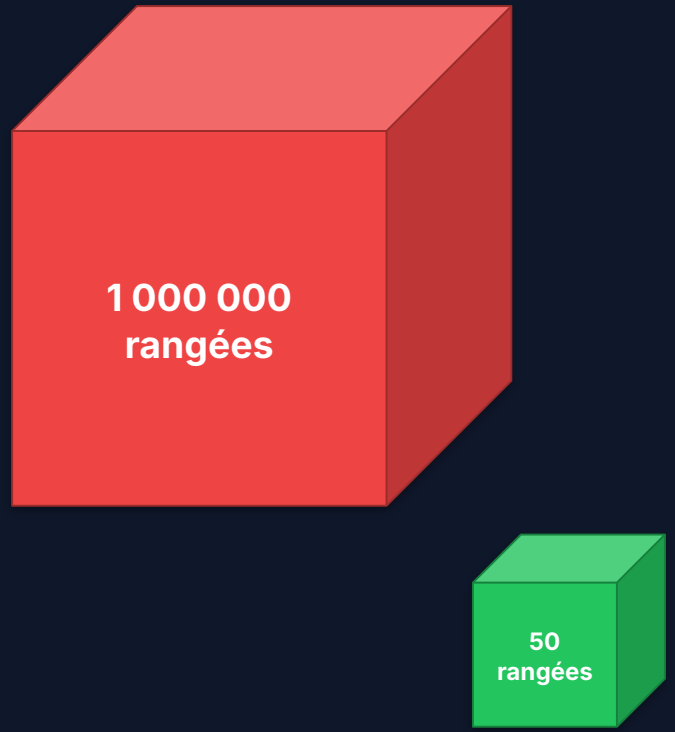
EXPLOITATION DE DONNÉES

Cubes transitoires

L'alchimie des requêtes SQL : ON ou WHERE, et l'ordre des jointures

Série Math, version imagée. La version détaillée est le diaporama 6.



Même résultat à l'écran, deux cubes en mémoire.

Que se passe-t-il sous le capot ?

ÉTAPE 01

Vous cliquez sur Exécuter

Le résultat final n'apparaît pas par magie. Entre votre clic et la grille de résultats, le moteur SQL travaille.

ÉTAPE 02

Le moteur construit en RAM

Il fabrique des tableaux intermédiaires en mémoire vive : des rangées croisées, filtrées, regroupées, une étape à la fois.

ÉTAPE 03

Les cubes transitoires

Ces tableaux n'existent qu'une fraction de seconde : ce sont les cubes transitoires. Leur taille fait la différence entre une requête qui file et une requête qui rame.

À RETENIR

Le coût d'une requête, c'est la taille des cubes qu'elle fabrique dans le noir. **Ce diaporama apprend à les garder petits.**

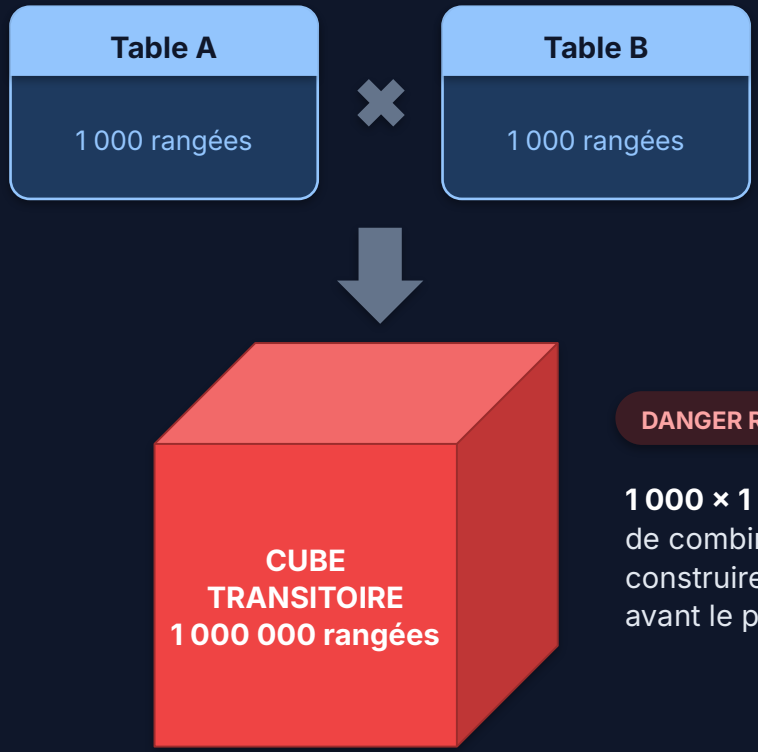
Le concept : le cube transitoire

DÉFINITION

Chaque fois que vous croisez deux tables, le moteur multiplie et croise les rangées avant de pouvoir filtrer.

