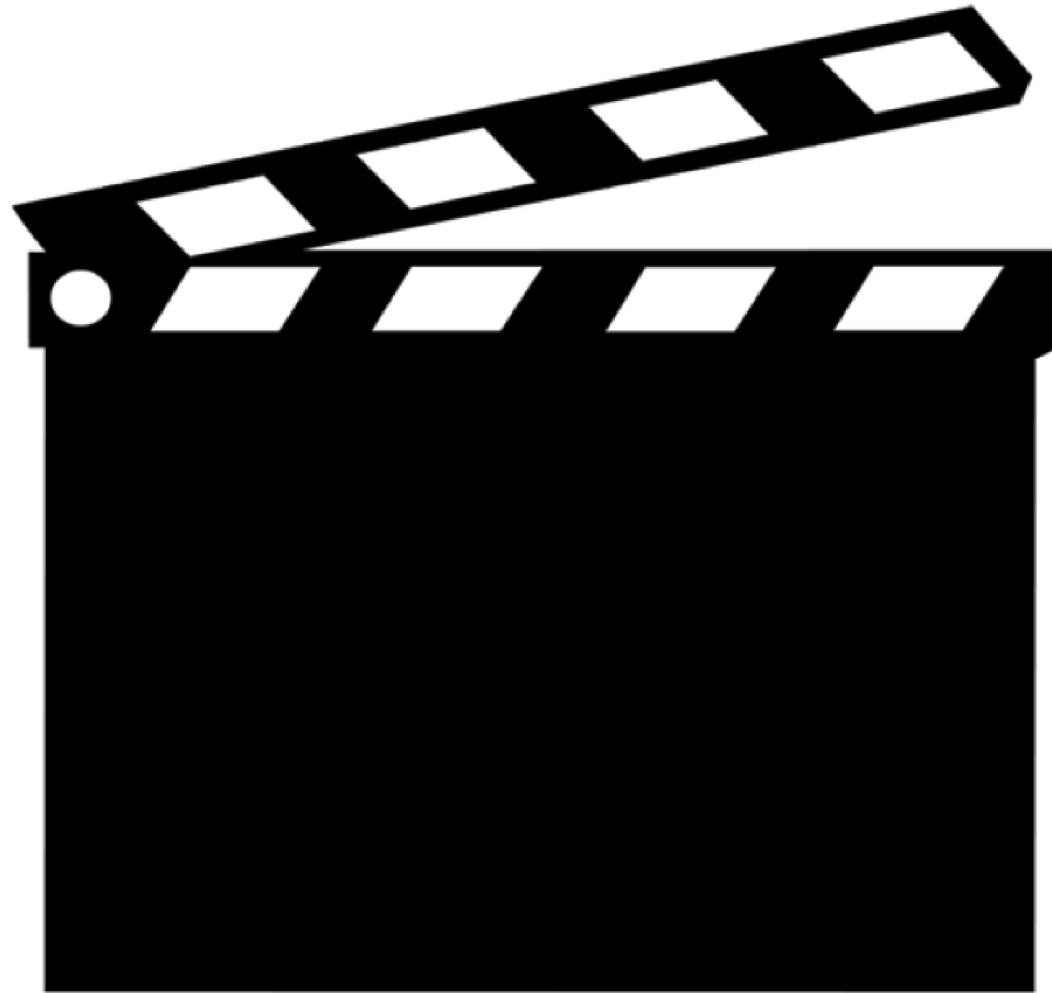
C018SA-W3-S1



Semaine 3 : Exécution et optimisation

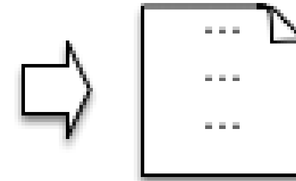
1. **Introduction**
2. Réécriture algébrique
3. Opérateurs
4. Plans d'exécution
5. Tri et hachage
6. Algorithmes de jointure
7. Optimisation

Philippe Rigaux

Inria informatiques mathématiques

Le problème étudié

```
select a1, a2, …
from T1, T2, …
where …
```



Forme *déclarative*

Forme *opératoire*

Résultat

Une requête SQL est **déclarative**. Elle ne dit pas **comment** calculer le résultat.

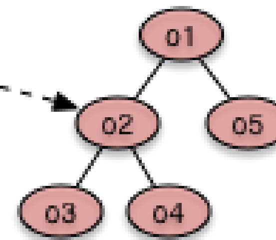
Nous avons besoin d'une **forme opératoire** : un programme.

La notion de plan d’exécution

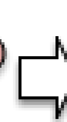
```
select a1, a2, …
from T1, T2, …
where …
```

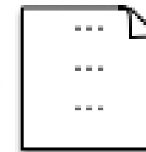
Forme
déclarative

?



Forme
opératoire



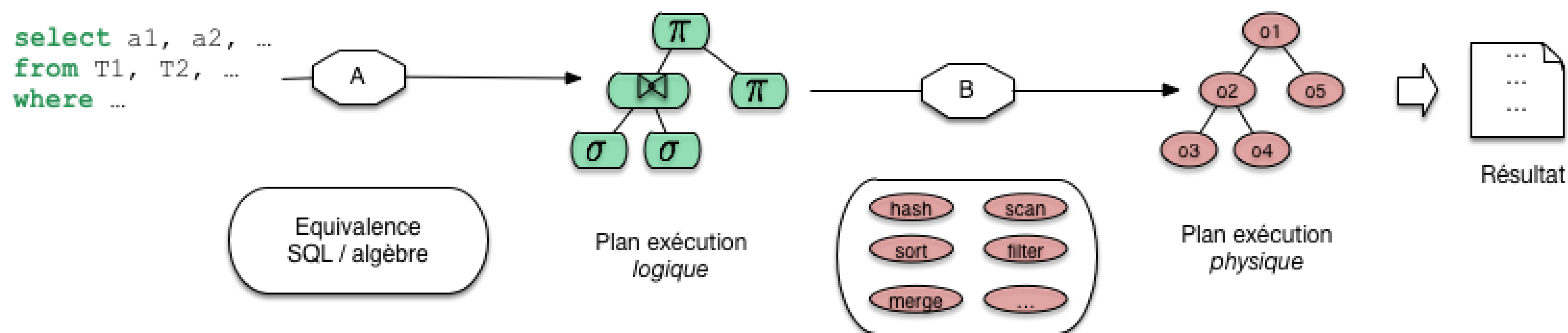


Résultat

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 SQL au plan d'exécution

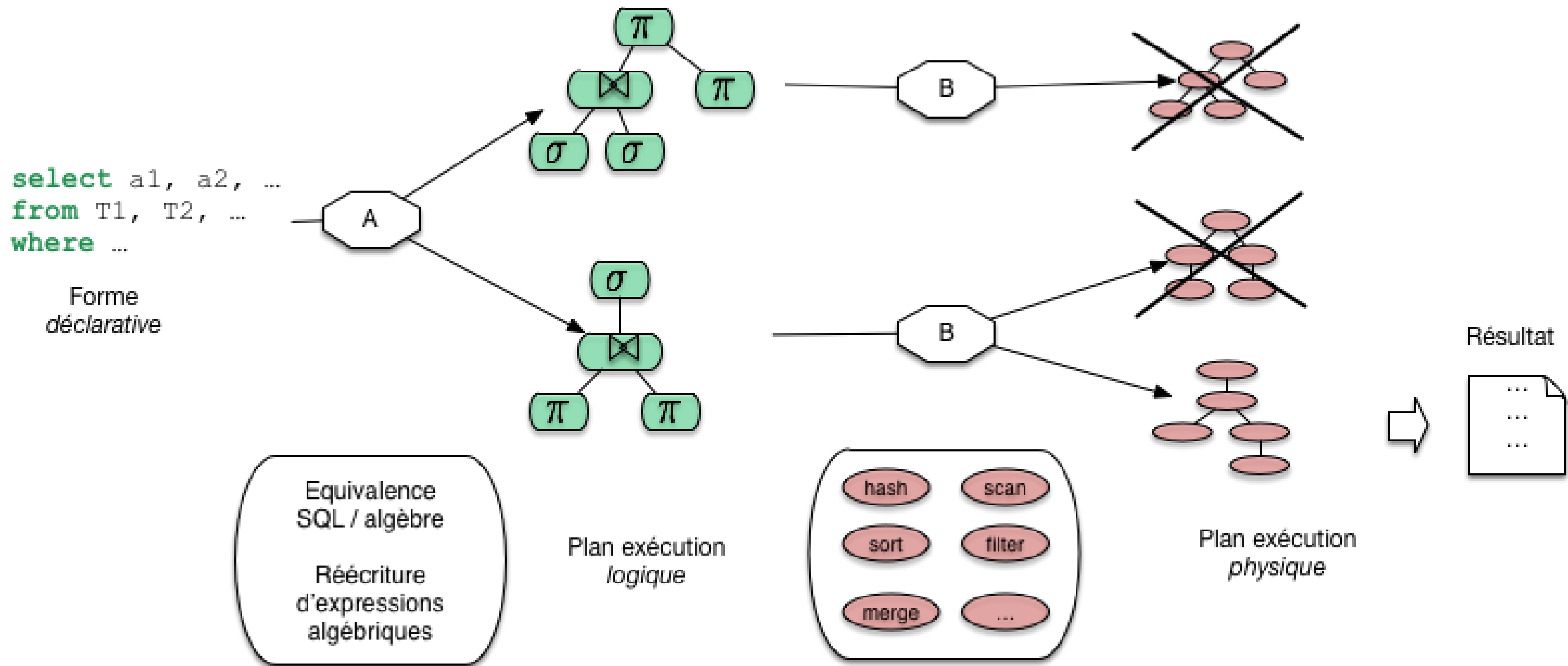


Deux étapes :

1. (A) plan d'exécution **logique** (l'algèbre) ;
2. (B) plan d'exécution **physique** (opérateurs).

Le SGBD s'appuie sur un **catalogue** d'opérateurs.

En quoi consiste l'optimisation ?



À chaque étape, plusieurs choix. Le système les évalue et choisit le « meilleur ».

Les séquences

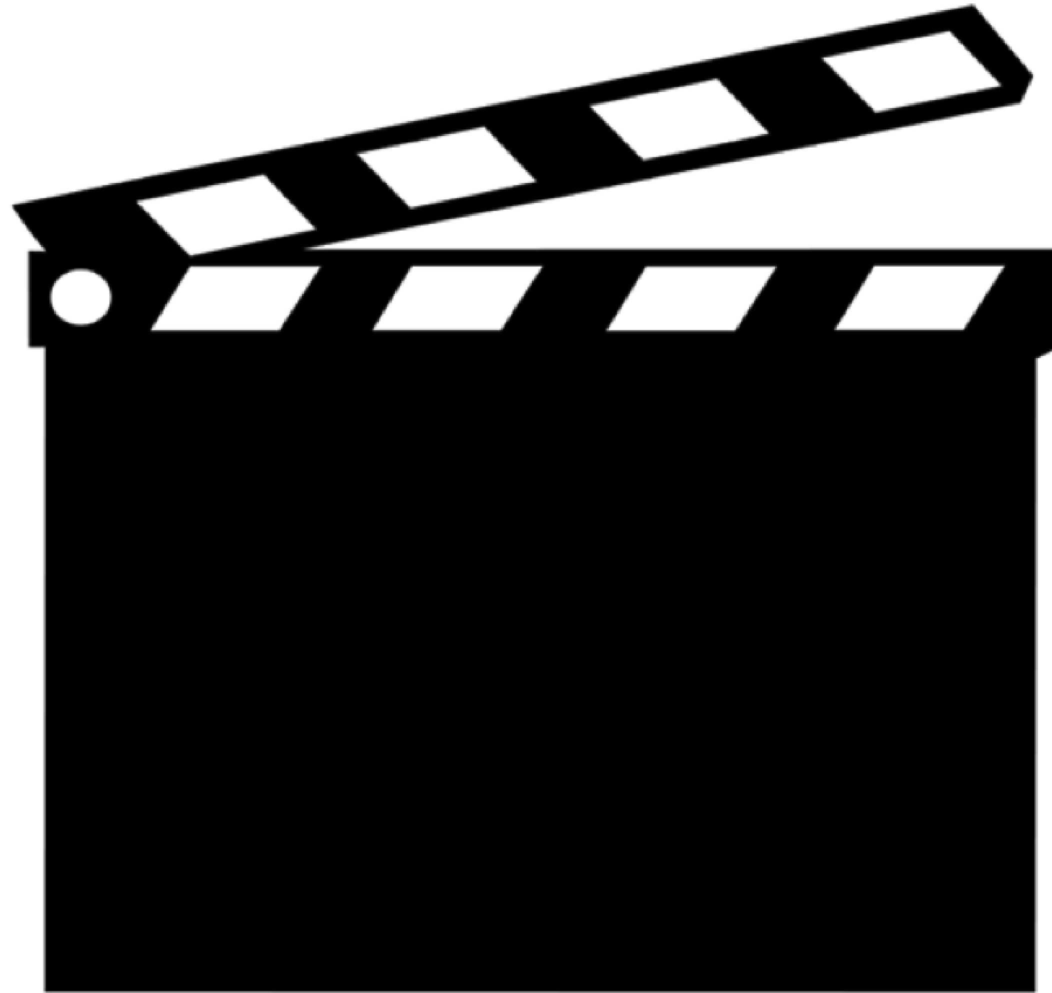
1. Introduction
2. Réécriture algébrique
3. Opérateurs
4. Plans d'exécution
5. Tri et hachage
6. Algorithmes de jointure
7. Optimisation

Les séquences

1. Introduction
2. Réécriture algébrique
3. Opérateurs
4. Plans d'exécution
5. Tri et hachage
6. Algorithmes de jointure
7. Optimisation

Merci !

C018SA-W3-S2



Semaine 3 : Exécution et optimisation

1. Introduction
2. **Réécriture algébrique**
3. Opérateurs
4. Plans d'exécution
5. Tri et hachage
6. Algorithmes de jointure
7. Optimisation

Philippe Rigaux

Inria informatiques mathématiques

De SQL vers une forme opératoire : l'algèbre

Titre des films parus en 1958, où l'un des acteurs joue le rôle de John Ferguson.

En SQL :

```
select titre
from   Film f, Role r
where  nom_role ='Ferguson'
and    f.id = r.id_ilm
and    f.annee = 1958
```

Deux sélections (l'année, le rôle), une jointure, une projection.

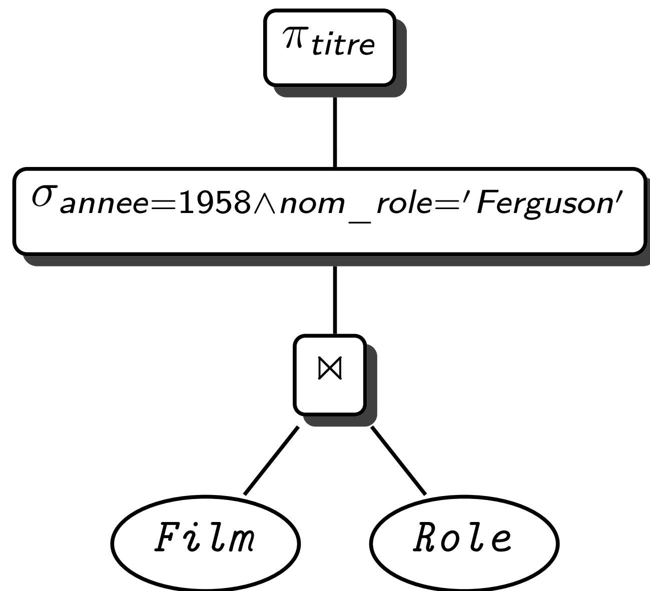
Le Plan d'Exécution Logique (PEL)

En algèbre, sous une forme normalisée projection-sélection-jointure :

$$\pi_{titre}(\sigma_{annee=1958 \wedge nom_role='\mathrm{Ferguson}'}(Film \bowtie_{id=id_film} Role))$$

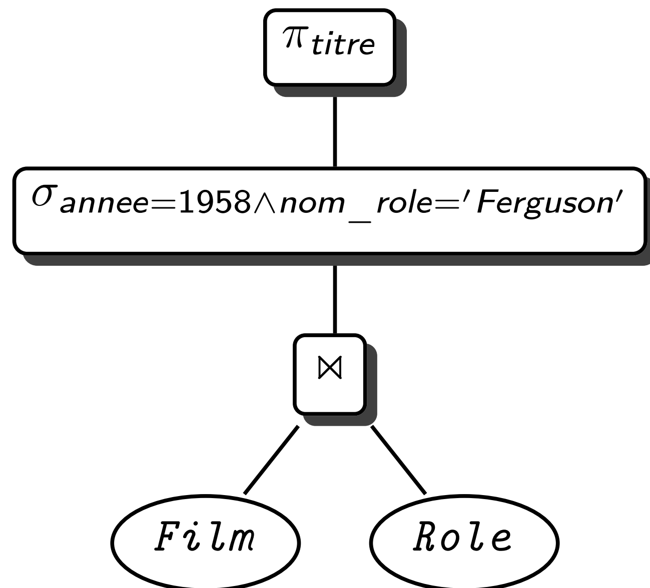
Le Plan d'Exécution Logique (PEL)

La même expression, sous forme d'un arbre.



Le Plan d'Exécution Logique (PEL)

La même expression, sous forme d'un arbre.



