NEXA Digital School

19 JANVIER 2026

Performance des bases de données

MASTÈRE DATA & IA
CHEF DE PROJET

FORMATEUR : DIALLO ALIMOU

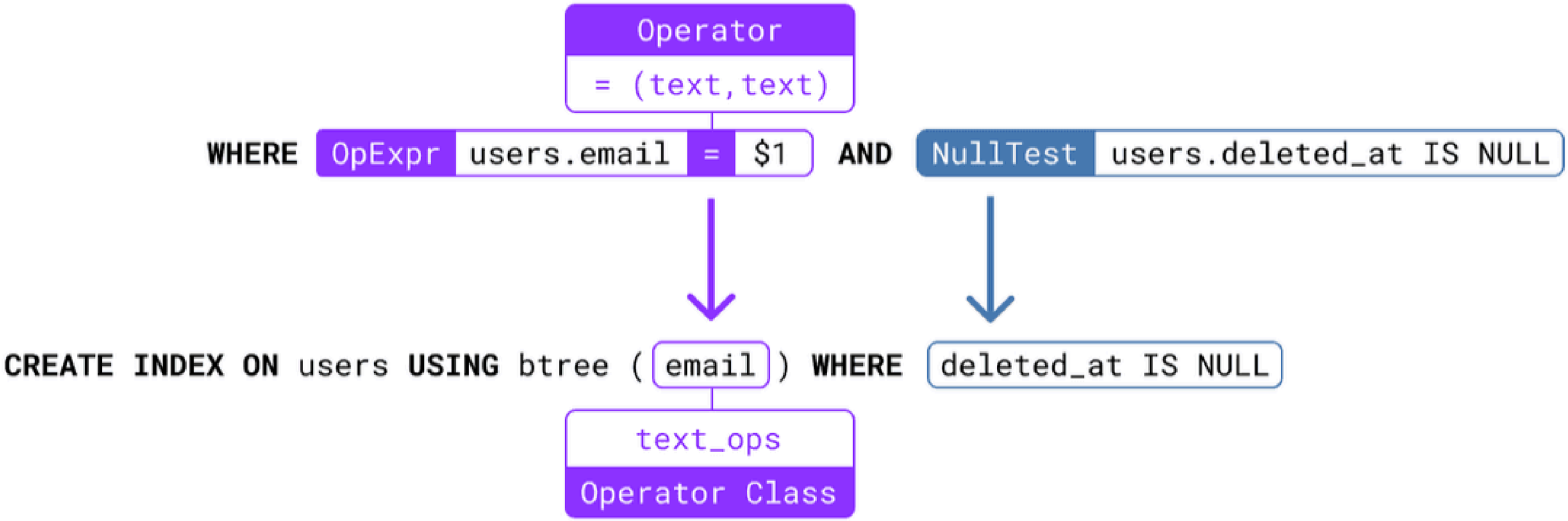
PostgreSQL : Types, Plans d’exécution, Index

C1.10 : Optimiser les performances d’une base de données en choisissant pour chaque variable le type et le nombre d’octets adaptés afin de rendre la donnée accessible rapidement et efficacement.

Savoir :

- Lire un plan d’exécution
- Mesurer objectivement une performance
- Choisir un type, un index, une structure
- Justifier techniquement leurs choix
- Mettre en place un monitoring

PostgreSQL : Types, Plans d'exécution, Index



Enjeux de performance

D’où viennent les problèmes de performance ?

Les performances peuvent se dégrader pour plusieurs raisons :

1. Problèmes de conception

- Mauvais schéma de base de données
- Mauvais choix de types de données
- Requêtes SQL mal écrites
- Mauvaise stratégie d’indexation

Une mauvaise décision au départ peut coûter très cher plus tard.

Enjeux de performance

D’où viennent les problèmes de performance ?

Les performances peuvent se dégrader pour plusieurs raisons :

2. Évolution du système

- Augmentation du volume de données
- Plus d’utilisateurs concurrents
- Nouveaux usages / nouveaux patterns d’accès
- Accumulation de modifications et de correctifs

Même un système bien conçu peut devenir lent avec le temps.

Enjeux de performance

D’où viennent les problèmes de performance ?

Les performances peuvent se dégrader pour plusieurs raisons :

3. Problèmes d’exploitation

- Manque de maintenance (VACUUM, ANALYZE, réindexation…)
- Paramètres système mal ajustés
- Manque de ressources (CPU, RAM, disque)

Comment détecte-t-on les problèmes de performance ?

Trois signaux d’alerte principaux :

1. **Ralentissement des applications**
   - Pages qui chargent lentement
   - Requêtes ou batchs qui prennent plus de temps qu’avant
   - Souvent détecté par :
     - Les utilisateurs (plaintes)
     - Les équipes métiers
     - Ou via des outils de monitoring

Comment détecte-t-on les problèmes de performance ?

