Entrepôts de Données et Big Data - HAI708I

Optimisation de Requête - Plan d'exécution Physique

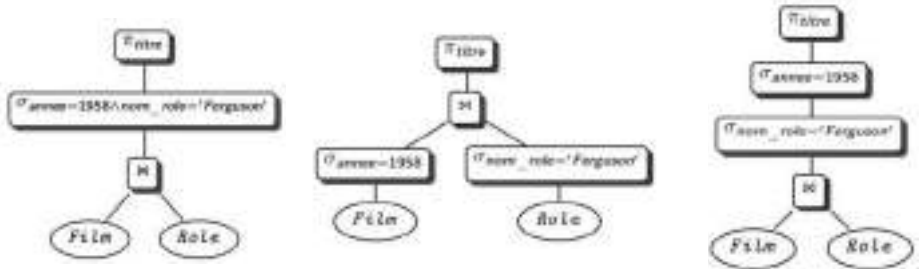
Référence : Serge Abiteboul

Le rôle de l'optimiseur

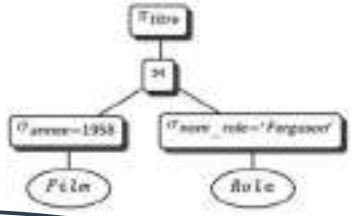
Trouver les expressions équivalentes

Requête SQL ➡ **Plan d'exécution logique - PEL (l'algèbre)**

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

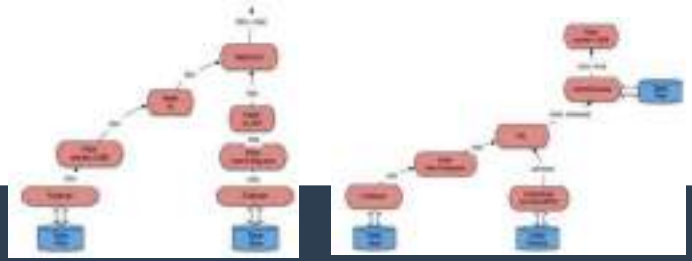


Évaluation des coûts pour trouver le meilleur PEL

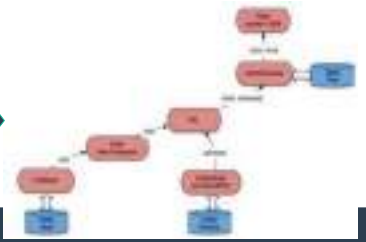


Choisir le bon algorithme pour chaque opération

Plan d'exécution physique – PEP (opérateurs)



Évaluation des coûts pour trouver le meilleur PEP ➡



Le rôle de l’optimiseur : les opérateurs (PEP)

- **Les opérations de l’algèbre relationnelle peuvent être évaluée à l’aide de plusieurs algorithmes (opérateurs)**
  - une même expression d’algèbre relationnelle peut être évaluée de plusieurs façons différentes

- **Une expression annotée indiquant les méthodes utilisées est un plan d’exécution physique**

  Par exemple :

  - s’il existe, il est possible d’utiliser un index pour obtenir tous les employés dont le salaire est supérieur à 30000€

  ou

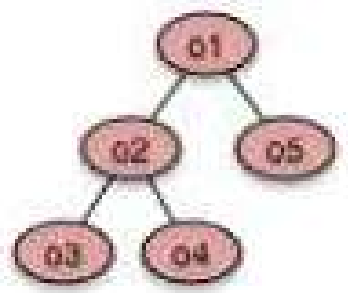
  - toute la table peut être lue et seuls les employés dont le salaire est supérieur à 30000€ sont conservés

Le rôle de l’optimiseur : les opérateurs (PEP)

- **Tout opérateur est implanté sous forme d’un itérateur**

- **Trois fonctions :**
  - open : initialise les ressources et positionne le curseur
  - next : ramène l’enregistrement courant et se place sur l’enregistrement suivant
  - close : libère les ressources

- **Un plan d'exécution est un arbre d'itérateurs**
  - un itérateur consomme des n-uplets d’autres itérateurs source ou de données
  - un itérateur produit des n-uplets à la demande



Plan exécution *physique*

Le rôle de l’optimiseur : les itérateurs (PEP)

- **Rôle des itérateurs : principes essentiels**
  - production à la demande : le serveur n'envoie un enregistrement au client que quand ce dernier le demande
  - pipeline : on essaie d'éviter le stockage en mémoire de résultats intermédiaires (résultat au fur et à mesure) mais parfois c’est nécessaire

  ➡ évite d'avoir à stocker des résultats intermédiaire

  ➡ temps de réponse minimisé mais attention aux opérateurs bloquants

  - temps de réponse : temps pour obtenir le premier n-uplet
  - temps d’exécution : temps pour obtenir tous les n-uplets.

- **Opérateur bloquant :** exemple `select min(date) from T`

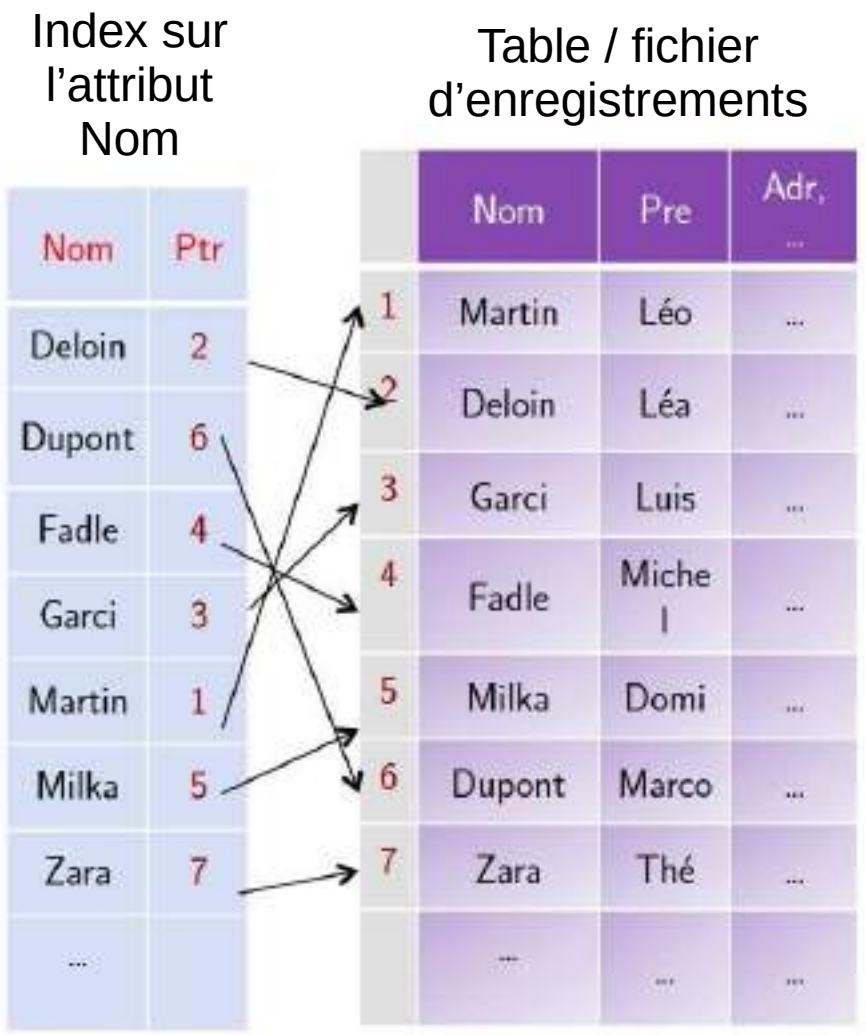
  ➡ on additionne le temps d’exécution et le temps de d’exécution

- **Utilisation d’index pour accéder aux données**

Le rôle de l'optimiseur : les index *(PEP)*

- **Un index =**
  - structure de données (fichier) construite pour accéder de façon rapide aux valeurs des lignes d'une table de la BD (aux enregistrements du fichier de données correspondant à une table)
  - MAIS impact négatif sur la mise à jour de la table (coût plus élevé)

- **Un index est composé de deux parties**
  - un fichier d'enregistrements à deux champs :
    - une clé de recherche (= valeur d'un attribut ou d'une liste d'attribut de la table)
    - un pointeur ou une liste de pointeurs vers des enregistrements du fichier de données
  - un mécanisme d'accès à un enregistrement de l'index à partir de la clé de recherche.



Le rôle de l'optimiseur : les index *(PEP)*

- **Création implicite d'index :**
  - Sur un attribut clé primaire
  - (Souvent) Sur un attribut pour lequel une contrainte d'unicité est définie

➡ Index dense et unique = toutes les clés de recherche sont présentes dans l'index et une seule fois

- **Organisation d'un index**
  - Séquentiel (index dense)
  - Séquentiel indexé (index creux)
  - Arbres B+ (par défaut dans ORACLE)
  - Bitmap (possible dans ORACLE)
  - Hashage dynamique / statique

- **Création explicite et suppression d'un index dans ORACLE**

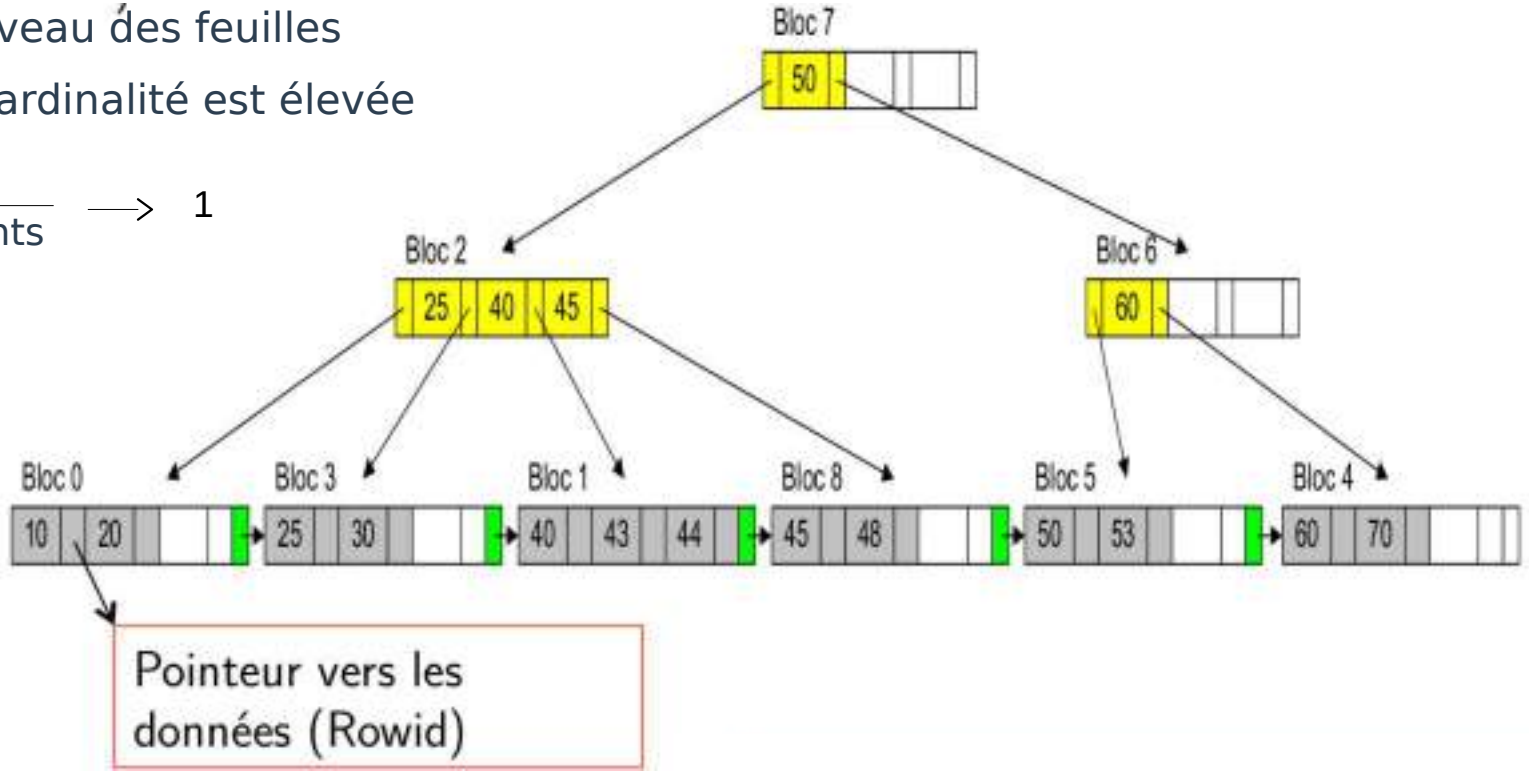
  CREATE [UNIQUE | BITMAP] INDEX nom_index ON nom_table (colonne1 [ASC|DESC], colonne2 [ASC|DESC]], …) ;

