Université de Sherbrooke

Systèmes de gestion de bases de données

Exécution et optimisation de requêtes

UdeS:SGBD_03

Christina KHNAISSER *(christina.khnaisser@usherbrooke.ca)*

Luc LAVOIE *(luc.lavoie@usherbrooke.ca)*

(les auteurs sont cités en ordre alphabétique nominal)

—

CoFELI/Scriptorum/SGBD_03-Execution-et-Optimisation (v103), version 1.0.0.a, en date du 2024-08-16

— document de travail, ne pas citer —

Sommaire

…

Mise en garde

Le présent document est en cours d'élaboration ; en conséquence, il est incomplet et peut contenir des erreurs.

Historique

diffusion	resp.	description
2024-08-09	CK	Récupération de notes diverses.

Table des matières

Introduction

Le présent document a pour but de présenter le processus d'évaluation d'une requête par un SGBDR.

Penser comme un SGBD !

Évolution du document

La première version du document a été établie sur les bases des différents travaux publiés par Codd, Darwen, Date, Delobel, Elmasri, Lorentzos, Navathe, Snodgrass et Ullman.

Travail en cours ou projeté

- DONE 2024-09-15 (CK) : première rédaction en s'inspirant de [Date2004a] et [Dombrovskaya2021a].
- DONE 2024-09-15 (CK) : ajout des notes à partir de [Elmasri2026a].
- TODO 2024-09-21 (CK) : ajouter des explications sur la parallélisation [Elmasri2026a] p.683.
- TODO 2024-09-21 (CK) : ajouter des explications sur optimisation avec des vues [Elmasri2026a] p.701.
- TODO 2024-09-21 (CK) : ajouter des explications sur les couts [Elmasri2026a] p.714.

Contenu des sections

- La section 2 présente …

1. Présentation

1.1. Mise en contexte

SQL est un langage déclaratif. C'est-à-dire, lorsque nous écrivons une requête, nous décrivons le résultat que nous cherchons et non comment l'obtenir. En utilisant un SGBD relationnel, les expressions relationnelles telles que définies théoriquement offrent une représentation abstraite facilitant l'optimisation.

La stratégie de mise en œuvre d'une requête est déterminée par le système en se basant sur plusieurs caractéristiques des données dans le catalogue, entre autres :

- le domaine des attributs ;
- la cardinalité de chaque relation de base ;
- le nombre de valeurs distinctes d'un attribut dans une relation ;
- les clés référentielles entre les relations ;

De plus, le traitement d'une requête dépend de plusieurs facteurs, entre autres :

- la modélisation (conceptuel, logique et physique)
- la qualité des données,
- l'utilisation et les exigences non fonctionnelles d'une application.

1.2. Motivation

Quelles sont les activités enseignées par le professeur 465768 ?

Activite(sigle, titre)

Enseignement(matricule, sigle, trimestre)

L'expression relationnelle équivalente :

π *(sigle, titre, trimestre)* φ*(matricule = 465768) (Enseignement* $\bowtie$ *Activite)*

L'expression en SQL :

```
select sigle, titre, trimestre
from Enseignement join Activite
```

```
where matricule = "465768"
```

Caractéristique des données :

- 100 activités
- 10 000 enseignements, dont 50 enseignés par l'enseignant dont le matricule est 465768

Le plan d'exécution sans aucune optimisation :

1. Effectuer la jointure «Enseignement» et «Activite» via «sigle» :
   a. lire les **10 000** tuplets de «Enseignement»,
   b. lire de chacune des **100 activités 10 000 fois**,
   c. construire le résultat intermédiaire **10 000** tuplets (en mémoire).
2. Effectuer la restriction du résultat de (1) matricule = 465768 :
   a. parcourir les **10 000** tuplets vérifier la condition matricule = 465768,
   b. construire le résultat intermédiaire de 50 tuplets.
3. Effectuer la projection du résultat de (2) pour sigle, titre et session :
   a. construire le résultat final de 50 tuplets avec les attributs spécifiés.

Ce plan nécessite 10 000 + (100 * 10 000) + 10 000 + 10 000 = **1 030 000 entreé/sortie**.

Un plan d'exécution avec certaines optimisations :

1. Effecuter la restriction sur «Enseignement» pour la matricule 465768
   a. lire les **10 000** tuplets de «Enseignement»
   b. vérifier la condition matricule = 465768
   c. construire le résultat intermédiaire de 50 tuplets
2. Effectuer la jointure du résultat de (1) avec «Activite»
   a. lire les **100** tuplets de «Activite»
   b. construire le résultat intermédiaire de 50 tuplets
3. Effectuer la projection du résultat de (2) pour sigle, titre et session
   a. construire le résultat final de 50 tuplets avec les attributs spécifiés.

Ce plan nécessite 10 000 + 100 = **10 100 entrée/sortie**.

Ce plan est environ 100 fois plus rapide que le plan sans optimisation.

Un simple changement d'ordre d'exécution des étapes peut avoir un grand impact sur la performance. L'amélioration pourra être en plus grande si nous disposons d'index.

Dans la pratique, on mesure les entrées/sorties de bloc.

1.3. Définition du processus

Le processus de traitement d'une requête se divise en 3 étapes :

1. Évaluation de la requête
2. Optimisation de la requête
3. Exécution du plan d'exécution

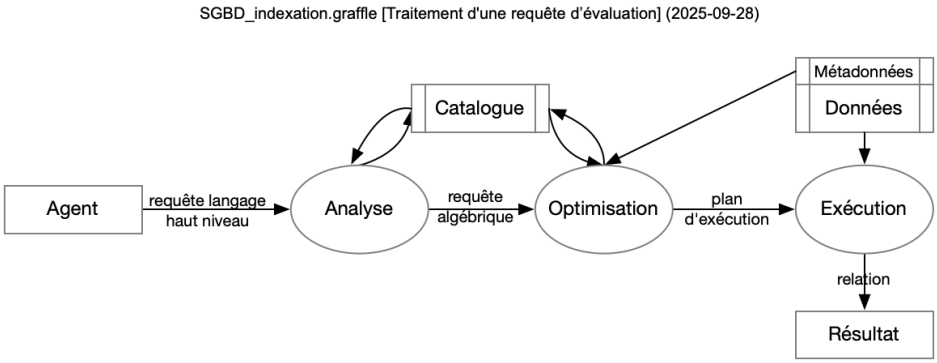


Figure 1. Processus de traitement de requêtes.

Évaluation de la requête

L'évaluation consiste à vérifier la construction syntaxique de la requête et à transformer la requête en une représentation interne.

Optimisation de la requête

L'optimisation consiste à choisir la meilleure stratégie de mise en œuvre pour exécuter une expression relationnelle (exécuter une requête) selon les caractéristiques de la base de données.