2. Pannes ou instabilités

- Manque d’espace disque
- Surcharge mémoire / CPU
- Blocages, timeouts, crashs

3. Fuite en avant matérielle

Ajout fréquent de RAM, de CPU, de serveurs

Souvent pour compenser un problème logiciel non résolu

Optimiser le SQL et le schéma coûte souvent beaucoup moins cher que changer le hardware.

Choisir le bon type de données

Le type de données influence directement :

- La taille sur disque
- L’usage mémoire (cache)
- La vitesse des index
- La vitesse des jointures et des tris

Des types trop gros = moins de lignes par page = plus d’I/O = plus lent.

Mauvais	Bon	Pourquoi
BIGINT	INT	2x plus petit
UUID	SERIAL	Index plus compact
TEXT	ENUM	Comparaison rapide
VARCHAR(255)	VARCHAR(n)	Moins de gaspillage

Choisir le bon type de données

PostgreSQL™ offre un large choix de types de données disponibles nativement. Les utilisateurs peuvent ajouter de nouveaux types à PostgreSQL™ en utilisant la commande **CREATE TYPE**

Types numériques : SMALLINT, INT, BIGINT, NUMERIC, REAL, DOUBLE
Types texte : TEXT, VARCHAR(n), CHAR(n)
Types date/temps : DATE, TIMESTAMP, TIMESTAMPTZ, INTERVAL
Types booléens : BOOLEAN
Types spéciaux : UUID, JSON, JSONB, ARRAY, ENUM

CREATE TYPE

Exemples :

- ENUM métier (statut de commande, type d'utilisateur…)

```
CREATE TYPE order_status AS ENUM (
  'pending',
  'paid',
  'shipped',
  'cancelled'
);
```

Cache hit ratio : comment ça marche ?

Deux types de lectures dans PostgreSQL

1. Blocs en mémoire — **blks_hit**

Nombre de blocs trouvés directement dans le cache PostgreSQL **(shared_buffers**)

➡ **Rapide**

2. Blocs lus sur disque — **blks_read**

Nombre de blocs absents du cache et lus depuis le disque (ou cache OS)

➡ **Lent**

Cache Hit Ratio (BCHR)

Mesure la proportion de lectures servies depuis la mémoire

BCHR = blks_hit / (blks_hit + blks_read) * 100

- 99% → Excellent
- 95–99% → Correct
- < 90% → Problème sérieux de performance

Comprendre le ratio de succès de cache de PostgreSQL

Un rapport de frappe de cache élevé indique généralement une utilisation efficace de la mémoire dans PostgreSQL, où la plupart des pages de données requises sont lues à partir du cache, minimisant ainsi les lectures coûteuses du disque.

PostgreSQL garde en mémoire (RAM) :

- Les pages de tables
- Les pages d’index

Ces pages sont stockées dans :

- Le cache du système d’exploitation
- Et le cache interne de PostgreSQL **(shared_buffers**)

Analyser les requêtes

Pourquoi analyser les requêtes ?

Une base de données lente est presque toujours causée par quelques requêtes mal optimisées.

Avant toute optimisation, il est indispensable de comprendre comment PostgreSQL exécute réellement une requête.

Le planificateur choisit un plan d’exécution en fonction :

- des statistiques
- des index
- et des coûts estimés.

Optimiser sans mesurer mène presque toujours à de mauvaises décisions.

```
SELECT * FROM orders WHERE user_id = 42;
```

Simuler l’impact des types de données

Montrer que :

❗ Le choix du type impacte :

- Taille disque
- Cache hit ratio
- Vitesse de scan
- Vitesse d’index
- Vitesse de tri

Exemple concret

On crée deux tables identiques mais typées différemment :

```
CREATE TABLE orders_text (
    id TEXT,
    user_id TEXT,
    status TEXT,
    total_amount TEXT,
    created_at TEXT
);
```

```
CREATE TABLE orders_typed (
    id UUID,
    user_id UUID,
    status SMALLINT,
    total_amount INT,
    created_at TIMESTAMP
);
```

On va comprendre :

- Que le type de donnée change :
  - la taille disque
  - la mémoire utilisée
  - la structure interne

voir le fichier Simuler l'impact des types de données pour la creation des tables

