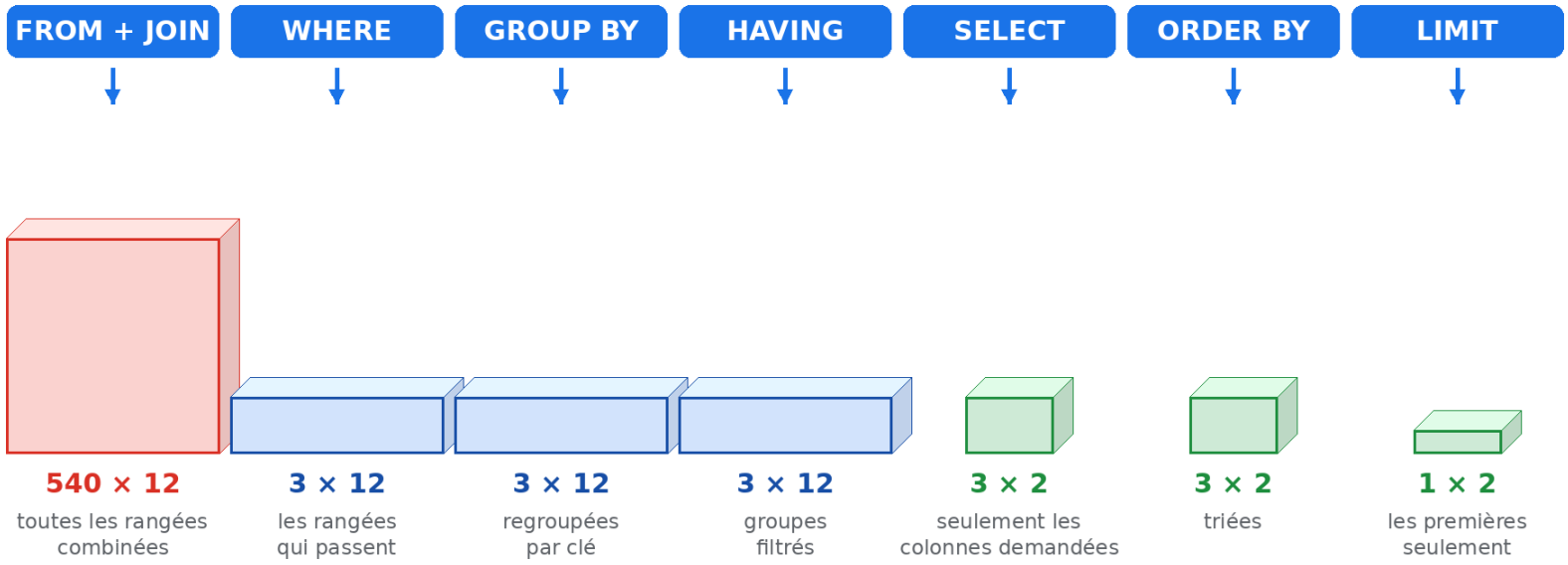
Les cubes de données transitoires

Ce qui se passe en mémoire pendant une requête SQL : l'ordre des conditions, leur place (ON ou WHERE) et l'ordre des jointures selon ce qu'on sait des données

Exploitation de données - série SQL - suite de « Combinatoire et optimisation »

Une requête se calcule par étapes, et chaque étape fabrique un cube



Chaque étape lit le cube précédent et en fabrique un nouveau en mémoire : rangées × colonnes.
Le moteur ne garde un cube que le temps de calculer le suivant. Plus le premier est gros, plus tout coûte.

(hauteur des cubes à l'échelle logarithmique)

Ordre logique : FROM et JOIN combinent, WHERE filtre les rangées, GROUP BY regroupe, HAVING filtre les groupes, SELECT garde les colonnes demandées, ORDER BY trie, LIMIT coupe. Chaque étape produit un cube transitoire en mémoire (rangées × colonnes), jeté dès que la suivante a fini. Le premier est souvent le plus gros : à garder petit.

Les données de l'exemple et la question posée

groupe
id
nom

3 rangées

Info 1, Info 2, Info 3

etudiant
id
prenom
sexe
groupe

90 rangées

30 par groupe, 12 filles
dont 3 dans Info 2

inscription
etudiant
cours
note

540 rangées

6 cours par étudiant

cours
id
titre

12 rangées

dont Exploitation de données,
suivi par les 90 étudiants

Clés étrangères : etudiant.groupe → groupe.id inscription.etudiant → etudiant.id inscription.cours → cours.id

La question : quelle est la moyenne des notes des filles du groupe Info 2 au cours Exploitation de données ?