Exécution du plan d'exécution

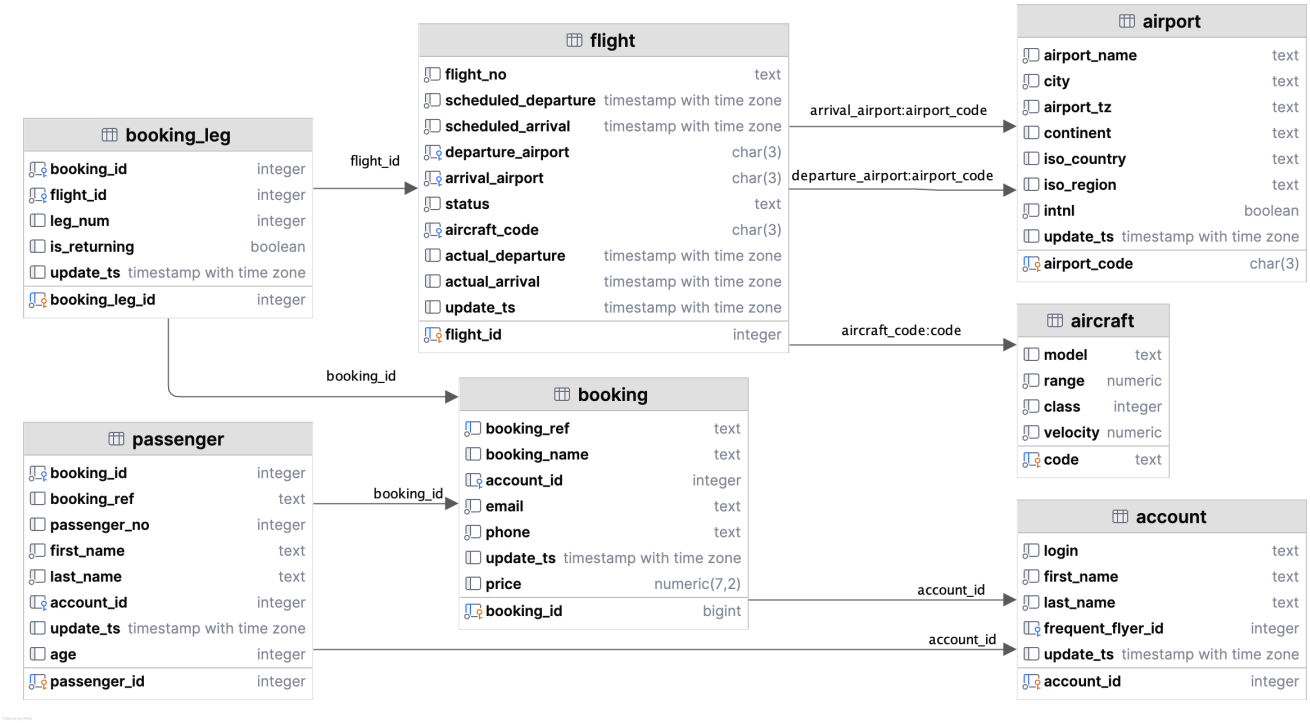
L'exécution consiste à construire la relation de la requête en suivant le plan d'exécution.

1.4. Exemple

Base de données d'une compagnie aérienne.

- Documentation des aéroports et des avions
- Planification des vols
- Réservation des vols

- Account : 864 962 tuplets
- Flight : 683 178 tuplets
- Airport : 692 tuplets
- Booking : 5 643 216 tuplets
- Booking_leg : 17 893 566 tuplets
- Boarding_pass : 25 293 491 tuplets
- Passenger : 16 313 693 tuplets



2. Évaluation de la requête

L'évaluation de la requête s'effectue en 2 étapes :

1. Analyse syntaxique de la requête.
2. Transformation de la requête en forme algébrique.

2.1. Analyse syntaxique

L'analyse syntaxique consiste à vérifier la syntaxe de la requête décrite selon un langage de haut niveau (comme SQL) :

1. Décomposition de la requête en plusieurs parties. Une partie est formée d'une seule expression SELECT-FROM-WHERE[GROUP BY-HAVING].
2. Vérification de la structure de la requête selon les règles syntaxiques de la grammaire.
3. Identification des termes de la requête (ex. les mots-clés, les identifiants des relations et des attributs) et leur existence dans la base de données.

2.2. Transformation en forme algébrique

La transformation de la requête en forme algébrique consiste à convertir le langage de haut niveau en une ou plusieurs expressions relationnelles et à construire l'arbre de requête (la représentation interne).

- Restriction (σ)
- Sélection : jointure ($\bowtie$)
- Projection (π)
- Renommage (ρ)
- Opération ensemblistes : union ($\cup$), intersection ($\cap$) et différence ($-$)
- Agrégation et groupement (ϑ)
- Ordonnancement (τ)

Chaque partie de la requête est transformée en une expression algébrique. Les expressions algébriques sont utilisées pour construire une représentation interne qui prend la forme d'un arbre également nommée l'arbre de requête.

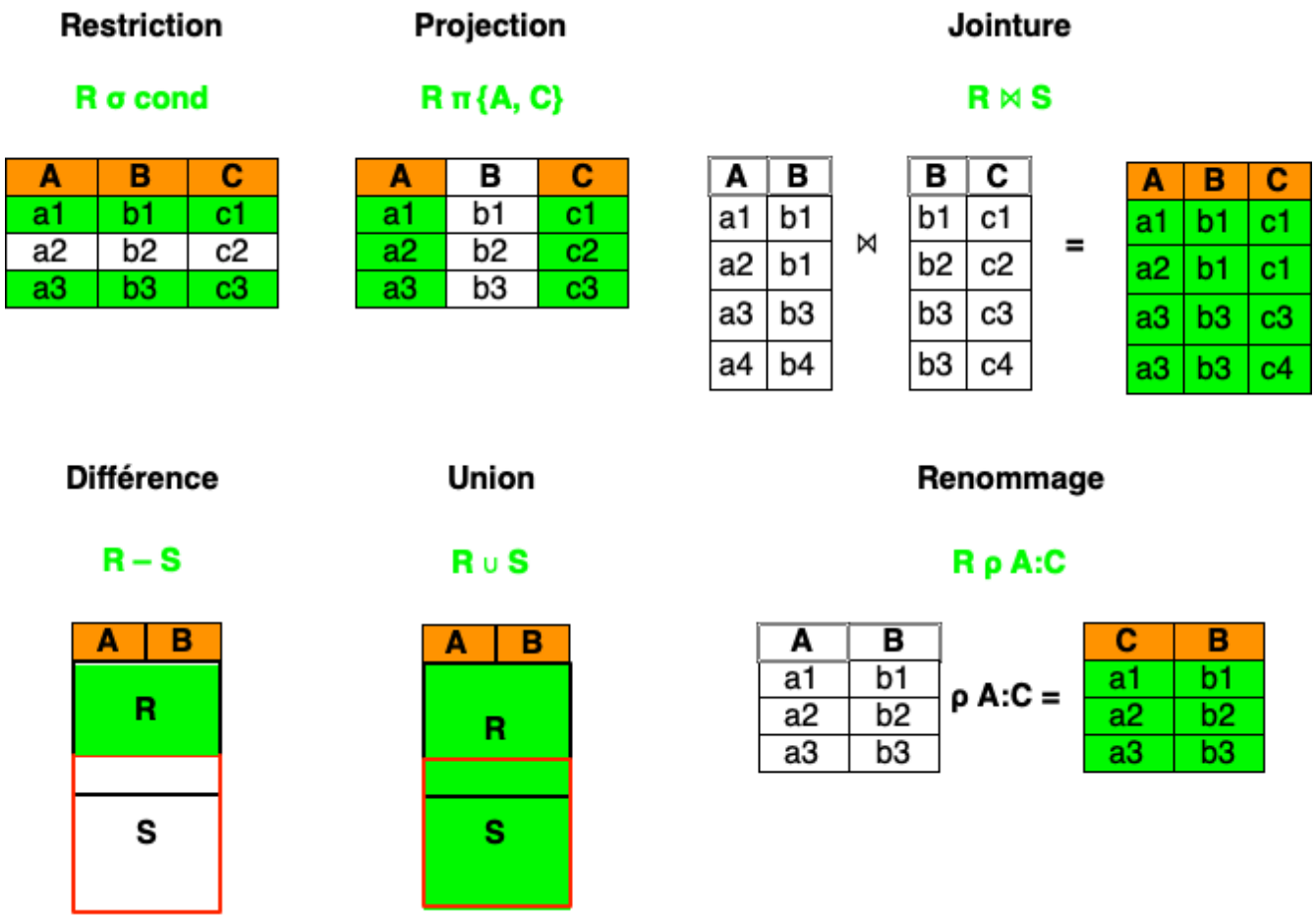


Figure 2. Illustration des opérateurs relationnels usuels

Exemple d'une requête avec deux parties

```
select model
from Aircraft
where velocity > ( select max(velocity)
                   from Aircraft
                   where class = 2 );
```

La requête engendre deux expressions relationnelles :

$$s \longleftarrow \vartheta_{MAX(velocity)} (\sigma_{(class=2} (Aircraft))$$

$$r \longleftarrow \pi_{(model)} (\sigma_{(velocity>s} (Aircraft))$$

Rappelons nous, toute opération relationnelle (de l'aglèbre relationnelle) prend une ou plusieurs relations comme arguments et produit une autre relation comme résultat.

