

BDay-MI

Bases de données avancées

Cours de Cristina Sirangelo

IRIF, Université Paris Diderot

Assuré en 2021-2022 par Amélie Gheerbrant

amelie@irif.fr

Évaluation et optimisation de requêtes

Sources (quelques slides empruntés et réadaptés) :

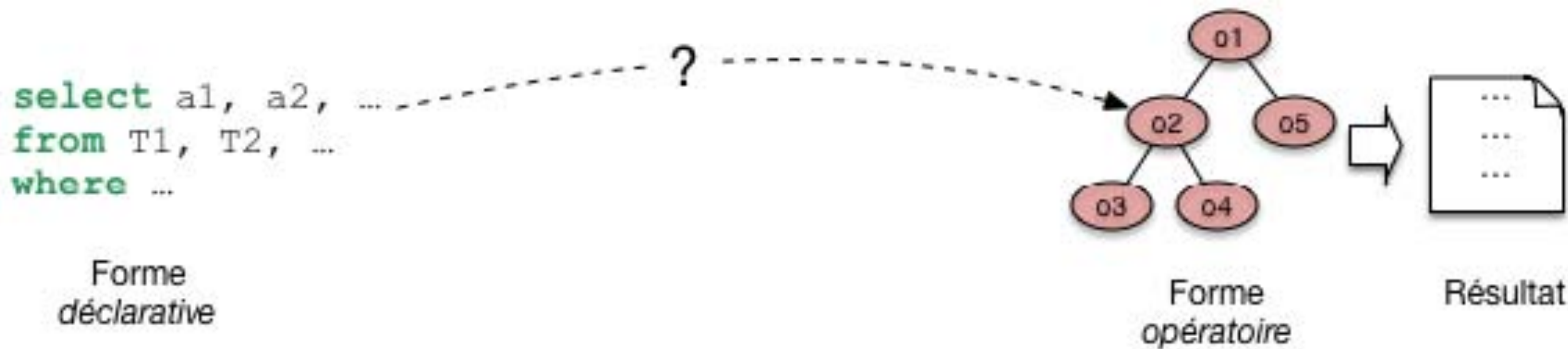
- MOOC DB, Philippe Rigaux, CNAM
- Cours Database systems principles - H.G. Molina, Stanford Univ.
- Slides du livre Database systems concepts -
A. Silberschatz, Yale U. & H. Korth, Lehigh U. & S.Sudarshan, IIT Bombay

Requêtes et plans d'exécutions



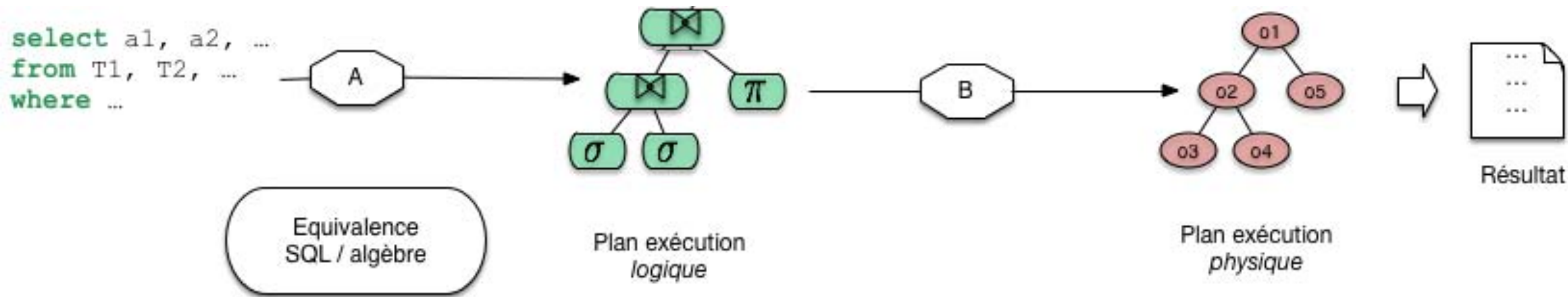
- Une requête SQL est **déclarative**. Elle ne dit pas comment calculer le résultat.
- Nous avons besoin d'une **forme opératoire** : un programme.

Requêtes et plans d'exécutions



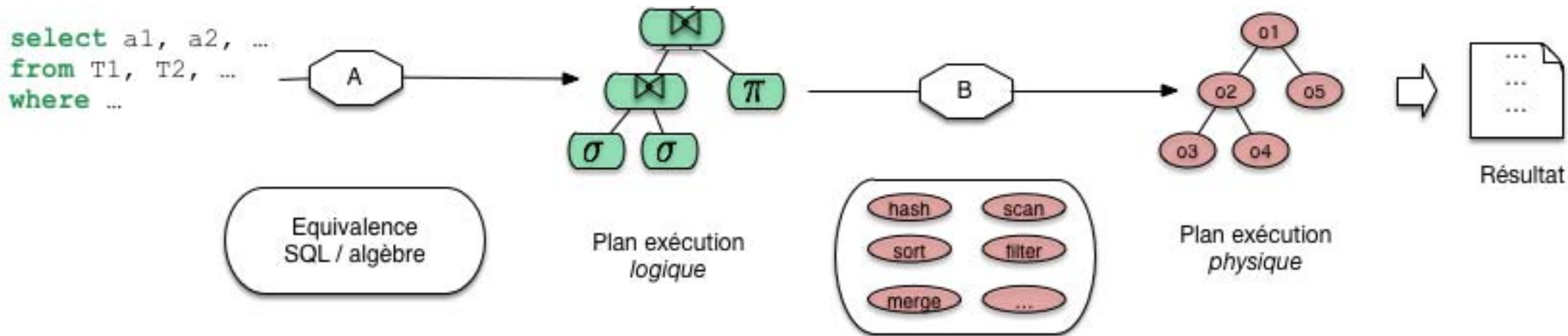
- Dans un SGBD le programme qui exécute une requête est appelé *plan d'exécution*.
- Il a une forme particulière : c'est un arbre, constitué d'opérateurs

De la requête au plan d'exécution



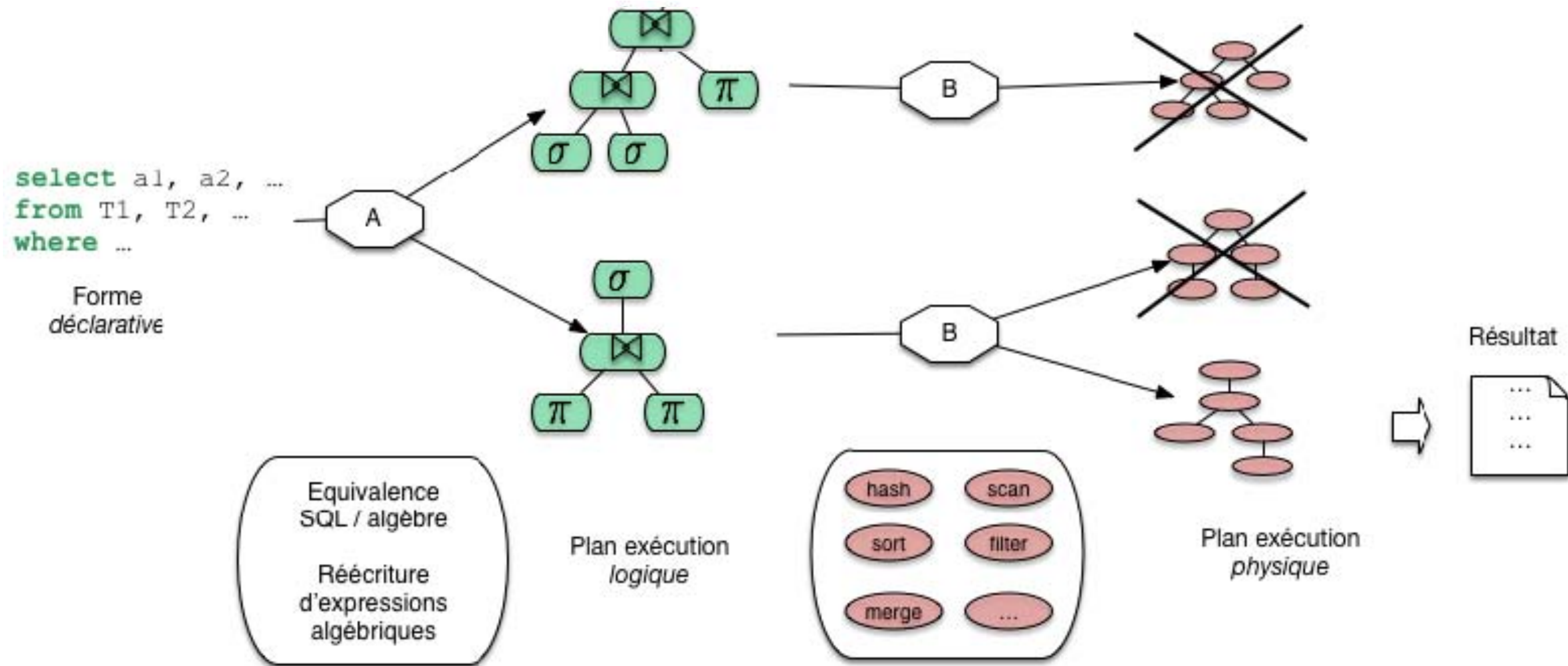
- Deux étapes :
 - (A) plan d'exécution logique (l'algèbre) ;
 - (B) plan d'exécution physique (opérateurs).

De la requête au plan d'exécution



- **Opérateurs** d'un plan d'exécution physique :
 - ▶ des algorithmes spécifiques pour implémenter certains opérateurs algébriques
 - *hash-join, merge-join, ...*
 - ▶ des opérateurs plus élémentaires, auxiliaires à l'implémentation des opérateurs algébriques
 - *sort, linear scan, index scan, filter, ...*

Optimisation de requêtes : une vision d'ensemble



- À chaque étape, plusieurs choix. Le système les évalue et choisit le “meilleur”

Optimisation de requêtes : une vision d'ensemble

- Un trop grand nombre de plans d'exécution possibles!
 - ▶ pas envisageable d'évaluer chacun
 - ▶ techniques pour “couper” des branches de l'arbre de plans possibles
 - techniques exactes (programmation dynamique)
 - heuristiques
 - combinaison des deux

Évaluation d'un plan d'exécution : une vision d'ensemble

- Évaluation d'un plan : une mesure du coût de son exécution sur la BD
 - ▶ meilleur \Leftrightarrow moins coûteux
- Évaluation d'un plan physique : sans exécution!
 - ▶ Notion de coût fondée sur :
 - fonction de coût des algorithmes choisis
 - statistiques sur la taille des relations / fréquence des valeurs
 - stockées dans le catalogue de la BD
- Évaluation possible déjà au niveau des plans logiques
 - ▶ Motivation : couper “tôt” une branche de l'arbre des plans
 - ▶ Notion de coût fondée sur la taille des résultats intermédiaires
- Il s'agit d'estimations de coût : souvent suffisant comme critère d'évaluation

Plan

- Introduction
- Plans d'exécution logiques
- Plans d'exécution physiques
 - ▶ Implémentations des opérateurs algébriques
 - ▶ Implémentation des expressions algébriques
- Evaluation des plans d'exécution
- Optimisation

Plans d'exécutions logiques

- Une requête SQL est traduite dans une expression de l'algèbre appelée *plan d'exécution logique*
- **Exemple** : le nom des enseignants du département d'informatique avec l'intitulé des cours qu'ils enseignent

SELECT nom, intitulé

FROM Enseignant NATURAL JOIN Enseigne NATURAL JOIN Cours

WHERE dpt = 'Info'



$\Pi_{\text{nom, intitulé}} (\sigma_{\text{dpt} = \text{'Info'}} (\text{Enseignant} \bowtie (\text{Enseigne} \bowtie (\text{Cours}))))$

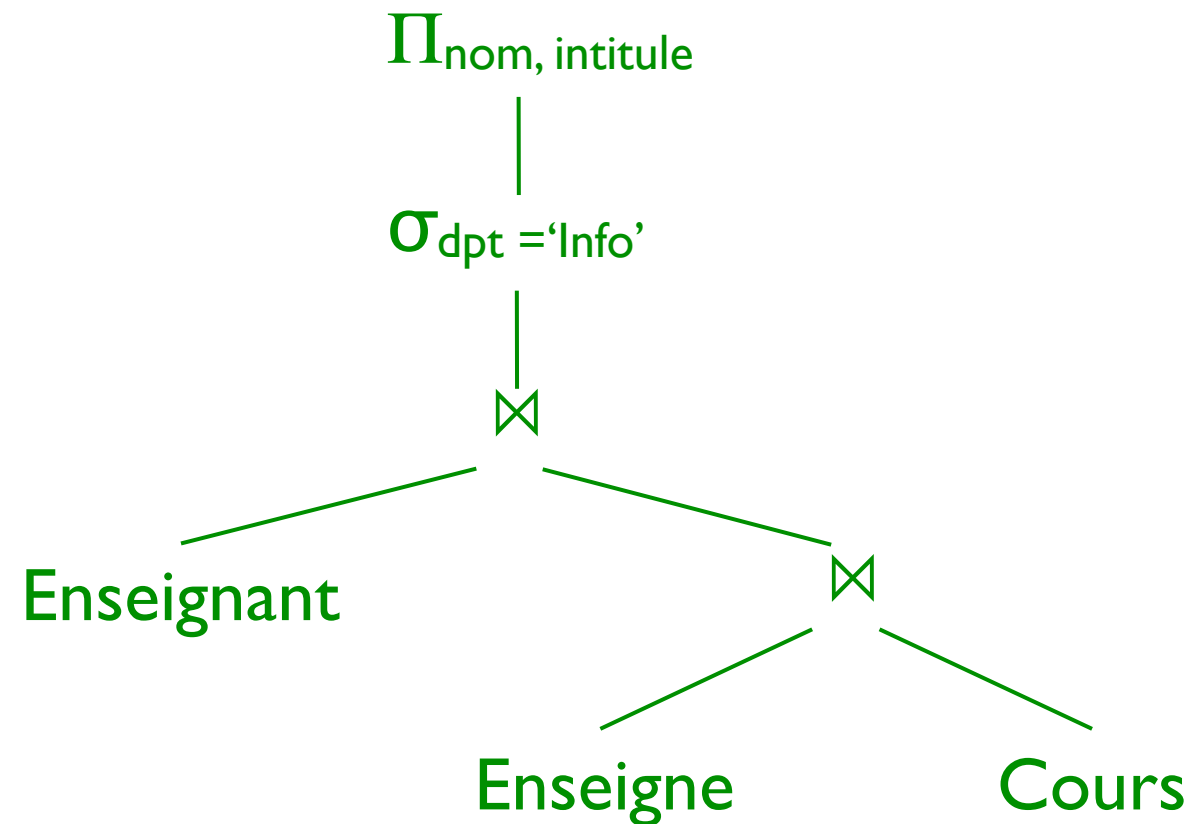
- Langage pour les expressions algébrique : **algèbre relationnelle étendue**
(multi-ensembles, élimination des doublons, ordre, agrégats et regroupement, ...)

Plans d'exécutions logiques : structure d'arbre

- On voit un plan d'exécution logique comme un arbre : l'arbre de “*parsing*” de l'expression

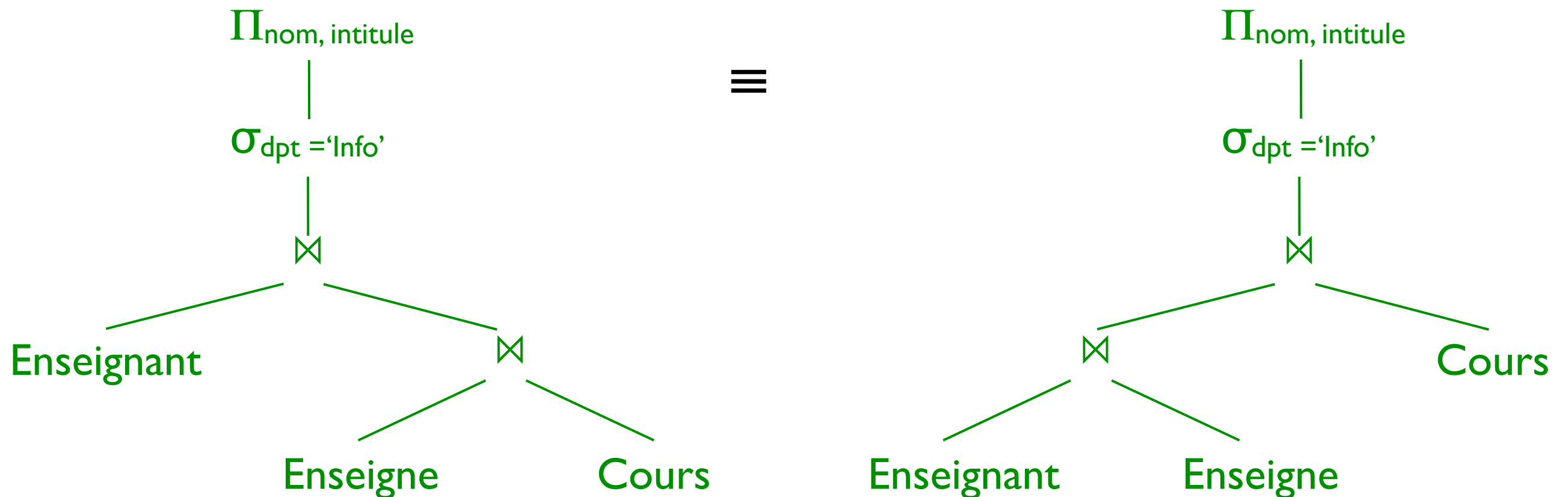
$\Pi_{\text{nom, intitulé}} (\sigma_{\text{dpt} = \text{'Info'}} (\text{Enseignant} \bowtie (\text{Enseigne} \bowtie (\text{Cours}))))$

\equiv



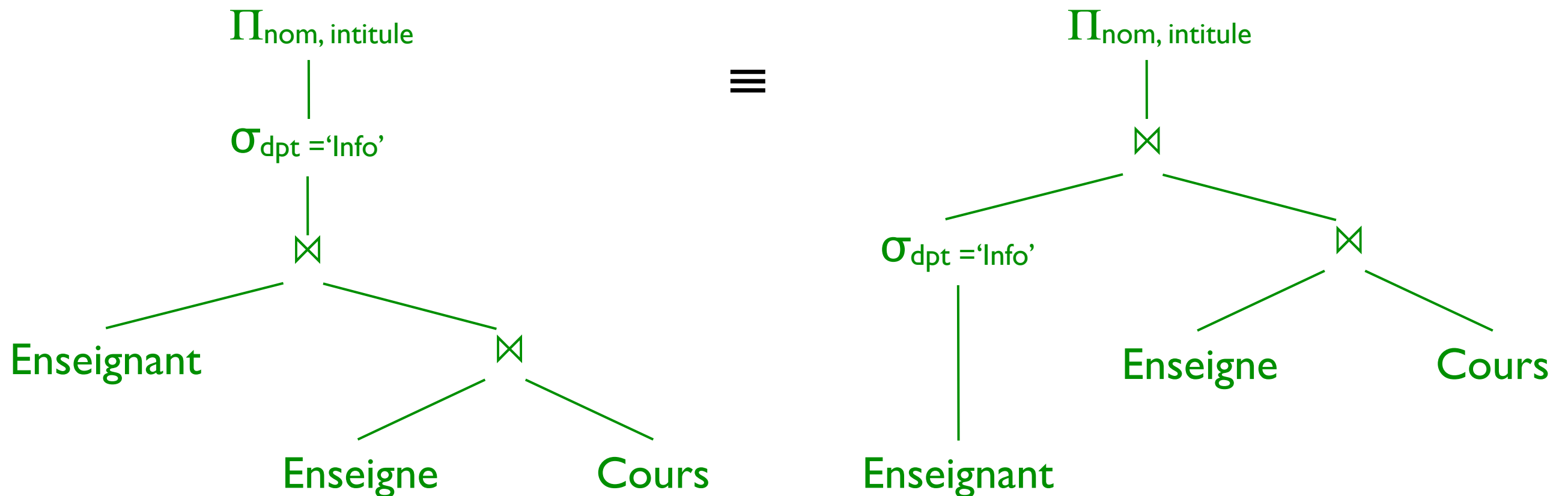
Plans d'exécutions logiques : équivalence

- Plusieurs plans d'exécution possibles pour la même requête SQL
 - ▶ deux plans d'exécution sont **équivalents** s'ils définissent la même requête (i.e. pour tout BD d'input il produisent le même output)



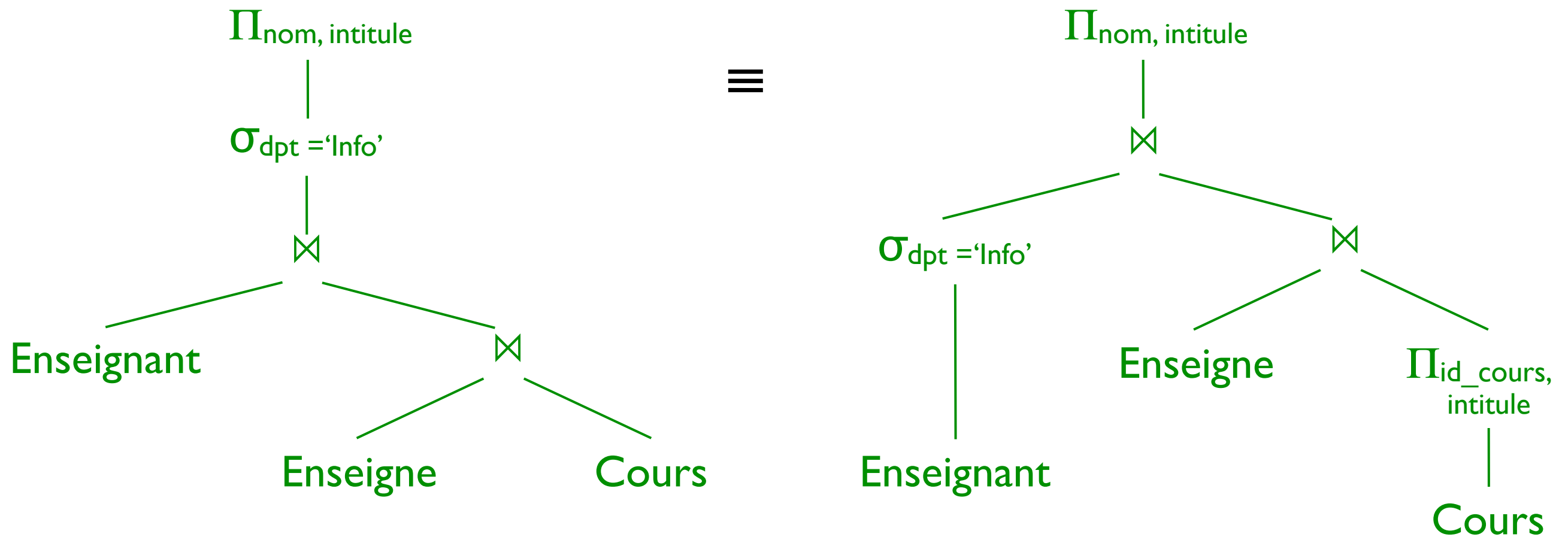
Plans d'exécutions logiques : équivalence

- Plusieurs plans d'exécution possibles pour la même requête SQL
 - ▶ deux plans d'exécution sont **équivalents** s'ils définissent la même requête (i.e. pour tout BD d'input il produisent le même output)



Plans d'exécutions logiques : équivalence

- Plusieurs plans d'exécution possibles pour la même requête SQL
 - ▶ deux plans d'exécution sont **équivalents** s'ils définissent la même requête (i.e. pour tout BD d'input il produisent le même output)

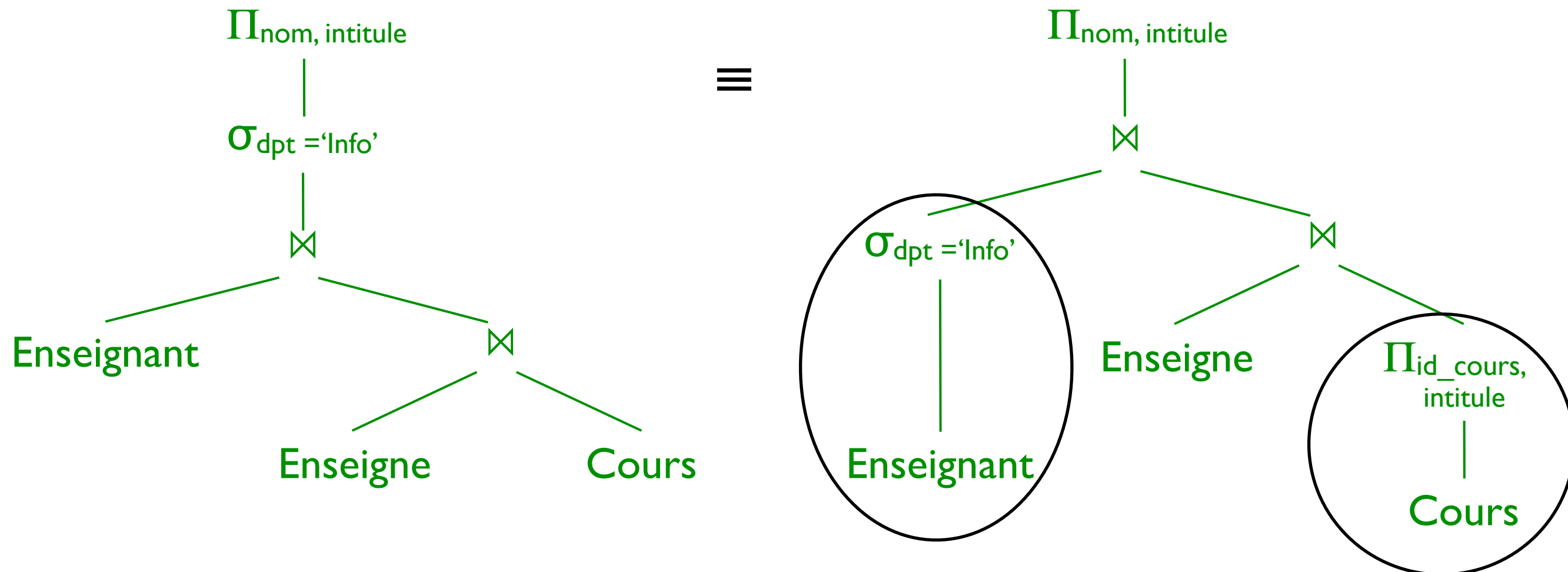


Pourquoi considérer plusieurs plans logiques possibles?

- Certains plans logiques sont plus susceptibles que d'autres de générer des plans d'exécutions physiques efficaces

Exemple

Plus efficace
que le plan de gauche



sélections et projections appliquées dès que possible réduisent la taille des relations sur lesquelles on fait des jointures

Règles de réécriture algébrique

- Un certain nombre de règles peuvent être appliquées pour réécrire un plan logique en un autre plan logique équivalent
 - ▶ But : énumérer tous (ou un nombre suffisant) de plans logiques alternatifs, par applications successives des règles
- Quelques unes des règles les plus importantes :

1. $R \bowtie S \equiv S \bowtie R$

2. $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$

3. $\sigma_{\alpha} (R \bowtie S) \equiv \sigma_{\alpha} (R) \bowtie S$ si α concerne uniquement des attributs de R

4. $\Pi_{AR,AS} (R \bowtie S) \equiv \Pi_{AR,AS} (\Pi_{AR,AJ} (R) \bowtie \Pi_{AS,AJ} (S))$ AR (AS) : attributs de R (S)
AJ : attributs de jointure

5. $\sigma_{\Theta_1 \wedge \Theta_2} (R) \equiv \sigma_{\Theta_1} (\sigma_{\Theta_2} (R))$

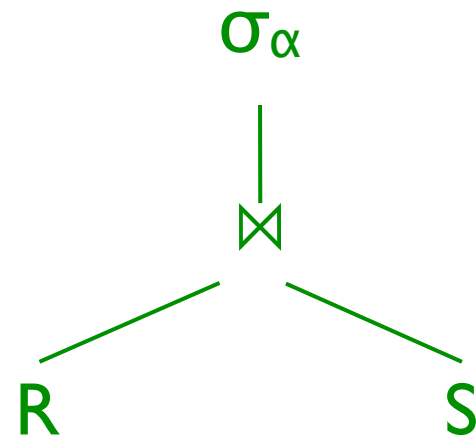
6. $\sigma_{\alpha} (R \times S) \equiv R \bowtie_{\alpha} S$

7. $\sigma_{\alpha} (R \bowtie_{\Theta} S) \equiv R \bowtie_{\alpha \wedge \Theta} S$

et bien d'autres....

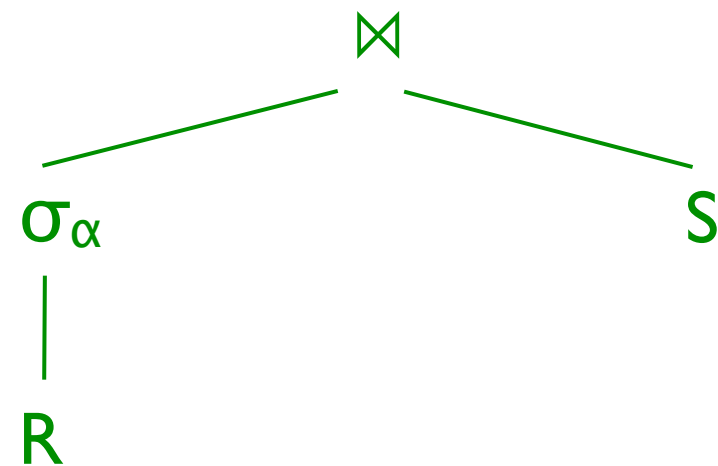
Règles de réécriture vues sur les arbres

- **Exemple** : règle 3

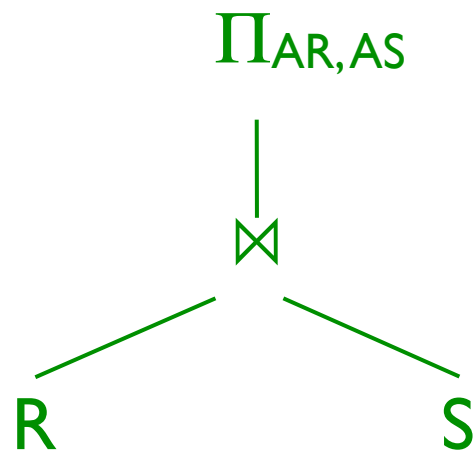


si α sur R

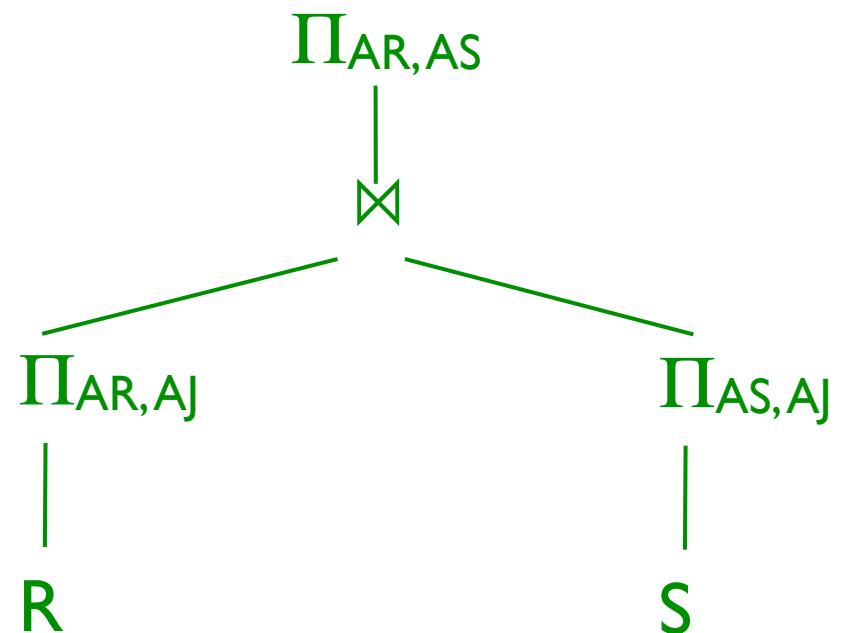
\equiv



- **Exemple** : règle 4



\equiv



- L'équivalence de deux expressions ne dit pas laquelle est la meilleure (dans les deux cas ci-dessus en générale celle de droite, mais pas toujours...)
- c'est le rôle de l'optimisation de faire les bons choix de réécriture (cf. plus loin)

Plan

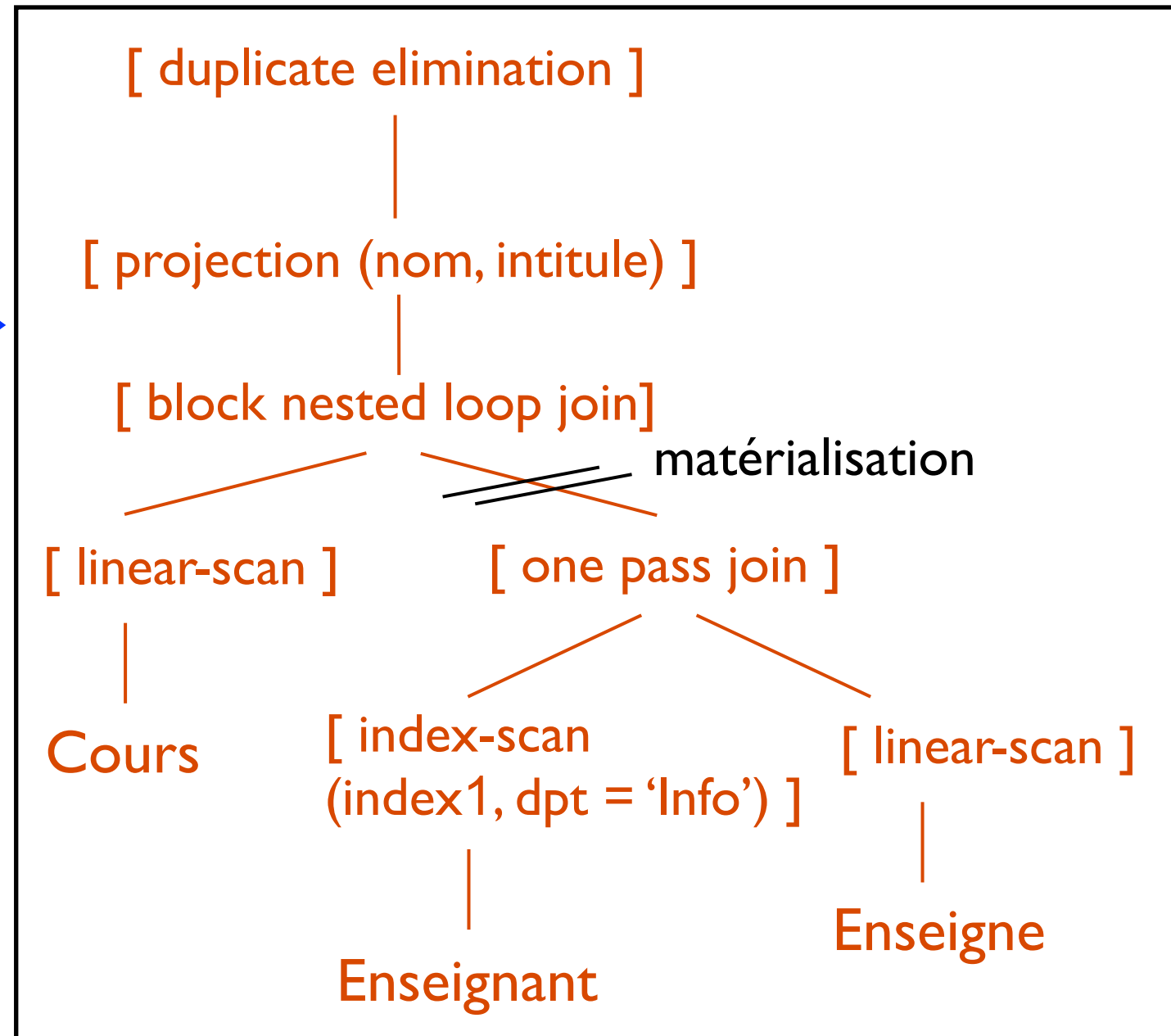
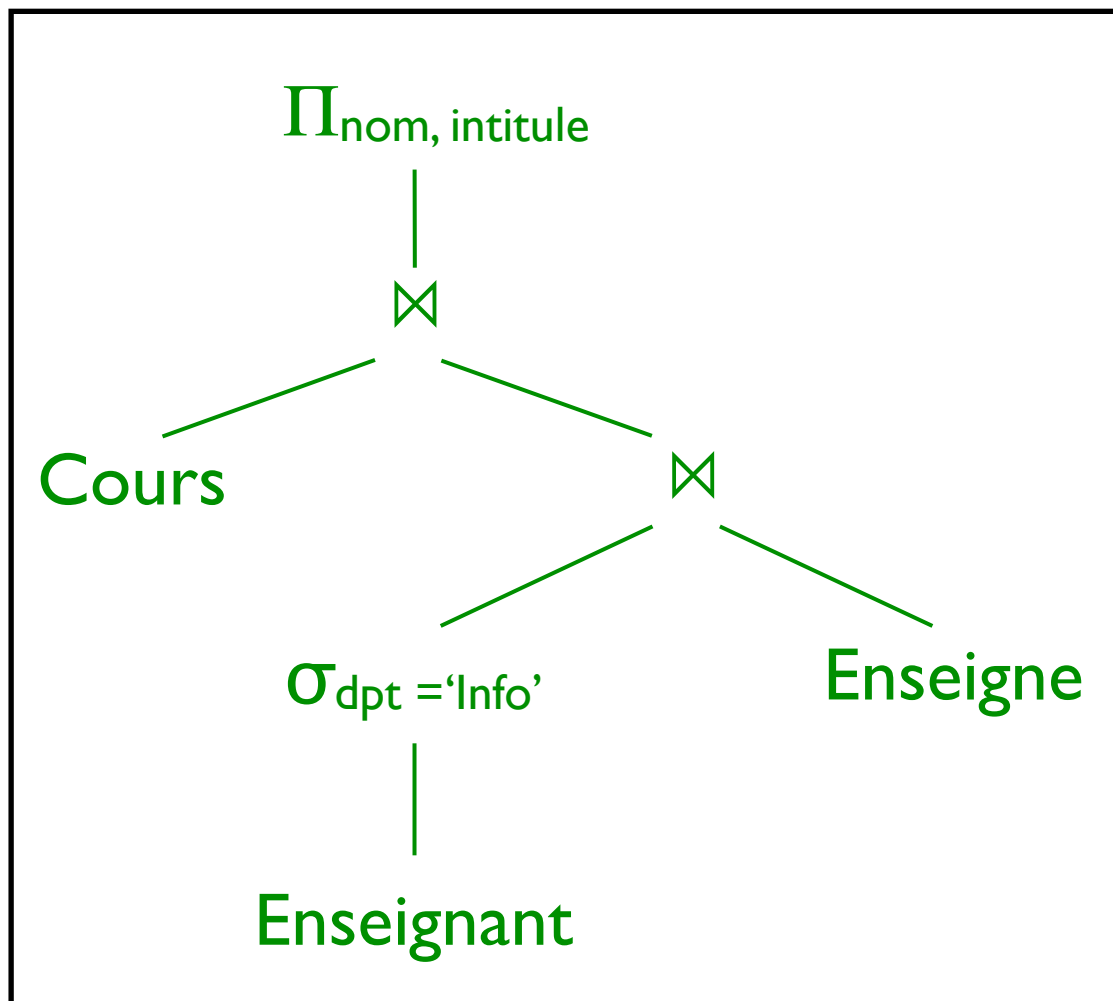
- Introduction
- Plans d'exécution logiques
- Plans d'exécution physiques
 - ▶ Implémentations des opérateurs algébriques
 - ▶ Implémentation des expressions algébriques
- Evaluation des plans d'exécution
- Optimisation

Plans d'exécution physiques

- Un plan d'exécution physique est obtenu d'un plan logique en choisissant :
 - ▶ les algorithmes pour chaque opérateur algébrique
 - ▶ des opérateurs physiques auxiliaires (*sort, filter, duplicate elimination ...*)
 - ▶ la façon des différents opérateurs d'une expression d'échanger les données

Plans d'exécution physiques

- ▶ Exemple de plan physique obtenu d'un plan logique

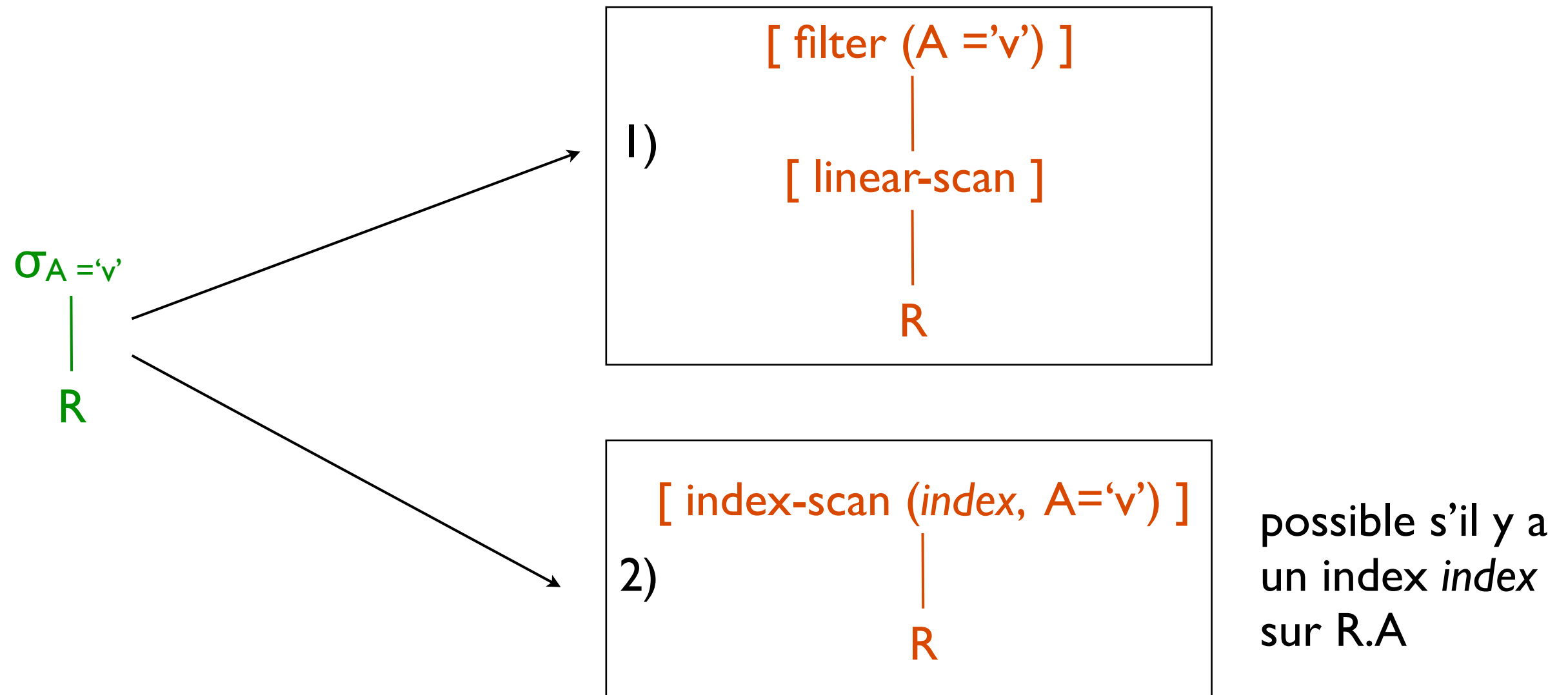


- ▶ Chaque SGBD a son ensemble d'opérateurs physiques et sa syntaxe pour les représenter