  DROP INDEX nom_index ;

Le rôle de l'optimiseur : les index – Arbre B+ *(PEP)*

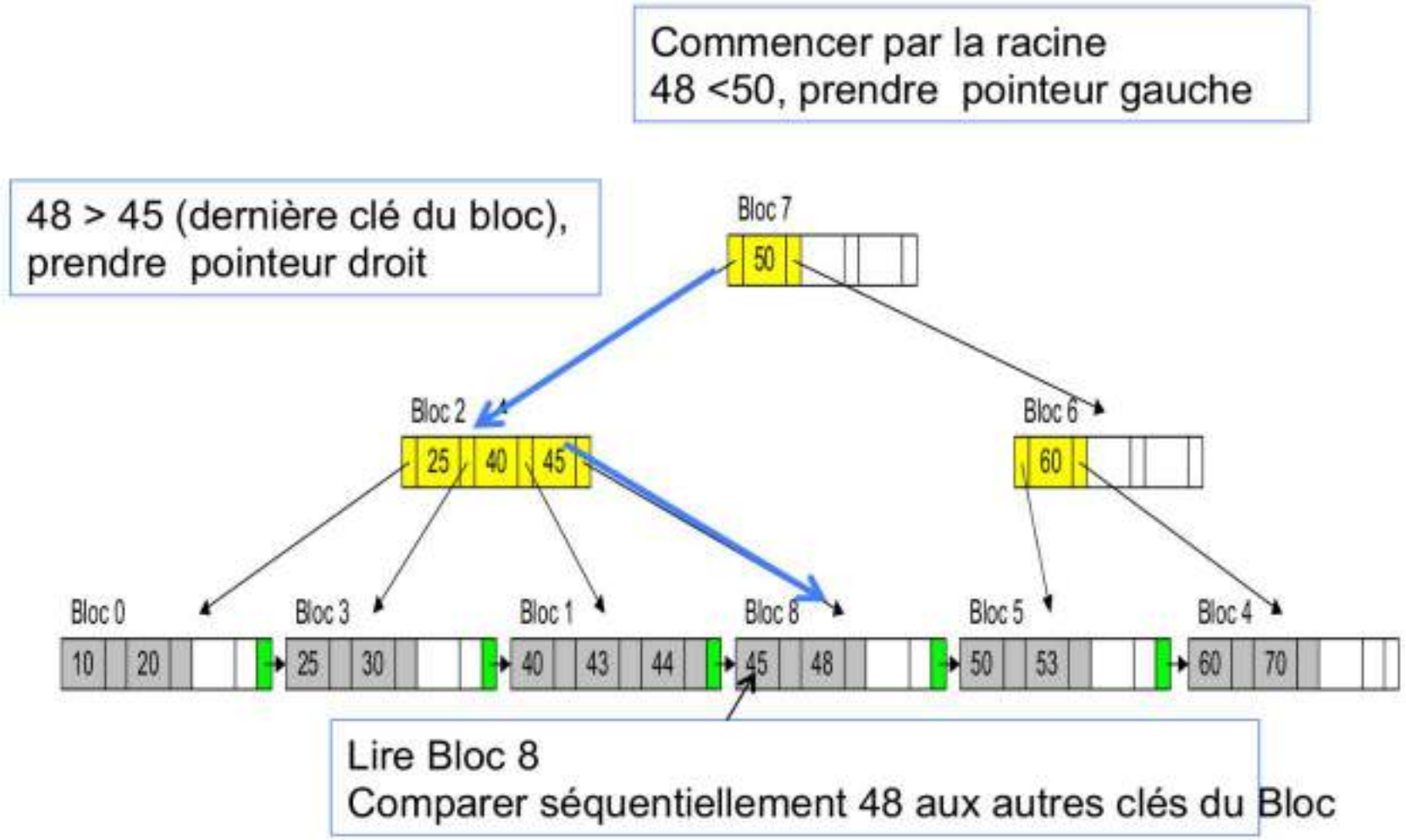
- **Index Arbre B+ (par défaut dans ORACLE)**
  - chaque nœud est un bloc contenant k clés triées et k+1 fils
  - arbre équilibré
  - feuilles = valeurs des clés et Rowid
  - clés répétées au niveau des feuilles
  - efficace quand la cardinalité est élevée

$$\frac{\text{nb valeurs}}{\text{nb enregistrements}} \longrightarrow 1$$



Le rôle de l'optimiseur : les index – Arbre B+ *(PEP)*

- **Index Arbres B+ (par défaut dans ORACLE)**
  - Exemple : recherche de la clé 48



Le rôle de l'optimiseur : les index – Bitmap *(PEP)*

- **Index Bitmap**
  - Considération de toutes les valeurs possibles pour un attribut.
  - Pour chaque valeur, stockage d'un tableau de bits (dit bitmap) avec autant de bits qu'il y a de lignes dans la table.
  - Efficace quand la cardinalité est faible

$$\frac{\text{nb valeurs}}{\text{nb enregistrements}} \longrightarrow 0$$

Table

	Titre	Genre
1	Vertigo	Suspense
2	Brazil	Science-fiction
3	Twin Peaks	Fantastique
4	Underground	Drame
5	Easy Rider	Drame
6	Psychose	Drame
7	Greystoke	Aventures
8	Shining	Fantastique
9	Annie Hall	Comédie
10	Jurassic Park	Science-fiction
11	Metropolis	Science-fiction
12	Manhattan	Comédie
13	Reservoir Dogs	Policier
14	Impitoyable	Western
15	Casablanca	Drame
16	Smoke	Comédie

Index Bitmap sur l'attribut Genre

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Drame	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0
Science-fiction	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0
Comédie	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1

Le rôle de l'optimiseur : les index – Bitmap *(PEP)*

- **Index Bitmap**
  - Exemple : sélectionner dans la table les numéros de ligne (RowId) ayant Drame="1"

Index Bitmap sur l'attribut Genre

RowId	Drame	Science-fiction	Comédie
1	0	0	0
2	0	1	0
3	0	0	0
4	**1**	**0**	**0**
5	**1**	**0**	**0**
6	**1**	**0**	**0**
7	0	0	0
8	0	0	0
9	0	0	1
10	0	1	0
11	0	1	0
12	0	0	1
13	0	0	0
14	0	0	0
15	**1**	**0**	**0**
16	0	0	1

Table

	Titre	Genre
1	Vertigo	Suspense
2	Brazil	Science-fiction
3	Twin Peaks	Fantastique
4	Underground	Drame
5	Easy Rider	Drame
6	Psychose	Drame
7	Greystoke	Aventures
8	Shining	Fantastique
9	Annie Hall	Comédie
10	Jurassic Park	Science-fiction
11	Metropolis	Science-fiction
12	Manhattan	Comédie
13	Reservoir Dogs	Policier
14	Impitoyable	Western
15	Casablanca	Drame
16	Smoke	Comédie

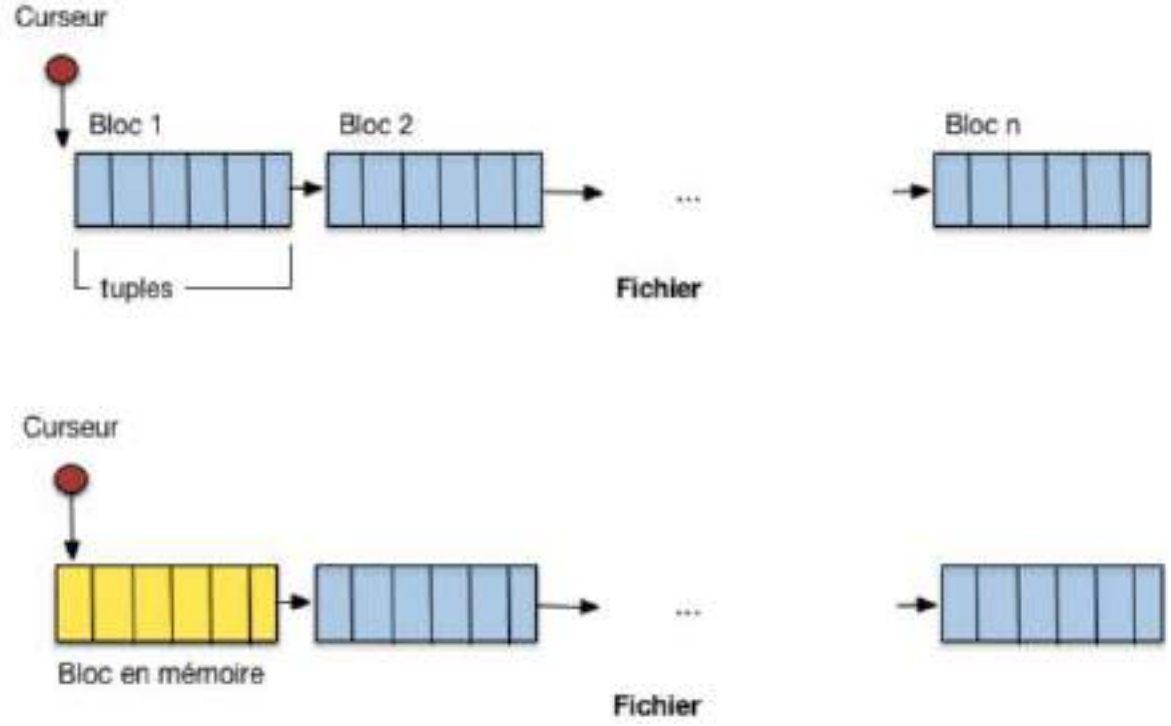
Le rôle de l’optimiseur : les opérateurs (PEP)

- **Choix des opérateurs pour construire le Plan d’Exécution Physique (PEP)**
- **3 principaux types d’opérateurs et plusieurs algorithmes pour chaque opération:**
  - Accès aux données (via les tables et les index)
    - parcours séquentiel de la table (*FullScan*)
    - parcours d’index (*IndexScan*)
    - accès par adresse (*DirectAccess*)
    - test de la condition (*Filter*)

Le rôle de l'optimiseur : l'opérateur d'accès aux données *FullScan (PEP)*

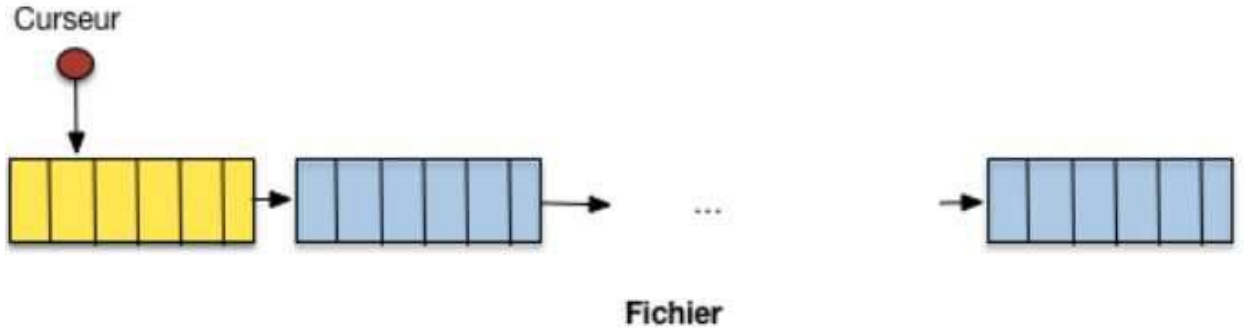
Accès aux données par parcours séquentiel de la table

- **Principe**
  - Lecture bloc par bloc du fichier.
  - Quand un bloc est en mémoire, traitement des enregistrements qu'il contient.
- **Fonctionnement**
  - Curseur positionné avant le premier n-uplet
  - open() = phase d'initialisation de l'opérateur
  - 1er next() = accès au premier bloc, placé en mémoire
    - Le curseur se place sur le premier n-uplet, qui est retourné comme résultat. Le temps de réponse est minimal.