Le tableau qui en résulte est un cube transitoire : il vit en mémoire vive le temps de la requête, puis disparaît.

Si vous croisez deux grosses tables sans filtre, le cube explose !



DANGER RAM

1 000 × 1 000 = 1 000 000 de combinaisons à construire en mémoire avant le premier filtre.

Le premier dilemme : quand filtrer ?

Où placer nos conditions ?

```
a.id = b.a_id  AND  a.statut = 'actif'
```

CLAUSE WHERE

Filtrage a posteriori

Filtre exécuté **après** la construction du cube transitoire : on croise tout, puis on passe le balai.

```
FROM a, b
WHERE a.id = b.a_id AND a.statut = 'actif'
```

CLAUSE ON

Filtrage a priori

Filtre appliqué **pendant** la construction du cube : les rangées refusées n'y entrent jamais.

```
FROM a JOIN b
ON a.id = b.a_id AND a.statut = 'actif'
```

IMPACT PERFORMANCE

Le résultat affiché à l'écran sera identique. **Mais la souffrance du serveur sera très différente !**

Scénario catastrophe : tout dans le WHERE

ÉTAPE 01

Le moteur SQL croise toutes les rangées de A avec toutes celles de B, sans se poser de question.

ÉTAPE 02

Il sature la RAM avec un cube géant : 1 000 × 1 000 = 1 000 000 de rangées.

ÉTAPE 03

Ensuite seulement, il passe le balai pour ne garder ce que le WHERE demande : 50 rangées.

```
SELECT * FROM a, b
WHERE a.id = b.a_id
  AND a.statut = 'actif';
```

ALERTE MOTEUR

Géant rouge

La requête rame, le serveur pleure.



Scénario optimal : le ON comme portier

ÉTAPE 01

Les conditions écrites dans le ON agissent comme un portier de boîte de nuit.

ÉTAPE 02

Les rangées qui ne correspondent pas sont refusées AVANT d'entrer dans le cube.

ÉTAPE 03

Le cube transitoire reste minuscule et maniable : 50 rangées, pas une de plus.

```
SELECT * FROM a
JOIN b ON a.id = b.a_id
      AND a.statut = 'actif';
```

MOTEUR SEREIN

Petit et rapide

La requête file, le serveur respire.

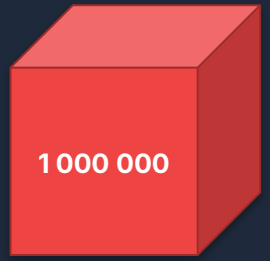


En pointillé : le cube géant qu'on a évité.

Face à face : 1 000 000 contre 50

TOUT DANS LE WHERE

```
SELECT * FROM a, b
WHERE a.id = b.a_id
  AND a.statut = 'actif';
```



Cube géant : la RAM surchauffe.

TOUT DANS LE ON

```
SELECT * FROM a
JOIN b ON a.id = b.a_id
      AND a.statut = 'actif';
```



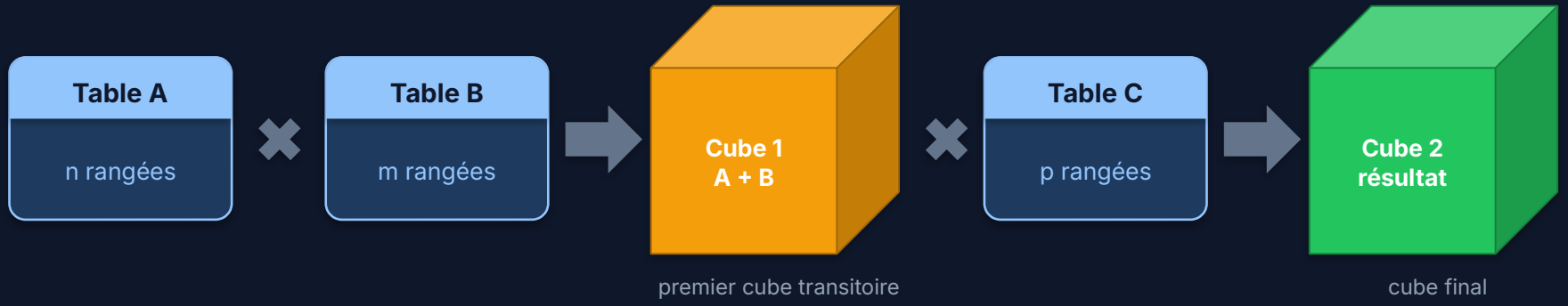
Cube minuscule : la requête est fluide.