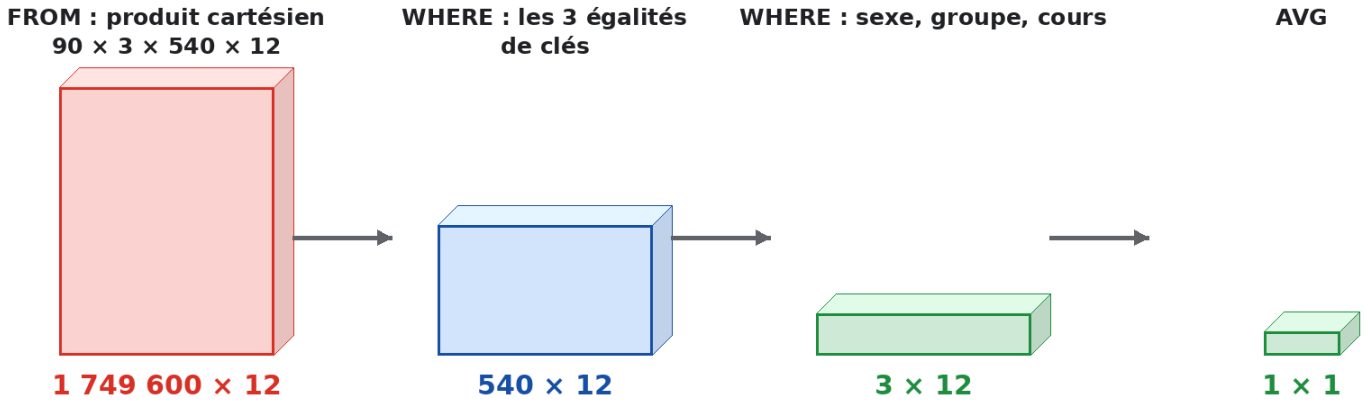
Réponse attendue : une seule valeur, calculée sur 3 rangées d'inscription.

```
SELECT AVG(inscription.note)
FROM etudiant JOIN groupe ON groupe.id = etudiant.groupe
              JOIN inscription ON inscription.etudiant = etudiant.id
              JOIN cours ON cours.id = inscription.cours
WHERE etudiant.sexe = 'F' AND groupe.nom = 'Info 2' AND cours.titre = 'Exploitation de données';
```

Quatre tables : groupe (3), etudiant (90, 12 filles dont 3 dans Info 2), inscription (540, 6 cours par étudiant), cours (12, dont Exploitation de données suivi par tous). La question : la moyenne des notes des filles d'Info 2 à ce cours. La réponse tient sur 3 rangées d'inscription ; tout le diaporama compare des façons d'y arriver.

Stratégie A : tout combiner, puis filtrer (le produit cartésien)

```sql
-- Stratégie A : l'ancienne écriture, toutes les tables dans le FROM, tout dans le WHERE
SELECT AVG(inscription.note)
FROM etudiant, groupe, inscription, cours
WHERE groupe.id = etudiant.groupe AND inscription.etudiant = etudiant.id AND cours.id = inscription.cours
  AND etudiant.sexe = 'F' AND groupe.nom = 'Info 2' AND cours.titre = 'Exploitation de données';
```



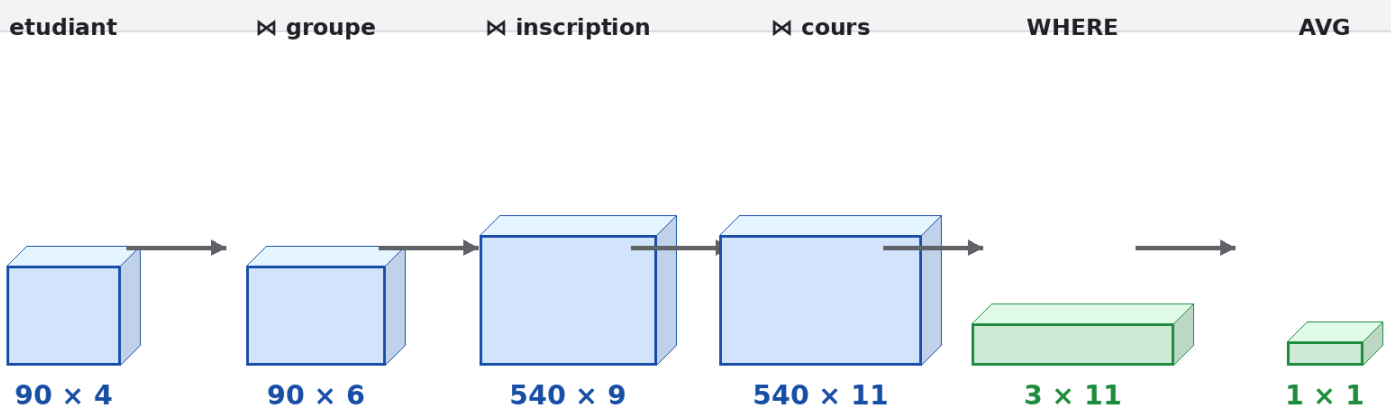
Ce que la requête dit mathématiquement : 1 749 600 combinaisons à vérifier avant de garder 3 rangées.