Est-ce le seul ? Non : des **réécritures** vont nous permettre d'en trouver d'autres.

Réécritures

Il y a plusieurs expressions **équivalentes** pour une même requête.

On peut explorer ces expressions grâce à des règles de réécriture. Exemple

1. **Commutativité des jointures** : $R \bowtie S \equiv S \bowtie R$
2. **Regroupement des sélections** : $\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$
3. **Commutativité de σ et de π** : $\pi_{A_1,A_2,\ldots A_p}(\sigma_{A_i='a'}(R)) \equiv \sigma_{A_i='a'}(\pi_{A_1,A_2,\ldots A_p}(R))$
4. **Commutativité de σ et de $\bowtie$** : $\sigma_{A='a'}(R[\ldots A \ldots] \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$
5. etc.

L'application dirigée d'une règle d'équivalence (réécriture) transforme une expression e en une expression e' **équivalente**.

Ce que fait l'optimiseur

Trouve les expressions équivalentes, évalue leur coût et choisit la meilleure.

Important : on ne peut pas **énumérer** tous les plans possibles (trop long) : on applique des **heuristiques**.

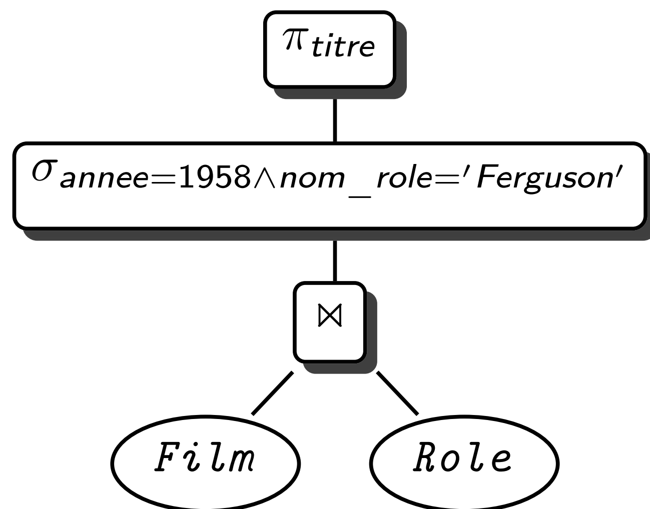
Heuristique classique : **réduire la taille des données**

- en filtrant les nuplets par des sélections
- en les simplifiant par des projections

dès que possible.

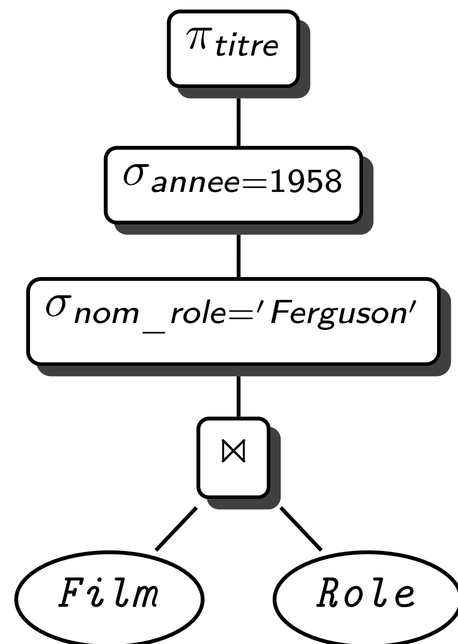
Illustration

Reprenons : le film paru en 1958 avec un rôle 'Ferguson'.



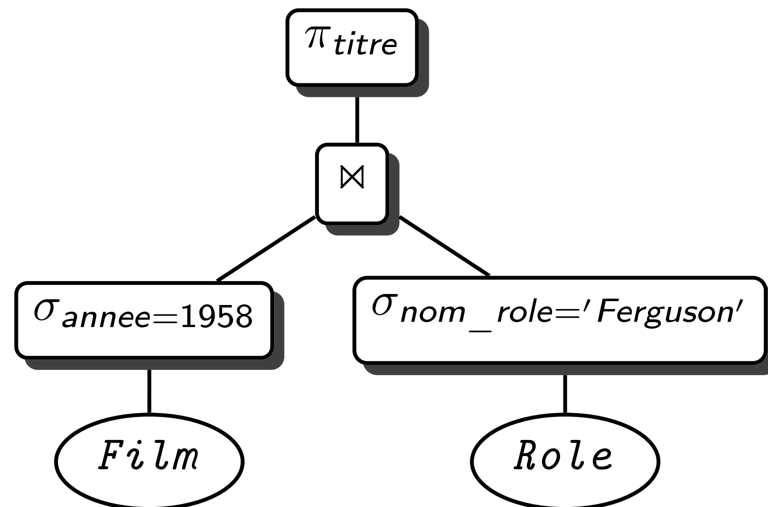
Illustration

Première étape : je sépare les sélections.



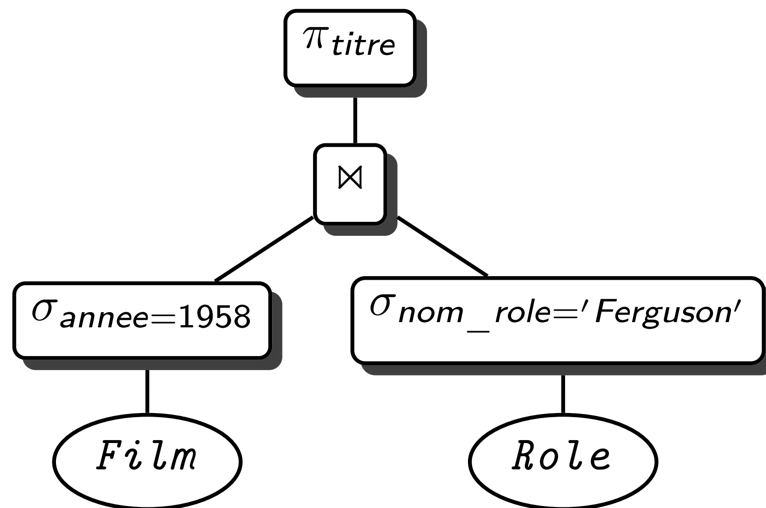
Illustration

Puis on pousse chaque sélection vers sa table.



Illustration

Puis on pousse chaque sélection vers sa table.



À ce stade : reste à choisir les chemins d'accès et les algorithmes de jointure.

Résumé : la réécriture algébrique

Principes essentiels :

1. L'algèbre permet d'obtenir une version opératoire de la requête.
2. Les équivalences algébriques permettent d'explorer un ensemble de plans.
3. L'optimiseur évalue le coût de chaque plan.

Bien retenir

- **Heuristique** : on ne peut pas **tout explorer**
- **Nécessaire mais pas suffisant** : il reste à choisir le bon algorithme pour chaque opération.

Résumé : la réécriture algébrique

Principes essentiels :

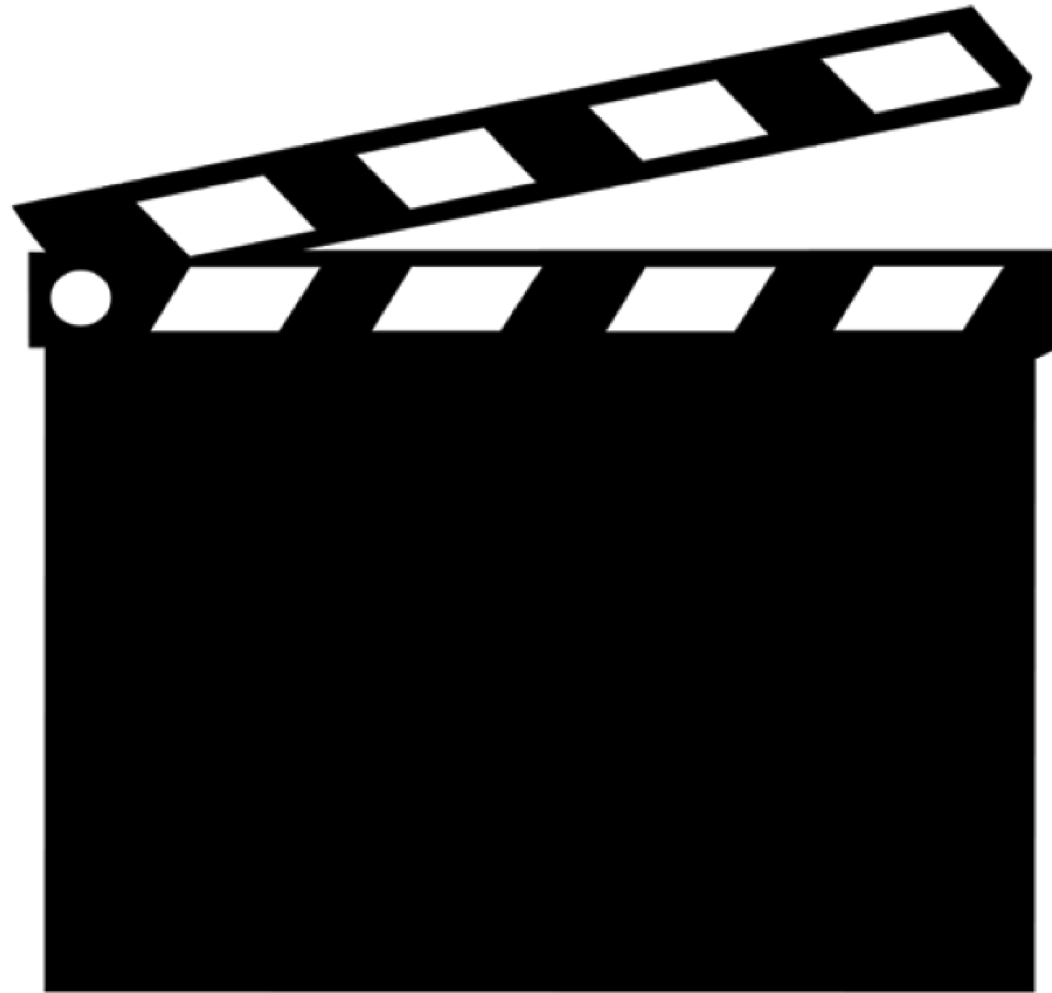
1. L'algèbre permet d'obtenir une version opératoire de la requête.
2. Les équivalences algébriques permettent d'explorer un ensemble de plans.
3. L'optimiseur évalue le coût de chaque plan.

Bien retenir

- **Heuristique** : on ne peut pas **tout explorer**
- **Nécessaire mais pas suffisant** : il reste à choisir le bon algorithme pour chaque opération.

Merci !

C018SA-W3-S3



Semaine 3 : Exécution et optimisation

1. Introduction
2. Réécriture algébrique
3. **Opérateurs**
4. Plans d'exécution
5. Tri et hachage
6. Algorithmes de jointure
7. Optimisation

Philippe Rigaux

Inria informatiques mathématiques

Plan d'exécution et opérateurs

Rappel : un **plan d'exécution** est un arbre constitué **d'opérateurs** échangeant des **flux de données**.

Plan d'exécution et opérateurs

Rappel : un **plan d'exécution** est un arbre constitué **d'opérateurs** échangeant des **flux de données**.

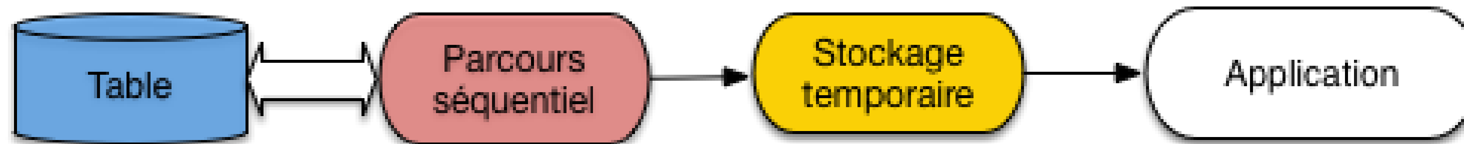
Caractéristiques des opérateurs

- ont une forme générique (**itérateurs**) ;
- fournissent une tâche spécialisée (cf. l'algèbre relationnelle)
- peuvent être ou non **bloquants**.

Un petit nombre suffit pour couvrir SQL !

Mode naïf : matérialisation

Dans ce mode, un opérateur calcule son résultat, puis le transmet.

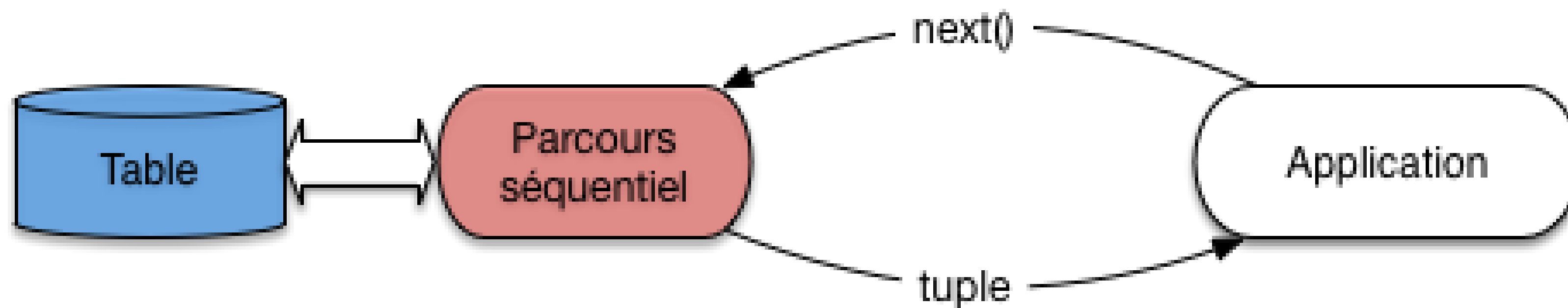


Deux inconvénients :

- Consomme de la mémoire.
- Introduit un temps de **latence**.

La bonne solution : pipelinage

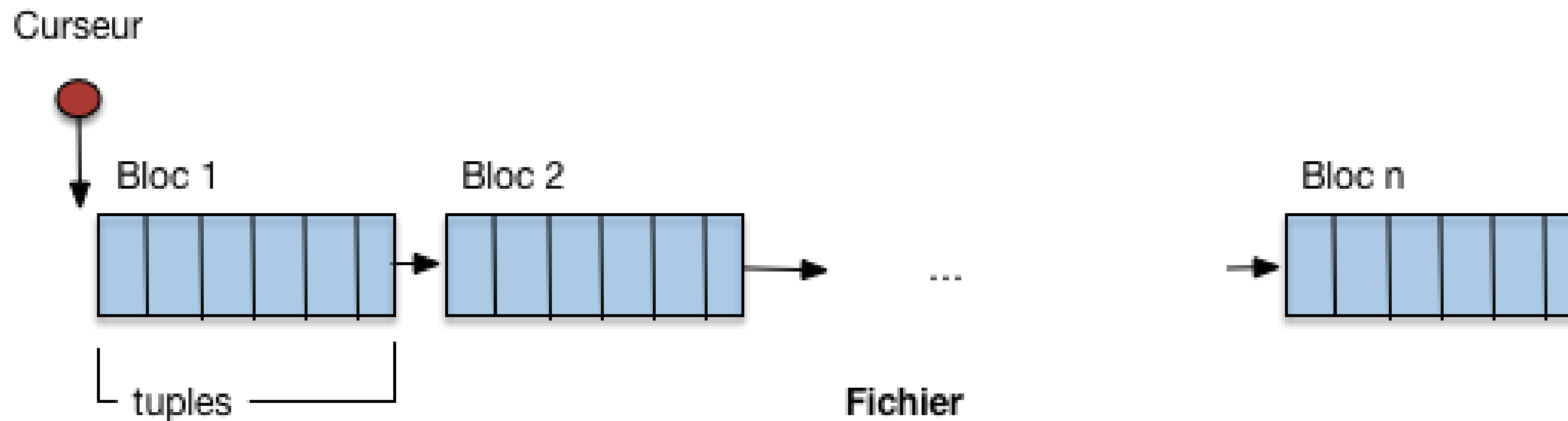
Le résultat est produit **à la demande**.



- **Plus de stockage intermédiaire**.
- **Latence minimale**.