BILAN

Même résultat à l'écran : 50 rangées. **Mais un cube 20 000 fois plus gros en mémoire dans le premier cas.**

Le deuxième dilemme : l'ordre des jointures

Avec trois tables ou plus (A JOIN B JOIN C), par quoi commencer ? L'ordre d'écriture influence l'ordre de construction des cubes : A et B donnent un premier cube, qui est ensuite joint à C.



LA QUESTION

Vaut-il mieux commencer par (A + B) ou par (B + C) ? Le premier cube est souvent le plus gros : c'est lui qu'il faut garder petit. La réponse dépend de ce qu'on sait des données, et c'est l'objet de la règle d'or.

La règle d'or : l'entonnoir

SÉLECTIVITÉ

Le but du jeu : réduire la taille des données le plus vite possible. C'est ce qu'on appelle la sélectivité.

Une condition très sélective élimine beaucoup de rangées d'un coup. Exemple : chercher une personne par son courriel exact.

Il faut TOUJOURS commencer ses jointures par la table ou la condition qui élimine le plus de monde.



Chaque condition est un étage de l'entonnoir : on met le plus étroit en premier.

Mise en situation : les filles d'un groupe

Nous cherchons les inscriptions des étudiantes (filles) du groupe 42. Voici ce qu'on sait des données.

etudiant	inscription	groupe 42, filles
10 000 rangées	50 000 rangées (5 par étudiant)	3 rangées seulement !

```
etudiant(id, prenom, sexe, groupe)
inscription(etudiant, cours, note)
```

LA CIBLE

3 filles × 5 cours = 15 rangées. Tout le reste, c'est du transport inutile en mémoire.

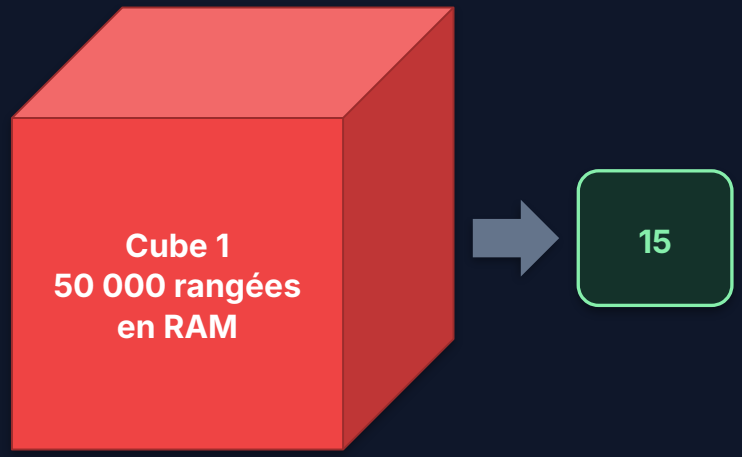
Le mauvais chemin : l'explosion RAM

ORDRE NAÏF

Si on joint d'abord TOUS les étudiants à TOUTES les inscriptions...

...puis qu'on filtre les filles du groupe 42, on a transporté une montagne pour trouver 3 cailloux.

```
SELECT * FROM etudiant
JOIN inscription
  ON inscription.etudiant = etudiant.id
WHERE etudiant.groupe = 42
  AND etudiant.sexe = 'F';
```



50 000 rangées construites en mémoire pour en garder 15.

Le serveur a fait 3 333 fois trop de travail.