2.2.1. Projection

La projection consiste à choisir une liste d'attribut pour une relation.

La projection permet également d'éliminer les doublons (DISTINCT).

2.2.2. Restriction

Une restriction est une opération de recherche visant à localiser les enregistrements qui satisfont une certaine condition.

- Restriction sur une relation (WHERE)
- Restriction sur un groupement (HAVING)

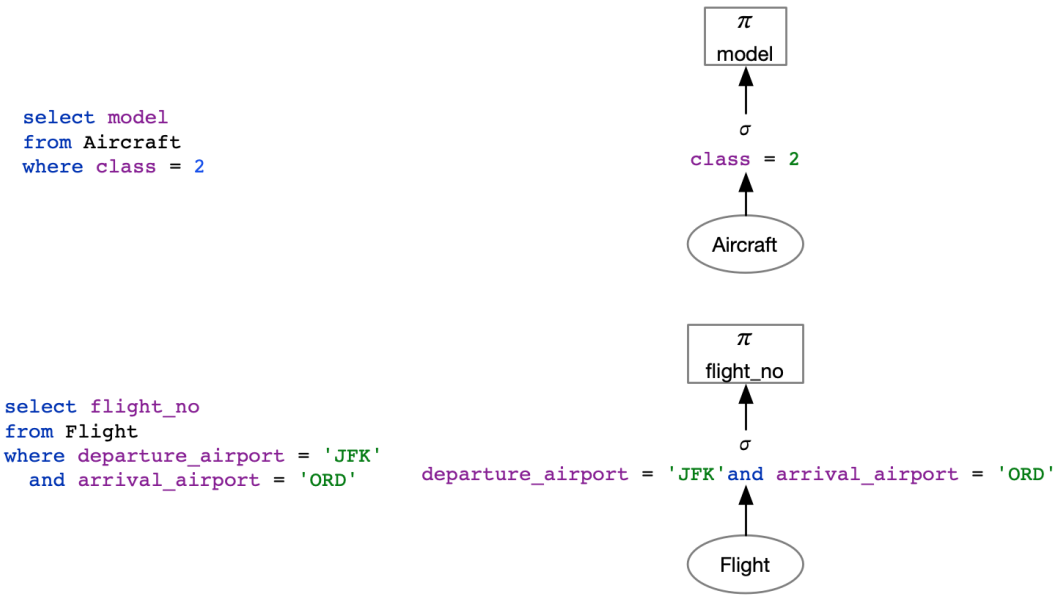


Figure 3. Exemple avec un restriction

2.2.3. Sélection

La sélection est une opération de combinaison de tuplets des relations d'entrée :

- **Produit cartésien** (CROSS) est l'ensemble de tous les combinaisons des tuplets.
- **Jointure naturelle** (JOIN) combine les tuplets des ayant des valeurs égales sur les attributs de jointure.
- **Semi-jointure** (LEFT/RIGHT JOIN IS NOT NULL, EXISTS, IN et ANY) : soit deux relations R et S, l'opération retient les tuplets de la relation R pour lesquelles il **existe au moins un tuplet** dans la relation S ayant des valeurs égales sur les attributs de jointure.
- **Anti-jointure** (LEFT/RIGHT JOIN IS NULL, NOT EXISTS, NOT IN, et ALL) : soit deux relations R et S, l'opération retient les tuplets de la relation R pour lesquelles il **n'existe aucun** tuplet dans la relation S ayant des valeurs égales sur les attributs de jointure.

2.2.4. Exemple — jointure naturelle

Requête

```
select f.flight_id
     , f.scheduled_departure
from Flight f
  join Aircraft on (aircraft_code = code)
  join Airport  on (departure_airport = airport_code)
where iso_country = 'US';
```

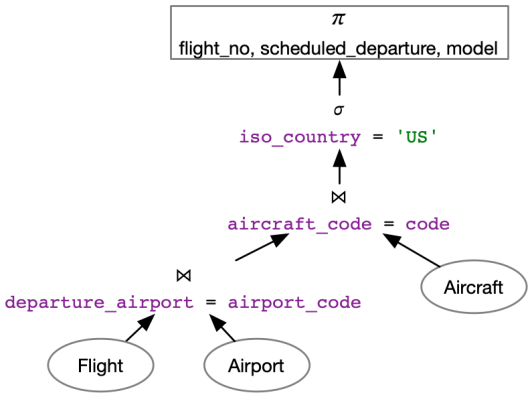


Figure 4. Arbre de requête

2.2.5. Exemple — semi-jointure

Requête

```
select *
from Flight f
where exists( select flight_id
              from Booking_leg
              where flight_id = f.flight_id );
```

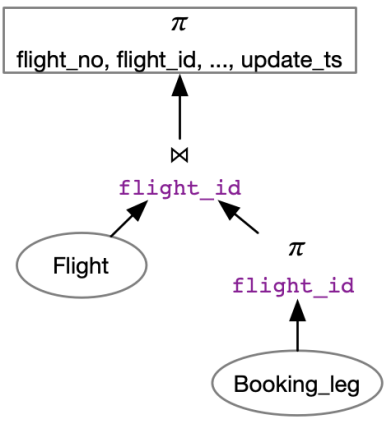


Figure 5. Arbre de requête

2.2.6. Opérations ensemblistes

Les opérations ensemblistes :

- Union : peut remplacer une restriction disjonctive (avec des OR).
- Intersection : peut remplacer une opération de semi-jointure.
- Différence : peut remplacer une opération d'anti-jointure.

Requête

```
select flight_id from Flight
intersect
select flight_id from Booking_leg;
```

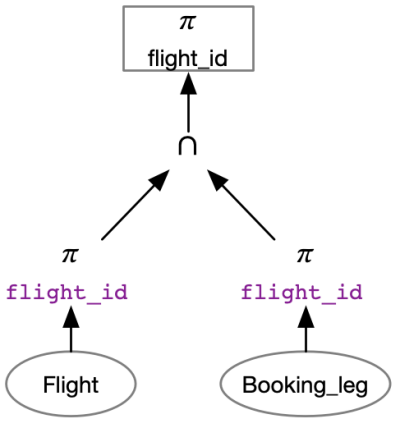


Figure 6. Arbre de requête

2.2.7. Ordonnancement

L'ordonnancement est une opération clé des algorithmes de jointure (*sort-merge algorithms*), des algorithmes ensemblistes (union, intersection et différence) et pour l'élimination de doublon de la projection.

