BASES DE DONNÉES

CM

Evaluation de requêtes

Vincent COUTURIER
Maitre de conférences – Université Savoie Mont-Blanc
vincent.couturier@univ-smb.fr

Traitement d'une requête

- 3 étapes :
  - **Analyse de la requête** : à partir d'une texte SQL, le SGBD vérifie la syntaxe et la validité de la requête : existences des tables ou vues et des champs. Résultat = plan d'exécution logique qui représente la requête sous la forme d'un arbre.
  - **Optimisation :** transformation du plan logique en un plan d'exécution physique comprenant les opérations d'accès aux données et les algos d'exécution placés dans un ordre optimal.
  - **Exécution de la requête :** compilation du plan d'exécution physique en un programme qui est ensuite exécuté.

Optimisation d'une requête

- L'optimisation consiste à trouver le plan qui minimisera le temps pris pour exécuter la séquence d'opérations.
- Principaux facteurs qui influent sur le temps d'exécution :
  - Mémoire centrale disponible.
  - Nombre d'accès en mémoire secondaire nécessaire.
  - Nombre d'utilisateurs accédant simultanément au système.

ÉVALUATION DE REQUETES

Evaluation de requêtes

- Le résultat de l'optimisation est un plan d'exécution, i.e. une séquence d'opérations à exécuter. On doit maintenant :
  - Appliquer les techniques d'accès appropriées pour chaque opération du plan d'exécution.
  - Gérer les flots de données entre chaque opération.
- Les algorithmes, techniques d'accès et heuristiques mis en œuvre relèvent de l'évaluation de requêtes.
- Pourquoi l’évaluation de requêtes ?
  - Pour limiter le nombre d'entrées/sorties (ces opérations sont les plus coûteuses !).

Exemple de référence

Vertigo 1958
Annie Hall 1977
Brazil 1984
Twin Peaks 1990
Jurassic Park 1992
Underground 1995
Metropolis 1926
Psychose 1960
Greystoke 1984
Shining 1979
Manhattan 1979
Easy Rider 1969

Films

Spielberg
Hitchcock
Allen
Lang
Hitchcock
Allen
Kubrik
Jurassic Park
Psychose
Manhattan
Metropolis
Vertigo
Annie Hall
Shining

