

Traitement et optimisation de requêtes

Dr N. BAME

Plan

- Traitement de requêtes SQL
- Modèles et estimation de coût
- Espace de recherche et règles de transformation
- Optimisation de requêtes avec SQL

Introduction

- Soit une requête : comment comprendre, analyser et améliorer son exécution.
 - En comprenant comment on passe de SQL à des opérations sur des tables.
 - En connaissant l'utilisation des index et des algorithmes implantant ces opérations
 - En comprenant comment l'optimiseur effectue un choix parmi les opérations possibles.
- Ce sont des techniques de base, implantées dans tout SGBD relationnel

Traitement de requêtes

Requête déclarative (SQL)

Compilateur/Optimiseur

Générateur
de plans

Estimation
de coûts

Plan d'exécution « optimal »
(accès physique aux données)

Exécution

Optimisation

Requête SQL

Analyse

Simplification

Normalisation

Restructuration

Plan d'exécution "optimal"

Répertoire
Système :

Schéma
Index
Statistiques

Analyse

- Analyse **lexicale** et **syntaxique**
 - ✓ Validation par rapport à la syntaxe SQL
 - ✓ Vérification des types :
 - ✓ **présence** des attributs et relations (ou vues) dans le schéma
 - ✓ **compatibilité** des types dans les prédicats
 - ✓ **Décomposition** en requêtes/sous-requêtes
- Le résultat est un ***plan d'exécution logique*** qui représente la requête sous **la forme d'un arbre composé des opérations de l'algèbre relationnelle**
 - Traduction algébrique «directe» : $\Pi (\sigma (x))$

Optimisation

- L'analyse traduit la requête SQL en un plan logique.
- Ce plan logique **est transformé en un plan d'exécution physique**,
 - comprenant les opérations **d'accès aux données** et les **algorithmes d'exécution**, placées dans un **ordre supposé optimal** (ou quasi-optimal).
- Le choix du ***meilleur plan*** dépend des opérations physiques (opérateurs de l'algèbre), **disponibles dans le processeur** de requêtes.
 - Il dépend également **des chemins d'accès aux fichiers**,
 - Existence d'index ou de tables de hachage.
 - Enfin, il peut s'appuyer sur **des données statistiques** enregistrées ou estimées pour chaque relation

Optimiseur

- Un module du SGBD, est chargé de :
 1. prendre en **entrée une requête**, et la mettre sous forme d'opérations
 2. se fixer comme **objectif l'optimisation d'un certain paramètre** (en général le temps d'exécution)
 3. **construire un programme** s'appuyant sur les index existant, et les opérations disponibles.

Exécution de la requête

- Le plan d'exécution est finalement compilé, ce qui fournit un programme d'un type assez particulier.
- L'exécution de ce programme calcule le résultat de la requête.
- Ce qui fait la particularité d'un plan d'exécution physique, c'est en premier lieu qu'il s'agit d'un programme sous forme **d'arbre**, dans lequel.
 - chaque **nœud** correspond à une **opération appliquée aux données** issues des **nœuds inférieurs**.
- Un mécanisme de transfert des données entre les nœuds (***pipelining***) évite de stocker les résultats intermédiaires.

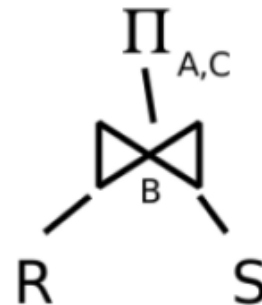
Arbre algébrique

- Arbre représentant une requête dont
 - les nœuds terminaux représentent les relations,
 - les nœuds intermédiaires des opérations de l'algèbre relationnelle,
 - le nœud racine le résultat d'une requête, et
 - les arcs les flux de données entre les opérations.

Traduction de SQL en AR

- Règles de passage d'une requête SQL en AR :
 - **SELECT** pour définir une **projection**
 - **FROM** pour définir les **feuilles de l'arbre**
 - **WHERE** pour définir :
 - avec les comparaisons **attribut/valeur** des sélections
 - avec les comparaisons **attribut/attribut** des jointures

```
SELECT R.a, S.c  
FROM R, S  
WHERE R.b = S.b;
```