Illustration : l'opérateur `FullScan`

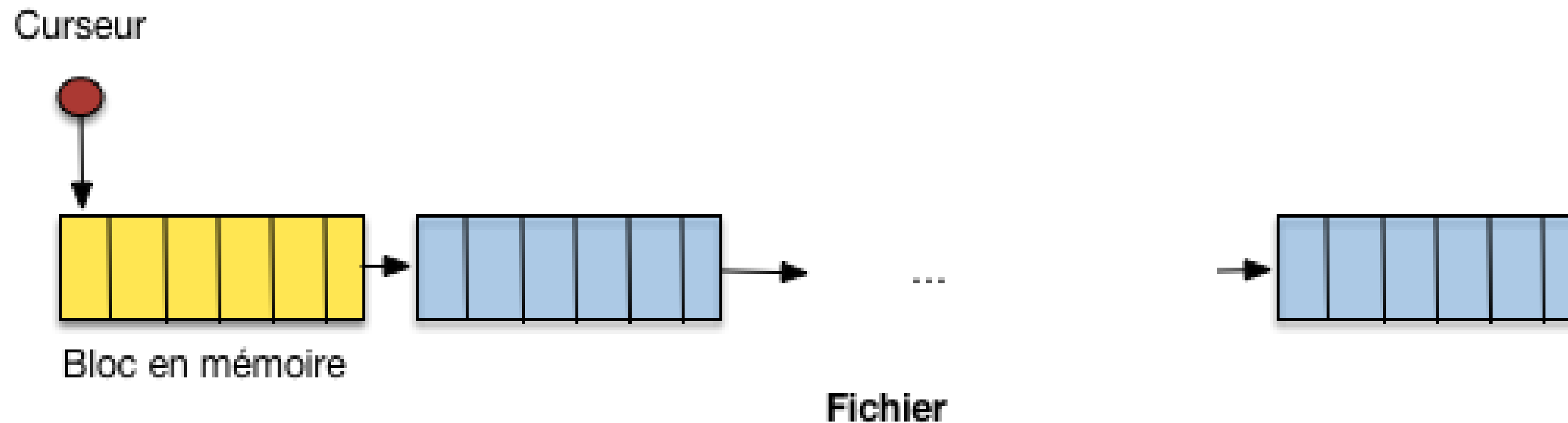
Au moment du `open()`, le curseur est positionné **avant** le premier nuplet.



`open()` désigne la phase d'initialisation de l'opérateur.

Illustration : l'opérateur `FullScan`

Le premier `next()` entraîne l'accès au premier bloc, placé en mémoire.



Le curseur se place sur le premier nuplet, qui est retourné comme résultat. **Le temps de réponse est minimal**.

Illustration : l'opérateur `FullScan`

Le deuxième `next()` avance d'un cran dans le parcours du bloc.

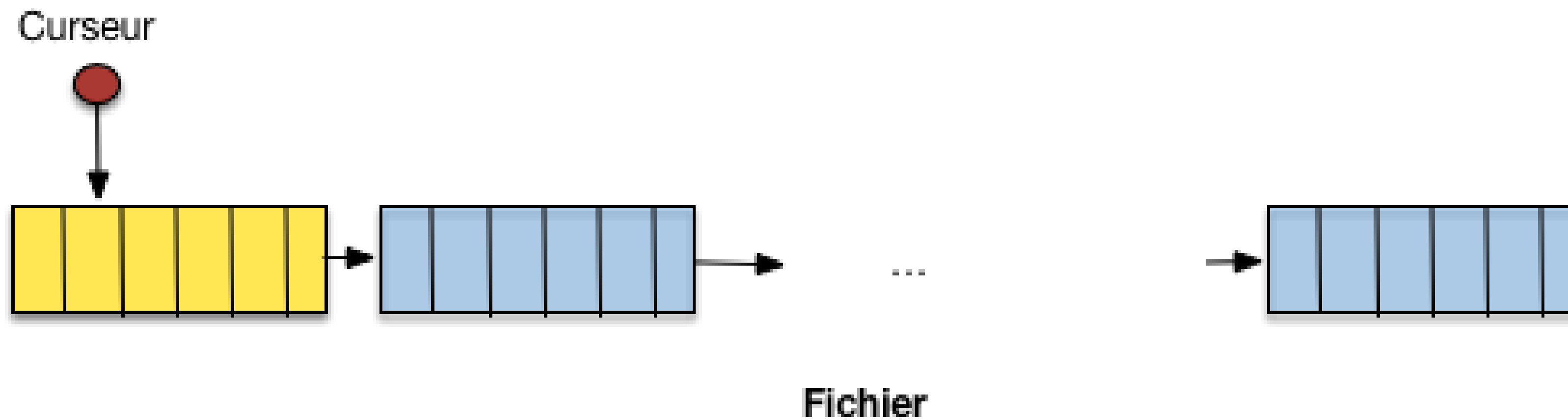


Illustration : l'opérateur `FullScan`

Après plusieurs `next()`, le curseur est positionné sur le dernier nuplet du bloc.

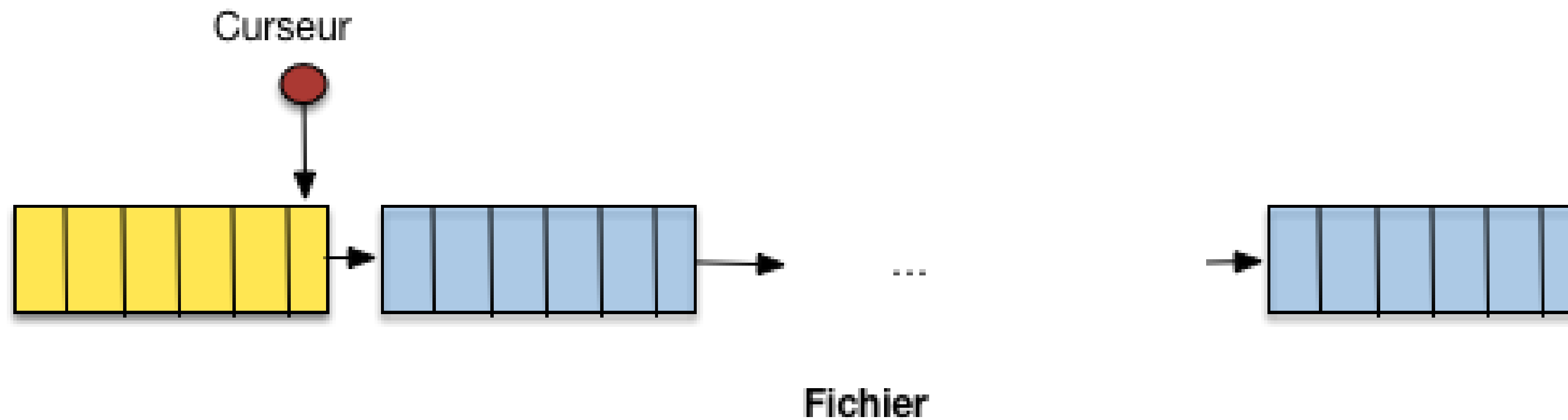
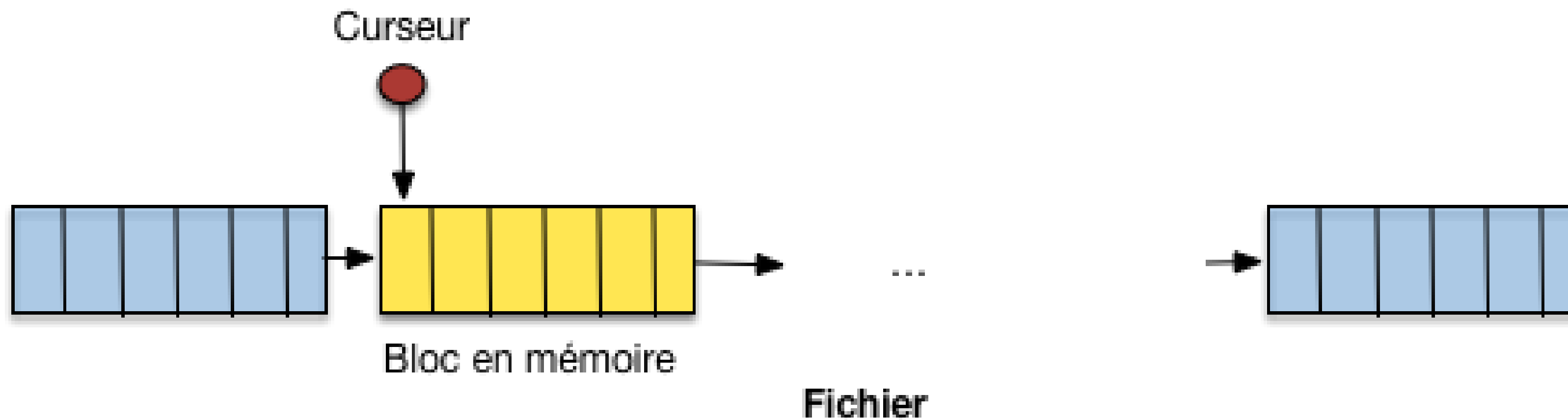


Illustration : l'opérateur `FullScan`

L'appel suivant à `next()` charge le second bloc en mémoire.



Bilan : besoin en mémoire réduit (1 bloc) ; temps de réponse très court.

Opérateur bloquant

Tous les opérateurs peuvent-ils fonctionner en mode pipelinage ?

```
select min(date) from T
```

On ne peut pas produire un nuplet avant d'avoir examiné **toute** la table.

Il faut alors introduire un opérateur **bloquant**, avec une latence forte.

Avec un opérateur bloquant, on **additionne** le temps d'exécution et le temps de traitement.

Résumé : opérateurs d'exécution

Principes essentiels :

1. **Itération** : dans tous les cas, un opérateur produit les nuplets à la demande.
2. **Pipelinage** : si possible, le résultat est calculé au fur et à mesure.
3. **Matérialisation** : parfois le résultat intermédiaire doit être calculé et stocké.

Bien distinguer

- **Temps de réponse** : temps pour obtenir le premier nuplet.
- **Temps d'exécution** : temps pour obtenir tous les nuplets.

Résumé : opérateurs d'exécution

Principes essentiels :

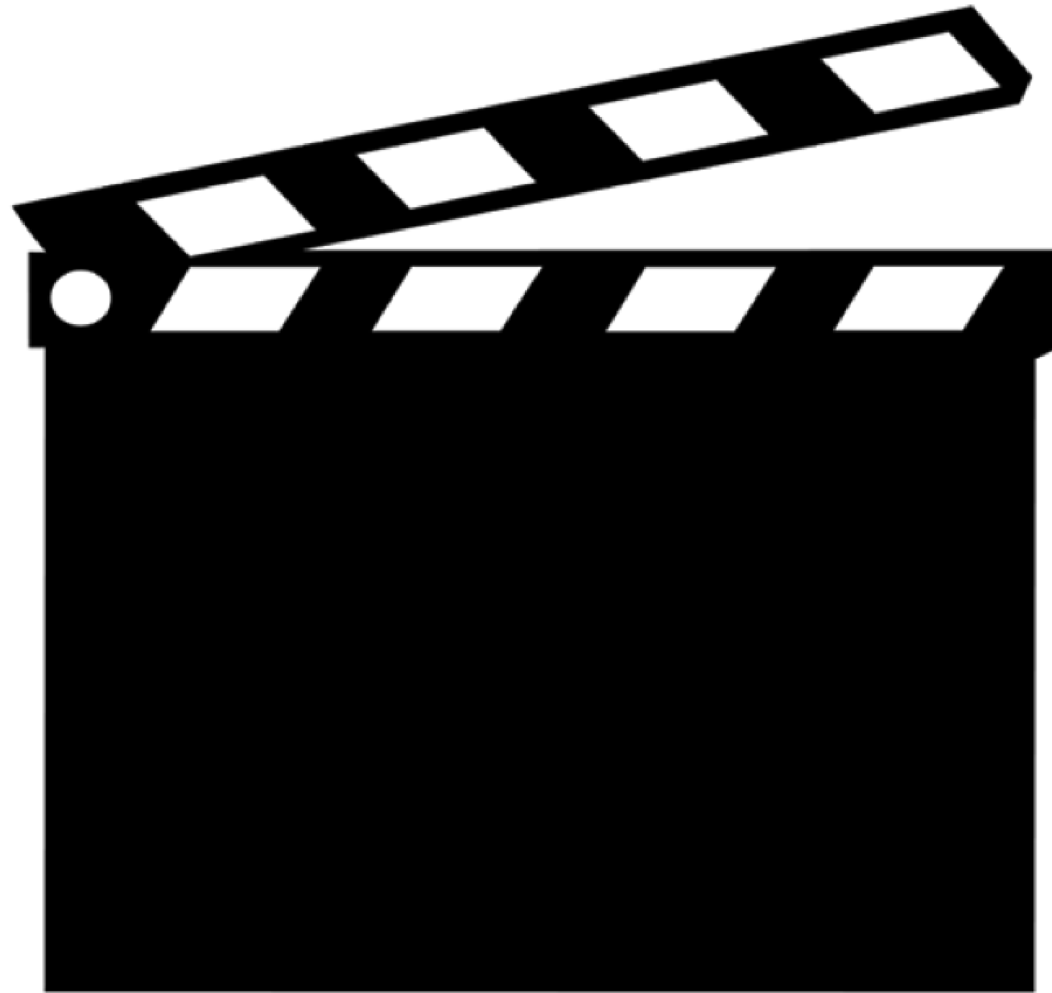
1. **Itération** : dans tous les cas, un opérateur produit les nuplets à la demande.
2. **Pipelinage** : si possible, le résultat est calculé au fur et à mesure.
3. **Matérialisation** : parfois le résultat intermédiaire doit être calculé et stocké.

Bien distinguer

- **Temps de réponse** : temps pour obtenir le premier nuplet.
- **Temps d'exécution** : temps pour obtenir tous les nuplets.

Merci !

C018SA-W3-S4



Semaine 3 : Exécution et optimisation

1. Introduction
2. Réécriture algébrique
3. Opérateurs
4. **Plans d'exécution**
5. Tri et hachage
6. Algorithmes de jointure
7. Optimisation

Philippe Rigaux

Inria informatiques mathématiques

Opérateur = itérateur

Tout opérateur est implanté sous forme d'un **itérateur**. Trois fonctions :

- `open` : initialise les ressources et positionne le curseur ;
- `next` : ramène l'enregistrement courant se place sur l'enregistrement suivant ;
- `close` : libère les ressources ;

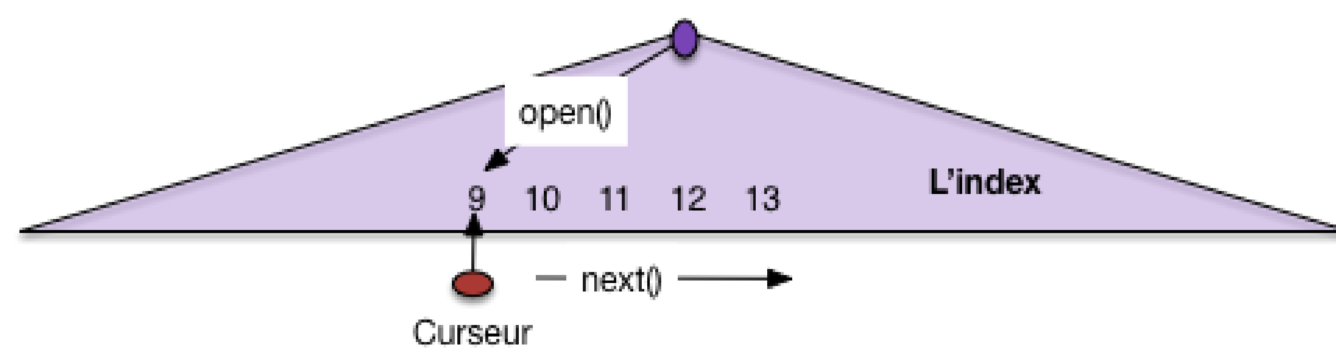
Échanges :

- Un itérateur **consomme** des nuplets d'autres itérateurs **source**.
- Un itérateur **produit** des nuplets pour un autre itérateur (ou pour l'application).

Exemple : parcours d'index (`IndexScan`)

Rappel : index = arbre B.

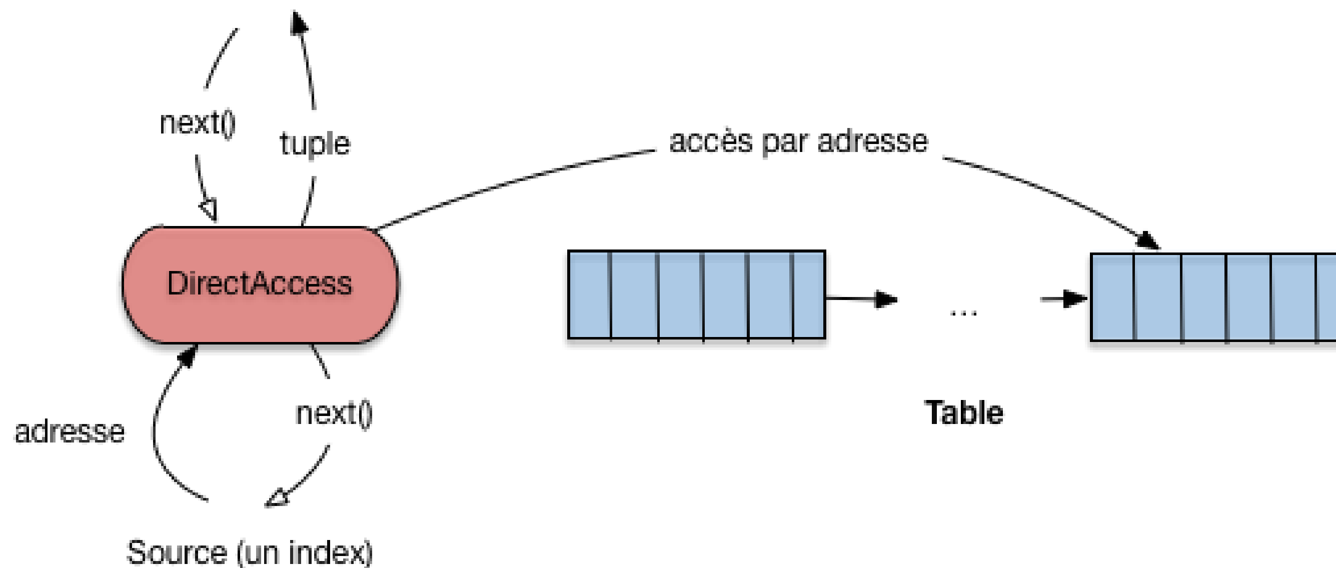
- Pendant le `open()` : parcours de la racine vers la feuille.
- À chaque appel à `next()` : parcours en séquence des feuilles.