Réalisateurs

ALGORITHMES DE BASE

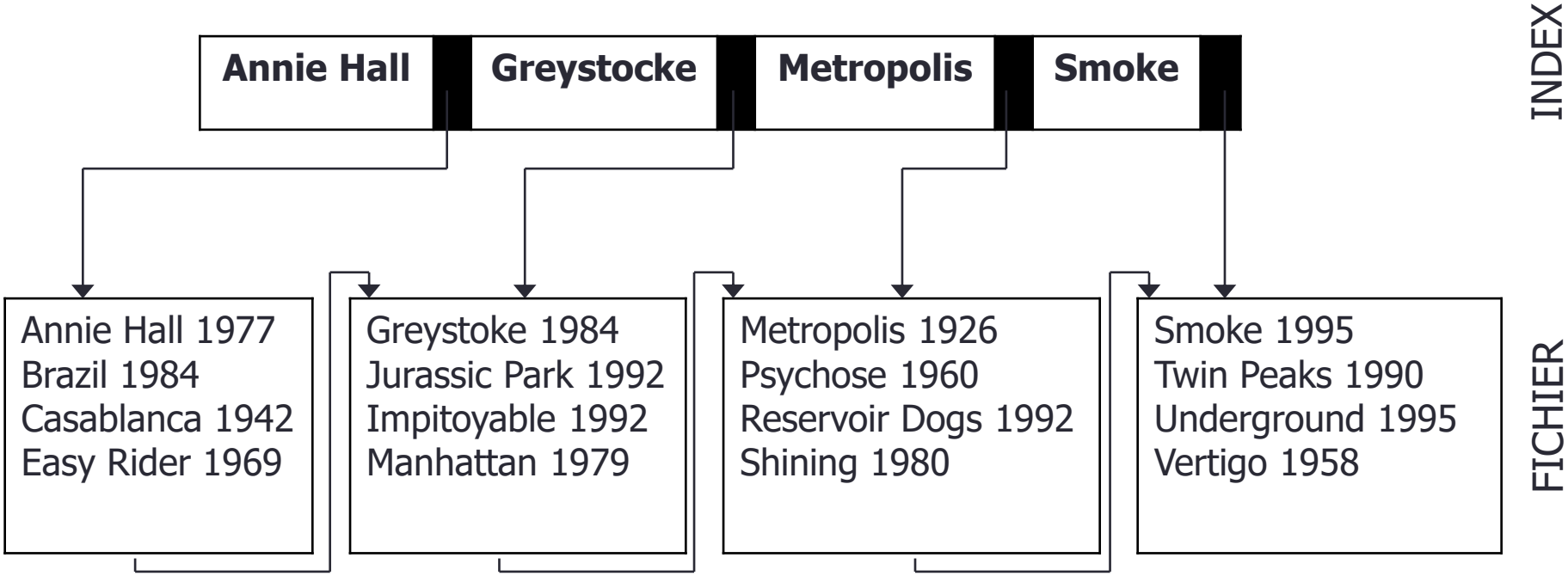
Introduction

- Principaux opérateurs utilisés dans l'évaluation d'une requête :
  - **Recherche dans un fichier (opération de sélection) :** par balayage ou par accès direct
  - **Tri :** utilisé lors d'une projection (select) ou une union et quand on désire éliminer les doublons (DISTINCT ou ORBER BY)
  - **Algorithmes de jointure,** dans l'ordre des performances décroissantes :
    - Si peu de lignes retournées **:**
      1. Jointure par boucle imbriquée
      2. Jointure par hachage
      3. Jointure par tri-fusion
    - Si beaucoup de lignes retournées **:**
      1. Jointure par hachage
      2. Jointure par boucle imbriquée
      3. Jointure par tri-fusion

Recherche dans un fichier (sélection)

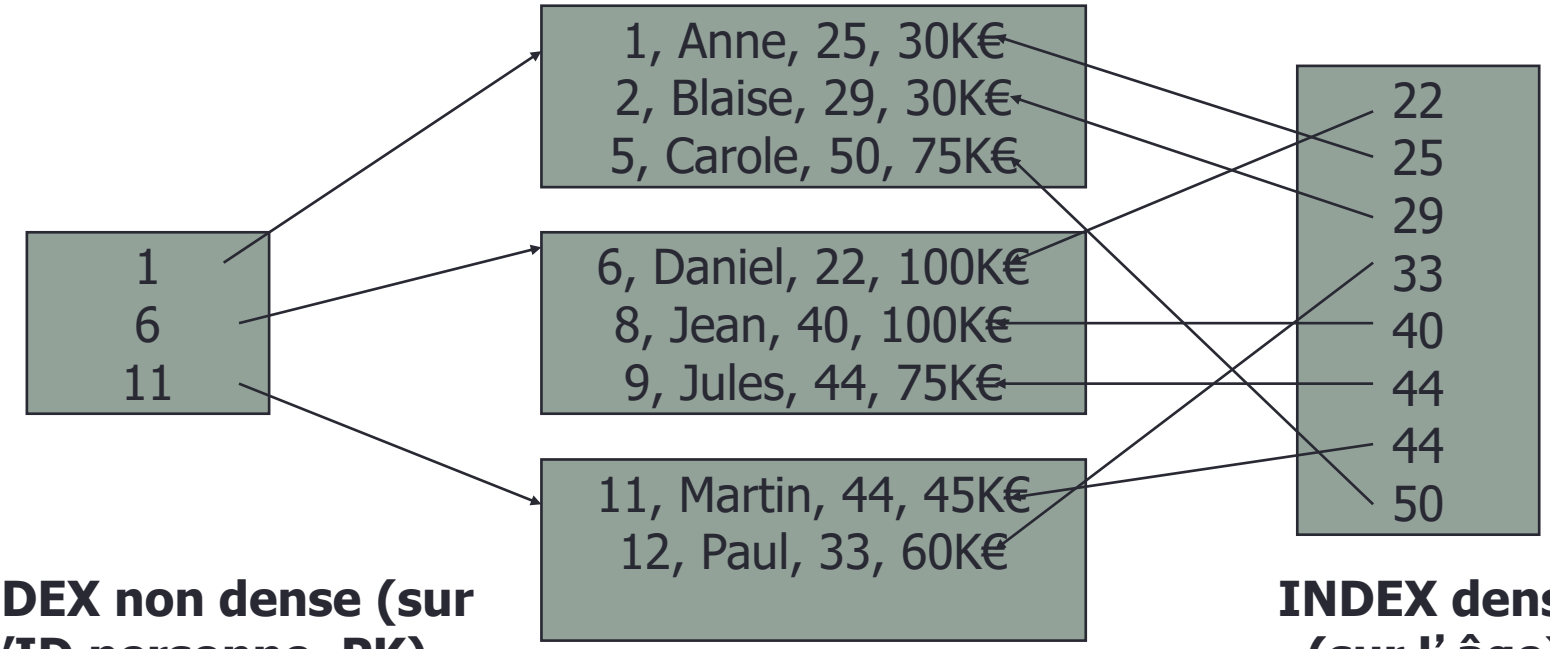
- 2 algos possibles pour effectuer une recherche dans un fichier (= une sélection dans une table) en fonction d'un critère de recherche :
  - **Accès séquentiel ou balayage.** Tous les enregistrements du fichier sont examinés lors de l'opération, en général suivant l'ordre dans lequel ils sont stockés.
  - **Accès par adresse.** Si on connaît l'adresse du ou des enregistrements concernés, on peut aller lire directement les blocs et ainsi obtenir un accès optimal.
    - Suppose qu'il existe un index ou une fonction de hachage permettant d'obtenir les adresses en fonction du critère de recherche.