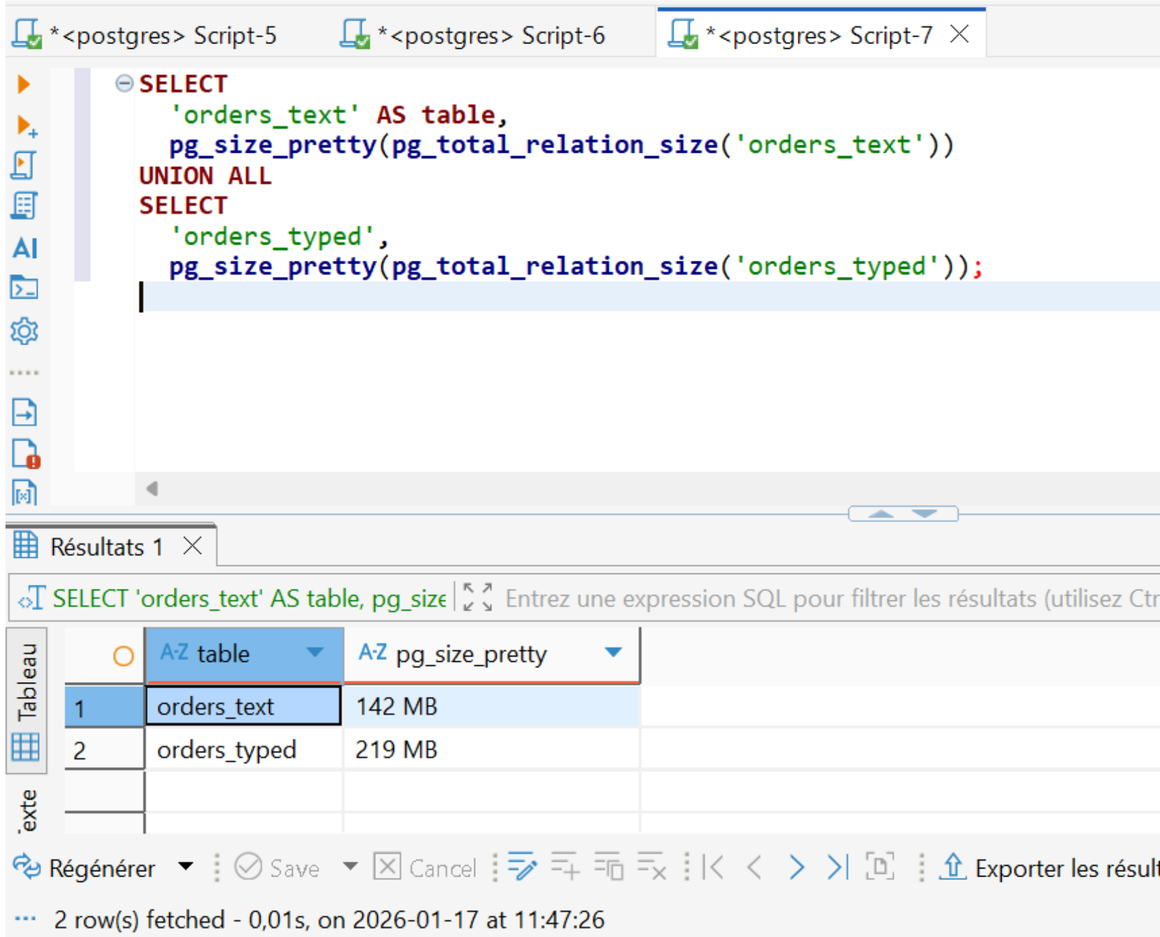
Exemple concret

Vérification du volume

```
SELECT COUNT(*) FROM orders_text;
SELECT COUNT(*) FROM orders_typed;
```

orders_text = 1 000 000 lignes
orders_typed = 3 000 000 lignes

Comparer la taille disque



On ne juge JAMAIS un schéma de données uniquement sur la taille disque.

Comparaison Full Table Scan (TEXT vs TYPÉ)

Requête

```
EXPLAIN ANALYZE
SELECT * FROM orders_text WHERE status = 'paid';


EXPLAIN ANALYZE
SELECT * FROM orders_typed WHERE status = 2;
```

Résultats observés

Comparaison Full Table Scan (TEXT vs TYPÉ)

Requête

```
Seq Scan on orders_text
(cost=0.00..30645.00 rows=249233 width=113)
(actual time=0.624..131.204 rows=250623 loops=1)
```

Seq Scan on orders_text

Seq Scan = Sequential Scan = Full Table Scan

Aucune utilisation d'index

PostgreSQL lit chaque ligne et teste :

cost=0.00..30645.00

0.00 = coût pour commencer

30645.00 = coût estimé total

Sert uniquement à comparer les plans entre eux

Plus le nombre est grand = plus PostgreSQL pense que c'est cher.