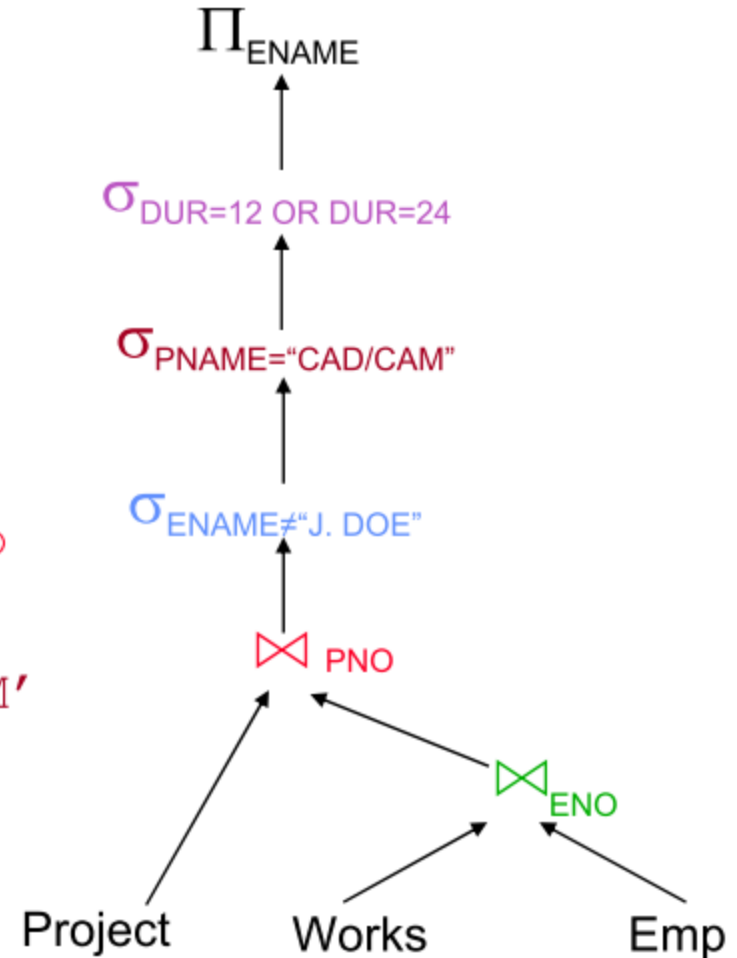


Traduction en algèbre

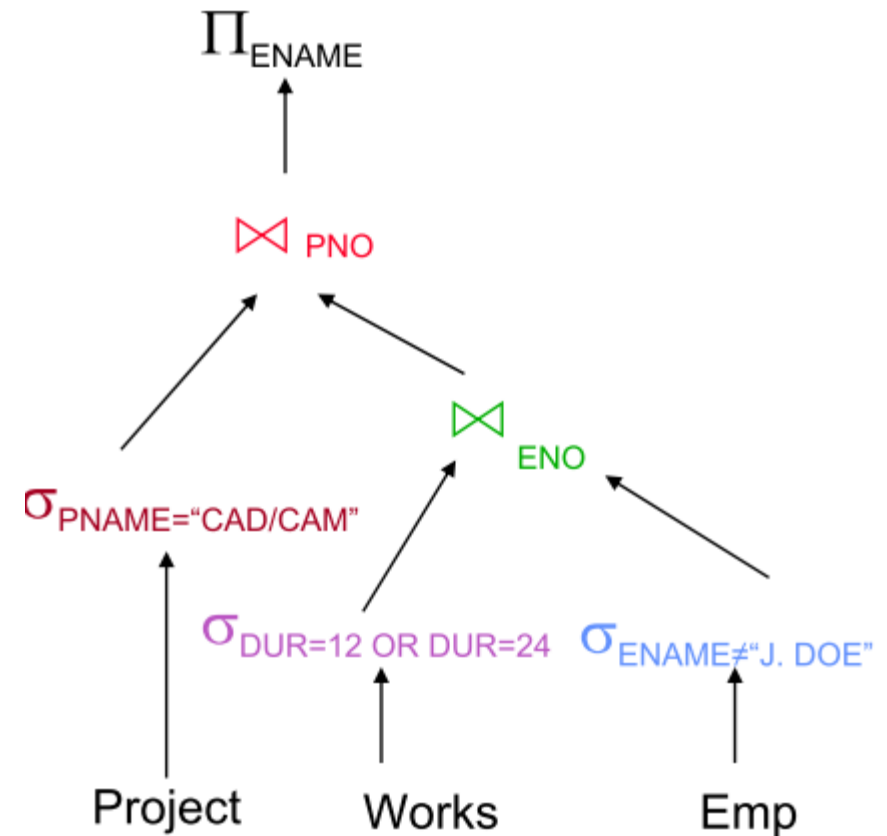
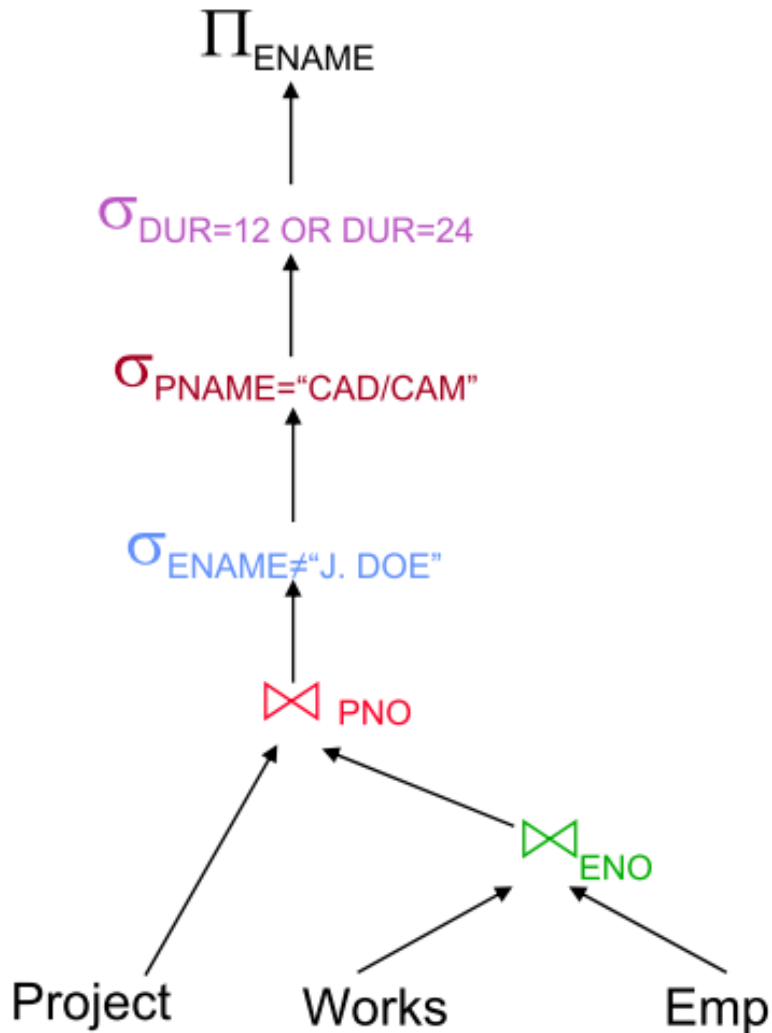
Conversion en arbre algébrique

Exemple :

```
SELECT  Ename
FROM    Emp, Works, Project
WHERE   Emp.Eno = Works.Eno
AND     Works.Pno = Project.Pno
AND     Emp.Ename <> 'J.Doe'
AND     Project.name = 'CAD/CAM'
AND     (Works.Dur=12 OR
           Works.Dur=24)
```



Alternatives



Exercice

- Soit le schéma relationnel ci-dessous :

Emp (Eno, Ename, #Title, City)

Pay(Title, Salary)

Project(Pno, Pname, Budget, City)

Works(#Eno, #Pno, Resp, Dur)

- Déterminer les arbres algébriques des requêtes suivantes :
 - ✓ Noms de tous les projets
 - ✓ Noms et budgets des projets de Paris?
 - ✓ Noms des projets de budget > 225?
 - ✓ Noms et budgets des projets où travaille l'employé E1?

