

## TP #02 : Optimisation des requêtes avec les index

Désolé d'avance pour les fautes d'orthographe on a pas vraiment eu le temps de relire 😊

### 1.1 Première analyse

	QUERY PLAN	
	text	🔒
1	Gather (cost=1000.00..278762.12 rows=427676 width=84) (actual time=42.220..3614.701 rows=440009 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on title_basics (cost=0.00..234994.52 rows=178198 width=84) (actual time=33.096..3487.638 rows=146670 loop...	
5	Filter: (start_year = 2020)	
6	Rows Removed by Filter: 3764950	
7	Planning Time: 2.021 ms	
8	JIT:	
9	Functions: 6	
10	Options: Inlining false, Optimization false, Expressions true, Deforming true	
11	Timing: Generation 2.985 ms (Deform 0.941 ms), Inlining 0.000 ms, Optimization 2.140 ms, Emission 24.500 ms, Total 29.625 ms	
12	Execution Time: 3659.309 ms	

### 1.2 Analyse du plan d'exécution

#### 1. Pourquoi PostgreSQL utilise-t-il un Parallel Sequential Scan ?

PostgreSQL utilise un "Parallel Sequential Scan" parce qu'il n'y a pas d'index sur la colonne "start\_year". Cela signifie que le moteur de base de données doit lire toutes les lignes de la table "title\_basics" pour appliquer le filtre "start\_year=2020".

#### 2. La parallélisation est-elle justifiée ici ? Pourquoi ?

Oui. Comme il n'y a pas d'index, PostgreSQL est obligé de lire toutes les lignes de la table. Utiliser plusieurs processus permet de répartir cette tâche coûteuse et de réduire le temps total d'exécution.

#### 3. Que signifie "Rows Removed by Filter" ?

"Rows Removed by Filter" indique le nombre de lignes qui ont été lues, mais qui ne respectaient pas la condition "start\_year = 2020", et ont donc été rejetées.

### 1.3 Création d'index

1  
2

```
CREATE INDEX idx_start_year ON title_basics(start_year);
```

Données Messages Notifications

CREATE INDEX

Requête exécutée avec succès en 11 à 594 msec.

### 1.4 Analyse après indexation

Critère	Avant index (1.1)	Après index (1.4)
Type de Scan	Parallel Sequential Scan	Bitmap Index Scan + Parallel Bitmap Heap Scan
Index utilisé	Aucun	idx_start_year
Lignes retournées	440 009	440 009

<b>Rows Removed by Filter</b>	3 764 950	671 153
<b>Temps d'exécution</b>	~3659 ms	~1638 ms
<b>Nombre de workers</b>	2	2
<b>Bloc mémoire analysés</b>	non précisé	12 063 exacts, 10 989 approximatifs
<b>Gain de performance</b>	—	Environ 2× plus rapide

## 1.5 Impact du nombre de colonnes

Cmd exécuté :

Requête Historique

```

1  EXPLAIN ANALYZE
2  SELECT tconst, primary_title, start_year
3  FROM title_basics
4  WHERE start_year = 2020;
5  |

```

### 1. Le temps d'exécution a-t-il changé ? Pourquoi ?

Oui, le temps d'exécution a fortement diminué. Il est passé d'environ 1638 ms à 710 ms.

Cela s'explique par le fait que la requête ne sélectionne plus toutes les colonnes mais uniquement celle que nous avons sélectionné

## 2. Le plan d'exécution est-il différent ?

Non, le plan reste le même.

La requête est plus rapide car la requête traite moins d'informations.

## 3. Pourquoi la sélection de moins de colonnes peut-elle améliorer les performances ?

Sélectionner moins de colonnes permet de :

- réduire la quantité de données extraites de la table ;
- accéder à moins de blocs mémoire ;
- transférer moins de données entre PostgreSQL et le client.

Tout cela rend la requête plus rapide, même si le nombre de lignes retournées est identique. Cette technique est une forme d'**optimisation simple mais efficace**, surtout sur des tables volumineuses.

## 1.6 Analyse de l'impact global

### 1. Quelle nouvelle stratégie PostgreSQL utilise-t-il maintenant ?

PostgreSQL utilise maintenant un Bitmap Index Scan suivi d'un Parallel Bitmap Heap Scan.

Il commence par repérer les lignes correspondantes dans l'index, puis va lire uniquement ces lignes dans la table, en parallèle.

C'est plus efficace que le scan complet utilisé au départ.

### 2. Le temps d'exécution s'est-il amélioré ? De combien ?

Oui, le temps est passé d'environ 3659 ms à 710 ms.

C'est une amélioration d'environ 2950 ms.

Cela montre que l'utilisation de l'index et la sélection de moins de colonnes ont bien optimisé la requête.