Plan

- Introduction
- Plans d'exécution logiques
- Plans d'exécution physiques
 - ▶ Implémentations des opérateurs algébriques
 - ▶ Implémentation des expressions algébriques
- Evaluation des plans d'exécution
- Optimisation

Sélection avec condition d'égalité



1. Parcourir tous les tuples de R (*linear scan*); sur chacun évaluer la condition; retenir celles qui la satisfont (*filter*)
2. Effectuer une recherche de la clef 'v' dans l'index $R.A$; accéder aux tuples de R référencés dans l'index

Sélection avec condition d'égalité

Coût des implémentations

(mesure de coût : nombre d'accès aux blocs du disque)

- *linear scan* : B_R
- *index scan* :
 - ▶ index primaire : $h_I + B_v$
beaucoup plus efficace que
linear scan

B_R : nombre de blocs de R
 h_I : hauteur de l'index
 B_v : nombre de blocs de R contenant les
tuples de clef 'v'
 n_v : nombre de tuples de R de clef 'v'

- ▶ index secondaire : $h_I + n_v$
coût potentiellement plus élevé que avec *linear scan*
(le même bloc peut devoir être chargé plusieurs fois)

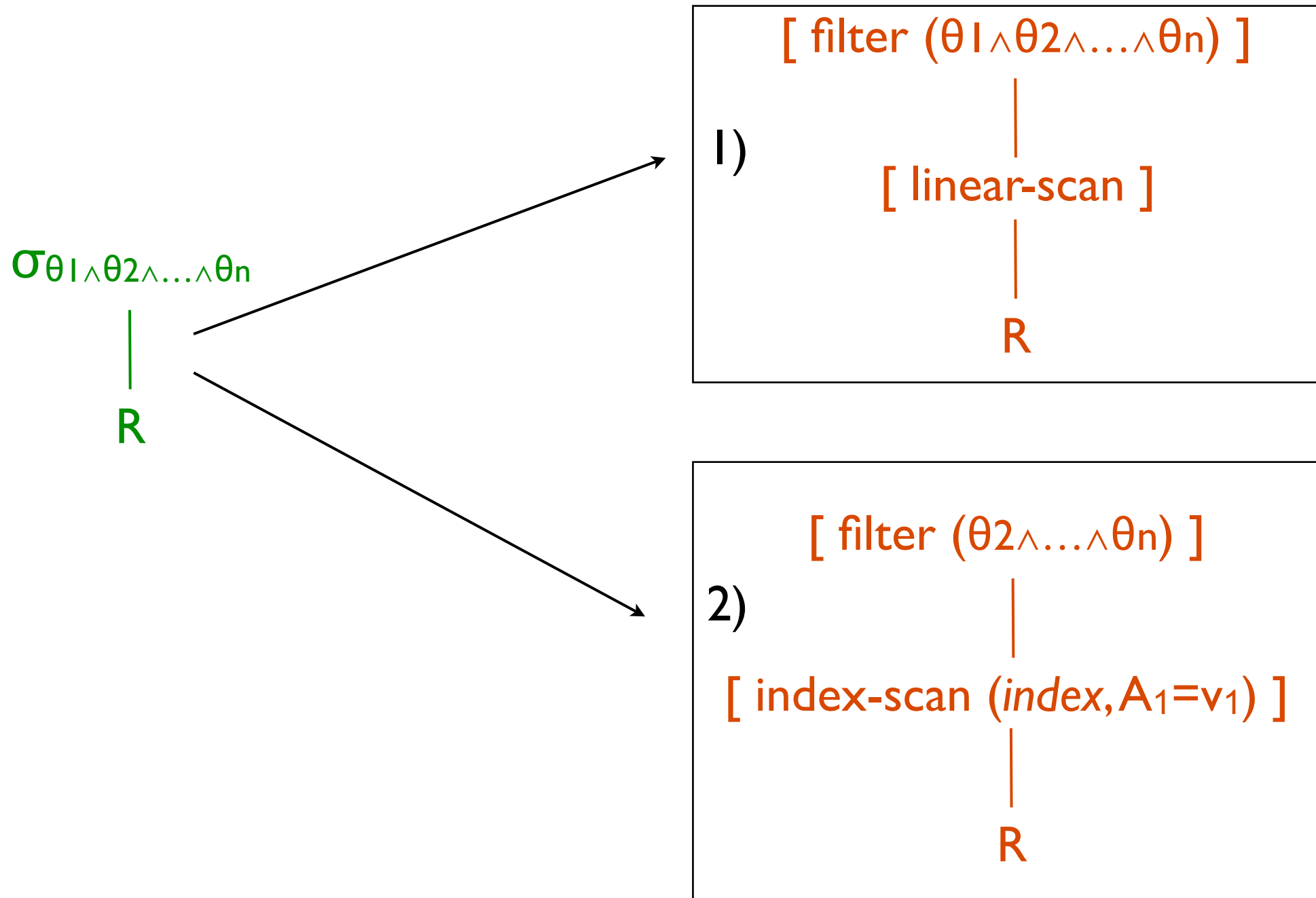
Sélection avec une condition d'inégalité ($A > v$)

- Similaire au cas de condition d'égalité
 - ▶ *index scan* possible s'il y a un *index B⁺-tree* sur R.A :
 - Effectuer une recherche de la clef v , et ensuite toutes les clefs $> v$ dans le B⁺-tree R.A;
 - accéder aux tuples de R référencés dans l'index

Sélection avec condition conjonctive

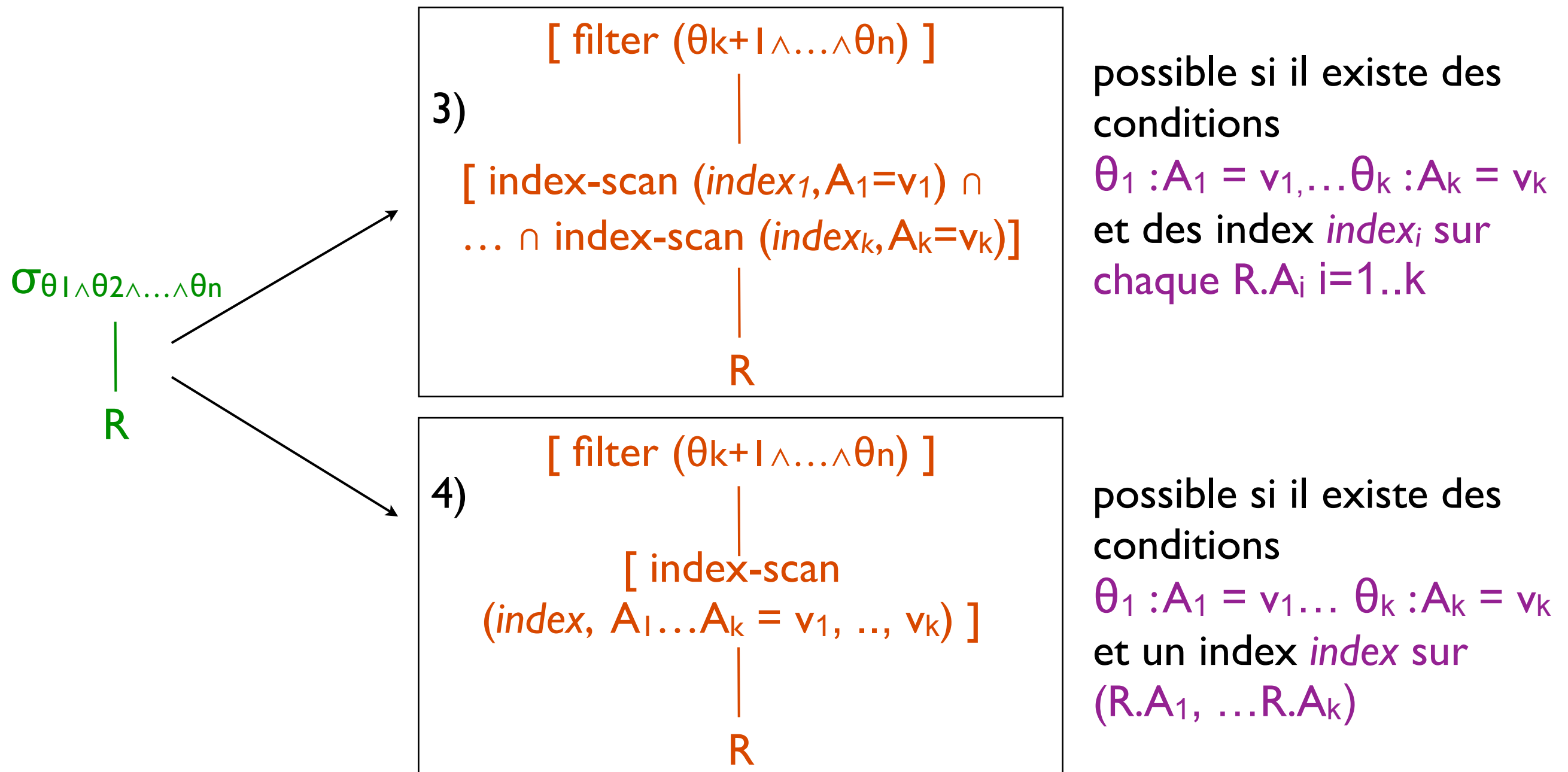
Plusieurs choix possibles.

- L'optimiseur évalue le coût de chacune et choisit la meilleure



possible si il existe une condition $\theta_1 : A_1 = v_1$ et un index *index* sur $R.A_1$

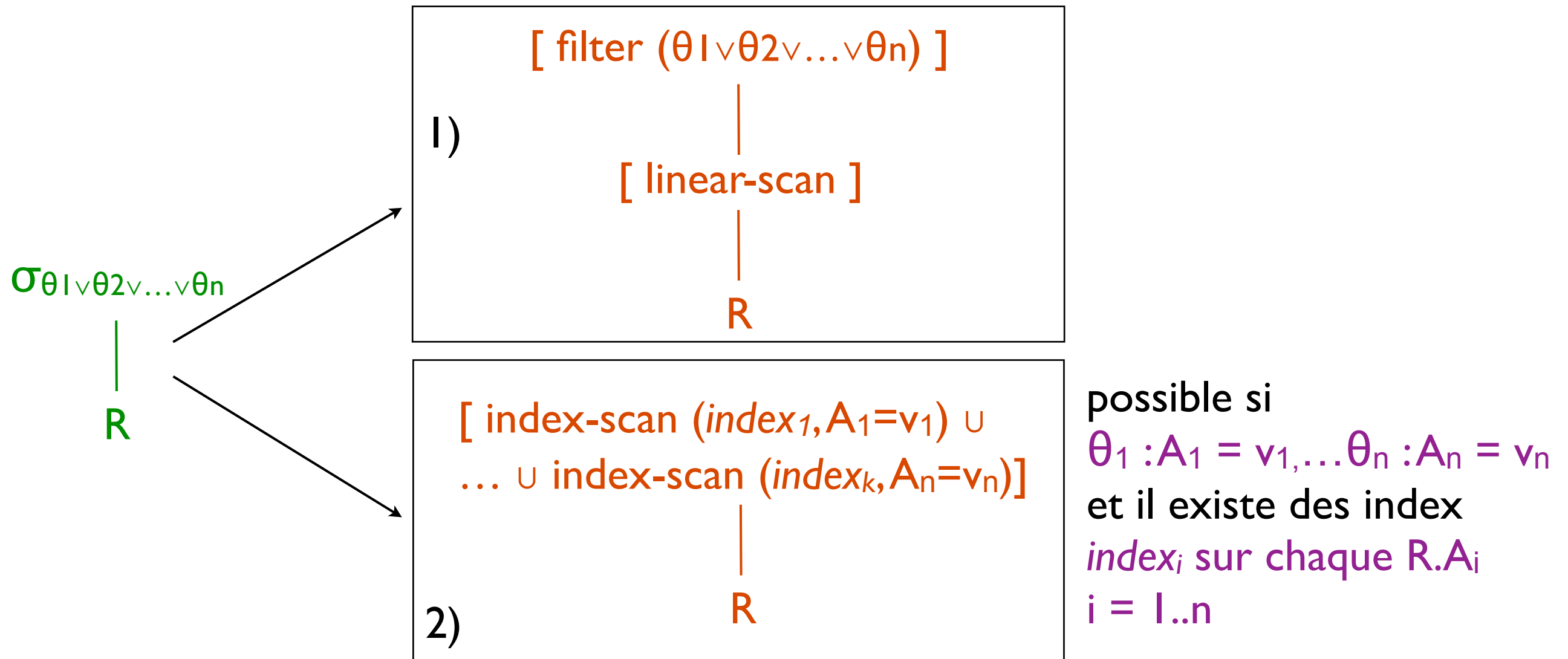
Sélections avec condition conjonctive



3) fait l'intersection au niveau des pointeurs à n-uplets, avant de récupérer les n-uplets dans le fichier de données

4) également possible s'il existe une condition $A_i = A_j$ où A_i, A_j font partie d'un index

Sélections avec condition disjonctive



2) fait l'union au niveau des pointeurs à n-uplets, avant de récupérer les n-uplets dans le fichier de données.

Si au moins un des A_i n'a pas d'index associé, un parcours linéaire serait nécessaire sur R pour tester $A_i = v_i \Rightarrow$ solution 1. plus efficace

Opérateurs auxiliaires : tri et hachage d'une relation

- L'implémentation de beaucoup d'autres opérateurs algébriques nécessite de techniques pour “regrouper” les tuples d'une table ayant la même valeur d'un (ou plusieurs) attribut(s)
 - ▶ Exemples : jointure, projection (élimination des doublons), group by, ...
- Deux techniques principales :
 - ▶ tri
 - ▶ hachage

Tri d'un fichier de données

- **But** : produire les tuples d'un fichier de données triés par la valeur d'un ou plusieurs attributs
- **Tri en mémoire externe**
(fichier de données en general trop grand par rapport à la mémoire principale)
 - ▶ différent par rapport aux algos de de tri d'un tableau en memoire principale :
 - accès au fichier de données **un certain nombre de blocs à la fois**
 - typiquement **accès séquentiel** au fichier pour plus d'efficacité
 - utilisation d'un buffer de **M blocs en mémoire interne**
 - **Variantes du tri fusion** (la fusion peut être réalisée efficacement en accès séquentiel)

Tri en memoire externe

- Principe:

- ▶ Charger dans le buffer en mémoire interne des “segments” du fichier à trier
- ▶ Trier les segments en mémoire interne et les recopier en mémoire externe
- ▶ “Fusionner” les segments triés en mémoire externe k à k en créant des segments triés plus gros
- ▶ Réitérer les fusions jusqu’à obtenir un seul segment trié

- Deux phases

- ▶ 1. Production des segments triés
- ▶ 2. Fusion des segments triés

Production des segments triés

BUFFER (M=3 blocs) mémoire principale

22	5	12	2	18	7

tri

lecture de 3 blocs

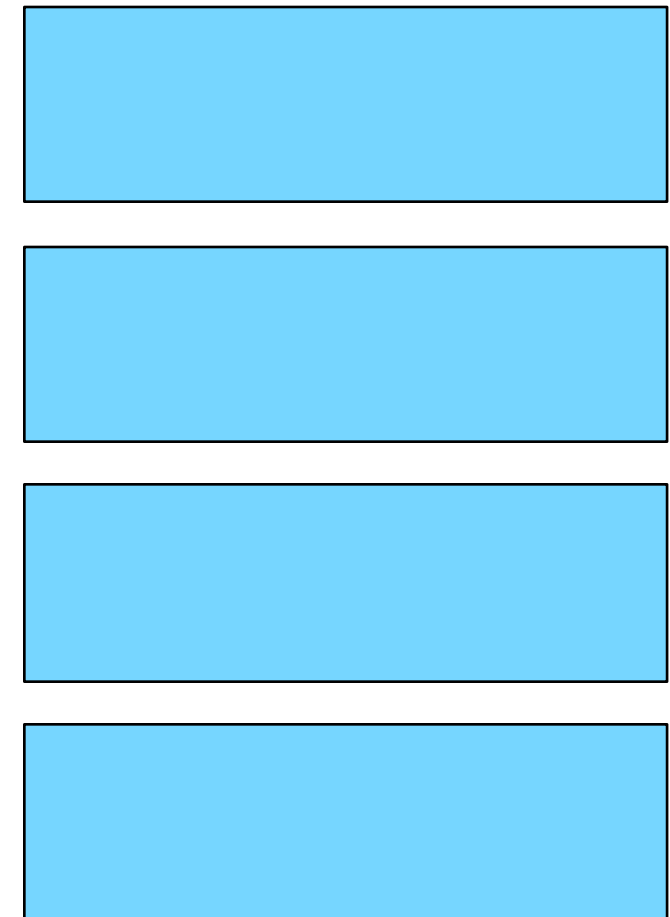
Fichier de données (mémoire externe)

															...
22	5	12	2	18	7	9	3	20	14	13	23	10	6	8	



Tant qu'il reste des blocs à lire dans le fichier de données

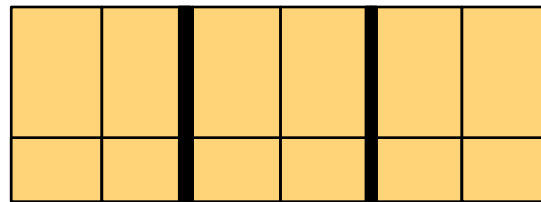
- ▶ lire les M prochains blocs du fichier de données dans le buffer (ou ceux qui restent, si moins que M)
- ▶ trier les tuples dans le buffer en mémoire principale
- ▶ écrire le segment trié de M blocs en mémoire externe



segments triés
(mémoire externe)

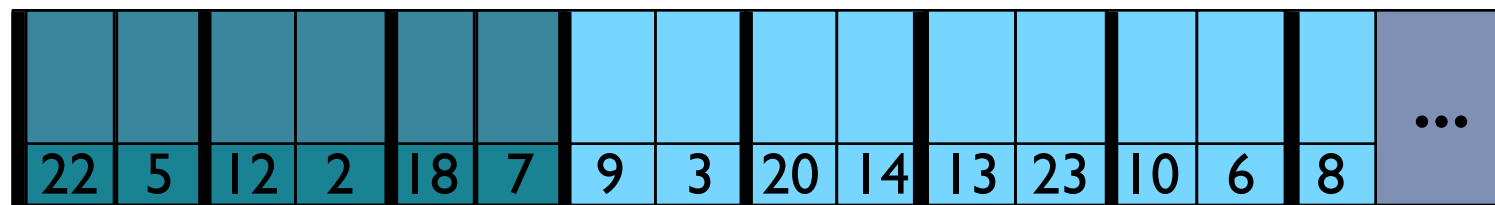
Production des segments triés

BUFFER (M=3 blocs) mémoire principale



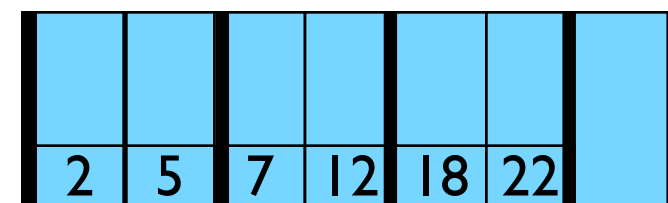
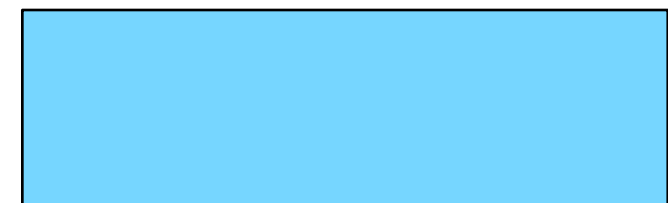
écriture de 3 blocs

Fichier de données (mémoire externe)



Tant qu'il reste des blocs à lire dans le fichier de données

- ▶ lire les M prochains blocs du fichier de données dans le buffer (ou ceux qui restent, si moins que M)
- ▶ trier les tuples dans le buffer en mémoire principale
- ▶ écrire le segment trié de M blocs en mémoire externe



segments triées
(mémoire externe)

Production des segments triés

BUFFER (M=3 blocs) mémoire principale

lecture de 3 blocs

9	3	20	14	13	23

tri

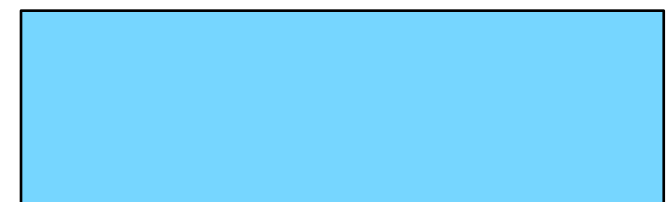
Fichier de données (mémoire externe)

22	5	12	2	18	7	9	3	20	14	13	23	10	6	8	...
----	---	----	---	----	---	---	---	----	----	----	----	----	---	---	-----



Tant qu'il reste des blocs à lire dans le fichier de données

- ▶ lire les M prochains blocs du fichier de données dans le buffer (ou ceux qui restent, si moins que M)
- ▶ trier les tuples dans le buffer en mémoire principale
- ▶ écrire le segment trié de M blocs en mémoire externe

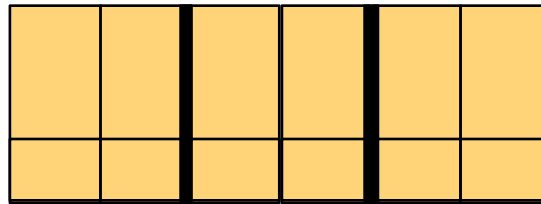


2	5	7	12	18	22	

segments triés
(mémoire externe)

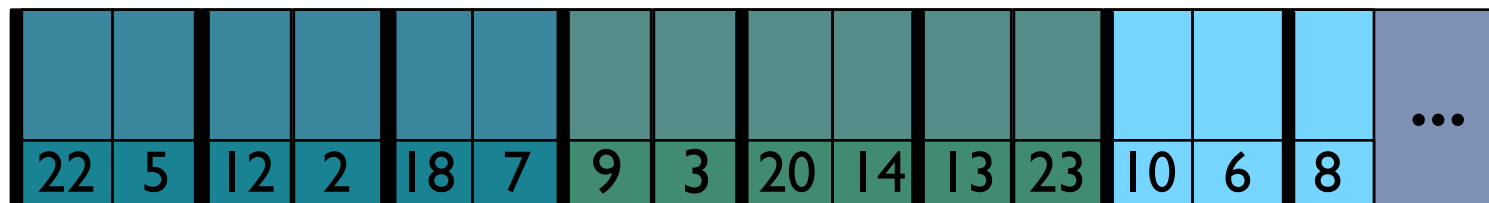
Production des segments triés

BUFFER (M=3 blocs) mémoire principale



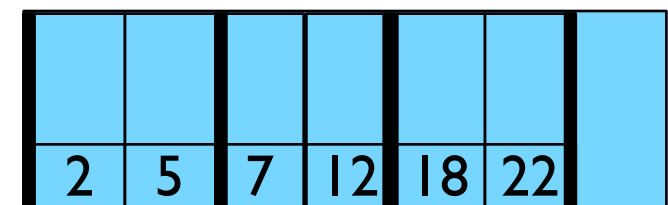
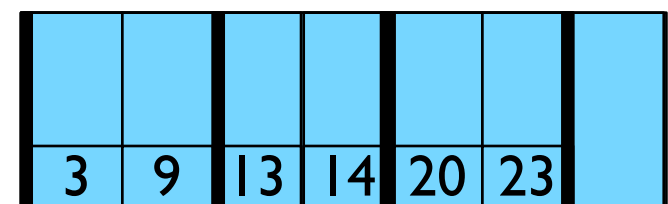
écriture de 3 blocs

Fichier de données (mémoire externe)



Tant qu'il reste des blocs à lire dans le fichier de données

- ▶ lire les M prochains blocs du fichier de données dans le buffer (ou ceux qui restent, si moins que M)
- ▶ trier les tuples dans le buffer en mémoire principale
- ▶ écrire le segment trié de M blocs en mémoire externe



segments triées
(mémoire externe)

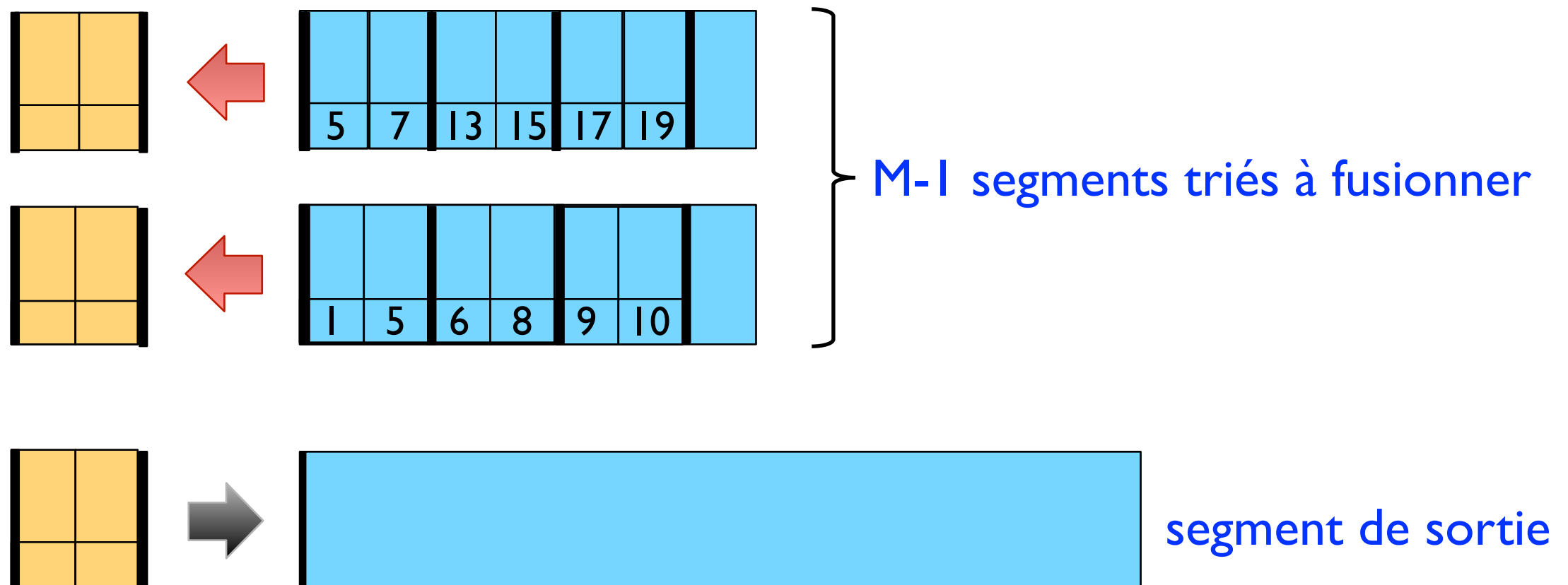
Production des segments triés

- Coût :
 - ▶ $2 B_R$ opérations d'entré-sortie (B_R lectures, B_R écritures)
 - ▶ $\lceil B_R / M \rceil$ tris internes
- Résultat :
 - ▶ $\lceil B_R / M \rceil$ segments triés de taille au plus M

Fusion des segments triés

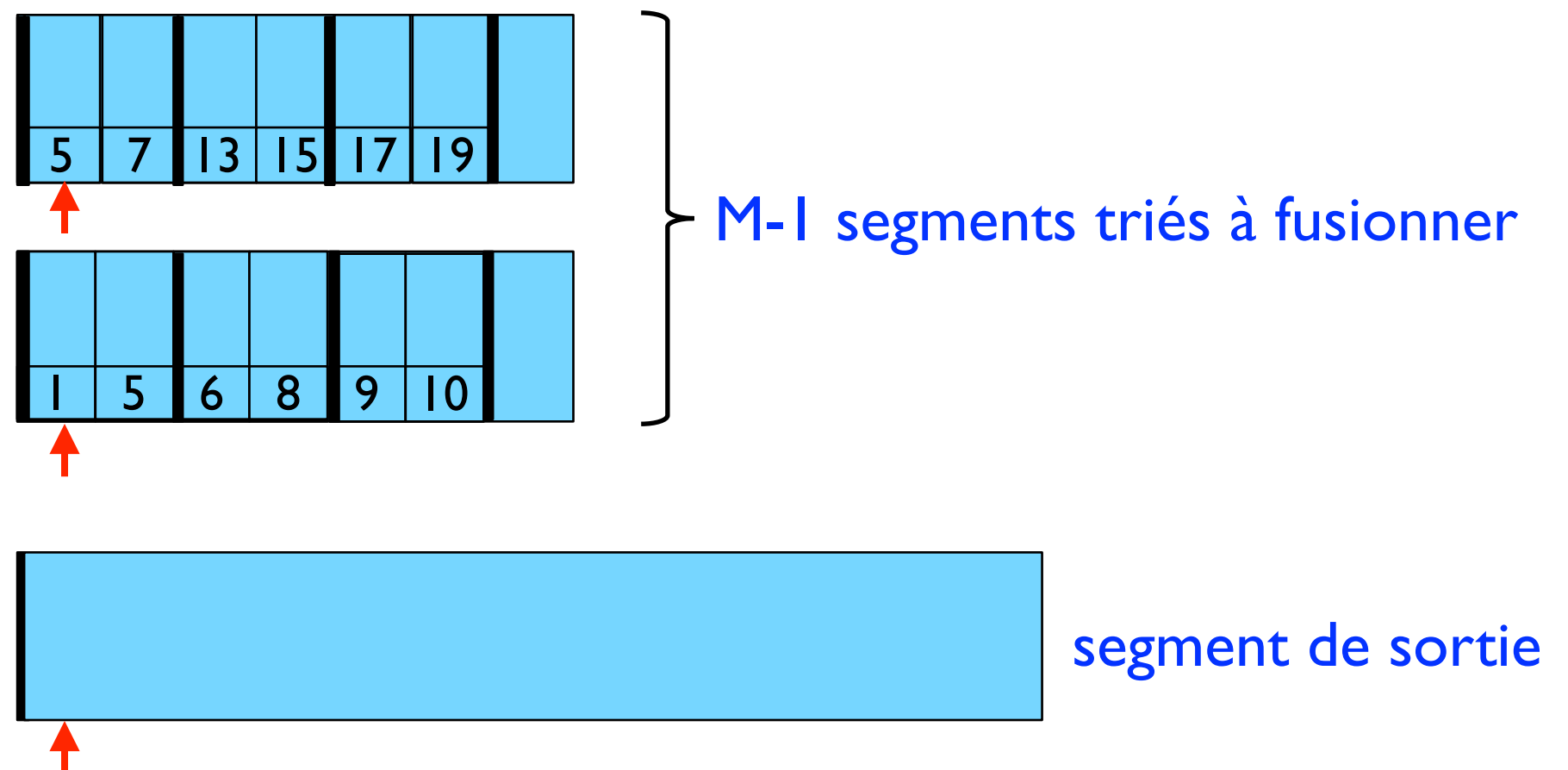
On utilise un bloc du buffer pour chaque segment trié (lecture du segment, un bloc à la fois) et un bloc pour le segment de sortie (écriture du segment, un bloc à la fois)

=> au plus $M-1$ à la fois segments peuvent être fusionnés



Fusion de M-1 segments triés

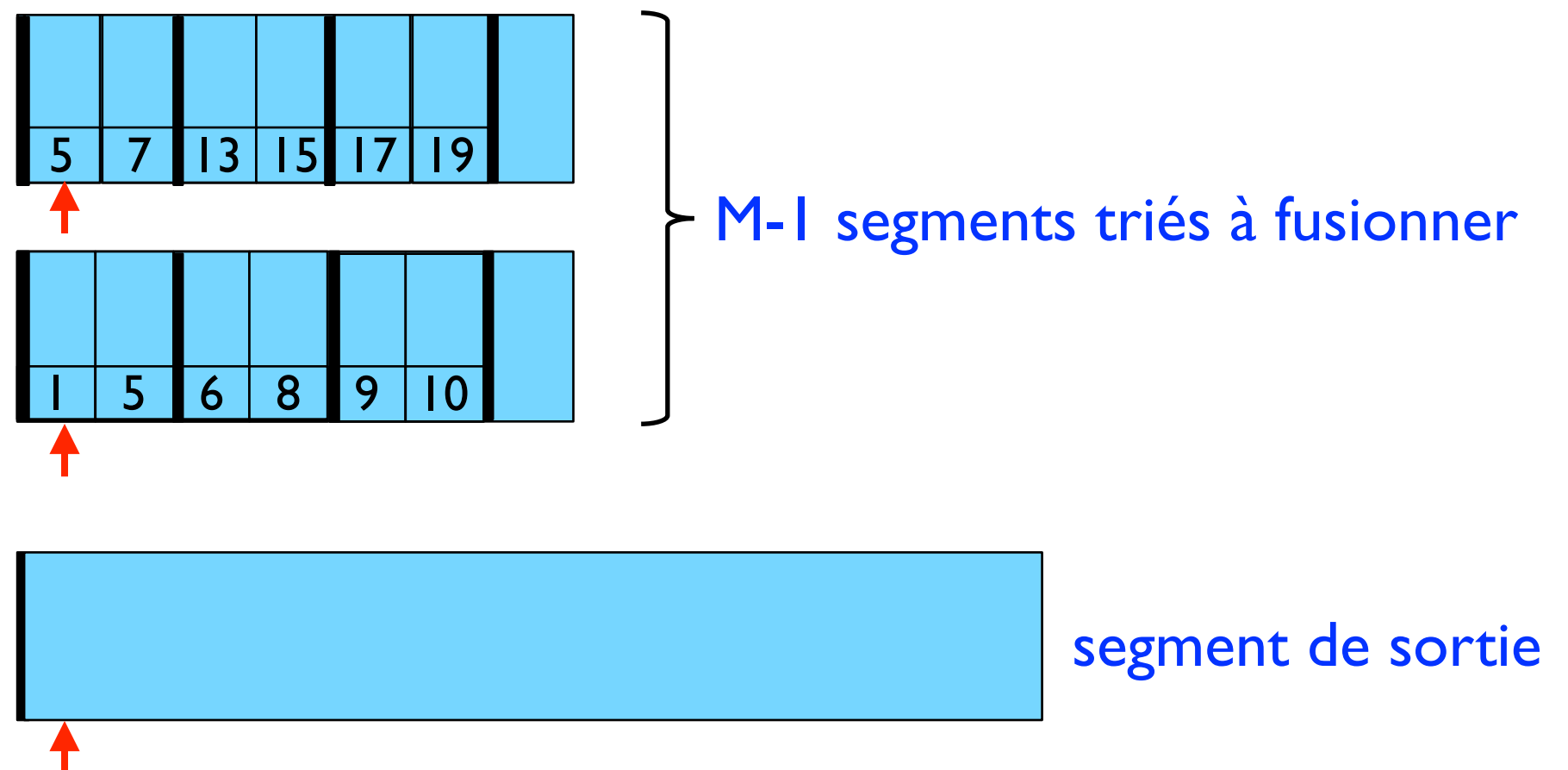
On fait abstraction des chargements / déchargements des blocs en memoire centrale : on imagine avoir une tête de lecture disponible sur chaque segment à fusionner et une tête d'écriture sur le segment de sortie



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

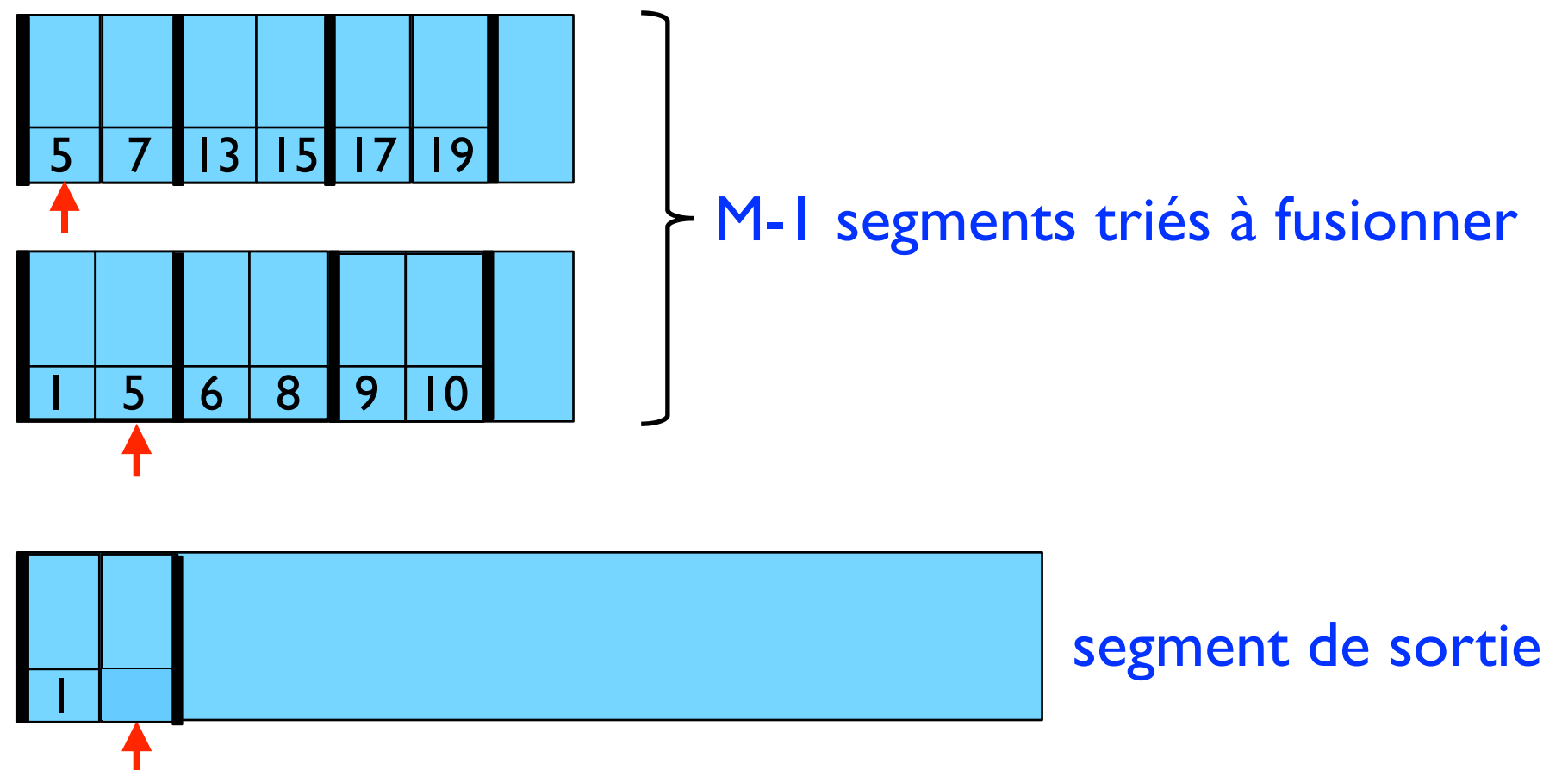
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

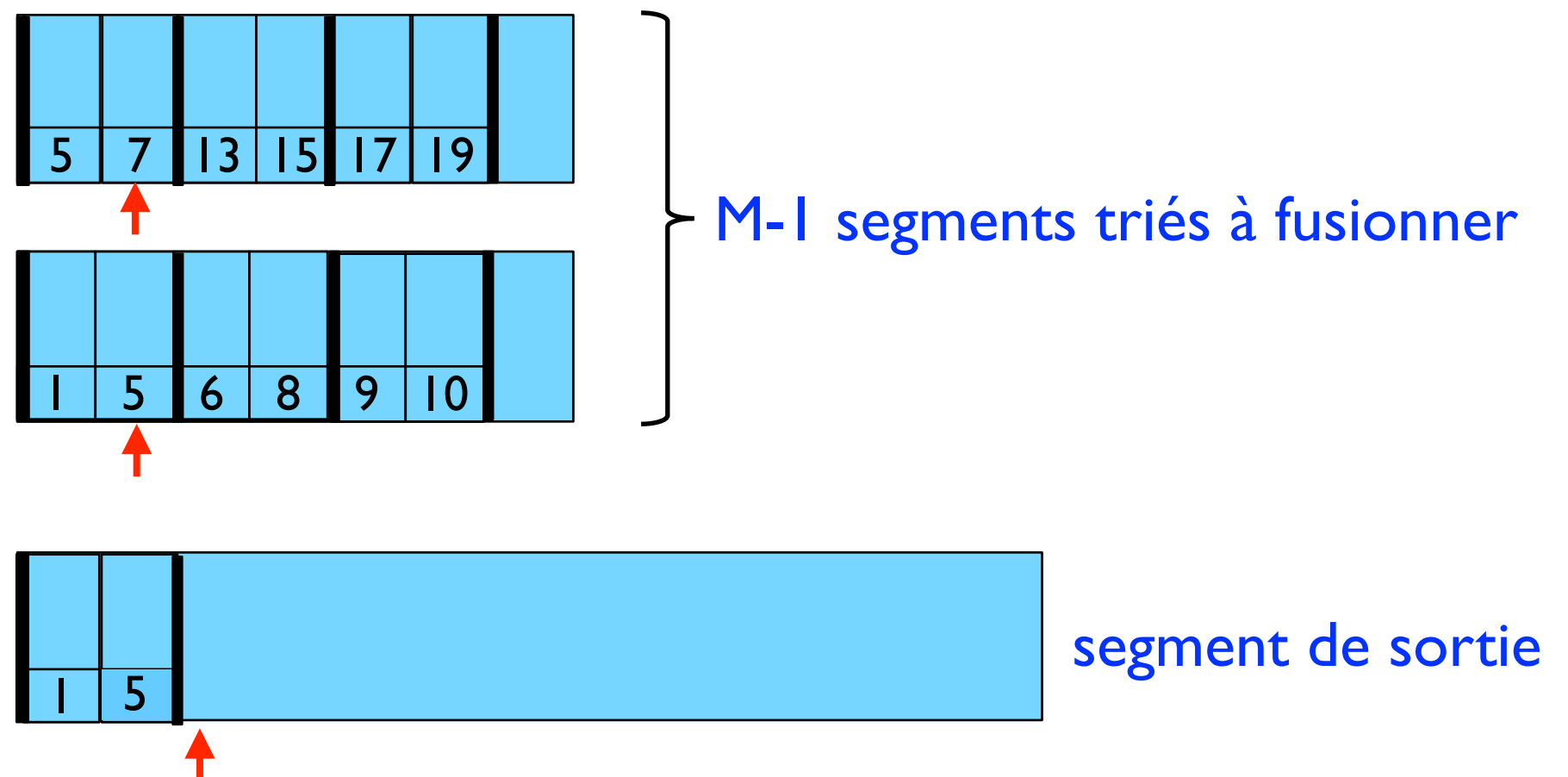
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