Optimisation de requête

Objectif : trouver le plan d'exécution d'une requête dont le **coût soit minimal** (i.e., le temps de réponse à la requête soit le plus rapide possible)

Fonction de coût : donne une estimation du coût total réel d'un plan d'exécution

- **Coût total = coût I/O + coût CPU**
- On peut **négliger le coût CPU** : **coût I/O = 1000 * coût CPU**

- **Problème 1** : Définition d'une bonne fonction de coût

- **Solution** : **statistiques et estimations**

- **Problème 2** : Taille de l'espace de recherche

- **Solution** : **recherche non-exhaustive** d'une **bonne solution** (pas forcément la meilleure) avec des **heuristiques**

Support de l'optimisation d'une requête

L'optimisation s'appuie sur les éléments suivants :

1. Le ***schéma de la base, description des tables*** et ***chemins d'accès*** (dans le catalogue)
2. Des **statistiques** : ***taille des tables, des index, distribution des valeurs***
3. **Des algorithmes** : il peuvent différer selon les systèmes

Important : on suppose que le temps d'accès à ces informations est négligeable par rapport à l'exécution de la requête.

Estimation du coût d'un plan

- La fonction de coût donne une **estimation** des temps I/O et CPU
 - nombre d'instructions et d'accès disques (écriture/lecture)
- Estimation du *nombre d'accès disque pendant l'évaluation* de chaque *nœud* de l'arbre algébrique
 - utilisation de **pipelines (en mémoire)** ou de **relations temporaires** (écrites sur disque)
- Estimation de *la taille du résultat de chaque nœud par rapport à ses entrées* :
 - **sélectivité des opérations** – “facteur de réduction”

Implémentation et coût des opérateurs

- **Sélection** (avec R contenant ***n pages*** disques) :
 - parcours séquentiel (**scan**)
 - le nombre d'***accès disques*** est en **$O(n)$**
 - ***sélection*** avec index
 - index B+ : **$O(\log(n))$**
 - index haché : **$O(1)$**
- **Projection** :
 - sans élimination des doublons : **$O(n)$**
 - avec élimination des doublons
 - en triant : **$O(n*\log(n))$**
 - en hachant : **$O(n+t)$** où t = nombre de pages du résultat

Implémentation et coût des opérateurs

Jointure (avec R et S contenant **n** et **m** pages disques)

- **boucle imbriquée (nested loop)**: $T = R \bowtie_{R.A=S.B} S$

```
foreach tuple r ∈ R do
  foreach tuple s ∈ S do
    if r.A == s.B then T = T + <r, s>
```

$$O(n * m)$$

- Amélioration possible pour réduire les accès disques
 - boucles imbriquées par **blocs de k pages**
 - $O(n + (n / k) * m)$, où $k \sim \#$ pages mémoire disponibles
 - Cas le plus simple : R et/ou S tient en mémoire centrale ($n/k = 1$)

Implémentation et coût des opérateurs

$$T = R \bowtie_{R.A=S.B} S$$

- **Boucle imbriquée et index sur attribut S.B : $O(n * \log(m))$**

```
foreach tuple r ∈ R do
    foreach tuples s ∈ S accédé par indexS.B(r.A) do
        T = T + <r, s>
```

- **Jointure par tri-fusion : $O(n * \log(n) + m * \log(m))$**
 - trier R et S sur les attributs de jointure (R.A, S.B) : $n * \log(n) + m * \log(m)$
 - fusionner les relations triées : $n+m$ (jointure sur une clé)
- **Jointure par hachage : $O(3 * (n+m))$**
 - hacher R et S avec la même fonction de hachage H : $(n+m)$ lectures, $(n+m)$ écritures
 - pour chaque paquet i de R et de S, joindre les tuplets où $r.A=s.B$: $m+n$ lectures

Estimation de la taille des résultats intermédiaires : Statistiques

- **Relation R:**
 - cardinalité : $\text{card}(R)$
 - largeur (arité) de R : taille d'un nuplet
 - fraction de nuplets participant à une sélection, jointure, ...
- **Attribut A:**
 - domaine de A : $\text{dom}(A)$
 - valeurs $\text{max}(A)$, $\text{min}(A)$, ... dans la base de données
 - nombre de valeurs distinctes dans la base de données
 - distribution des valeurs (histogrammes) dans la base de données
- *Hypothèses (dans la suite) :*
 - indépendance entre les différentes valeurs d'attributs
 - distribution uniforme des valeurs d'attribut dans leur domaine

Tailles des relations intermédiaires

Sélection :

$$taille(R) = card(R) * largeur(R)$$

$$card(\sigma_F(R)) = SF_\sigma(F) * card(R)$$

où SF_σ est une estimation de la *sélectivité* du prédicat :

$$SF_\sigma(A = valeur) = \frac{1}{card(\Pi_A(R))}$$

$$SF_\sigma(A > valeur) = \frac{max(A) - valeur}{max(A) - min(A)} \quad SF_\sigma(A < valeur) = \frac{valeur - min(A)}{max(A) - min(A)}$$

$$SF_\sigma(p(A_i) \wedge p(A_j)) = SF_\sigma(p(A_i)) * SF_\sigma(p(A_j))$$

$$SF_\sigma(p(A_i) \vee p(A_j)) = SF_\sigma(p(A_i)) + SF_\sigma(p(A_j)) - (SF_\sigma(p(A_i)) * SF_\sigma(p(A_j)))$$

$$SF_\sigma(A \in valeur) = SF_\sigma(A = valeur) * card(\{valeurs\})$$

Tailles des relations intermédiaires

Projection :

$$\text{card}(\Pi_A(R)) \leq \text{card}(R) \text{ (égalité si A est unique)}$$

Produit cartésien :

$$\text{card}(R \times S) = \text{card}(R) * \text{card}(S)$$

Union :

$$\text{borne sup. : } \text{card}(R \cup S) = \text{card}(R) + \text{card}(S)$$

$$\text{borne inf. : } \text{card}(R \cup S) = \max\{\text{card}(R), \text{card}(S)\}$$