Comparaison Full Table Scan (TEXT vs TYPÉ)

Requête

Seq Scan on orders_text
(cost=0.00..30645.00 rows=249233 width=113)
(actual time=0.624..131.204 rows=250623 loops=1)

rows=249233

Estimation du nombre de lignes retournées

rows=250623

Nombre RÉEL de lignes trouvée

width=113

Taille moyenne d'une ligne retournée

- Chaque ligne fait environ 113 octets
- Plus c'est grand → plus c'est lent à :
  - lire
  - copier

	Estimation	Réel
Rows	249 233	250 623

Estimation très correcte

loops=1

Le plan a été exécuté une seule fois

(Parfois dans des jointures on a loops=1000 etc)

Tableau comparatif

Critère	orders_text (TEXT)	orders_typed (TYPÉ)
Type de scan	Seq Scan	Seq Scan
Temps total	**131 ms**	**322 ms**
Lignes retournées	250 623	749 627
Taille moyenne ligne (`width`)	**113 bytes**	**48 bytes**
Lignes lues	1 000 000	1 000 000
Travail CPU	Comparaison de chaînes	Comparaison d'entiers
Sélectivité	~25%	~75%

Tableau comparatif

orders_typed est :

plus compacte par ligne

mais retourne 3x plus de lignes

PostgreSQL doit :

lire autant de lignes

mais traiter beaucoup plus de résultats

Le typage améliore la structure des données,

mais ne remplace PAS :

une bonne sélectivité

ni une bonne stratégie d’indexation

Une requête qui retourne trop de lignes reste lente,

même sur une table bien typée.

Vitesse d’index

Créer les index

```
CREATE INDEX idx_orders_text_status ON orders_text(status);
CREATE INDEX idx_orders_typed_status ON orders_typed(status);
```

Tester

```
EXPLAIN ANALYZE
SELECT * FROM orders_text WHERE status = 'paid';
EXPLAIN ANALYZE
SELECT * FROM orders_typed WHERE status = 2;
```

Vitesse d’index

Critère	orders_text	orders_typed
Type de scan	Bitmap Heap Scan	Bitmap Heap Scan
Temps total	**~98 ms**	**~160 ms**
Lignes retournées	250 623	749 627
Taille ligne (`width`)	113 octets	48 octets
Index utilisé	Oui	Oui
Travail total	Moyen	Très élevé

Pourquoi **orders_typed** est plus lent ?

Pas à cause du typage.

Mais parce que :

Il retourne 3x plus de lignes

PostgreSQL doit :

- lire plus de blocs
- reconstruire plus de tuples
- envoyer plus de données

EXPLAIN vs EXPLAIN ANALYZE

EXPLAIN affiche le plan théorique choisi par PostgreSQL(Collecter les statistiques)

EXPLAIN ANALYZE exécute réellement la requête et affiche :

Pourquoi analyser les requêtes ?

❝ Une requête lente n’est pas une fatalité. ❞

- PostgreSQL exécute un plan
- Il faut voir ce plan pour comprendre

Comprendre EXPLAIN ANALYZE

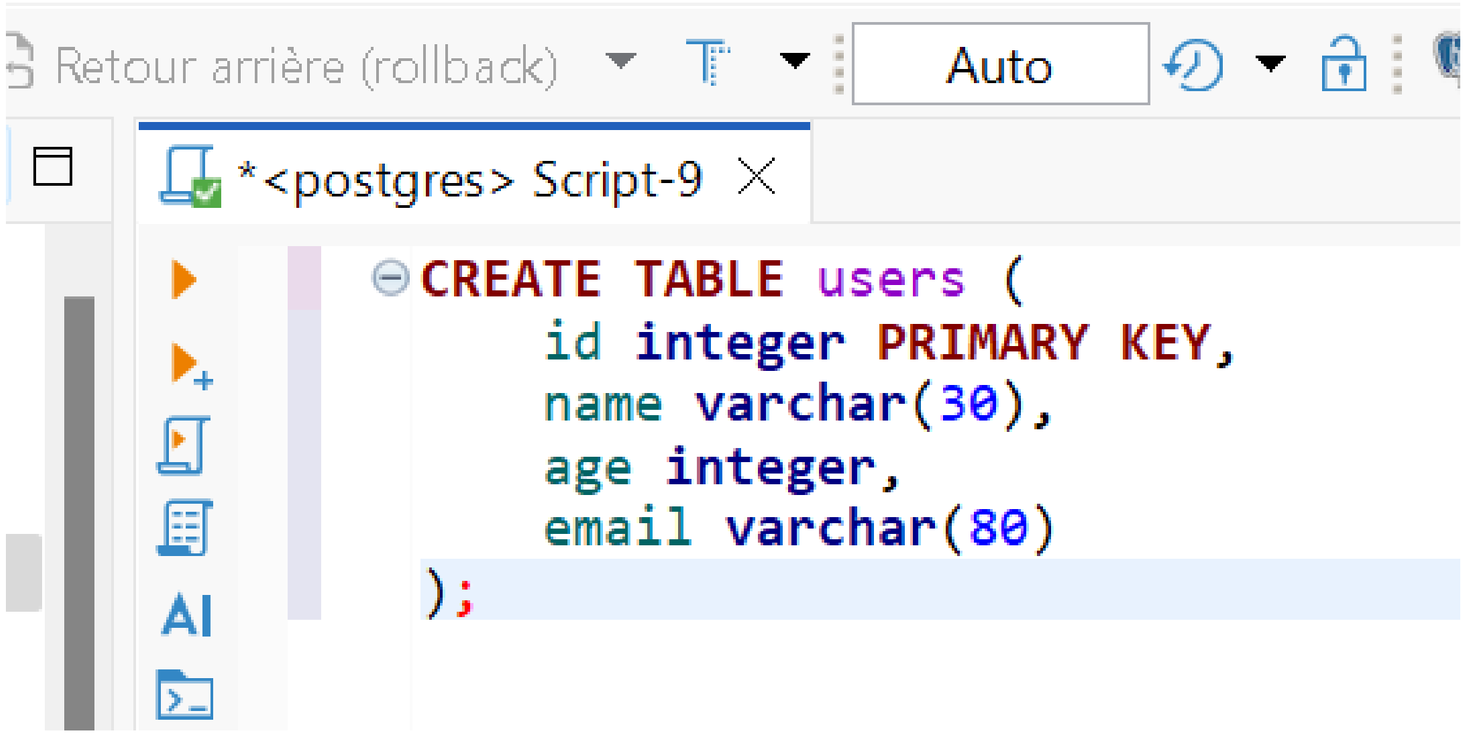
Pour commencer, il faut savoir qu'un moteur de base de données, que ce soit PostgreSQL ou un autre ne "lit" pas une requête comme un humain pourrait la lire, de **gauche à droite**. Il élabore plutôt un plan d'exécution : il créé un ensemble d'étape qui se veulent le plus optimisées possible pour aller chercher les données que vous lui demander de la façon la plus efficace possible.