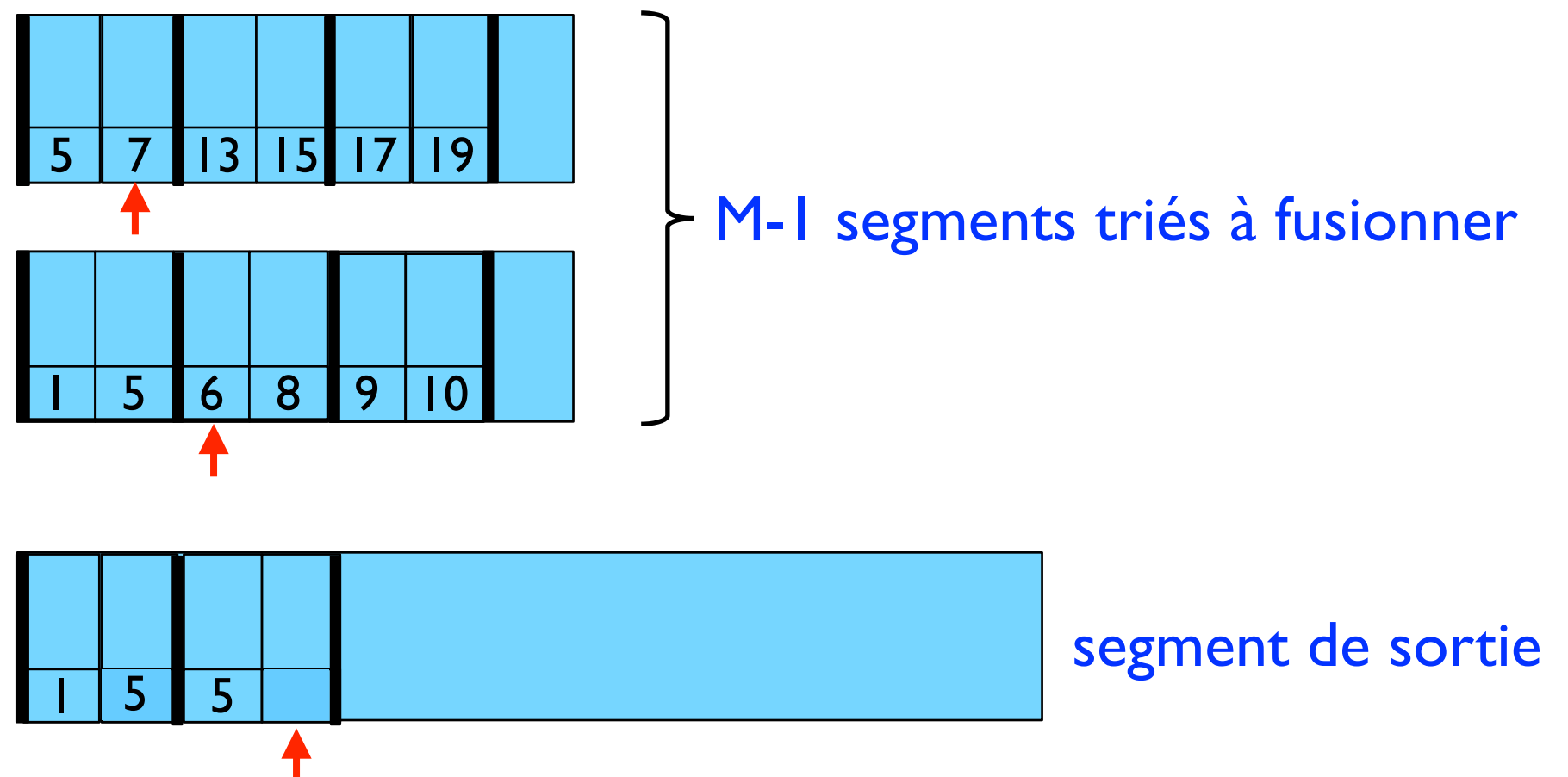
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

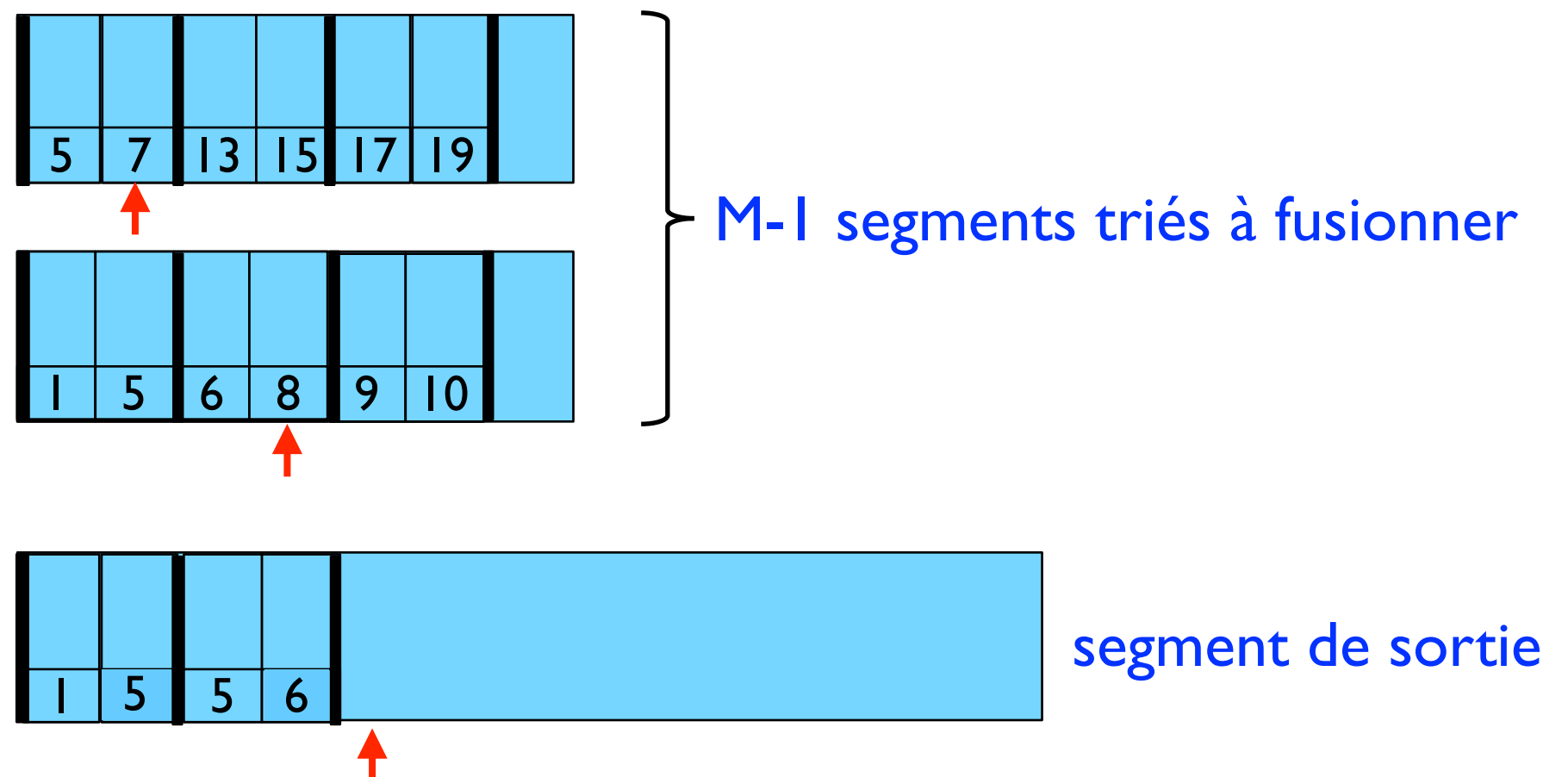
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

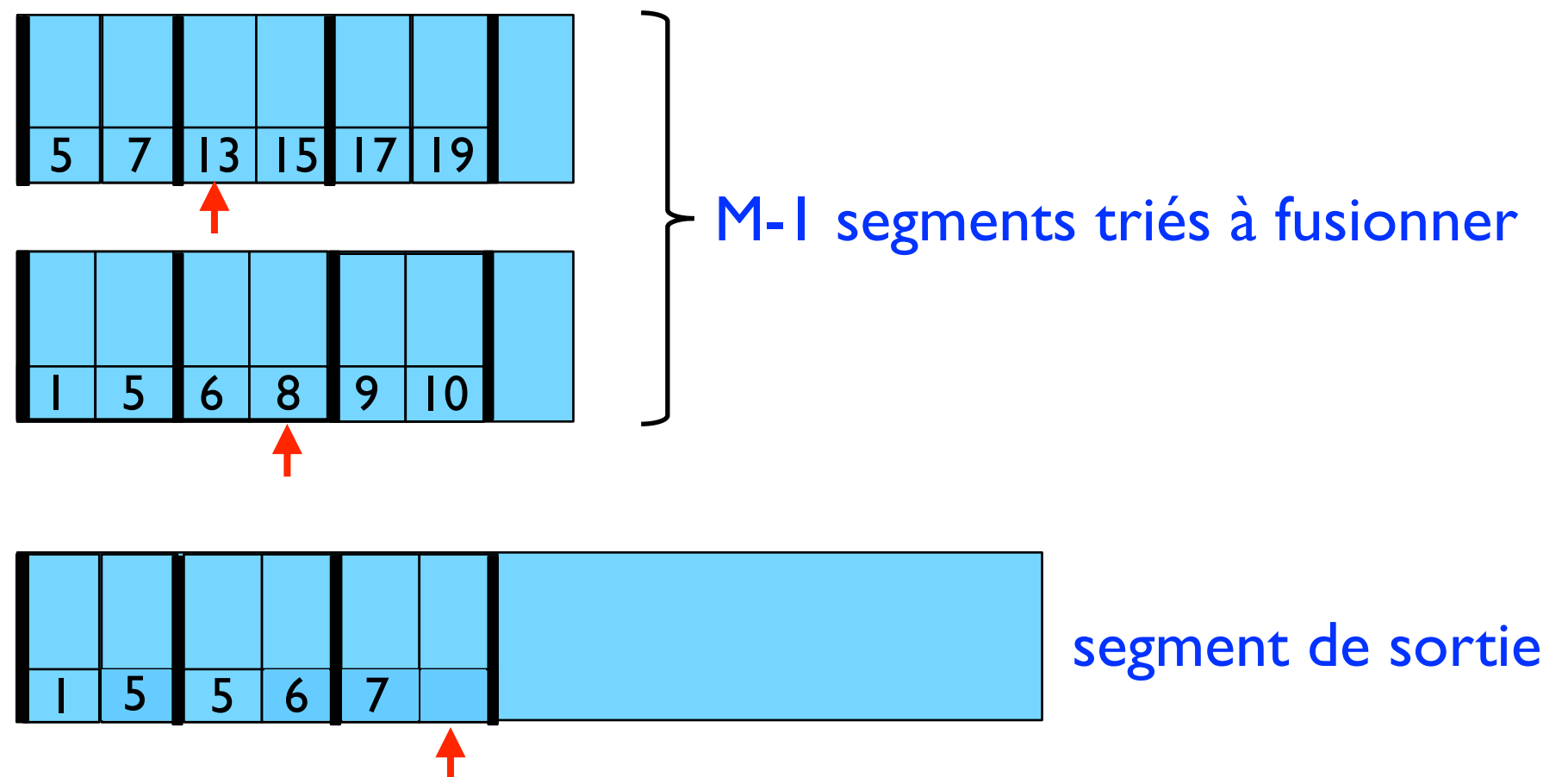
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

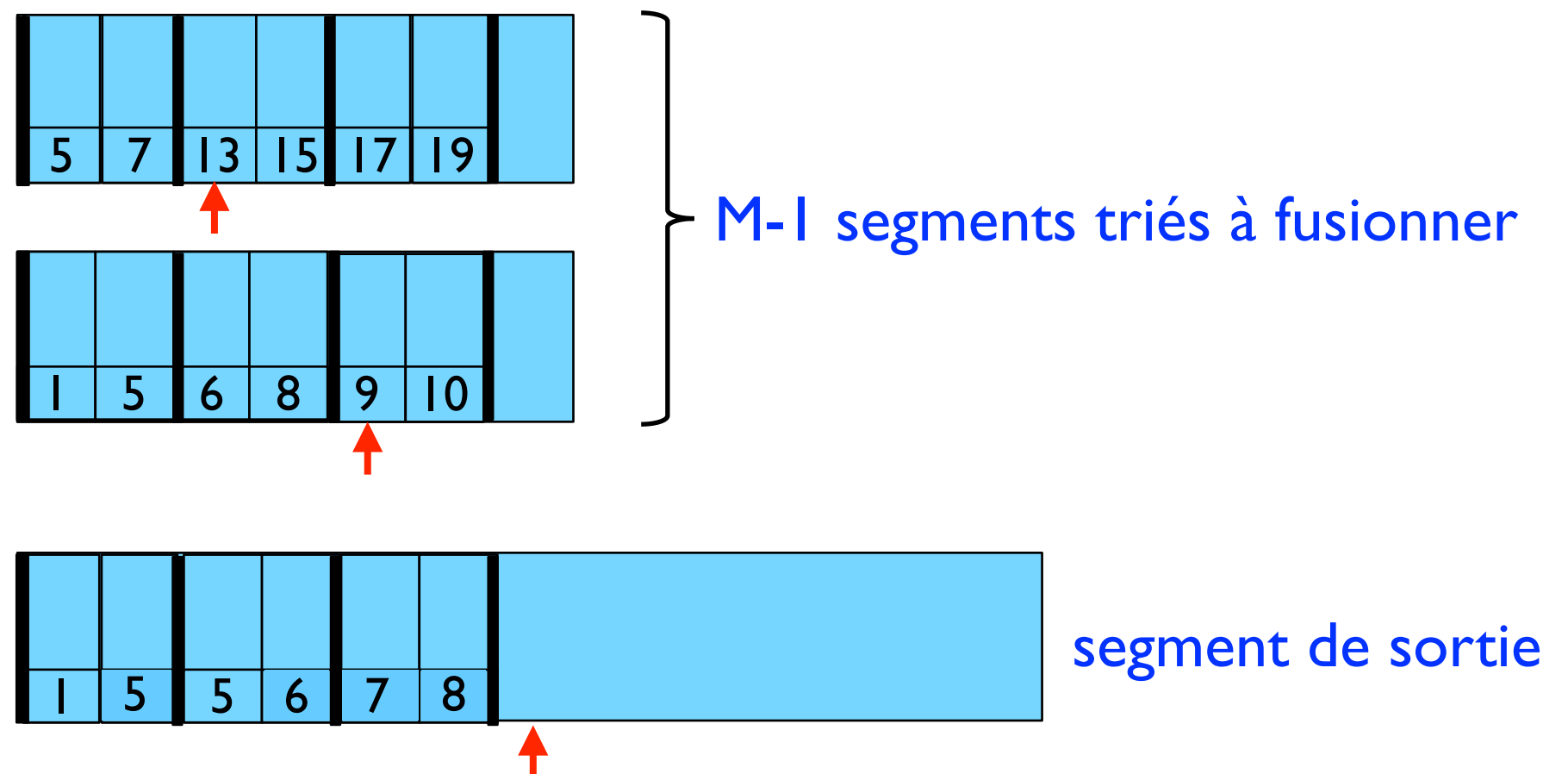
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

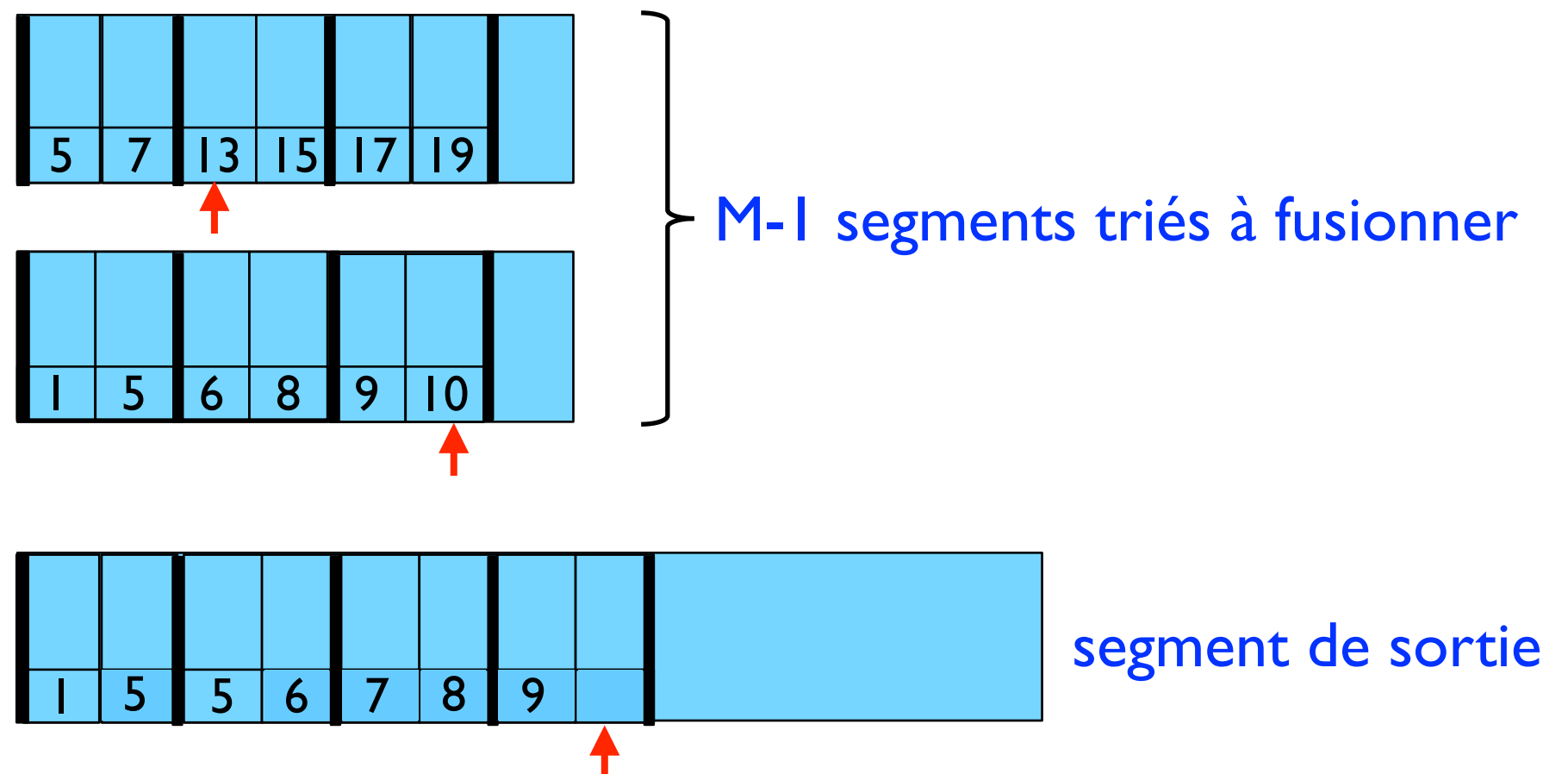
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

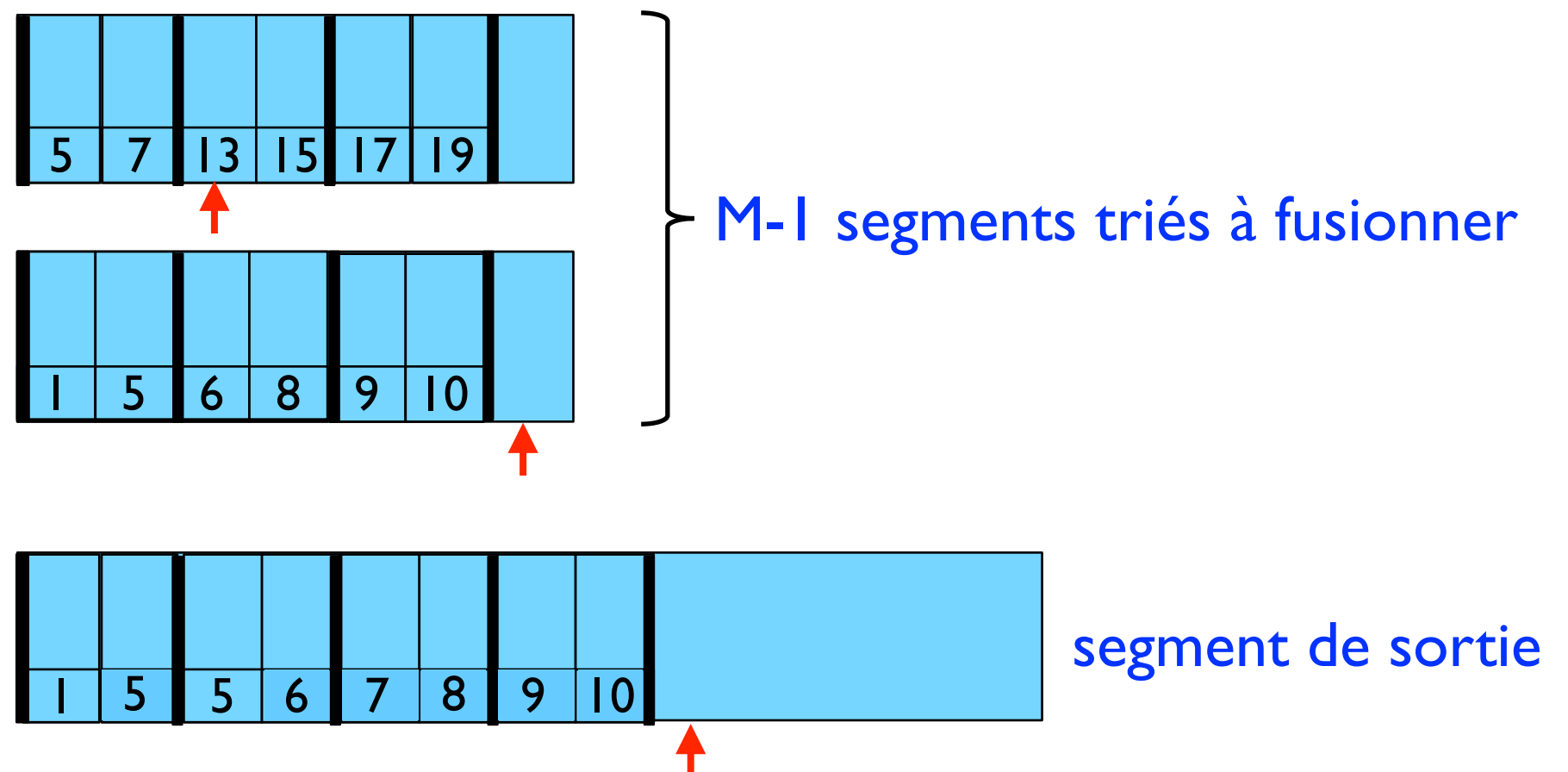
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

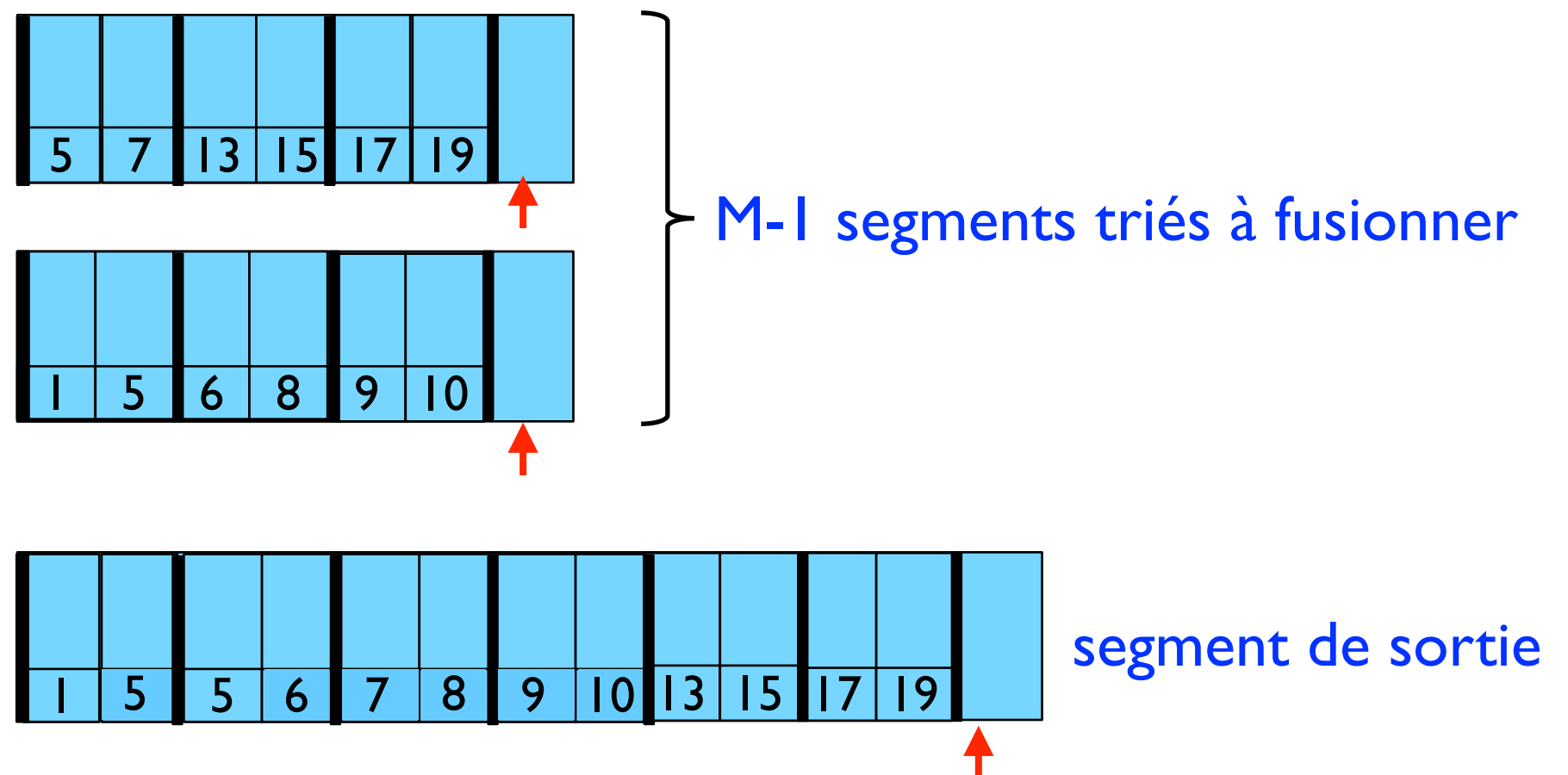
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion de M-1 segments triés

Tant que au moins une tête de lecture n'a pas atteint EOF

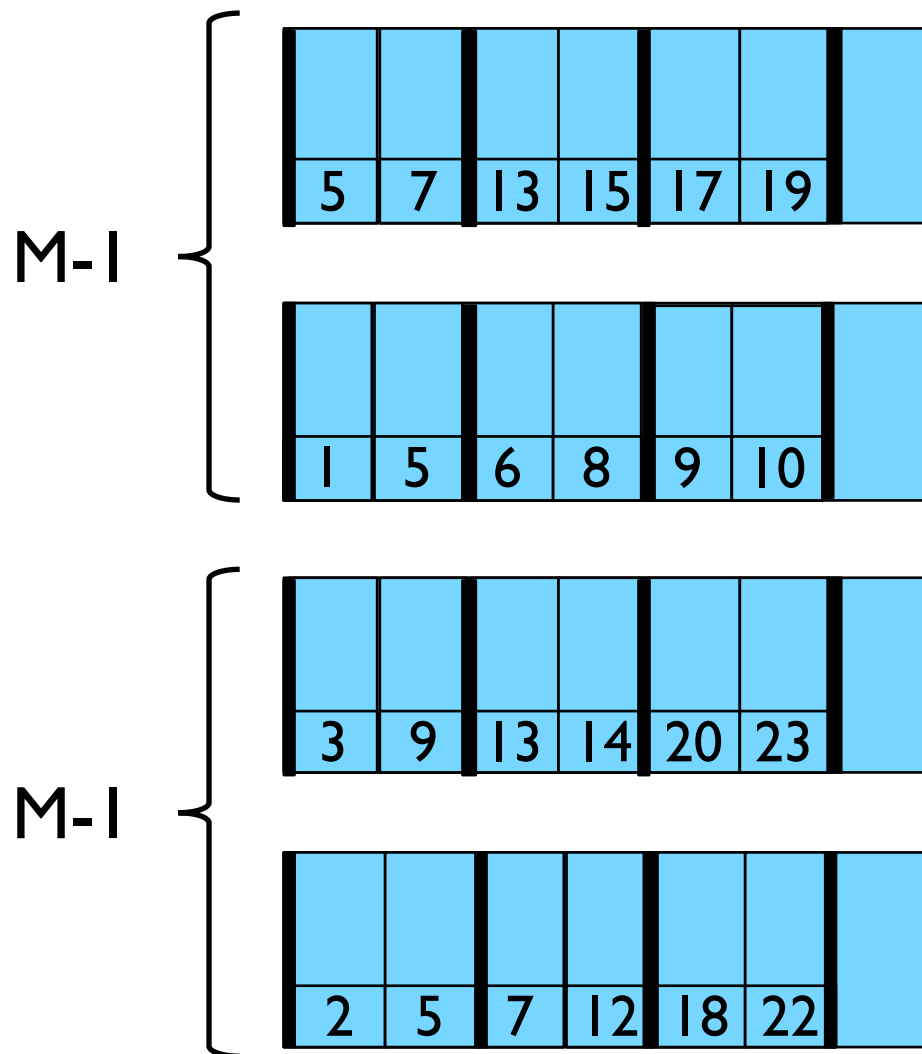
- ▶ choisir la plus petite clef sous une tête de lecture
- ▶ la copier dans le bloc d'output
- ▶ faire avancer la tête d'écriture ainsi que la tête de lecture du segment lu



Fusion des segments triés

Tant qu'il existe $N > 1$ segments triés

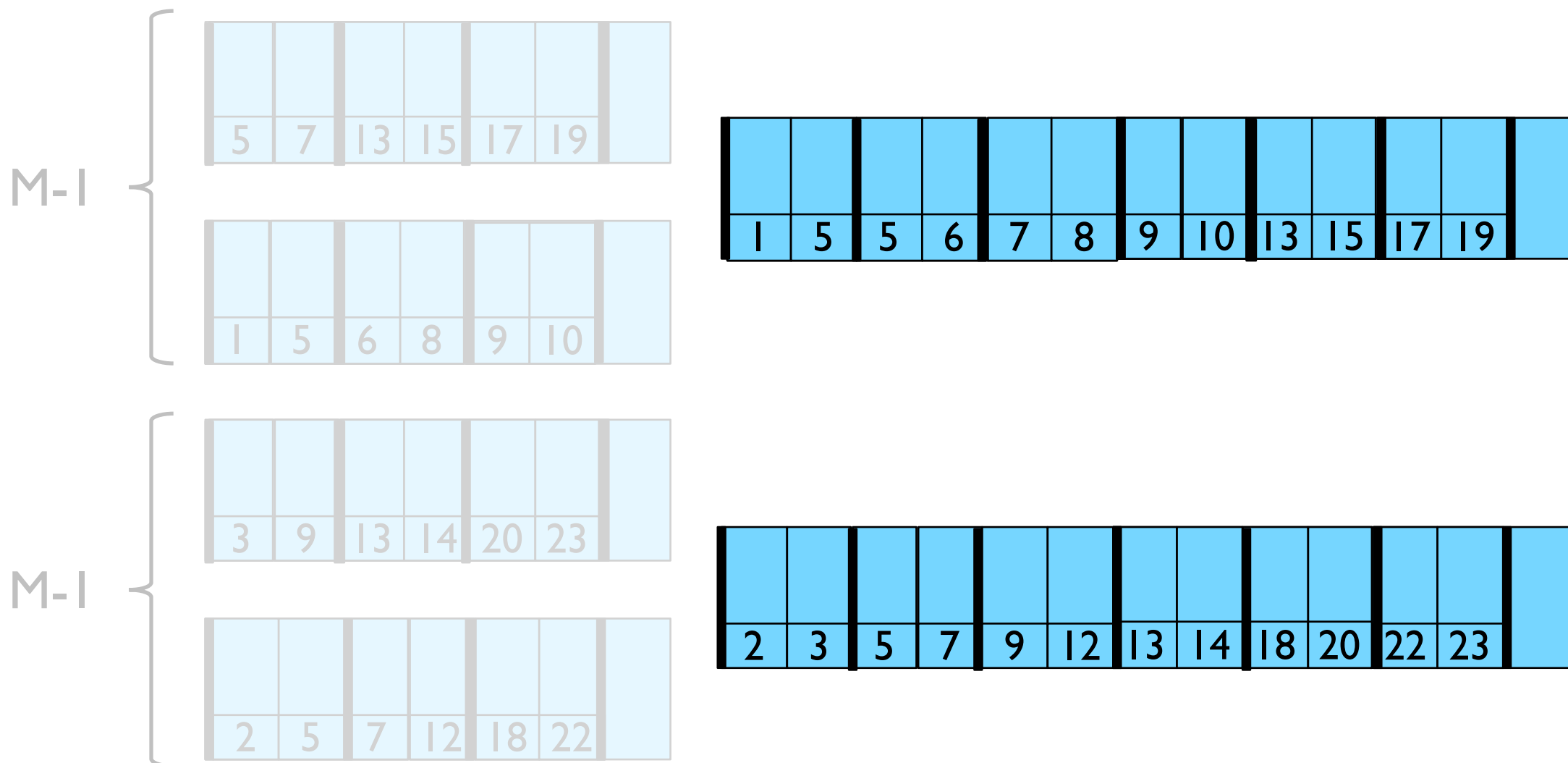
- ▶ Exécuter une passe de fusion :
 - fusionner les N segments triés, $M-1$ à la fois
 - Résultat de la passe : $\lceil N / (M-1) \rceil$ segments triés de taille double



Fusion des segments triés

Tant qu'il existe $N > 1$ segments triés

- ▶ Exécuter une passe de fusion :
 - fusionner les N segments triés, $M-1$ à la fois
 - Résultat de la passe : $\lceil N / (M-1) \rceil$ segments triés de taille double

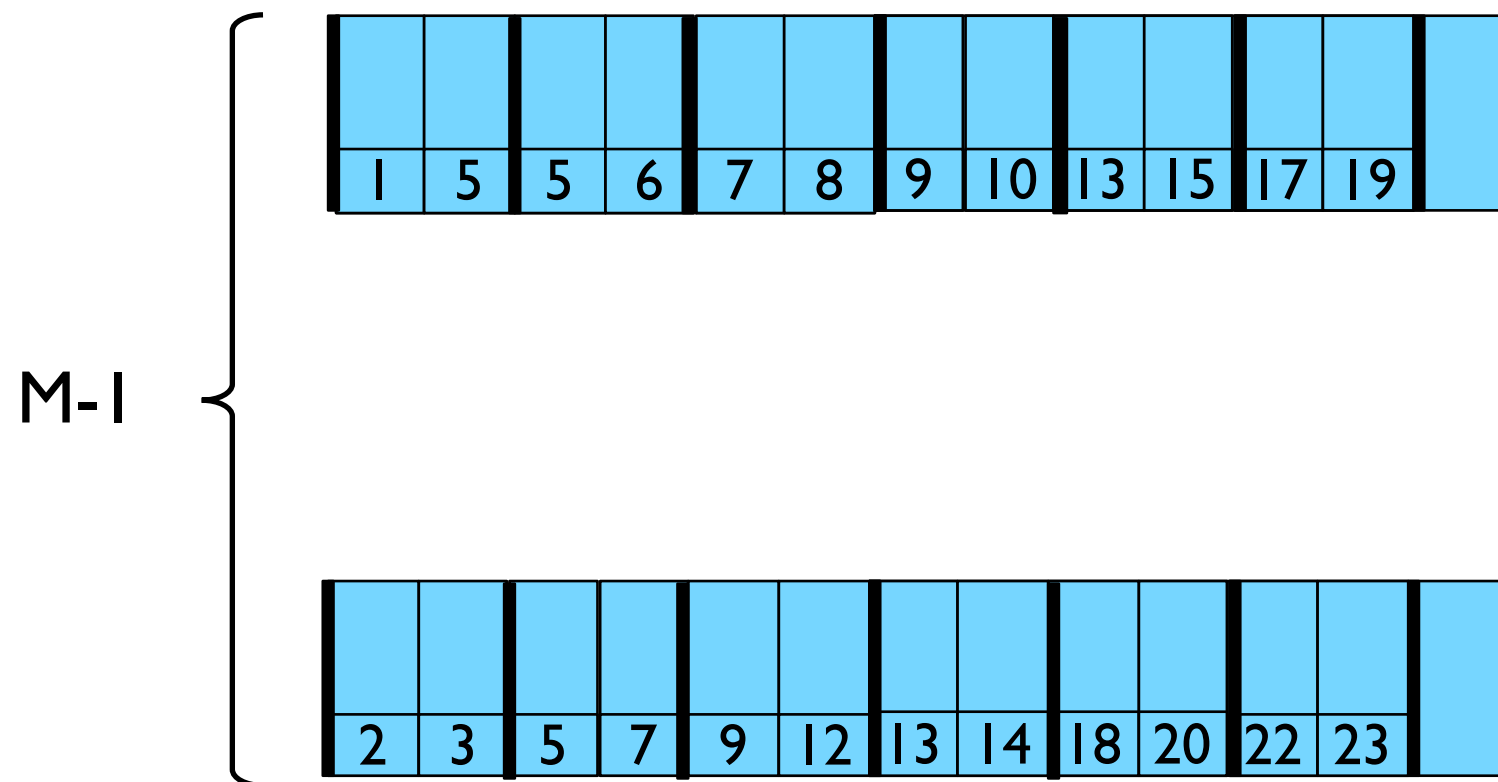


Première passe de fusion

Fusion des segments triés

Tant qu'il existe $N > 1$ segments triés

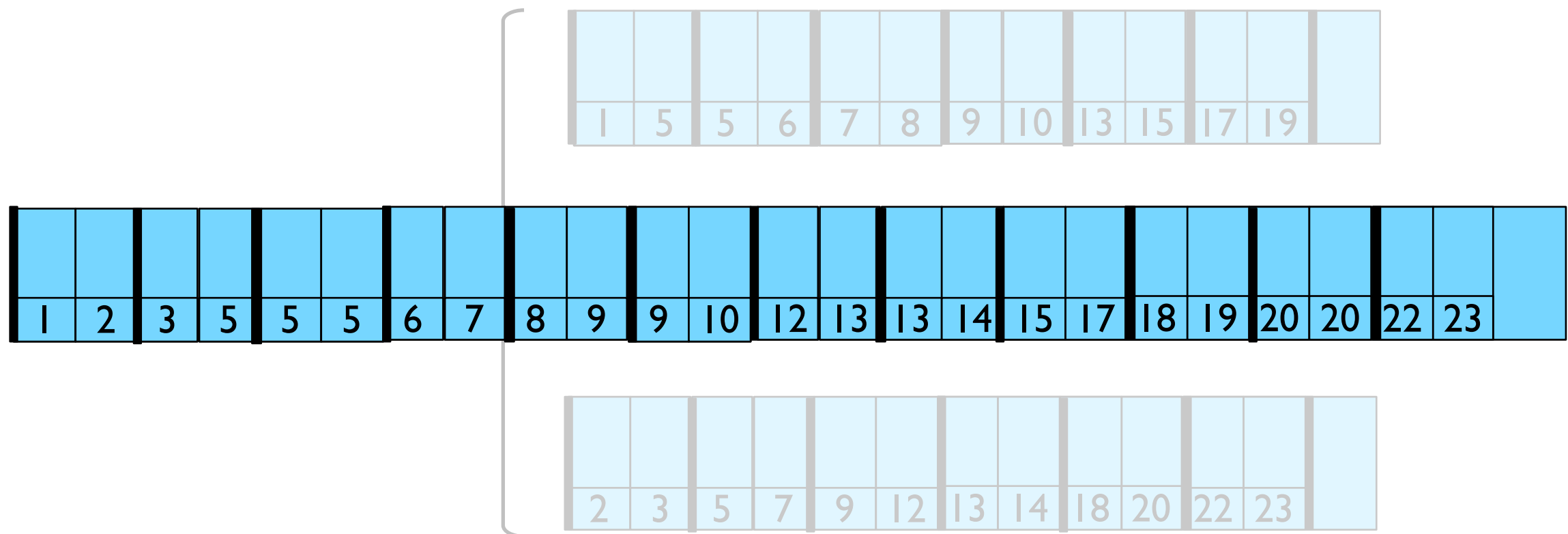
- ▶ Exécuter une passe de fusion :
 - fusionner les N segments triés, $M-1$ à la fois
 - Résultat de la passe : $\lceil N / (M-1) \rceil$ segments triés de taille double



Fusion des segments triés

Tant qu'il existe $N > 1$ segments triés

- ▶ Exécuter une passe de fusion :
 - fusionner les N segments triés, $M-1$ à la fois
 - Résultat de la passe : $\lceil N / (M-1) \rceil$ segments triés de taille double



Deuxième passe de fusion

Fusion des segments triés

Tant qu'il existe $N > 1$ segments triés

- ▶ Exécuter une passe de fusion :
 - fusionner les N segments triés, M-1 à la fois
 - Résultat de la passe : $\lceil N / (M-1) \rceil$ segments triés de taille double

1	2	3	5	5	5	6	7	8	9	9	10	12	13	13	14	15	17	18	19	20	20	22	23	

Un seul segment trié \Rightarrow FIN

Fusion des segments triés

- Nombre de segments initial : $\lceil B_R / M \rceil$
- Chaque passe réduit le nombre de segments d'un facteur $M-1$
- \Rightarrow après $\lceil \log_{M-1} (B_R / M) \rceil$ passes, on obtient un seul segment trié
- Coût de la fusion
 - ▶ $\lceil \log_{M-1} (B_R / M) \rceil \cdot$ coût d'une passe
- Coût d'une passe de fusion : tous les blocs des segments fusionnés sont lus une fois et écrits une fois : $2B_R$

Coût du tri en mémoire externe

- Coût de production des segments triés : $2 B_R$ (B_R lectures, B_R écritures)
- Coût d'une passe de fusion
 - ▶ $2 B_R$ (B_R lectures et B_R écritures)
- Coût de la fusion
 - ▶ $\lceil \log_{M-1} (B_R / M) \rceil \cdot 2 B_R$