rangées × colonnes de chaque cube transitoire (hauteur à l'échelle logarithmique)

L'écriture ancienne, toutes les tables dans le FROM et tout dans le WHERE, décrit un produit cartésien : 90 × 3 × 540 × 12 = 1 749 600 combinaisons pour en garder 3. Les moteurs modernes l'évitent quand les conditions de clés sont là ; oubliez-en une, le bogue classique, et le cube existe pour vrai. C'est la combinatoire du diaporama précédent.

Stratégie B : joindre sur les clés, filtrer à la fin

```sql
-- Stratégie B : jointures sur les clés dans l'ordre d'écriture, les conditions de sélection à la fin
SELECT AVG(inscription.note)
FROM etudiant JOIN groupe ON groupe.id = etudiant.groupe
              JOIN inscription ON inscription.etudiant = etudiant.id
              JOIN cours ON cours.id = inscription.cours
WHERE etudiant.sexe = 'F' AND groupe.nom = 'Info 2' AND cours.titre = 'Exploitation de données';
```



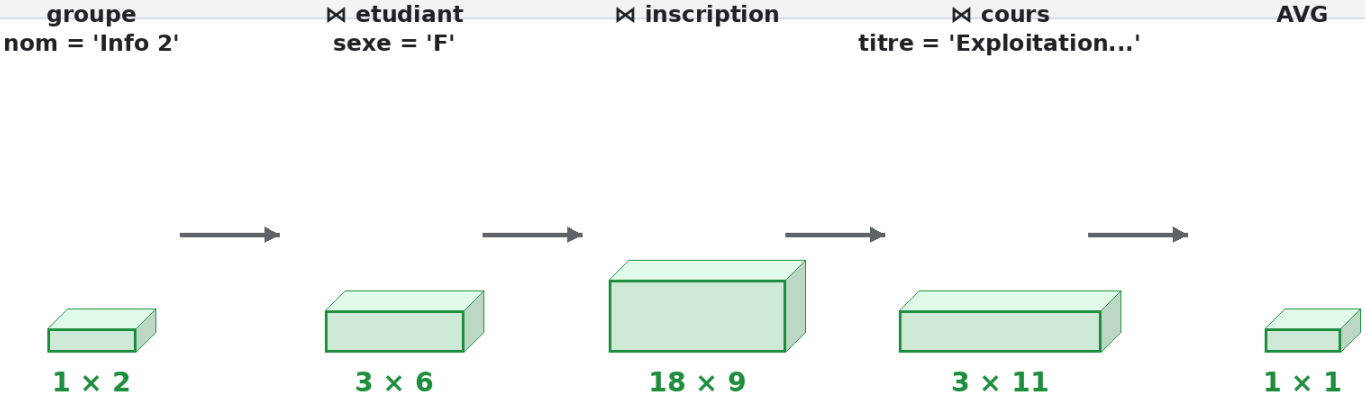
Plus de produit cartésien, mais on traîne 540 rangées à 11 colonnes jusqu'au WHERE pour en garder 3.

rangées × colonnes de chaque cube transitoire (hauteur à l'échelle logarithmique)

Les JOIN ... ON sur les clés évitent le produit cartésien : 90 étudiants avec leur groupe, puis 540 inscriptions, puis 540 rangées avec le cours. Mais les conditions n'arrivent qu'au WHERE : on a construit et transporté un cube de 540 rangées × 11 colonnes pour en garder 3. Correct et lisible, mais pas économe si le moteur suit cet ordre.