Exemple fichier indexé



Nous avons ici un index non-dense sur le fichier des 16 films, la clé étant le titre du film. On suppose que chaque bloc du fichier de données contient 4 enregistrements, ce qui donne un minimum de 4 blocs. Il suffit alors de 4 paires [titre, Adr] pour indexer le fichier. Les titres utilisés sont ceux des premiers enregistrements de chaque bloc, soit respectivement Annie Hall, Greystoke, Metropolis et Smoke.

Index dense / non dense exemple



INDEX non dense (sur l'ID personne, PK)

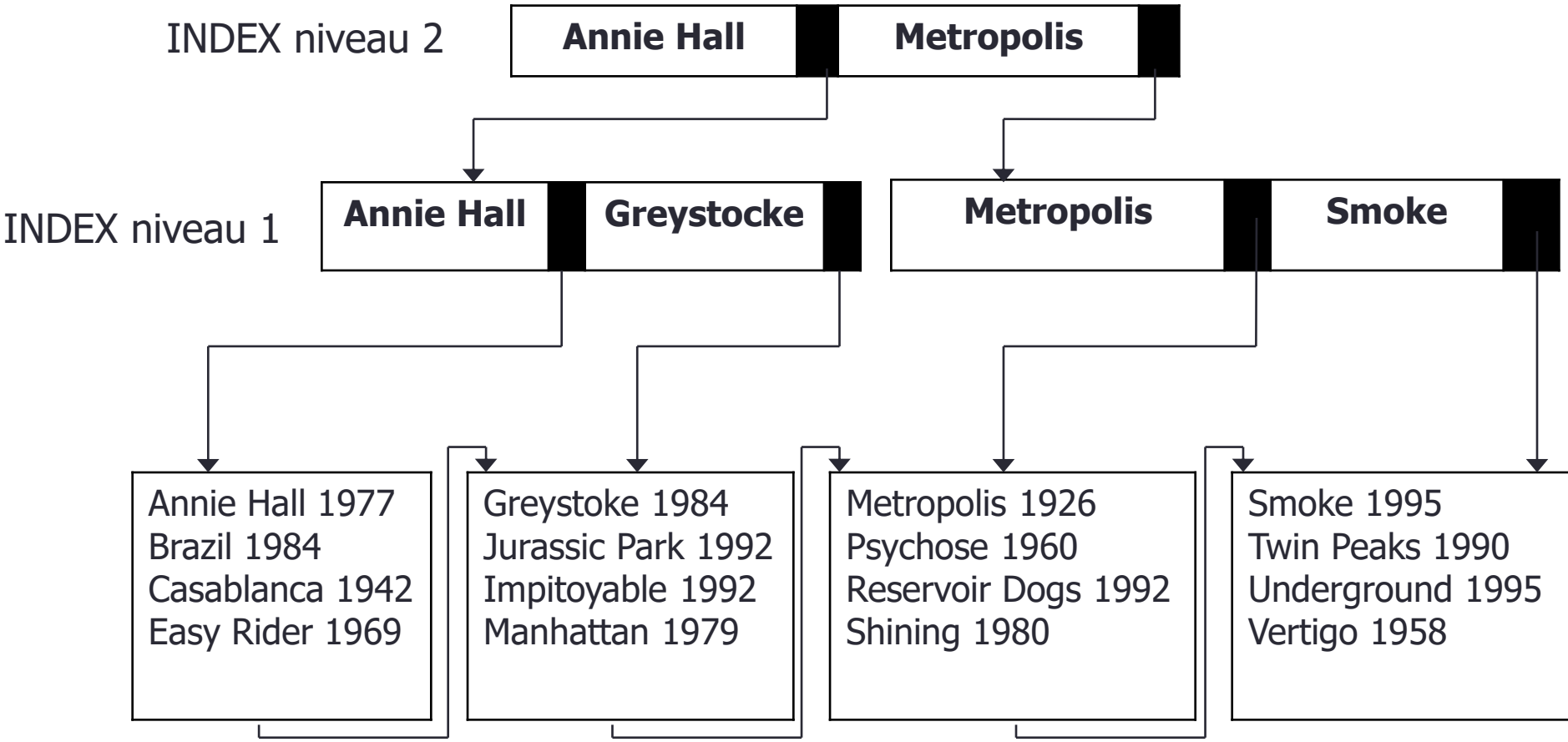
INDEX dense (sur l' âge)