### 3. Que signifie "Bitmap Heap Scan" et "Bitmap Index Scan" ?

Le Bitmap Index Scan lit l'index pour trouver les lignes à récupérer.

Le Bitmap Heap Scan utilise ces informations pour aller lire uniquement les lignes nécessaires dans la table.

Cette méthode permet de gagner du temps en évitant de lire toute la table.

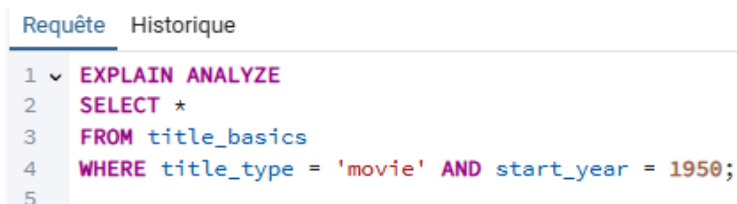
### 4. Pourquoi l'amélioration n'est-elle pas plus importante ?

L'amélioration reste limitée par plusieurs facteurs :

- Le nombre de lignes retournées est élevé (440 000), donc PostgreSQL doit quand même lire beaucoup de données.
- L'index ne couvre pas toutes les colonnes nécessaires, donc PostgreSQL doit lire dans la table (pas d'Index Only Scan possible).


## 2.1 Requête avec conditions multiples

cmd exécuté :



```
Requête  Historique
1  EXPLAIN ANALYZE
2  SELECT *
3  FROM title_basics
4  WHERE title_type = 'movie' AND start_year = 1950;
5
```

résultat :

	QUERY PLAN	
	text	
1	Bitmap Heap Scan on title_basics (cost=87.23..25993.49 rows=477 width=84) (actual time=12.304..1453.542 rows=2009 loo...	
2	Recheck Cond: (start_year = 1950)	
3	Filter: ((title_type)::text = 'movie'::text)	
4	Rows Removed by Filter: 6268	
5	Heap Blocks: exact=3275	
6	-> Bitmap Index Scan on idx_start_year (cost=0.00..87.11 rows=7823 width=0) (actual time=11.167..11.169 rows=8277 loop...	
7	Index Cond: (start_year = 1950)	
8	Planning Time: 1.184 ms	
9	Execution Time: 1454.180 ms	

### 1. Quelle stratégie est utilisée pour le filtre sur 'start\_year' ?

La stratégie utilisée est un Bitmap Index Scan. PostgreSQL utilise l'index sur la colonne 'start\_year' pour repérer les lignes où l'année est 1950. Cela lui permet de ne pas lire toute la table.

### 2. Comment est traité le filtre sur 'title\_type' ?

Le filtre 'title\_type = 'movie'' est appliqué ensuite, pendant le Bitmap Heap Scan. PostgreSQL lit les lignes trouvées avec l'index sur 'start\_year', puis filtre celles dont le type est bien 'movie'. Ce second filtre est donc fait en mémoire, sans index.

### 3. Combien de lignes passent le premier filtre, puis le second ?

L'index sur 'start\_year' renvoie 8277 lignes. Ensuite, PostgreSQL en rejette 6268 qui ne sont pas des films. Il reste donc 2009 lignes gardées au final.

### 4. Quelles sont les limitations de l'index actuel ?

L'index ne concerne que start\_year. Il ne permet pas d'optimiser le filtre sur 'title\_type'. PostgreSQL doit donc lire plus de lignes que nécessaire, puis filtrer après, ce qui ralentit la requête. Un index combinant les deux colonnes serait plus efficace.

## 2.3 Index composite

Requête

Historique

1

2

CREATE INDEX idx\_start\_year\_type ON title\_basics(start\_year, title\_type);

Données

Messages

Notifications

CREATE INDEX

Requête exécutée avec succès en 9 s 848 msec.

## 2.4 Analyse après index composite

Critère	Étape 2.1 (sans index composite)	Étape 2.4 (avec index composite)
Type de Scan	Bitmap Heap Scan	Bitmap Heap Scan
Index utilisé	idx_start_year	idx_start_year_type
Filtres dans l'index	start_year uniquement	start_year et title_type
Lignes trouvées via l'index	8277	2009

Lignes conservées après filtrage	2009	2009
Lignes rejetées après lecture	6268	0
Temps d'exécution	~1454 ms	~1.6 ms
Gain de performance	Aucun	Environ 1450 ms gagnées, 900× plus rapide

## 2.5 Impact du nombre de colonnes

### 1. Le temps d'exécution a-t-il changé ?

Oui, il est passé de 1.6 ms à 2.46 ms. La différence est très faible. Cela montre que réduire le nombre de colonnes dans ce cas n'a pas apporté de gain réel, car la requête était déjà très rapide grâce à l'index composite.