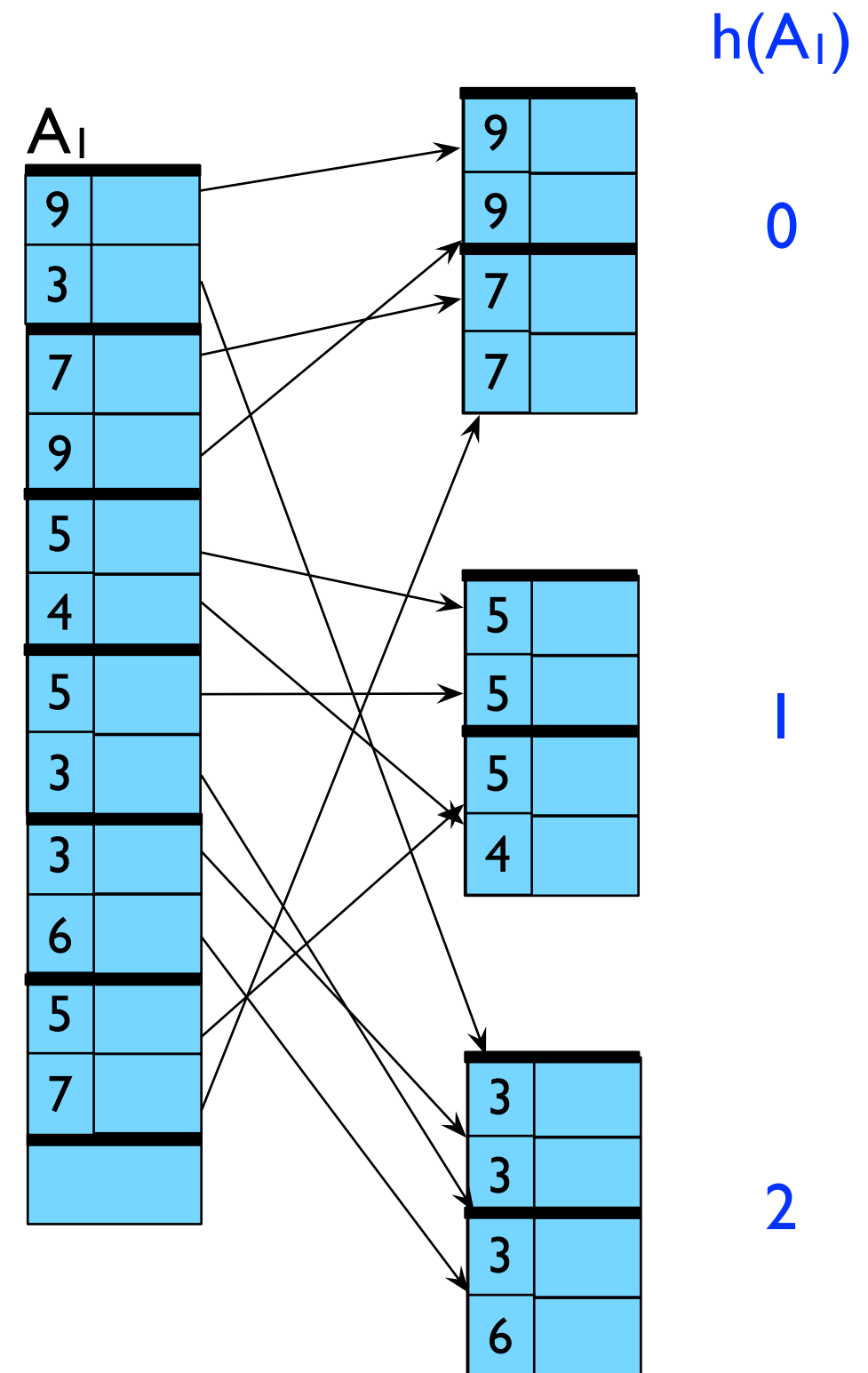
Coût du tri (production des segments triés + fusion)

▶ $2 B_R + \lceil \log_{M-1} (B_R / M) \rceil \cdot 2 B_R$

L'écriture du dernier segment trié (coût B_R) est économisée si le résultat n'est pas matérialisé

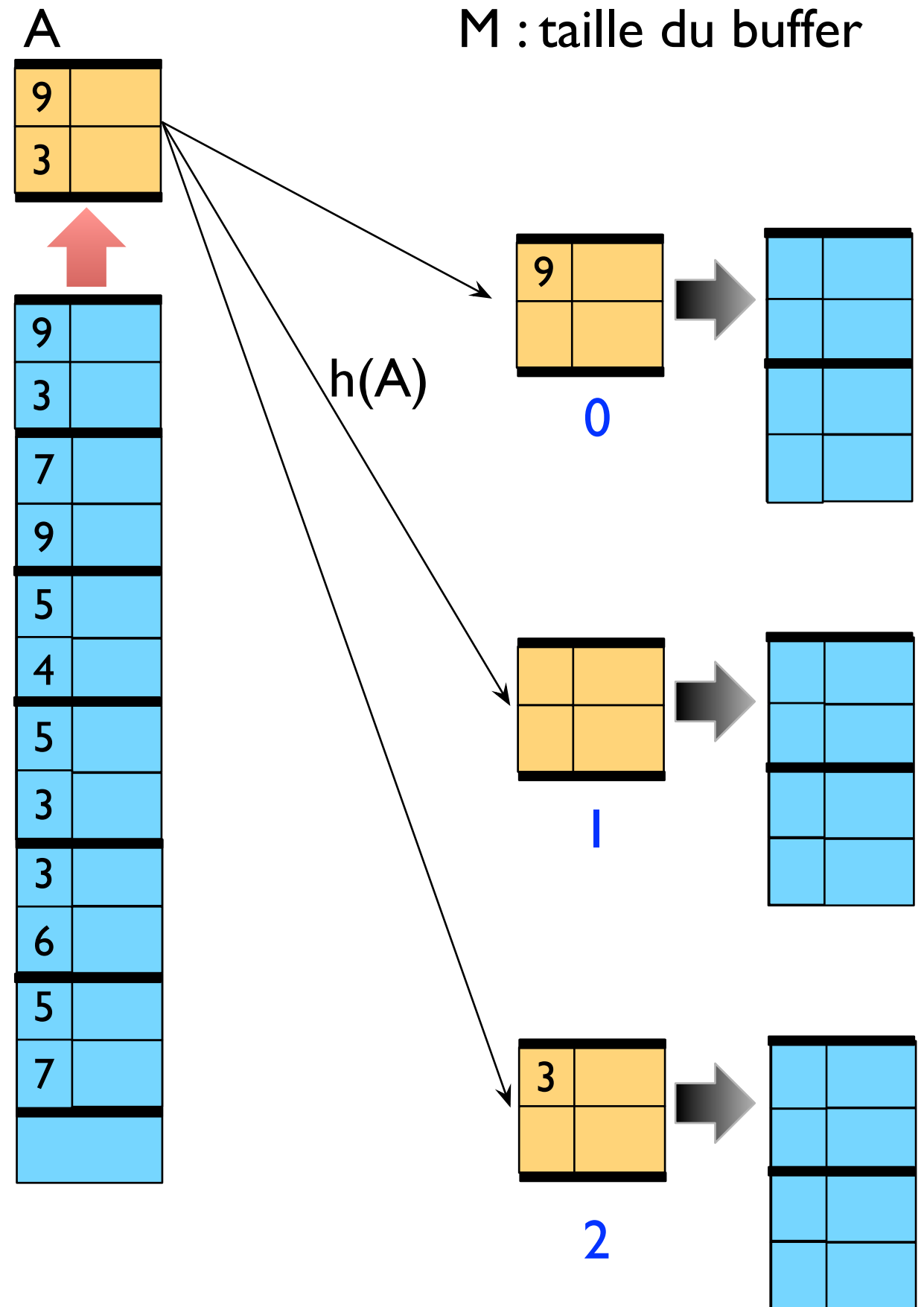
Hachage d'un fichier de données

- **But** : partitionner les tuples d'un fichier de données sur la base d'un ensemble d'attributs $A_1 \dots A_n$ (Exemple $n = 1$)
 - ▶ Utiliser une fonction de hachage h définie sur le domaine de $A_1 \dots A_n$
 - ▶ Dans la partition i :
les tuples t tels que $h(t[A_1 \dots A_n]) = i$
- **Remarque**
 - ▶ les tuples avec la même valeur des attributs $A_1 \dots A_n$ sont placés dans la même partition
 - ▶ une partition peut contenir des tuples avec différents valeurs de $A_1 \dots A_n$



Hachage d'un fichier de données

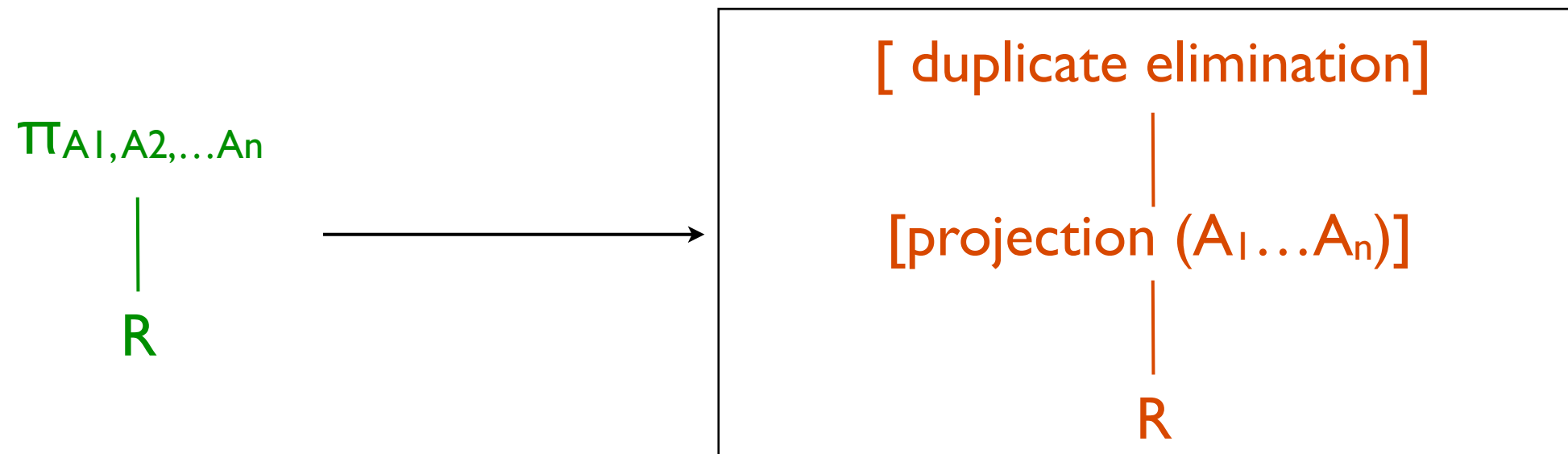
- **Implémentation :**
 - ▶ un bloc du buffer alloué à l'input
 - ▶ un bloc du buffer alloué à chaque partition
- Génère au plus $M-1$ partitions
- Pour obtenir $N \geq M$ partitions : **partition récursive**
 - ▶ générer $M-1$ partitions en une passe
 - ▶ tant que le nombre de part. $< N$
 - re-hacher chaque partition en $M-1$ nouvelles partitions
 - ▶ utiliser une fonction de hachage différente à chaque passe



Coût du hachage

- Chaque passe lit une fois et écrit une fois tous les blocs du fichier \Rightarrow
 - ▶ coût d'une passe : $2 B_R$
- Chaque passe multiplie le nombre de partitions par $M-1 \Rightarrow$
 - ▶ nombre de passes pour obtenir au moins N partitions : $\lceil \log_{M-1} N \rceil$
- Coût pour au moins N partitions : $2 B_R \cdot \lceil \log_{M-1} N \rceil$

Projection



- Élimination des doublons nécessaire sous sémantique ensembliste de la projection (correspondante à `SELECT DISTINCT` en SQL)

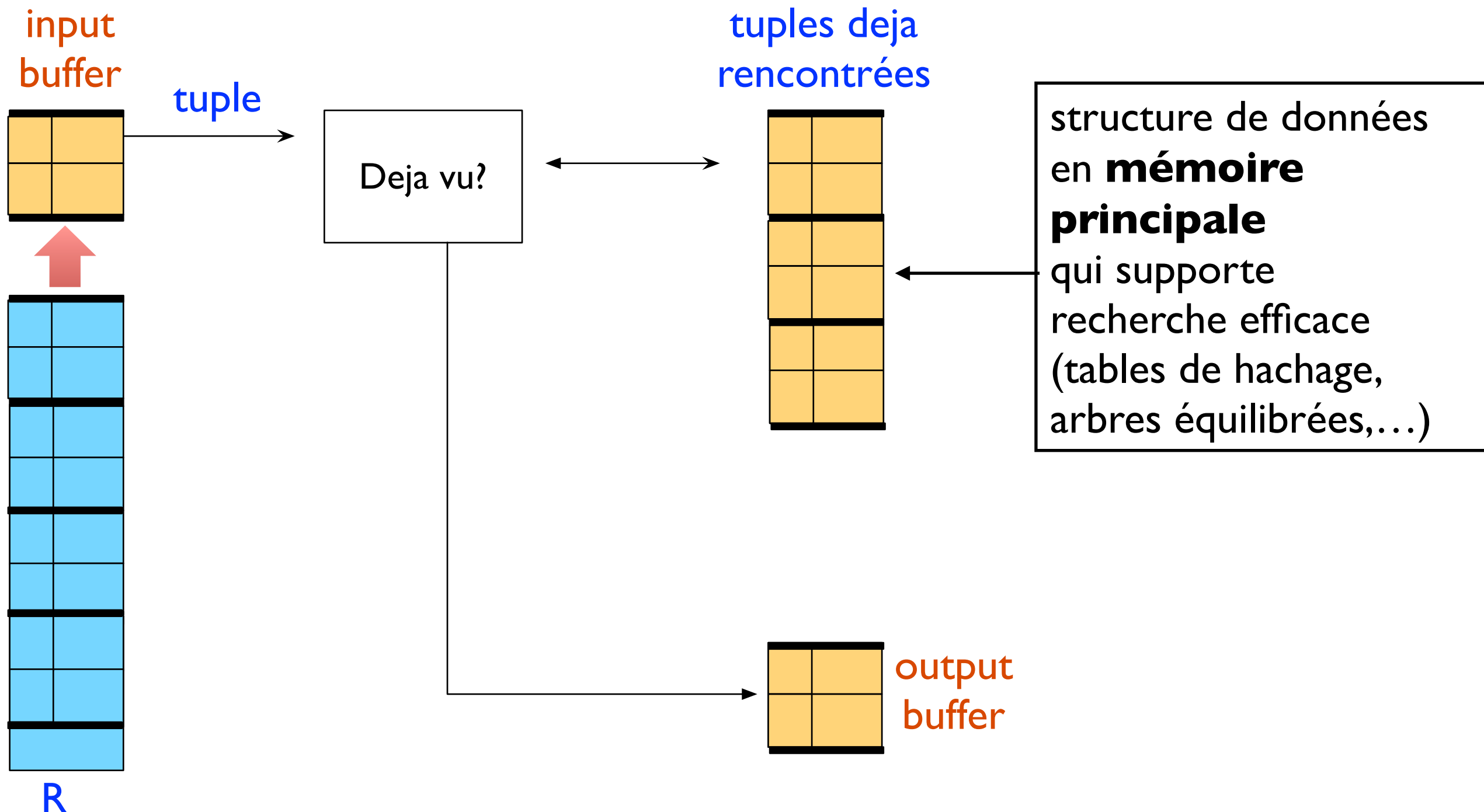
Elimination des doublons : implementations possibles

I. Par tri

- ▶ effectuer un **tri externe** sur tous les attributs
- ▶ pendant la phase de fusion du fichier, les doublons des segments fusionnés apparaissent proches
 - à chaque fusion copier dans l'output un seul des doublons
- ▶ le même coût que le tri : $2 B_R + \lceil \log_{M-1} (B_R / M) \rceil \cdot 2 B_R$

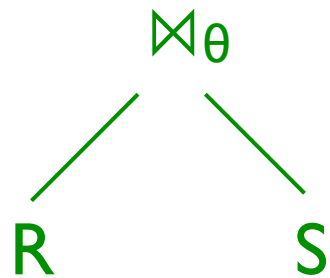
Elimination des doublons : implementations possibles

2. En une seule passe, si le nombre de tuples distincts de la relation occupe moins que $M-2$ blocs (M = nombre de blocs dans le buffer)

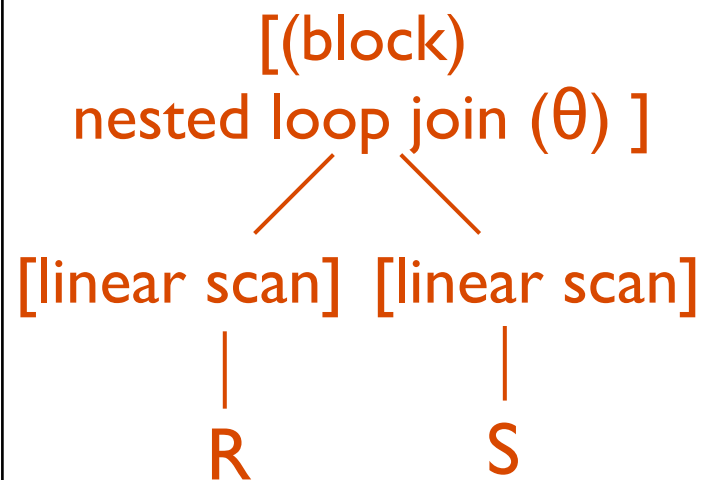


Jointure

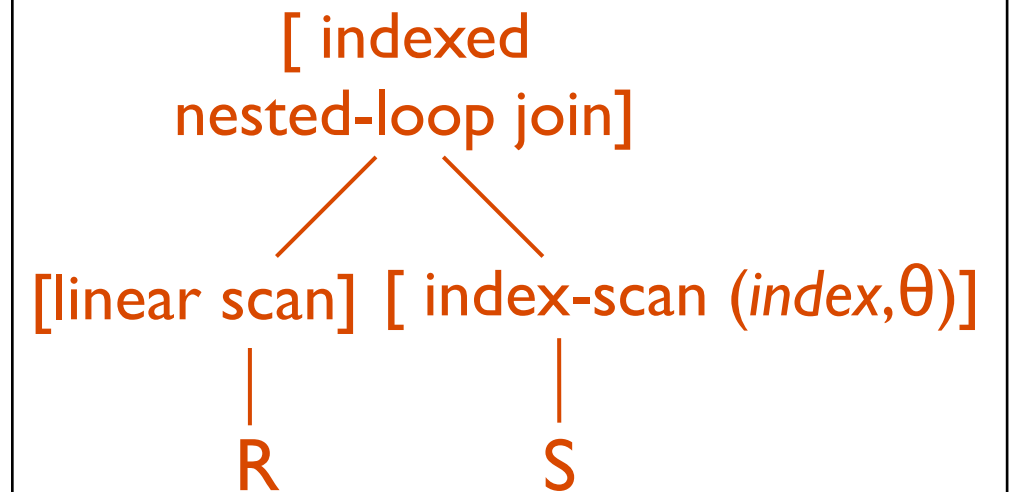
- Implementations du *theta-join*
(jointure naturelle : implémentée comme un *theta-join* suivi d'une projection)



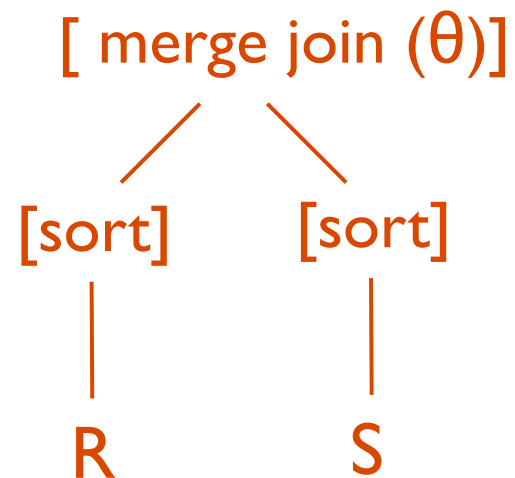
1)



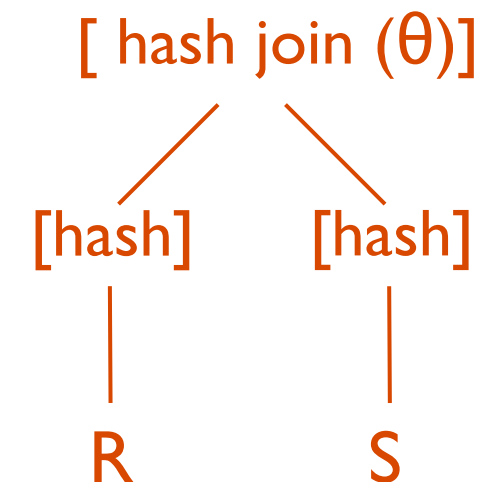
2)



3)



4)



Nested loop join

- Implémentation la moins efficace
- Générale : applicable avec des condition de jointure arbitraires

pour tout tuple t_R dans R
pour tout tuple t_S dans S
tester si $(t_R, t_S) \models \theta$
si oui, ajouter $t_R \cdot t_S$ au résultat

- Coût :

$$B_R + n_R \cdot B_S$$

- ▶ n_R, n_S : nombre de tuples de R, S, B_R, B_S : nombre de blocs de R, S
- ▶ parcours linéaire de R (B_R)
 - pour chaque tuple dans R, un scan complet de S
- ▶ on ne compte pas le coût d'écriture du résultat sur disque
- ▶ en générale plus efficace si la relation externe est la plus petite

Nested loop join

pour tout tuple t_R dans R
pour tout tuple t_S dans S
tester si $(t_R, t_S) \models \theta$
si oui, ajouter $t_R \cdot t_S$ au résultat

- Amélioration :
 - Si $\theta: R.A_1 = S.A_1, \dots, R.A_n = S.A_n$ avec (A_1, \dots, A_n) clef de S , la boucle sur S peut s'arrêter au premier tuple t_S tel que $(t_R, t_S) \models \theta$

Nested loop join \rightarrow jointure à une passe

- Si une des deux relations est suffisamment petite pour être stockée entièrement en mémoire centrale
 - ▶ la charger en memoire au debut
 - ▶ l'utiliser come relation interne du *nested loop join*

lire S en memoire principale
pour tout tuple t_R dans R
 pour tout tuple t_S dans S
 tester si $(t_R, t_S) \models \theta$
 si oui, ajouter $t_R \cdot t_S$ au résultat

- Coût

$$B_S + B_R$$

Block nested loop join

- Une amélioration de *nested loop join*

```
pour tout bloc  $b_R$  de R  
  pour tout bloc  $b_S$  de S  
    pour tout tuple  $t_R$  dans  $b_R$   
      pour tout tuple  $t_S$  dans  $b_S$   
        tester si  $(t_R, t_S) \models \theta$   
        si oui, ajouter  $t_R \cdot t_S$  au résultat
```

- Coût :

$$B_R (1 + B_S)$$

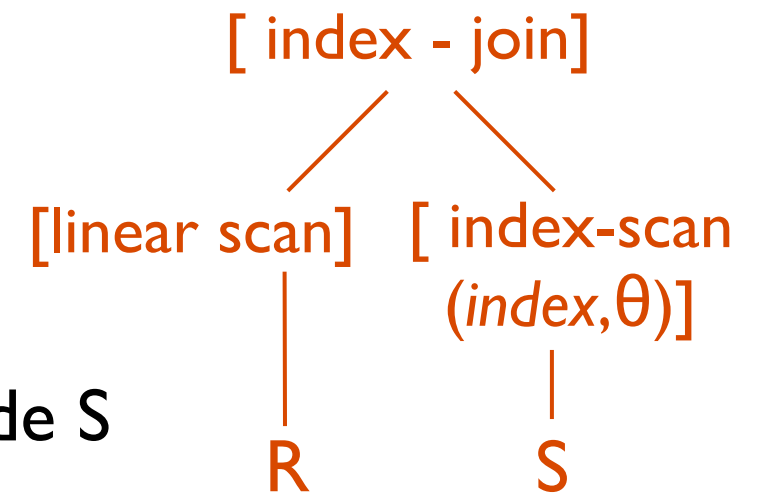
- ▶ pour chaque bloc de R : on lit le bloc et tous les blocs de S
- ▶ les opérations sur les tuples ont lieu en mémoire centrale
- ▶ plus efficace si la relation externe est la plus petite

Indexed nested loop join (ou index join)

- Condition d'applicabilité :

- $\theta: R.A_1 = S.A_1, \dots, R.A_n = S.A_k$
- un index I disponible sur $(S.A_1, \dots, S.A_k)$

- Dans nested loop join, on remplace le parcours interne de S par une recherche sur l'index



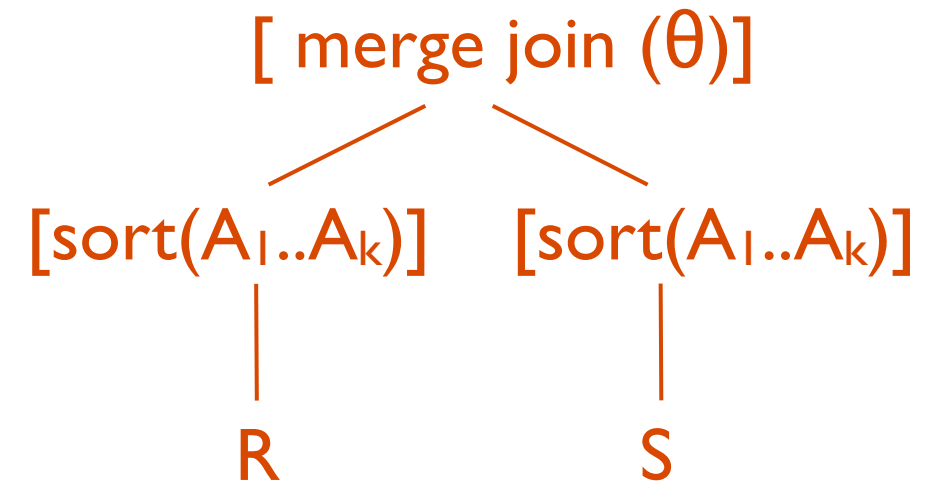
pour tout tuple t_R dans R
chercher la clef $v = t_R[A_1, \dots, A_k]$ dans I
pour chaque tuple t_S pointé par v dans I
ajouter $t_R \cdot t_S$ au résultat

- Coût :

$$B_R + n_R \cdot c$$

- c est le coût d'une sélection par *index-scan*
 - Rappel : index primaire : $c = h_I + B_v$, index secondaire : $c = h_I + n_v$
- plus efficace quand la relation externe est la plus petite

Merge join (aussi appelé sort join)

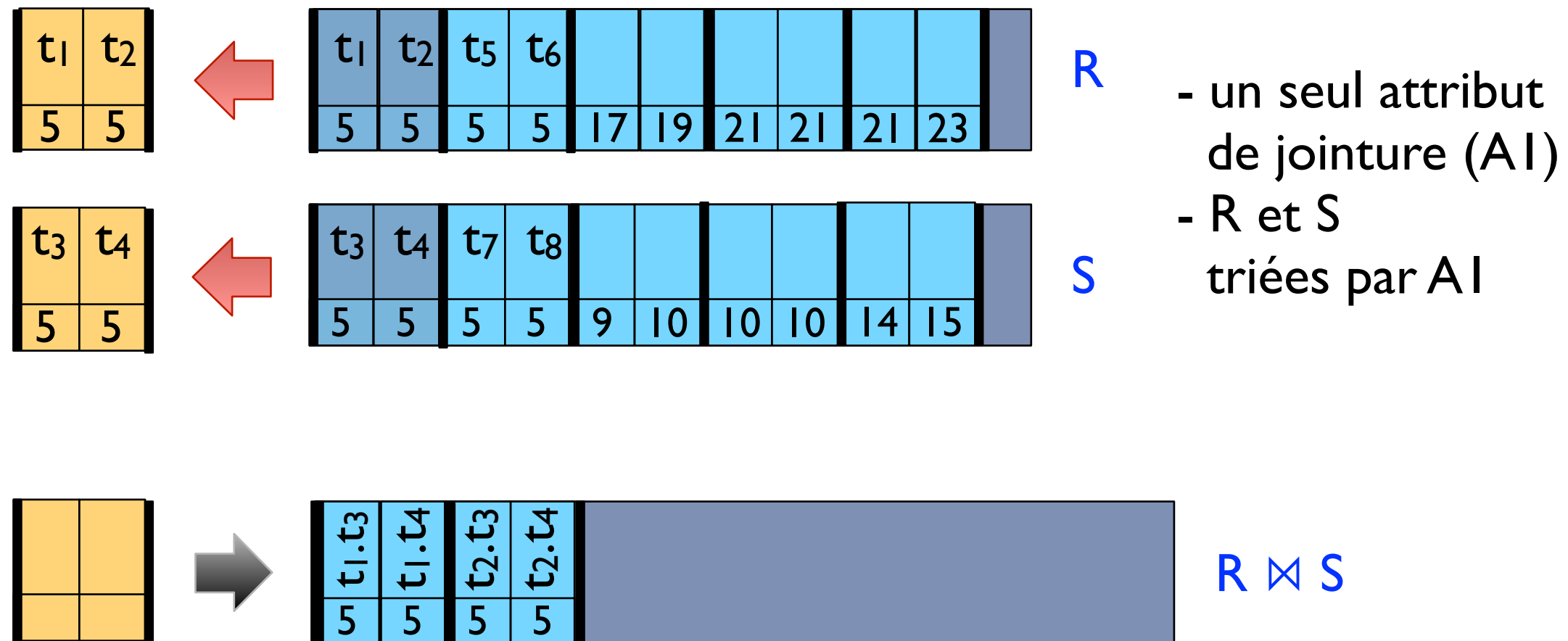


- applicable si θ : $R.A_1 = S.A_1, \dots, R.A_n = S.A_k$
- Idée :
 - ▶ Trier R et S sur A_1, \dots, A_k
 - ▶ Fusionner R et S comme deux segments triés dans le tri externe
 - ▶ Pendant la fusion les tuples avec la même valeur de A_1, \dots, A_k dans les deux tables, sont accédés consécutivement
 - joindre ces tuples pendant la fusion
- Difficulté :
 - ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
 - ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas pour la fusion

Merge join

- Difficulté :

- ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
- ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas :

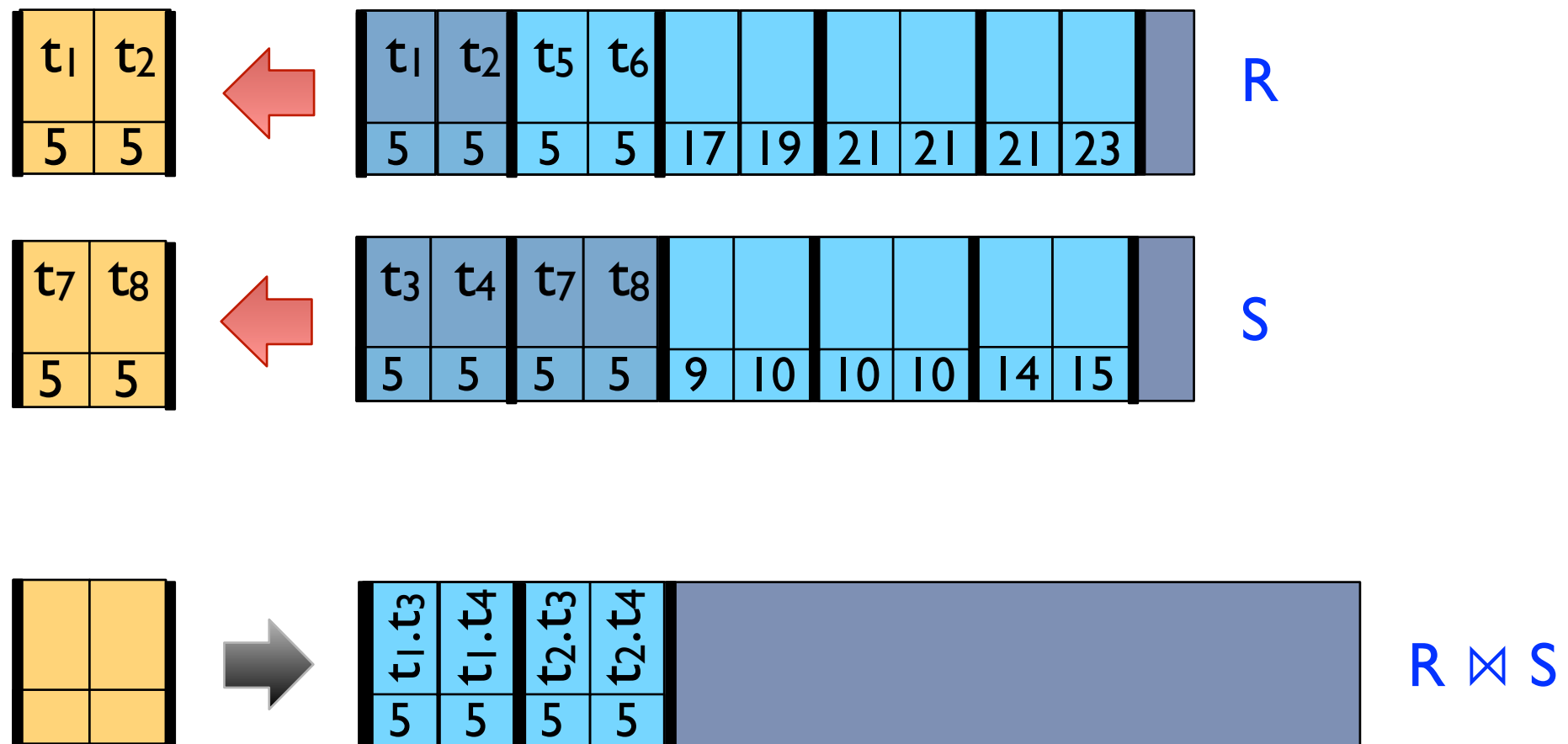


Exemple : un bloc de buffer par relation

Merge join

- Difficulté :

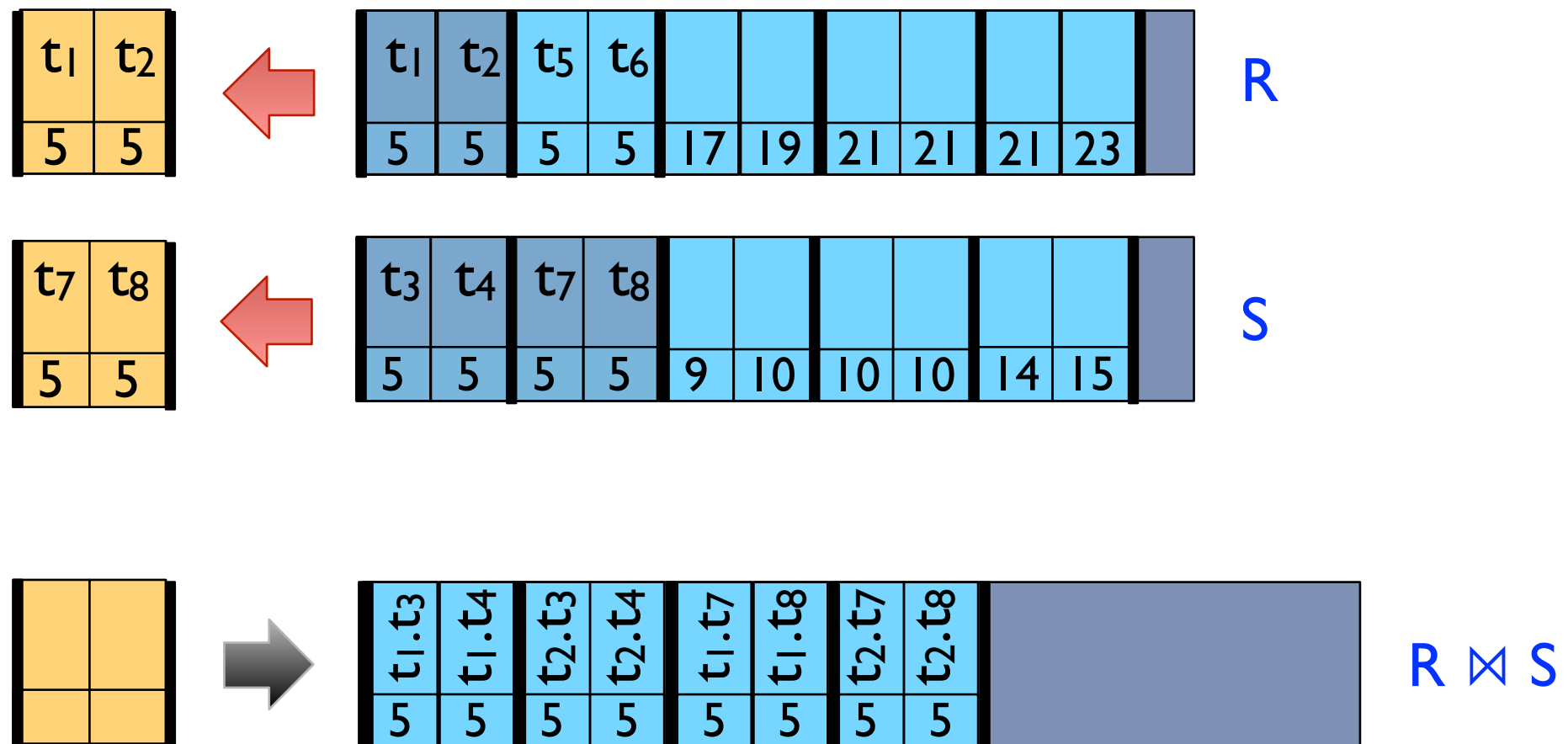
- ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
- ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas :



Merge join

- Difficulté :

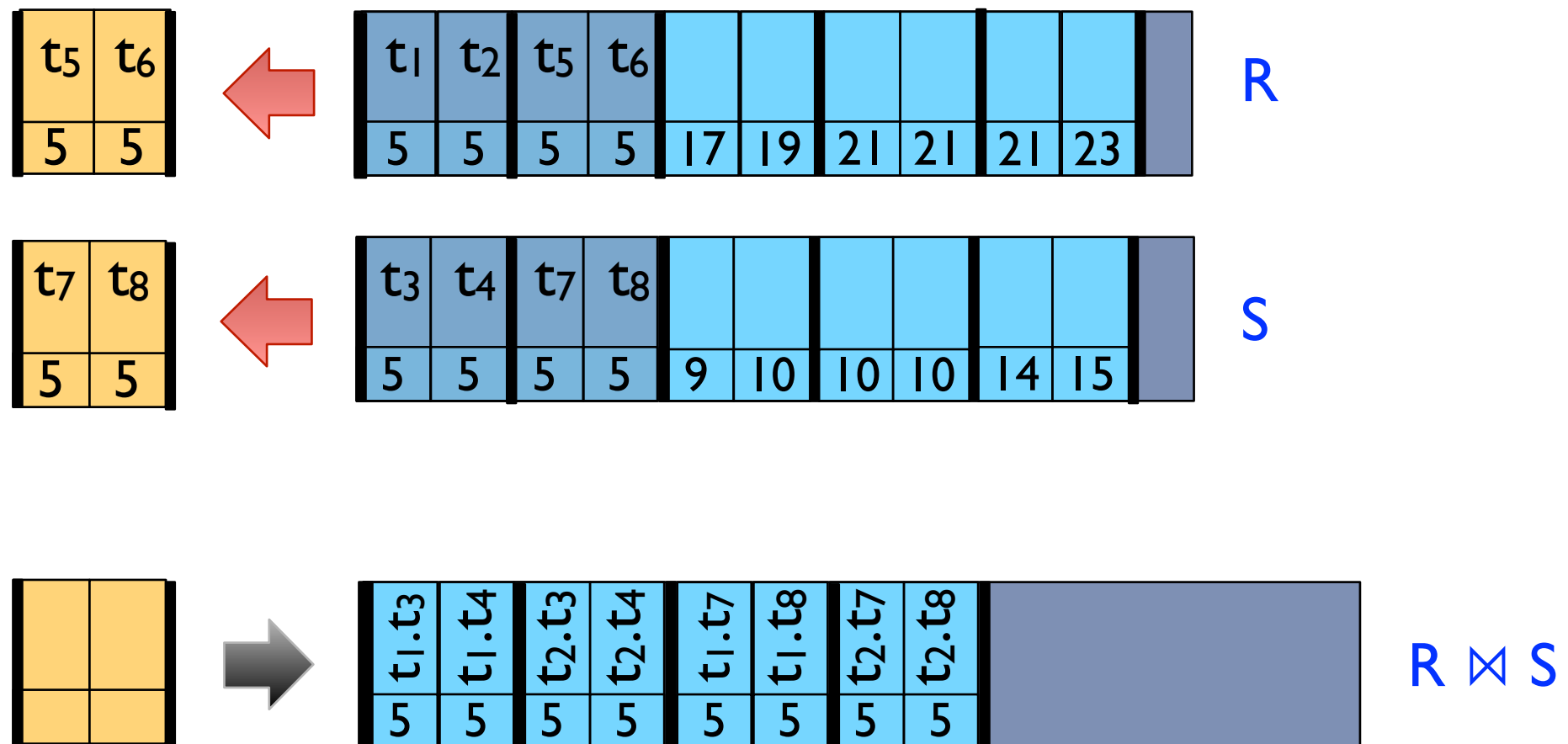
- ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
- ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas :



Merge join

- Difficulté :

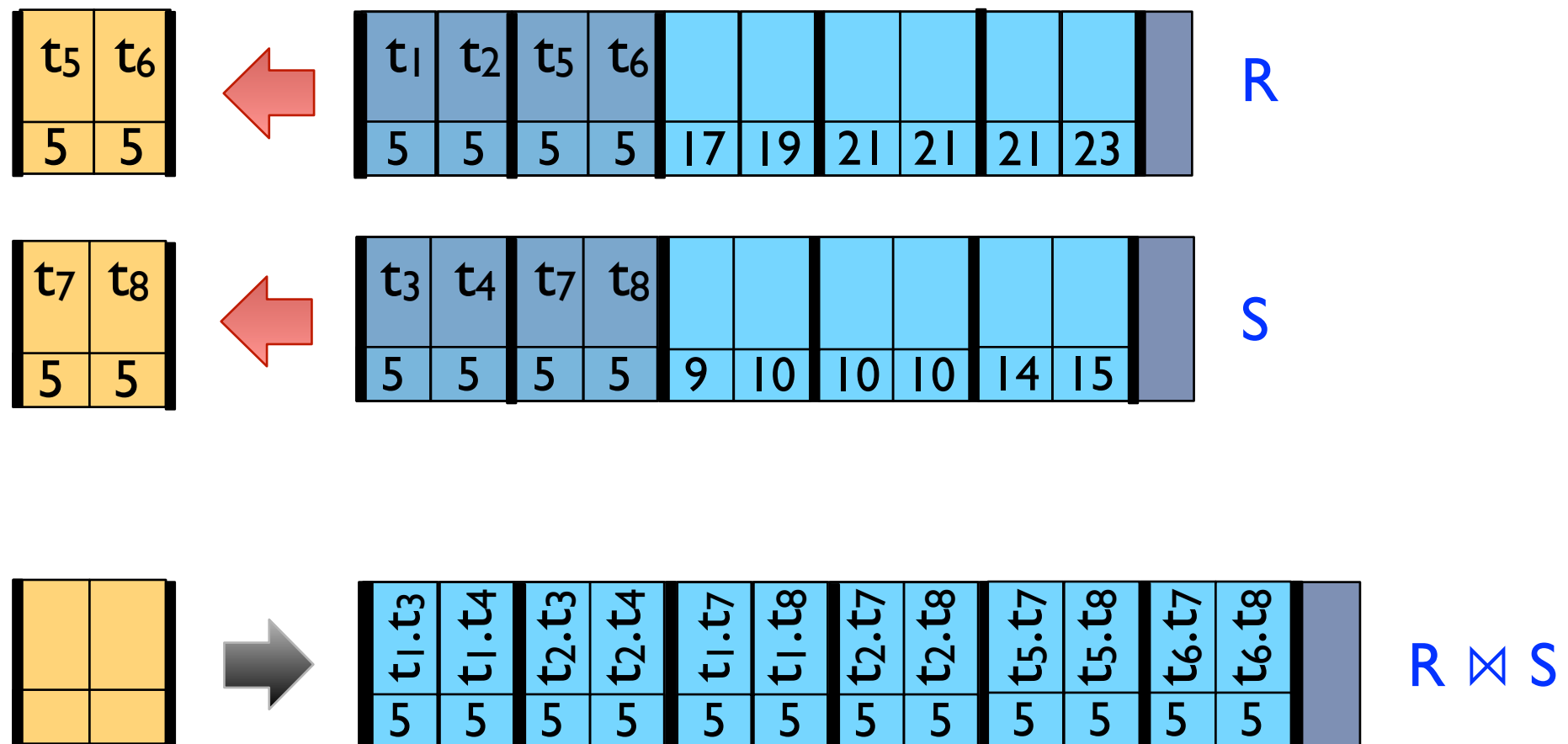
- ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
- ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas :



Merge join

- Difficulté :

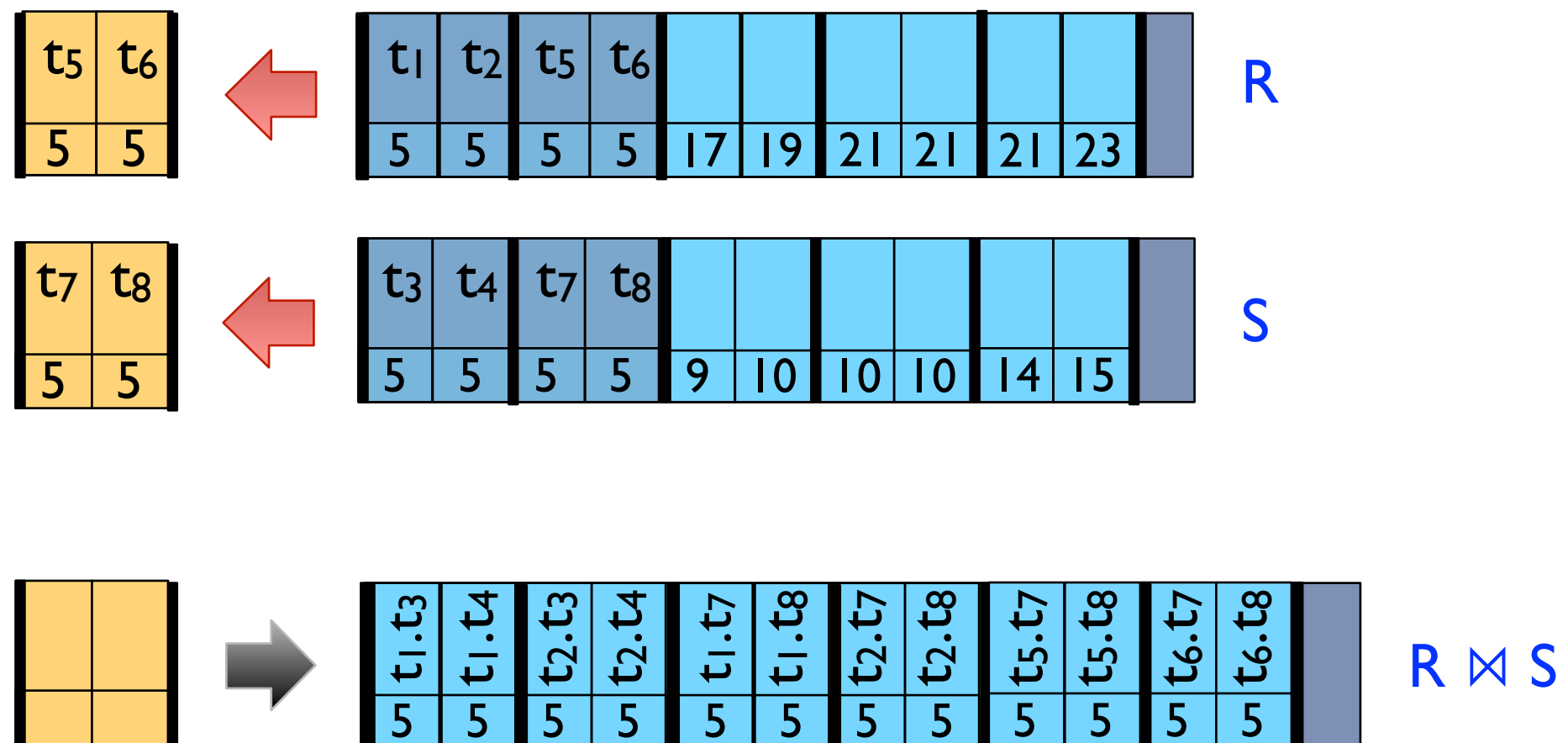
- ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
- ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas :



Merge join

- Difficulté :

- ▶ Il doit être possible de stocker en mémoire principale le nombre maximal de tuples de R (ou de S) ayant la même valeur de $A_1..A_k$
- ▶ Si ce n'était pas le cas, une passe sur R et S ne suffirait pas :



- Le premier bloc de S doit être relu pour compléter la jointure sur $A=5$
 \Rightarrow une seule passe sur R et S ne suffit pas dans cet exemple

Merge join

- Conditions d'applicabilité :
 - $\theta: R.A_1 = S.A_1, \dots, R.A_n = S.A_k$
 - ▶ l'ensemble maximale de tuples de R avec la même valeur de $A_1..A_k$ peut être stocké en mémoire principale
- Merge join :
 - ▶ une variante de l'algorithme de fusion qui utilise un “buffer de jointure” en mémoire principale pour stocker l'ensemble maximale de tuples de R avec la même valeur de $A_1..A_k$

Merge join - coût

- Coût

- ▶ avec les assumptions faites, une seule passe sur R et S est suffisante

$$B_R + B_S$$

- ▶ auquel s'ajoute

1) le coût du tri de R et S si elle ne sont pas déjà triées

2) $B_{R \bowtie S}$ si le résultat est matérialisé

- Variante

S'il existe des valeurs de $A_1..A_k$ tels que l'ensemble des tuples de R ayant ces valeurs ne peut pas être stocké en mémoire principale

- ▶ *merge join* avec

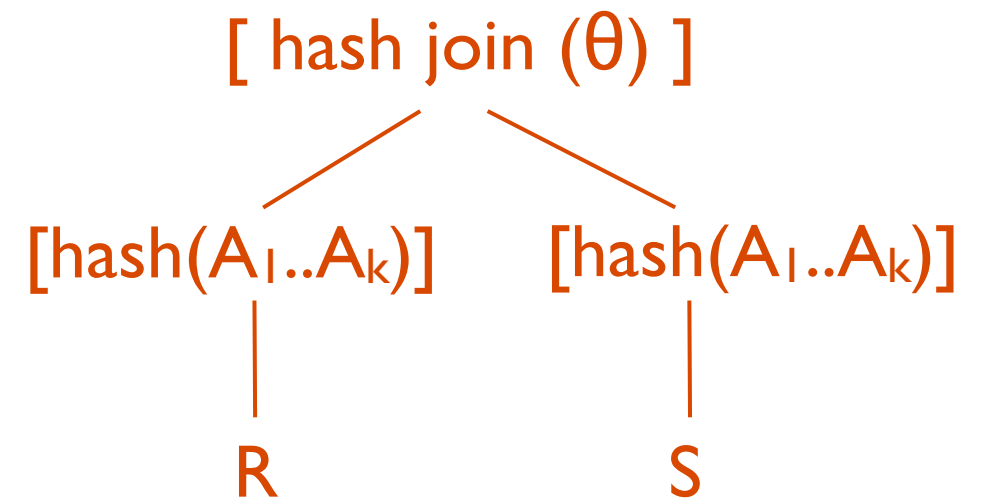
- ▶ *block nested loop* pour les jointures sur ces valeurs de $A_1..A_k$

- Le coût de cette variante se rapproche du cout de *block nested loop*, selon la taille des ensembles de tuples de R qu'on ne peut pas stocker

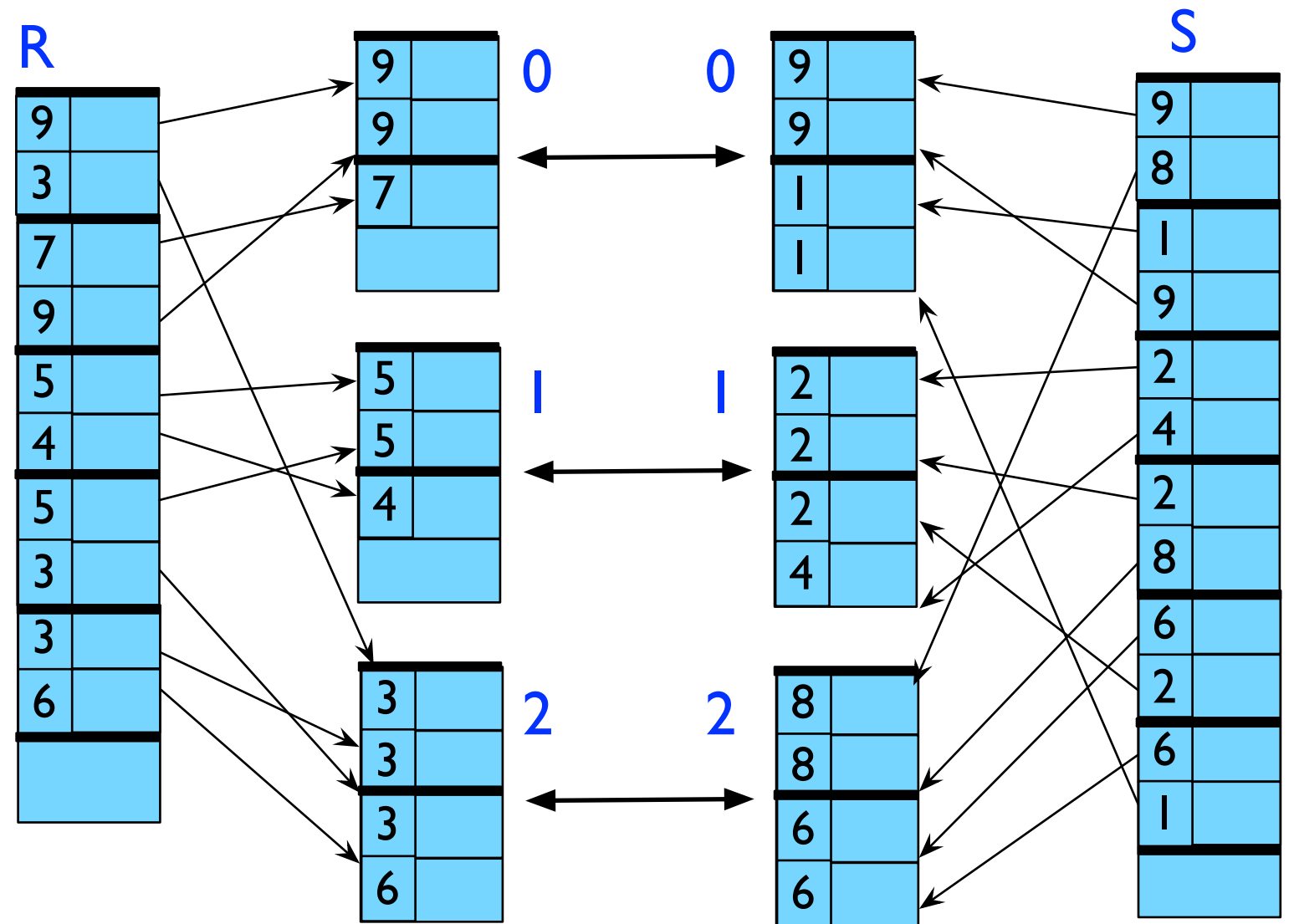
Merge join “hybrid”

- Si un B+-tree sur $A_1..A_k$ est disponible sur R et/ou S
 - ▶ accéder aux tuples en parcourant l’index permet d’obtenir les tuples triés sans un tri additionnel
 - ▶ la fusion peut être faite directement sur les éléments des index, pour accéder au tuples seulement ensuite

Hash join



$\theta: R.A_1 = S.A_1, \dots R.A_k = S.A_k$



Idée :

- Hacher R et S sur A_1, \dots, A_k **avec la même fonction de hachage** en p partitions
- Les tuples de R et de S avec la même valeur de $A_1 \dots A_k$ sont dans des partitions correspondantes
- Jointure uniquement entre partitions correspondantes :

$$R \bowtie S = R_0 \bowtie S_0 \cup R_1 \bowtie S_1 \cup \dots \cup R_p \bowtie S_p$$

Hash join - conditions d'applicabilité

- Avantage (par rapport à block nested loop) : pour chaque bloc de R seulement des blocs de S susceptibles de participer à la jointure sont parcourus
- Pour avoir un réel bénéfice :
 - ▶ partitions petites, quand c'est possible
 - ▶ bonne fonction de hachage
- Conditions idéales d'applicabilité :
 - ▶ Pour au moins une des relations (ex. S) chaque partition rentre en mémoire principale

Hash join

- Phase de hachage
 - ▶ hachage récursif de R et S jusqu'à ce que chaque partition de S rentre en mémoire principale
- Phase de jointure :

pour chaque i de 1 à p (= nombre de partitions de S)

 - charger S_i en mémoire principale
 - calculer $R_i \bowtie S_i$ par jointure à une passe (parcours de R_i , accès à S_i en mémoire principale) et l'ajouter au résultat
- Parfois il est impossible que chaque partition de S rentre en mémoire principale (beaucoup de tuples de S ont la même valeur des attributs de jointure)
 - ▶ $R_i \bowtie S_i$ par block nested loop pour les S_i qui ne peuvent pas être chargés en mémoire principale

Hash join - coût

- Soit M le nombre de blocs disponibles en mémoire principale
- À chaque itération, $M-1$ blocs pour charger S_i en mémoire et un bloc pour lire R_i
 - ▶ \Rightarrow chaque partitions de S_i doit occuper au plus $M-1$ blocs
- En supposant hachage uniforme, la phase de hachage doit produire au moins $N = \lceil B_S / (M-1) \rceil$ partitions
 - ▶ \Rightarrow coût de la phase de hachage : $2 B_S \cdot \lceil \log_{M-1} N \rceil + 2 B_R \cdot \lceil \log_{M-1} N \rceil =$
 $2 (B_S + B_R) \cdot \lceil \log_{M-1} B_S - 1 \rceil$
- Coût de la phase de jointure : $B_R + B_S$
- **Coût** (sans prendre en compte le coût de l'écriture du résultat) :

$$2 (B_S + B_R) \cdot \lceil \log_{M-1} B_S - 1 \rceil + B_R + B_S$$

Implémentation d'autres opérateurs

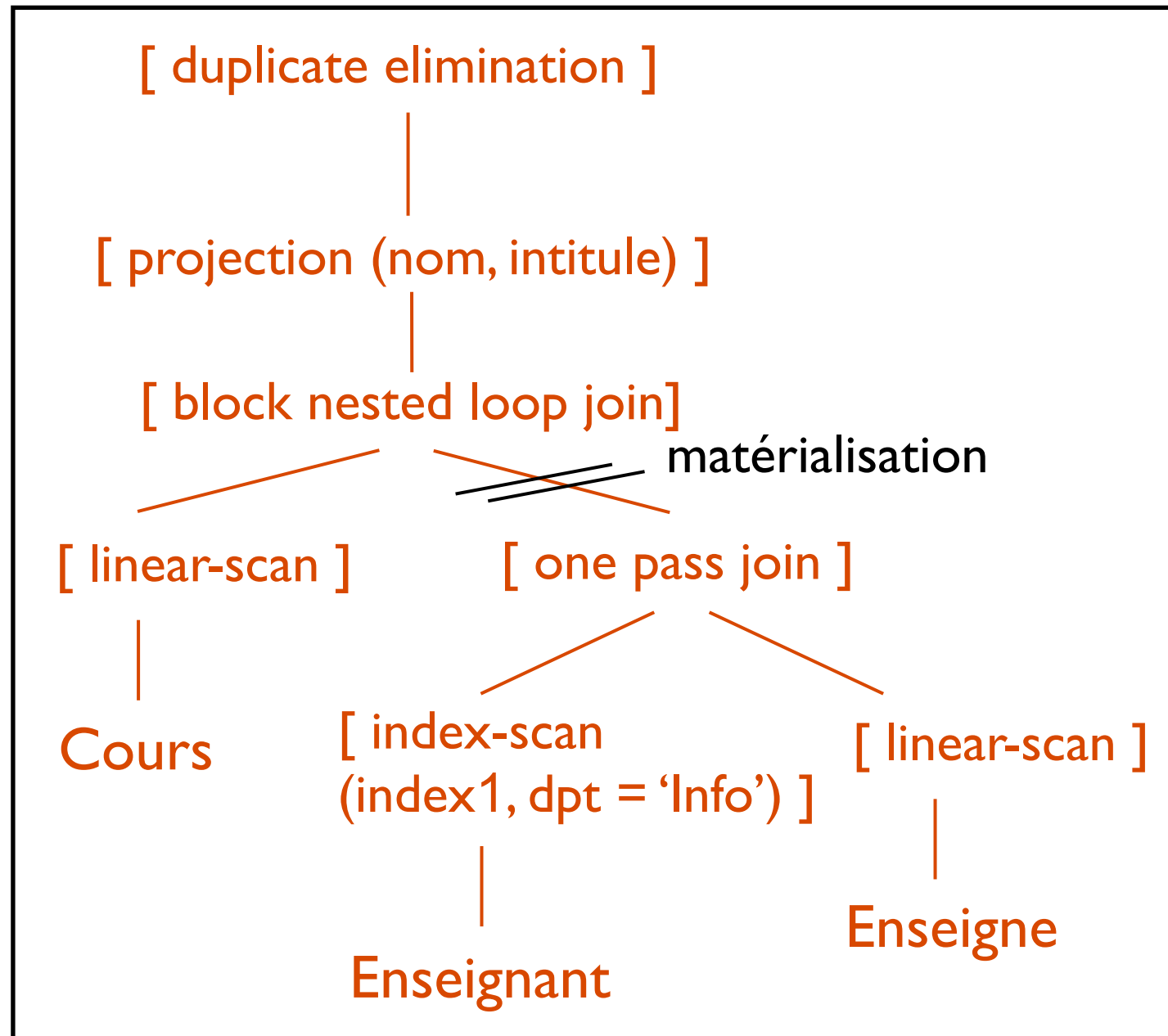
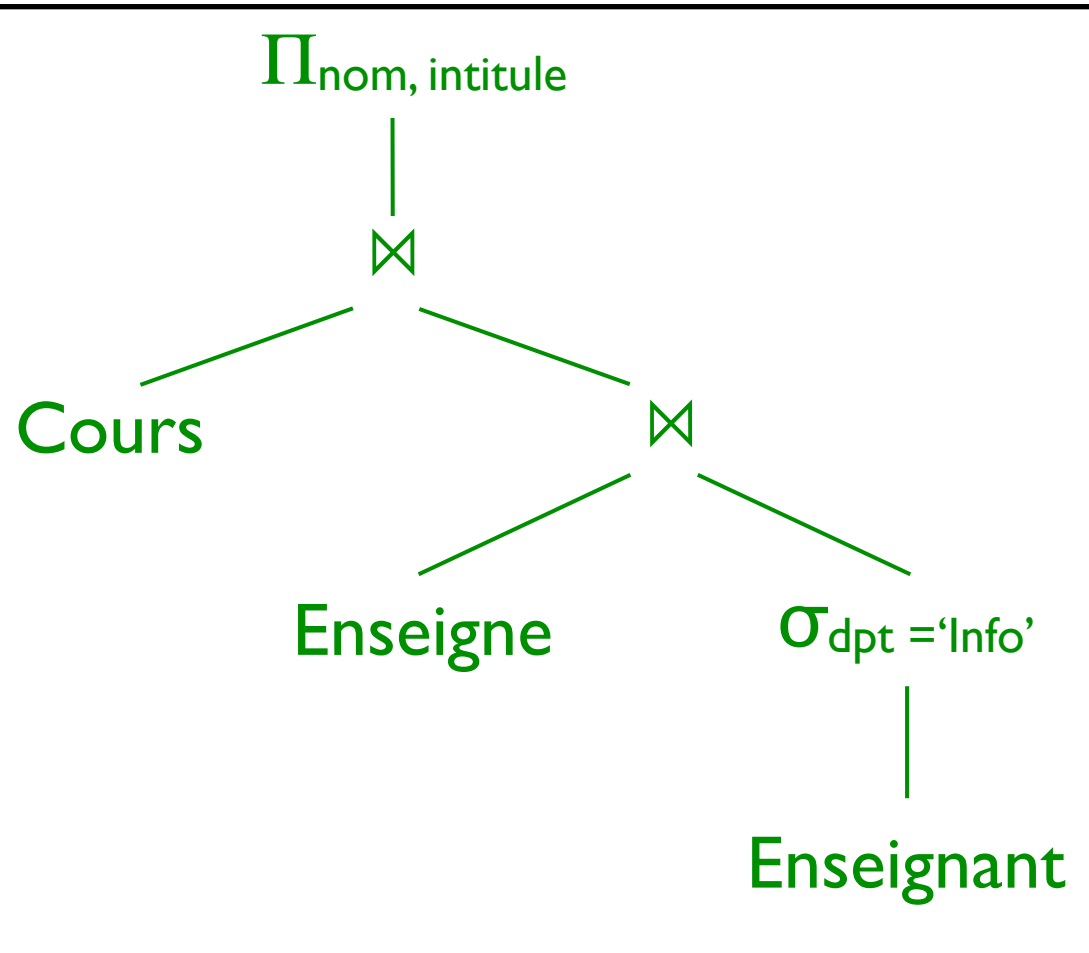
- Jointure avec plusieurs conditions
- Opérateurs ensemblistes
- Aggregation

Pas abordés

Plan

- Introduction
- Plans d'exécution logiques
- Plans d'exécution physiques
 - ▶ Implémentations des opérateurs algébriques
 - ▶ Implémentation des expressions algébriques
- Evaluation des plans d'exécution
- Optimisation

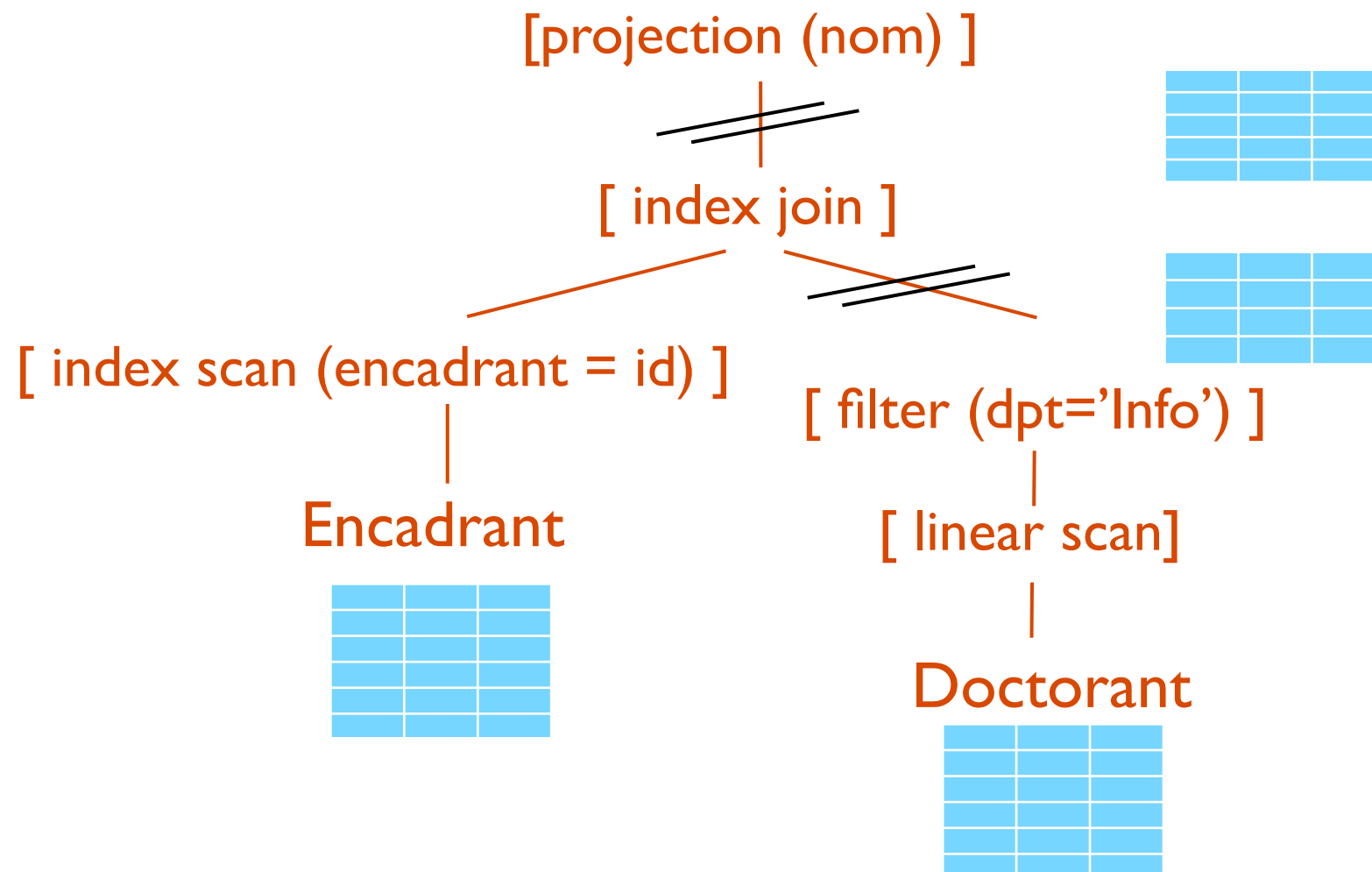
Implémentation des expressions algébriques



- Plan logique : composition de plusieurs opérateurs algébriques (expression algébrique)
- Plan d'exécution physique :
 - 1) comment implémenter chaque opérateur
 - 2) comment composer les implémentations des différents opérateurs pour obtenir une implémentation de l'expression algébrique

Implémentation des expressions algébriques

- Deux approches principaux (ou combinaison des deux)
 - ▶ **Materialisation**
 - les sous-expressions d'un plan physique sont évaluées *bottom-up*
 - les résultats de chaque sous-expression (noeud de l'arbre) sont matérialisés pour être utilisés comme argument de l'opérateur au niveau supérieur

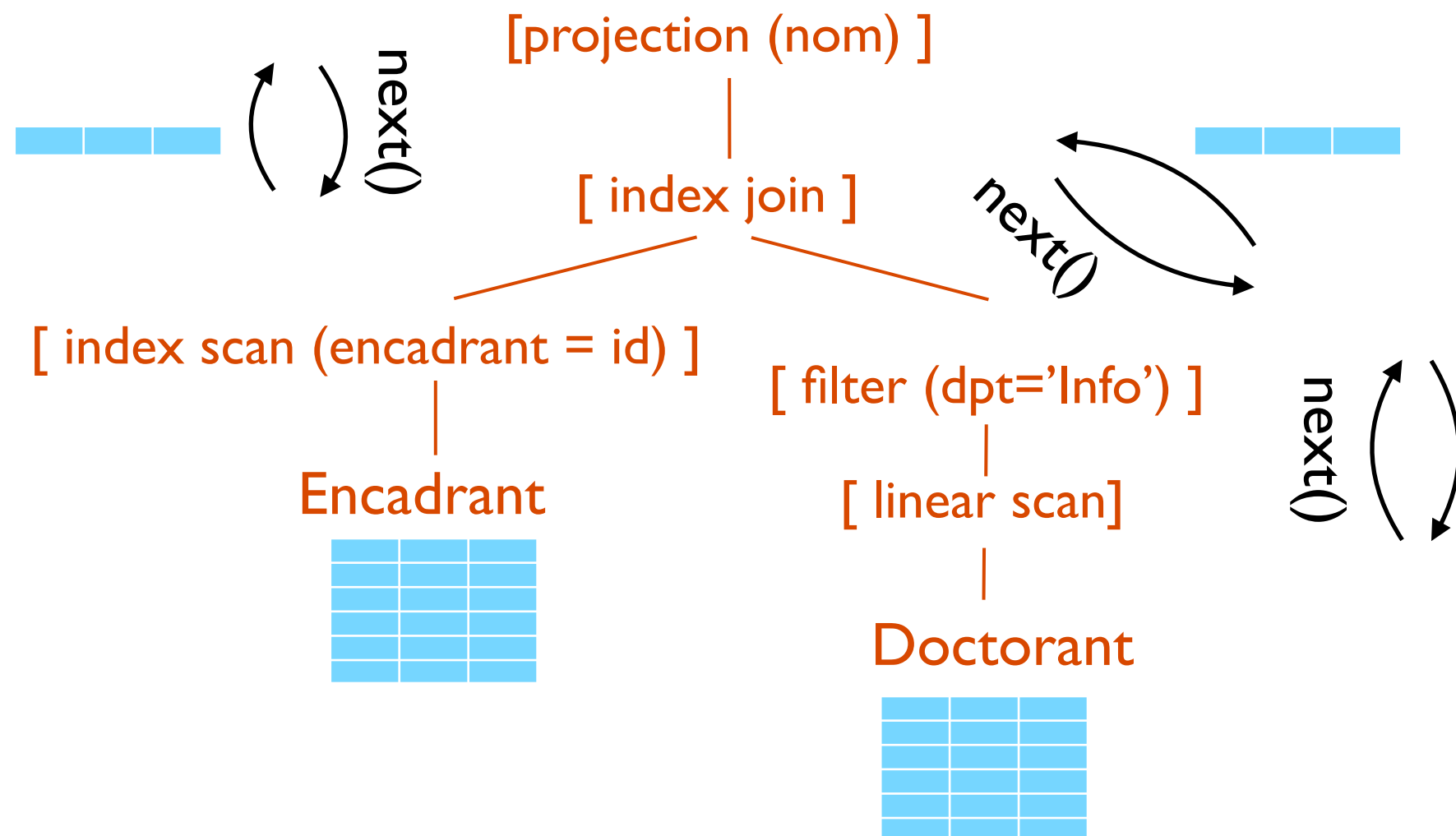


Le coût de l'écriture des résultats intermédiaires s'ajoute au coût des opérateurs

Implémentation des expressions algébriques

► Pipelining

- chaque nouveau tuple calculé par une sous-expression est passé immédiatement à l'opérateur du niveau supérieur pour être traité
- en général *top-down*, i.e chaque opérateur, pour produire un nouveau tuple “demande” un (ou plusieurs) nouveaux tuples à ses arguments



Pipelining des opérateurs

- Evite la matérialisation et donc le coût d'écriture des résultats intermédiaires
- Permet de produire les tuples du résultat de façon progressive, dès qu'il sont disponibles
- En général obtenue en implémentant les opérateurs comme itérateurs
- Un **itérateur** est une implémentation d'un opérateur conforme à l'interface suivante :
 - ▶ **open()** : opérations préliminaires pour démarrer le calcul du résultat
 - ▶ **next()** : renvoie le prochain tuple du résultat et modifie l'état interne de l'itérateur pour tenir trace du dernier tuple produit
 - ▶ **close()** : termine le calcul du résultat
- **Pipeline sur demande** : l'opération next() d'un itérateur appelle next() sur les itérateurs qui implémentent ses arguments

Exemples d'itérateurs

- Itérateur pour implementer $\text{filter}(\theta)$ (i.e. la sélection) :

- ▶ $\text{open}()$:

$\text{op.open}()$

- ▶ $\text{next}()$:

$t = \text{op.next}()$

tant que ($t \neq \text{null}$ et $t \neq \theta$)

$t = \text{op.next}()$

return t

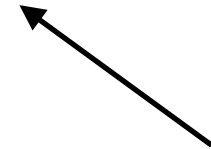
- ▶ $\text{close}()$:

$\text{op.close}()$

[filter (θ)]

|

[op]



un autre opérateur :
e.g. index-join

Exemples d'itérateurs

- Itérateur pour implementer linear-scan :

- ▶ état interne :

pointeur **t** au prochain enregistrement de **R**

- ▶ **open ()** :

initialiser **t** au premier enregistrement de **R**

- ▶ **next()** :

r = enregistrement pointé par **t**

avancer **t** au prochain enregistrement

renvoyer **r**

- ▶ **close()** :

détruire **t**

[linear scan]

|
R

Exemples d'itérateurs

- Itérateur pour merge-join :

- ▶ état interne :

pointeurs t_S et t_R aux fichiers triés

- ▶ open () :

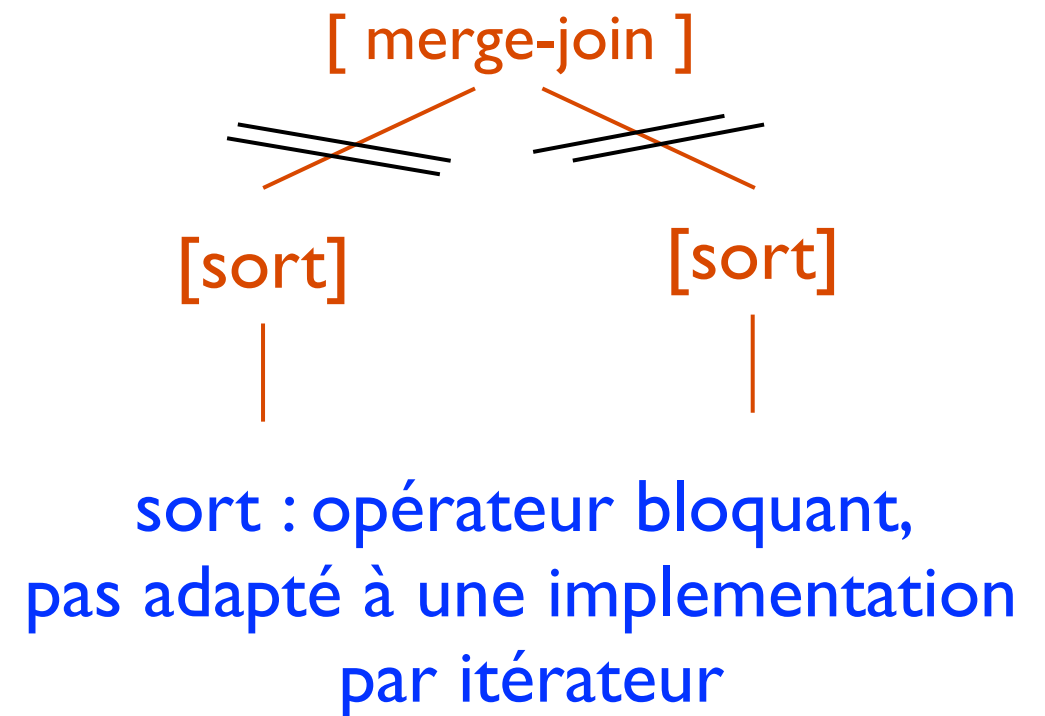
appelle [sort] sur ses arguments

initialise t_R et t_S

- ▶ next() :

avancer t_S et t_R jusqu'aux prochains deux tuples qui peuvent être joints,
et les joindre

retourner le tuple obtenu



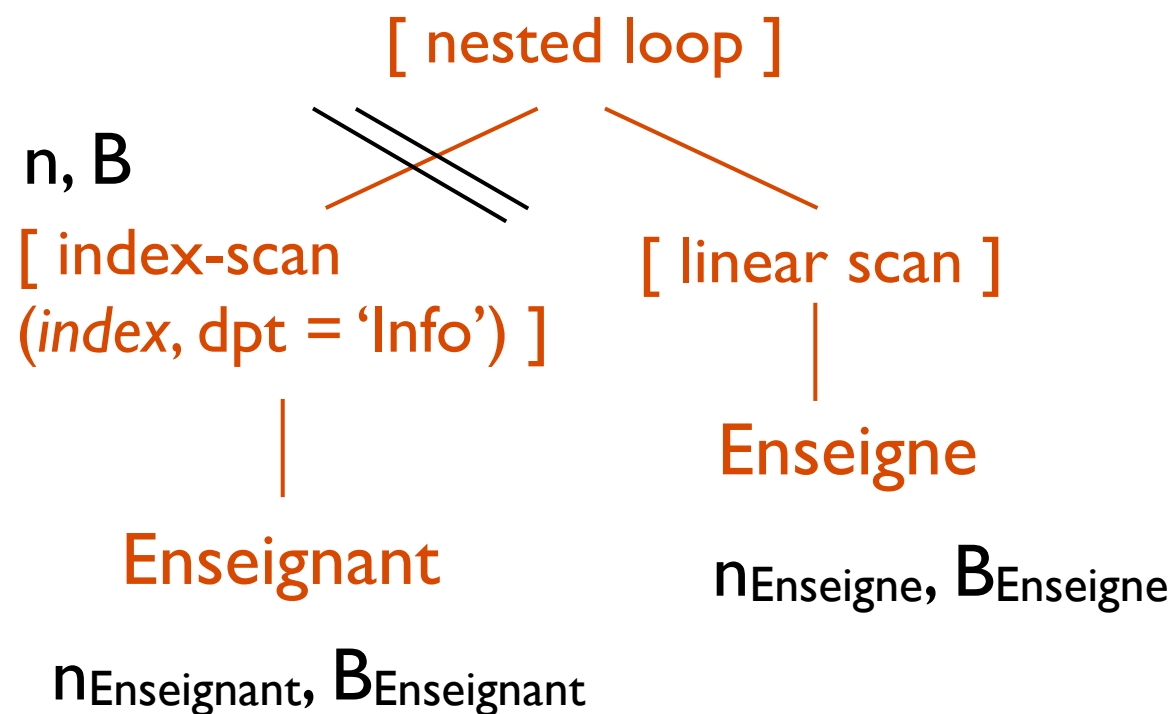
Plan

- Introduction
- Plans d'exécution logiques
- Plans d'exécution physiques
 - ▶ Implémentations des opérateurs algébriques
 - ▶ Implémentation des expressions algébriques
- Evaluation des plans d'exécution
- Optimisation

Evaluation des plans d'exécution

- Pour choisir le “meilleur” plan d'exécution :
 - ▶ évaluer les coûts de différents plans
- **Mesure de coût** d'un plan physique
 - ▶ fonctions de coût des algorithmes choisis appliquées à une estimation de la taille des arguments

Exemple



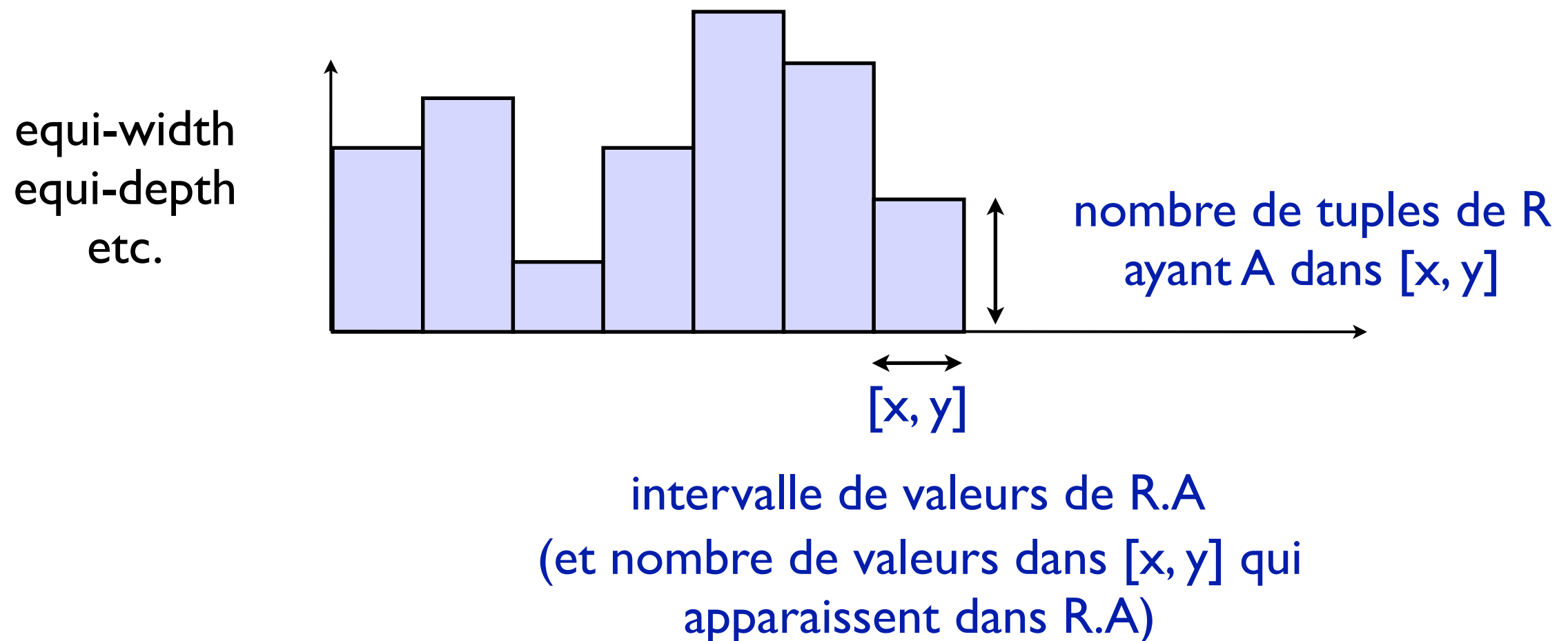
$$\underbrace{B + n \cdot B_{\text{Enseigne}}}_{\text{coût du nested loop (sauf écriture du résultat)}} + \underbrace{h_{\text{index}} + n_{\text{Enseignant}, \text{dpt} = \text{Info}} + B}_{\text{coût de la sélection (index-scan) avec matérialisation du résultat}}$$

Calcul du coût d'un plan d'exécution

- Pour mesurer le coût d'un plan d'exécution on a donc besoin de connaître :
 - ▶ le nombre de tuples et de blocs occupés par chaque relation dans la BD
 - ▶ la tailles des résultats intermédiaires de la requête (e.g. le résultat de la sélection dans l'exemple précédent)
- On doit pouvoir estimer ces valeur efficacement sans executer la requête
- À cette fin, **le catalogue de la BD** stocke des estimations pour chaque relation

Statistiques et catalogue

- **Le catalogue de la BD** stocke des estimations pour chaque relation R (recalculées de temps en temps):
 - ▶ n_R, B_R : le nombre de tuples et de blocs occupés par R
 - ▶ f_R : le nombre de tuples de R qui peuvent être stockés dans un bloc
 - ▶ $V(A, R)$: le nombre de valeurs distincts de l'attribut A dans la relation R
 - ▶ des **histogrammes** pour la distribution des valeurs de certains attributs