Différence :

$$\text{borne sup. : } \text{card}(R - S) = \text{card}(R)$$

$$\text{borne inf. : } 0$$

Tailles des relations intermédiaires

Jointure :

cas particulier: A est clé de R et B est clé étrangère de S :

$$\text{card}(R \bowtie_{A=B} S) = \text{card}(S)$$

plus généralement

$$\text{card}(R \bowtie S) = SF_J * \text{card}(R) * \text{card}(S)$$

Exercice

- Soit le schéma relationnel ci-dessous :

Emp (Eno, Ename, #Title, City)

Pay(Title, Salary)

Project(Pno (5), Pname(50), Budget(15), City(50))

Works(#Eno(5), #Pno (5), Resp(30), Dur(10))

- **Stats**

- Project :

- Card=**1200**, Sél(city='v') = **5%**, Sél(budget=v) = **10%**, Min(Budget)=**50**, Max(Budget)=**800**

- Works

- Card=**5000**, sel(Eno='v')=**2%**

- Taille d'une page=**2000**

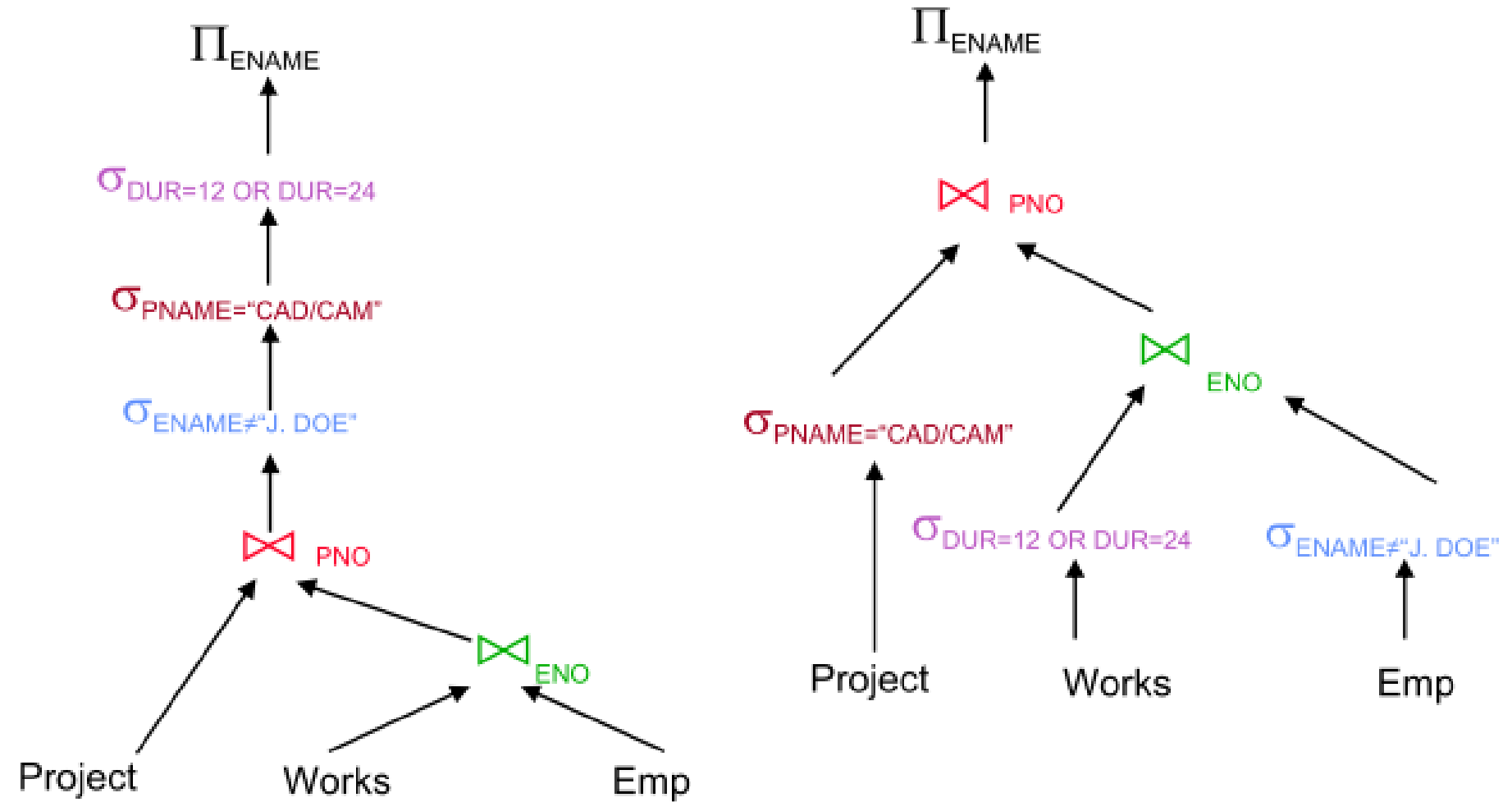
- Estimer les coût des plans d'exécution des requêtes suivantes:

- ✓ Noms de tous les projets
- ✓ Noms et budgets des projets de Paris?
- ✓ Noms des projets de budget > 225?
- ✓ Noms et budgets des projets où travaille l'employé E1?