⇒ Si on veut indexer un fichier qui n'est pas trié sur la clé de recherche, on ne peut tirer partie de l'ordre des enregistrements pour introduire seulement dans l'index la valeur de clé du premier élément de chaque bloc.

⇒ Il faut donc baser l' index sur **toutes** les valeurs de clé existant dans le fichier, et les associer à l'adresse d'un enregistrement, et pas à l'adresse d'un bloc => **index dense**. On constate que tous les âges du fichier de données sont reportés dans l'index et que le même âge (ex. 44) apparaît autant de fois qu'il y a de personnes ayant cet âge là.

⇒ Un index dense est généralement beaucoup plus volumineux qu'un index non dense

Index multi-niveaux



Quand l'index grossit, il s'organise en multi-niveaux.

Hachage

- Idée de base : organiser un ensemble d'éléments d'après une clé et utiliser une fonction (dite *de hachage*) qui, pour chaque valeur de clé *c*, donne l'adresse *H(c)* d'un espace de stockage où l'élément doit être placé (ex. bloc(s) sur le disque)
- Accès direct à la page contenant l'article recherché :
  - On estime le nombre *N* de pages qu'il faut allouer au fichier.
  - **Fonction de hachage *H :*** à toute valeur de la clé de domaine V, on associe un nombre entre 0 et *N* - 1.

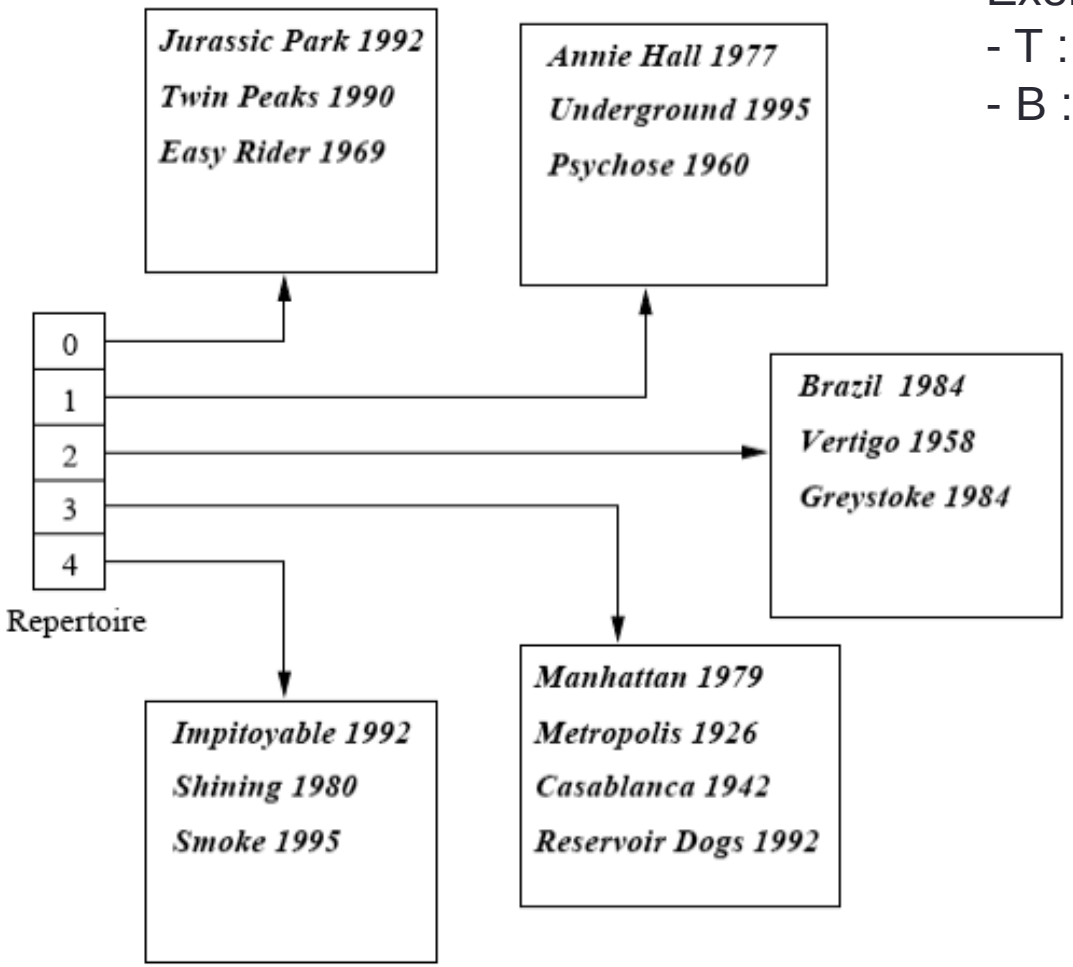
    $H : V \rightarrow \{0,1,...,N\text{-}1\}$
  - On range dans la page de numéro *i* tous les articles dont la clé *c* est telle que *H(c) = i.*