Il y a un moyen simple de voir ce plan d'exécution, c'est d'utiliser la commande **EXPLAIN ANALYZE.** Elle nous permet en plus de comprendre de quelle façon le moteur à réellement exécuté notre ordre SQL (et pas seulement comment il pensait le faire). C'est un peu comme lire un journal de bord de l'exécution de la requête.

Comprendre EXPLAIN ANALYZE

Prenons un exemple simple :

Imaginons que nous avons une table users, constituée comme tel :



Et que nous souhaitons utiliser la requête suivante dans notre application :

```
SELECT * FROM users WHERE email = 'foo@example.com';
```

Comprendre EXPLAIN ANALYZE

Sur cette table, nous n’avons pas d’index. Voici donc ce à quoi pourrait ressembler notre plan d’éxecution si nous utilisons la commande :

EXPLAIN ANALYZE
SELECT * FROM users WHERE email = 'foo@example.com';

```
1  EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'foo@example.com';
2
3  Seq Scan on users (cost=0.00..35.50 rows=1 width=100)
4  Filter: (email = 'foo@example.com')
5  Rows Removed by Filter: 999
6  Actual time=0.010..0.420 rows=1 loops=1
```

PostgreSQL a :

Lu 1000 lignes

Jeté 999 lignes

Gardé 1 seule

C’est lent et inutile

Un index sur email éviterait ça

Comprendre EXPLAIN ANALYZE

Comment lire ce plan

Seq Scan / Index Scan : Le type de parcours utilisé (séquentiel ou via un index)

cost=… : Estimation du “coût” total de l’opération, selon PostgreSQL. Le coût n’est pas exprimé dans une unité particulière, c’est juste un indicatif. Plus il est élevé, plus l’opération est “coûteuse”, notre but étant d’éviter les coûts énormes pour que tout soit plus simple.

rows=… : Estimation du nombre de lignes que l’étape va retourner

actual time=… : Le temps réel que cette étape a pris (en millisecondes)