Espace de recherche

- **Espace de recherche :**
 - ensemble des plans *équivalents* pour une **même** requête
 - peut être généré en appliquant des **règles de transformation**
- **Observations :**
 - Deux plans *équivalents* ont en général des **coûts différents**.
 - **L'ordre** des opérations est importante : opérations plus ou moins coûteuses et sélectives

Espace de recherche



Exemples de plans

Schéma de la base

EMPLOYE(Eid, Enom, Titre, Ville)

PROJET(Pid, Pnom, Budget, Ville)

TRAVAUX(#Eid, #Pid, Respid, Durée)

- Requête
 - Le nom et le titre des employés qui travaillent dans des projets avec un budget supérieur à 250

```
SELECT DISTINCT Enom, Titre  
FROM Employe E, Projet P, Travaux T  
WHERE P.Budget > 250  
AND E.Eid=T.Eid  
AND P.Pid=T.Pid ;
```

Exemples de plans

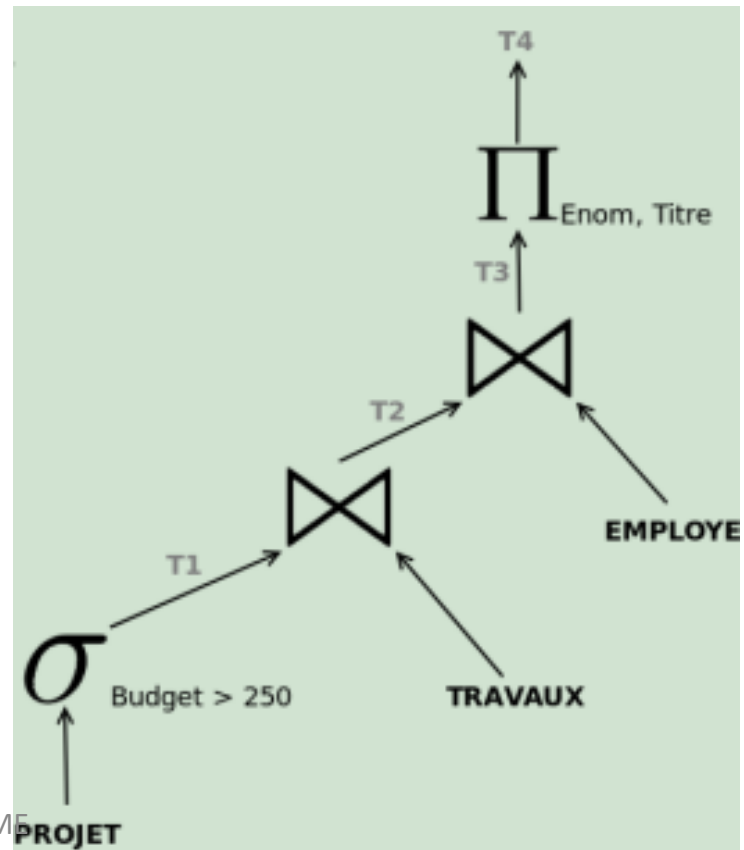
- Plan 1

$T_1 \leftarrow$ Lire la table projet et sélectionner les tuples de Budget > 250

$T_2 \leftarrow$ Joindre T_1 avec la relation TRAVAUX

$T_3 \leftarrow$ Joindre T_2 avec la relation EMPLOYE

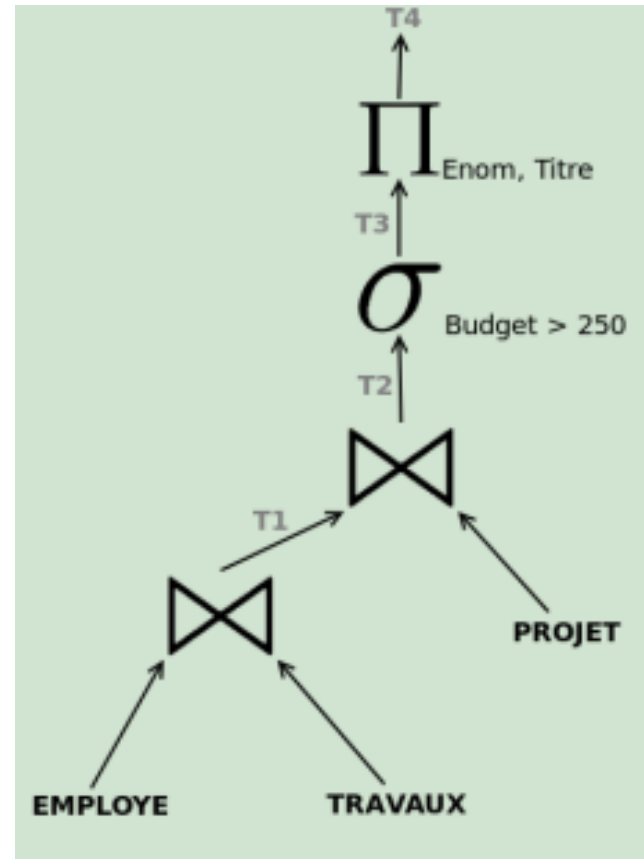
$T_4 \leftarrow$ Projeter T_3 sur Enom, Titre



Exemples de plans

• Plan 2

- $T1 \leftarrow$ Joindre la relation TRAVAUX avec la relation EMPLOYE
- $T2 \leftarrow$ Joindre $T1$ avec la relation PROJET
- $T3 \leftarrow$ Lire $T2$ et sélectionner les tuples de Budget > 250
- $T4 \leftarrow$ Projeter $T3$ sur Enom, Titre



Règles de transformation

- Commutativité des opérations binaires

- $R \times S \equiv S \times R$
- $R \bowtie S \equiv S \bowtie R$
- $R \cup S \equiv S \cup R$

- Associativité des opérations binaires

- $(R \times S) \times T \equiv R \times (S \times T)$
- $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$