Exemple : hachage sur le fichier Films

- On suppose qu'une page contient 4 articles :
  - On alloue 5 pages au fichier.
  - On utilise une fonction de hachage *H* définie comme suit :
    - Clé : nom d'un film, on ne s'intéresse qu'à l'initiale de ce nom.
    - On numérote les lettres de l'alphabet de 1 à 26 :

      No('a') = 1, *No('m')* = 13, etc.
  - Si *L* est une lettre de l'alphabet, *H(L) = MODULO(No(L), 5).*

Exemple : hachage sur le fichier Films



Exemple :
- T : 20ème lettre => Modulo(20,5)=0
- B : 2ème lettre => Modulo(2,5)=2

Remarques

- Le nombre *H(c) = i* n'est pas une adresse de page, mais l'indice d'une table ou "répertoire" *R. R(i)* contient l'adresse de la page associée à *i*
- Si ce répertoire (ou table de hachage) ne tient pas en mémoire centrale, la recherche coûte plus cher.
- Une propriété essentielle de *H* est que la distribution des valeurs obtenues soit uniforme dans {0,...,*N* - 1}
- Quand on alloue un nombre *N* de pages, il est préférable de prévoir un remplissage partiel (non uniformité, grossissement du fichier). On a choisi 5 pages alors que 4 (16 articles / 4) auraient suffi.

Remarques

- Problème de distribution uniforme :
  - On est assuré avec la fonction précédente d'obtenir toujours un chiffre entre 0 et 4
  - En revanche la distribution risque de ne pas être uniforme :
    - Si, comme on peut s'y attendre, beaucoup de titres commencent par la lettre L (le, les, la), le bloc 2 risque d'être surchargé et l'espace initialement prévu s'avèrera insuffisant.
    - En pratique, les SGBD utilisent un calcul beaucoup moins sensible à ce genre de biais : on prend par exemple les 4 ou 8 premiers caractères de la chaînes, on traite ces caractères comme des entiers dont on effectue la somme et on définit la fonction sur le résultat de cette somme.

Hachage : recherche

- Etant donné une valeur de clé v :
  1. On calcule $i = H(v)$.
  2. On consulte dans la case *i* du répertoire l'adresse de la page *p.*
  3. On lit la page *p* et on y recherche l'article.