Efficacité : très efficace, quelques lectures logiques (index en mémoire)

Accès par adresse : `DirectAccess`

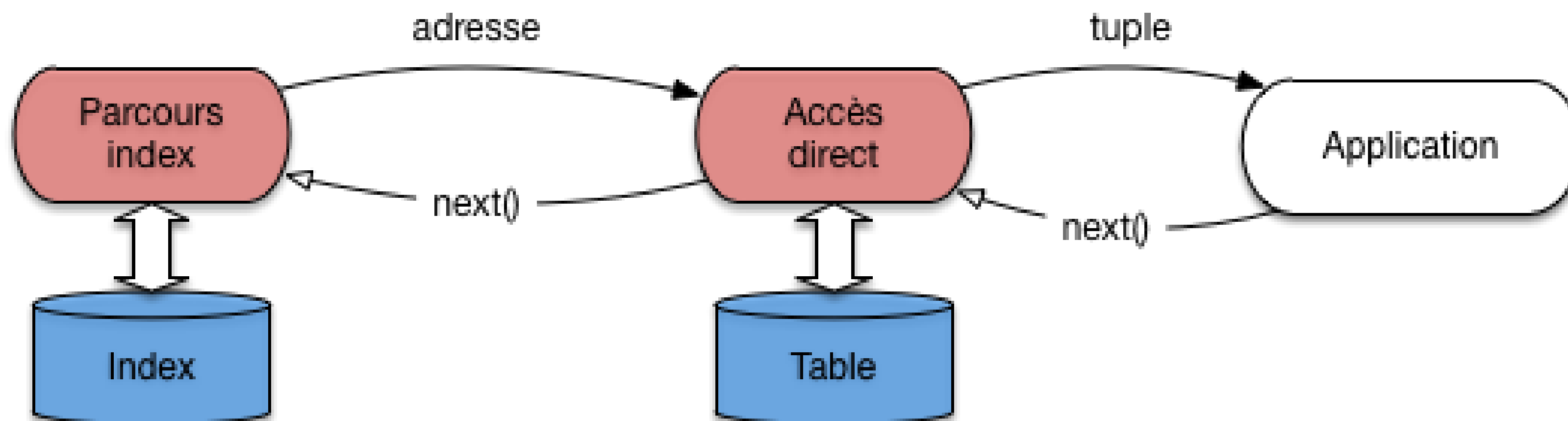
- Pendant le `open()` : rien à faire.
- À chaque appel à `next()` : on reçoit une adresse, on produit un nuplet.



Très efficace : un accès bloc, souvent en mémoire.

Plan d'exécution

Un plan d'exécution connecte les opérateurs. Ici, recherche avec index.



Le pipelinage reste complet.

Un exemple de base

Nous allons étudier les plans permettant d'exécuter les requêtes **mono-table**.

```
select a1, a2, ..., an
from T
where condition
```

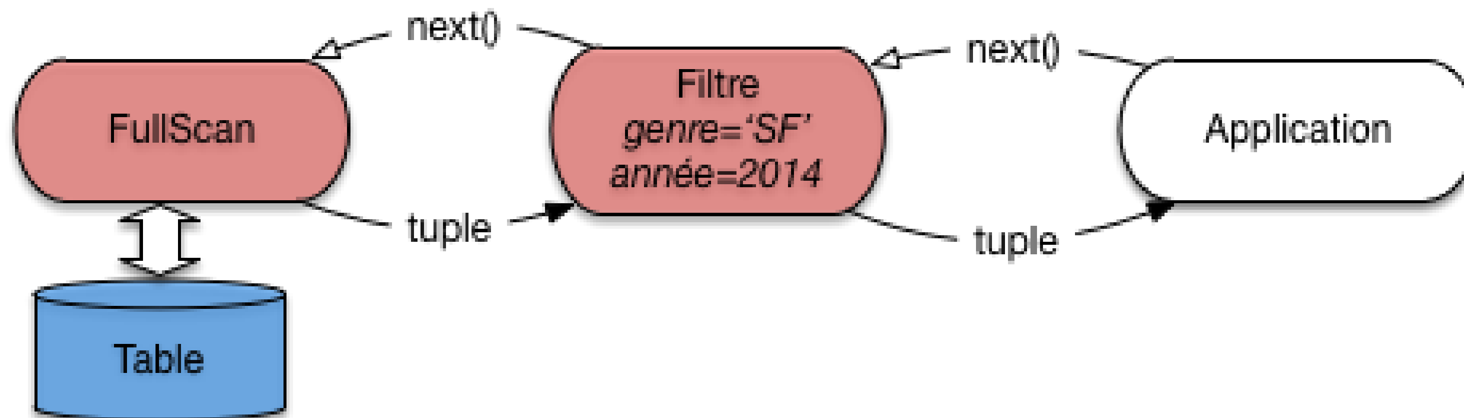
De quels opérateurs a-t-on besoin ?

- [FullScan] : parcours séquentiel de la table (déjà vu).
- [IndexScan] : parcours d'un index (si disponible).
- [DirectAccess] : accès **par adresse** à un nuplet.
- [Filter] : test de la condition.

Nous obtenons **deux** plans d'exécution possibles.

Premier plan d'exécution : sans index

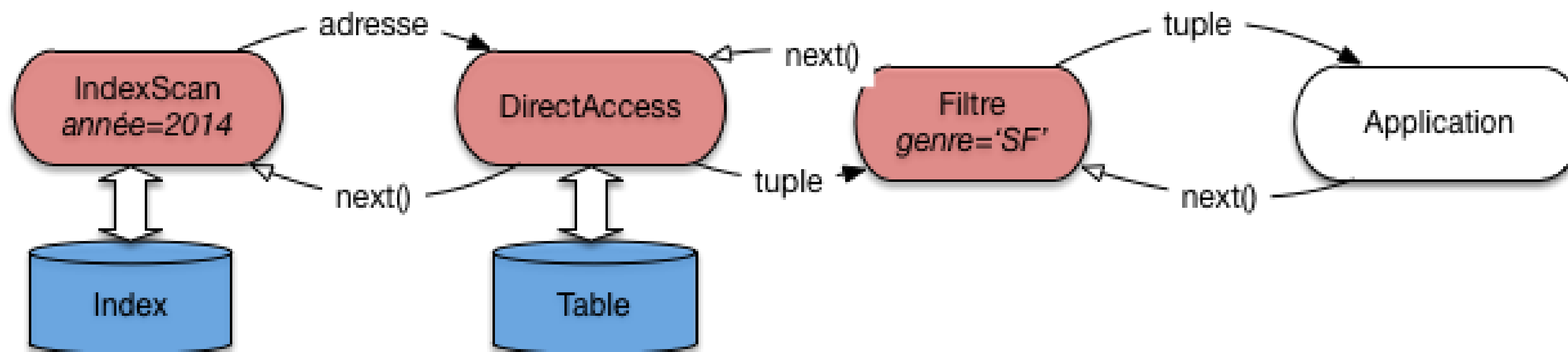
```sql
select titre from Film where genre='SF' and annee = 2014
```



N'utilise pas d'index

Second plan d'exécution : avec index

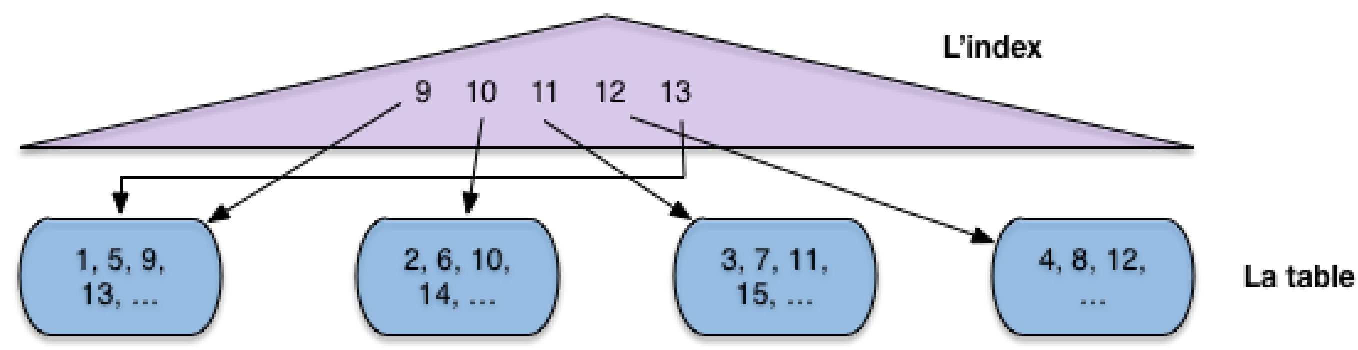
```
select titre from Film where genre='SF' and annee = 2014
```



Utilise un index sur l'année.

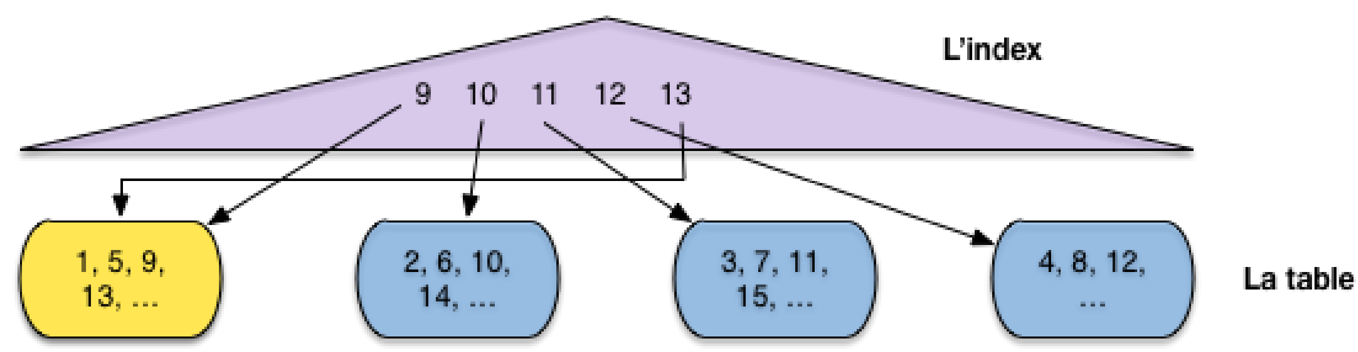
Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.



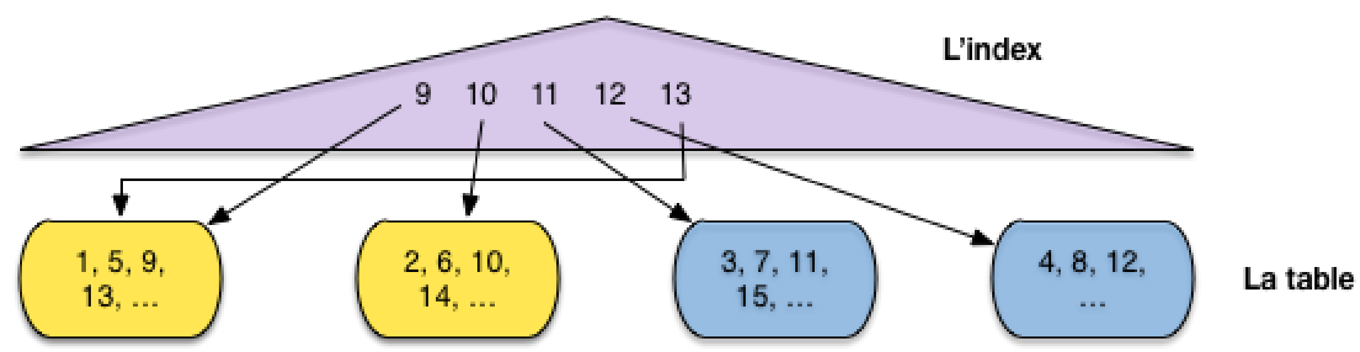
Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.



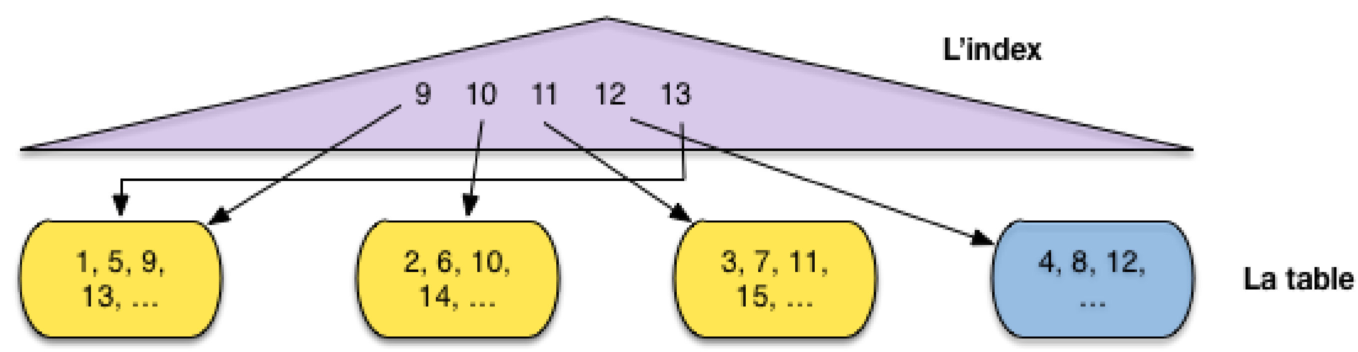
Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.



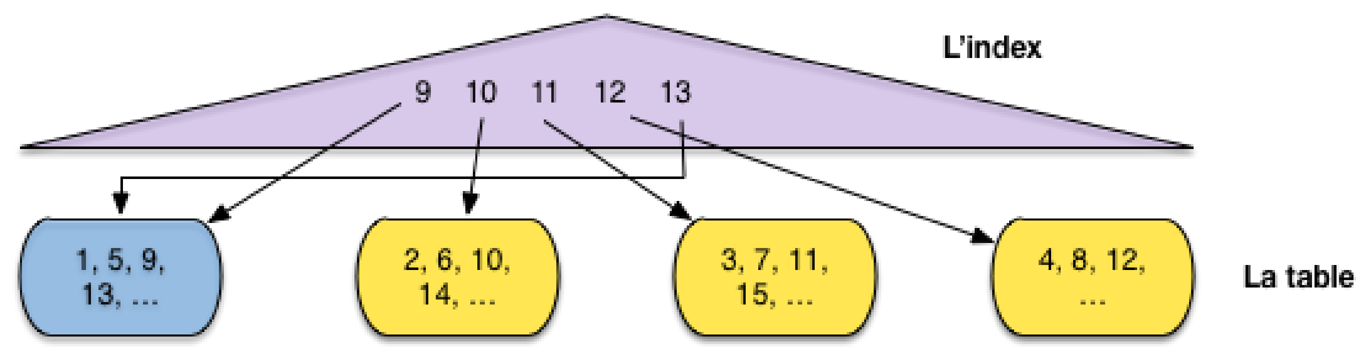
Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.



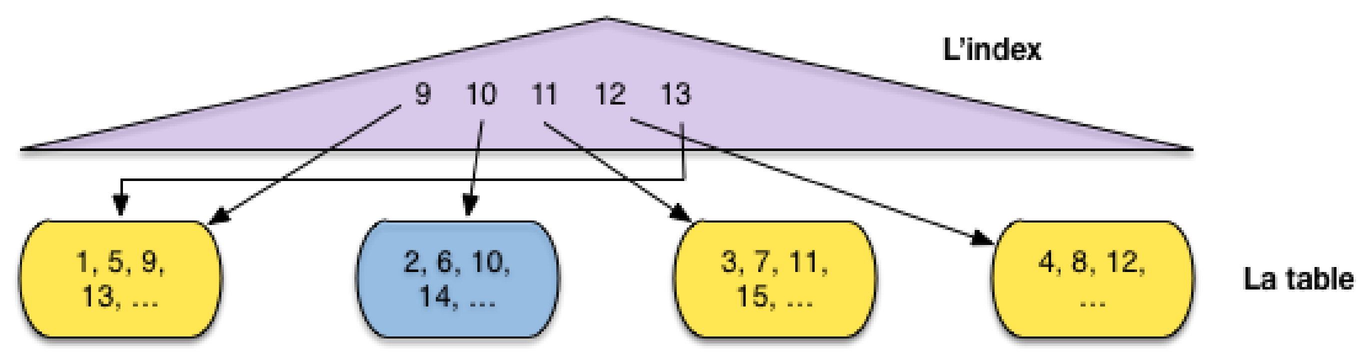
Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.



Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.



Index ou pas index ?

Recherche des nuplets entre 9 et 13, avec 3 blocs en mémoire.

Avec l'index, il faut lire 5 blocs ! Parcours séquentiel bien préférable.

Un cas extrême ! **En pratique** : le SGBD décide en fonction des statistiques et des ressources disponibles.

Résumé : plans pour requêtes mono-table

Permier aperçu de l'optimisation

- Le système a le choix entre plusieurs plans possibles.
- Distinguer le plus efficace n'est pas toujours trivial.
- Le choix peut changer selon le contexte.