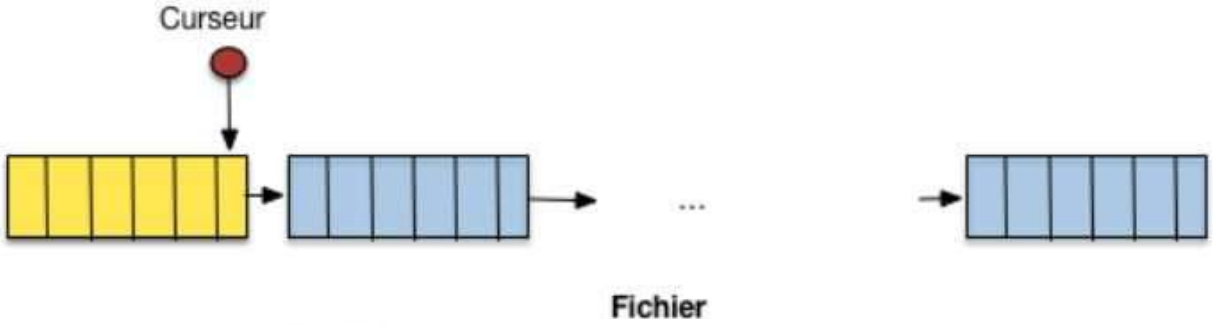


Le rôle de l'optimiseur : l'opérateur d'accès aux données *FullScan (PEP)*

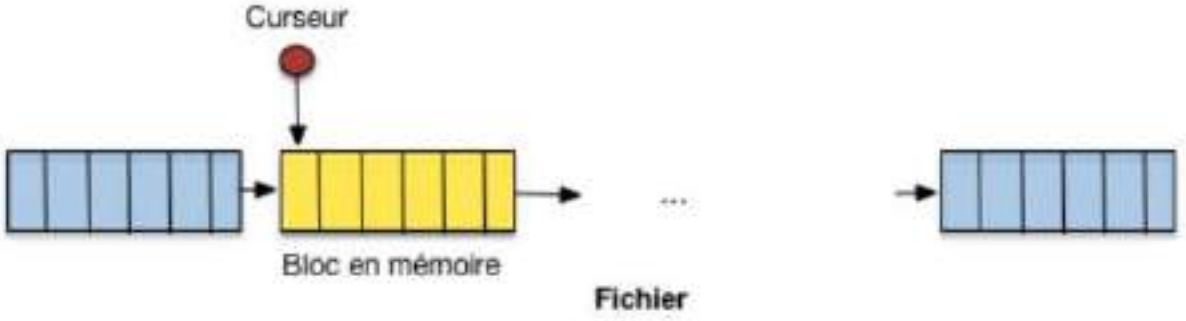
- 2ème next() = avancée d'un cran dans le parcours du bloc



- Après plusieurs next() : curseur positionné sur le dernier nuplet du bloc



- Appel suivant à next() = charge du second bloc en mémoire

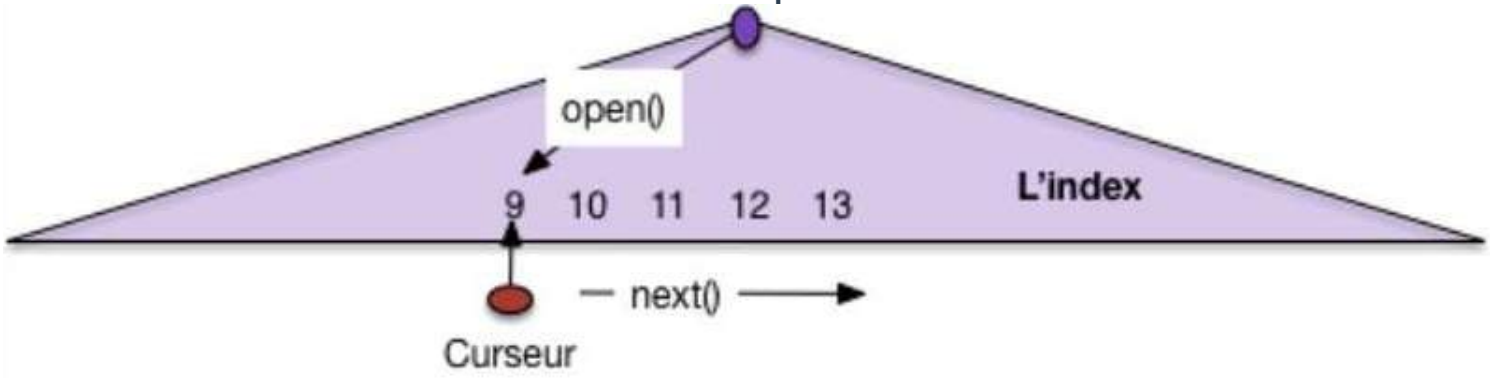


➡ Besoin en mémoire réduit (1 bloc) ; temps de réponse très court.

Le rôle de l’optimiseur : l’opérateur d’accès aux données *IndexScan (PEP)*

Accès aux données par parcours d’index

- **L'opérateur prend en entrée une valeur, ou un intervalle de valeurs.**
- **Parcours de l’index = arbre équilibré (arbre B+)**
  - Pendant le open() : parcours de la racine vers la feuille
  - À chaque appel à next() : parcours en séquence des feuilles
  - Renvoie des adresses courantes a chaque feuille

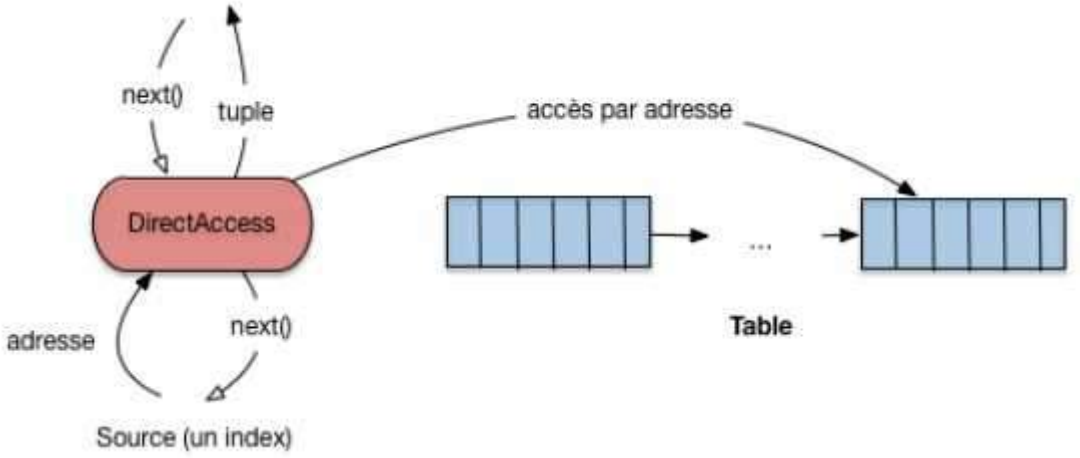


➡ **Très efficace, quelques lectures logiques (index en mémoire)**

Le rôle de l’optimiseur : l’opérateur d’accès aux données *DirectAccess (PEP)*

Accès aux données par accès par adresse

- **L’opérateur s'appuie sur un itérateur qui fournit des adresses d'enregistrement**
- **Fonctionnement**
  - 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.**

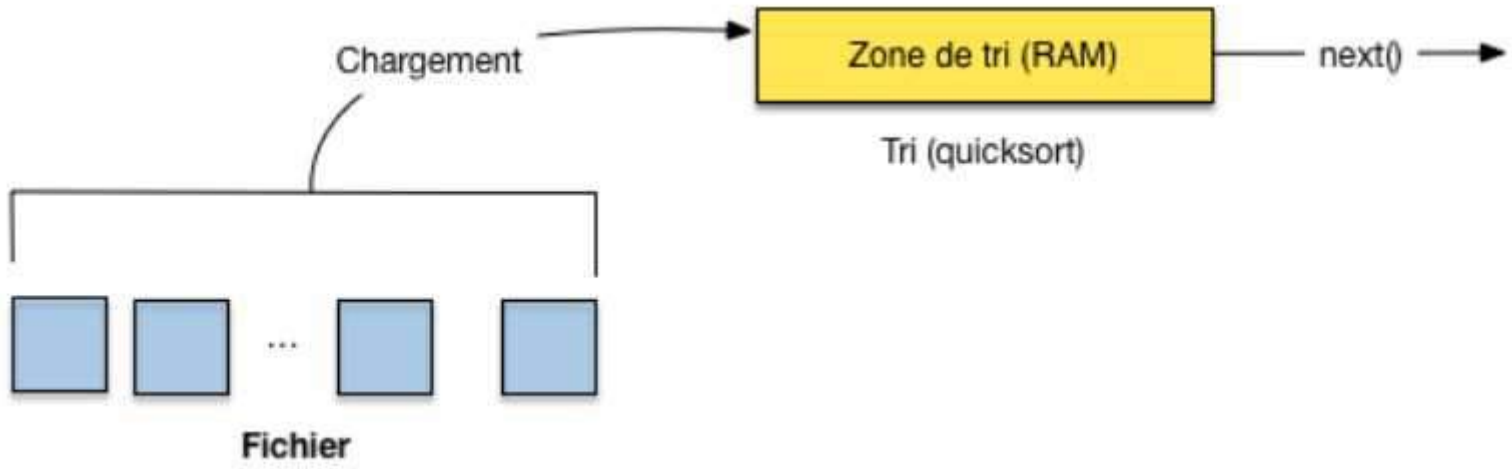
Le rôle de l’optimiseur : les opérateurs (PEP)

- **Choix des opérateurs pour construire le Plan d’Exécution Physique (PEP)**
- **3 principaux types d’opérateurs et plusieurs algorithmes pour chaque opération:**
  - Accès aux données (via les tables et les index)
    - parcours séquentiel de la table (*FullScan*)
    - parcours d’index (*IndexScan*)
    - accès par adresse (*DirectAccess*)
    - test de la condition (*Filter*)
  - Tri externe (*Tri-Fusion*) utilisé pour
    - algorithmes de jointure (sort/merge)
    - élimination des doublons (clause DISTINCT)
    - opérations de regroupement (GROUP BY)
    - opérations d’ordonnancement (ORDER BY)

    C’est une opération qui peut être très coûteuse sur de grands jeux de données !