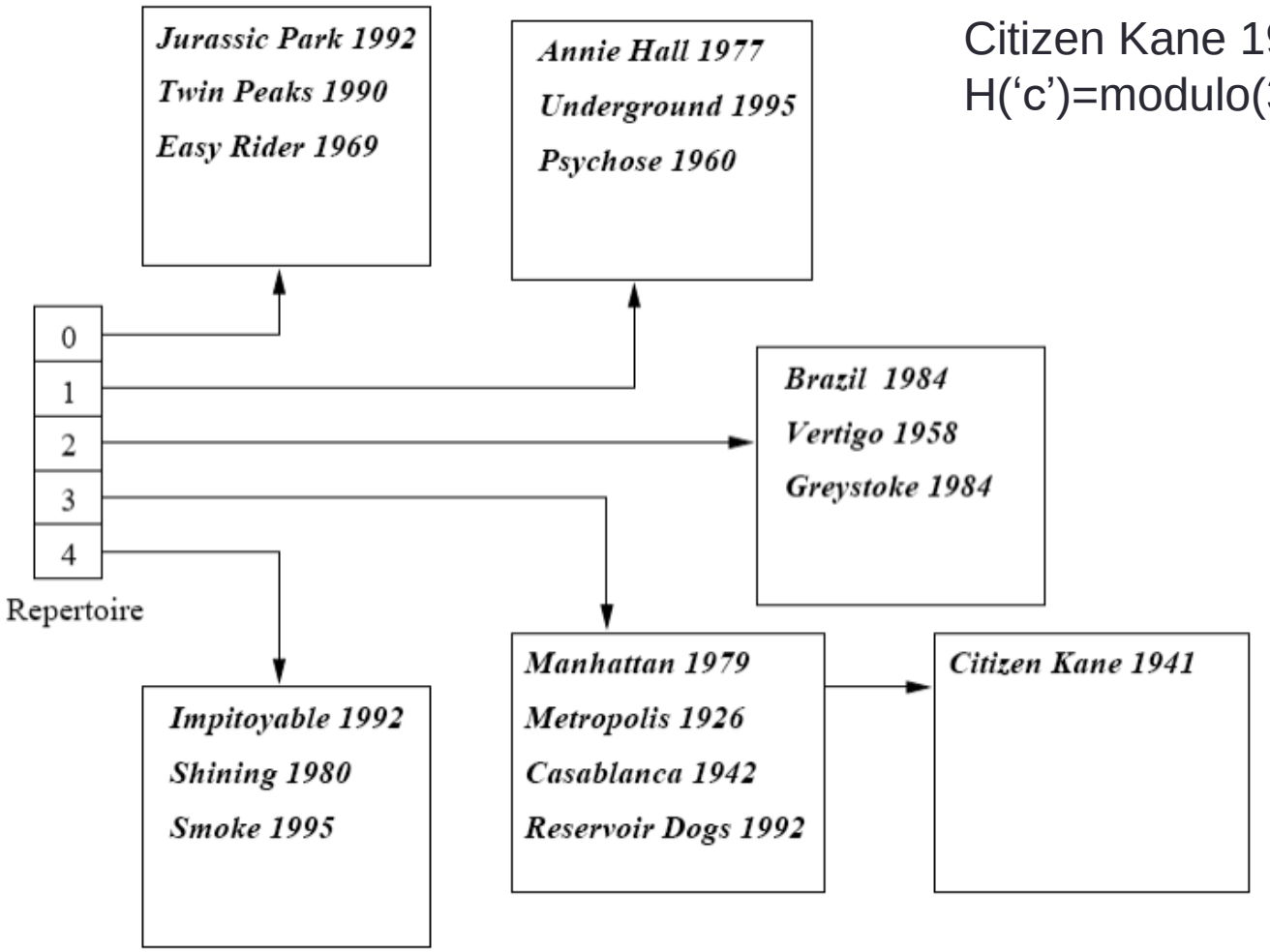
Donc une recherche ne coûte qu'une seule lecture.

Hachage : insertion

- Recherche par *H(c)* la page *p* où placer *A* et l'y insérer.
- Si la page *p* est pleine, il faut:
  - Allouer une nouvelle page *p'* (de débordement).
  - Chaîner *p'* à *p*.
  - Insérer *A* dans *p'*.
- => Lors d'une recherche, il faut donc en fait parcourir la liste des pages chaînées correspondant à une valeur de *H(v)*.