Estimation de la taille des résultats intermédiaires

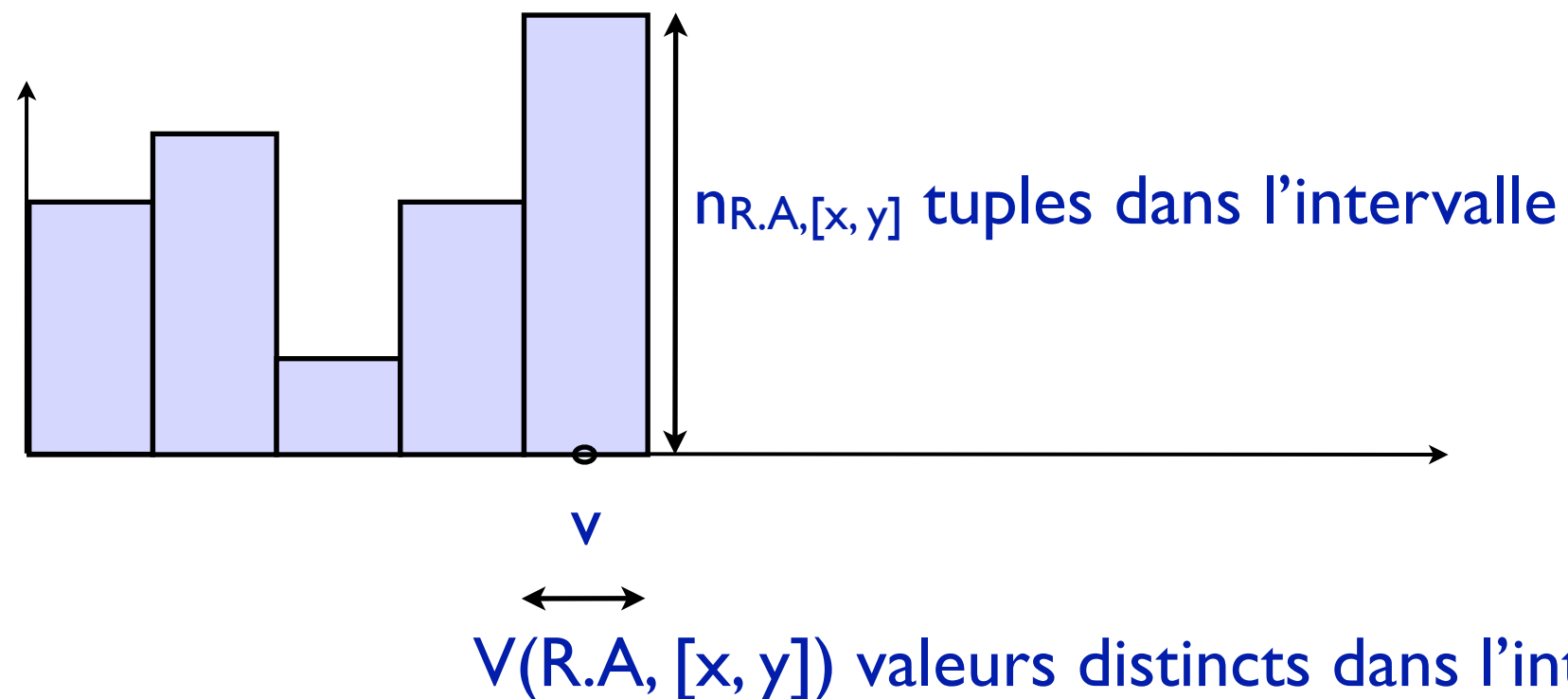
- Les informations dans le catalogue (et l'utilisation de certaines heuristiques) en général suffisent pour estimer la tailles de résultats intermédiaires
- **Remarque.** La taille des résultats intermédiaires ne dépend pas des algorithmes choisis pour chaque opérateur
 - ▶ **estimée à partir du plan logique**
- Deux exemples
 - ▶ estimation de la taille d'une sélection
 - ▶ estimation de la taille d'une jointure naturelle
- Pour plus d'heuristiques d'estimation de la taille des résultats d'expressions algébriques, voir les livres conseillés

Exemple : estimation de la taille d'une sélection

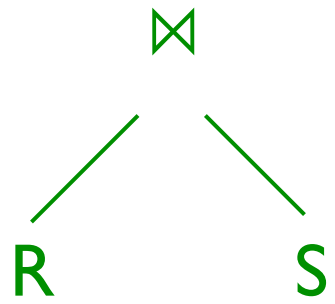
Nombre de tuples dans le résultat :

$\sigma_{A=v}$
|
R

- en absence d'histogramme sur A :
 - ▶ $n_R / V(A, R)$ (hypothèse de distribution uniforme des valeurs)
- si un histogramme sur A est disponible
 - ▶ $n_{R.A, [x,y]} / V(R.A, [x,y])$



Exemple : estimation de la taille d'une jointure naturelle



- Nombre de tuples dans le résultat
 - ▶ $n_R \cdot n_S$, si R et S n'ont pas d'attributs en communs
 - ▶ **au plus** n_R si les attributs en commun sont une clef pour S
 - ▶ n_R si les attributs en commun sont une clef étrangère de R, qui fait référence à S
 - ▶ $\frac{n_R n_S}{\max \{V(A, S), V(A, R)\}}$ si les attributs en commun {A} ne sont une clef ni de R ni de S
- sous hypothèse de distribution uniforme des valeurs, chaque tuple de R se joint à $n_S / V(A, S)$ tuples de S
- mais aussi chaque tuple de S se joint à $n_R / V(A, R)$ tuples de R
- \Rightarrow on obtient deux estimations $n_R n_S / V(A, S)$ et $n_R n_S / V(A, R)$
- on prend la plus petite pour tenir compte de tuples “dangling”

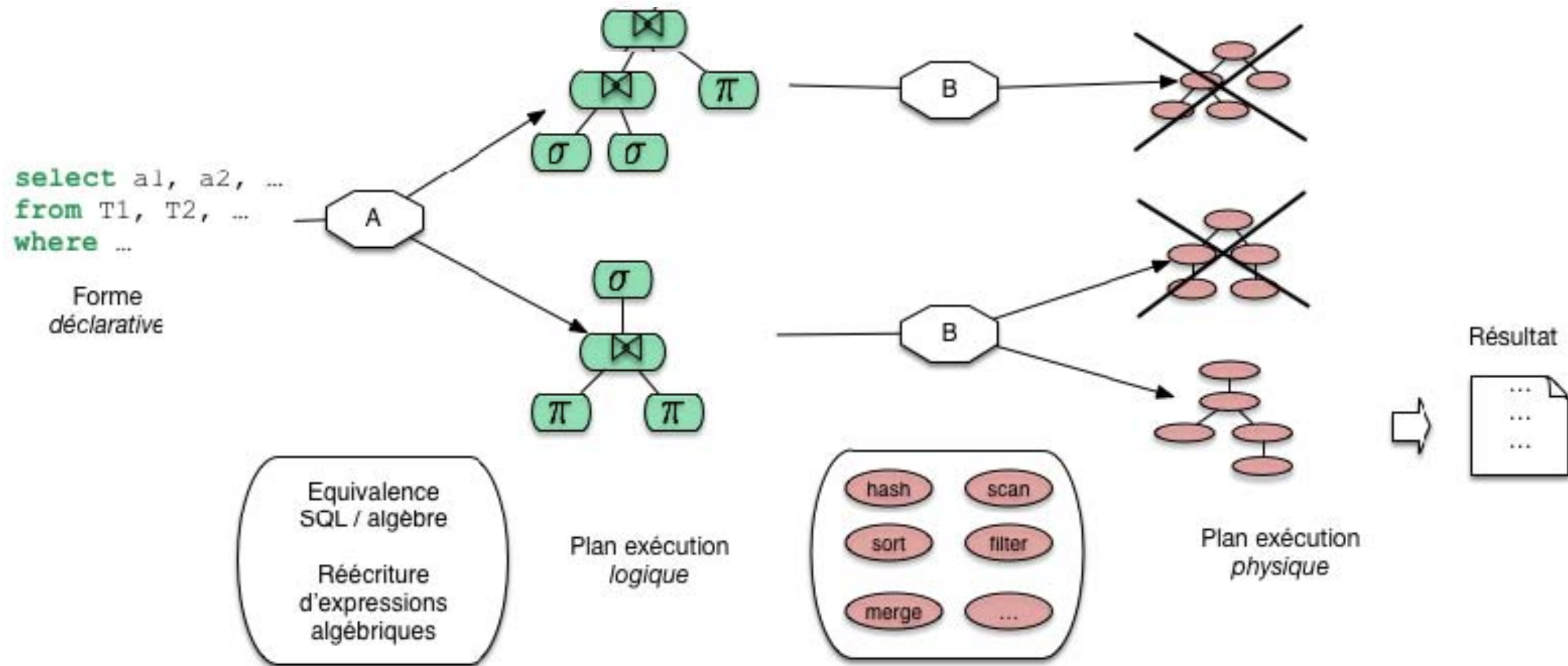
Exemple : estimation de la taille d'une jointure naturelle

- Si les attributs en commun $\{A\}$ ne sont une clef ni de R ni de S ,
- et des histogrammes sont disponibles pour $R.A$ et $S.A$
(par simplicité définis sur les mêmes intervalles)
 - ▶ Chaque tuple de R avec $A \in [x, y]$ se joint avec $n_{S.A, [x, y]} / V(S.A, [x, y])$ tuples de S
 - ▶ il y a $n_{R.A, [x, y]}$ tuples de R avec $A \in [x, y]$
 - ▶ \Rightarrow première estimation de la taille du résultat :
$$\sum_{[x, y]} \frac{n_{R.A, [x, y]} n_{S.A, [x, y]}}{V(S.A, [x, y])}$$
 - ▶ la deuxième estimation est symétrique ($R.A$ et $S.A$ échangés)
 - ▶ prendre la plus petite

Plan

- Introduction
- Plans d'exécution logiques
- Plans d'exécution physiques
 - ▶ Implémentations des opérateurs algébriques
 - ▶ Implémentation des expressions algébriques
- Evaluation des plans d'exécution
- Optimisation

Optimisation de requêtes



• Principe

- une requête donne lieu à plusieurs plan logiques possibles (étape A)
- chaque plan logique donne lieu à plusieurs plan physiques possibles (étape B)
- Le système les évalue et choisit le “meilleur” plan physique
- celui là sera exécuté pour obtenir le résultat de la requête

Optimisation : énumération et heuristiques

- Couramment les systèmes implémentent une **combinaison** d'heuristiques et de techniques pour énumérer les plans d'exécution
 - ▶ **Enumération** : On considère de façon plus ou moins exhaustive plusieurs plans et on évalue leur coût pour choisir le meilleur
 - ▶ **Heuristiques** :
 - On transforme un plan initial par application directionnelle de certaines règles heuristiques (à la fois au niveau du plan logique et physique) sans évaluer les alternatives
 - idée sous-jacente : dans la plus part des cas les transformations fondées sur les heuristiques aboutissent à un meilleur plan d'execution
- **Typiquement** :
 - ▶ Heuristiques tant que c'est possible
 - ▶ Enumération pour les choix non couverts par les heuristiques
 - ▶ Enumération pour certain choix critiques comme l'ordre des jointures

Enumération des plans d'exécution

- À partir d'un plan logique initial l'algorithme applique toutes les transformations possibles et obtient tous les plans logiques et physiques équivalents
- Ensuite il faut évaluer le coût des plans physiques générés (cf. section Evaluation des plan d'exécution) et choisir celui à moindre coût
- Algorithme très coûteux!
- Pas implémenté tel quel dans les systèmes
- Plusieurs techniques pour rendre plus efficace la recherche dans l'espace des plans possibles :
 - ▶ programmation dynamique
 - ▶ “branch and bound”
 - ▶ optima locaux

(Pas abordées)

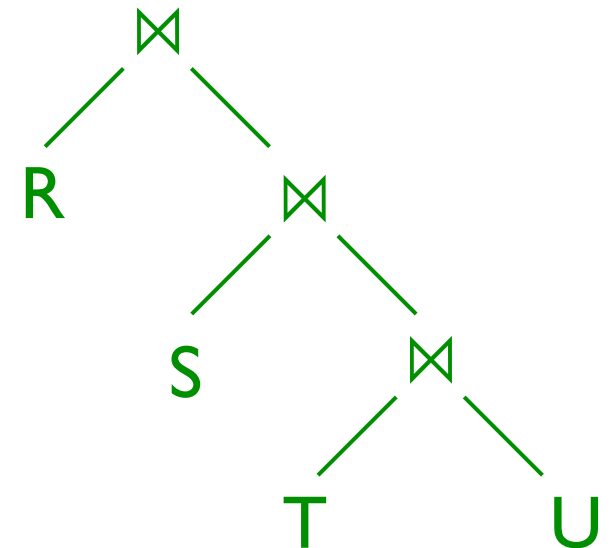
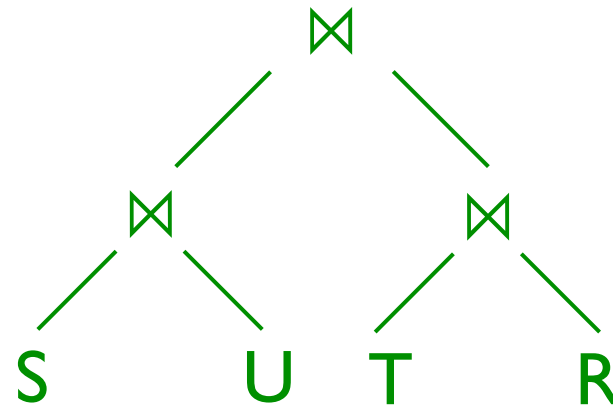
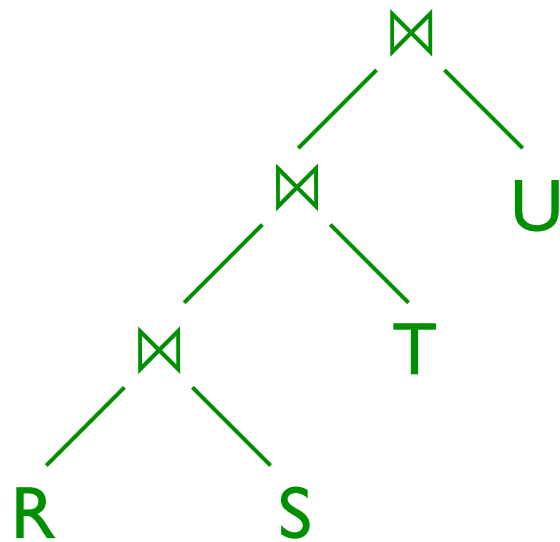
Enumération des plans : ordre des jointures

- Typiquement l'énumération est utilisée pour choisir l'ordre des jointures

$R \bowtie S \bowtie T \bowtie U$

- Règles d'associativité et commutativité pour la jointure \Rightarrow différents plans logiques possibles

Ex. :



etc.

Enumération des plans : ordre des jointures

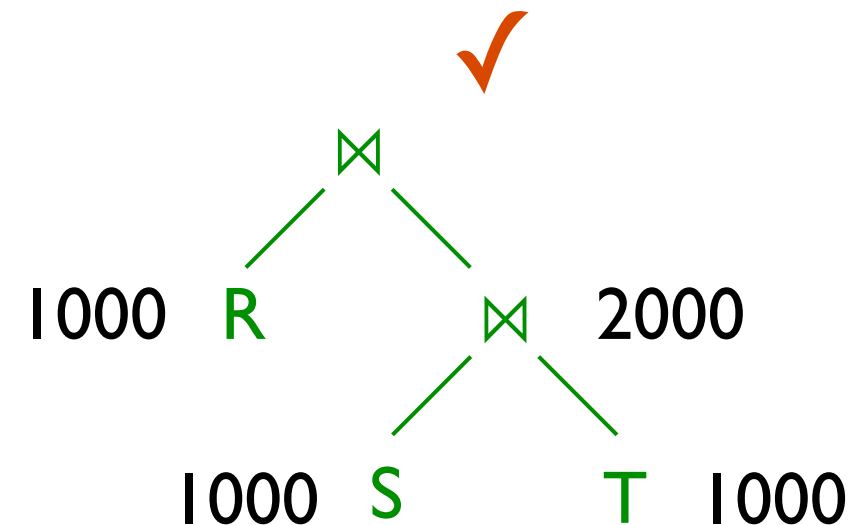
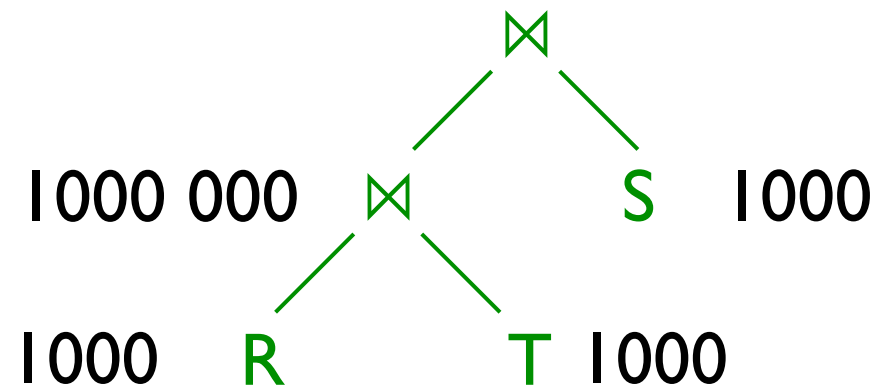
- Impact de l'ordre des jointures sur le coût :
 - ▶ résultats intermédiaires de taille différente
 - ▶ algorithmes de jointures asymétriques
 - ▶ possibilité d'utiliser des index
 - ▶ efficacité des implémentations par itérateur

Enumération des plans : ordre des jointures

- **Example I**

$R(a, b)$	$S(b, c)$	$T(c, d)$
$n_R = 1000$	$n_S = 1000$	$n_T = 1000$
$V(a, R) = 100$		
$V(b, R) = 200$	$V(b, S) = 200$	
	$V(c, S) = 500$	$V(c, T) = 20$
		$V(d, T) = 50$

Tailles des résultats intermédiaires :



Enumération des plans : ordre des jointures

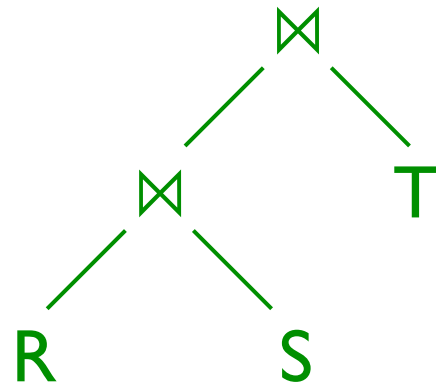
- **Example 2**

$R(a, b)$

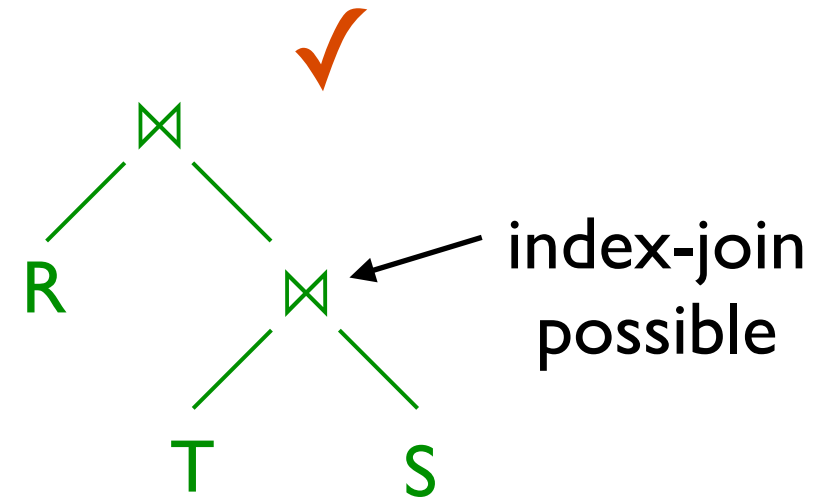
$S(b, c)$

$T(c, d)$

index sur $S.c$



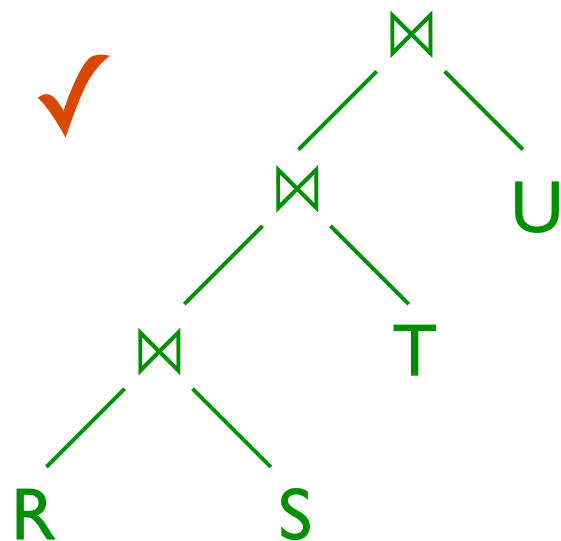
pas de index-join



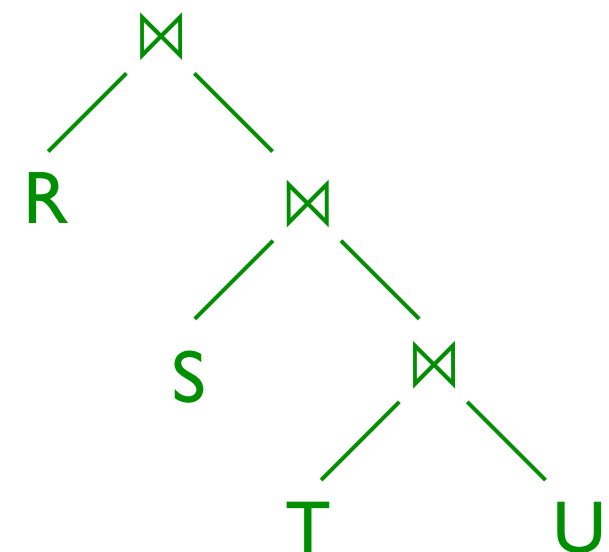
Enumération des plans : ordre des jointures

- **Example 3.** En supposant

- ▶ pipelining
- ▶ block-nested-loop join pour chaque jointure (à gauche la relation de la boucle externe)



l'itérateur à la racine (et à chaque niveau) fait une seule passe sur le résultat intermédiaire à gauche et plusieurs passes sur la relation stockée
⇒ résultats intermédiaires calculés une seule fois

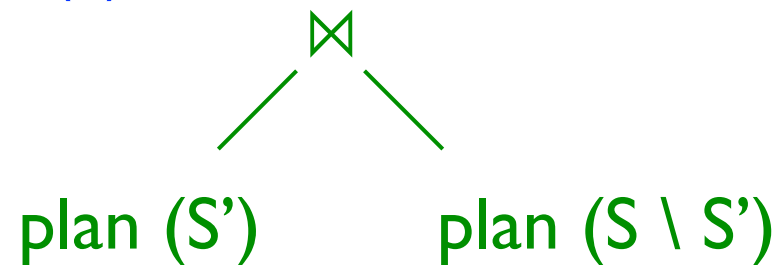


l'itérateur à la racine (et à chaque niveau) fait une seule passe sur la relation stockée à gauche et plusieurs passes sur le résultat intermédiaire à droite
⇒ résultats intermédiaires calculés plusieurs fois

Programmation dynamique pour optimiser l'ordre des jointures

- Utilise la sous-structure optimale du problème
 - ▶ Soit $S = \{R_1, \dots, R_n\}$. Chaque plan d'exécution pour $R_1 \bowtie \dots \bowtie R_n$ est de la forme

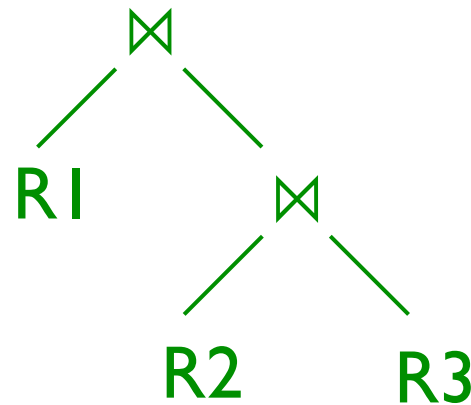
plan (S) :



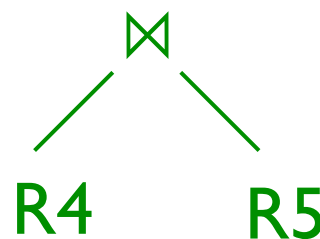
où $\text{plan}(S')$ est un plan pour la jointure d'un sous-ensemble non-vide S' de S

Programmation dynamique pour optimiser l'ordre des jointures

- Cela veut dire - exemple $S = \{R1, R2, R3, R4, R5\}$
 - ▶ Pour établir un plan pour joindre S je dois :
 - décider une **partition** initiale, par ex. $S' = \{R1, R2, R3\}$. $S \setminus S' = \{R4, R5\}$
 - Etablir un **plan pour joindre** $S' = \{R1, R2, R3\}$, par exemple

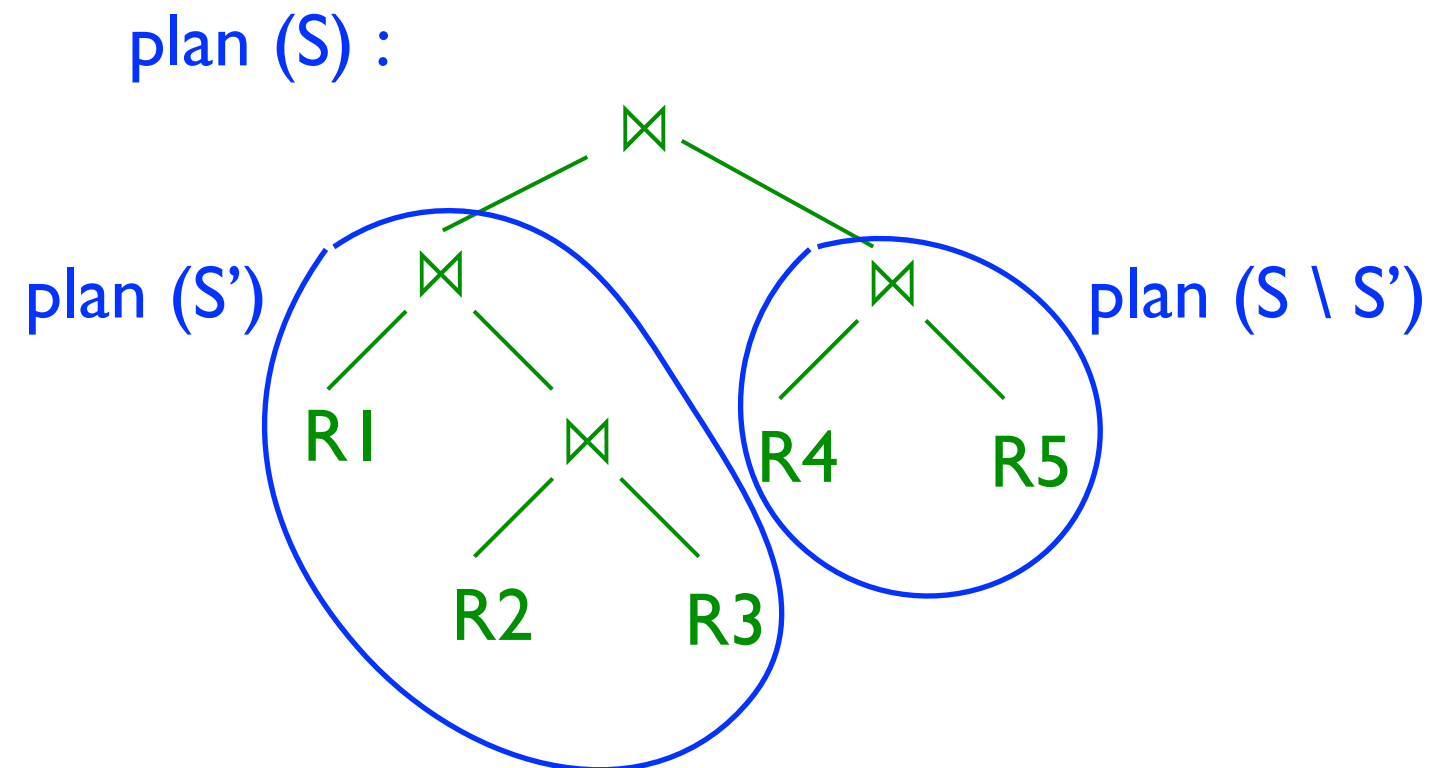


- Etablir un **plan pour joindre** $S \setminus S' = \{R4, R5\}$, par exemple :



Programmation dynamique pour optimiser l'ordre des jointures

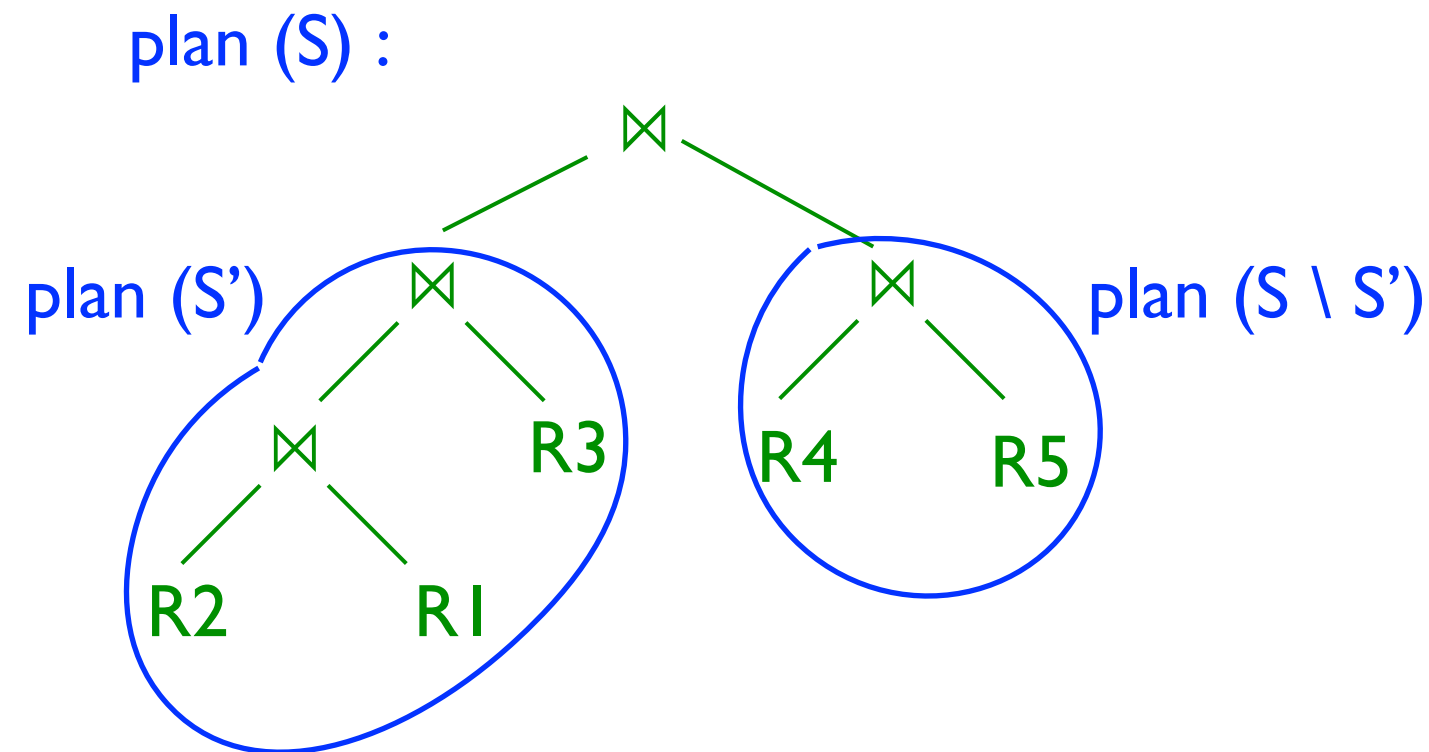
- Exemple suite
 - Le plan établi sera :



Programmation dynamique pour optimiser l'ordre des jointures

- Exemple suite

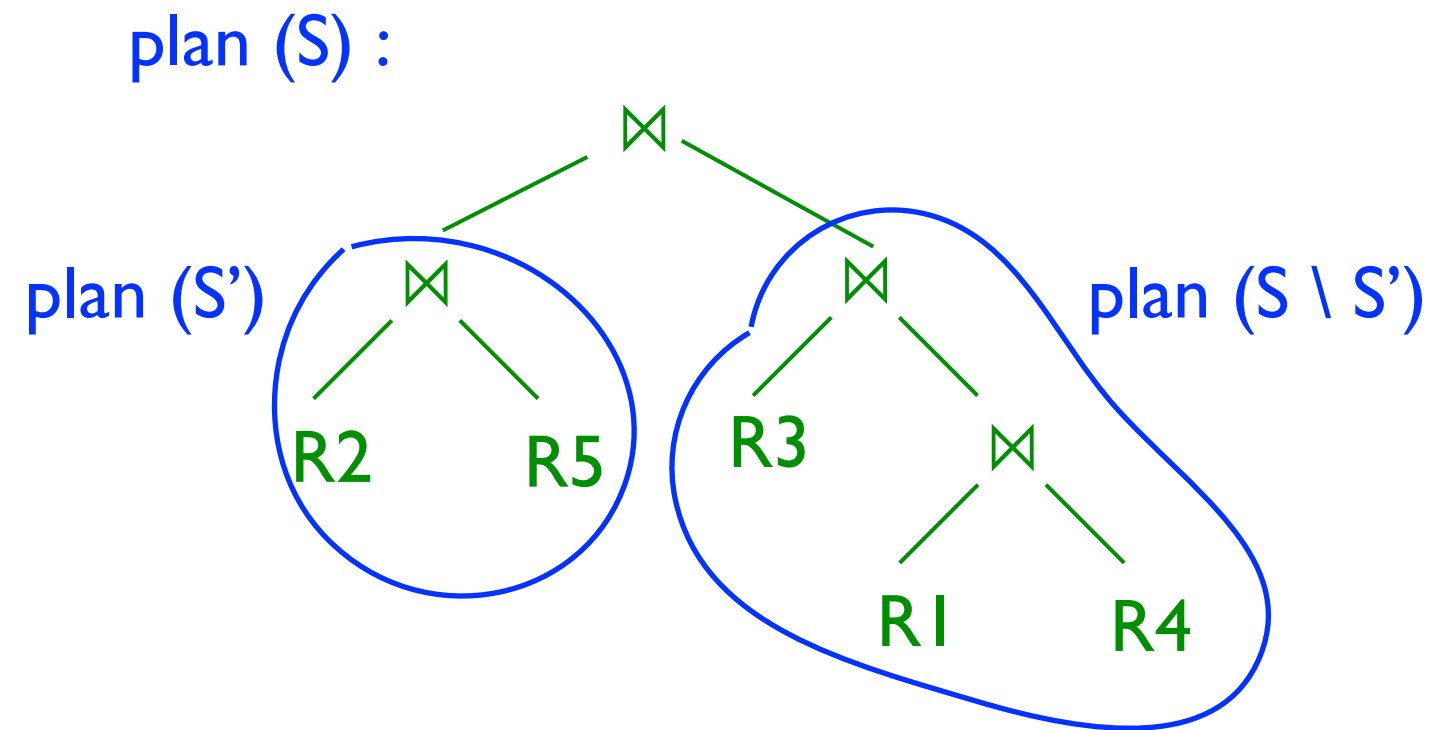
- ▶ Mais on aurait pu faire des choix différents :
 - Par exemple un autre plan pour $S' = \{R1, R2, R3\}$



Programmation dynamique pour optimiser l'ordre des jointures

- Exemple suite

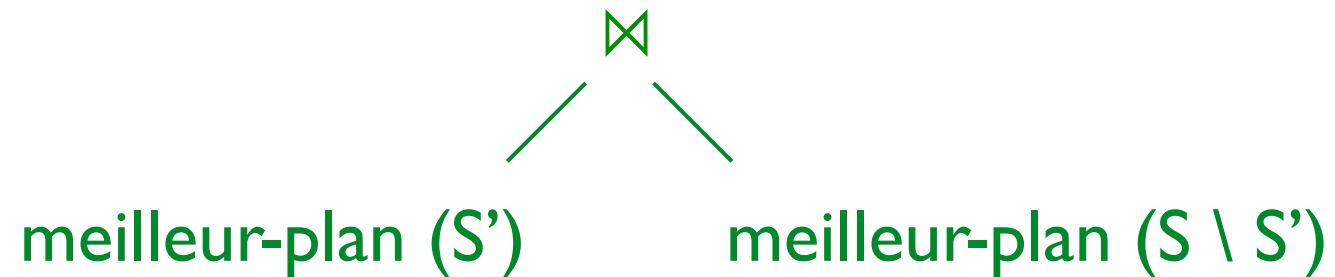
- ▶ Mais on aurait pu faire des choix différents :
 - Ou bien une autre partition, p. ex. $S' = \{R2, R5\}$, et des plans pour S' et $S \setminus S'$



Programmation dynamique pour optimiser l'ordre des jointures

- Sous-structure optimale :

- Si je fixe une partition $(S', S \setminus S')$ de S le meilleur plan est :

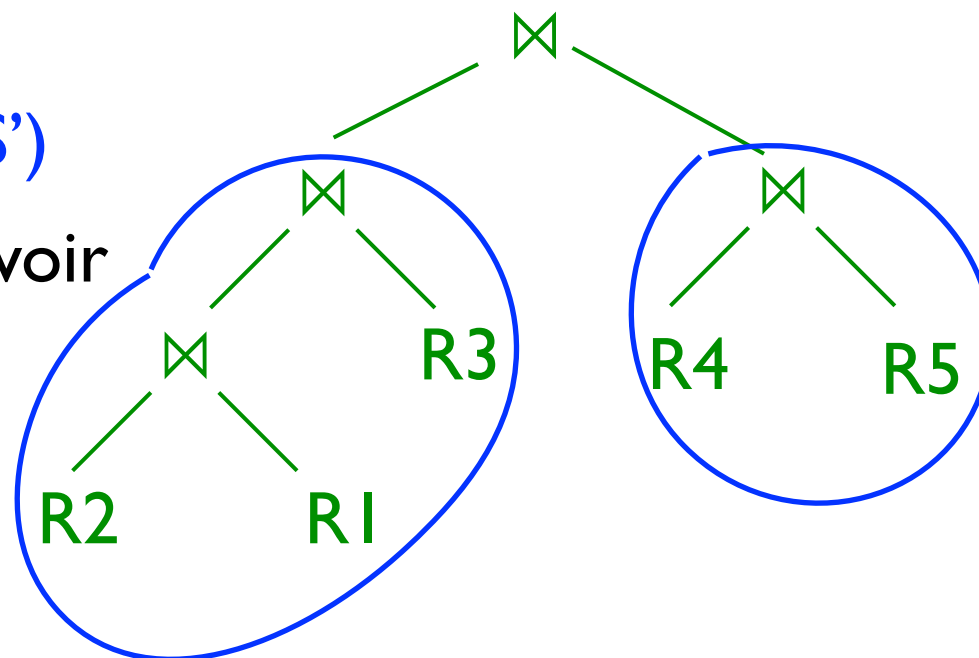


- Par exemple : pour la partition $S' = \{R1, R2, R3\}$ et $S \setminus S' = \{R4, R5\}$

meilleur-plan pour la partition :

meilleur-plan (S')

(supposons d'avoir
vérifié
qu'il est le
meilleur)

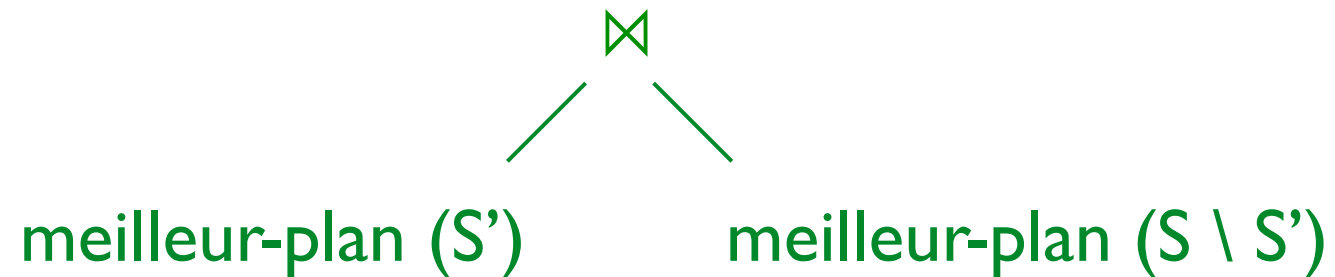


meilleur-plan ($S \setminus S'$)

Programmation dynamique pour optimiser l'ordre des jointures

- Sous-structure optimale :

- Si je fixe une partition $(S', S \setminus S')$ de S le meilleur plan est :



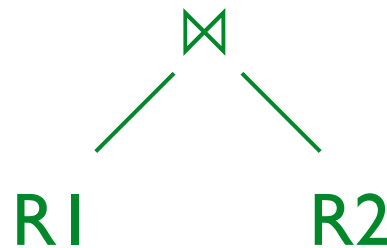
- Meilleur plan pour S :
 - pour chaque partition fixée de S je calcule le meilleur plan : je les compare tous et je prend le meilleur
- Comment je calcule meilleur-plan (S') et meilleur-plan $(S \setminus S')$?
 - Récursivement de la même façon que pour S
- Cela donne naturellement un algorithme de programmation dynamique

Programmation dynamique pour optimiser l'ordre des jointures

- Algorithme de programmation dynamique top-down (récursif avec mémorisation)

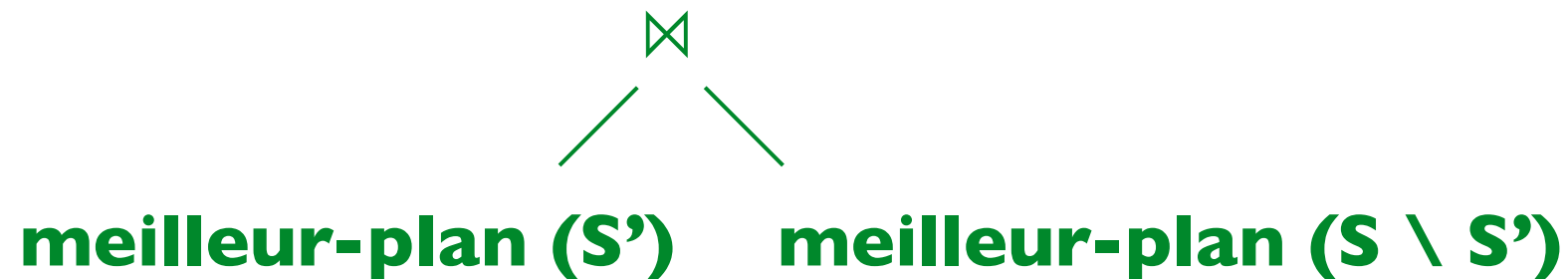
meilleur-plan(S)

Si $S = \{R1, R2\}$ retourner



Sinon

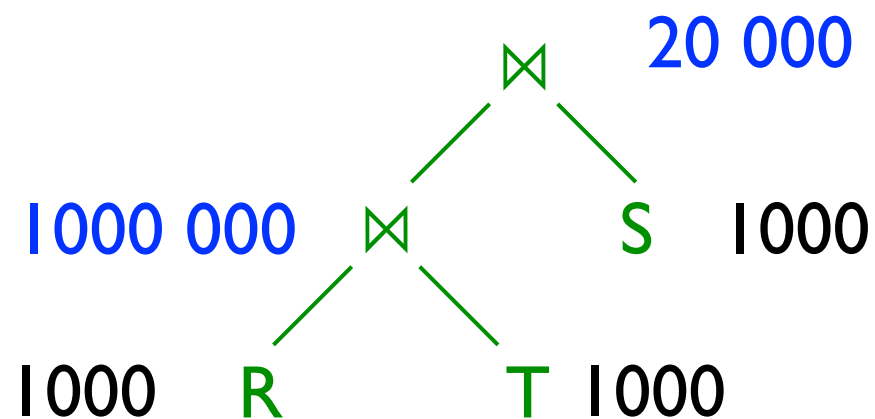
A. Pour chaque partition $(S', S \setminus S')$ de S ($S' \subset S$ et $S' \neq \emptyset$) calculer le meilleur plan, i.e :