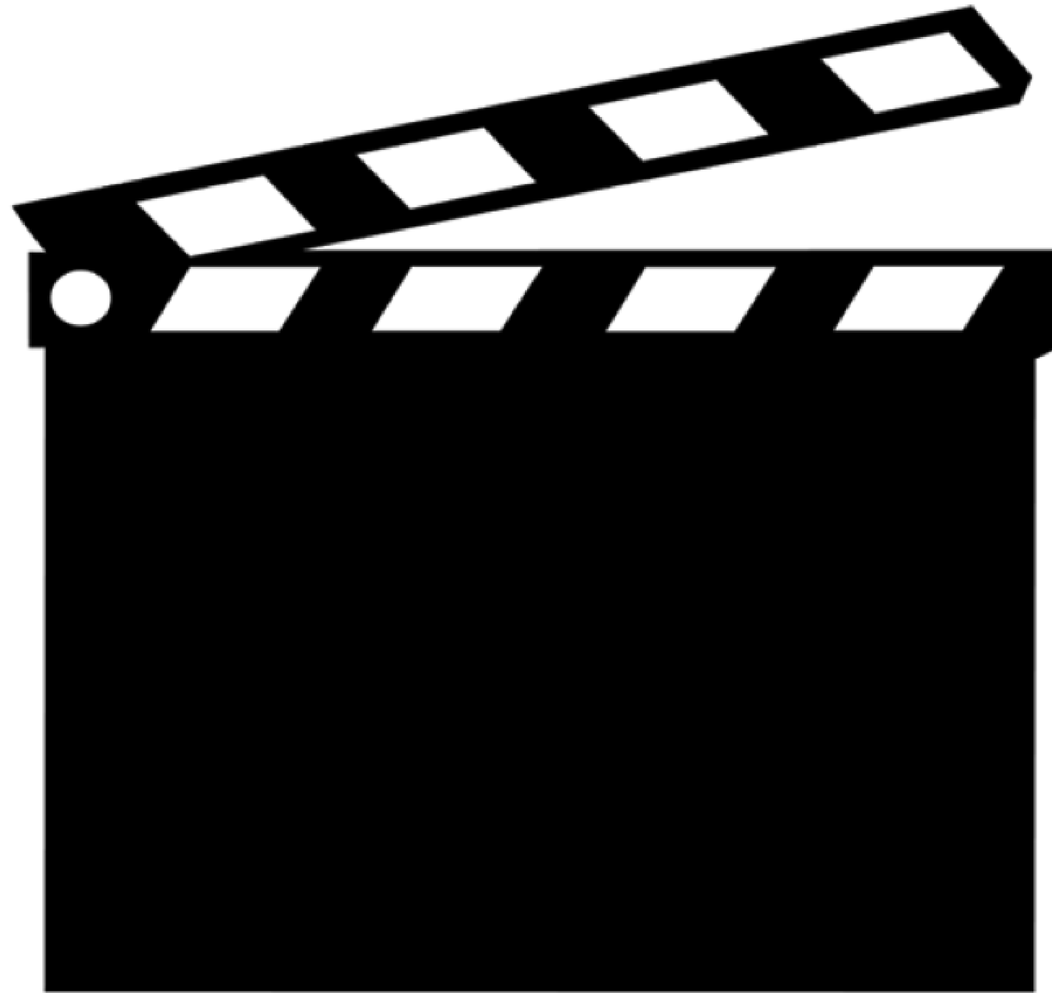
Résumé : plans pour requêtes mono-table

Permier aperçu de l'optimisation

- Le système a le choix entre plusieurs plans possibles.
- Distinguer le plus efficace n'est pas toujours trivial.
- Le choix peut changer selon le contexte.

Merci !

C018SA-W3-S5



Semaine 3 : Exécution et optimisation

1. Introduction
2. Réécriture algébrique
3. Opérateurs
4. Plans d'exécution
5. **Tri et hachage**
6. Algorithmes de jointure
7. Optimisation

Philippe Rigaux

Inria informatiques mathématiques

Tri externe

Le **tri externe** est utilisé,

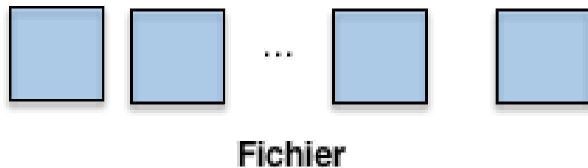
- pour les algorithmes de jointure (*sort/merge*)
- l'élimination des doublons (clause `distinct`)
- pour les opérations de regroupement (`group by`)
- ... et bien sûr pour les `order by`

Opération coûteuse sur de grands jeux de données.

Algorithme de **tri-fusion** (externe).

Première phase : le tri

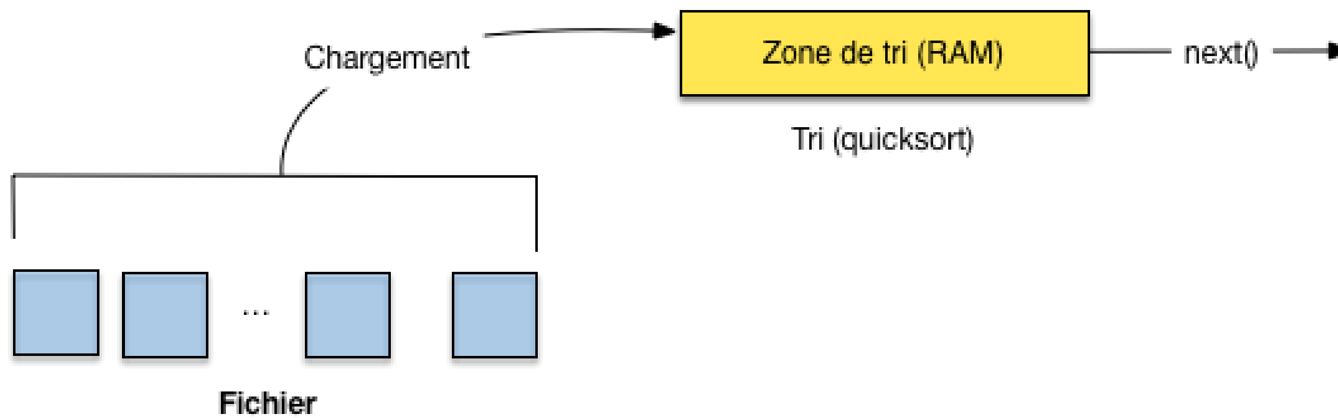
Il faut une zone de tri, en RAM, allouée par le système.



On veut trier une source de données (ici, un fichier).

Première phase : le tri

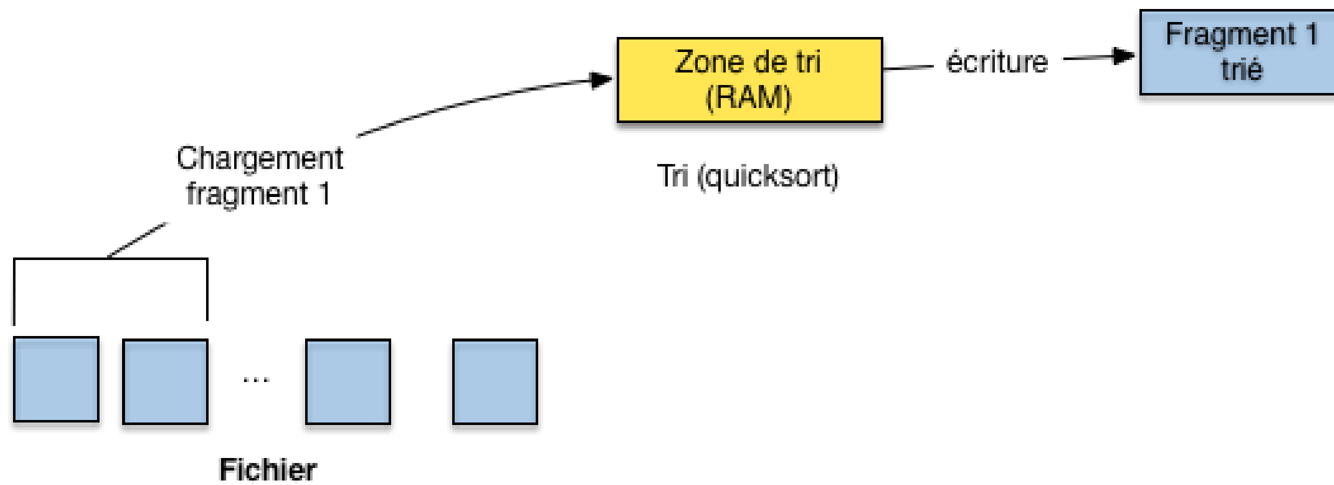
Cas favorable : tout le fichier tient en mémoire.



On charge, on trie avec *quicksort* : prêt pour l'itération.

Première phase : le tri

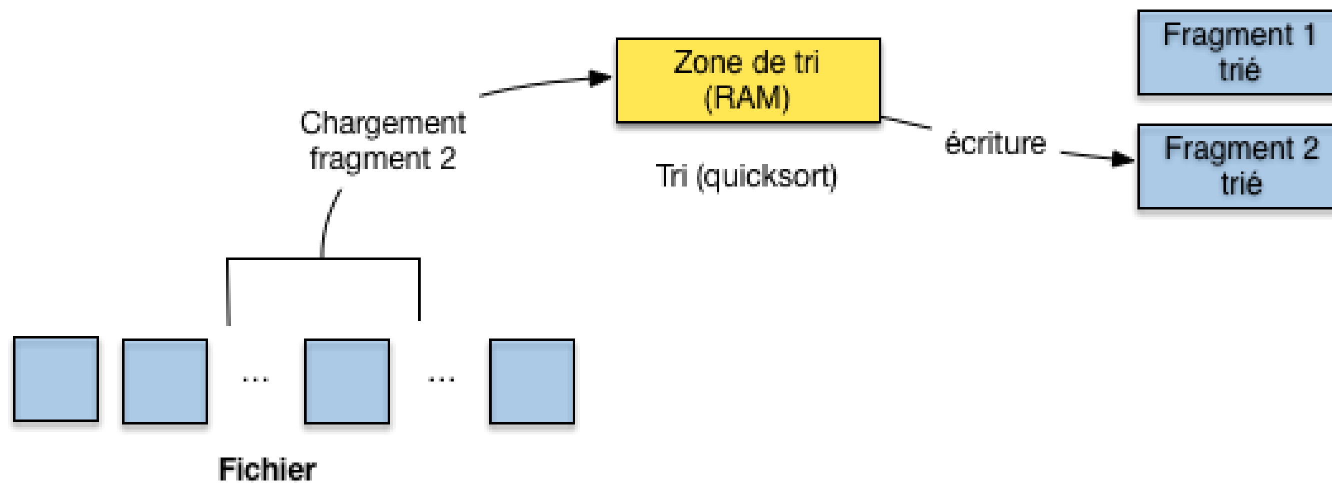
Et si le fichier ne tient pas en mémoire ?



On lit le premier fragment, on trie, on stocke.

Première phase : le tri

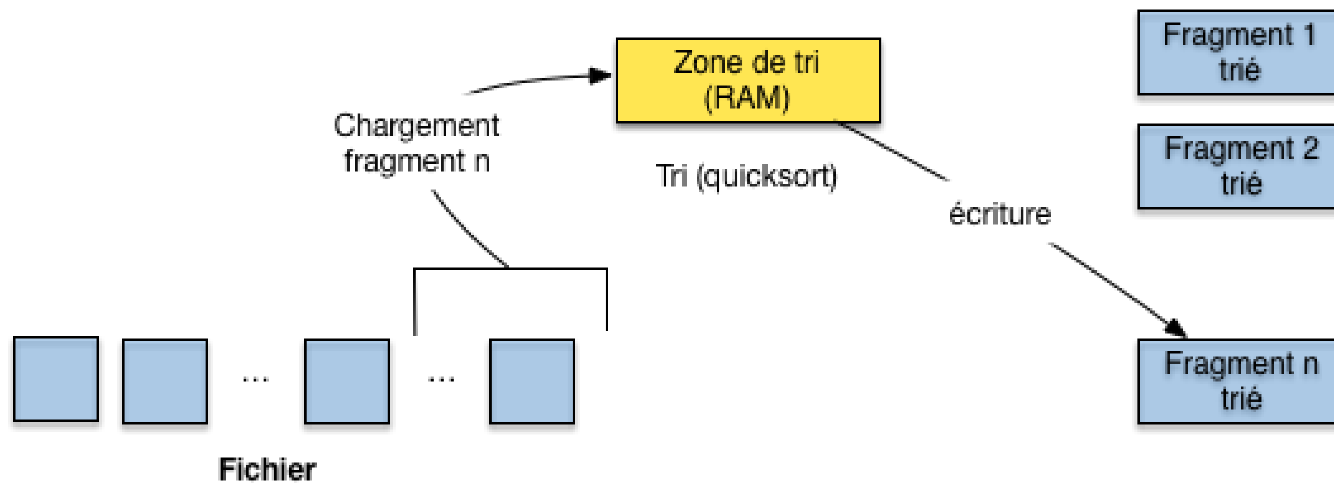
On lit le fragment suivant, on trie, on stocke.



Chaque fragment est **trié**.

Première phase : le tri

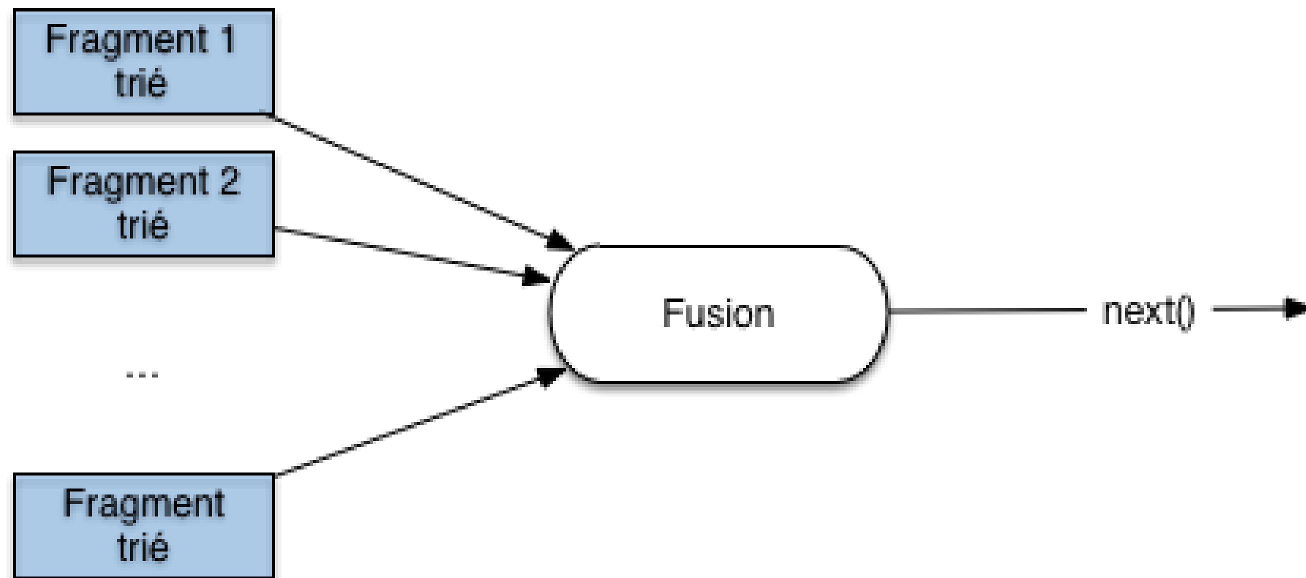
Production du dernier fragment trié



Prêt pour la phase suivante.

Fusion de listes triées

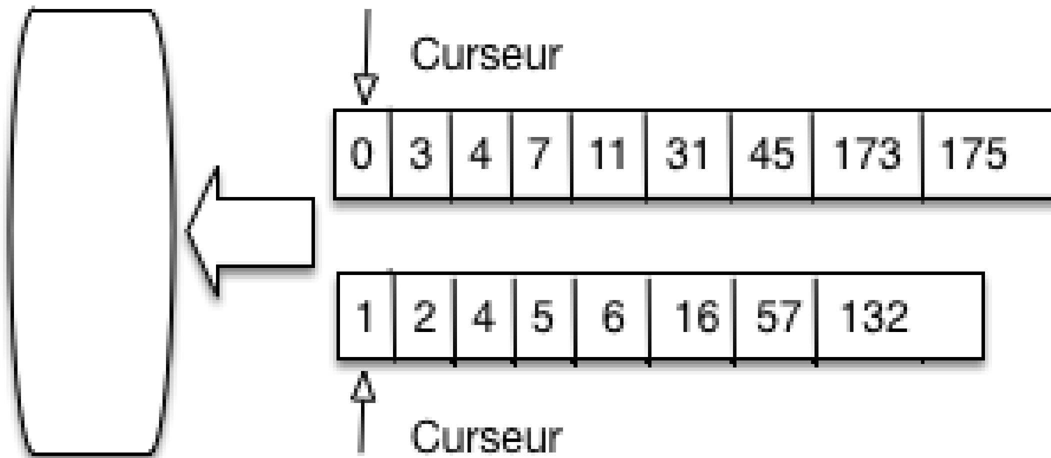
À l'issue de la phase de tri : n fragments triés.