Stratégie C : filtrer tôt, joindre du plus petit au plus grand

```
-- Stratégie C : filtrer tôt, joindre du plus petit au plus grand, les conditions dans les ON
SELECT AVG(inscription.note)
FROM groupe JOIN etudiant ON etudiant.groupe = groupe.id AND etudiant.sexe = 'F'
            JOIN inscription ON inscription.etudiant = etudiant.id
            JOIN cours ON cours.id = inscription.cours AND cours.titre = 'Exploitation de données'
WHERE groupe.nom = 'Info 2';
```

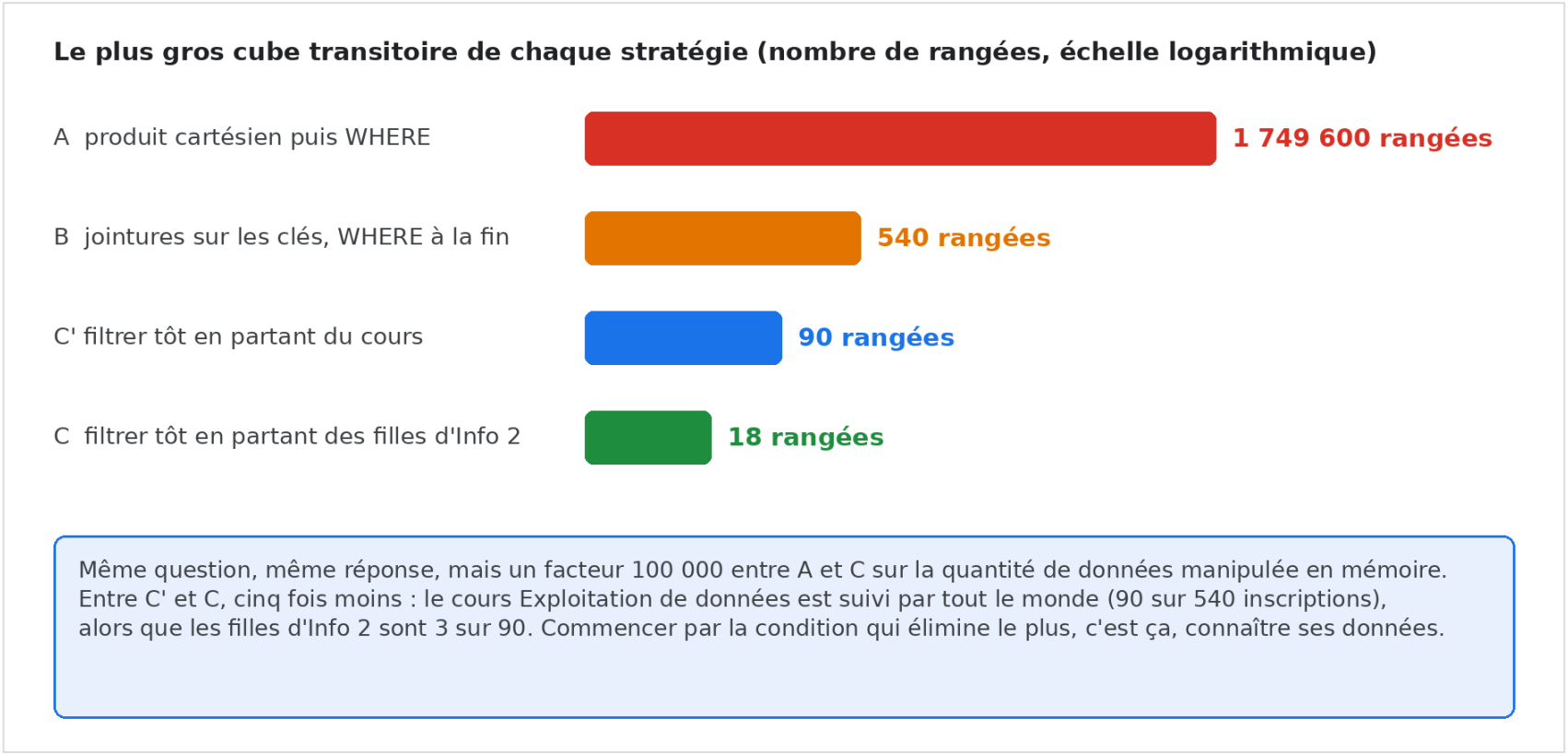


Le plus gros cube ne dépasse jamais 18 rangées : on part du plus petit ensemble et il ne grossit presque pas.

rangées × colonnes de chaque cube transitoire (hauteur à l'échelle logarithmique)

On part de ce qu'on sait être petit : le groupe Info 2 (1 rangée), ses filles (3), leurs inscriptions (3 × 6 = 18), puis le cours voulu (3). Les conditions sont dans les ON, là où elles réduisent le cube au moment où il se construit. Le plus gros cube ne dépasse pas 18 rangées, contre 540 en B et 1 749 600 en A.

Trois stratégies, la même réponse, trois tailles de cube



Le plus gros cube transitoire de chaque stratégie : 1 749 600 rangées (A), 540 (B), 90 si on filtre tôt mais en partant du cours (C', suivi par tout le monde), 18 en partant des filles d'Info 2 (C). Entre C' et C, seule la connaissance des données change : 3 filles sur 90 éliminent plus que 90 inscrits sur 540.