- Idempotence des opérations unaires

- $\Pi_A(\Pi_A(R)) \equiv \Pi_A(R)$
- $\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) \equiv \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$

où $R[A]$ et $A' \subseteq A, A'' \subseteq A$ et $A' \subseteq A''$

Règles de transformation

Commutativité de la sélection et de la projection

- Commutativité de la sélection avec les opérations binaires

- $\sigma_{p(A)}(R \times S) \equiv (\sigma_{p(A)}(R)) \times S$
- $\sigma_{p(A_i)}(R \bowtie_{(A_j B_k)} S) \equiv (\sigma_{p(A_i)}(R)) \bowtie_{(A_j B_k)} S$
- $\sigma_{p(A_i)}(R \cup T) \equiv \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$

où A_i appartient à R et T

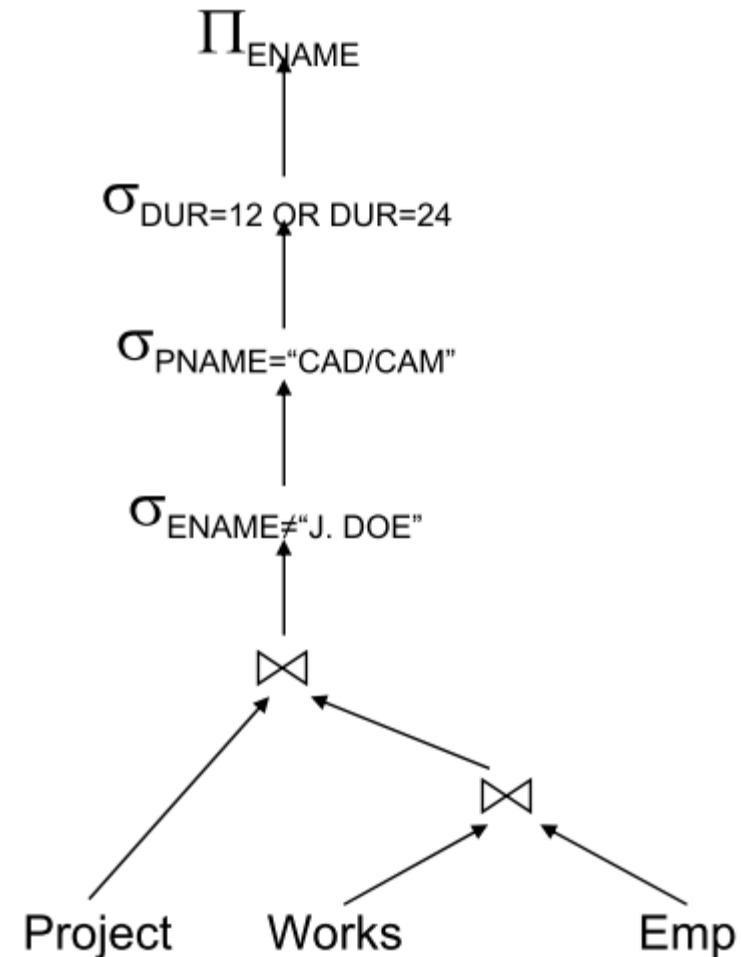
- Commutativité de la projection avec les opérations binaires

- $\Pi_C(R \times S) \equiv \Pi_{A'}(R) \times \Pi_{B'}(S)$
- $\Pi_C(R \bowtie_{(A_j B_k)} S) \equiv \Pi_{A'}(R) \bowtie_{(A_j B_k)} \Pi_{B'}(S)$
- $\Pi_C(R \cup S) \equiv \Pi_C(R) \cup \Pi_C(S)$

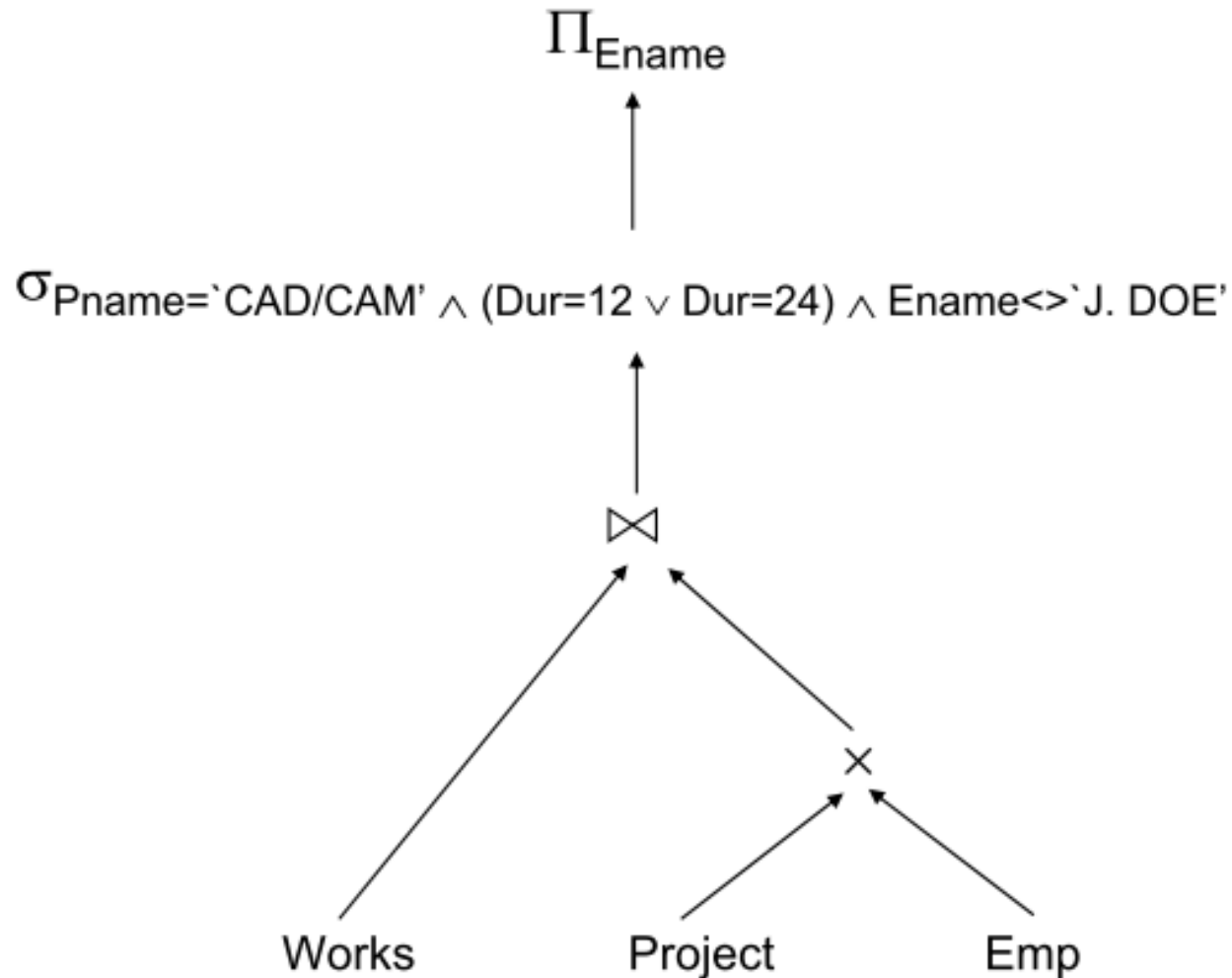
où $R[A]$ et $S[B]$; $C = A' \cup B'$ où $A' \subseteq A$, $B' \subseteq B$, $A_j \subseteq A'$, $B_k \subseteq B'$