Requête avec ordonnencement

```
select flight_no, scheduled_departure
from flight
order by scheduled_departure
```

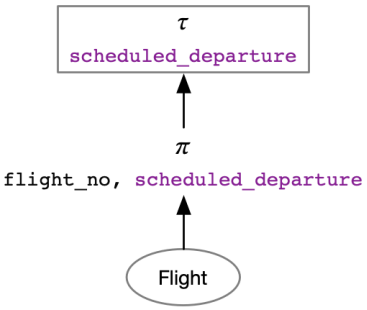


Figure 7. Arbre de requête

2.2.8. Agrégation

L'agrégation est une opération qui permet de calculer une fonction sur un ensemble de tuplets (COUNT, MIN, MAX, SUM, AVG).

L'ensemble de tuplets peut être groupé selon un ou plusieurs attributs. Dans ce cas, la fonction d'agrégation est calculée sur chaque groupement.

Exemple agrégation sans groupement

```
select model
from Aircraft
where velocity > ( select max(velocity)
                   from Aircraft
                   where class = 2 );
```

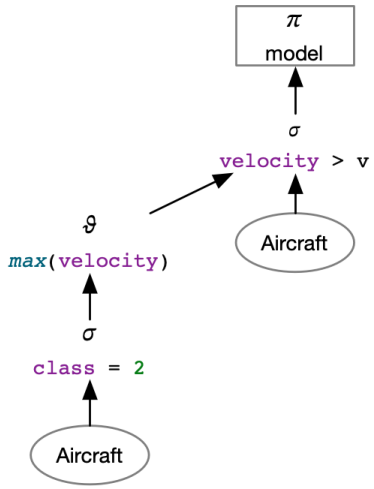


Figure 8. Arbre de requête

Exemple agréagation avec groupement

```
select iso_country, count(airport_code)
from Airport
group by iso_country
having count(airport_code) > 2
order by 2;
```

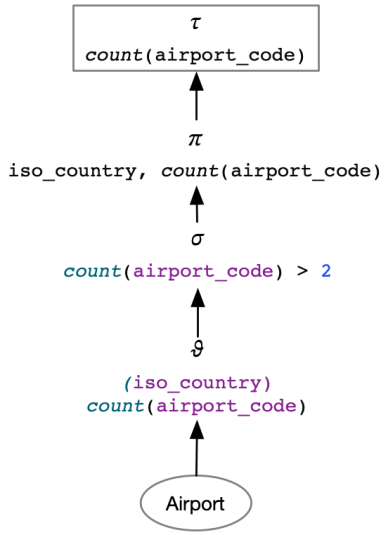


Figure 9. Arbre de requête

3. Optimisation de la requête

L'optimisation de la requête s'effectue en plusieurs étapes :

1. Détermination de l'ordre d'exécution des opérations relationnelles.
2. Détermination de l'algorithme d'exécution pour chaque opération relationnelle.
3. Choix d'un le plan d'exécution optimal (comparer les couts de chaque plan d'exécution).

Parfois le cout de construction d'un plan optimal est plus couteux de l'exécution de la requête elle-même. Dans ce cas, le SGBD va choisir un plan raisonable.

Avec PostgreSQL :

```
explain (analyse, costs, verbose)
```

L'objectif principal est de parvenir au plan le plus efficace et le plus rentable en utilisant les informations disponibles sur le schéma et le contenu des relations impliquées dans la requête dans un délai raisonnable. Autrement dit, le programme d'optimisation décide de la meilleure façon d'exécuter la requête en prenant en considération plusieurs informations telles que les index, l'emplacement des données, la distribution des valeurs des attributs, les cardinalités des relations impliquées, etc.

Un plan de requête est une séquence d'opération. Une opération est mise en œuvre par un algorithme. L'optimiseur essaye de choisir le plus optimal.

Parfois, une opération relationnelle est remplacée par plusieurs opérations physiques, ou plusieurs opérations relationnelles sont fusionnées en une seule opération physique.

3.1. Algorithmes d'accès aux données

Pour exécuter les algorithmes, les données doivent être extraites.

Des algorithmes d'accès aux données sont utilisés :

- **Parcours complet (*full table scan*)** : lecture consécutivement toutes les lignes d'une table et vérifie la condition de filtrage pour chaque ligne.
- **Parcours à partir d'un index (*index based table scan*)** : lecture de chaque entrée de l'index qui satisfait à la condition de restriction et récupère les blocs dans l'ordre de l'index.
- **Parcours d'un index seulement (*index-only scan*)** : si l'index contient tous les attributs nécessaires pour la requête, il suffit de vérifier si l'entrée de l'index satisfait la condition de restriction.

3.2. Notion de Sélectivité

L'efficacité d'une opération dépend du taux de sélectivité de l'opération.

La sélectivité est définie comme le rapport entre le nombre de tuplets à traiter (qui satisfont à la condition) et le nombre total de tuplets dans la relation.

Plus la sélectivité est basse, plus les recherches sont rapides.

Souvent, la création d'un index améliore la sélectivité de la requête.

La sélectivité d'un index est le nombre de valeurs uniques dans l'index par rapport au nombre total d'entrées dans l'index.

Pour créer un index efficace, la sélectivité d'un index doit être basse.

Autrement dit, plus le nombre d'occurrences d'une valeur correspondant à une valeur de l'index est faible, plus la valeur de sélectivité de l'index est basse.

Mais dans certains cas, il **NE** faut **PAS** d'utiliser les index :

- Lorsque la relation est de petite taille : une table qui peut être entièrement en mémoire.
- Lorsque l'extraction nécessite une grande partie des tuplets d'une relation.

Normalement, un bon optimiseur de requête peut détecter ces situations.

3.3. Projection

Une projection est mise en œuvre par un des algorithmes suivant :

- Si la liste des attributs contient la clé,
  - Parcours linéaire (*sequential scan*) :

    - Sinon
- Parcours linéaire avec ordonnancement : ordonne les tuplets pour éliminer les doublons.
- Parcours avec adressage dispersé : chaque tuplet est haché puis inséré dans un bloc, si le tuplet existe déjà le tuplet ne sera pas inséré.

3.3.1. Exemple - projection sur tous les attributs

Requête

```
select * from flight;
```

Plan d'exécution avant index

```
Seq Scan on postgres_air.flight (cost=0.00..15455.78 rows=683178 width=71)
  (actual time=0.017..55.383 rows=683178 loops=1)
  " Output: flight_id, flight_no, scheduled_departure, scheduled_arrival,
departure_airport, arrival_airport, status, aircraft_code, actual_departure,
actual_arrival, update_ts"
Planning Time: 0.906 ms
Execution Time: 78.256 ms
```

3.3.2. Exemple - projection sur les attributs non-clé

Requête

```
select distinct departure_airport
from flight;
```

Plan d'exécution avant index

```
Unique  (actual time=77.151..77.336 rows=647 loops=1)
  Output: departure_airport
  ->  Sort  (actual time=77.151..77.221 rows=1939 loops=1)
        Output: departure_airport
        Sort Key: flight.departure_airport
        Sort Method: quicksort  Memory: 79kB
        ->  Gather (actual time=70.329..70.487 rows=1939 loops=1)
              Output: departure_airport
              ->  HashAggregate (actual time=62.724..62.764 rows=646 loops=3)
                    Output: departure_airport
                    Group Key: flight.departure_airport
                    Memory Usage: 73kB
                    ->  Parallel Seq Scan on postgres_air.flight (actual
time=0.095..22.446 rows=227726 loops=3)
"                         Output: flight_id, flight_no, scheduled_departure,
scheduled_arrival, departure_airport, arrival_airport, status, aircraft_code,
actual_departure, actual_arrival, update_ts"
Planning Time: 0.138 ms
Execution Time: 77.532 ms
```

Index

```
create index flight_departure_airport
  on flight(departure_airport);
```

Plan d'exécution après index

```
Unique (actual time=51.804..52.858 rows=647 loops=1)
  Output: departure_airport
  ->  Gather Merge (actual time=51.803..52.756 rows=884 loops=1)
        Output: departure_airport
        ->  Unique (actual time=0.160..27.725 rows=295 loops=3)
              Output: departure_airport
              ->  Parallel Index Only Scan using flight_departure_airport on
postgres_air.flight
                    (actual time=0.159..14.633 rows=227726 loops=3)
                      Output: departure_airport
Planning Time: 1.328 ms
Execution Time: 52.941 ms
```

3.4. Restriction

La restriction est effectuée en premier pour réduire le nombre de tuplets des résultats intermédiaires.