Il faut maintenant les **fusionner**.

Fusion de listes triées

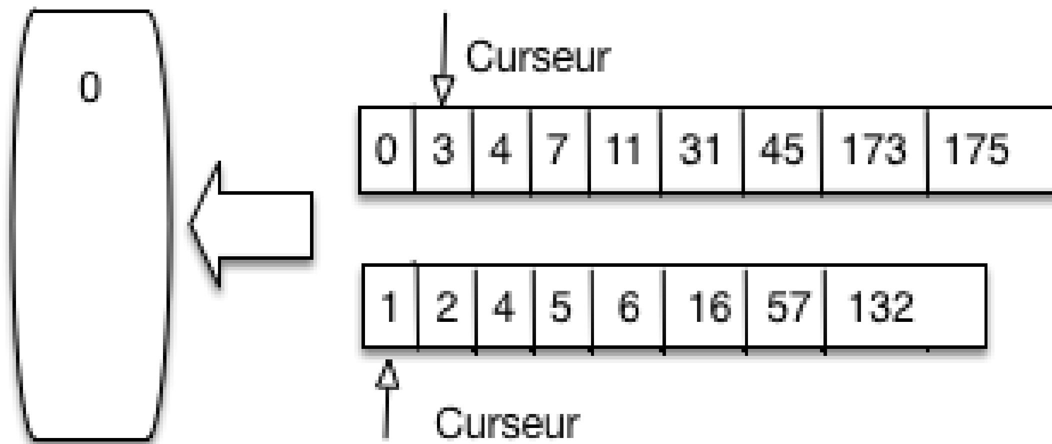
On place un curseur au début de chaque liste.



On compare les valeurs, et on prend la plus petite.

Fusion de listes triées

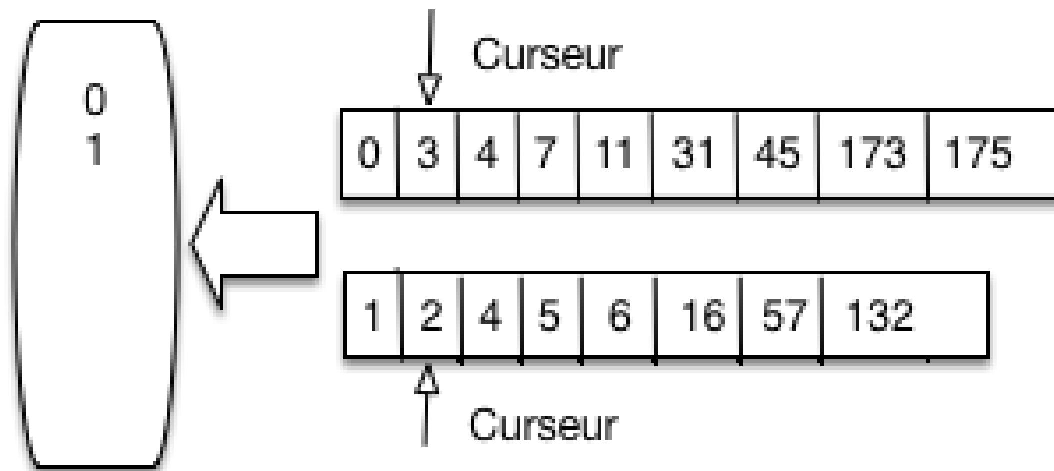
On a avancé sur le curseur de la valeur extraite.



On applique la même méthode.

Fusion de listes triées

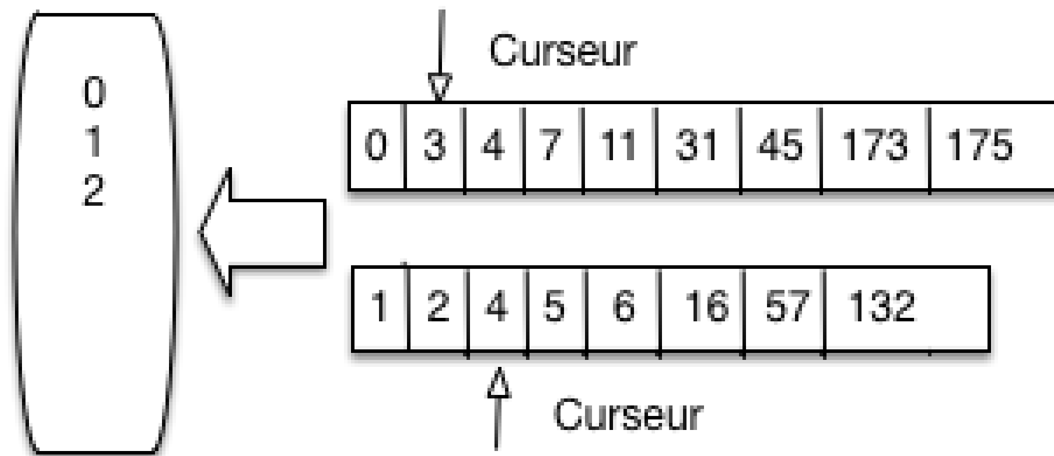
On continue sur le même principe.



On ne revient jamais en arrière.

Fusion de listes triées

Une dernière fois...



Le résultat est la liste triée globale.

Résumé : tri et opérateurs bloquants

L'opérateur de tri est **bloquant**

- La phase de tri est effectuée pendant le *open()*
- Le *next()* correspond à la progression de l'étape de fusion.

Conséquence : **latence importante des requêtes impliquant un tri.**

- Il faut lire **au moins une fois** toute la table.
- Si mémoire insuffisante, il faut lire deux fois, écrire une fois.
- Parfois pire...

Résumé : tri et opérateurs bloquants

L'opérateur de tri est **bloquant**

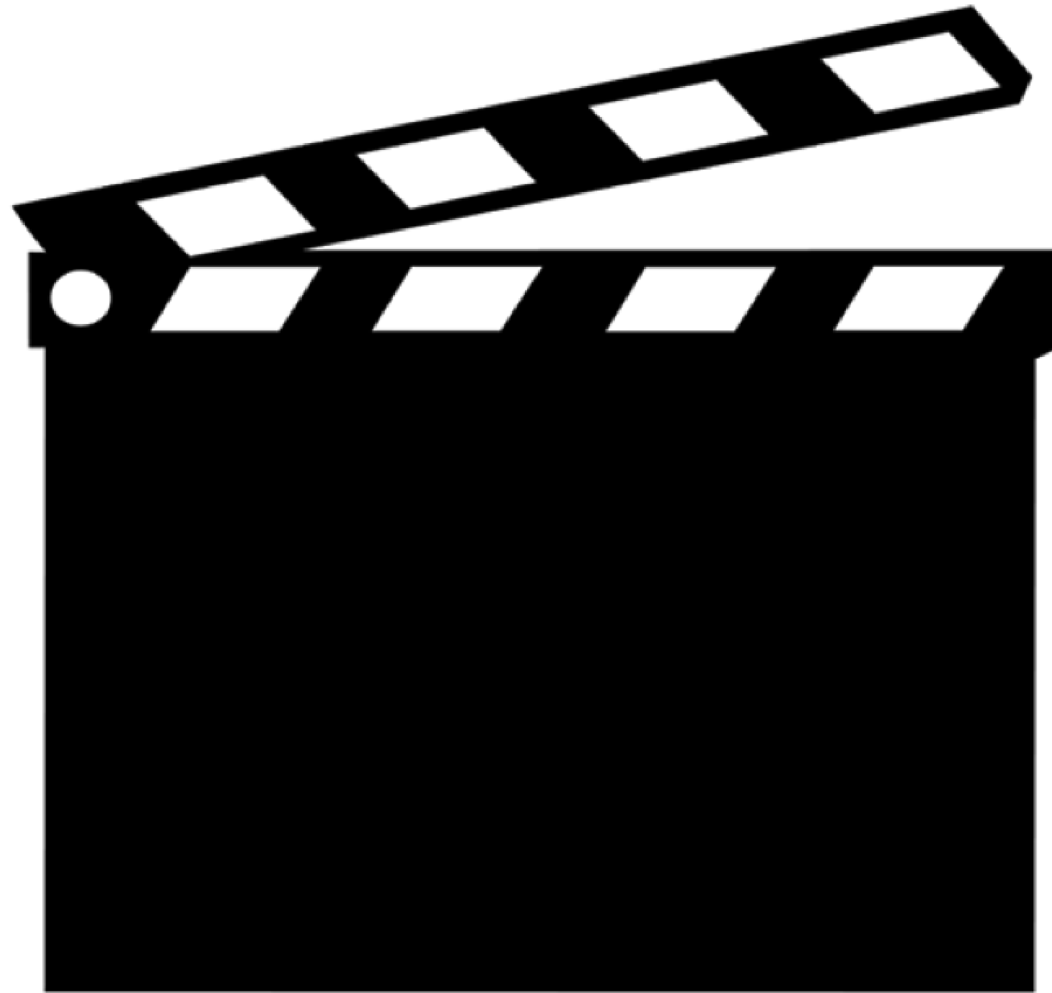
- La phase de tri est effectuée pendant le *open()*
- Le *next()* correspond à la progression de l'étape de fusion.

Conséquence : **latence importante des requêtes impliquant un tri.**

- Il faut lire **au moins une fois** toute la table.
- Si mémoire insuffisante, il faut lire deux fois, écrire une fois.
- Parfois pire...

Merci !

C018SA-W3-S6



Semaine 3 : Exécution et optimisation

1. Introduction
2. Réécriture algébrique
3. Opérateurs
4. Plans d'exécution
5. Tri et hachage
6. **Algorithmes de jointure**
7. Optimisation

Philippe Rigaux

Inria informatiques mathématiques

L'opérateur qui nous manque

La jointure : opération très courante, potentiellement coûteuse.

L'opérateur de jointure complète notre petit catalogue pour (presque) toutes les requêtes SQL.

```
select a1, a2, .., an
from T1, T2, ..., Tm
where T1.x = T2.y and ...
order by ...
```

Plusieurs algorithmes possibles.

Principaux algorithmes

On ve se limiter à quelques exemples représentatifs :

Jointure avec index

- Algorithme de jointure par boucles imbriquées indexées.

Jointure sans index

- Le plus simple : **boucles imbriquées (non indexée)**.
- Plus sophistiqué : la **jointure par hachage**.

Jointure avec index

Très **courant** ; on effectue **naturellement** la jointure sur les clés primaires/étrangères.

- Les films et leur metteur en scène

```
select * from Film as f, Artiste as a
where f.id_realisateur = a.id
```

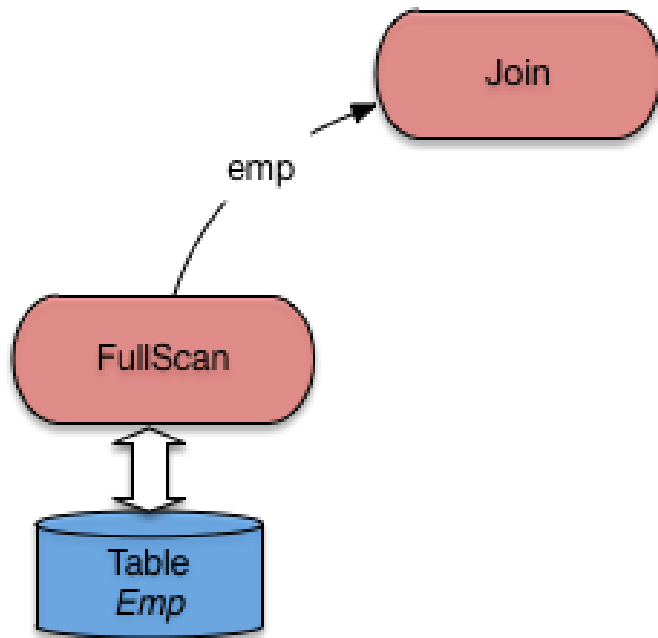
- Les employés et leur département

```
select * from emp e, dept d
where e.dnum = d.num
```

Garantit **au moins** un index sur la condition de jointure.

Jointure avec index : l'algorithme

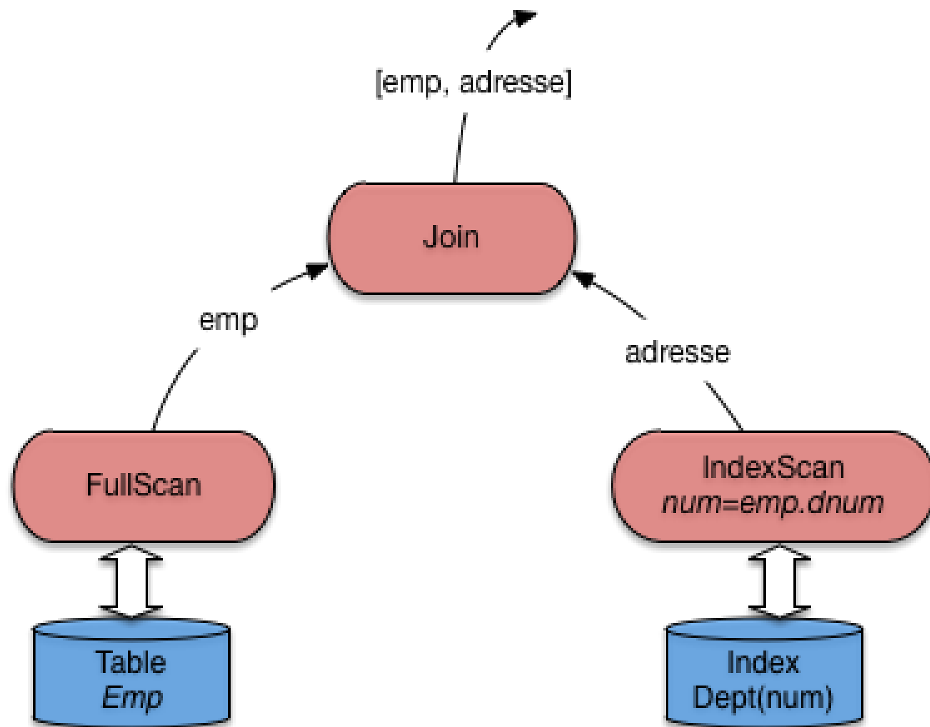
On parcourt séquentiellement la table contenant la clé étrangère.



On obtient des nuplets `employé`, avec leur no de département.

Jointure avec index : l'algorithme

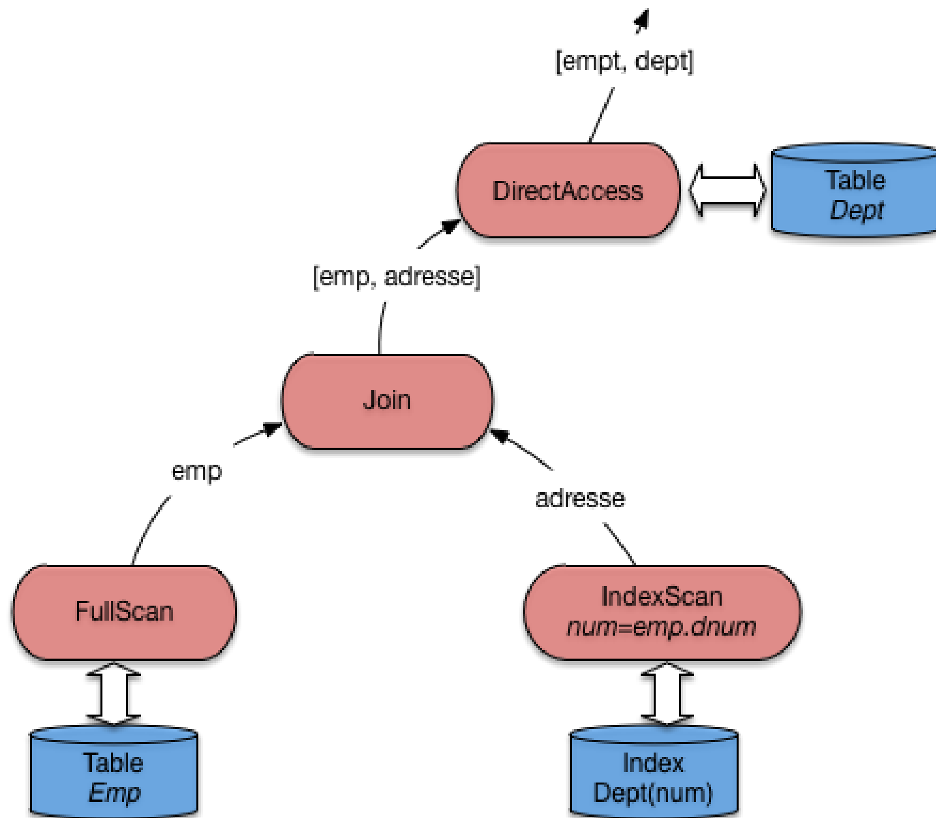
On utilise le no de département pour un accès à l'index `Dept`.



On obtient des paires `[employé, adrDept]`.

Jointure avec index : l'algorithme

Il reste à résoudre l'adresse du département avec un accès direct.



On obtient des paires `[employé, dept]`.