Par où commencer ? La sélectivité de chaque condition

Par où commencer ? La sélectivité d'une condition = la part des rangées qu'elle laisse passer

Condition	Rangées gardées	Sélectivité
etudiant.sexe = 'F'	12 sur 90	13 %
groupe.nom = 'Info 2' (via etudiant.groupe)	30 sur 90	33 %
les deux à la fois	3 sur 90	3 %
cours.titre = 'Exploitation de données' (via inscription.cours)	90 sur 540	17 %
inscription.note >= 60	environ 480 sur 540	89 % (presque inutile pour réduire)

Les deux règles pour ordonner les jointures

1. Commencer par l'ensemble le plus petit après filtrage : la condition la plus sélective, ou la table la plus petite. Ici, les filles d'Info 2 (3 rangées) plutôt que les inscrits au cours (90 rangées).
2. Chaque jointure multiplie : taille du cube suivant = rangées du cube courant × correspondances par rangée. 3 filles × 6 cours chacune = 18 ; 90 inscrits × 1 étudiant chacun = 90. Joindre d'abord ce qui multiplie le moins.

Ces chiffres, PostgreSQL les connaît aussi : ce sont ses statistiques (commande ANALYZE, vue pg_stats).

Sélectivité = part des rangées qu'une condition laisse passer. Deux règles : partir de l'ensemble le plus petit après filtrage (condition la plus sélective ou table la plus petite), puis joindre d'abord ce qui multiplie le moins, car chaque jointure produit rangées × correspondances par rangée. PostgreSQL tient ces chiffres à jour avec ANALYZE.

Écrire la condition dans le ON ou dans le WHERE ?

Conditions dans le WHERE

```
SELECT etudiant.prenom, inscription.note
FROM etudiant
JOIN inscription
  ON inscription.etudiant = etudiant.id
WHERE inscription.cours = 7
  AND etudiant.sexe = 'F';
```

=

La condition sur inscription dans le ON

```
SELECT etudiant.prenom, inscription.note
FROM etudiant
JOIN inscription
  ON inscription.etudiant = etudiant.id
 AND inscription.cours = 7
WHERE etudiant.sexe = 'F';
```

Avec une jointure interne (JOIN, INNER JOIN), les deux écritures sont équivalentes

Même résultat, et PostgreSQL produit le même plan : son optimiseur pousse chaque condition le plus tôt possible, vers la table qu'elle concerne (predicate pushdown), qu'elle soit écrite dans le ON ou dans le WHERE.

Pourquoi l'écrire dans le ON, alors ? Pour dire ce qu'on veut : « je joins seulement les inscriptions au cours 7 ». Le lecteur voit le cube qu'on construit, et c'est l'écriture qui reste juste le jour où le JOIN devient un LEFT JOIN.

En jointure interne, les deux écritures donnent le même résultat et le même plan : l'optimiseur pousse chaque condition vers la table qu'elle concerne, où qu'elle soit écrite (predicate pushdown). Le ON dit au lecteur quel cube on construit (« seulement les inscriptions au cours 7 ») et reste juste si le JOIN devient un LEFT JOIN.

Avec un LEFT JOIN, la place de la condition change le résultat

Condition dans le ON : on garde tous les étudiants

```
SELECT etudiant.prenom, inscription.note
FROM etudiant
LEFT JOIN inscription
  ON inscription.etudiant = etudiant.id
 AND inscription.cours = 7;
```

prenom	note
Léa	82
Maya	75
Sofia	NULL
Noah	90

4 rangées : Sofia n'est pas inscrite au cours 7, elle reste avec une note NULL.

Condition dans le WHERE : les absents disparaissent

```
SELECT etudiant.prenom, inscription.note
FROM etudiant
LEFT JOIN inscription
  ON inscription.etudiant = etudiant.id
WHERE inscription.cours = 7;
```

prenom	note
Léa	82
Maya	75
Noah	90

3 rangées : le WHERE rejette la rangée où inscription.cours est NULL. C'est redevenu un JOIN.

Avec un LEFT JOIN, le ON décide de ce qui est apparié (le cube de droite est filtré avant l'appariement) et le WHERE filtre le cube après l'appariement, NULL compris. Une condition sur la table de droite dans le WHERE annule le LEFT. Règle : ce qui décrit la jointure va dans le ON ; ce qui décrit le résultat final va dans le WHERE.

Condition dans le ON : le cube de droite est filtré avant l'appariement, tous les étudiants restent, Sofia avec une note NULL (4 rangées). Dans le WHERE : le filtre passe après et rejette la rangée où cours est NULL, Sofia disparaît (3 rangées), le LEFT JOIN est redevenu un JOIN. Le ON décrit la jointure, le WHERE décrit le résultat.