Example

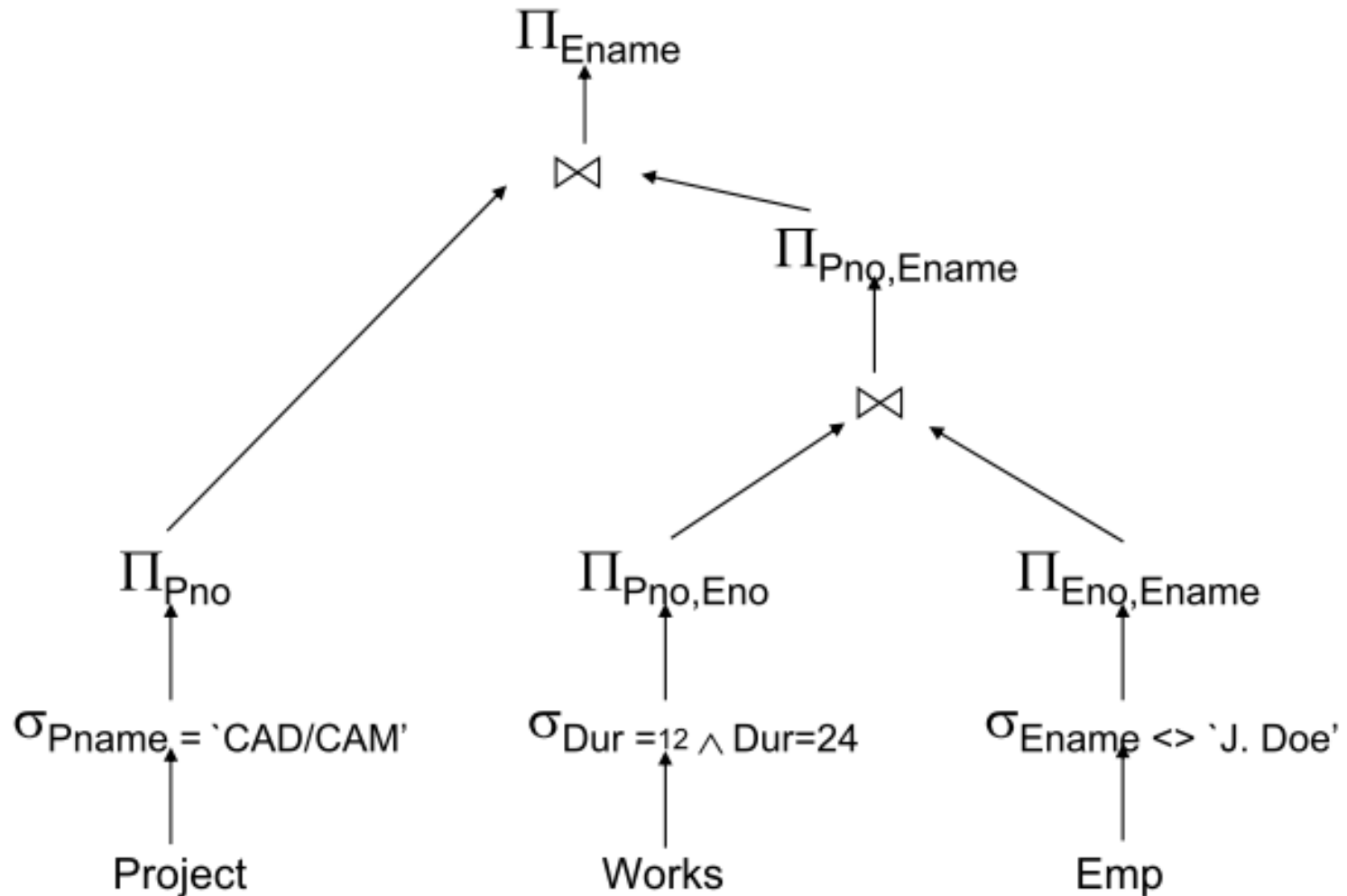
```
SELECT Ename
FROM Project p, Works w,
      Emp e
WHERE w.Eno=e.Eno
AND   w.Pno=p.Pno
AND   Ename<>`J. Doe`
AND   p.Pname=`CAD/CAM`
AND   (Dur=12 OR Dur=24)
```