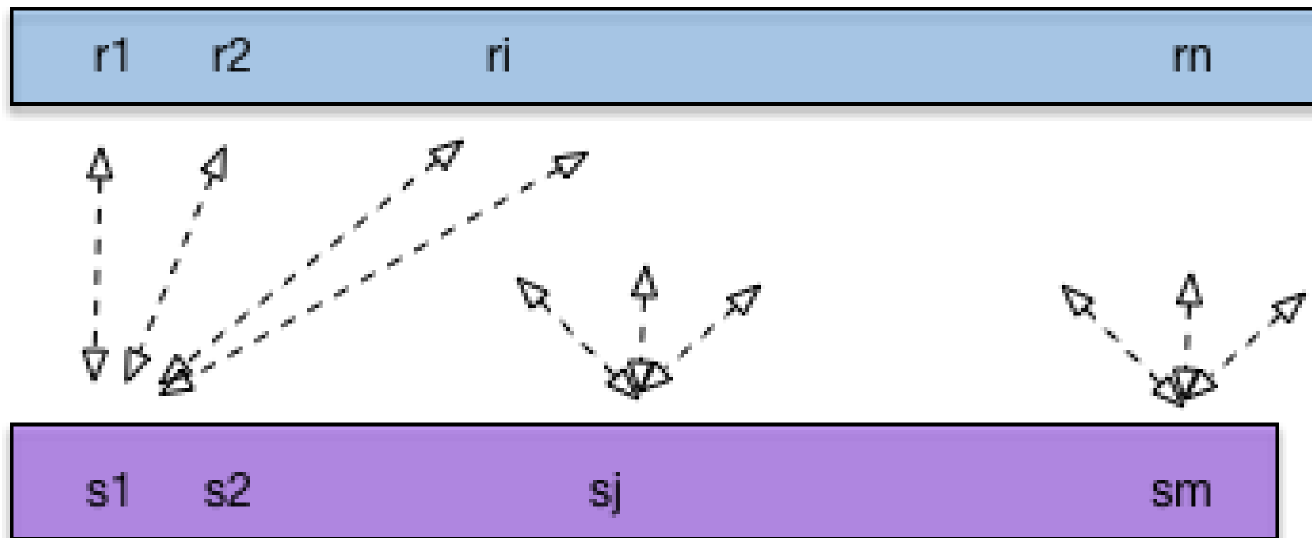
Jointure avec index : l'algorithme

Avantages :

- Efficace (un parcours, plus des recherches par adresse)
- Favorise le temps de réponse **et** le temps d'exécution

Jointures par boucles imbriquées

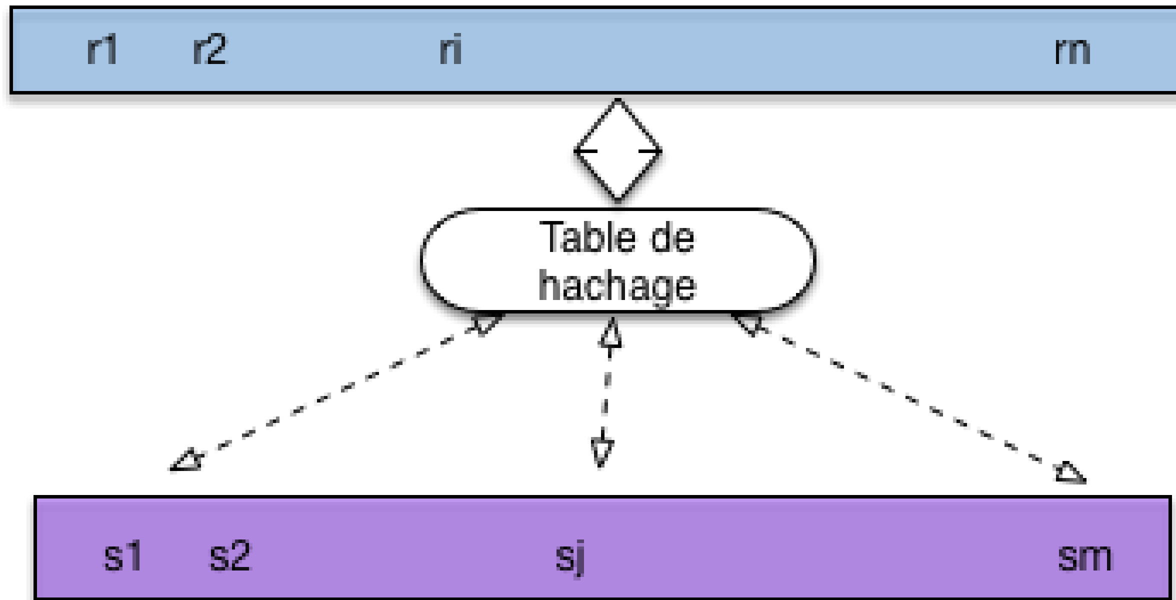
Pas d'index ? La méthode de base est d'énumérer **toutes** les solutions possibles.



Coût quadratique. Acceptable pour deux petites tables.

Jointures par hachage

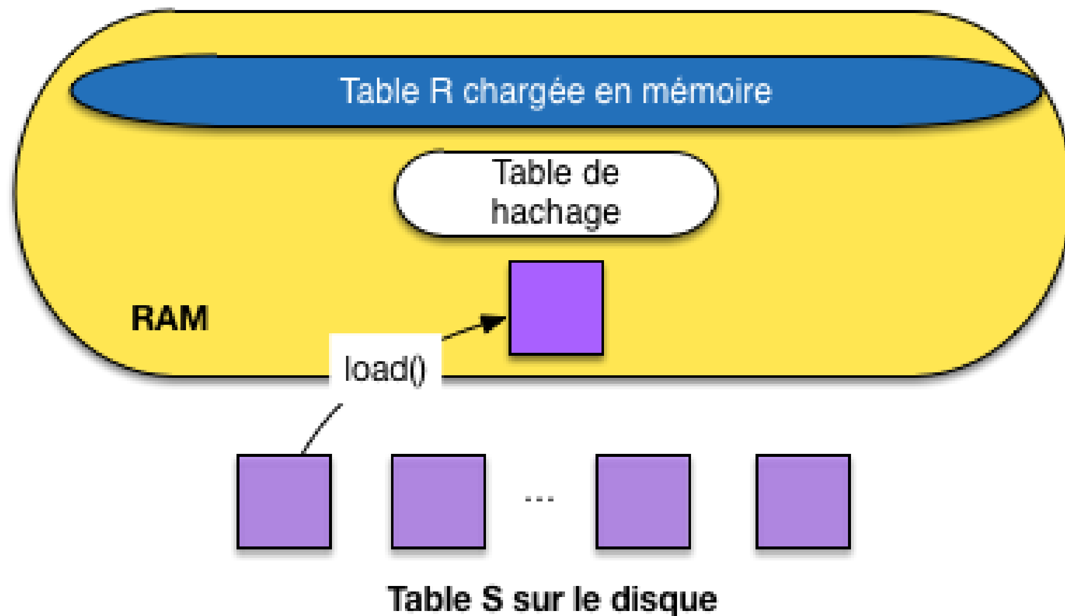
Meilleure solution : construire une table de hachage sur une des tables.



Evite les $O(n^2)$ comparaisons. Appelons cette méthode **JoinList**.

Mémoire insuffisante ?

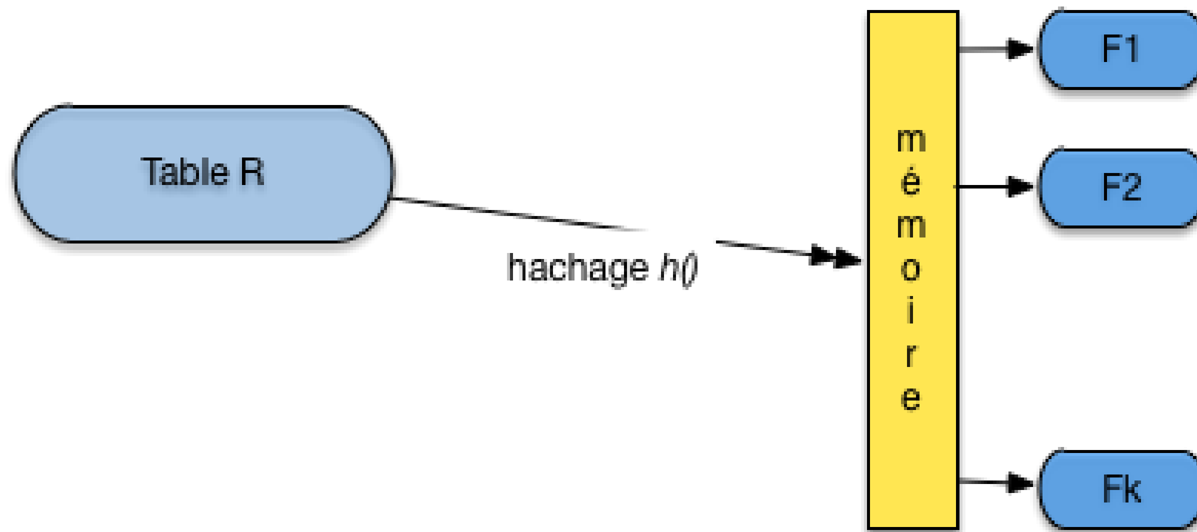
Essayons de placer **une** des deux tables en mémoire.



On charge l'autre bloc par bloc ; on applique **JoinList**.
Une seule lecture de chaque table suffit.

Et quand *aucune* table ne tient en mémoire ?

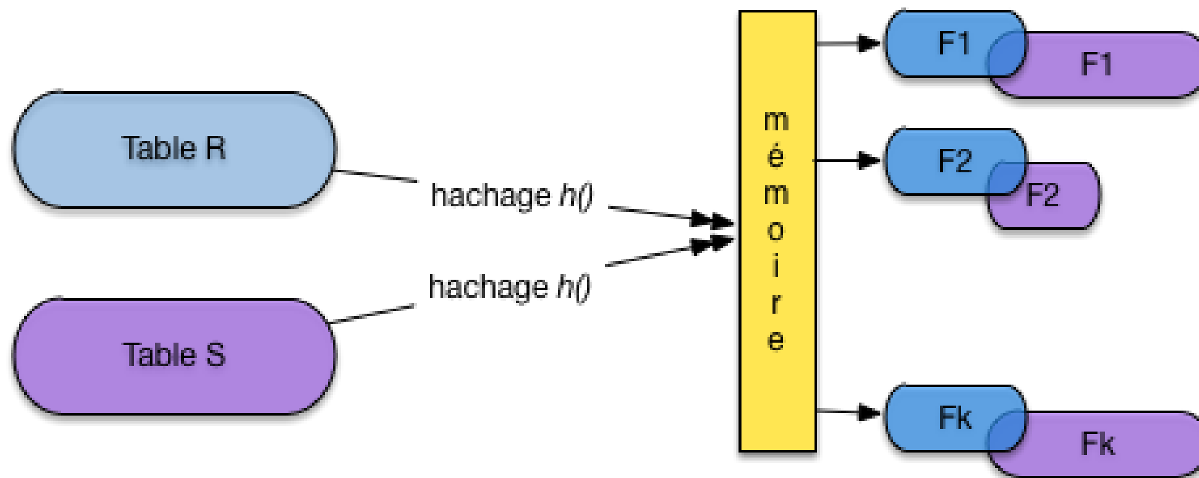
On hache la plus petite des deux tables en k fragments.



Essentiel : les fragments doivent tenir, chacun, en mémoire.

Et quand *aucune* table ne tient en mémoire ?

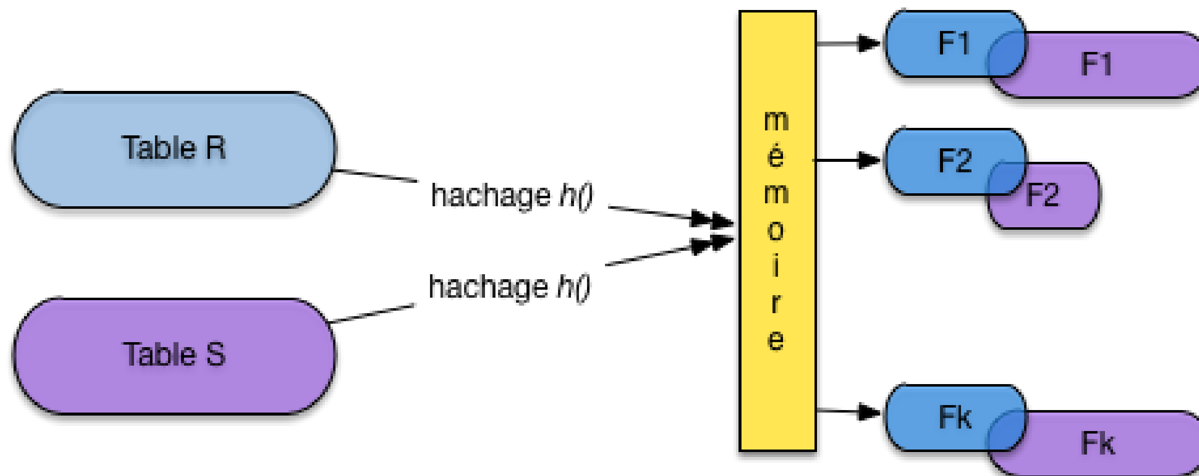
On hache la seconde table, avec la même fonction $h()$, en k autres fragments.



Cette fois, on n'impose pas la contrainte que les fragments tiennent en mémoire.

Et quand *aucune* table ne tient en mémoire ?

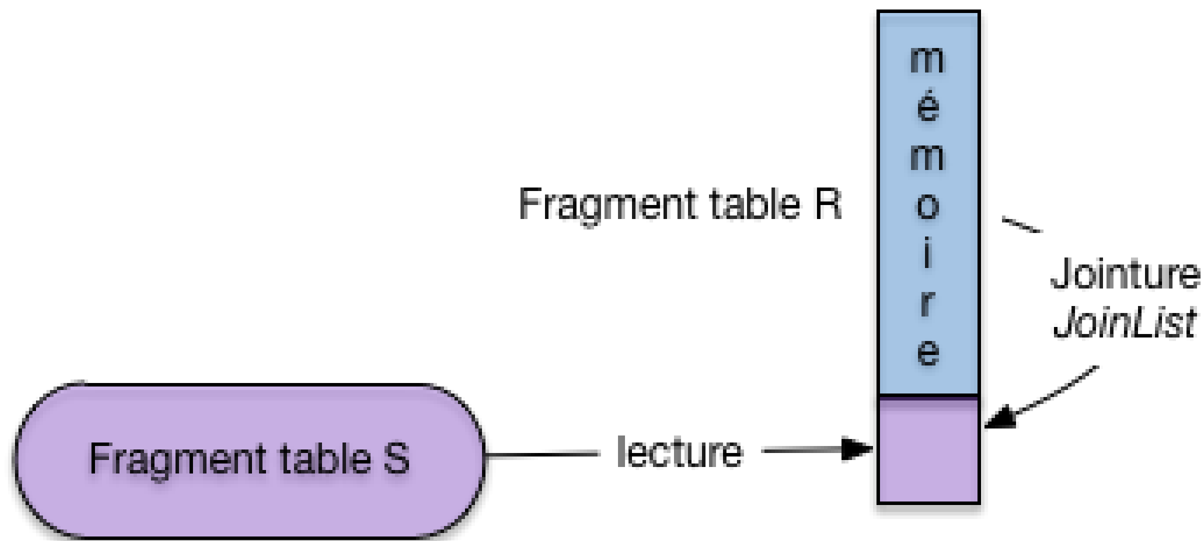
On effectue la jointure sur les paires de fragments $(F_1^R, F_1^S), (F_2^R, F_2^S), (F_k^R, F_k^S)$,



Propriété : Deux nuplets *r* et *s* doivent être joints si et seulement si ils sont dans des fragments associés.

Illustration : phase de jointure

On charge F_R^i de R en mémoire ; on parcourt F_S^i de S et on joint.



Déjà vu ? Oui : jointure par boucles imbriquées quand une table tient en mémoire.

Résumé : la jointure

Un opérateur potentiellement coûteux. Quelques principes généraux :

- Si **une** table tient en mémoire : jointure par boucle imbriquées, ou hachage.
- Si **au moins un** index est utilisable : jointure par boucle imbriquées indexée.
- Si une des deux tables beaucoup plus petite que l'autre : jointure par hachage.
- Sinon : jointure par tri-fusion (non présenté).

Décision très complexe, prise par la système en fonction des statistiques.