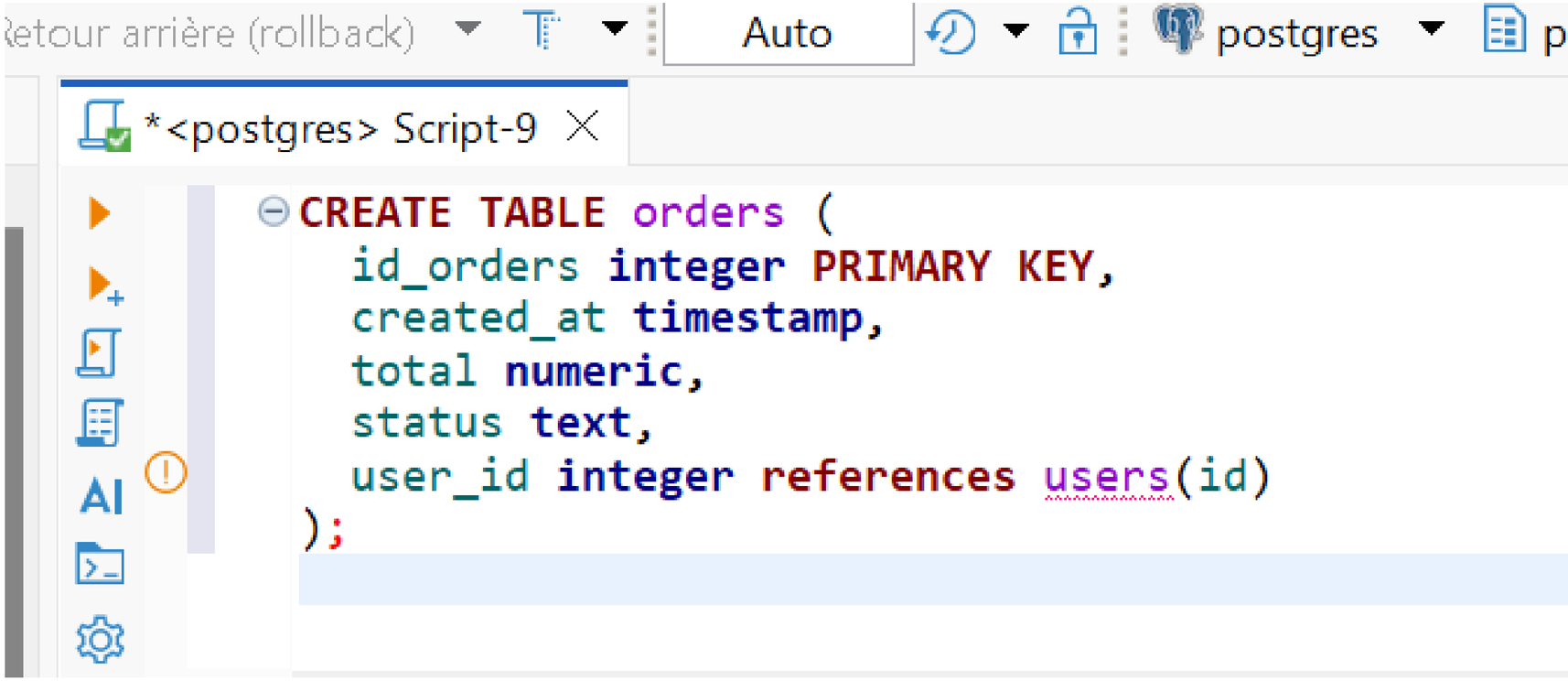
Rows Removed by Filter : Nombre de lignes lues mais éliminées par un filtre

loops=1 : Nombre de fois que cette étape a été exécutée (ex. dans une boucle)

Puis un exemple plus compliqué :

commandes passées par un user :

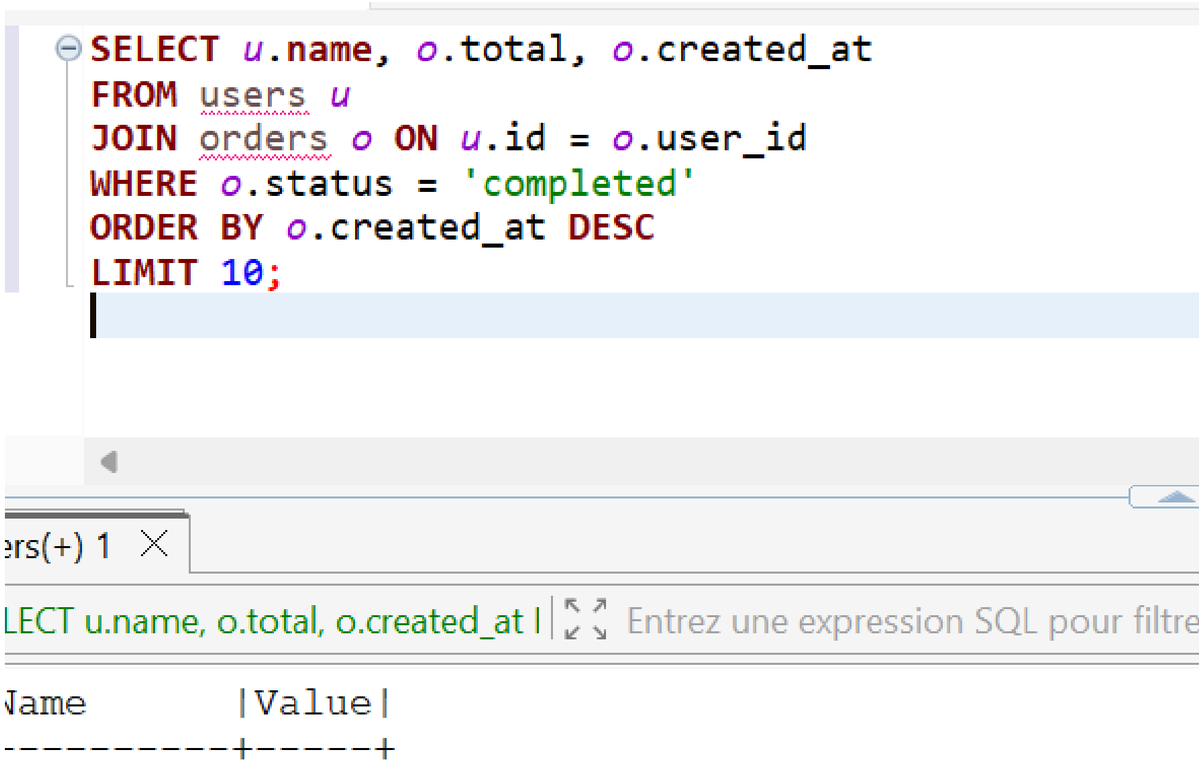
Schéma utilisé



Puis un exemple plus compliqué :

commandes passées par un user :