Forcer l'ordre quand on en sait plus que l'optimiseur

1. Une sous-requête dans le FROM : le petit cube d'abord

```
SELECT AVG(inscription.note)
FROM (SELECT id FROM etudiant
      WHERE sexe = 'F' AND groupe = 2) AS filles
JOIN inscription ON inscription.etudiant = filles.id
JOIN cours ON cours.id = inscription.cours
WHERE cours.titre = 'Exploitation de données';
```

2. Une CTE MATERIALIZED : calculée une fois, telle quelle

```
WITH filles AS MATERIALIZED (
  SELECT id FROM etudiant WHERE sexe = 'F' AND groupe = 2
)
SELECT AVG(inscription.note)
FROM filles
JOIN inscription ON inscription.etudiant = filles.id
JOIN cours ON cours.id = inscription.cours
 AND cours.titre = 'Exploitation de données';
```

3. L'ordre d'écriture devient l'ordre d'exécution

```
SET join_collapse_limit = 1;   -- pour cette session : PostgreSQL ne réordonne plus les JOIN explicites
SELECT AVG(inscription.note)
FROM groupe JOIN etudiant ON etudiant.groupe = groupe.id AND etudiant.sexe = 'F'
            JOIN inscription ON inscription.etudiant = etudiant.id
            JOIN cours ON cours.id = inscription.cours AND cours.titre = 'Exploitation de données'
WHERE groupe.nom = 'Info 2';
```

Par défaut, PostgreSQL réordonne librement jusqu'à 8 tables (join_collapse_limit) et choisit lui-même par où commencer. Ces trois écritures servent quand on en sait plus que lui : au-delà de 8 tables, statistiques périmées, colonnes corrélées, ou simplement pour rendre le cube voulu visible dans le code. À vérifier avec EXPLAIN ANALYZE, jamais à l'aveugle.

Trois écritures pour imposer le petit cube d'abord : une sous-requête dans le FROM, une CTE WITH ... AS MATERIALIZED (calculée une fois, telle quelle), ou SET join_collapse_limit = 1 pour que l'ordre d'écriture devienne l'ordre d'exécution. À réserver aux cas où l'optimiseur se trompe, et à vérifier avec EXPLAIN ANALYZE.

Ce que PostgreSQL fait tout seul avant d'exécuter

- **L'optimiseur (planner) :** il réécrit la requête en un plan d'exécution, un arbre de noeuds dont chacun produit un cube pour le noeud au-dessus.
- **Il pousse les conditions :** chaque condition est appliquée le plus tôt possible, sur la table qu'elle concerne, que vous l'ayez écrite dans le ON ou dans le WHERE.
- **Il réordonne les jointures :** jusqu'à 8 tables (join_collapse_limit), il essaie les ordres possibles et garde le moins coûteux ; votre ordre d'écriture n'est qu'une suggestion.
- **Il s'appuie sur ses statistiques :** nombre de rangées, valeurs les plus fréquentes, histogrammes par colonne, mis à jour par ANALYZE (ou l'autovacuum). Statistiques périmées, mauvais plan.
- **Il estime, il ne mesure pas :** EXPLAIN montre ses estimations, EXPLAIN ANALYZE exécute vraiment et montre les rangées réelles de chaque cube.

Pour rafraîchir les statistiques d'une table et voir ce qu'il en sait :

```
ANALYZE etudiant;
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'etudiant';
```

En clair : sur une base bien entretenue, PostgreSQL trouve souvent la stratégie C tout seul, même si vous écrivez B. Votre travail, c'est de lui donner des clés, des index et des statistiques à jour, puis de vérifier son plan.

Lire un plan avec EXPLAIN ANALYZE : chaque noeud est un cube

```
EXPLAIN ANALYZE SELECT AVG(inscription.note) FROM ...  -- stratégie C

Aggregate                                          (actual rows=1)
  -> Nested Loop                                   (actual rows=3)
       -> Nested Loop                              (actual rows=18)
            -> Nested Loop                         (actual rows=3)
                 -> Seq Scan on groupe             (actual rows=1)
                      Filter: (nom = 'Info 2')     Rows Removed by Filter: 2
                 -> Seq Scan on etudiant           (actual rows=3)
                      Filter: (sexe = 'F' AND groupe = groupe.id)   Rows Removed: 87
            -> Index Scan on inscription (etudiant) (actual rows=6, loops=3)
       -> Index Scan on cours (id)                  (actual rows=1, loops=18)
            Filter: (titre = 'Exploitation de données')   Rows Removed: 15
```

Comment lire

De l'intérieur vers l'extérieur : la ligne la plus indentée s'exécute en premier.

actual rows = la hauteur réelle du cube produit par ce noeud.

loops = combien de fois le noeud a été rejoué ; rangées totales = rows × loops.