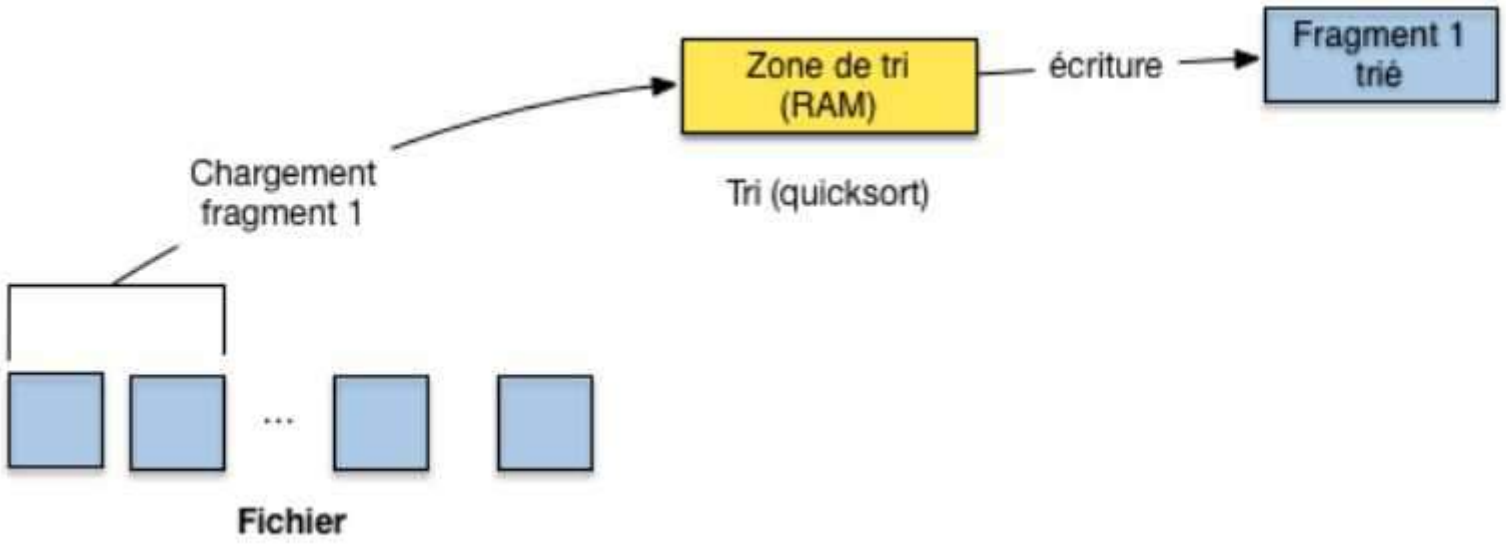
Le rôle de l’optimiseur : l’opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**



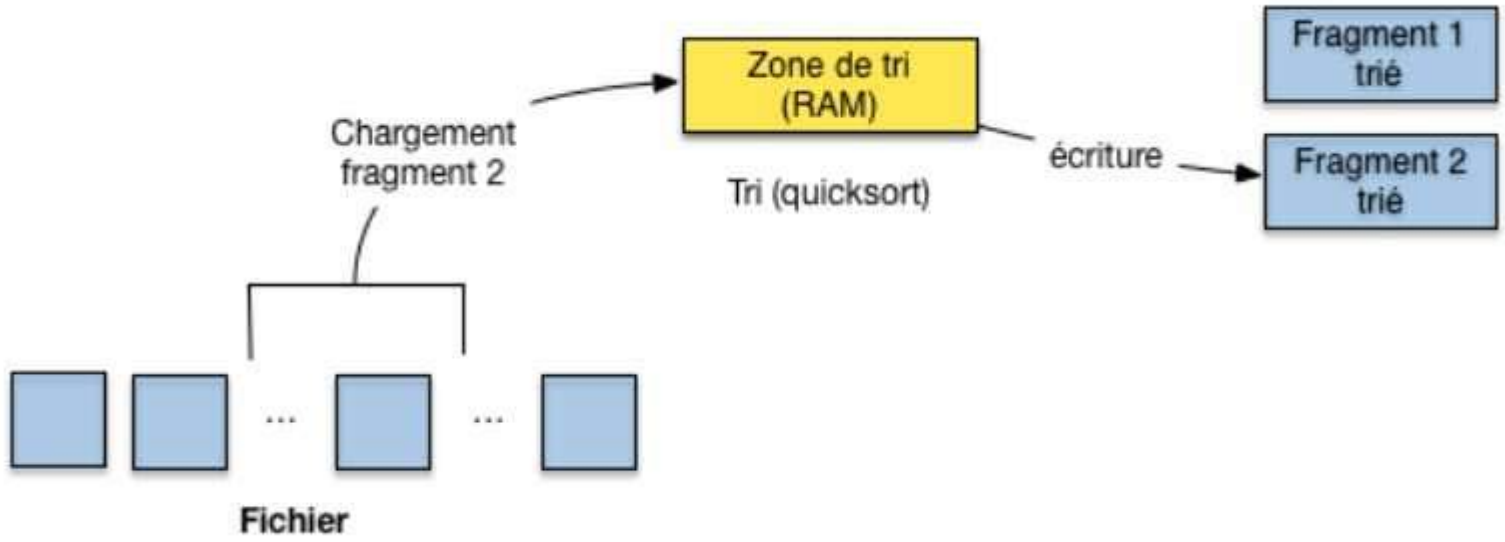
Le rôle de l'optimiseur : l'opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d'une phase de tri puis d'une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire



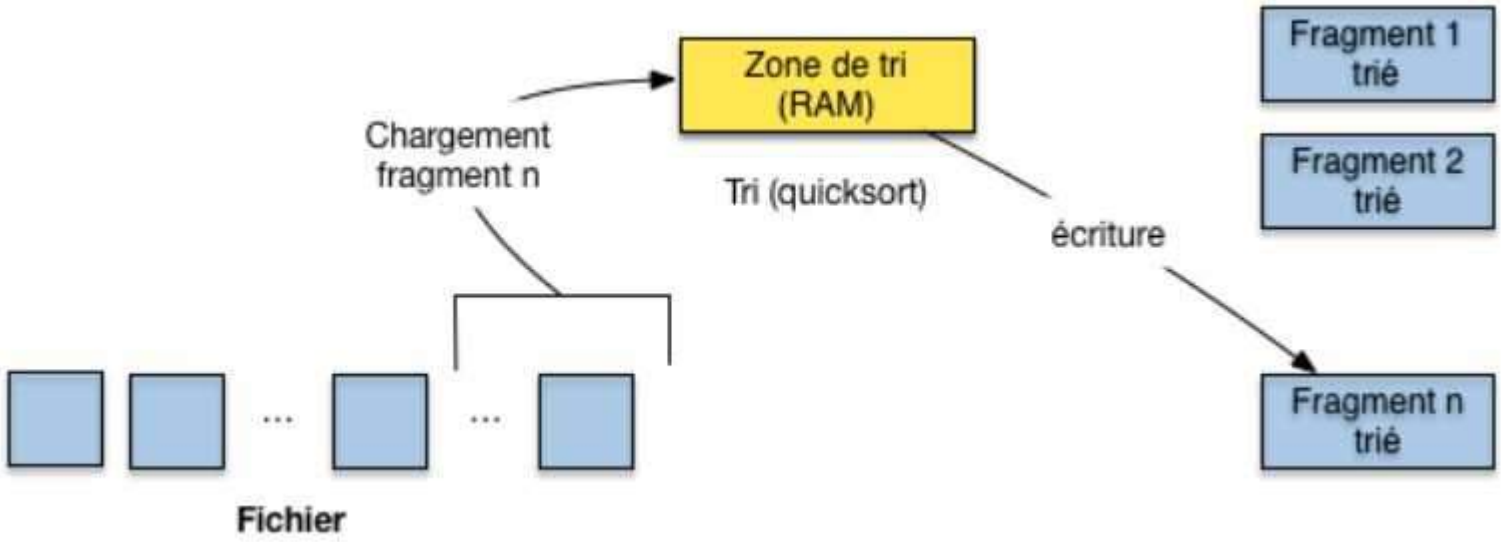
Le rôle de l'optimiseur : l'opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d'une phase de tri puis d'une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire



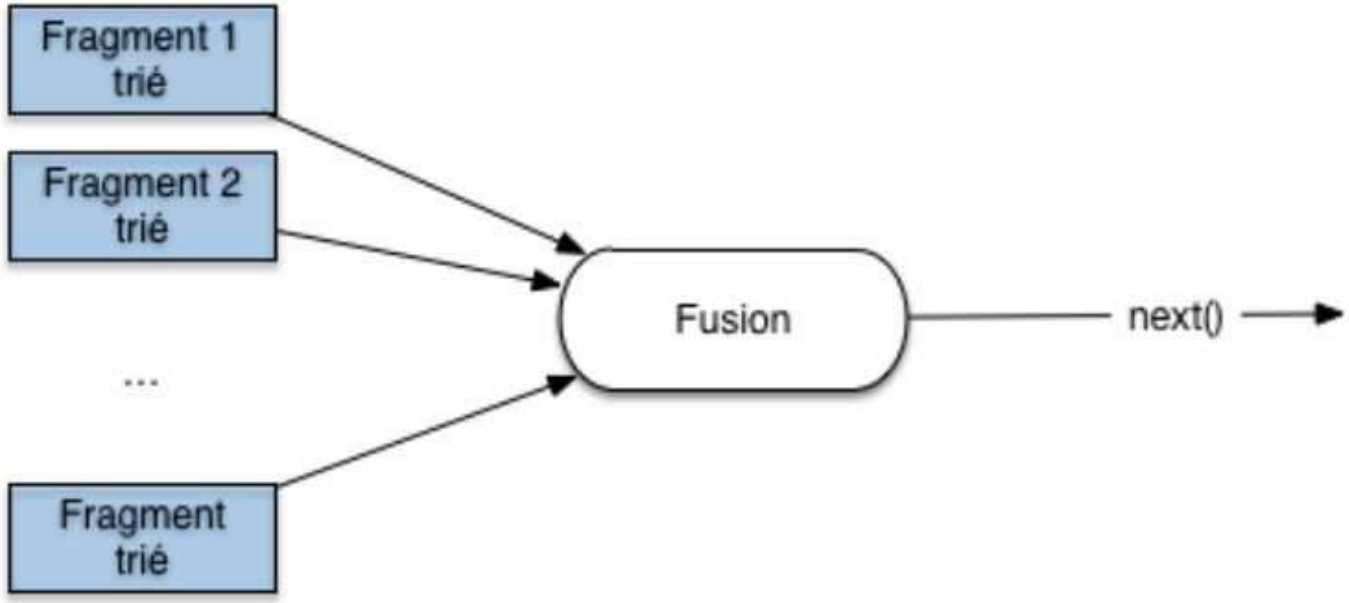
Le rôle de l'optimiseur : l'opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d'une phase de tri puis d'une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire



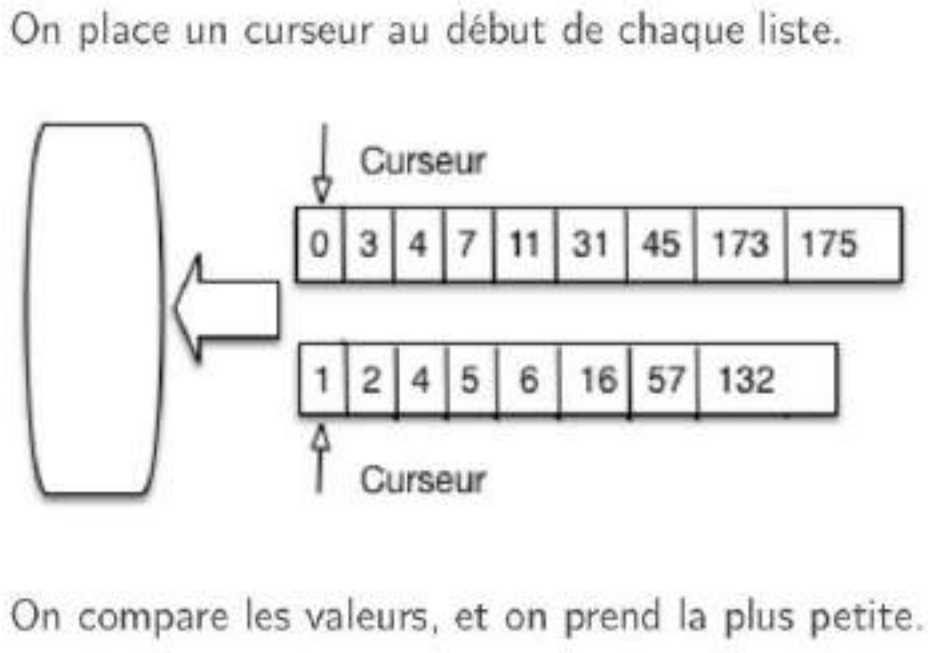
Le rôle de l’optimiseur : l’opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d’une phase de tri puis d’une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire
  - Fusion des fragments triés