Nous pouvons alors imaginer la requête suivante :

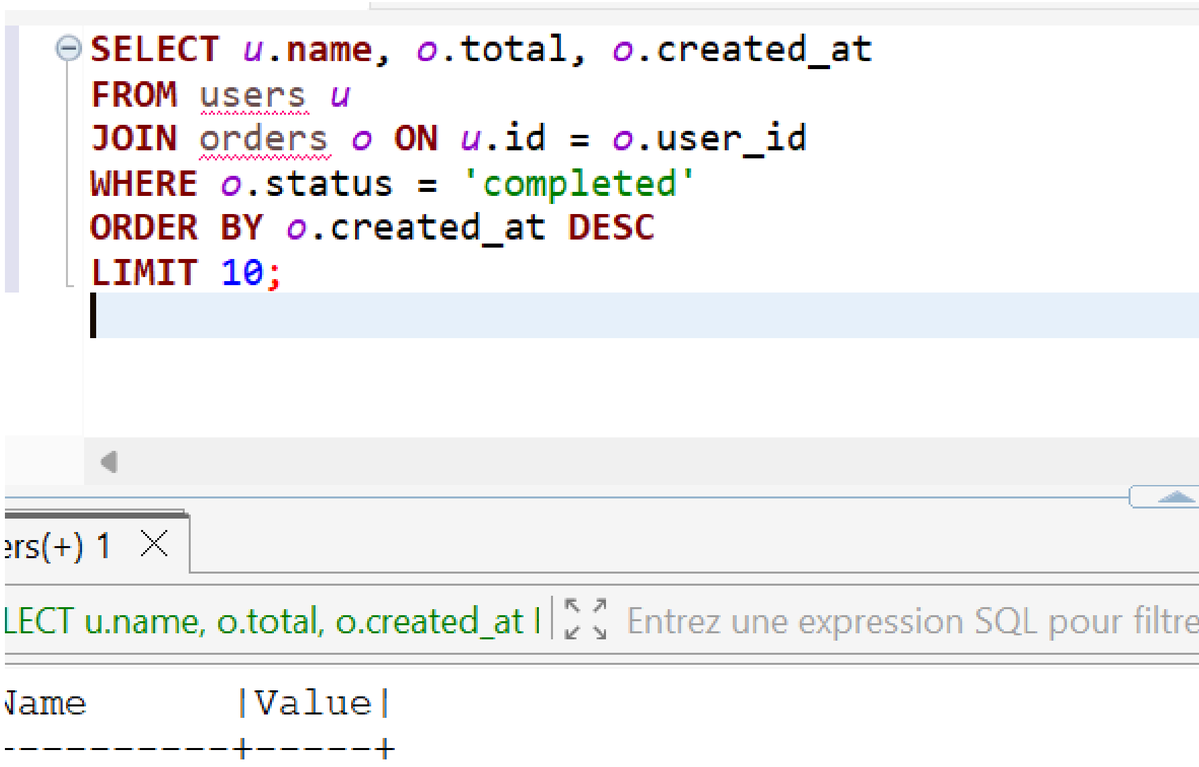


Qui remonte les dix premières commandes, avec le nom de l’acheteur, le total de la commande et sa date, pour les 10 premières commandes dans l’ordre des dates dont le statut est “completed”.

Puis un exemple plus compliqué :

commandes passées par un user :

Nous pouvons alors imaginer la requête suivante :



Qui remonte les dix premières commandes, avec le nom de l’acheteur, le total de la commande et sa date, pour les 10 premières commandes dans l’ordre des dates dont le statut est “completed”.