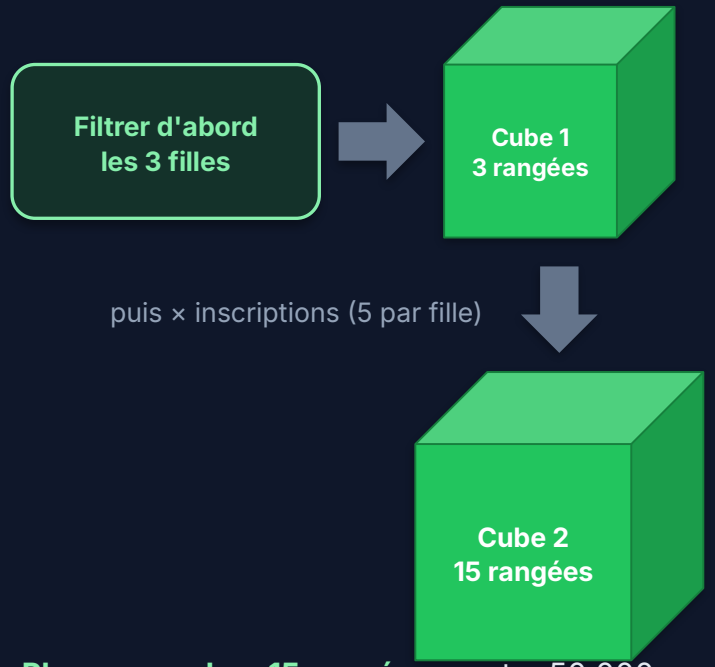
Le bon chemin : la voie des ninjas SQL

ORDRE NINJA

On attaque par l'entonnoir le plus serré : les 3 filles du groupe 42 d'abord, les inscriptions ensuite.

En joignant ces 3 rangées aux inscriptions, le cube final reste ridicule. **Le serveur vous remercie.**

```
SELECT * FROM (
  SELECT * FROM etudiant
  WHERE groupe = 42 AND sexe = 'F'
) AS filles
JOIN inscription
  ON inscription.etudiant = filles.id;
```

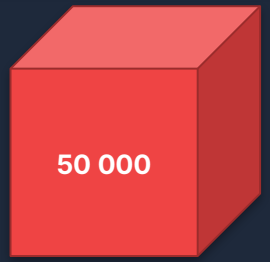


Plus gros cube : 15 rangées, contre 50 000 sur le mauvais chemin.

Face à face : 50 000 contre 15

ORDRE NAÏF

```
SELECT * FROM etudiant
JOIN inscription
  ON inscription.etudiant = etudiant.id
WHERE etudiant.groupe = 42
  AND etudiant.sexe = 'F';
```



Montagne de 50 000 rangées pour 3 cailloux.

ORDRE NINJA

```
SELECT * FROM (SELECT * FROM etudiant
  WHERE groupe = 42 AND sexe = 'F') AS filles
JOIN inscription
  ON inscription.etudiant = filles.id;
```



Trois rangées, puis quinze : le cube reste ridicule.

BILAN

Même 15 rangées à l'écran. **Mais un cube 3 333 fois plus gros en mémoire dans le premier cas.**

Bilan de l'alchimiste

LA BONNE QUESTION

Un bon développeur ne se demande pas seulement « Est-ce que ma requête fonctionne ? ». Il se demande : **« Quelle est la taille des monstres que je crée dans le noir, en RAM, pendant l'exécution ? »**

RÈGLE 1

Filtrer au plus tôt

Mettre chaque condition dans le ON de la jointure qu'elle concerne : le portier refuse les rangées avant qu'elles entrent dans le cube.



RÈGLE 2

Partir du plus petit

Commencer les jointures par la table ou la condition qui réduit le plus la voilure : l'entonnoir le plus serré d'abord.



Et en vrai ? L'optimiseur veille au grain

CE QUE POSTGRESQL FAIT TOUT SEUL

- Il pousse chaque condition le plus tôt possible, qu'elle soit écrite dans le ON ou dans le WHERE : en jointure interne, même plan d'exécution.
- Il réordonne lui-même les jointures (jusqu'à 8 tables) à partir de ses statistiques, mises à jour par ANALYZE.
- Le cube géant existe pour vrai quand on oublie une condition de clé, quand un LEFT JOIN est mal filtré ou quand les statistiques sont périmées.
- Écrire comme un ninja reste la bonne habitude : la requête reste bonne quand les données grossissent et quand quelqu'un d'autre la relit.

```
EXPLAIN ANALYZE SELECT ...;   -- la hauteur réelle de chaque cube
```

POUR ALLER PLUS LOIN

Diaporama 7 : Les cubes de données transitoires

La version détaillée : stratégies A, B, C, EXPLAIN ANALYZE, algorithmes de jointure, exercice et corrigé.

donnees.projet.autos/serie-math/7-cubes-transitoires.html

Le simulateur des cubes transitoires

Composez la requête bloc par bloc, regardez les cubes se dessiner, battez le record.

donnees.projet.autos/outils/cubes-transitoires/