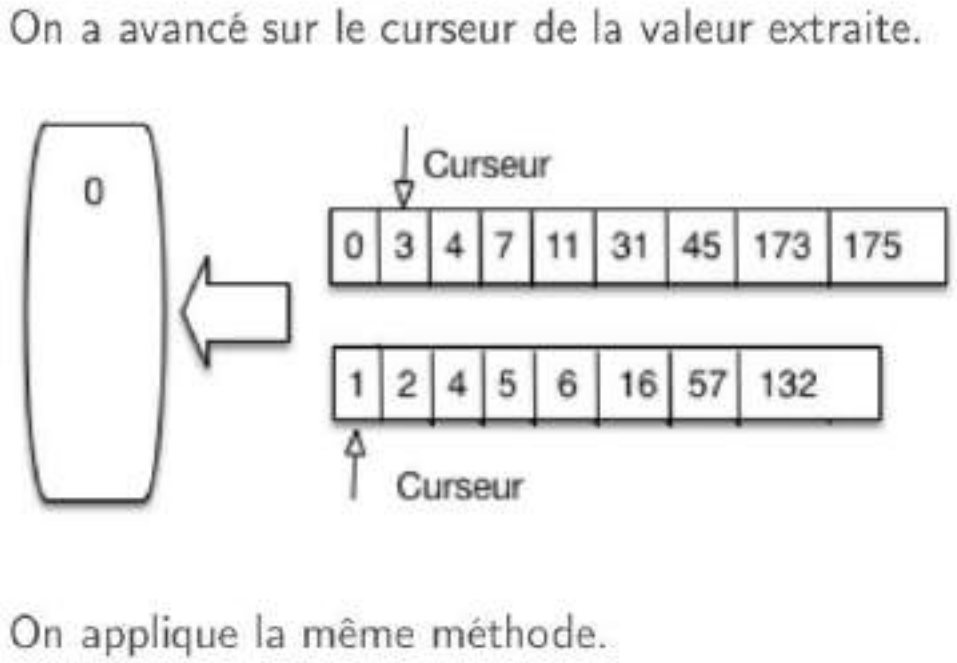
Le rôle de l’optimiseur : l’opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d’une phase de tri puis d’une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire
  - Fusion des fragments triés



Le rôle de l'optimiseur : l'opérateur de Tri-Fusion *(PEP)*

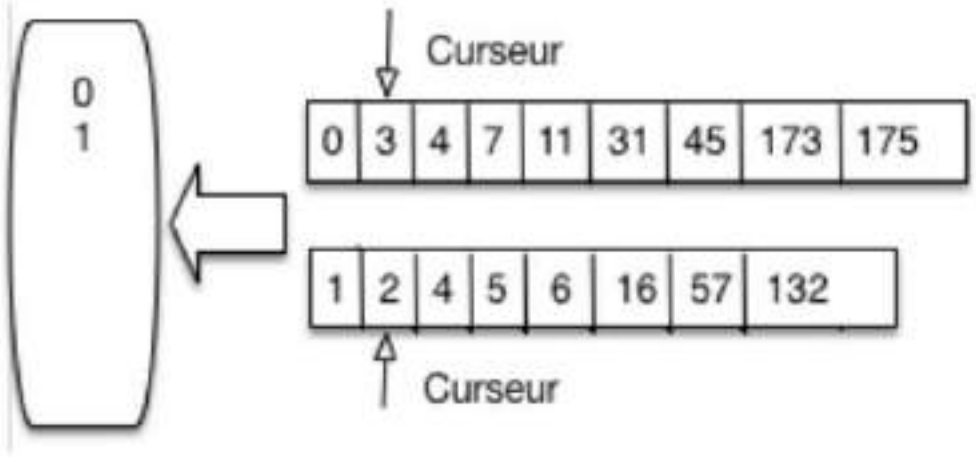
- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d'une phase de tri puis d'une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire
  - Fusion des fragments triés



Le rôle de l'optimiseur : l'opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d'une phase de tri puis d'une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire
  - Fusion des fragments triés

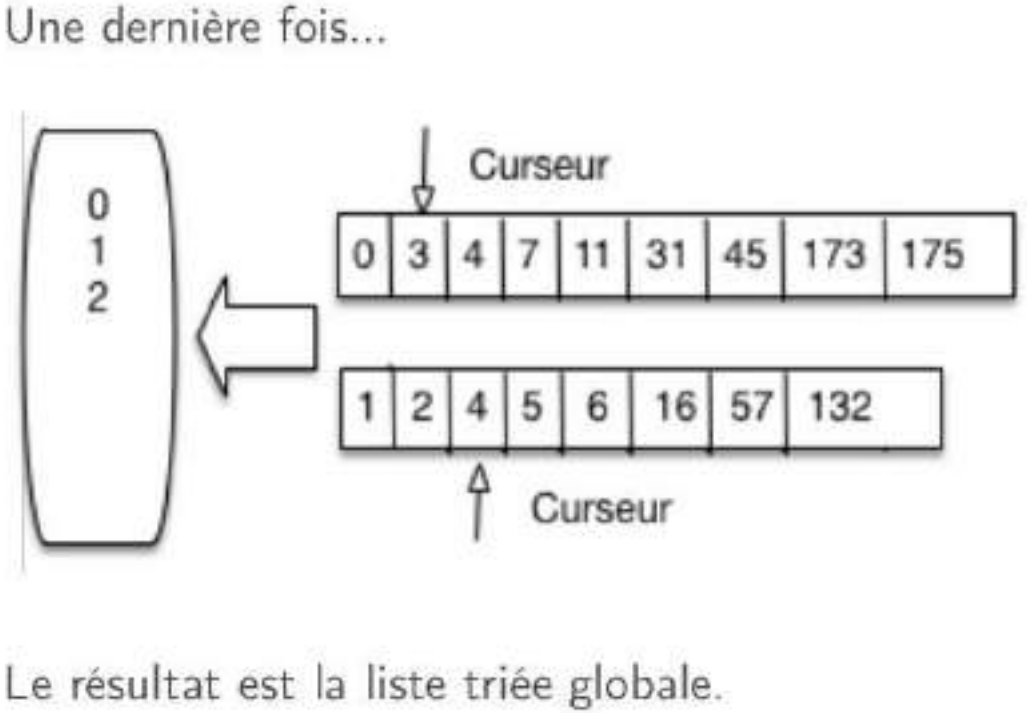
On continue sur le même principe.



On ne revient jamais en arrière.

Le rôle de l'optimiseur : l'opérateur de Tri-Fusion *(PEP)*

- **Si le fichier à trier tient en mémoire : chargement puis trie (quicksort), pas besoin de fusion**
- **Si le fichier ne tient pas en mémoire : besoin d'une phase de tri puis d'une phase de fusion**
  - fragmentation du fichier, pour chaque fragment : lecture, trie puis stockage en mémoire
  - Fusion des fragments triés



Le rôle de l’optimiseur : l’opérateur de Tri-Fusion *(PEP)*

- **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.
  - Et plus, si le volume de données est très important

Le rôle de l'optimiseur : les opérateurs (PEP)

- **Choix des opérateurs pour construire le Plan d'Exécution Physique (PEP)**
- **3 principaux types d'opérateurs et plusieurs algorithmes pour chaque opération:**
  - Accès aux données (via les tables et les index)
    - parcours séquentiel de la table (*FullScan*)
    - parcours d'index (*IndexScan*)
    - accès par adresse (*DirectAccess*)
    - test de la condition (*Filter*)
  - Tri externe (*Tri-Fusion*) utilisé pour
    - algorithmes de jointure (sort/merge)
    - élimination des doublons (clause DISTINCT)
    - opérations de regroupement (GROUP BY)
    - opérations d'ordonnancement (ORDER BY)

    C'est une opération qui peut être très coûteuse sur de grands jeux de données !
  - Jointures avec index : boucles imbriquées indexée (*Index join*)
  - Jointures sans index :
    - boucles imbriquées (*Nested loop join*)
    - tri-fusion (*Merge sort join*)
    - hachage (*Hash join*)

Le rôle de l'optimiseur : l'opérateur de jointure Index *join (PEP)*

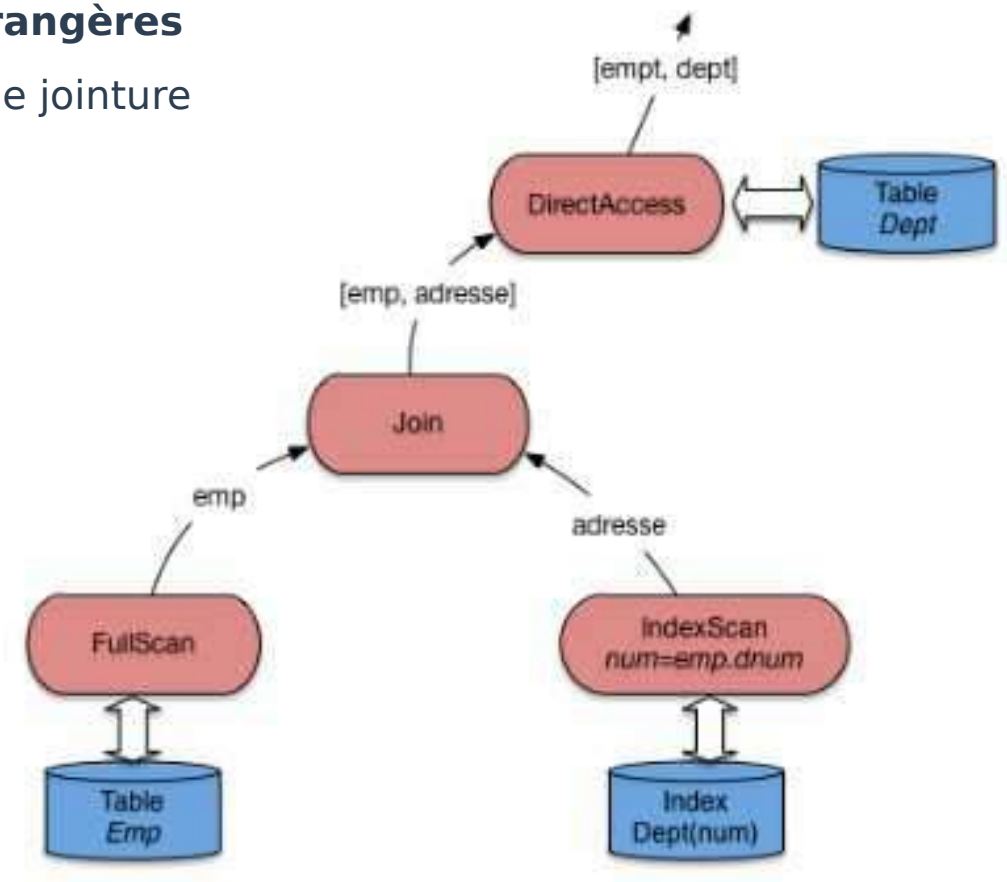
Jointure avec index

- **Très courant**
- **Jointure naturelle sur les clés primaires/étrangères**

  ➡ Garantit au moins un index sur la condition de jointure

- **Fonctionnement**
  - parcours séquentiel de la 1ère table contenant la clé étrangère
  - utilisation de la clé étrangère pour un accès par index à la clé primaire de la 2ème table
  - utilisation de l'adresse (obtenu avec l'index) avec un accès direct pour récupérer les informations de la 2ème table
- **Avantages** :
  - Efficace (un parcours, plus des recherches par adresse)
  - Favorise le temps de réponse et le temps d'exécution

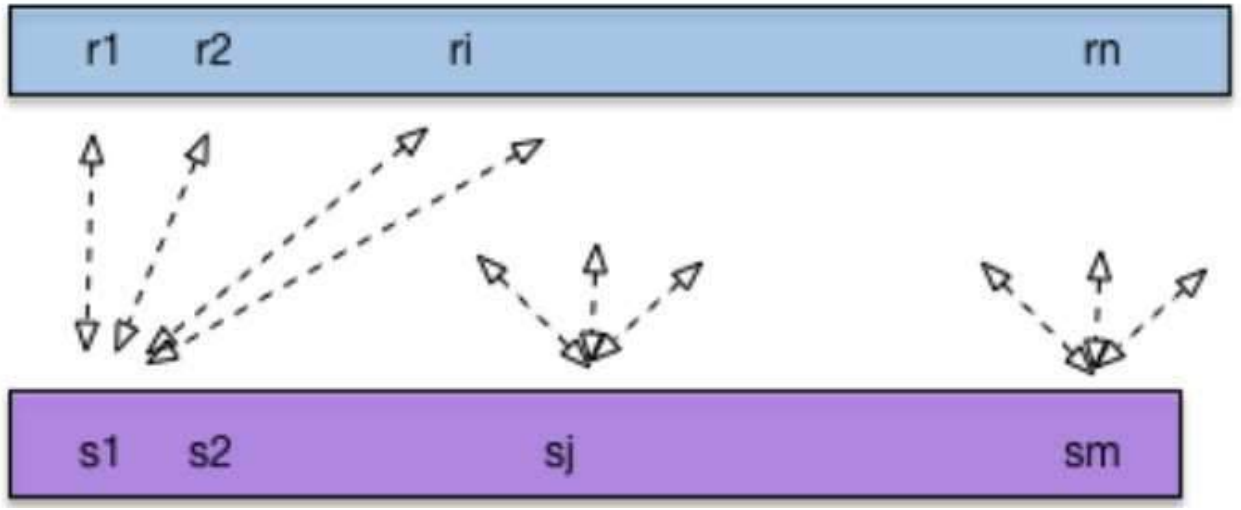
```
select * from emp e, dept d
where e.dnum = d.num
```