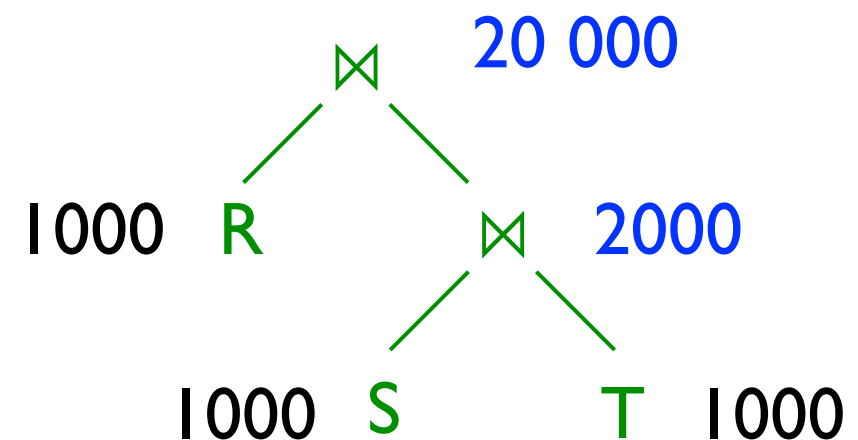
B. Choisir et retourner le meilleur parmi tous les plans calculés à l'étape A

Programmation dynamique pour optimiser l'ordre des jointures

- Algorithme de programmation dynamique top-down (récursif avec mémorisation)
 - ▶ Reste à définir comment évaluer le coût d'un plan (pour que l'algorithme puisse les comparer)
 - Pour l'instant on est au niveau des plans logiques (pas d'algorithme choisi, on ne peut pas estimer le vrai coût d'un plan)
 - Une mesure de coût pourrait être la somme des estimations des résultats intermédiaires :



Mesure de coût : 1 020 000



Mesure de coût : 22 000

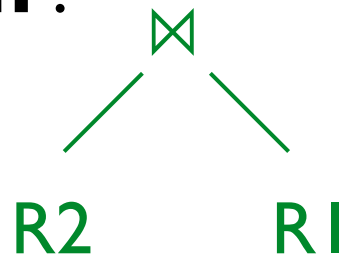
Programmation dynamique pour optimiser l'ordre des jointures

- Algorithme de programmation dynamique top-down (récursif avec mémorisation)

meilleur-plan(S)

Si $S = \{R1, R2\}$ retourner :

1) le **plan** :



2) **t** : la taille de son résultat (estimation)

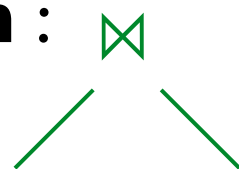
p.ex. $|R1| \times |R2| / \max\{V(a, R1), V(a, R2)\}$

3) **c** : sa mesure de coût = t

Sinon

A. Pour chaque partition $(S', S \setminus S')$ de S ($S' \subset S$ et $S' \neq \emptyset$) calculer le meilleur plan, i.e calculer :

1) le **plan** :



meilleur-plan (S')
c1, t1

meilleur-plan (S \ S')
c2, t2

2) **t** : sa taille (estimation)

p.ex. $t1 \times t2$

3) **c** : sa mesure de coût
 $= t + c1 + c2$

B. Choisir et retourner le meilleur (plus petit c) parmi tous les plans calculés à l'étape A, avec sa taille t et son coût c

Optimiser l'ordre des jointures - plans physiques

- ▶ Programmation dynamique possible aussi pour choisir directement le meilleur plan physique de jointure :
 - Si je fixe une partition $(S', S \setminus S')$ de S le meilleur plan est :

[meilleur algorithme de \bowtie
pour les résultats des deux plans]

meilleur-plan (S')

meilleur-plan $(S \setminus S')$

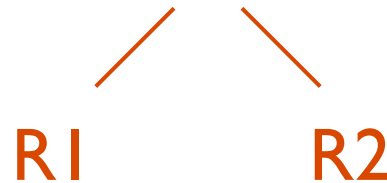
Optimiser l'ordre des jointures - plans physiques

- Algorithme de programmation dynamique top-down (récursif avec mémorisation)

meilleur-plan(S)

Si $S = \{R1, R2\}$ calculer et retourner

[meilleur algorithme de \bowtie]



Sinon

A. Pour chaque partition $(S', S \setminus S')$ de S ($S' \subset S$ et $S' \neq \emptyset$) calculer le meilleur plan, i.e :

[meilleur algorithme de \bowtie
pour les résultats des deux plans]



B. Choisir et retourner le meilleur parmi tous les plans calculés à l'étape A

Optimiser l'ordre des jointures - plans physiques

- Algorithme de programmation dynamique top-down (récursif avec mémorisation)
 - ▶ Un appel récursif retourne également :
 - Une estimation de la taille du résultat
 - Une estimation du coût du plan
 - Mesure de coût d'un plan :
 - Cout des deux sous-plans +
 - coût de l'algorithme de jointure choisi
(peut être estimé à partir des tailles des résultats des sous-plans)
- Coûteux (exponentiel!), mais on fait rarement la jointure de plus de 10 relations, ça reste donc un coût raisonnable

Rappel : énumération et heuristiques

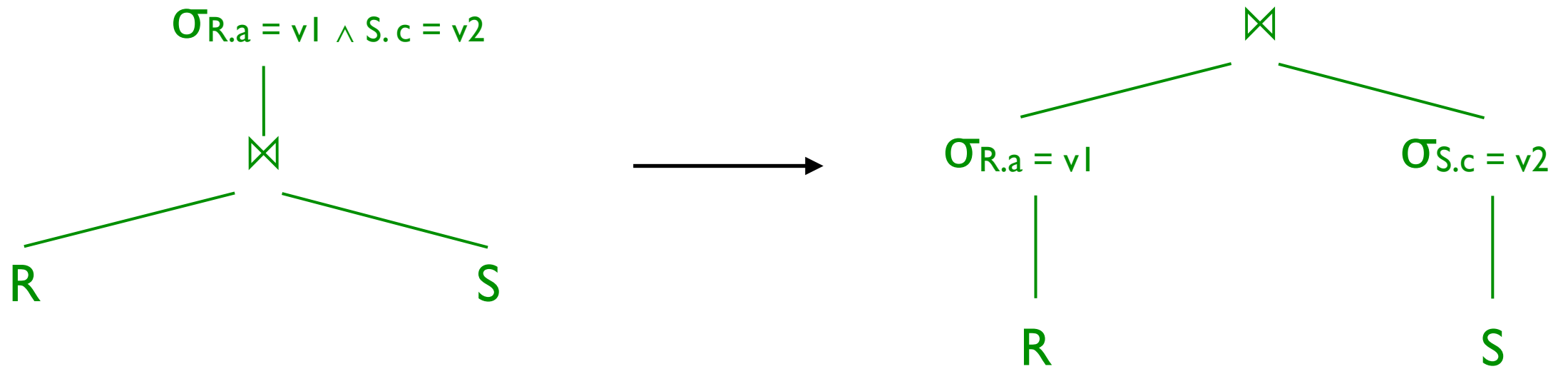
- Couramment les systèmes implémentent une **combinaison** d'heuristiques et de techniques pour énumérer les plans d'exécution
 - ▶ **Enumération** : On considère de façon plus ou moins exhaustive plusieurs plans et on évalue leur coût pour choisir le meilleur
 - ▶ **Heuristiques** :
 - On transforme un plan initial par application directionnelle de certaines règles heuristiques (à la fois au niveau du plan logique et physique) sans évaluer les alternatives
 - idée sous-jacente : dans la plus part des cas les transformations fondés sur les heuristiques aboutissent à un meilleur plan d'exécution
- **Typiquement** :
 - ▶ Heuristiques tant que c'est possibles
 - ▶ Enumération pour les choix non couverts par les heuristiques
 - ▶ Enumération pour certain choix critiques comme l'ordre des jointures

Exemples d'heuristiques

- Heuristiques de transformation du plan logique
 - ▶ Sélections le plus tôt possible
 - ▶ Projections le plus tôt possible
 - ▶ Combinaison des sélections avec les jointures et les produits cartésiens
 - ▶ Choix de l'ordre des jointures
 - ▶ ...
- Heuristique de choix des opérateurs physiques
 - ▶ Choix de méthodes de jointures
 - ▶ ...

Exemples d'heuristiques

- Sélections le plus tôt possible
 - ▶ Pousser les sélections vers le bas dans le plan

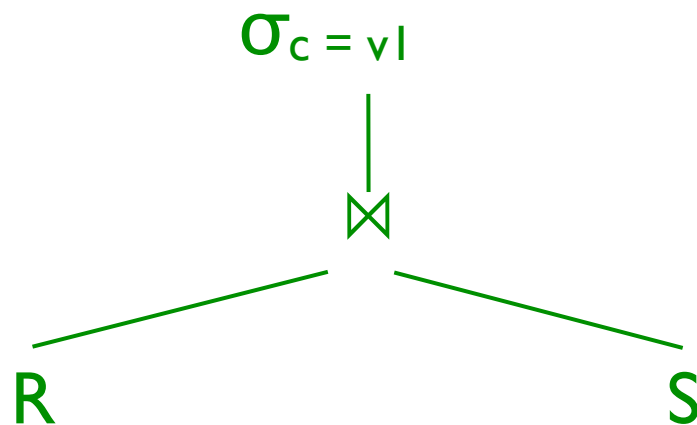


- ▶ Raison : les sélections réduisent la taille des relations sur lesquelles la jointure est calculée

Exemples d'heuristiques

- Sélections le plus tôt possible : pas toujours le meilleur choix!

► Exemple 1 : $R(a, b) \bowtie S(b, c)$, index sur $S.b$, $B_R \ll B_S$

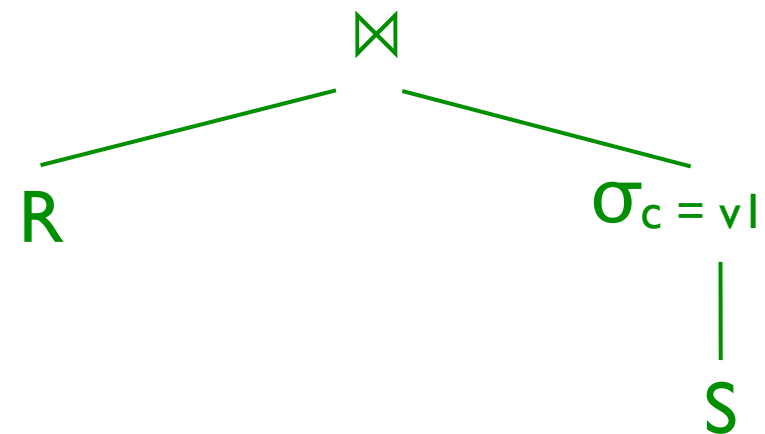


index-join possible

- coût $B_R + n_R \cdot c$
 c : coût d'une sélection
 par index-scan sur S (efficace)

sélection par linear scan :

- coût : $B_{R \bowtie S} \ll B_R \cdot B_S$



sélection par linear scan :

- coût : B_S

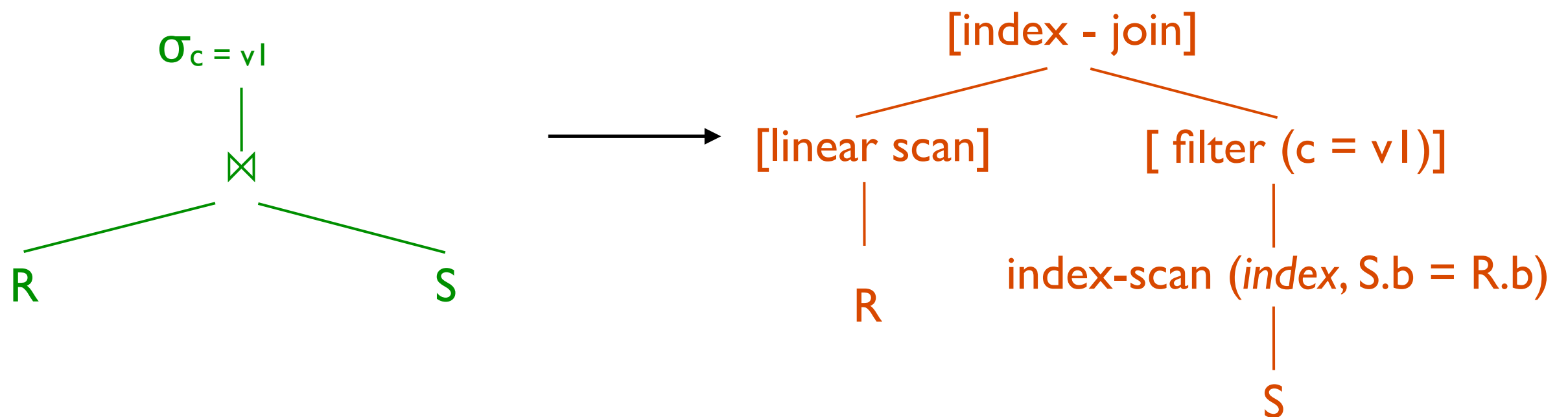
index-join pas possible

- coût $B_R + B_R \cdot B_S$

► la deuxième solution pourrait être beaucoup plus coûteuse

Exemples d'heuristiques

- Sélections le plus tôt possible : pas toujours le meilleur choix!
 - ▶ Exemple 1 : $R(a, b) \bowtie S(b, c)$, index sur $S.b$, $B_R \ll B_S$
 - ▶ la première solution peut être implémentée efficacement en testant $c = v_l$ sur les tuples sélectionnés par l'index, pendant l'index-join

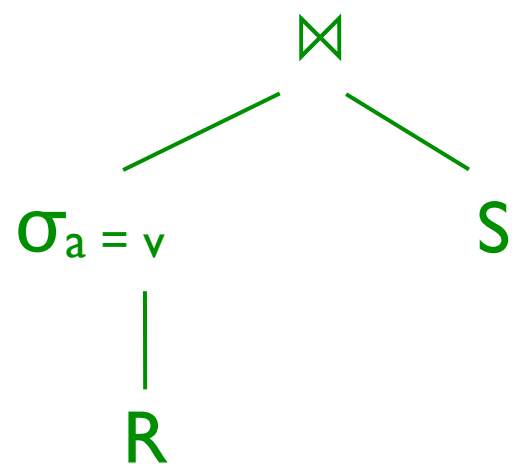


- ▶ cela n'est pas équivalent à pousser la sélection $\sigma_{c=v_l}$ sur S (ce qui demanderait un linear scan sur S)

Exemples d'heuristiques

- Sélections le plus tôt possible : pas toujours le meilleur choix!
 - ▶ Exemple 2 $R(a, b, c), S(a, b, d)$

plan original



transformation potentiellement rentable

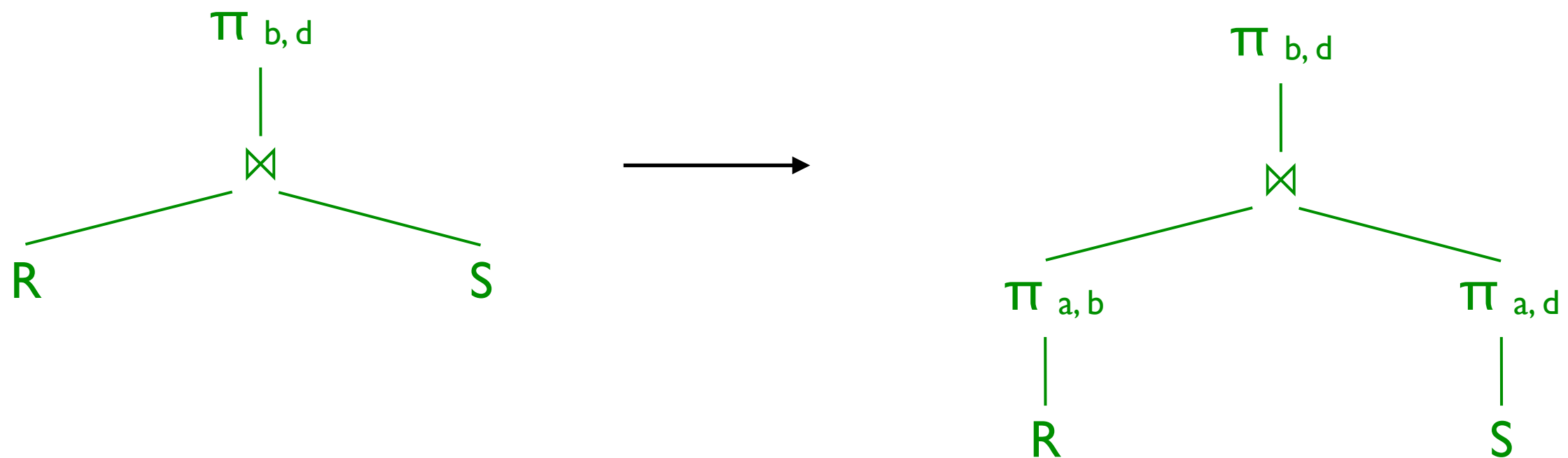


- ▶ la meilleure transformation pourrait nécessiter de commencer par pousser une sélection **en haut**

Exemples d'heuristiques

- **Projections le plus tôt possible** \Leftrightarrow introduire des projections le plus bas possible dans le plan logique
- Une projection peut être introduite à tout endroit, à condition qu'elle n'élimine pas d'attributs utilisés par les opérateurs ancêtres

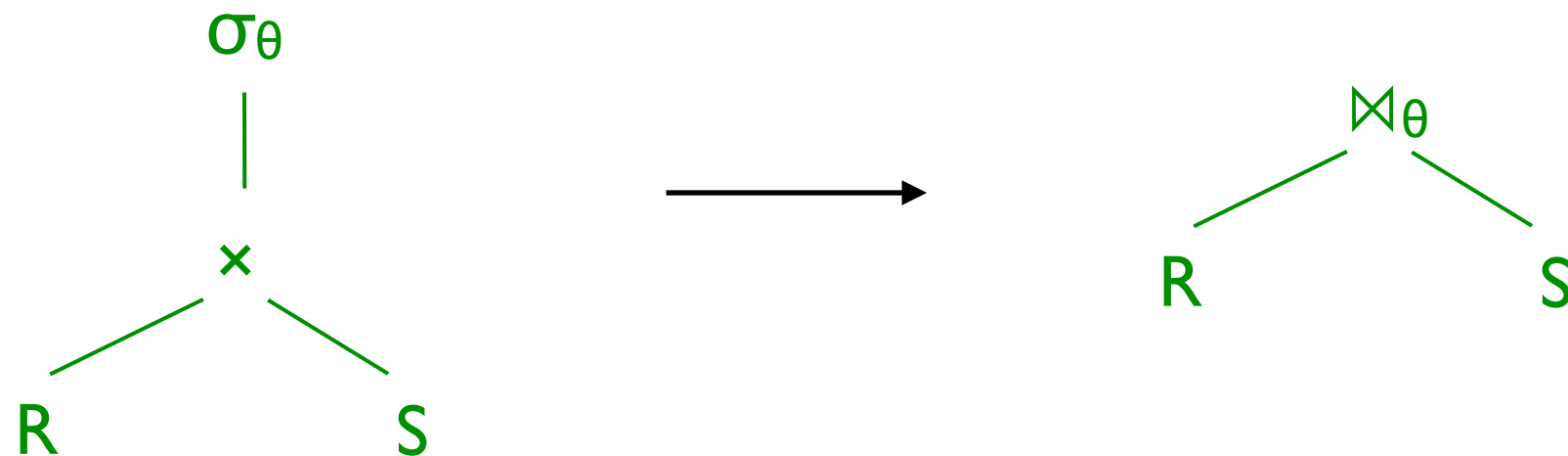
Example $R(a, b, c) \bowtie S(a, d, e)$



- **Effet moins avantageux que celui de pousser les sélections vers le bas**
 - ▶ la projection réduit seulement la taille de chaque tuple, mais en général pas le nombre de tuples (à part les doublons)

Exemples d'heuristiques

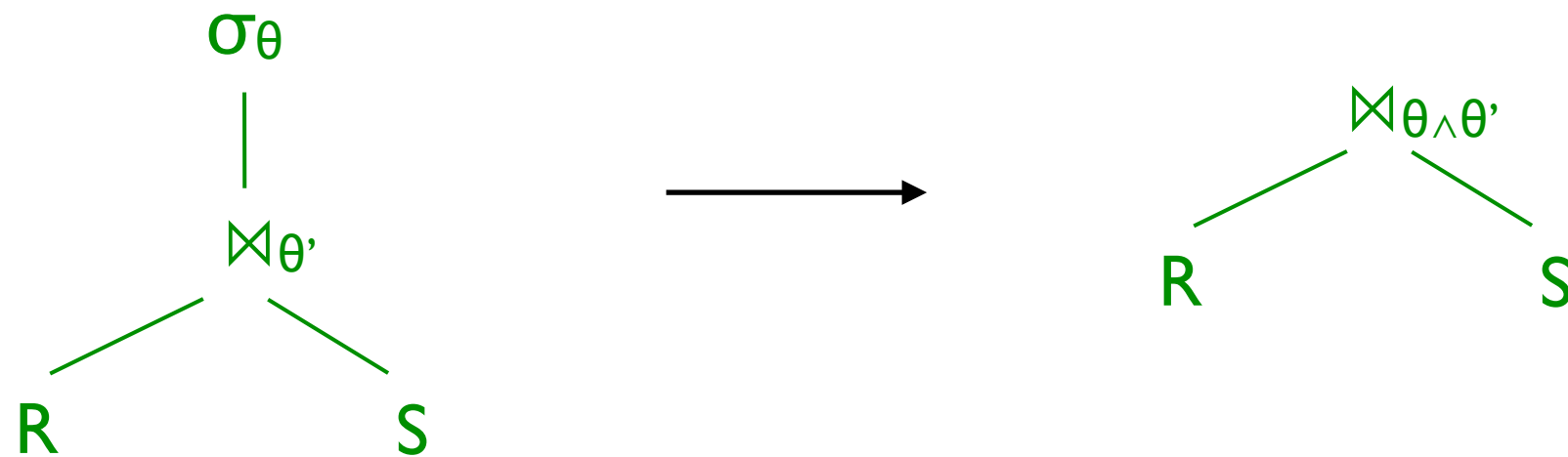
- Combinaison de sélections et produits cartésiens



- ▶ algorithmes de jointure en général bien plus efficaces que le calcul du produit cartésien (quadratique) suivi d'une sélection.
- ▶ Spécialement utile en présence d'index sur les attributs de θ (index-join possible)

Exemples d'heuristiques

- Combinaison de sélections et jointures



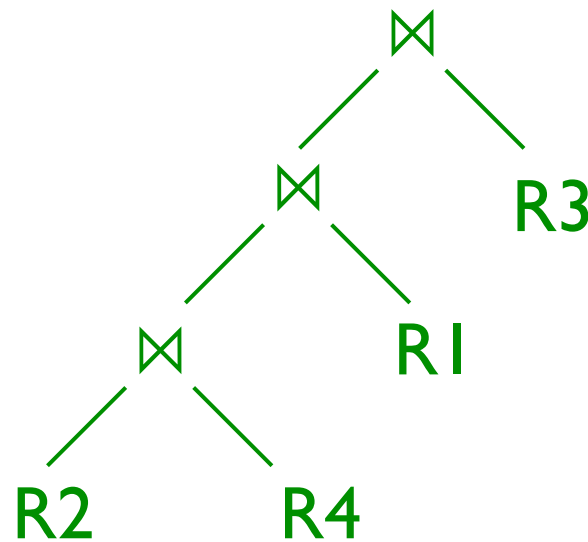
- Il pourrait être plus avantageux de vérifier θ pendant la jointure, spécialement en présence d'index sur les attributs de θ (index-join possible)

Exemples d'heuristiques

- Choix de l'ordre et des algorithmes de jointure
 - ▶ Heuristiques si l'énumération basée sur le coût est
 - trop coûteuse ou
 - inapplicable par manque d'information (histogrammes, nombre de buffers disponibles etc)

Exemples d'heuristiques

- **Choix de l'ordre des jointures.** Heuristique courante :
 - ▶ pour une jointure entre n relations R_1, \dots, R_n , considérer n plans
 - ▶ le plan i sélectionne la relation R_i au départ
 - ensuite à chaque étape il sélectionne, parmi les relations restantes, la “meilleure” à joindre au résultat



on obtient des plans
de jointure profonds
à gauche

- ▶ plusieurs critères pour établir la “meilleure” prochaine relation :
 - la présence d'index qui permettent index-join
 - une estimation de la taille du résultat
- ▶ les n plans sont comparés, et celui qui minimise le nombre de jointures coûteuse (nested-loop) est choisi

Exemples d'heuristiques

- Choix des méthodes de jointure ($R \bowtie S$). heuristiques à appliquer dans l'ordre :
 1. Si une des relations peut être chargée entièrement en mémoire
 \Rightarrow jointure à une seule passe
 2. Si une des relations peut être parcourue avec peu d'accès au disque
 \Rightarrow block nested loop avec relation externe = relation petite
 3. Si R n'est pas trop grande et il existe un index sur les attributs de jointure de S
 \Rightarrow index join
 4. Si une ou les deux relations sont déjà triées \Rightarrow merge join
 5. S'il y a plusieurs jointures sur le même attribut \Rightarrow merge join
 $(R(a, b) \bowtie S(a, c)) \bowtie T(a, d)$: le résultat du premier merge join est déjà trié sur l'attribut a pour le deuxième merge join
 6. Dans tous les autres cas : hash join ou block nested loop join
hash join de préférence (plus efficace)

Exemples d'optimisation - Exemple I

Une optimisation possible de la requête :

```
SELECT nom, titre  
FROM Enseignant, Cours, Enseigne  
WHERE univ = 'P7' AND année = 2015  
AND cid = cours AND eid = enseignant
```

Sur le schema :

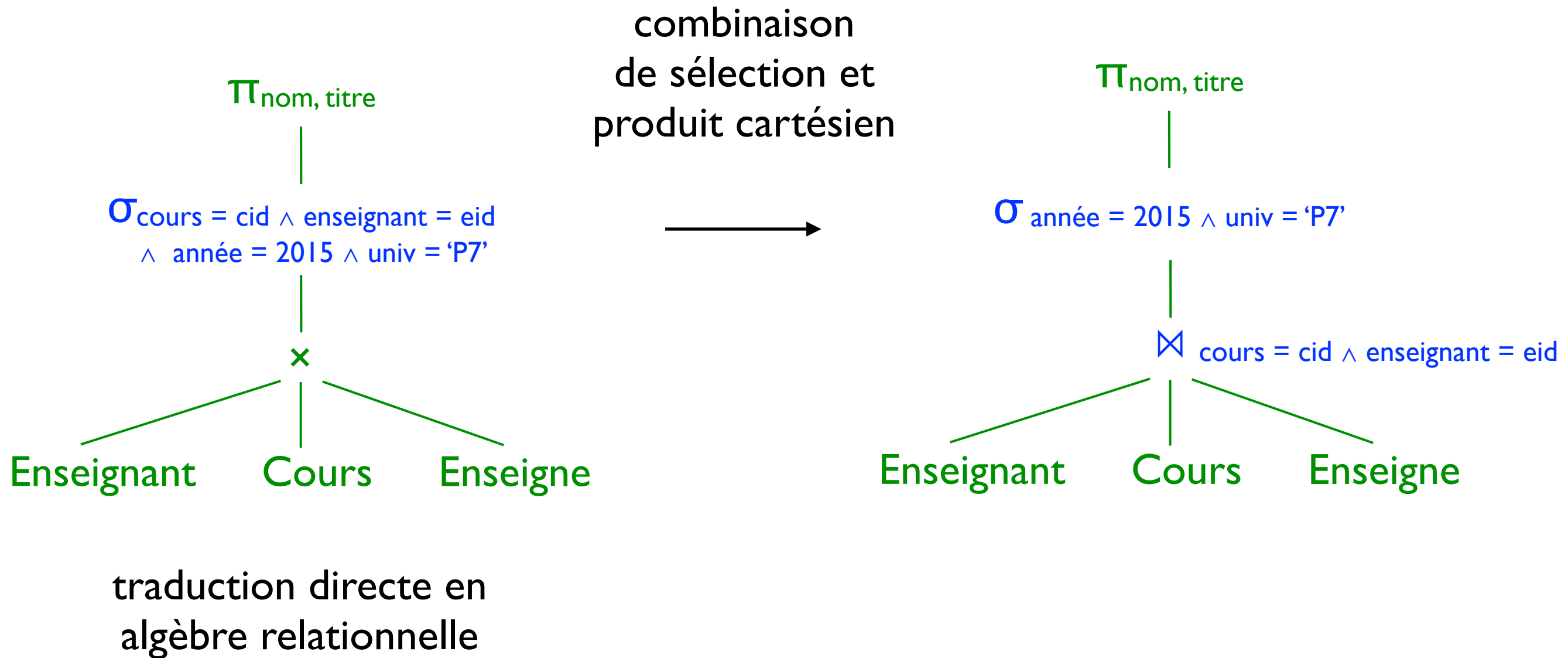
Enseignant (eid, nom, univ)

Enseigne (enseignant, cours, année) enseignant \subseteq eid cours \subseteq cid



Cours (cid, titre)

Remarque : un index est toujours présent sur la clef d'une relation

Exemples d'optimisation - Exemple I

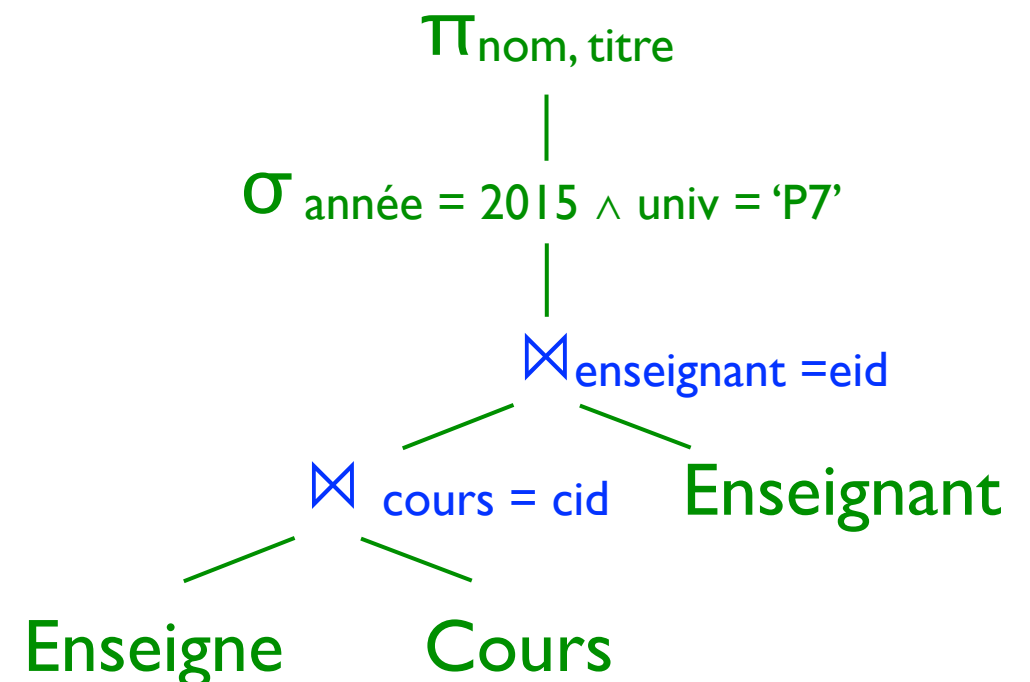
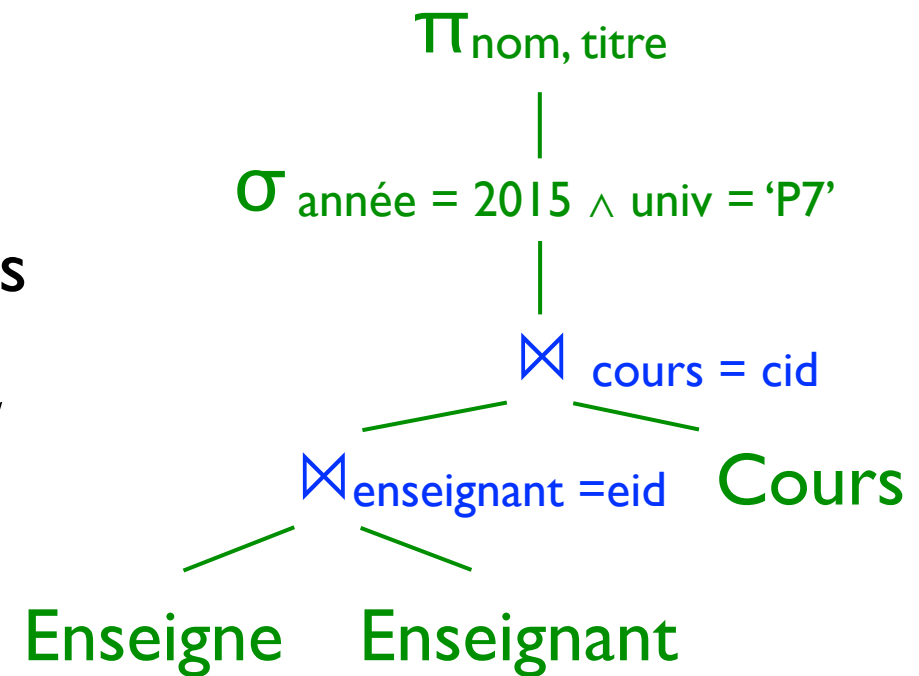
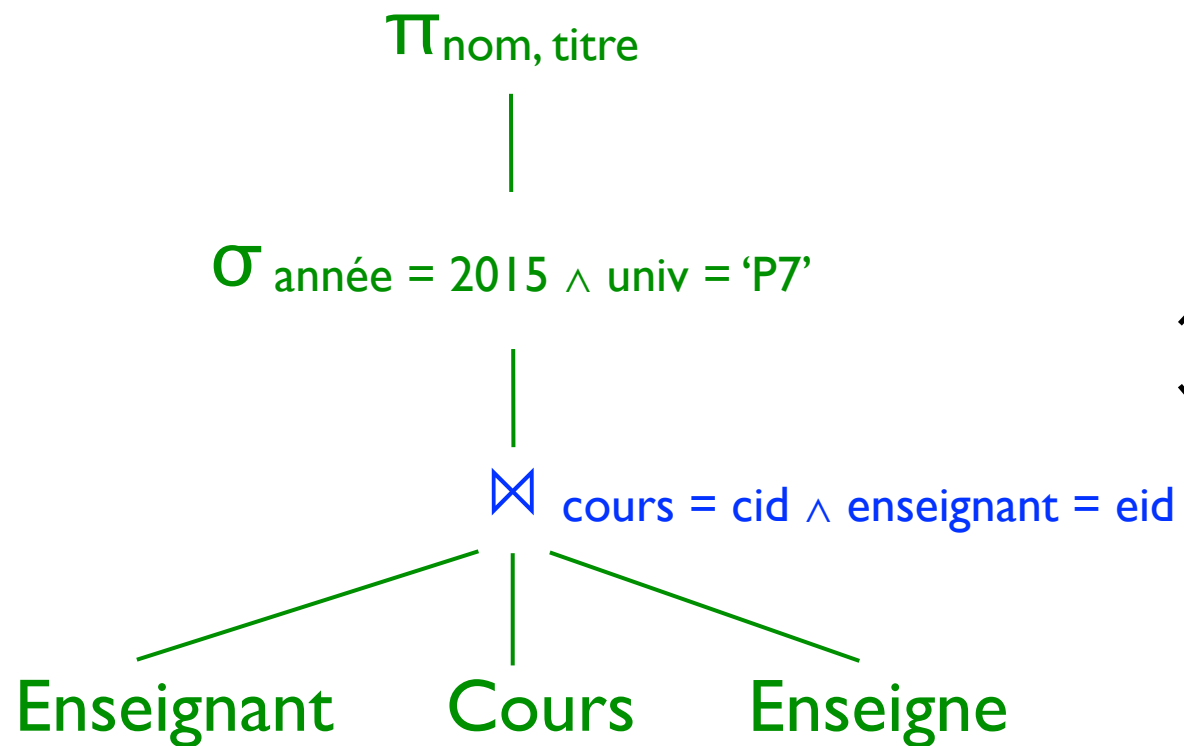


Exemples d'optimisation - Exemple I, cont.

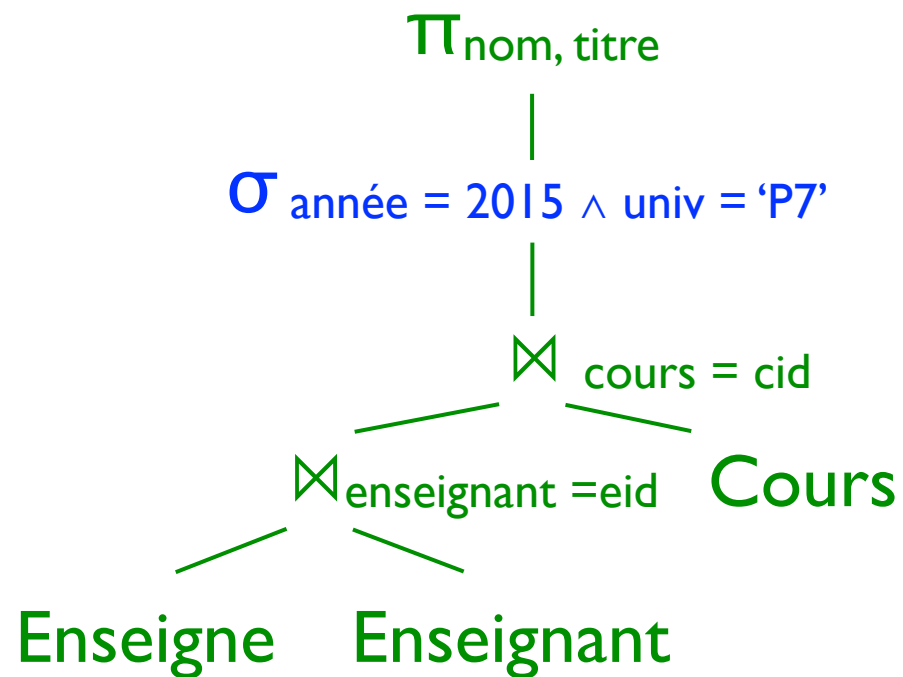
- Ordre des jointures - heuristique : maximiser le nombre d'index-join
 - ▶ $\text{Enseigne} \bowtie (\text{Enseignant} \bowtie \text{Cours})$:
 - un produit cartésien, un index-join possible
 - ▶ $(\text{Enseigne} \bowtie \text{Enseignant}) \bowtie \text{Cours}$: 
 - deux index-join possibles
 - ▶ $(\text{Enseigne} \bowtie \text{Cours}) \bowtie \text{Enseignant}$: 
 - deux index-join possibles

Exemples d'optimisation - Exemple I, cont.