L'optimiseur réordonne les nœuds de l'arbre de la requête pour placer la restriction avec la plus basse sélectivité en premier.

Une restriction est mise en œuvre par un des algorithmes de recherche suivant :

- Recherche sur un attribut :
  - Recherche linéaire (*sequential scan*) : récupère chaque enregistrement du fichier et vérifie si les valeurs des attributs satisfont à la condition de restriction.
  - Recherche binaire : si la condition implique une **comparaison d'égalité** avec un attribut sur lequel **le fichier est ordonné**.
  - Recherche par l'index primaire : si la condition implique une **comparaison d'égalité, <, >, ⇐ ou >=** avec un **attribut clé**.
  - Recherche par un index de classification : si la condition implique une **comparaison d'égalité** avec un **attribut non-clé**.
  - Recherche par un index secondaire : si la condition implique une **comparaison d'égalité, <, >, ⇐ ou >=** avec un **attribut clé ou non-clé indexé**.
  - Recherche par un bitmap : si la condition implique une ou une combinaison de **comparaison d'égalité** (AND, OR) avec un ensemble de valeur d'un attribut.
- Recherche sur plusieurs attributs :
  - Recherche par plusieurs index :
    - si la condition implique une restriction conjonctive (avec des AND) une recherche sur chaque attribut est effectuée séparément, à la fin une intersection des résultats est effectuée.
    - si la condition implique une restriction disjonctive (avec des OR) une recherche linéaire est utilisée si une des conditions n'a pas un index (un autre type de recherche si toutes les conditions sont indexées), à la fin une union des résultats est effectuée.
  - Recherche par un index composite : si un index composite existe sur les attributs.

3.4.1. Exemple — restriction utilisant une fonction

Requête

```
select *
from account
where lower(last_name) = 'daniels';
```

Plan d'exécution avant index

```
Gather (actual time=1.055..84.202 rows=764 loops=1)
  ->  Parallel Seq Scan on account (actual time=0.233..71.291 rows=255 loops=3)
        Filter: (lower(last_name) = 'daniels'::text)
        Rows Removed by Filter: 288066
Planning Time: 1.881 ms
Execution Time: 84.302 ms
```

Index sur last_name de la relation Account

```
create index account_lower_last_name
  on account(lower(last_name));
```

Plan d'exécution avec un index

```
Bitmap Heap Scan on account (actual time=0.224..3.352 rows=764 loops=1)
  Recheck Cond: (lower(last_name) = 'daniels'::text)
  Heap Blocks: exact=695
  ->  Bitmap Index Scan on account_lower_last_name (actual time=0.135..0.135 rows=764
loops=1)
        Index Cond: (lower(last_name) = 'daniels'::text)
Planning Time: 0.408 ms
Execution Time: 3.416 ms
```

3.4.2. Exemple — restriction sur un attribut non-clé

Requête

```
select flight_id, status
from Flight
where status = 'Canceled';
```

Plan d'exécution sans un index

```
Gather (actual time=19.238..40.254 rows=171 loops=1)
"  Output: flight_id, flight_no, scheduled_departure, scheduled_arrival,
departure_airport, arrival_airport, status, aircraft_code, actual_departure,
actual_arrival, update_ts"
  ->  Parallel Seq Scan on postgres_air.flight (actual time=11.463..30.442 rows=57
loops=3)
"        Output: flight_id, flight_no, scheduled_departure, scheduled_arrival,
departure_airport, arrival_airport, status, aircraft_code, actual_departure,
actual_arrival, update_ts"
        Filter: (flight.status = 'Canceled'::text)
        Rows Removed by Filter: 227669
Planning Time: 0.128 ms
Execution Time: 40.298 ms
```

Index sur status

```
create index flight_status
  on Flight(status);
```

Plan d'exécution avec index

```
Index Scan using flight_status on postgres_air.flight
  (actual time=0.087..0.616 rows=171 loops=1)
"  Output: flight_id, status"
  Index Cond: (flight.status = 'Canceled'::text)
Planning Time: 1.137 ms
Execution Time: 0.683 ms
```

3.4.3. Exemple — restriction conjonctive

Requête

```
select flight_no
from Flight
where departure_airport = 'JFK'
  and arrival_airport = 'ORD';
```

Plan d'exécution sans index

```
Gather (actual time=0.581..43.789 rows=182 loops=1)
  Output: flight_no
  ->  Parallel Seq Scan on postgres_air.flight (actual time=0.534..34.968 rows=61
loops=3)
        Output: flight_no
        Filter: ((flight.departure_airport = 'JFK'::bpchar) AND (flight.arrival_airport
= 'ORD'::bpchar))
        Rows Removed by Filter: 227665
Planning Time: 0.217 ms
Execution Time: 43.833 ms
```

Ajout d'un index sur departure_airport

```
create index flight_departure_airport
  on flight(departure_airport);
```

Plan d'exécution avec index 1

```
Bitmap Heap Scan on postgres_air.flight (actual time=1.962..22.781 rows=182 loops=1)
  Output: flight_no
  Recheck Cond: (flight.departure_airport = 'JFK'::bpchar)
  Filter: (flight.arrival_airport = 'ORD'::bpchar)
  Rows Removed by Filter: 10348
  ->  Bitmap Index Scan on flight_departure_airport (actual time=0.874..0.874 rows=10530
loops=1)
        Index Cond: (flight.departure_airport = 'JFK'::bpchar)
Planning Time: 0.244 ms
Execution Time: 22.815 ms
```

Ajout d'un index sur arrival_airport

```
create index flight_arrival_airport
  on flight(arrival_airport);
```

Plan d'exécution avec index 1

```
Bitmap Heap Scan on postgres_air.flight  (actual time=7.798..8.492 rows=182 loops=1)
  Output: flight_no
  Recheck Cond: ((flight.departure_airport = 'JFK'::bpchar) AND (flight.arrival_airport
= 'ORD'::bpchar))
  ->  BitmapAnd  (actual time=7.741..7.742 rows=0 loops=1)
        ->  Bitmap Index Scan on flight_departure_airport (actual time=4.285..4.285
rows=10530 loops=1)
              Index Cond: (flight.departure_airport = 'JFK'::bpchar)
        ->  Bitmap Index Scan on flight_arrival_airport (actual time=3.195..3.195
rows=12922 loops=1)
              Index Cond: (flight.arrival_airport = 'ORD'::bpchar)
Planning Time: 1.878 ms
Execution Time: 8.588 ms
```

Ajout d'un index composite

```
create index flight_departure_arrival_airport
  on flight(departure_airport, arrival_airport);
```