Le rôle de l'optimiseur : l'opérateur de jointure *Nested loop join (PEP)*

Jointure sans index par boucles imbriquées

- **Énumérer toutes les solutions possibles : la table à droite de la jointure est confrontée à chaque tuple de la table à gauche**

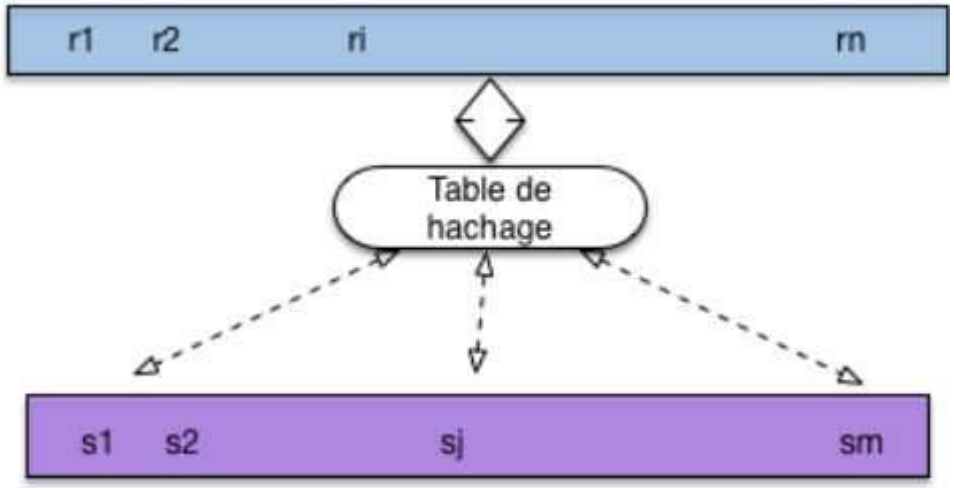


- **Si la table à droite est indexée sur l'attribut qui sert pour la jointure, cette approche peut donner des résultats corrects (sinon elle peut s'avérer coûteuse)**
- **Coût quadratique. Acceptable pour deux petites tables.**

Le rôle de l'optimiseur : l'opérateur de jointure *Hash Join (PEP)*

Jointure par hachage

- **Principe**
  - Les données des deux tables sont lues en mémoire, et les données de la table principale sont hachées
  - Pour chaque ligne de la table secondaire, la fonction de hachage est appliquée sur les colonnes de la jointure pour décider s'il y a un match ou non avec une ligne de la table principale.



- **Le plus efficace dans le meilleur des cas**
  - Très rapide quand une des deux tables est petite (n fois la taille de la mémoire avec n petit, <3)
  - Pas très robuste (efficacité dépend de plusieurs facteurs : fonction de hachage, nb/taille casiers, ...)

Le rôle de l’optimiseur : les opérateurs de jointure *(PEP)*

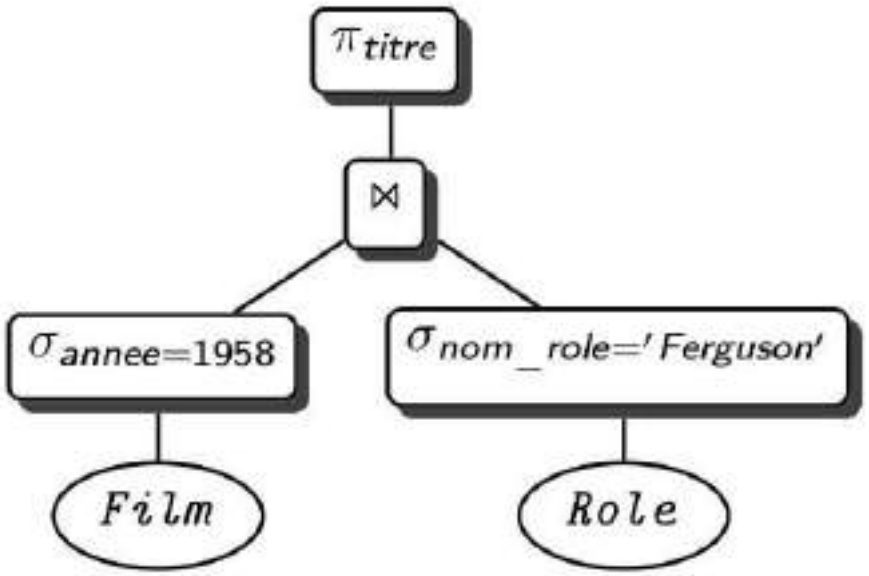
- **Un opérateur potentiellement coûteux**

- **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 imbriquée indexée.
  - Si une des deux tables est beaucoup plus petite que l’autre : jointure par hachage.
  - Sinon : jointure par tri-fusion

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

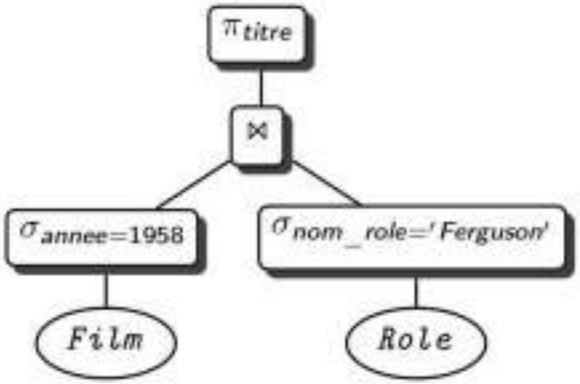
Un exemple de plan d'exécution physique (PEP)

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



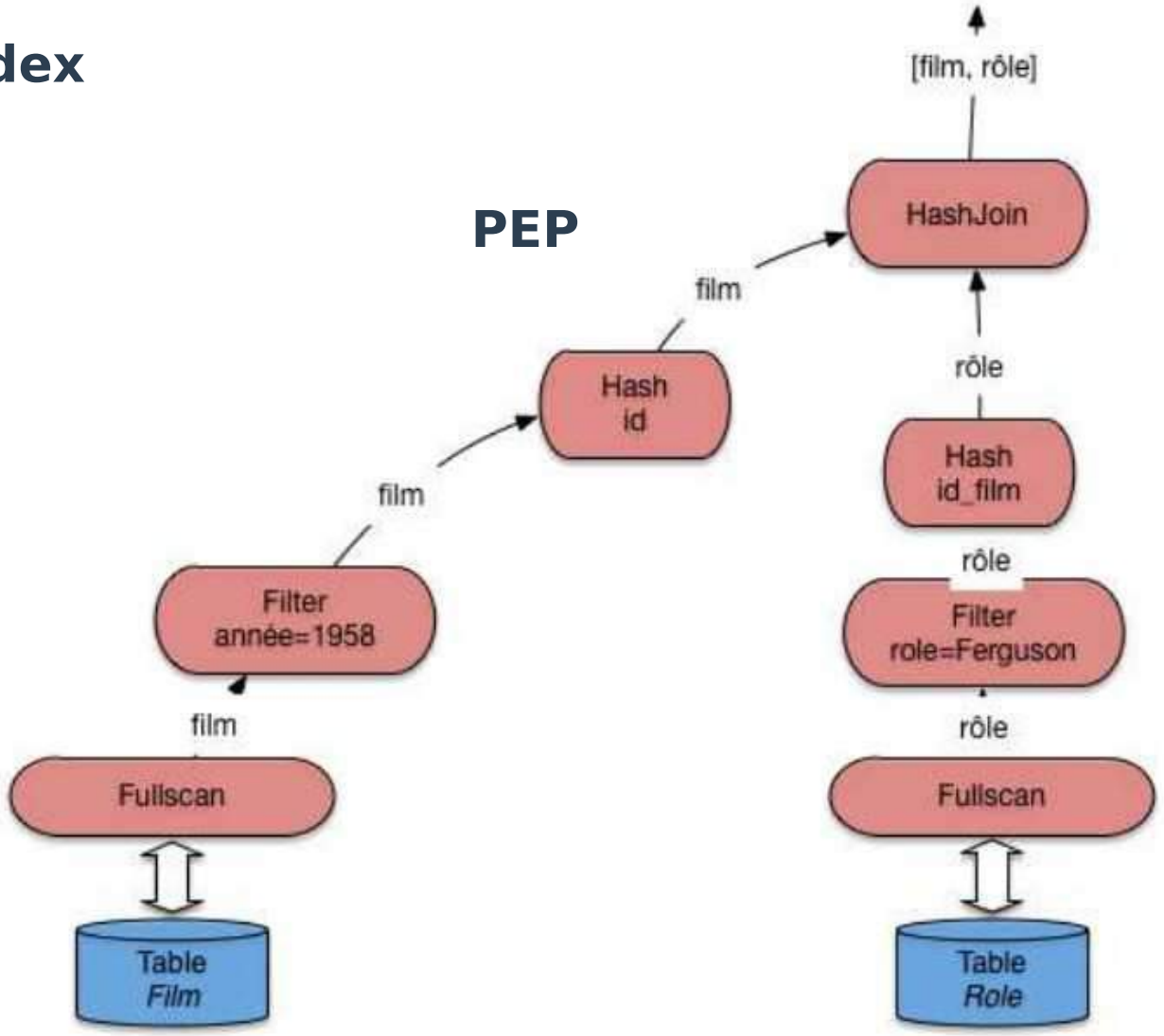
Un exemple de plan d'exécution physique (PEP)