Puis un exemple plus compliqué :

Plan d’exécution obtenu avec EXPLAIN ANALYZE

```sql
EXPLAIN ANALYZE
SELECT u.name, o.total, o.created_at
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.status = 'completed'
ORDER BY o.created_at DESC
LIMIT 10;
```

Résultats 1

EXPLAIN ANALYZE SELECT u.name, o

Entrez une expression SQL pour filtrer les résultats (utilisez Ctrl+Espace)

Tableau

Texte

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Limit  (cost=33.22..33.23 rows=4 width=118) (actual time=0.026..0.027 rows=0.00 loops=1)
  ->  Sort  (cost=33.22..33.23 rows=4 width=118) (actual time=0.025..0.026 rows=0.00 loops=1)
        Sort Key: o.created_at DESC
        Sort Method: quicksort  Memory: 25kB
        ->  Hash Join  (cost=19.43..33.18 rows=4 width=118) (actual time=0.016..0.017 rows=0.00 loops
              Hash Cond: (u.id = o.user_id)
              ->  Seq Scan on users u  (cost=0.00..12.70 rows=270 width=82) (actual time=0.014..0.014
```

Régénérer | Save | Cancel | Exporter les résultats ... | 200 | 14

Puis un exemple plus compliqué :

Plan d’exécution obtenu avec EXPLAIN ANALYZE

```sql
EXPLAIN ANALYZE
SELECT u.name, o.total, o.created_at
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.status = 'completed'
ORDER BY o.created_at DESC
LIMIT 10;
```

Résultats 1

EXPLAIN ANALYZE SELECT u.name, o

Entrez une expression SQL pour filtrer les résultats (utilisez Ctrl+Espace)

Tableau

Texte

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Limit  (cost=33.22..33.23 rows=4 width=118) (actual time=0.026..0.027 rows=0.00 loops=1)
  ->  Sort  (cost=33.22..33.23 rows=4 width=118) (actual time=0.025..0.026 rows=0.00 loops=1)
        Sort Key: o.created_at DESC
        Sort Method: quicksort  Memory: 25kB
        ->  Hash Join  (cost=19.43..33.18 rows=4 width=118) (actual time=0.016..0.017 rows=0.00 loops
              Hash Cond: (u.id = o.user_id)
              ->  Seq Scan on users u  (cost=0.00..12.70 rows=270 width=82) (actual time=0.014..0.014
```

Régénérer | Save | Cancel | Exporter les résultats ... | 200 | 14

Options avancées de EXPLAIN ANALYZE

VERBOSE: plus de détails sur les colonnes et les expressions :

Affiche le nom exact des colonnes internes et les expressions utilisées dans chaque étape du plan

```
EXPLAIN (ANALYZE, VERBOSE)
SELECT name FROM users WHERE age > 30;
```

Très utile quand on travaille avec des fonctions, des agrégats ou des vues complexes.

Options avancées de EXPLAIN ANALYZE

L’option BUFFERS permet de voir d’où viennent réellement les données :

- Depuis la mémoire (cache PostgreSQL)
- Ou depuis le disque

EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM orders WHERE status = 'completed';

Idéal pour identifier si une requête est ralentie par des accès disques trop nombreux.

Champ	Signification
shared hit	Blocs lus depuis la **RAM** (cache PostgreSQL)
shared read	Blocs lus depuis le **DISQUE**
shared dirtied / written	Blocs modifiés / écrits (pour INSERT/UPDATE)

Options avancées de EXPLAIN ANALYZE

Option	Exemple SQL	Description	Utilité principale
WAL	`EXPLAIN (ANALYZE, WAL) INSERT INTO logs SELECT * FROM events;`	Affiche ce qui est écrit dans le Write-Ahead Log	Mesurer le **coût caché des écritures** (INSERT/UPDATE/DELETE)
COSTS	`EXPLAIN (ANALYZE, COSTS OFF) SELECT * FROM users;`	Affiche ou masque les coûts estimés	Se concentrer **uniquement sur les temps réels**
SETTINGS	`EXPLAIN (ANALYZE, SETTINGS) SELECT * FROM users;`	Affiche les paramètres PostgreSQL utilisés pour ce plan	Comprendre **pourquoi ce plan a été choisi**
SUMMARY	`EXPLAIN (ANALYZE, SUMMARY OFF) SELECT * FROM users;`	Affiche ou masque le résumé global	Nettoyer l'affichage pour se concentrer sur le plan
TIMING	`EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM users;`	Active/désactive la mesure fine du temps par étape	Réduire le surcoût de mesure / Benchmarks propres
FORMAT	`EXPLAIN (ANALYZE, FORMAT JSON) SELECT * FROM users;`	Change le format de sortie (TEXT, JSON, YAML, XML)	Exploitation par outils / visualisation automatique

P — Optimisation et analyse de performances PostgreSQL

1. Question ouverte à rédiger

1. Question ouverte à rédiger

Phase 1 — Mise en place de la plateforme
Concevoir le schéma relationnel
Créer les tables :
students
courses
enrollments
access_logs
Choisir et justifier les types de données
Générer automatiquement les volumes .
Vérifier :
la cohérence des données
la taille disque
le temps de chargement

Phase 2 — Diagnostic des performances
Écrire plusieurs requêtes métier réalistes :
filtres simples
jointures
tris + LIMIT
Analyser chaque requête avec :
EXPLAIN
EXPLAIN ANALYZE
EXPLAIN (ANALYZE, BUFFERS)