Plan d'exécution avec index 3

```
Bitmap Heap Scan on postgres_air.flight (actual time=0.078..0.288 rows=182 loops=1)
  Output: flight_no
  Recheck Cond: ((flight.departure_airport = 'JFK'::bpchar) AND (flight.arrival_airport
= 'ORD'::bpchar))
  ->  Bitmap Index Scan on fligt_departure_arrival_airport (actual time=0.057..0.058
rows=182 loops=1)
        Index Cond: ((flight.departure_airport = 'JFK'::bpchar) AND
(flight.arrival_airport = 'ORD'::bpchar))
Planning Time: 0.387 ms
Execution Time: 0.313 ms
```

3.5. Sélection

Une jointure est mise en œuvre par un des algorithmes suivant :

- Boucle imbriquée (*nested-loop join*) : combine des tuplets de deux ou plusieurs relations en utilisant des conditions aléatoires.
- Boucle imbriquée par index : si un index est défini pour un des attributs de jointure la boucle externe s'effectue sur la relation sans l'index et la boucle interne sur la relation ayant l'index.
- Tri-fusion (*Sort-merge join*) : si les tuplets des relations sont physiquement triés par l'attribut de jointure les deux relations sont parcourues en parallèle, puis chaque tuplet est fusionné si la condition de jointure est satisfaite. Parfois le SGBD décide de trier avant de faire le parcours.
- Adressage dispersé (*Hash join*) : le fichier des enregistrements de chaque relation est décomposé en plusieurs partitions. Le partitionnement est effectué sur le premier fichier par une fonction de hachage sur l'ensemble des attributs de jointure (*partitionning phase*). Ensuite, le partitionnement est effectué sur le deuxième fichier par la même fonction de hachage (*probing phase*). Tous les tuplets ayant la même valeur sur l'ensemble des attributs de hachage sont stockés dans la même partition.

Une requête joignant n relations aura n-1 opérations de jointure et n! ordres de jointure différents. Les optimiseurs de requêtes limitent généralement la structure d'un arbre de requête (de jointure) à celle d'arbres profonds à gauche (ou à droite). L'arbre profond à gauche, c'est un arbre binaire où l'enfant de droite de chaque nœud non-feuille est une relation de base. Ce type d'arbre est optimal parce que

l'optimiseur peut utiliser le chemin d'accès le plus optimal sur la relation de base (l'enfant de droite) pour exécuter la jointure.

3.5.1. Exemple — jointure naturelle

Requête

```
select f.flight_id
     , f.scheduled_departure
from Flight f
  join Airport a on (f.departure_airport = a.airport_code)
where iso_country = 'US'; -- iso_country = 'CZ'
```

Plan d'exécution

```
Hash Join (actual time=0.214..123.990 rows=338858 loops=1)
"  Output: f.flight_id, f.scheduled_departure"
  Inner Unique: true
  Hash Cond: (f.departure_airport = a.airport_code)
  ->  Seq Scan on postgres_air.flight f  (actual time=0.008..39.602 rows=683178 loops=1)
"        Output: f.flight_id, f.flight_no, f.scheduled_departure, f.scheduled_arrival,
f.departure_airport, f.arrival_airport, f.status, f.aircraft_code, f.actual_departure,
f.actual_arrival, f.update_ts"
  ->  Hash (actual time=0.199..0.200 rows=141 loops=1)
        Output: a.airport_code
        Buckets: 1024  Batches: 1  Memory Usage: 13kB
        ->  Seq Scan on postgres_air.airport a (actual time=0.034..0.170 rows=141
loops=1)
              Output: a.airport_code
              Filter: (a.iso_country = 'US'::text)
              Rows Removed by Filter: 551
Planning Time: 0.948 ms
Execution Time: 133.188 ms
```

Index seondaire sur iso_country

```
create index airport_country
  on Airport(iso_country);
```

Plan d'exécution avec index 1

```
Hash Join (actual time=0.195..122.490 rows=338858 loops=1)
"  Output: f.flight_id, f.scheduled_departure"
  Inner Unique: true
  Hash Cond: (f.departure_airport = a.airport_code)
  ->  Seq Scan on postgres_air.flight f  (actual time=0.014..41.557 rows=683178 loops=1)
"        Output: f.flight_id, f.flight_no, f.scheduled_departure, f.scheduled_arrival,
f.departure_airport, f.arrival_airport, f.status, f.aircraft_code, f.actual_departure,
f.actual_arrival, f.update_ts"
  ->  Hash (actual time=0.160..0.162 rows=141 loops=1)
        Output: a.airport_code
        Buckets: 1024  Batches: 1  Memory Usage: 13kB
        ->  Bitmap Heap Scan on postgres_air.airport a  (actual time=0.056..0.128
rows=141 loops=1)
              Output: a.airport_code
              Recheck Cond: (a.iso_country = 'US'::text)
              Heap Blocks: exact=9
              ->  Bitmap Index Scan on airport_country  (actual time=0.037..0.037
```

```
rows=141 loops=1)
                    Index Cond: (a.iso_country = 'US'::text)
Planning Time: 1.382 ms
Execution Time: 131.283 ms
```

Index de couverture departure_airport

```
create index flight_departure_airport
 on flight(departure_airport) include (flight_id, scheduled_departure);
```

Plan d'exécution avec index 2

```
Nested Loop (actual time=0.066..68.508 rows=338858 loops=1)
"  Output: f.flight_id, f.scheduled_departure"
  ->  Seq Scan on postgres_air.airport a (actual time=0.016..0.116 rows=141 loops=1)
"        Output: a.airport_code, a.airport_name, a.city, a.airport_tz, a.continent,
a.iso_country, a.iso_region, a.intnl, a.update_ts"
        Filter: (a.iso_country = 'US'::text)
        Rows Removed by Filter: 551
  ->  Index Only Scan using flight_departure_airport on postgres_air.flight f (actual
time=0.017..0.302 rows=2403 loops=141)
"        Output: f.departure_airport, f.flight_id, f.scheduled_departure"
        Index Cond: (f.departure_airport = a.airport_code)
        Heap Fetches: 0
Planning Time: 0.952 ms
Execution Time: 80.529 ms
```

3.5.2. Exemple - jointure 3 relations

Requête

```
select f.flight_id
     , f.scheduled_departure
     , model
from flight f
  join aircraft on (aircraft_code = code)
  join airport on  (departure_airport = airport_code)
where  iso_country = 'US';
```