- **Sans utiliser d'index**



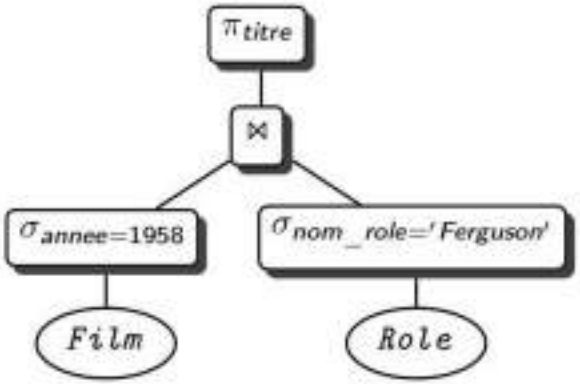
PEL

PEP



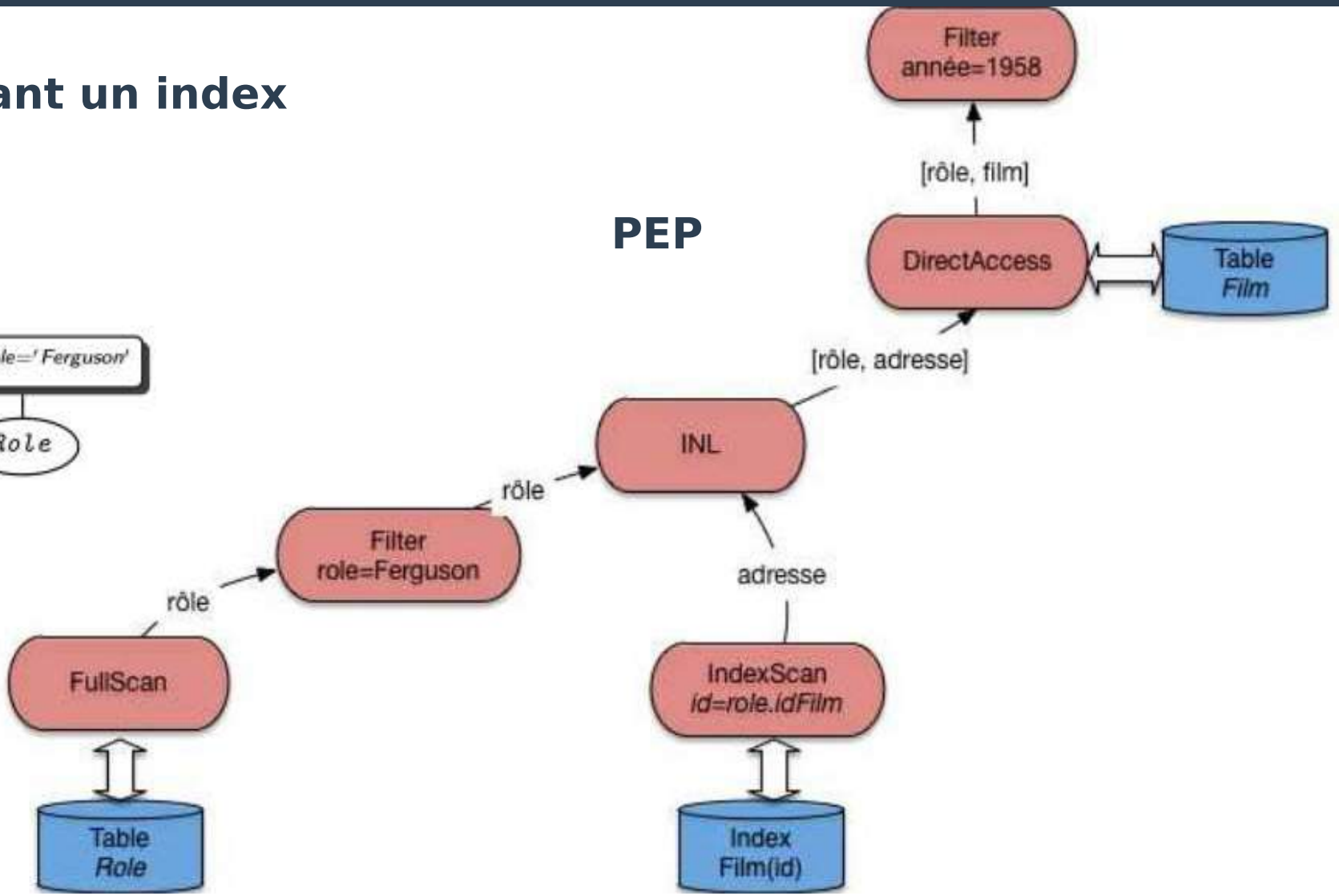
Un exemple de plan d'exécution physique (PEP)

- **En utilisant un index**



PEL

PEP



Le rôle de l’optimiseur : les opérateurs – calcul de coût(PEP)

- **Le coût correspond au temps nécessaire pour obtenir le résultat d’une requête : plusieurs facteurs (accès disques, CPU, échange réseau)**
- **Le coût dépend de l'algorithme (index, hachage ou balayage).**
- **Parmi tous les plans d’exécutions, celui avec le coût le moins élevé est choisi**
  - Le coût estimé en fonction de statistiques sur les tables (nombre de n-uplet, taille des n-uplets, nombre de valeurs distinctes, plus basse/haute valeur, ….)
- **Les accès disques correspondent généralement à la plus grande part du coût d’une requête. Il est possible d’estimer le nombre d’accès disque pour une requête. Il faut prendre en compte :**
  - Le nombre de recherche de données * le temps moyen d’une recherche
  - Le nombre de lecture des blocs * le temps moyen d’une lecture de bloc
  - Le nombre de bloc à écrire * le temps moyen d’écriture d’un bloc
    - Le temps d’écriture est plus important que le temps de lecture
    - Après une écriture, une lecture est effectuée afin de vérifier qu’il n’y a pas d’erreur

Le rôle de l’optimiseur : les opérateurs - calcul de coût (PEP)

- **Méthode statique ou encore optimisation basée sur des règles (RBO) – ancienne méthode (ORACLE < v7)**
  - en fonction des chemins possibles d’accès aux tables
  - en privilégiant en premier lieu les opérations les moins coûteuses

- **Méthode statistique ou encore optimisation basée sur le coût (CBO) – méthode actuelle (ORACLE ≥ v7)**
  - en fonction de l’ensemble des statistiques collectées en tâches de fond sur les différentes tables du schéma
  - 2 objectifs privilégiés : le temps d’exécution / débit (par défaut) ou le temps de réponse
  - optimisation paramétrable (mode, tuning, directive, ...)

Le rôle de l’optimiseur : les opérateurs – un exemple de calcul de coût (PEP)

- **Coût des opérateurs d’accès aux données : parcours séquentiel** (*FullScan*) **VS parcours d’index** (*IndexScan*)
  - Le fichier fait 500 Mo, une lecture de bloc prend 0,01 s (10 millisecondes).
  - Un parcours séquentiel lira tout le fichier (ou la moitié pour une recherche par clé). Donc ça prendra 5 secondes.
  - Une recherche par index implique 2 ou 3 accès pour parcourir l’index, et un seul accès pour lire l’enregistrement : soit 4 × 0.01 = 0.04 s, (4 ms).

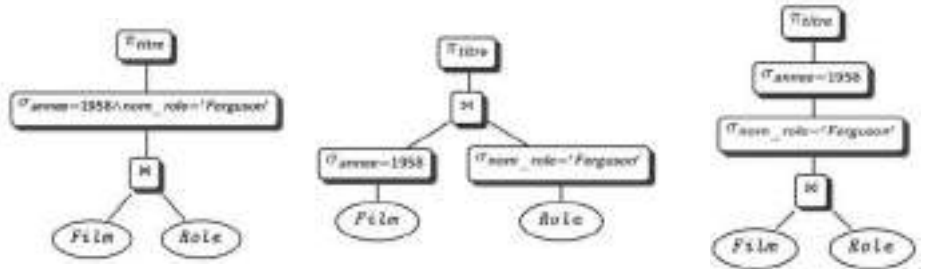
➡ Donc le parcours séquentiel est mille fois plus cher.

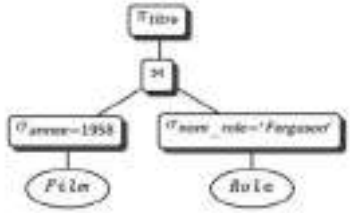
Le rôle de l'optimiseur

Trouver les expressions équivalentes

Requête SQL ➡ **Plan d'exécution logique - PEL (l'algèbre)**

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

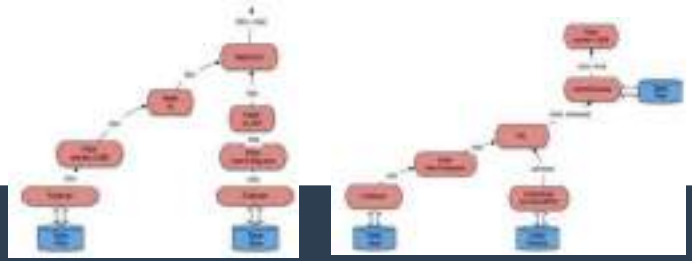




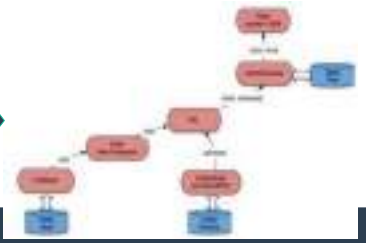
Évaluation des coûts pour trouver le meilleur PEL

Choisir le bon algorithme pour chaque opération

Plan d'exécution physique – PEP (opérateurs)



Évaluation des coûts pour trouver le meilleur PEP ➡



Le rôle de l’optimiseur : résumé…

- **Un plan d'exécution**
  - **Logique** : c’est une expression algébrique combinant des opérateurs algébriques
  - **Physique** : c'est un programme combinant des opérateurs physiques
- **Ils ont la forme d'un arbre : chaque nœud est un opérateur qui**
  - prend des données en entrée
  - applique un traitement
  - produit les données traitées en sortie
- **La phase d'optimisation proprement dite :**
  - Pour une requête, le système a le choix entre plusieurs plans d'exécution (logiques puis physiques).
  - Ils diffèrent par l'ordre des opérations, les algorithmes, les chemins d'accès.
  - Pour chaque plan on peut estimer le coût de chaque opération et la taille du résultat

➡ **Objectif : diminuer le plus vite possible la taille des données manipulées.**

Retour au début … Mais pourquoi on s’intéresse à l’optimisation ?

- **Car cela a un impact sur le mouvement des blocs de données en mémoire et que les performances du SGBD dépendent des mouvements des blocs (cf. CM1 sur le stockage : Database Storage)**
  - Volume de données (>~1000 lignes)
  - Requêtes consommatrices de temps et de ressources

- **Optimisation = tâche du SGBD**
- **Mais l’utilisateur a un rôle à jouer :**
  - requête mal écrite
    ➡ mauvaise optimisation
  - possibilité d’influer sur l’optimisation faite par le SGBD
    ➡ meilleure optimisation

➡ **Comprendre les mécanismes de l’optimisation de requêtes afin d’écrire des requêtes bien construites**

Les bonnes pratiques pour écrire une requête

- **Choisir la bonne méthode de jointure :**
  - Jointure interne ([INNER] JOIN) : enregistrements quand la condition est vraie dans les deux tables.
  - Jointure externe gauche (LEFT [OUTER] JOIN) : enregistrements de la table de gauche + enregistrements correspondants de la table de droite (NULL si pas de correspondance)
  - Jointure externe droite (RIGHT [OUTER] JOIN) : enregistrements de la table de droite + enregistrements correspondants de la table de gauche (NULL si pas de correspondance).
  - Jointure externe complète (FULL [OUTER] JOIN): enregistrements des 2 tables et mise en correspondance si possible (NULL si pas de correspondance).

- **Décomposer les requêtes complexes en utilisant des vues (matérialisées)**

Les bonnes pratiques pour écrire une requête

- **Éviter les opérations inutiles**
  - le select *
  - le tri
  - limiter l'utilisation de DISTINCT
  - Éviter les ORDER BY et GROUP BY inutiles
  - filtrer les données le plus tôt possible dans le cas de requêtes imbriquées et de jointures

- **Favoriser les opérations les moins coûteuses**
  - Favoriser les UNION/UNION ALL aux OR
  - Favoriser le EXISTS par rapport au IN lorsque la liste à parcourir est issue d'une sous-requête et pas d'une liste statique
  - Attention au IN/NOT IN lorsqu'il y a des valeurs nulles : il ne peut pas les comparer et considère qu'elles n'existent pas.

Les bonnes pratiques pour écrire une requête

- **Utiliser une indexation correcte :**
  - Indexer les colonnes fréquemment interrogées MAIS éviter d'utiliser des index inutiles (ralentissement lors des mise à jour)
  - Utiliser des index sur les colonnes de jointure
  - Éviter d’utiliser des index si la sélectivité (pourcentage des lignes concernées) est basse
  - index non utilisé si :
    - fonction ou d’opérateur utilisés sur une colonne indexée
    - comparaison des colonnes indexées avec la valeur null

- **Comprendre les mécanismes de l’optimisation de requêtes pour :**
  - Analyser les plans d'exécution des requêtes pour influer sur l’optimisation en utilisant des instructions spéciales
  - Surveiller et optimiser les statistiques des bases de données

L’optimiseur ORACLE

L’optimiseur ORACLE

- **L’optimiseur ORACLE suit une approche classique :**
  - Génération de plusieurs plans d’exécution.
  - Estimation du coût de chaque plan généré.
  - Choix du meilleur et exécution.
- **Tout ceci est automatique, mais il est possible d’influer, voire de forcer le plan d’exécution**
- **Paramètres pour l’estimation du coût :**
  - Les chemins d’accès disponibles.
  - Les opérations physiques de traitement des résultats intermédiaires.
  - Des statistiques sur les tables concernées (taille, sélectivité) : appel explicite à l’outil ANALYZE.
  - Les ressources disponibles.

L’optimiseur ORACLE : les paramètres

- **Principaux paramètres (doc Oracle) :**
  - OPTIMIZER_MODE
    - RULE : heuristique, utilisé seulement lorsque des statistiques ne sont pas disponibles
    - CHOOSE : Oracle choisit (RULE / ALL_ROWS) en fonction de la présence de statistiques
    - FIRST_ROW [1, 10, 100, 1000] : minimise temps de réponse (obtention des 1ère lignes)
    - ALL_ROWS : minimise le temps total
  - SORT_AREA_SIZE (taille de la zone de tri).
  - HASH_AREA_SIZE (taille de la zone de hachage).
  - HASH_JOIN_ENABLED considère les jointures par hachage.