### 2. Pourquoi cette optimisation est-elle plus ou moins efficace que dans l'exercice 1 ?

Elle est moins efficace ici parce que l'index composite filtre déjà précisément les lignes. Dans l'exercice 1, on était parti d'un scan complet de la table, donc le fait de réduire les colonnes avait un impact plus net. Ici, on est déjà sur un plan optimisé.

### 3. Dans quel cas un "covering index" serait idéal ?

Un covering index serait utile si toutes les colonnes demandées dans la requête étaient déjà présentes dans l'index. PostgreSQL n'aurait alors pas besoin d'aller lire les données dans la table.

## 2.6 Analyse de l'amélioration

### 1. Quelle est la différence de temps d'exécution par rapport à l'étape 2.1 ?

Dans l'étape 2.1, la requête prenait environ 1454 millisecondes.

Après les optimisations, dans l'étape 2.4, elle prend environ 1.6 millisecondes.

Cela représente un gain de temps d'environ 1452 millisecondes. La requête est donc environ 900 fois plus rapide.

### 2. Comment l'index composite modifie-t-il la stratégie ?

L'index composite permet à PostgreSQL d'appliquer les deux filtres (start\_year et



title\_type) directement dans l'index. Cela évite de lire des lignes inutiles. Le filtre n'est plus appliqué après coup mais dès l'accès aux données. Cela rend le plan plus précis et plus rapide.

### 3. Pourquoi le nombre de blocs lus (Heap Blocks) a-t-il diminué ?

Avec un index simple, PostgreSQL trouvait trop de lignes et devait ensuite en rejeter une grande partie. Il lisait donc beaucoup de blocs dans la table pour ne garder que quelques résultats.

Avec l'index composite, seules les lignes utiles sont ciblées, donc PostgreSQL lit beaucoup moins de blocs.

### 4. Dans quels cas un index composite est-il particulièrement efficace ?

Un index composite est efficace quand une requête filtre sur plusieurs colonnes. Il est encore plus utile si ces colonnes sont souvent utilisées ensemble dans des conditions 'WHERE'.

Il est aussi très adapté quand ces filtres permettent de cibler une petite partie des lignes, comme ici.

## 3.1 Jointure avec filtres

cmd exécuté :

Requête Historique

```
1  EXPLAIN ANALYZE
2  SELECT b.primary_title, r.average_rating
3  FROM title_basics b
4  JOIN title_ratings r ON b.tconst = r.tconst
5  WHERE b.start_year = 1994 AND r.average_rating > 8.5;
```

## 3.2 Analyse du plan de jointure

### 1. Quel algorithme de jointure est utilisé ?

PostgreSQL peut utiliser différents types de jointures selon les index disponibles et la taille des tables. Le plus souvent dans ce type de requête, il utilise un Hash Join. Cela signifie qu'il charge en mémoire l'une des deux tables (souvent la plus petite), puis il balaie l'autre pour chercher les correspondances. C'est une stratégie efficace quand il n'y a pas d'index sur la colonne de jointure.

### 2. Comment l'index sur start\_year est-il utilisé ?

Si un index sur start\_year a été créé auparavant, PostgreSQL peut l'utiliser pour filtrer rapidement les titres de l'année 1994. Il peut le faire via un Index Scan ou un

Bitmap Index Scan sur la table 'title\_basics'. Cela permet de ne lire que les lignes utiles, sans parcourir toute la table.

### 3. Comment est traitée la condition sur 'average\_rating'?

Si aucun index n'existe encore sur 'average\_rating', PostgreSQL lit d'abord tous les ratings liés aux titres trouvés, puis applique le filtre `average_rating > 8.5` après la jointure. Cela signifie qu'il examine plus de données qu'il ne le faudrait. Si un index existe, il peut filtrer plus tôt et limiter les lignes à traiter dès le départ.

## 3.3 Analyse du plan de jointure

Requête

Historique

1

2

CREATE INDEX idx\_average\_rating ON title\_ratings(average\_rating);

|

Données

Messages

Notifications

CREATE INDEX

Requête exécutée avec succès en 1 s 134 msec.

### 3.4 Analyse du plan de jointure

Résultat :