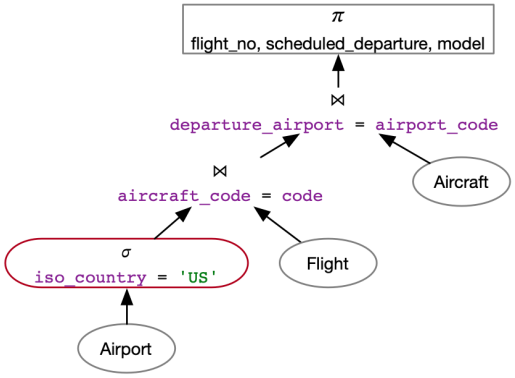


Figure 10. Arbres de requête plan 1

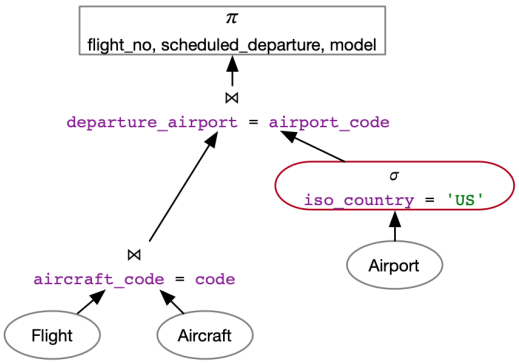


Figure 11. Arbres de requête plan 2

Plan d'exécution

```
Gather  (cost=1021.68..14172.81 rows=8352 width=12) (actual time=1.239..114.615
rows=338858 loops=1)
"  Output: f.flight_id, f.scheduled_departure"
  Workers Planned: 2
  Workers Launched: 2
  ->  Hash Join  (cost=21.68..12337.61 rows=3480 width=12) (actual time=1.037..100.783
rows=112953 loops=3)
"        Output: f.flight_id, f.scheduled_departure"
        Inner Unique: true
        Hash Cond: (f.departure_airport = airport.airport_code)
        Worker 0:  actual time=0.303..98.899 rows=116422 loops=1
        Worker 1:  actual time=1.856..106.109 rows=120458 loops=1
        ->  Hash Join  (cost=1.27..12272.09 rows=17080 width=16) (actual
time=0.576..77.823 rows=227726 loops=3)
"              Output: f.flight_id, f.scheduled_departure, f.departure_airport"
              Inner Unique: true
              Hash Cond: ((f.aircraft_code)::text = aircraft.code)
              Worker 0:  actual time=0.154..76.313 rows=234745 loops=1
              Worker 1:  actual time=1.504..82.070 rows=240805 loops=1
              ->  Parallel Seq Scan on postgres_air.flight f  (cost=0.00..11470.58
rows=284658 width=20) (actual time=0.055..17.123 rows=227726 loops=3)
"                    Output: f.flight_id, f.flight_no, f.scheduled_departure,
f.scheduled_arrival, f.departure_airport, f.arrival_airport, f.status, f.aircraft_code,
f.actual_departure, f.actual_arrival, f.update_ts"
                    Worker 0:  actual time=0.028..16.994 rows=234745 loops=1
                    Worker 1:  actual time=0.097..18.199 rows=240805 loops=1
              ->  Hash  (cost=1.12..1.12 rows=12 width=32) (actual time=0.043..0.044
rows=12 loops=3)
                    Output: aircraft.code
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    Worker 0:  actual time=0.028..0.028 rows=12 loops=1
                    Worker 1:  actual time=0.086..0.086 rows=12 loops=1
                    ->  Seq Scan on postgres_air.aircraft  (cost=0.00..1.12 rows=12
width=32) (actual time=0.034..0.036 rows=12 loops=3)
                          Output: aircraft.code
                          Worker 0:  actual time=0.022..0.023 rows=12 loops=1
                          Worker 1:  actual time=0.073..0.076 rows=12 loops=1
        ->  Hash  (cost=18.65..18.65 rows=141 width=4) (actual time=0.424..0.425
rows=141 loops=3)
              Output: airport.airport_code
              Buckets: 1024  Batches: 1  Memory Usage: 13kB
              Worker 0:  actual time=0.115..0.115 rows=141 loops=1
              Worker 1:  actual time=0.283..0.284 rows=141 loops=1
              ->  Seq Scan on postgres_air.airport  (cost=0.00..18.65 rows=141 width=4)
(actual time=0.043..0.399 rows=141 loops=3)
                    Output: airport.airport_code
                    Filter: (airport.iso_country = 'US'::text)
```

```
                    Rows Removed by Filter: 551
                    Worker 0:  actual time=0.022..0.099 rows=141 loops=1
                    Worker 1:  actual time=0.094..0.248 rows=141 loops=1
Planning Time: 1.088 ms
Execution Time: 123.940 ms
```

3.6. Opérations ensembliste

Pour les opérations ensemblistes des variantes de l'algorithme de tri-fusion (*sort-merge*) sont utilisées :

1. les relations sont ordonnées selon les mêmes attributs
2. un parcours linéaire sur chaque relation est effectué (en parallèle) pour comparer les tuplets :
   a. pour l'union, si les tuplets sont identiques un des deux tuplets est gardé.
   b. pour l'intersection, si les tuplets sont identiques un des deux tuplets est gardé sinon les deux sont éliminé.
   c. pour la différence, si les tuplets sont identiques, ils sont éliminés.

3.6.1. Exemple — intersection

Requête

```
select flight_id
from Flight f
intersect
select flight_id
from Booking_leg;
```

Plan d'exécution avec INTERSECT

```
HashSetOp Intersect (actual time=3399.830..3415.690 rows=508844 loops=1)
"  Output: ""*SELECT* 2"".flight_id, (1)"
  ->  Append (actual time=0.012..2247.227 rows=18576744 loops=1)
"        ->  Subquery Scan on ""*SELECT* 2"" (actual time=0.011..1525.269 rows=17893566
loops=1)"
"              Output: ""*SELECT* 2"".flight_id, 1"
              ->  Seq Scan on postgres_air.booking_leg (actual time=0.009..804.481
rows=17893566 loops=1)
                    Output: booking_leg.flight_id
"        ->  Subquery Scan on ""*SELECT* 1"" (actual time=0.010..58.763 rows=683178
loops=1)"
"              Output: ""*SELECT* 1"".flight_id, 0"
              ->  Seq Scan on postgres_air.flight f (actual time=0.009..31.605
rows=683178 loops=1)
                    Output: f.flight_id
Planning Time: 0.087 ms
Execution Time: 3429.350 ms
```

Requête

```
select *
from Flight f
where exists( select flight_id
              from Booking_leg
              where flight_id = f.flight_id);
```

Plan d'exécution avec EXISTS

```
Hash Semi Join (actual time=1995.315..2925.570 rows=508844 loops=1)
"  Output: f.flight_id, f.flight_no, f.scheduled_departure, f.scheduled_arrival,
f.departure_airport, f.arrival_airport, f.status, f.aircraft_code, f.actual_departure,
f.actual_arrival, f.update_ts"
  Hash Cond: (f.flight_id = booking_leg.flight_id)
  ->  Seq Scan on postgres_air.flight f  (actual time=0.029..29.349 rows=683178 loops=1)
"        Output: f.flight_id, f.flight_no, f.scheduled_departure, f.scheduled_arrival,
f.departure_airport, f.arrival_airport, f.status, f.aircraft_code, f.actual_departure,
f.actual_arrival, f.update_ts"
  ->  Hash (actual time=1992.549..1992.549 rows=17893566 loops=1)
        Output: booking_leg.flight_id
        Buckets: 262144  Batches: 128  Memory Usage: 7172kB
        ->  Seq Scan on postgres_air.booking_leg (actual time=0.036..897.228
rows=17893566 loops=1)
              Output: booking_leg.flight_id
Planning Time: 1.305 ms
Execution Time: 2935.095 ms
```

Index

```
create index Booking_leg_flight_id
  on Booking_leg(flight_id);
```

Plan d'exécution après index

```
Nested Loop Semi Join (actual time=0.080..715.770 rows=508844 loops=1)
"  Output: f.flight_id, f.flight_no, f.scheduled_departure, f.scheduled_arrival,
f.departure_airport, f.arrival_airport, f.status, f.aircraft_code, f.actual_departure,
f.actual_arrival, f.update_ts"
  ->  Seq Scan on postgres_air.flight f (actual time=0.027..37.928 rows=683178 loops=1)
"        Output: f.flight_id, f.flight_no, f.scheduled_departure, f.scheduled_arrival,
f.departure_airport, f.arrival_airport, f.status, f.aircraft_code, f.actual_departure,
f.actual_arrival, f.update_ts"
  ->  Index Only Scan using booking_leg_flight_id on postgres_air.booking_leg
     (actual time=0.001..0.001 rows=1 loops=683178)
        Output: booking_leg.flight_id
        Index Cond: (booking_leg.flight_id = f.flight_id)
        Heap Fetches: 0
Planning Time: 4.371 ms
Execution Time: 725.913 ms
```

3.7. Agrégation

Une agrégation est mise en œuvre par :

- un parcours linéaire si l'attribut n'est pas indexé.
- une recherche binaire est effectuée si l'attribut est indexé par un B-Tree.