Rows Removed by Filter = ce que la condition a éliminé à cet endroit.

Plan simplifié (coûts et temps retirés) pour l'exemple de ce diaporama : les 3 filles d'Info 2, leurs 18 inscriptions, les 3 au cours.
Sans EXPLAIN, on suppose ; avec EXPLAIN (estimations) et EXPLAIN ANALYZE (mesures réelles), on voit chaque cube et sa vraie taille.

Plan simplifié de la stratégie C, lu de l'intérieur vers l'extérieur : groupe garde 1 rangée, etudiant 3, l'index sur inscription en ramène 6 par étudiant (18), l'index sur cours en garde 3, Aggregate en fait 1. actual rows est la hauteur réelle de chaque cube, loops le nombre de passages, Rows Removed ce que le filtre a éliminé.

Comment les cubes se fabriquent en mémoire : trois algorithmes de jointure

- **Nested Loop :** pour chaque rangée du petit cube, on cherche ses correspondances dans l'autre table, idéalement par un index. Parfait quand le cube de départ est petit (nos 3 filles).
- **Hash Join :** le moteur met le plus petit des deux cubes dans une table de hachage en mémoire, puis parcourt l'autre une seule fois. Le petit cube est littéralement en RAM : le filtrer d'abord, c'est réduire cette table.
- **Merge Join :** les deux cubes triés sur la clé de jointure sont parcourus en parallèle. Bon pour deux gros ensembles déjà triés ou indexés.
- **work_mem :** la mémoire accordée à chaque tri ou hachage (4 Mo par défaut). Un cube plus gros que ça déborde sur le disque, en lots (batches), et tout ralentit.

Le plan vous dit lequel a été choisi et s'il a débordé :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT ...;
-- dans le plan : Hash Join  Buckets: 1024  Batches: 1  Memory Usage: 9kB  (Batches > 1 = déborde)
SET work_mem = '64MB';   -- pour la session, si un hachage ou un tri déborde
```

Le choix de l'algorithme découle de la taille estimée des cubes. Un cube de 18 rangées donne un Nested Loop léger ; un cube de 540 rangées donne un Hash Join qui occupe de la mémoire ; un cube de 1 749 600 rangées déborde de work_mem et va sur le disque.

Quand votre écriture compte vraiment

- **Plus de 8 tables :** au-delà de join_collapse_limit, PostgreSQL garde votre ordre d'écriture pour les JOIN explicites. Écrivez alors du plus petit au plus grand.
- **LEFT JOIN et conditions :** la place de la condition (ON ou WHERE) change le résultat, pas seulement le plan.
- **Fonctions sur une colonne :** WHERE lower(prenom) = 'léa' ou WHERE note + 5 >= 60 empêche l'index de servir ; comparez la colonne nue à une valeur calculée.
- **OR entre tables et NOT IN :** souvent mal estimés ; un UNION ou un NOT EXISTS donne des cubes plus petits et plus prévisibles.
- **Colonnes corrélées :** l'optimiseur suppose les conditions indépendantes (sexe et groupe) ; si elles ne le sont pas, CREATE STATISTICS sur les deux colonnes corrige ses estimations.
- **CTE MATERIALIZED et sous-requêtes :** une barrière volontaire, le cube est calculé tel quel avant le reste.