Requête équivalente



Autre requête équivalente



Heuristique

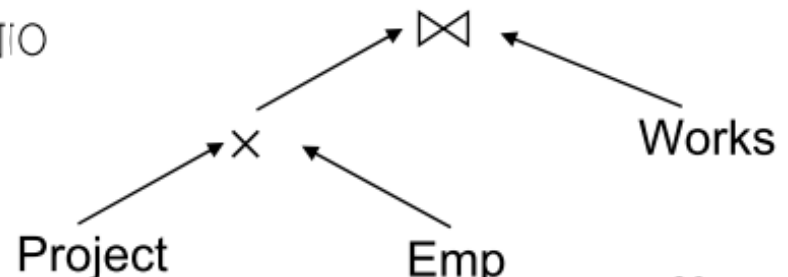
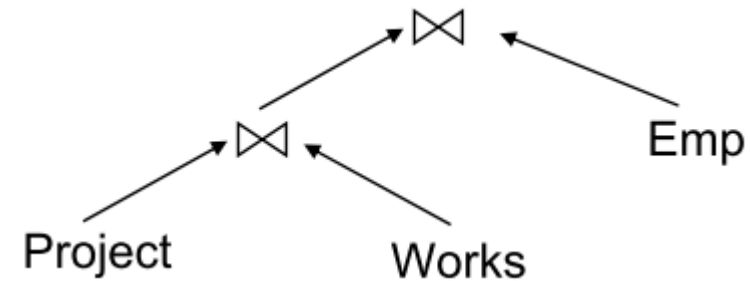
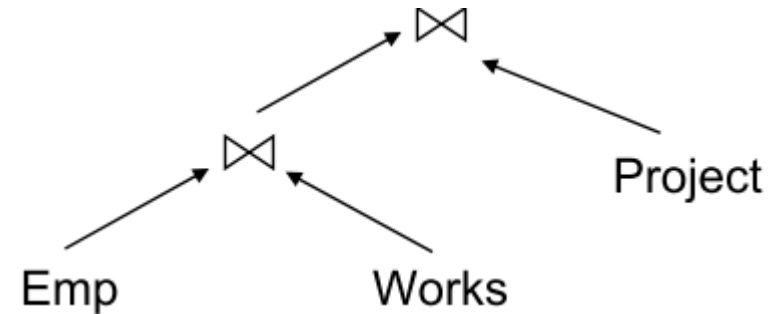
- *L'ordre des opérations* est importante : opérations plus ou moins coûteuses et sélectives
- **Idée :**
faire les opérateurs les **moins coûteux** (projection, sélection) et les plus sélectives en **premier**, de manière à **réduire la taille des données d'entrée** pour les opérateurs les **plus coûteux** (jointure)

Méthode heuristique :

descendre les sélections, puis les projections au maximum grâce aux règles de transformation

Arbres de jointures

Avec N relations, il y a $O(N!)$ arbres de jointures équivalents qui peuvent être obtenus en appliquant les règles de *commutativité* et *d'associativité*



SELECT Ename, Resp
FROM Emp, Works, Project
WHERE Emp.Eno=Works.Eno
AND Works.PNO=Project.PNO

Optimisation de requêtes SQL : index

Utilisation des index

- Créer les **index** **après l'insertion** des données
- Créer des **index** sur les **colonnes** qui servent lors de la **jointure**
- Créer des **index** pour les **colonnes fréquemment utilisées** dans la clause **WHERE**

Optimisation de requêtes SQL

Utilisation de *

- Utiliser les **noms de colonnes** au lieu de *
- Certains SGBD cachent les noms des colonnes retournées par un `SELECT * FROM T` => Si ajout d'une colonne dans T, cette colonne peut ne pas figurer lors de l'exécution de la requête à nouveau
- Perte de ressources (mémoire) lorsqu'on n'a pas besoin de lister toutes les colonnes (surtout dans le cas d'une jointure où les attributs en commun sont dupliqués)

Exemple

```
SELECT
    ETUDIANT.idEtudiant, nom, idMatiere, note
FROM
    ETUDIANT,
    NOTE
WHERE
    ETUDIANT.idEtudiant = NOTE.idEtudiant;
```

au lieu

```
SELECT
    *
FROM
    ETUDIANT,
    NOTE
WHERE
    ETUDIANT.idEtudiant = NOTE.idEtudiant;
```

	idEtudiant	nom	idMatiere	note
►	1	Samantha	1	12
	1	Samantha	4	9
	1	Samantha	7	9
	2	Francis	1	20
	3	Charles	1	13
	4	Penelope	1	14
	5	Craig	1	10
	6	Steve	1	9

	idEtudiant	nom	idEtudiant	idMatiere	note
►	1	Samantha	1	1	12
	1	Samantha	1	4	9
	1	Samantha	1	7	9
	2	Francis	2	1	20
	3	Charles	3	1	13
	4	Penelope	4	1	14
	5	Craig	5	1	10
	6	Steve	6	1	9

duplication de la colonne idEtudiant

Optimisation de requêtes SQL

Utilisation de HAVING

- **HAVING** est utilisée pour créer un filtre **sur les colonnes sélectionnées**
- **WHERE** crée un filtre **avant de sélectionner** les colonnes
=> **plus rapide**
- **Ne pas utiliser HAVING avec des conditions qui peuvent être utilisées dans le WHERE**

```
SELECT
    *
FROM
    NOTE
WHERE
    idMatiere = 1
;
-- au lieu de

SELECT
    *
FROM
    NOTE
HAVING idMatiere = 1;
```


Optimisation de requêtes SQL

Minimiser les sous-requêtes

- Chaque requête est un accès à la BD
- **Minimiser le nombre de sous requêtes**, si possible
- **Convertir les sous-requêtes en jointures**, si possible

Exemple

- Conversion des sous-requêtes en jointures

```
-- Utiliser  
SELECT  
    nom  
FROM  
    ETUDIANT, NOTE  
WHERE  
    NOTE.idEtudiant = ETUDIANT.idEtudiant;
```

```
-- au lieu de
```

```
SELECT  
    nom  
FROM  
    ETUDIANT  
WHERE  
    idEtudiant IN (SELECT  
                    idEtudiant  
                    FROM  
                    NOTE);
```

Optimisation de requêtes SQL

EXISTS and IN

- EXISTS est plus rapide que IN si le résultat de la sous-requête est large
- IN est plus rapide que EXISTS si le résultat de la sous-requête est petit
- **IN** est souvent moins performant que **EXISTS**

Optimisation de requêtes SQL






UNION vs UNION ALL

- **Union** élimine les lignes dupliquées alors que UNION ALL rend toutes les lignes, même celles dupliquées
- **Utiliser UNION ALL en unifiant des lignes non dupliquées**, ou quand il n'y a pas besoin d'éliminer les lignes dupliquées

```
1 • SELECT idEtudiant FROM ETUDIANT
2 UNION
3 SELECT idEtudiant FROM NOTE
```

160% 6:2

Filter:






Edit:    File:  

idEtudiant
1
2
3
4
5
6
7

```
1 • SELECT idEtudiant FROM ETUDIANT
2 UNION ALL
3 SELECT idEtudiant FROM NOTE
```

160% 28:3

Filter:

Edit:    File:  

idEtudiant
1
2
3
4
5
6
7
1
2
3
4
5
6
1
1