Dans le cas où un groupement est défini (GROUP BY), les tuplets sont groupés puis la fonction d'agrégation est calculée. Si un index de classification est défini sur l'attribut, seulement, le calcul sera effectué.

3.7.1. Exemple — agrégation sur un attribut non-clé

Requête

```
select max(price)
from booking;
```

Plan d'exécution sans index

```
Aggregate (actual time=647.915..647.915 rows=1 loops=1)
  Output: max(price)
  ->  Seq Scan on postgres_air.booking (actual time=0.175..281.690 rows=5643216 loops=1)
"        Output: booking_id, booking_ref, booking_name, account_id, email, phone,
update_ts, price"
Planning Time: 0.292 ms
Execution Time: 647.978 ms
```

Index

```
create index Booking_price
 on booking(price);
```

Plan d'exécution avec index

```
Result (actual time=0.135..0.137 rows=1 loops=1)
    ->  Limit (actual time=0.127..0.128 rows=1 loops=1)
          Output: booking.price
          ->  Index Only Scan Backward using booking_price on postgres_air.booking
              (actual time=0.123..0.124 rows=1 loops=1)
                Output: booking.price
                Index Cond: (booking.price IS NOT NULL)
                Heap Fetches: 0
Planning Time: 0.848 ms
Execution Time: 0.178 ms
```

3.7.2. Exemple — agrégation avec groupement sur un attribut non-clé

Requête

```
select flight_id, count(booking_id)
from Booking_leg
group by flight_id;
```

Plan d'exécution avant index

```
HashAggregate  (actual time=3049.099..4695.085 rows=508844 loops=1)
"  Output: flight_id, count(booking_id)"
  Group Key: booking_leg.flight_id
  Planned Partitions: 4  Batches: 21  Memory Usage: 8249kB  Disk Usage: 407400kB
  ->  Seq Scan on postgres_air.booking_leg  (actual time=0.037..676.445 rows=17893566
loops=1)
"        Output: booking_leg_id, booking_id, flight_id, leg_num, is_returning,
update_ts"
Planning Time: 0.203 ms
```

```
Execution Time: 4717.243 ms
```

Index

```
create index Booking_leg_flight_id
  on Booking_leg(flight_id);
```

Plan d'exécution après index

```
GroupAggregate (actual time=0.210..3575.061 rows=508844 loops=1)
"  Output: flight_id, count(booking_id)"
  Group Key: booking_leg.flight_id
  ->  Index Scan using booking_leg_flight_id on postgres_air.booking_leg
      (actual time=0.103..2891.493 rows=17893566 loops=1)
"     Output: booking_leg_id, booking_id, flight_id, leg_num, is_returning, update_ts"
Planning Time: 0.960 ms
Execution Time: 3585.077 ms
```

Conclusion

La performance des requêtes dépend de plusieurs facteurs. Une surveillance et une maintenance régulière sont requises.

Ne pas créer des index pour rien. La mise à jour des index peut être couteuse.

Dans PostgreSQL, pg_stat_all_indexes, cela permet d'extraire des informations sur l'utilisation des index.

Références

[Elmasri2016]
Ramez ELMASRI et Shamkant B. NAVATHE;
Fundamentals of database systems;
7th Edition, Pearson, Hoboken (NJ, US), 2016;
ISBN 978-0-13-397077-7.

[Dombrovskaya2021a]
Henrietta DOMBROVSKAYA, Boris NOVIKOV et Anna BAILLIEKOVA;
PostgreSQL Query Optimization;
Apress, Berkeley (CA, US), 2021;
ISBN 978-1-4844-6884-1.

[Date2004a]
Chris J. DATE;
Traduit par Martine CHALMOND, et Jean-Marie THOMAS;
Introduction aux bases de données;
Paris, Vuibert, 2004.
ISBN 978-2711786640.

Jeu de données

https://drive.google.com/drive/folders/13F7M80Kf_somnjb-mTYAnh1hW1Y_g4kJ https://github.com/hettie-d/postgres_air?tab=readme-ov-file

PostgreSQL Statistics Used by the Planner

https://www.postgresql.org/docs/current/planner-stats.html

PostgreSQL Conseils sur les performances

`[fr]` https://docs.postgresql.fr/16/performance-tips.html (2024-09-17)

Query converter - SQL to Relational Algebra (2024-09-23)
https://www.grammaticalframework.org/qconv/qconv-a.html

PostgreSQL execution plan visualizer (2024-09-22)
https://explain.dalibo.com

A. Annexe

Algorithme boucles imbriquées

L'algorithme de boucles imbriquées est l'algorithme de base pour toute opération relationnelle qui combine des tuplets de deux ou plusieurs relations en utilisant des conditions aléatoires.

Le cout de cet algorithme est proportionnel au produit des tailles des relations d'entrées : cost(R1,...Rn) = #(R1) * ... * #(Rn).

- Avec un balayage complet, la première boucle itère sur les blocs des des relations d'entrées et la deuxième itère sur les enregistrements des blocs.
- Avec un balayage par un index, le nombre d'itérations de la deuxième boucle est réduit et peut même ne pas exister, si une relation possède un index clé sur les attributs utilisés pour la jointure (naturelle).

Algorithme d'adressage dispersé

L'algorithme d'adressage dispersé consiste à construire une table de hachage à partir des tuplets de l'une ou des deux relations jointes, puis les seuls tuplets ayant le même code de hachage sont comparés pour l'égalité.

Tout d'abord, une table de hachage est construite en utilisant le contenu d'une relation, idéalement celle qui est le plus petit nombre de tuplets après l'application des restrictions locaux. Cette relation est appelée «construction» de la jointure. Les entrées de la table de hachage sont des correspondances entre la valeur des attributs de jointure et les autres attributs du tuplet (ceux qui sont nécessaires).

Une fois la table de hachage construite, il convient de parcourir l'autre relation (la relation de recherche). Pour chaque tuplet de la relation de recherche, trouvez les lignes pertinentes de la relation de construction en consultant la table de hachage.

Cet algorithme est simple, mais il exige que la relation de jointure la plus petite tienne dans la mémoire, ce qui n'est parfois pas le cas.

Le cout de cet algorithme est proportionnel à la somme des tailles des relations d'entrées. Soit JA l'ensemble d'attributs de jointure : cost(R1,R2) = (R1) * log((R2)) + (R2) * log((R1))

Cet algorithme est plus efficace pour des relations ayant un grand nombre de valeurs différentes pour l'ensemble d'attributs de jointure.

Définitions

Sources consultées de juin 2023 à juillet 2024

* Antidote : Antidote 11 v4.2 (2023), voir https://www.antidote.info
* Le Larousse : https://www.larousse.fr/dictionnaires/francais
* Le Robert : https://dictionnaire.lerobert.com
* Wikipédia : https://fr.wikipedia.org/wiki

Sigles

SQL (*Structure Query Language*)

Langage de programmation axiomatique fondé sur un modèle inspiré de la théorie relationnelle proposée par E. F. Codd.
[Normes applicables : ISO 9075:2016, ISO 9075:2023]

Produit le 2025-10-01 10:24:46 UTC

Université de Sherbrooke