QUERY PLAN		
text		
1	Hash Join (cost=141812.22..155754.00 rows=966 width=28) (actual time=145.416..254.040 rows=735 loops=1)	
2	Hash Cond: ((r.tconst)::text = (b.tconst)::text)	
3	-> Bitmap Heap Scan on title_ratings r (cost=2843.75..16387.44 rows=151655 width=18) (actual time=10.424..72.918 rows=151388 loo...	
4	Recheck Cond: (average_rating > '8.5'::double precision)	
5	Heap Blocks: exact=11135	
6	-> Bitmap Index Scan on idx_average_rating (cost=0.00..2805.84 rows=151655 width=0) (actual time=8.562..8.563 rows=151388 loo...	
7	Index Cond: (average_rating > '8.5'::double precision)	
8	-> Hash (cost=138034.57..138034.57 rows=74712 width=30) (actual time=133.756..133.758 rows=68964 loops=1)	
9	Buckets: 131072 Batches: 1 Memory Usage: 5201kB	
10	-> Bitmap Heap Scan on title_basics b (cost=835.45..138034.57 rows=74712 width=30) (actual time=10.887..107.884 rows=68964 l...	
11	Recheck Cond: (start_year = 1994)	
12	Heap Blocks: exact=20823	
13	-> Bitmap Index Scan on idx_start_year (cost=0.00..816.77 rows=74712 width=0) (actual time=2.526..2.526 rows=68964 loops=1)	
14	Index Cond: (start_year = 1994)	
15	Planning Time: 0.434 ms	
16	JIT:	
17	Functions: 15	
18	Options: Inlining false, Optimization false, Expressions true, Deforming true	
19	Timing: Generation 0.637 ms (Deform 0.367 ms), Inlining 0.000 ms, Optimization 0.353 ms, Emission 5.072 ms, Total 6.063 ms	
20	Execution Time: 254.896 ms	

Critère	Avant index (ex. 3.1)	Après index (ex. 3.4)
Type de jointure	Parallel Hash Join	Hash Join
Parallélisme	2 workers	Non
Filtrage sur average_rating	Fait après lecture complète	Fait via index
Type de scan sur title_ratings	Parallel Seq Scan	Bitmap Index Scan
Lignes rejetées par filtre	476 666	Aucune (filtre appliqué avant)

Temps d'exécution	~2267 ms	~254 ms
Gain de performance	Aucun	~2013 ms gagnés, 9× plus rapide

### 1. L'algorithme de jointure a-t-il changé ?

PostgreSQL utilise un hash join. Cela signifie qu'il construit une table en mémoire avec les tconst d'une des deux tables, puis il cherche les correspondances dans l'autre. Ce type de jointure est adapté aux cas où il y a beaucoup de lignes à traiter et où les colonnes jointes ne sont pas triées.

### 2. Comment l'index sur average\_rating est-il utilisé ?

L'index est bien utilisé. PostgreSQL commence par un bitmap index scan sur la colonne average\_rating pour repérer les lignes avec une note supérieure à 8.5. Il enchaîne avec un bitmap heap scan pour lire uniquement les lignes correspondantes dans la table. Cela permet de limiter les lectures inutiles.

### 3. Le temps d'exécution s'est-il amélioré ? Pourquoi ?

Oui, le temps d'exécution est passé à environ 254 millisecondes. C'est une bonne amélioration, car les deux filtres sont maintenant pris en compte dès le départ grâce aux index. Le moteur lit moins de blocs et traite moins de données.

### 4. Pourquoi PostgreSQL abandonne-t-il le parallélisme ?

PostgreSQL n'utilise pas le parallélisme ici, probablement parce qu'il estime que ce n'est pas nécessaire. Grâce aux index, le volume de données à lire est suffisamment réduit, donc l'ajout de workers n'apporterait pas de gain significatif. Le moteur choisit la stratégie la plus simple et la plus rapide dans ce cas.

## 4.1 Requête complexe

cmd exécuté :

	Requête	Historique
1	EXPLAIN ANALYZE	
2	SELECT start_year, COUNT(*) AS nb_films	
3	FROM title_basics	
4	WHERE title_type = 'movie'	
5	GROUP BY start_year;	

resultat :

	QUERY PLAN	
	text	🔒
1	Finalize GroupAggregate (cost=1000.46..134255.60 rows=132 width=12) (actual time=342.162..353.947 rows=139 loops=1)	
2	Group Key: start_year	
3	-> Gather Merge (cost=1000.46..134252.96 rows=264 width=12) (actual time=342.107..353.853 rows=242 loops=1)	
4	Workers Planned: 2	
5	Workers Launched: 2	
6	-> Partial GroupAggregate (cost=0.43..133222.47 rows=132 width=12) (actual time=6.366..192.832 rows=81 loops=3)	
7	Group Key: start_year	
8	-> Parallel Index Only Scan using idx_start_year_type on title_basics (cost=0.43..131731.47 rows=297935 width=4) (actual time=1.458..175.185 rows=23944...	
9	Index Cond: (title_type = 'movie')::text)	
10	Heap Fetches: 17	
11	Planning Time: 1.050 ms	
12	JIT:	
13	Functions: 15	
14	Options: Inlining false, Optimization false, Expressions true, Deforming true	
15	Timing: Generation 3.704 ms (Deform 0.000 ms), Inlining 0.000 ms, Optimization 1.156 ms, Emission 12.635 ms, Total 17.495 ms	
16	Execution Time: 356.065 ms	

## 4.2 Analyse du plan complexe