (ici, c'est la partie la plus gourmande), le planificateur n'ajoute rien au coût estimé pour considérer ce travail. C'est dû au fait que le travail à effectuer est le même pour chaque plan de requête correct, et n'affecte donc pas les décisions du planificateur.

La phrase Planning time affichée par EXPLAIN ANALYZE correspond au temps pris pour générer et optimiser le plan de requêtes à partir de la requête analysée. Cela n'inclut pas l'analyse syntaxique et la réécriture.

Le Temps total d'exécution donné par EXPLAIN ANALYZE inclut le temps de démarrage et d'arrêt de l'exécuteur, ainsi que le temps d'exécution de tous les triggers pouvant être déclenchés, mais n'inclut pas les temps d'analyse, de réécriture ou de planification. Le temps passé à exécuter les triggers BEFORE, s'il y en a, est inclus dans le temps passé à l'exécution des nœuds Insert, Update ou Delete associés, mais le temps passé à exécuter les triggers AFTER n'est pas compté, car les triggers AFTER sont déclenchés après l'achèvement du plan entier. Le temps total passé dans chaque trigger (que ce soit BEFORE ou AFTER) est affiché séparément. Notez que les triggers de contrainte ne seront pas exécutés avant la fin de la transaction et par conséquent ne seront pas affichés du tout par EXPLAIN ANALYZE.

Le temps affiché sur le premier nœud n'inclut pas le temps nécessaire pour convertir les données en sortie de la requête en leur forme affichable et ne les envoie pas au client. Bien que EXPLAIN ANALYZE n'envoie jamais les données au client, il est possible de lui demander de convertir les données en sortie de la requête en leur forme affichable et de mesurer le temps nécessaire pour ce faire, grâce à l'option SERIALIZE. Ce temps sera affiché séparément, et il sera aussi inclus dans le total de Execution time.

### 14.1.3. Avertissements

Il existe deux raisons importantes pour lesquelles les temps d'exécution mesurés par EXPLAIN ANALYZE peuvent dévier de l'exécution normale de la même requête. Tout d'abord, comme aucune ligne n'est réellement envoyée au client, les coûts de conversion réseau ne sont pas inclus. Les coûts de conversion des

entrées/sorties disque ne sont pas non plus inclus sauf si l'option `SERIALIZE` est demandée. Ensuite, le surcoût de mesure induit par `EXPLAIN ANALYZE` peut être significatif, plus particulièrement sur les machines avec un appel système `gettimeofday()` lent. Vous pouvez utiliser l'outil `pg_test_timing` pour mesurer le surcoût du calcul du temps sur votre système.

Les résultats de `EXPLAIN` ne devraient pas être extrapolés pour des situations autres que celles de vos tests en cours ; par exemple, les résultats sur une petite table ne peuvent être appliqués à des tables bien plus volumineuses. Les estimations de coût du planificateur ne sont pas linéaires et, du coup, il pourrait bien choisir un plan différent pour une table plus petite ou plus grande. Un exemple extrême est celui d'une table occupant une page disque. Vous obtiendrez pratiquement toujours un parcours séquentiel, que des index soient disponibles ou non. Le planificateur réalise que cela va nécessiter la lecture d'un seul bloc disque pour traiter la table dans ce cas, il n'y a donc pas d'intérêt à étendre des lectures de blocs supplémentaires pour un index. (Nous voyons cela arriver dans l'exemple `polygon_tbl` au-dessus.)

Ici, ce sont des cas dans lesquels les valeurs réelles et estimées ne correspondent pas vraiment, mais qui ne sont pas non plus totalement fausses. Un tel cas peut se produire quand un nœud d'exécution d'un plan est arrêté par un `LIMIT` ou autre chose avec un effet similaire. Par exemple, dans la requête `LIMIT` utilisée précédemment,

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
```

QUERY PLAN

-----

Limit (cost=0.29..14.33 rows=2 width=244) (actual time=0.051..0.071 rows=2 loops=1)

-> Index Scan using tenk1\_unique2 on tenk1 (cost=0.29..70.50 rows=10 width=244) (actual time=0.051..0.070 rows=2 loops=1)

Index Cond: (unique2 > 9000)

Filter: (unique1 < 100)

Rows Removed by Filter: 287

Planning Time: 0.077 ms

Execution Time: 0.086 ms

les estimations de coût et de nombre de lignes pour le nœud de parcours d'index sont affichées comme s'ils devaient s'exécuter jusqu'à la fin. Mais en réalité le nœud `Limit` arrête la récupération des lignes après la deuxième ligne récupérée, et donc le vrai nombre de lignes n'est que de 2 et le temps d'exécution est moindre que ne le suggérerait le coût estimé. Ce n'est pas une erreur d'estimation, juste une contradiction entre la façon dont l'estimation et les valeurs réelles sont affichées.

Les jointures d'assemblage ont également leurs artefacts de mesure qui peuvent embrouiller une personne non avertie. Une jointure d'assemblage arrêtera la lecture d'une entrée si l'autre entrée est épuisée et que la prochaine valeur clé dans la première entrée est supérieure à la dernière valeur clé de l'autre entrée ; dans un cas comme ça, il ne peut plus y avoir de correspondance et il est donc inutile de parcourir le reste de la première entrée. Cela a donc pour conséquence de ne pas lire entièrement un des fils, avec des résultats similaires à ceux mentionnés pour `LIMIT`. De même, si le fils externe (premier fils) contient des lignes avec des valeurs de clé dupliquées, le fils externe (second fils) est sauvegardé et les lignes correspondant à cette valeur clé sont parcourues de nouveau. `EXPLAIN ANALYZE` compte ces émissions répétées de mêmes lignes internes comme si elles étaient de vraies lignes supplémentaires. Quand il y a de nombreux doublons externes, le nombre réel de lignes affiché pour le nœud de plan du fils interne peut être significativement plus grand que le nombre de lignes qu'il y a vraiment dans la relation interne.

Les nœuds `BitmapAnd` et `BitmapOr` affichent toujours un nombre de lignes réelles à 0, du fait des limitations d'implémentation.

Habituellement, la sortie d'`EXPLAIN` affichera chaque nœud de plan généré par le planificateur de requêtes. Néanmoins, il existe des cas où l'exécuteur peut déterminer que certains nœuds n'ont pas besoin d'être exécutés car ils ne produisent aucune ligne. (Actuellement, ceci peut n'arriver qu'aux nœuds enfants du nœud `Append` qui parcourt une table partitionnée.) Quand cela arrive, ces nœuds sont omis de la sortie de la commande `EXPLAIN` et une annotation `Subplans Removed: N` apparaît à la place.