Résumé : la jointure

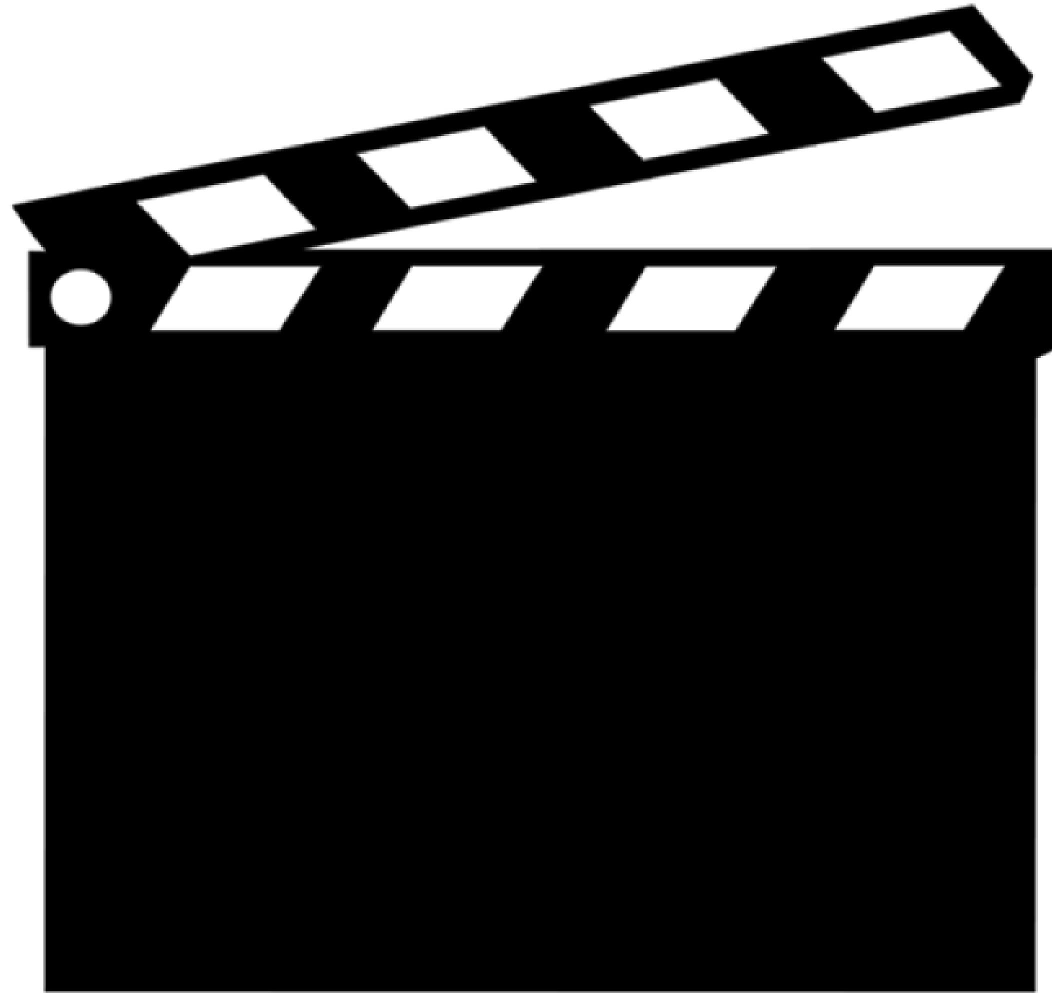
Un opérateur potentiellement coûteux. Quelques principes généraux :

- Si **une** table tient en mémoire : jointure par boucle imbriquées, ou hachage.
- Si **au moins un** index est utilisable : jointure par boucle imbriquées indexée.
- Si une des deux tables beaucoup plus petite que l'autre : jointure par hachage.
- Sinon : jointure par tri-fusion (non présenté).

Décision très complexe, prise par la système en fonction
des statistiques.

Merci !

C018SA-W3-S7



Semaine 3 : Exécution et optimisation

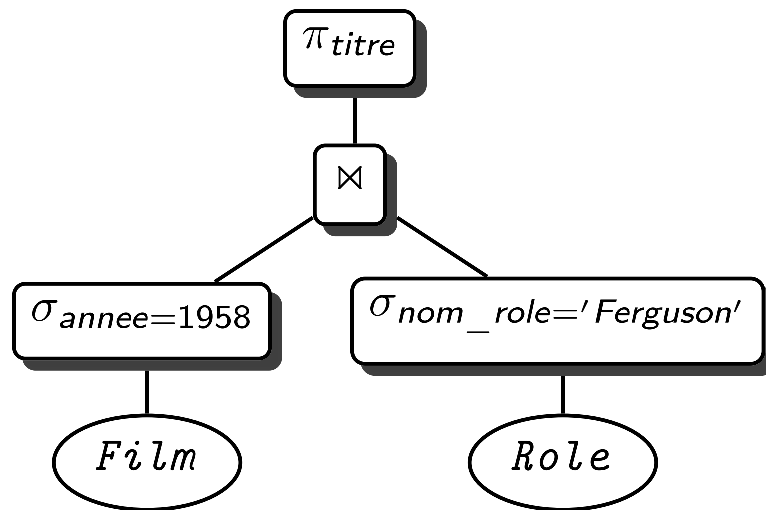
1. Introduction
2. Réécriture algébrique
3. Opérateurs
4. Plans d'exécution
5. Tri et hachage
6. Algorithmes de jointure
7. **Optimisation**

Philippe Rigaux

Inria informatiques mathématiques

Retour en arrière

Notre requête sur les films avec John Ferguson, après optimisation algébrique.



Il nous restait à choisir les chemins d'accès et les algorithmes de jointure.

Continuons à optimiser notre requête

Hyp. : le système cherche à utiliser les index pour les jointures .

$$Film \bowtie_{id=id_film} Role$$

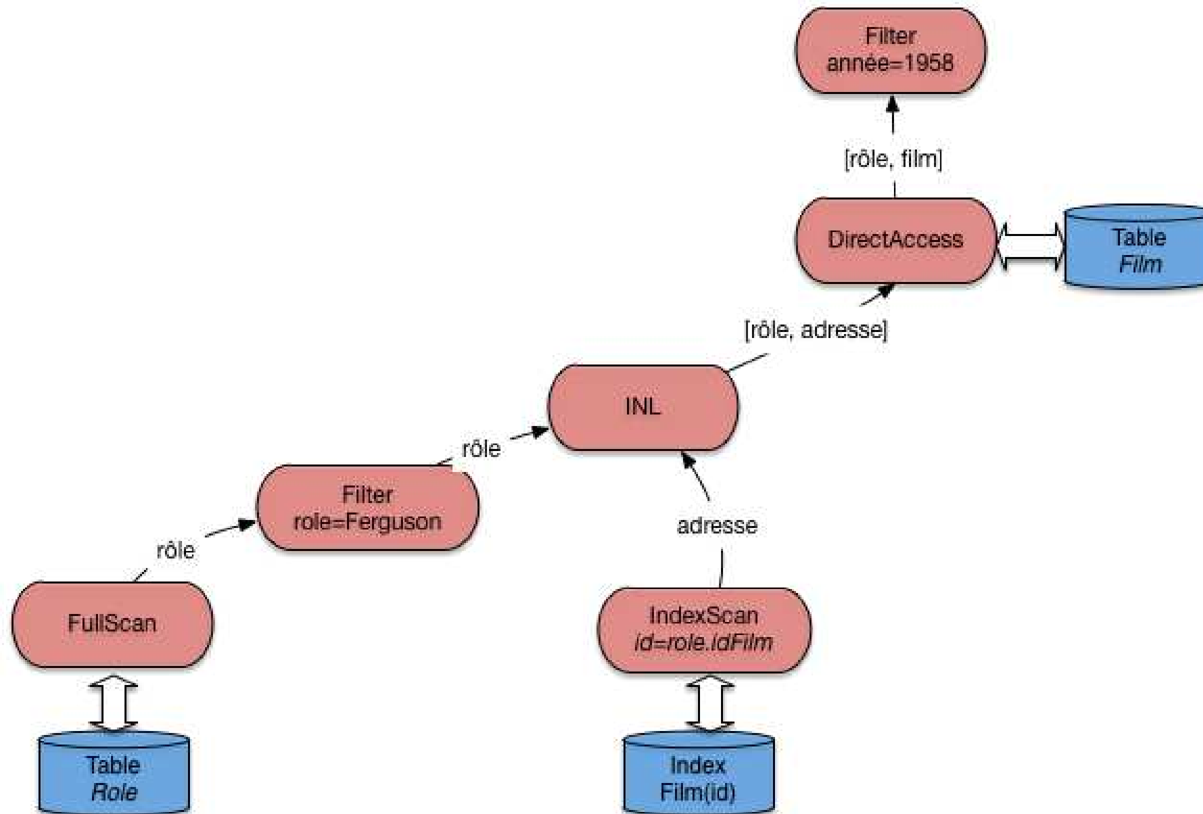
ou

$$Role \bowtie_{id_film=id} Film$$

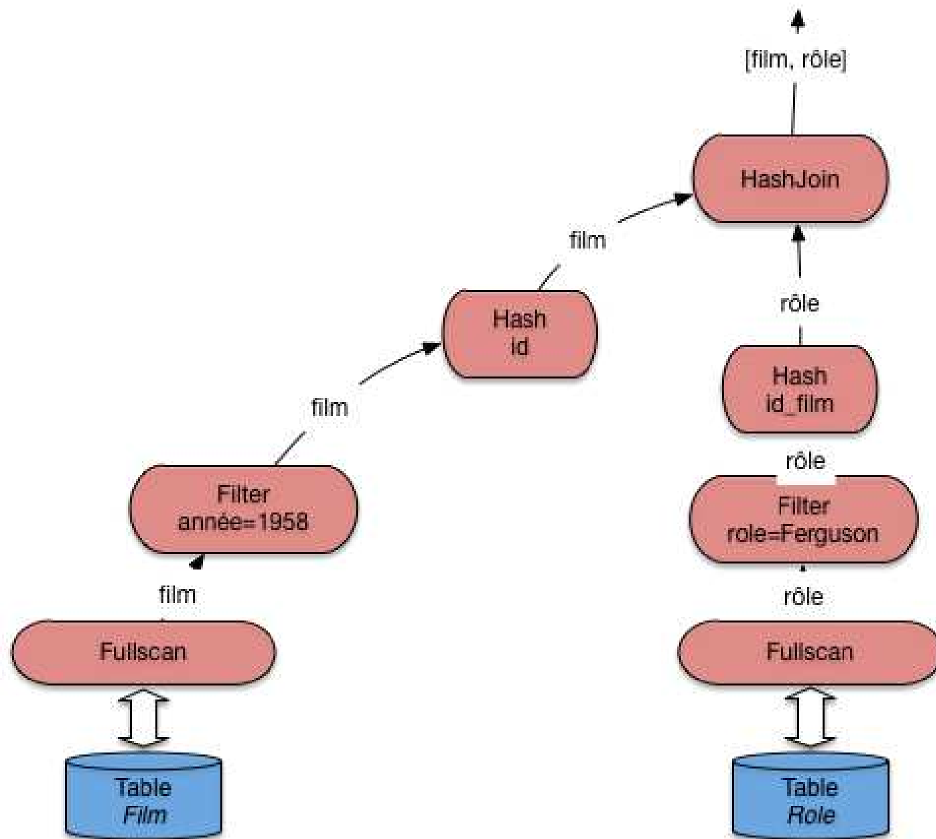
Après analyse des index, la bonne forme est

$$\pi_{titre}(\sigma_{nom_role='John\ Ferguson'}(Role) \bowtie_{id_film=id} \sigma_{annee=1958}(Film))$$

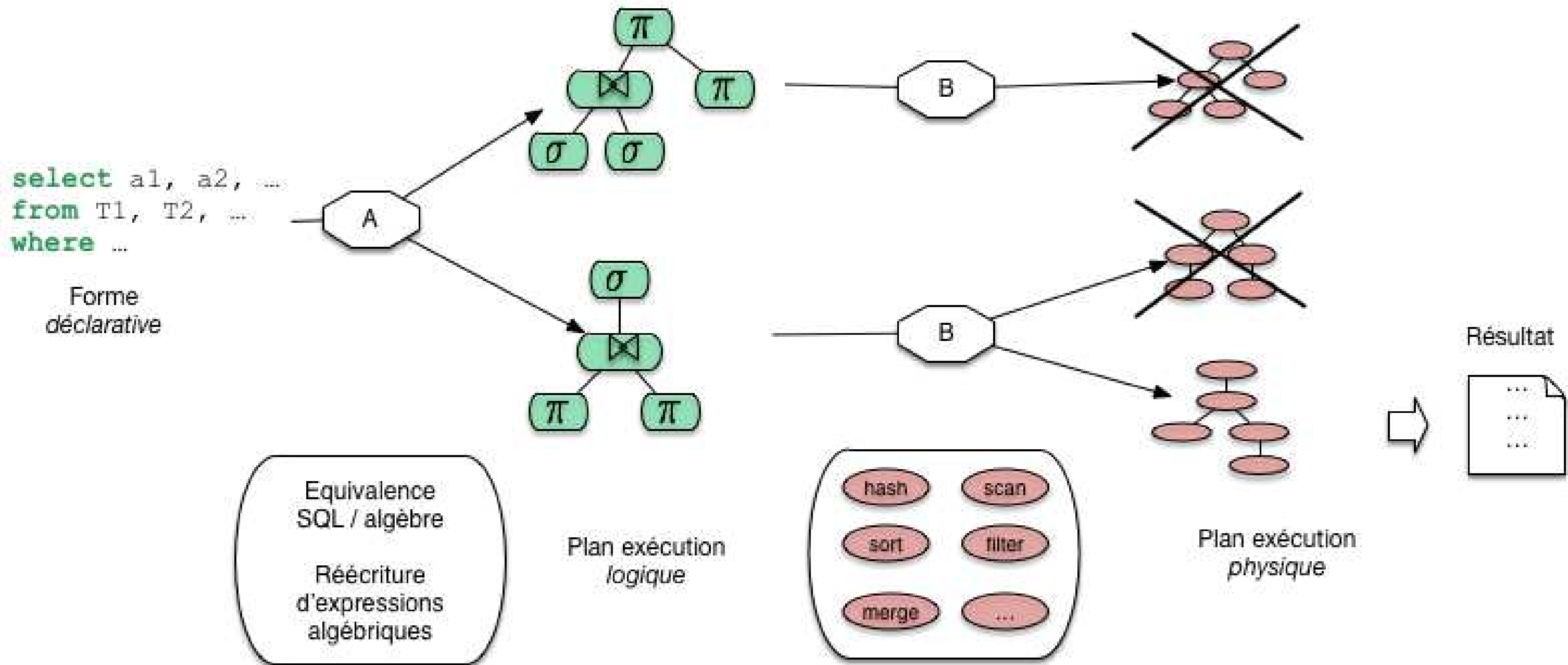
Le plan d'exécution



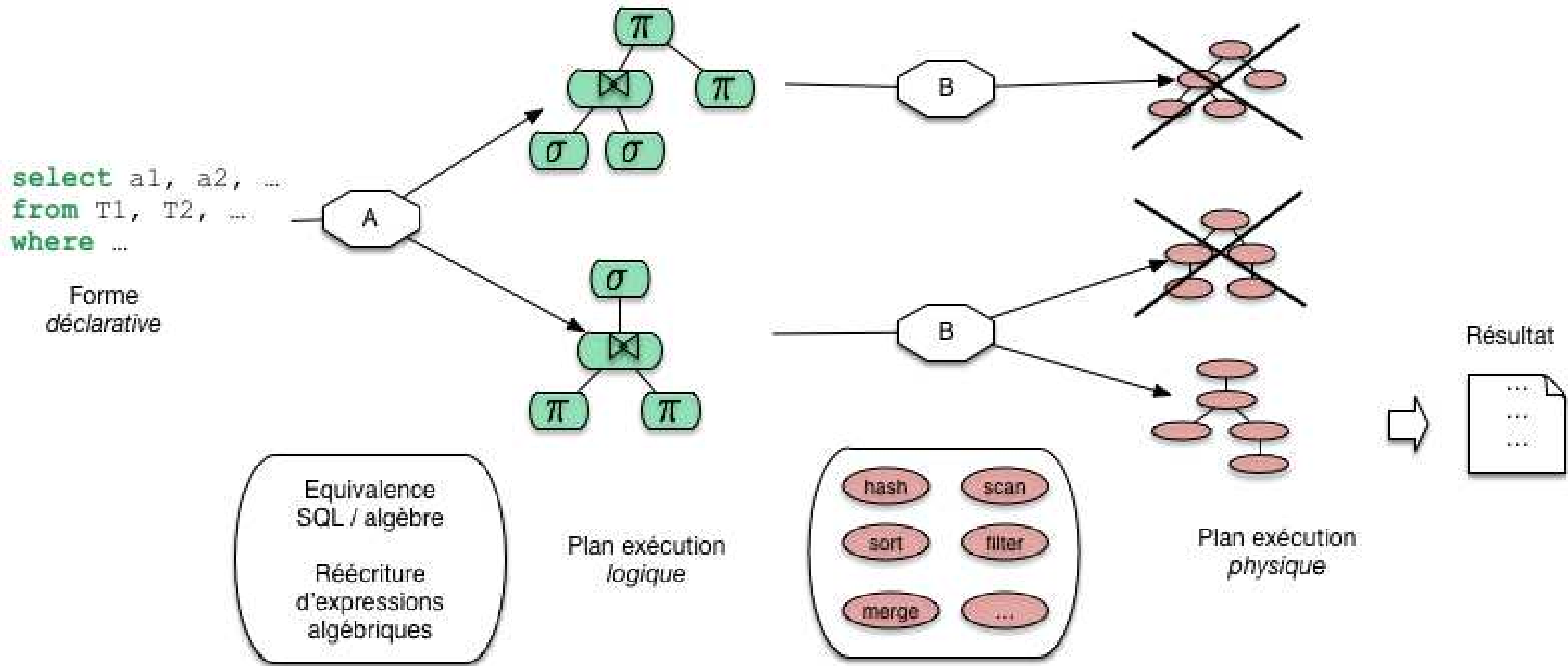
Et si je n'ai pas d'index ?



Résumé : la phase d'optimisation



Résumé : la phase d'optimisation



Merci !