➡ **ALTER SESSION SET [OPTIMIZER_MODE | SORT_AREA_SIZE | …] = …**

L'optimiseur ORACLE : les règles

- **Basé sur des règles**
  - Mode RULE
  - Priorité des règles

Rang	Chemin d'accès
1	Sélection par ROWID
2	Sélection d'une ligne par jointure dans une organisation par index groupant ou hachage hétérogène (CLUSTER)
3	Sélection d'une ligne par hachage sur clé candidate (PRIMARY ou UNIQUE)
4	Sélection d'une ligne par clé candidate
5	Jointure par une organisation par index groupant ou hachage hétérogène (CLUSTER)
6	Sélection par égalité sur clé de hachage (HASH CLUSTER)
7	Sélection par égalité sur clé d'index groupant (CLUSTER)
8	Sélection par égalité sur clé composée
9	Sélection par égalité sur clé simple d'index secondaire
10	Sélection par intervalle borné sur clé indexée
11	Sélection par intervalle non borné sur clé indexée
12	Tri-fusion
13	MAX ou MIN d'une colonne indexée
14	ORDER BY sur colonne indexée
15	Balayage

L'optimiseur ORACLE : les statistiques

Création des statistiques : utiliser les fonctions de DBMS_STATS

- **Calcul pour une table ou tout le shéma :**
  - `execute dbms_stats.gather_table_stats('login','nom_table');`
  - `execute dbms_stats.gather_schema_stats('login');`

- **Calcul de la taille des n-uplets et du nombre de lignes (blocs) :**

```
ANALYZE TABLE Film COMPUTE STATISTICS FOR TABLE;
```

- **Analyse des index (nombre blocs feuilles, profondeur, nombre de blocs concernés par la requête) :**

```
ANALYZE TABLE Film COMPUTE STATISTICS FOR ALL INDEXES;
```

- **Analyse de la distribution des valeurs (nombre valeurs distinctes, valeurs nulles, histogramme)**

```
ANALYZE TABLE Film COMPUTE STATISTICS FOR COLUMNS titre, genre;
```

L’optimiseur ORACLE : les chemins d’accès

- **Parcours séquentiel**

  `TABLE ACCESS FULL`

- **Accès direct par adresse**

  `TABLE ACCESS BY (INDEX|USER|...) ROWID`

- **Accès par index**

  `INDEX (UNIQUE|RANGE|...) SCAN`

- **Accès par hachage**

  `TABLE ACCESS HASH`

- **Accès par cluster (structure où plusieurs tables partagent le même bloc de données basé sur une clé commune)**

  `TABLE ACCESS CLUSTER`

L’optimiseur ORACLE : les opérateurs

- **ORACLE peut utiliser trois algorithmes de jointures :**
  - Boucles imbriquées quand il y a au moins un index : NESTED LOOP
  - Tri/fusion quand il n’y a pas d’index : SORT / MERGE JOIN
  - Jointure par hachage : HASH JOIN

- **Autres opérations**
  - Union d'ensembles d'articles: CONCATENATION, UNION
  - Intersection d'ensembles d'articles: INTERSECTION
  - Différence d'ensembles d'articles: MINUS
  - Filtrage d'articles d'une table basé sur une autre table: FILTER
  - Intersection d'ensembles de ROWID: AND-EQUAL
  - ...

L'optimiseur ORACLE : les opérateurs

OPERATIONS	OPTIONS	SIGNIFICATION
AGGREGATE	GROUP BY	Une recherche d'une seule ligne qui est le résultat de l'application d'une fonction de group à un groupe de lignes sélectionnées.
AND-EQUAL		Une opération qui a en entrée des ensembles de rowids et retourne l'intersection de ces ensembles en éliminant les doublants. Cette opération est utilisée par le chemin d'accès par index.
CONNECT BY		Recherche de ligne dans un ordre hiérarchique
COUNTING		Opération qui compte le nombre de lignes sélectionnées.
FILTER		Accepte un ensemble de ligne, appliqué un filter pour en éliminer quelque unes et retourne le reste.
FIRST ROW		Recherché de le première ligne seulement.
FOR UPDATE		Opération qui recherche et verrouille les lignes pour une mise à jour
INDEX	UNIQUE SCAN	Recherche d'une seule valeur ROWID d'un index.
INDEX	RANGE SCAN	Recherche d'une ou plusieurs valeurs ROWID d'un index. L'index est parcouru dans un ordre croissant.
INDEX	RANGE SCAN DESCENDING	Recherche d'un ou plusieurs ROWID d'un index. L'index est parcouru dans un ordre décroissant.
INTERSECTION		Opération qui accepte deux ensembles et retourne l'intersection en éliminant les doublons.
MARGE JOIN+		Accepte deux ensembles de lignes (chacun est trié selon un critère), combine chaque ligne du premier ensemble avec ses correspondants du deuxième et retourne le résultat.
MARGE JOIN+	OUTER	MARGE JOIN pour effectuer une jointure externe
MINIUS		Différence de deux ensembles de lignes.
NESTED LOOPS		Opération qui accepte deux ensembles, l'un externe et l'autre interne. Oracle compare chaque ligne de l'ensemble externe avec chaque ligne de l'ensemble interne et retourne celle qui satisfont une condition.
NESTED LOOPS	OUTER	Une boucle imbriquée pour effectuer une jointure externe.
PROJECTION		Opération interne
REMOTE		Recherche de données d'une base distante.
SEQUENCE		Opération nécessitant l'accès à des valeurs du séquenceur
SORT	UNIQUE	Tri d'un ensemble de lignes pour éliminer les doublons.
SORT	GROUP BY	Opération qui fait le tri à l'intérieur de groupes
SORT	JOIN	Tri avant la jointure (MERGE-JOIN).
SORT	ORDER BY	Tri pour un ORDER BY.
TABLE ACCESS	FULL	Obtention de toutes lignes d'une table.
TABLE ACCESS	CLUSTER	Obtention des lignes selon la valeur de la clé d'un cluster indexé.
TABLE ACCESS	HASH	Obtention des lignes selon la valeur de la clé d'un hash cluster
TABLE ACCESS	BY ROW ID	Obtention des lignes on se basant sur les valeurs ROWID.

L’optimiseur ORACLE : les directives

- **Directive (hint) utilisée pour influencer l’optimiseur**
  - imposer un opérateur spécifique,
  - faire le choix sur l’exploitation d’un index ou non, ...

➡ **Utile en mode de conception ou lorsque l’optimiseur ne choisit pas un plan optimal (ex: mauvaises statistiques)**

- **Insertion des directives dans la requête à exécuter**
  - Exemples de directives (doc ORACLE)
    - Ne pas exploiter l’index d’une table : /*+ NO_INDEX(nom_table) */
      ```
      select /*+ NO_INDEX(Commune) */ * from Commune ;
      ```
    - Utiliser l’opérateur FullScan sur une table : /*+ full(nom_table) */
      ```
      select /*+ full(f) */ * from f where nom_f like ’d%’;
      ```

- **Les directives ne peuvent être efficaces que si le traitement demandé fait partie intégrante des plans d’exécution initialement envisagés par l’optimiseur.**

L’optimiseur ORACLE : le plan d’exécution

- **Comment obtenir le plan d’exécution d’une requête ?**
  - commande pour le calculer : « explain plan for requête ; »
  - Exemple : `explain plan for select * from emp where num=33000 ;`
  - commande pour l’afficher : utilisation d’une table des métadonnées qui stocke les plans d’exécution
    « `Select plan_table_output from table(dbms_xplan.display());` »

```
|    0 | SELECT STATEMENT
|*   1 |    TABLE ACCESS FULL| EMP

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("NUM"=33000)
```

  - A NOTER : avec ces commandes, la requête n’est pas exécutée !!!

L’optimiseur ORACLE : le plan d’exécution

- **Comment lire un plan d’exécution ?**
  - Parcours des étapes de haut en bas jusqu’à en trouver une qui n’a pas de fille (pas d’étape indentée en dessous)
  - Traitement de cette étape sans fille ainsi que de ses sœurs (étapes de même indentation)
  - Traitement de toutes les étapes mères jusqu’à trouver une étape qui a une sœur
  - Traitement de la sœur conformément à l’étape 1.

```
-----------------------------------------------------
|   0 | SELECT STATEMENT
|   1 |  NESTED LOOPS
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP
|*  3 |    INDEX FULL SCAN           | N_DEPT_IDX |
|   4 |   TABLE ACCESS BY INDEX ROWID| DEPT       |
|*  5 |    INDEX UNIQUE SCAN         | DEPT_PK    |
-----------------------------------------------------
```

L’optimiseur ORACLE : le plan d’exécution

- **Comment lire un plan d’exécution ?**
  - Id : Identifiant de l’opérateur ;
  - Operation : type d’opération utilisée
  - Name : nom de la relation utilisée ;
  - Rows : le nombre de lignes qu’Oracle pense transférer. ,
  - Bytes : nombre d’octets qu’oracle pense transférer.
  - Cost : coût estimé par oracle

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	24	11 (10)	00:00:01
1	SORT AGGREGATE		1	24		
* 2	HASH JOIN		9940	232K	11 (10)	00:00:01
* 3	TABLE ACCESS FULL	C	1000	8000	4 (0)	00:00:01
* 4	HASH JOIN		995	15920	7 (15)	00:00:01
5	TABLE ACCESS BY INDEX ROWID	B	100	800	2 (0)	00:00:01
* 6	INDEX RANGE SCAN	B_STATUS_IDX	100		1 (0)	00:00:01
* 7	TABLE ACCESS FULL	A	1000	8000	4 (0)	00:00:01

L’optimiseur ORACLE : le plan d’exécution

- **3 méthodes différentes pour appliquer les clauses where :**
  - Prédicats d'accès (« access »)
  - Prédicat de filtre d'index
  - Prédicat de filtre au niveau table
- **Information sur les prédicats :**
  - Numérotation des prédicats = colonne « Id » du plan d'exécution.
  - Étoile dans le plan d’exécution pour marquer les opérations qui ont des informations de prédicats

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |               |     1 |    24 |    11  (10)| 00:00:01 |
|   1 |  SORT AGGREGATE                |               |     1 |    24 |            |          |
|*  2 |   HASH JOIN                    |               |  9940 |  232K|    11  (10)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL           | C             |  1000 |  8000 |     4   (0)| 00:00:01 |
|*  4 |    HASH JOIN                   |               |   995 | 15920 |     7  (15)| 00:00:01 |
|   5 |     TABLE ACCESS BY INDEX ROWID| B             |   100 |   800 |     2   (0)| 00:00:01 |
|*  6 |      INDEX RANGE SCAN          | B_STATUS_IDX  |   100 |       |     1   (0)| 00:00:01 |
|*  7 |     TABLE ACCESS FULL          | A             |  1000 |  8000 |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("B"."ID"="C"."B_ID")
   3 - filter("C"."STATUS"='OPEN')
   4 - access("A"."STATUS"="B"."STATUS" AND "A"."B_ID"="B"."ID")
   6 - access("B"."STATUS"='OPEN')
   7 - filter("A"."STATUS"='OPEN')
```

L’optimiseur ORACLE : les statistiques des requêtes

- **Comment obtenir les statistiques des requêtes ?**
  - utilisation de l’outil Autotrace
  - permet de visualiser à la fois le résultat d’une requête, le plan d’exécution et les statistiques sur la requête

    => Attention si la requête prend du temps .... !
  - commande : « set autotrace on »
  - désactivation d’autotrace : « set autotrace off ».

```
Statistiques
----------------------------------------------------------
        196  recursive calls
          0  db block gets
         48  consistent gets
          0  physical reads
          0  redo size
       1073  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
         16  rows processed
```

L’optimiseur ORACLE : les statistiques des requêtes

- **Accès mémoire : nb de pages/blocs logiques lus**
  - consistent gets : nb d’accès à une donnée consistante en RAM (non modifiée)
  - db block gets : nb d’accès à une donnée en RAM
- **Accès physiques :**
  - physical reads : nb total de pages/blocs lues sur le disque
  - recursive calls : nb d’appels à un sous-plan (requête imbriquée, tris)
- **redo size : taille du fichier de log produit (écriture, mis à jour,...)**
- **sorts (disk) : nb d’opérations de tri avec au moins une écriture**
- **sorts (memory) : nb d’opérations de tri en mémoire**

Attention : statistiques sur les requêtes différentes des statistiques sur les tables ou le schéma de la BD (cardinalité, densité, sélectivité,...)