```
CREATE STATISTICS etudiant_sexe_groupe (dependencies) ON sexe, groupe FROM etudiant;
ANALYZE etudiant;
```

Dans tous les autres cas, l'ordre des conditions et des jointures que vous écrivez est surtout une question de lisibilité : dites ce que vous savez des données, laissez l'optimiseur exécuter, et vérifiez.

Les règles du pouce pour garder les cubes petits

- **Filtrer tôt :** la condition la plus sélective en premier, dans le ON de la jointure qu'elle concerne ou dans une sous-requête.
- **Partir du plus petit :** l'ensemble le plus réduit après filtrage, puis joindre ce qui multiplie le moins (rangées × correspondances par rangée).
- **Joindre sur des clés indexées :** clé primaire d'un côté, clé étrangère indexée de l'autre, sinon chaque jointure relit toute la table.
- **Ne demander que les colonnes utiles :** SELECT * transporte des cubes larges ; la largeur compte autant que la hauteur.
- **Agréger le plus tard possible mais sur le plus petit cube possible :** filtrer avant GROUP BY (WHERE), pas après (HAVING), quand la condition ne porte pas sur l'agrégat.
- **Mesurer :** EXPLAIN ANALYZE avant et après chaque réécriture ; les statistiques à jour avec ANALYZE.

Ces règles décrivent ce qu'un bon optimiseur fait déjà. Les appliquer en écrivant, c'est s'assurer que la requête reste bonne quand les données grossissent, quand la base change de moteur, et quand quelqu'un d'autre la relit.

Exercice : ordonner les jointures d'une requête

- **Le contexte :** une école de 2 000 étudiants dans 70 groupes, 15 000 inscriptions, 60 cours, 300 enseignants. On veut les prénoms des étudiants du groupe « Info 2 » qui ont eu moins de 60 au cours « Exploitation de données », avec le nom de l'enseignant du cours.
- **Ce qu'on sait des données :** un groupe compte environ 30 étudiants ; un étudiant a environ 7 inscriptions ; un cours est donné par un seul enseignant ; environ 10 % des notes sont sous 60.
- **À faire :** 1. donner la sélectivité de chaque condition ; 2. choisir l'ordre des jointures et la place de chaque condition (ON ou WHERE) ; 3. estimer la hauteur du plus gros cube transitoire ; 4. écrire la requête ; 5. dire ce que vous vérifieriez avec EXPLAIN ANALYZE.

```
-- Tables : groupe(id, nom)  etudiant(id, prenom, groupe)  inscription(etudiant, cours, note)
--          cours(id, titre, enseignant)  enseignant(id, nom)
```

Travaillez sur papier avec les chiffres donnés, sans base de données. Le corrigé est à la diapositive suivante : ne le regardez qu'après avoir estimé vos cubes.

Corrigé de l'exercice

- **Sélectivités :** groupe « Info 2 » garde 1 groupe sur 70, soit 30 étudiants sur 2 000 (1,5 %) ; le cours « Exploitation de données » garde 1 cours sur 60 ; note < 60 garde 10 % des inscriptions.
- **Ordre :** partir du groupe (1 rangée), joindre ses étudiants (30), joindre leurs inscriptions (30 × 7 = 210), joindre le cours voulu (30 au plus, un cours par étudiant), filtrer la note (environ 3), joindre l'enseignant (3). Plus gros cube : 210 rangées.
- **Mauvais ordre à titre de comparaison :** partir des notes sous 60 (1 500 rangées), puis les étudiants (1 500), puis le groupe : plus gros cube 1 500, sept fois plus, pour la même réponse.
- **Place des conditions :** nom du groupe et titre du cours dans les ON (elles décrivent la jointure) ; la note dans le WHERE ou dans le ON de l'inscription, c'est équivalent en jointure interne.

```
SELECT etudiant.prenom, enseignant.nom
FROM groupe JOIN etudiant ON etudiant.groupe = groupe.id
JOIN inscription ON inscription.etudiant = etudiant.id AND inscription.note < 60
JOIN cours ON cours.id = inscription.cours AND cours.titre = 'Exploitation de données'
JOIN enseignant ON enseignant.id = cours.enseignant WHERE groupe.nom = 'Info 2';
```

À vérifier avec EXPLAIN ANALYZE : que le premier noeud exécuté soit bien le groupe (1 rangée), que les jointures utilisent des index (Index Scan) et que actual rows reste près de 210 au plus haut. Si le plan part des 1 500 notes, les statistiques sont probablement périmées.

En résumé : penser en cubes

- Une requête est une suite de cubes transitoires en mémoire ; le coût, c'est leur taille, rangées et colonnes.
- **Filtrer tôt et partir du plus petit ensemble :** la condition la plus sélective d'abord, puis les jointures qui multiplient le moins.
- **ON ou WHERE :** équivalents en jointure interne, différents en LEFT JOIN ; le ON décrit la jointure, le WHERE décrit le résultat.
- **PostgreSQL réordonne et pousse les conditions lui-même jusqu'à 8 tables, à partir de statistiques :** gardez-les à jour, donnez-lui des index.
- **Quand on en sait plus que lui :** sous-requête, CTE MATERIALIZED, join_collapse_limit, CREATE STATISTICS. Et toujours EXPLAIN ANALYZE pour voir les vrais cubes.

La phrase à retenir : connaître ses données, c'est savoir par où commencer.

Essayer le simulateur des cubes transitoires (donnees.projet.autos/outils/cubes-transitoires)

Le simulateur reprend les données de ce diaporama : composez la requête par blocs, déplacez une condition, changez l'ordre des jointures et regardez les cubes changer de taille, avec cinq défis. Ensuite, sur une vraie base : EXPLAIN ANALYZE sur une de vos requêtes, trouver le plus gros cube et le réduire.