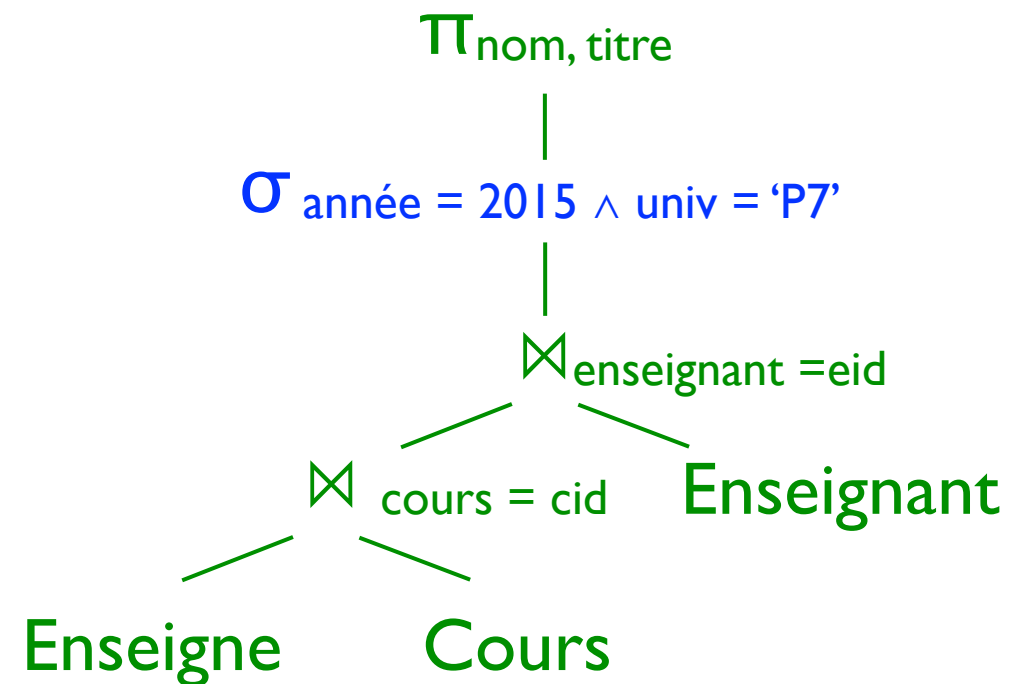
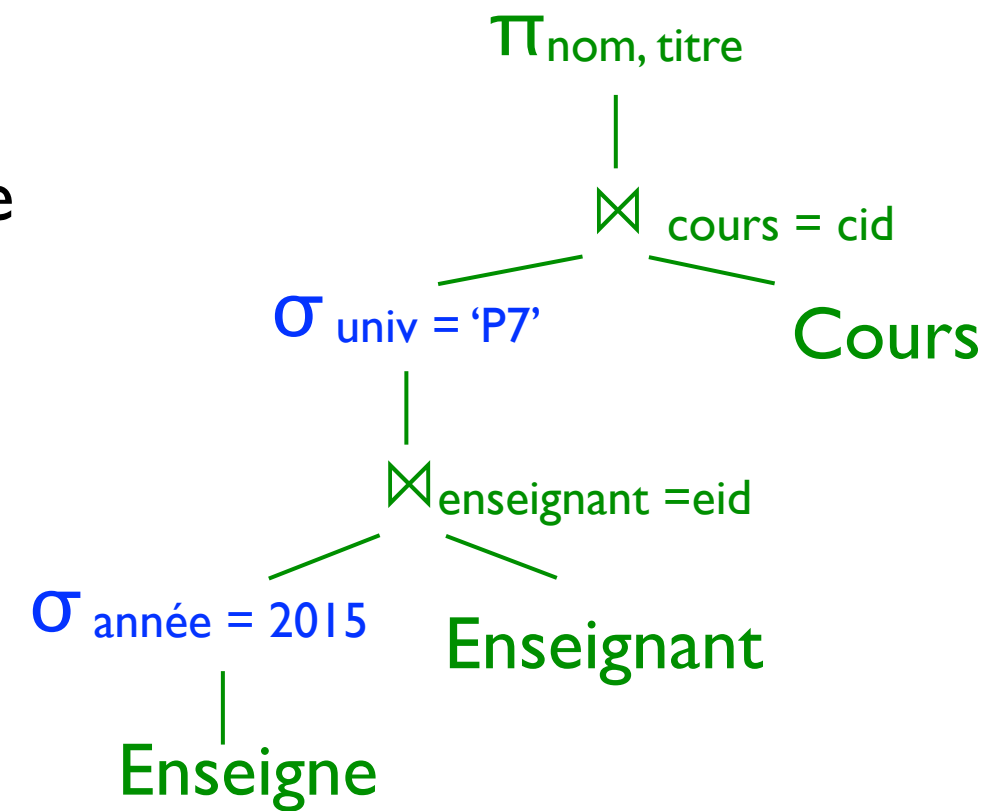
maximisation des
index-join possibles



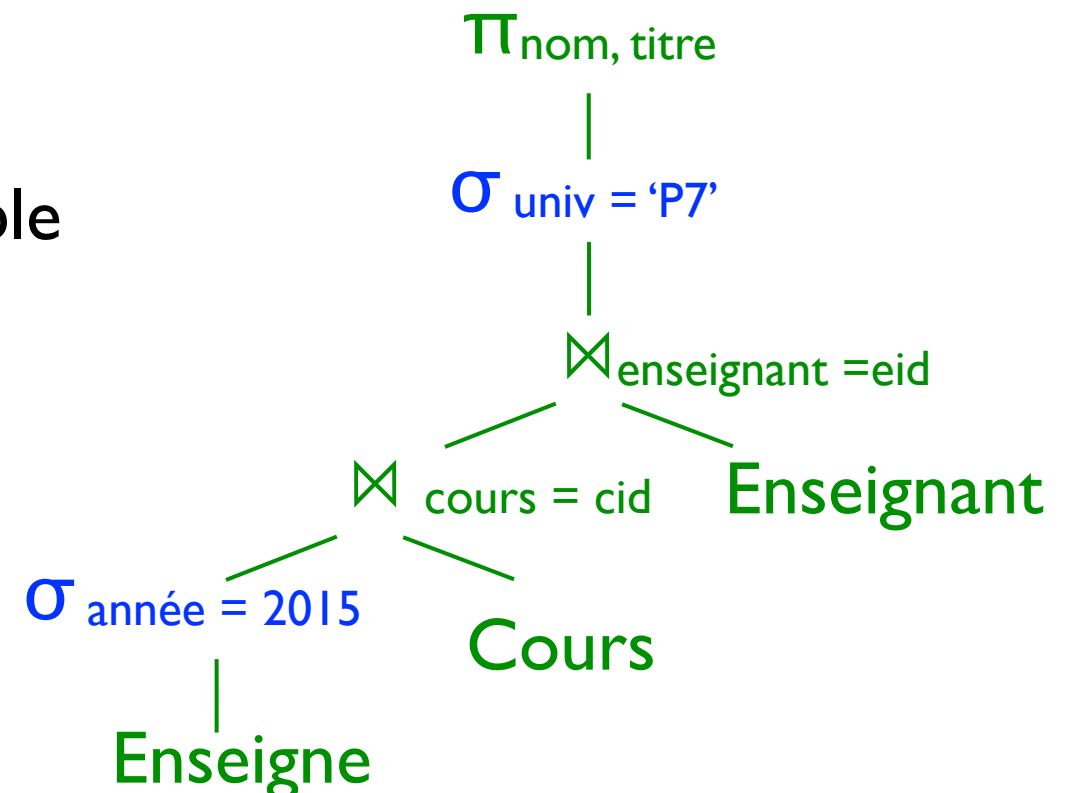
Exemples d'optimisation - Exemple I, cont.



sélections le plus tôt possible

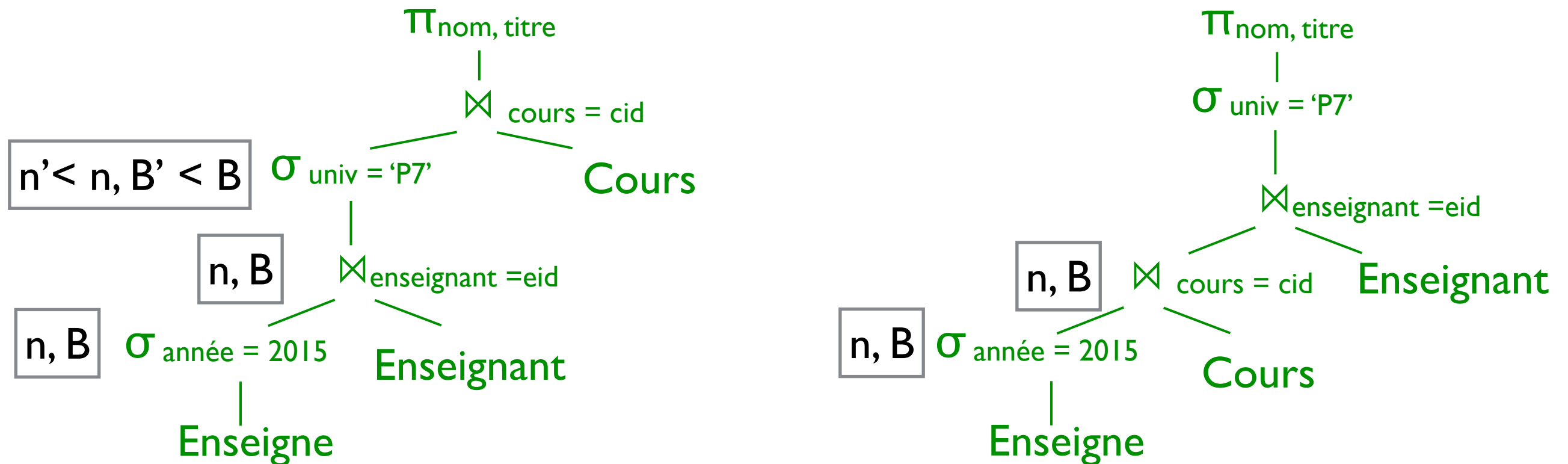


sélections le plus tôt possible



Exemples d'optimisation - Exemple I, cont.

- Comparaison des deux plans logiques obtenus

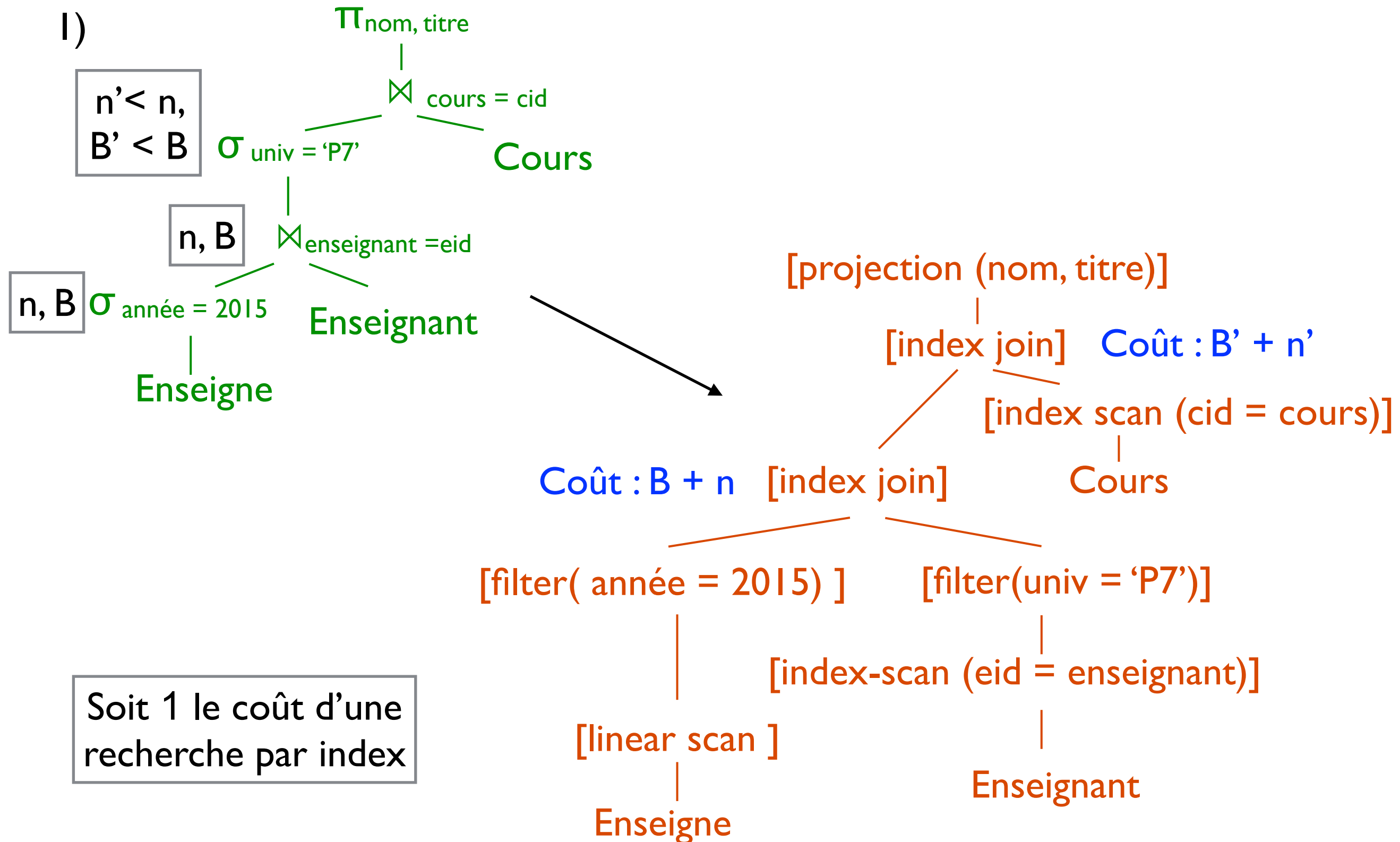


- ▶ Estimation des résultats intermédiaires :
 - résultat de la jointure la plus profonde : même taille que le résultat de la sélection sur année (attributs de jointure = clef étrangère)
 - **Première solution intuitivement meilleure** : la jointure la plus externe a un operand plus petit

Exemples d'optimisation - Exemple I, cont.

- Plus précisément on peut comparer les coût des plans physiques visés

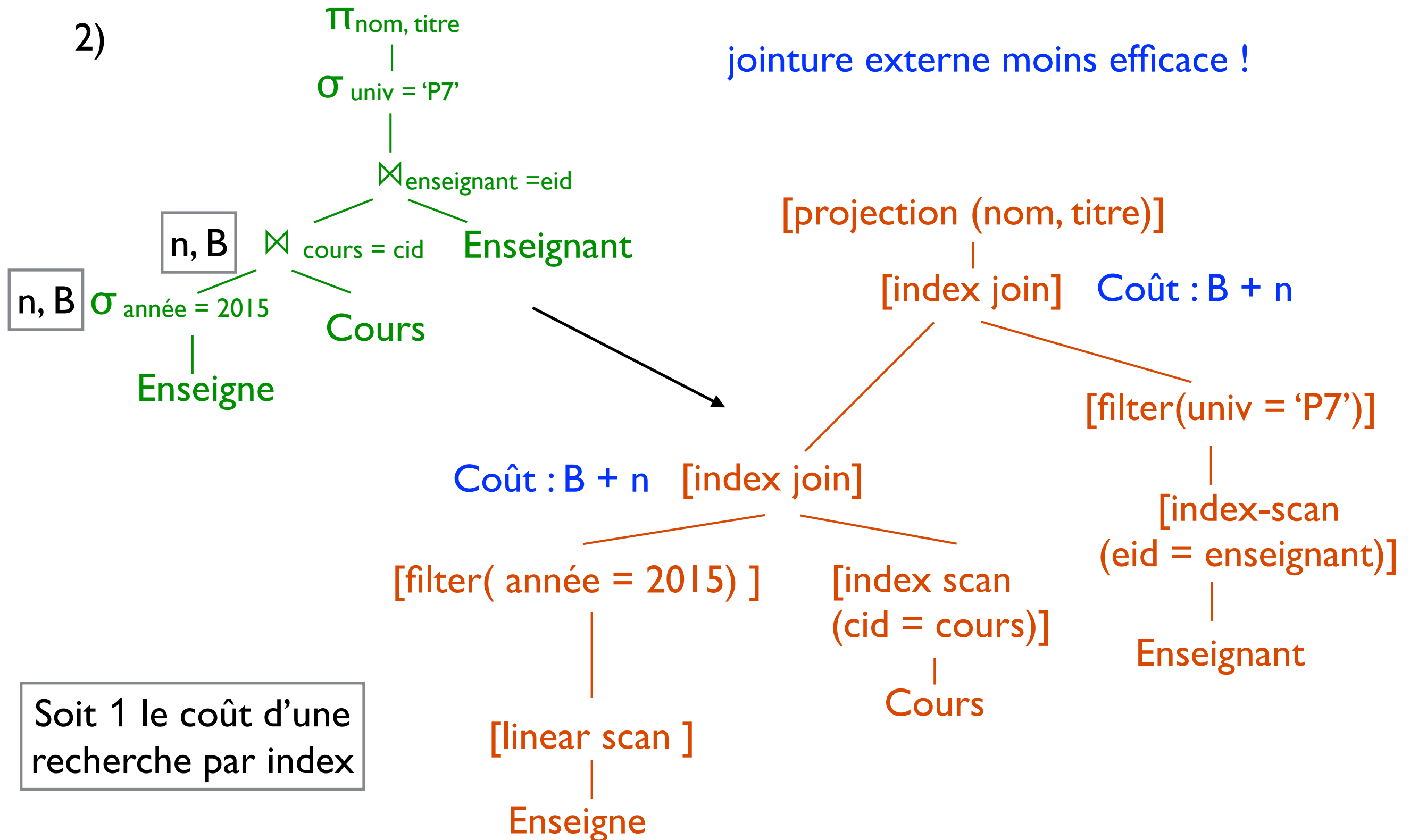
I)



Exemples d'optimisation - Exemple I, cont.

- Plus précisément on peut comparer les coût des plans physiques visés

2)



Exemples d'optimisation - Exemple I, fin

SELECT nom, titre

FROM Enseignant, Cours, Enseigne

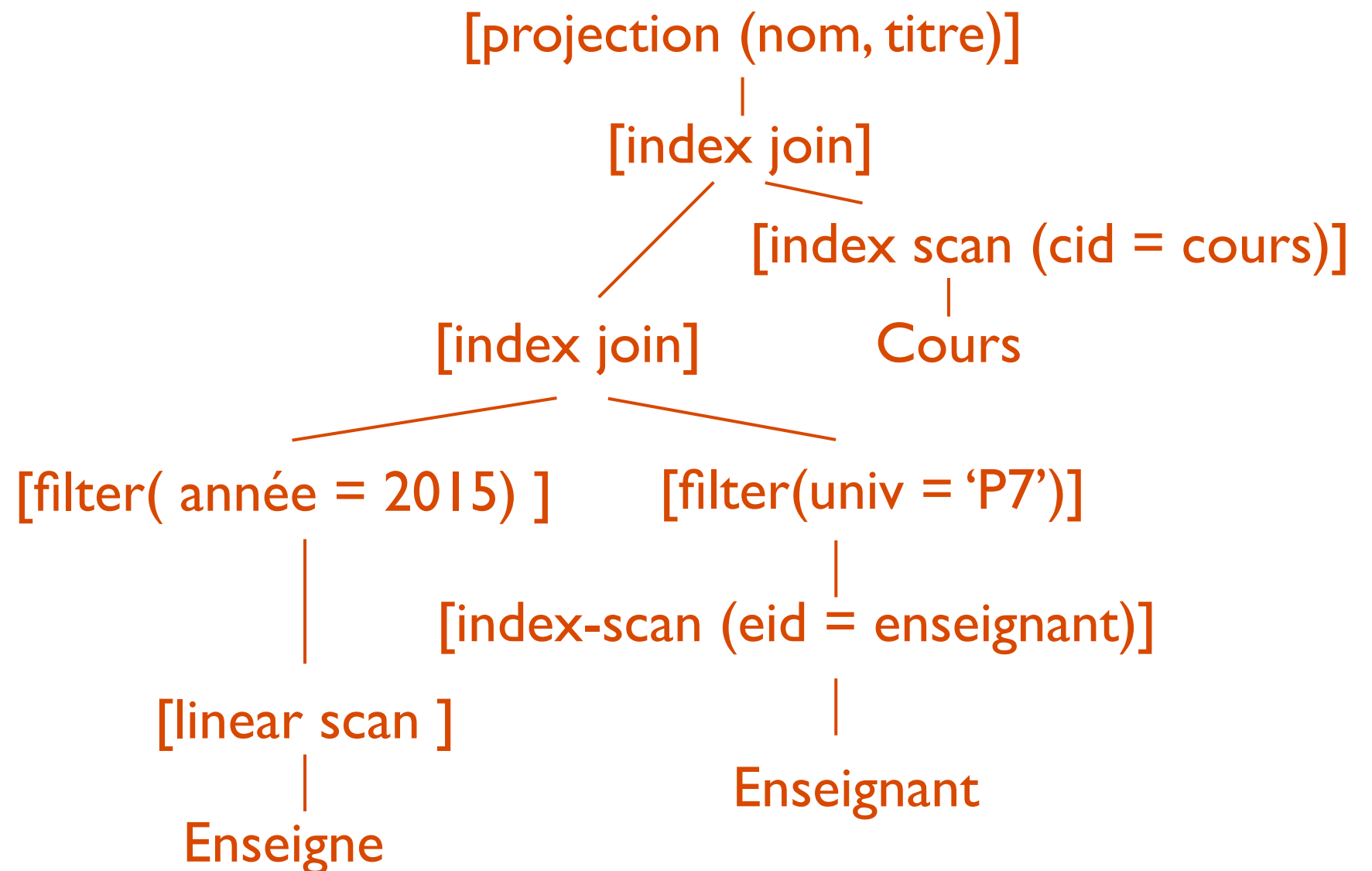
WHERE univ = 'P7'

AND année = 2015

AND cid = cours

AND eid = enseignant

Plans I) sélectionné :



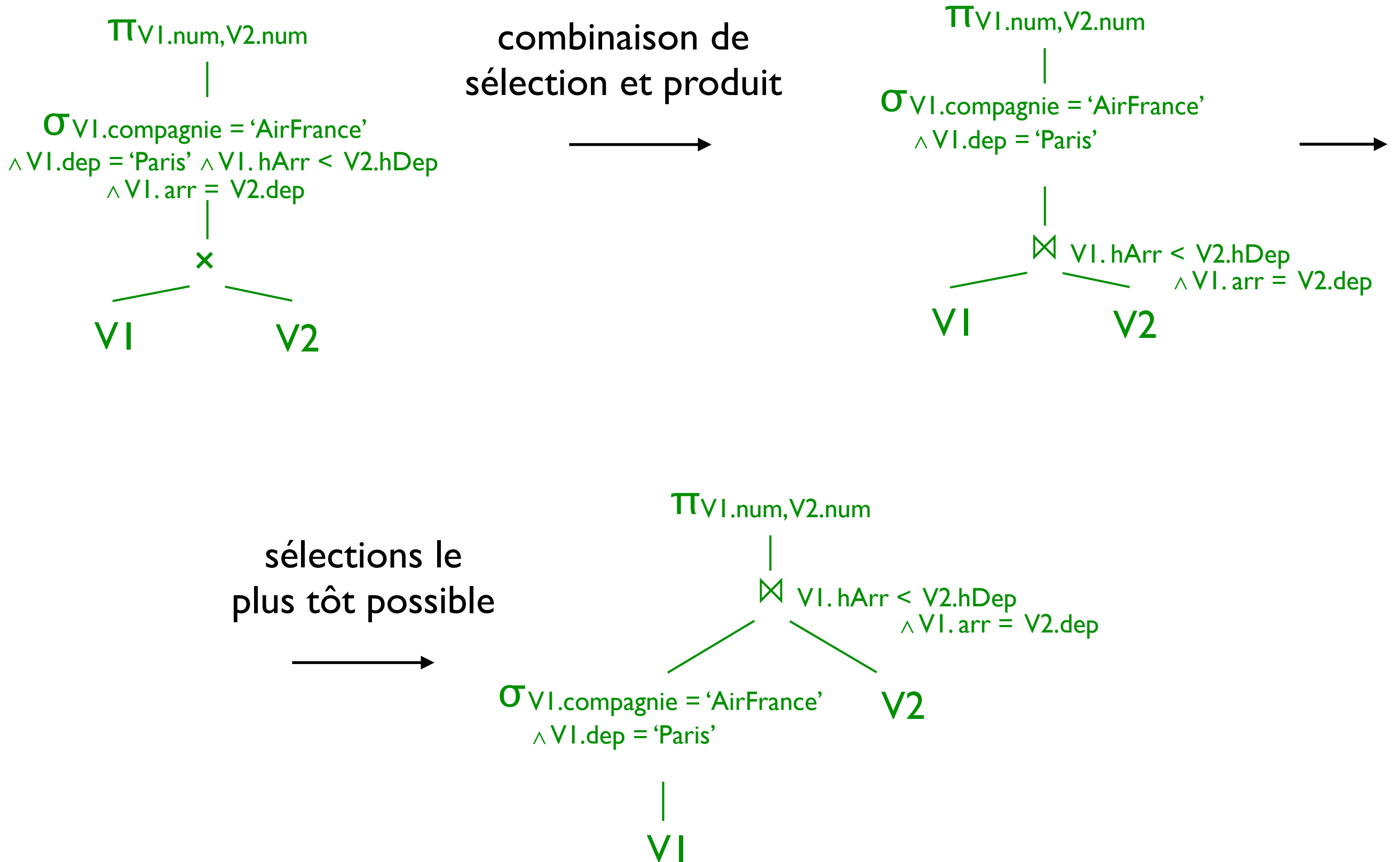
Exemples d'optimisation - Exemple 2

Une optimisation possible de la requête :

```
SELECT V1.num, V2.num  
FROM Vol AS V1, Vol AS V2  
WHERE V1.compagnie = 'Air France' AND V1.dep = 'Paris'  
AND V1.arr = V2.dep AND V1.hArr < V2.hDep
```

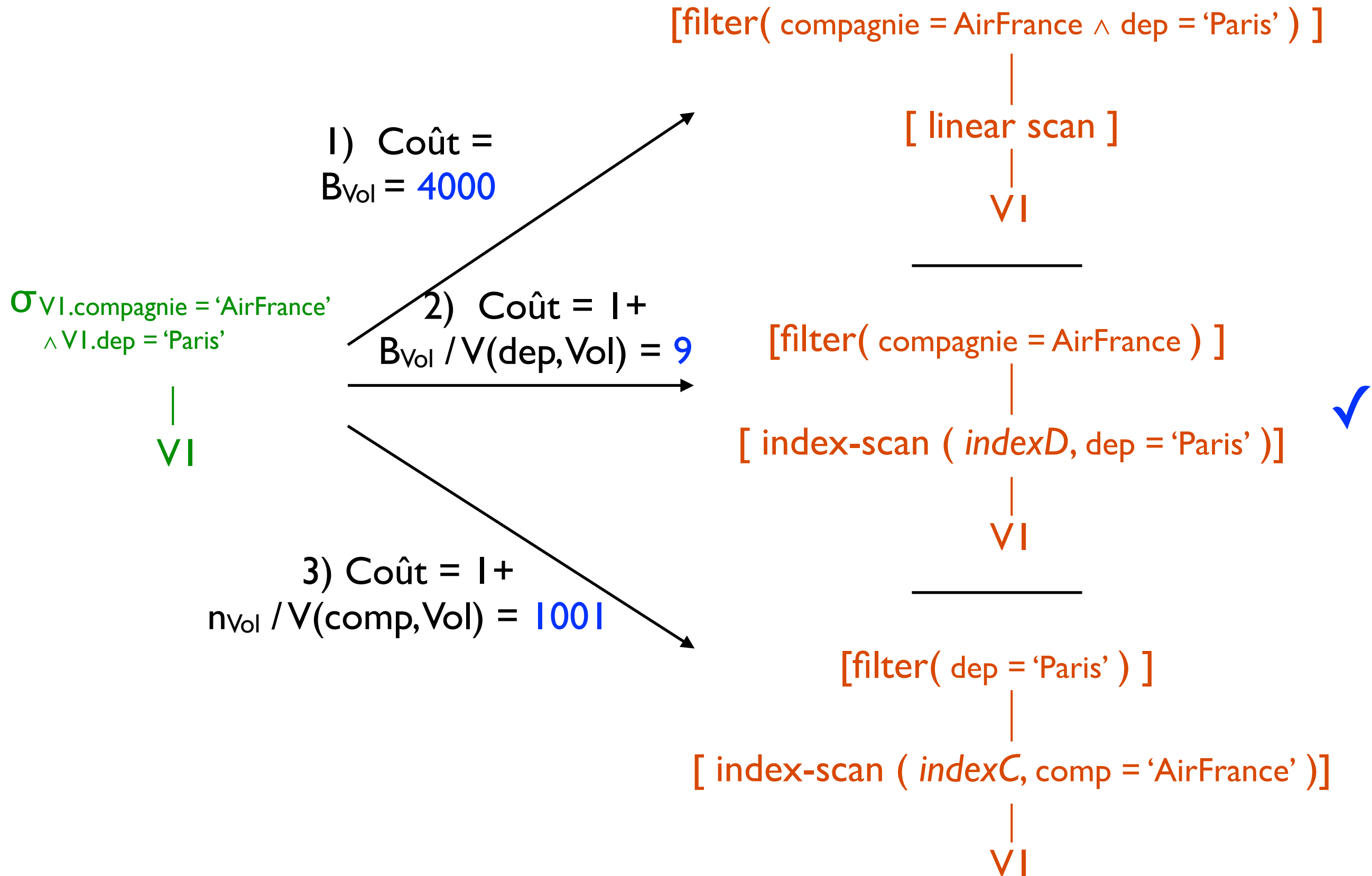
- ▶ Sur le schema : Vol (num, compagnie, dep, arr, hDep, hArr)
- ▶ Avec les statistiques suivantes :
 - $n_{Vol} = 100\ 000$, $B_{Vol} = 4000$
 - $V(compagnie, Vol) = 100$, $V(dep, Vol) = 500$
 - $V(<compagnie, dep>, Vol) = 2000$
- ▶ $M = 4$ blocs de buffer
- ▶ Index secondaire *indexC* sur Vol.compagnie
- ▶ Index primaire *indexD* sur Vol. dep

Exemples d'optimisation - Exemple 2, cont.



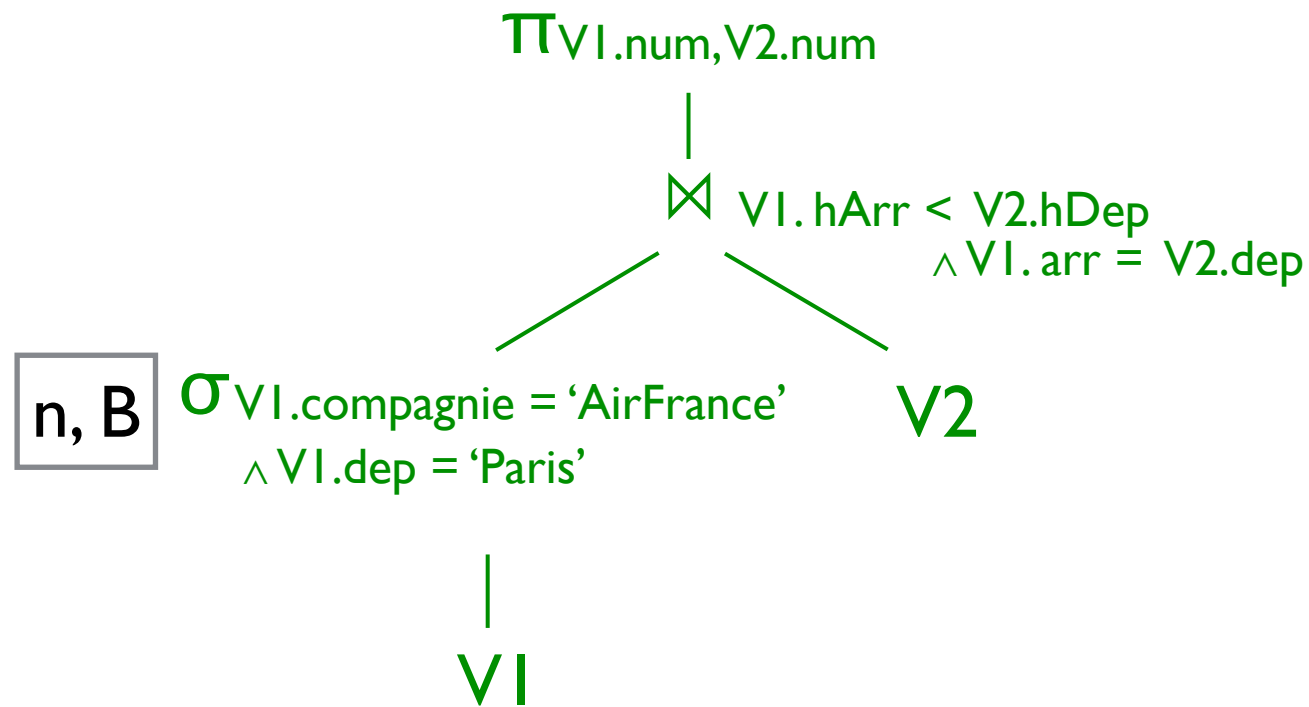
Exemples d'optimisation - Exemple 2, cont.

- Choix de la méthode de sélection



Exemples d'optimisation - Exemple 2, cont.

- Choix de la méthode de jointure



$$n = n_{Vol} / V(<comp, dep>, Vol) \\ = 100\,000 / 2000 = 50$$

$$B = n \cdot B_{Vol} / n_{Vol} \\ = 50 \cdot 4000 / 100\,000 = 2$$

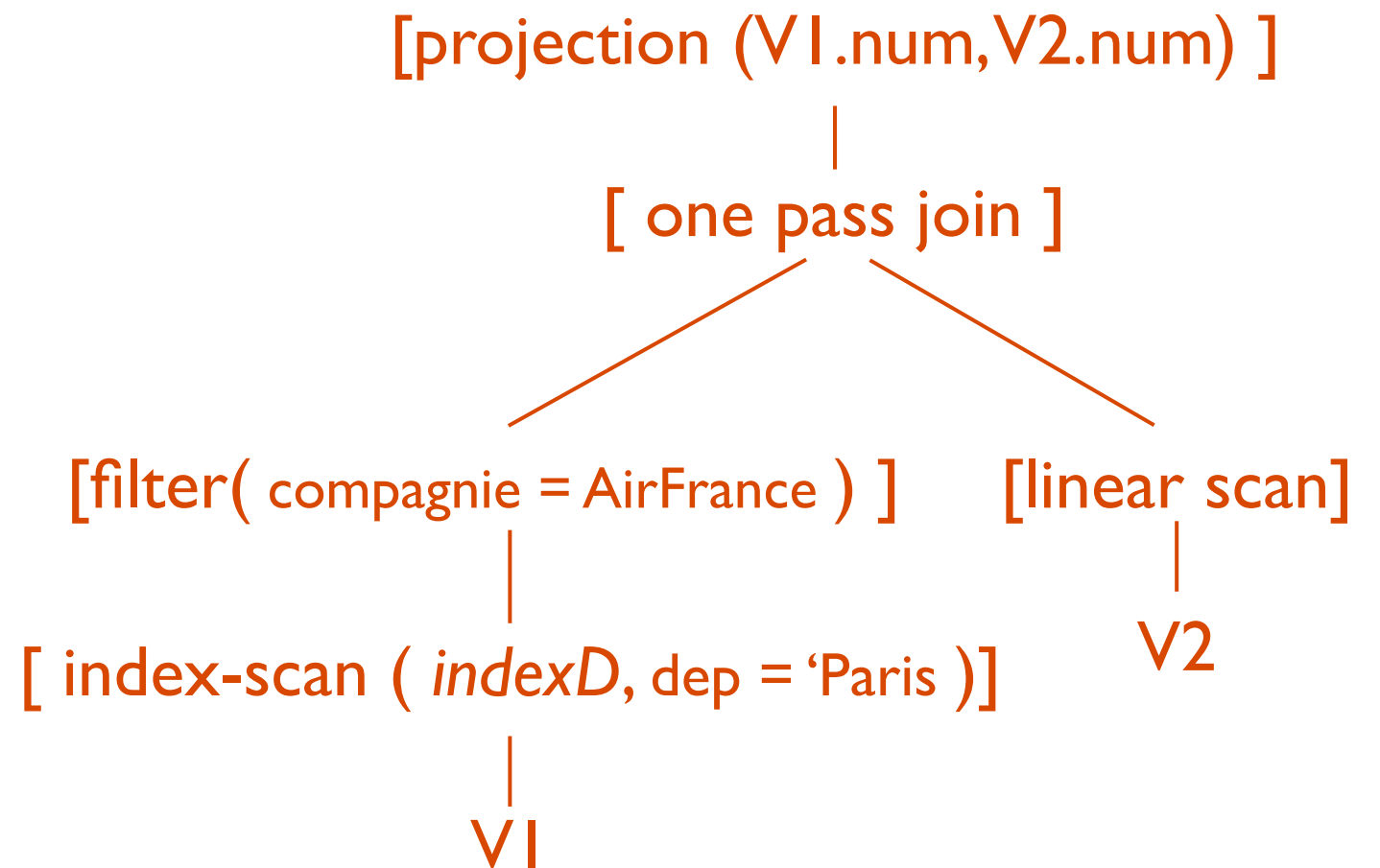
\Rightarrow le résultat intermédiaire de la sélection peut être stocké en mémoire principale ($M = 4$)

cela laisse dans le buffer un autre bloc d'entrée et un bloc de sortie pour effectuer une **jointure à une passe** (qu'on préfère malgré la possibilité d'index-join)

Exemples d'optimisation - Exemple 2, fin

```
SELECT V1.num, V2.num  
FROM Vol AS V1, Vol AS V2  
WHERE V1.compagnie = 'Air France'  
AND V1.dep = 'Paris'  
AND V1.arr = V2.dep  
AND V1.hArr < V2.hDep
```

Plan physique choisi



Optimisation de requêtes imbriquées

- **Requêtes imbriquées** : sous-requêtes présentes dans la condition d'une autre requête :
Ex.

```
SELECT DISTINCT nom
FROM Enseignant
WHERE EXISTS (
    SELECT *
    FROM Enseigne
    WHERE enseignant = eid
    AND année = '2015'
)
```

- Typiquement **avec correlation** : une condition dans la requête interne fait référence à des attributs de la requête externe (appelés paramètres de la requête imbriquée)

Ex. **WHERE** enseignant = eid (eid est le paramètre)

Optimisation de requêtes imbriquées

- L'évaluation sans optimisation préalable des requêtes avec imbrication est très inefficace (**évaluation corrélée**)
- Soit **p** l'attribut de la requête principale utilisé comme paramètre de la requête imbriquée
 - ▶ calculer le résultat de la partie FROM-WHERE de la requête externe
 - ▶ pour chaque tuple **t** de ce résultat :
 - évaluer la requête imbriquée avec valeur **t[p]** du paramètre
- Pour éviter l'évaluation corrélée les systèmes essaient d'**éliminer l'imbrication** en phase de création du plan logique (**décorrelation**)
- la requête obtenue est ensuite optimisée avec des techniques classiques
- décorrelation plus ou moins complexe selon les cas, et pas toujours possible

Optimisation de requêtes imbriquées - decorrelation

- Un exemple simple de décorrelation

```
SELECT DISTINCT nom
FROM Enseignant
WHERE EXISTS (
  SELECT *
  FROM Enseigne
  WHERE enseignant = eid
  AND année = 2015
)
```



```
SELECT DISTINCT nom
FROM Enseignant, Enseigne
WHERE enseignant = eid
AND année = 2015
```

Optimisation de requêtes imbriquées - decorrelation

- Un autre exemple simple

```
SELECT DISTINCT nom
FROM Etudiant
WHERE eid IN (
    SELECT étudiant
    FROM Exam
    WHERE date = '01/06/2015'
)
```



```
SELECT DISTINCT nom
FROM Etudiant, Exam
WHERE eid = étudiant
AND date = '01/06/2015'
```

Optimisation de requêtes imbriquées - decorrelation

- L'agrégation peut rendre la décorrelation plus complexe
- D'abord un exemple avec agrégation sans corrélation :

```
SELECT DISTINCT étudiant
```

```
FROM Exam
```

```
WHERE note > = (  
    SELECT AVG (note)
```

```
FROM Exam
```

```
)
```



```
SELECT DISTINCT étudiant
```

```
FROM Exam, ( SELECT AVG(note) as m  
              FROM Exam)
```

```
WHERE note > = m
```

Optimisation de requêtes imbriquées - decorrelation

- Un exemple avec aggregation et correlation :

```
SELECT DISTINCT étudiant, matière
FROM Exam AS E
WHERE note > = (
    SELECT AVG (note)
    FROM Exam
    WHERE matière = E.matière
)
```



```
SELECT DISTINCT étudiant, matière
FROM Exam, ( SELECT matière AS s,
              AVG(note) as m
            FROM Exam
            GROUP BY matière)
WHERE matière = s
AND note > = m
```

- Nécessaire de remonter les attributs de la sous-requêtes qui sont utilisés dans la corrélation, avant de pouvoir déplacer la sous-requête dans la clause FROM

Optimisation de requêtes imbriquées - decorrelation

- Quand la décorrelation est complexe
 - ▶ créer une **table temporaire** correspondante à la **sous-requête sans les conditions de corrélation**
 - ▶ ajouter ensuite les conditions de corrélation dans la requête principale

```
SELECT DISTINCT nom
FROM Enseignant
WHERE EXISTS (
  SELECT *
  FROM Enseigne
  WHERE enseignant = eid
  AND année = 2015
)
```



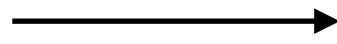
```
CREATE TABLE T AS
  SELECT *
  FROM Enseigne
  AND année = 2015

SELECT DISTINCT nom
FROM Enseignant,T
WHERE enseignant = eid
```

Optimisation de requêtes imbriquées - decorrelation

- Quand la décorrelation est complexe
 - ▶ créer une **table temporaire** correspondante à la **sous-requête sans les conditions de corrélation**
 - ▶ ajouter ensuite les conditions de corrélation dans la requête principale

```
SELECT DISTINCT étudiant, matière
FROM Exam AS E
WHERE note > = (
    SELECT AVG (note)
    FROM Exam
    WHERE matière = E.matière
)
```



```
CREATE TABLE T AS
```

```
SELECT matière AS s, AVG(note) as m
FROM Exam
GROUP BY matière
```

```
SELECT DISTINCT étudiant, matière
FROM Exam, T
WHERE matière = s
AND note > = m
```

Optimisation de requêtes imbriquées

- La décorrelation est difficile dans le cas général
- Plusieurs systèmes d'optimisation la réalisent en forme très limitée
- Le plus souvent
 - ▶ chaque bloc (i.e. sous-requête) de la requête est optimisé séparément
 - ▶ évaluation corrélée pour la requête obtenue
- Conséquence :
 - ▶ l'évaluation de requêtes imbriquées est en général très inefficace
 - ▶ l'efficacité est inversement proportionnelle au nombre de blocs imbriqués
 - ▶ bonne pratique pour l'utilisateur :
éviter le plus possible d'écrire des requêtes imbriquées !

Plans d'exécution en SQL

- SQL permet de visualiser le plan d'exécution d'une requête avant de l'exécuter
- Syntaxe dans la plupart des SGBD : `EXPLAIN <requete>`

- Ex.

```
EXPLAIN SELECT nom, titre
FROM Enseignant, Cours, Enseigne
WHERE univ = 'P7'      AND année = 2015
AND cid = cours      AND eid = enseignant
```

Construit un plan d'exécution et le visualise sans exécuter la requête

- La syntaxe des plans d'exécutions affichés varie d'un SGBD à l'autre

Plans d'exécution en SQL

- PostgreSQL affiche l'arbre du plan d'exécution par lignes
 - ▶ une ligne pour chaque noeud de l'arbre, flèches = relation enfant

Ex. `EXPLAIN SELECT *`
`FROM T1, T2`
`WHERE T1.a = T2.a;`

QUERY PLAN

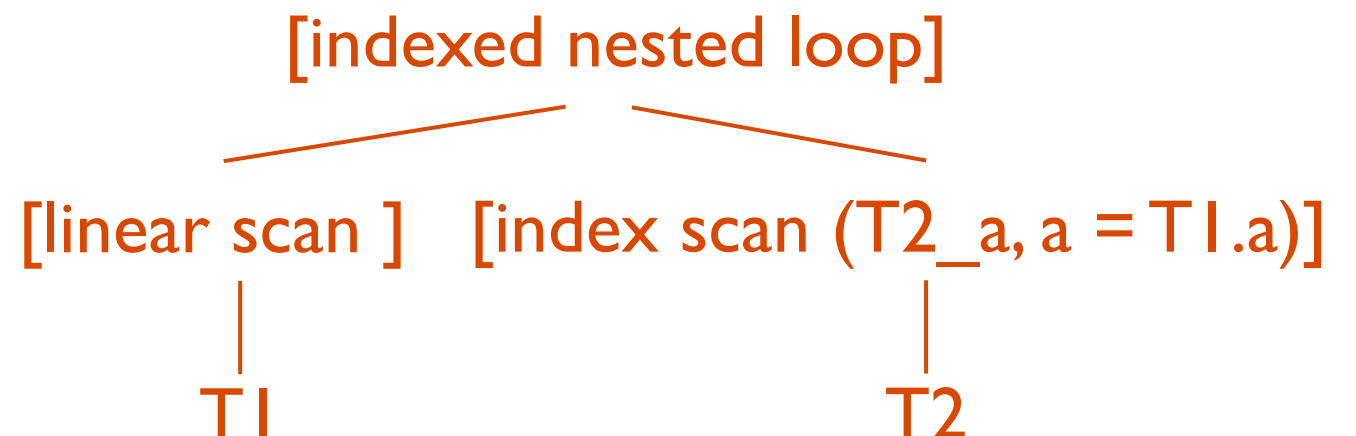
Nested Loop (cost=2.37..553.11 rows=106 width=488)

-> **Seq Scan** on T1 (cost=0.00..483.00 rows=7033 width=244)

-> **Index Scan** using T2_a on T2 (cost=0.00..3.01 rows=1 width=244)

Index Cond: (a = T1.a)

- ▶ représente le plan :



Plans d'exécution en SQL

- PostgreSQL représente pour chaque noeud de l'arbre des informations supplémentaires
 - **Seq Scan** on T1 (cost=0.00..483.00 rows=7033 width=244)
 - ▶ une estimation du coût de l'opérateur pour obtenir le premier résultat
 - ▶ une estimation du coût de l'opérateur (pour obtenir tous les résultats)
 - ▶ une estimation du nombre de lignes du résultat
 - ▶ une estimation de la taille moyenne des tuples dans le résultat
- Les coûts sont exprimés dans l'unité de coût adoptée par PostgreSQL, seuls les valeurs relatifs sont significatifs

Plans d'exécution en SQL

- L'ensemble des opérateurs physiques varie aussi selon le SGBD
- En PostgreSQL :
 - ▶ Seq Scan ([linear scan]), Index Scan, Nested Loop, Hash Join etc...
 - ▶ Une variante de Index Scan : Bitmap Index Scan opère en deux phases
 1. première phase (Bitmap Index Scan) : à travers l'index récupère les pointeurs aux tuples et les trie par bloc du fichier de données
 2. deuxième phase (Bitmap Heap Scan) : récupère les tuples pointées par ces pointeurs dans le fichier de données
 - la phase de tri permet d'accéder une seule fois à chaque bloc du fichier de données contenant des tuples pointés

Plans d'exécution en SQL

- On peut améliorer les performances du système d'optimisation en demandant périodiquement de recalculer les statistiques sur les tables :
- Syntaxe PostgreSQL :

`ANALYZE` recalcule les statistiques sur toutes les tables

`ANALYZE Table` sur une table en particulier

`ANALYZE Table (att1,...,attk)` sur les attributs spécifiés d'une table