- **Moins la répartition est uniforme, plus il y aura de débordements et plus la structure dégénère vers un simple fichier séquentiel.**

Hachage : insertion



Insertion d'un nouveau film :
Citizen Kane 1941.
H('c')=modulo(3,5)=3

Hachage : avantages et inconvénients

- Intérêt du hachage :
  - **Très rapide.** Une seule E/S dans le meilleur des cas pour une recherche.
  - Le hachage, contrairement à un index, **n'occupe aucune place disque.**
- En revanche :
  - Il faut penser à réorganiser les fichiers qui évoluent beaucoup.
  - **Les recherches par intervalle sont impossibles.**

Recherche dans un fichier : balayage

- Dans le cas de la sélection, le balayage est utilisé soit parce qu'il n'y a pas d'index ou clé de hachage sur le critère de sélection, soit parce que la table est stockée sur un petit nombre de pages (petite table).
- Algo :
  - Lecture en séquence les pages de la relation, une à une, en mémoire centrale. Les enregistrements sont parcourus séquentiellement et traités au fur et à mesure.
  - Ce balayage séquentiel permet de faire en même temps que la sélection (`where`) une projection (`select`). Chaque enregistrement du tampon est testé en fonction du critère de sélection. S'il le satisfait, il est projeté et rangé dans un tampon de sortie.
- Coût :
  - B E/S, avec B = nbre de pages du fichier
  - Ce coût peut être diminué si regroupement des pages disques sur des espaces contigus (ex. : cluster sous Oracle ou SQL Server).

Recherche dans un fichier : accès direct

- Quand on connait l'adresse d'un enregistrement, donc l'adresse de la page où il est stocké, y accéder coûte une seule lecture de page.
- Avant, il faut cependant parcourir **l'index** ou **le répertoire lié à la clé de hachage (table de hachage)**

Recherche dans un fichier : comparaison des performances Accès direct / Balayage

- Exemple : fichier de 500 Mo, une lecture de bloc prend 0,01 s (10 millisecondes).
  - Un parcours séquentiel lit tout le fichier (ou la moitié pour une recherche par clé). Donc cela prend 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 = 0.04s (40 millisecondes).

Recherche dans un fichier : quand utiliser l'index ?

- Choix d'utiliser le parcours séquentiel ou l'accès par index: on regarde si un index est disponible, et si oui on l'utilise comme chemin d'accès.
- **En fait ce choix est rendu plus compliqué par les considérations suivantes :**
  - Le critère de recherche porte-t-il sur un ou sur plusieurs attributs ? S'il y a plusieurs attributs, les critères sont-ils combinés par des **and** ou des **or** (Cas 1) ?
  - **Quelle est la sélectivité de la recherche ?** Quand une partie significative de la table est sélectionnée, il devient inutile, voire contre-performant, d'utiliser un index (Cas 2)

Recherche dans un fichier : quand utiliser l'index ? CAS 1

- Exemples (index unique sur idFilm. Pas d'index sur titre) :

```
SELECT * FROM Film
WHERE idFilm = 20
AND titre = 'Vertigo'
```

  - => on utilise l'index pour accéder à l'unique film (s'il existe) ayant l'identifiant 20. Puis, une fois l'enregistrement en mémoire, on vérifie que son titre est bien Vertigo.

```
SELECT * FROM Film
WHERE idFilm = 20
OR titre = 'Vertigo'
```

  - => on peut utiliser l'index pour trouver le film 20, mais il faut un parcours séquentiel pour rechercher Vertigo => balayage.

Recherche dans un fichier : quand utiliser l'index ? CAS 2

- Sélectivité d'un attribut A d'une table R = rapport entre le nombre d'enregistrements pour lesquels A a une valeur donnée et le nombre total d'enregistrements.
  - Si une clé est unique (clé primaire), nbvals= |R| => sélectivité = 1/R
  - Si 2 valeurs possibles OUI ou NON, nbvals=2 => sélectivité = 1/2 => attribut peu sélectif => pas d'utilisation de l'index **en théorie. En réalité** si 99% de NON et 1% de OUI, le SGBD utilisera l'index pour une recherche sur les OUI et le balayage pour une recherche sur les NON.
- Les informations sur la sélectivité sont fournies par les statistiques.
  - On parle **d'optimisation dynamique**

Règle du DBA : 40% du volume d'une base de données doit être indexée, mais c'est au développeur de créer ses index…

Statistiques

- Récupération de diverses informations concernant la volumétrie des tables, la distribution des différentes valeurs des champs indexés, la taille moyenne des tuples, etc.
- Cet ensemble d'informations génèrera via un algorithme propre au SGBD un coût pour chaque plan d'exécution.
- PostgreSQL : outil ANALYZE permet de calculer/mettre à jour les statistiques
  - Penser à le lancer régulièrement
- Oracle :
  - Mise en jour en continue des stats

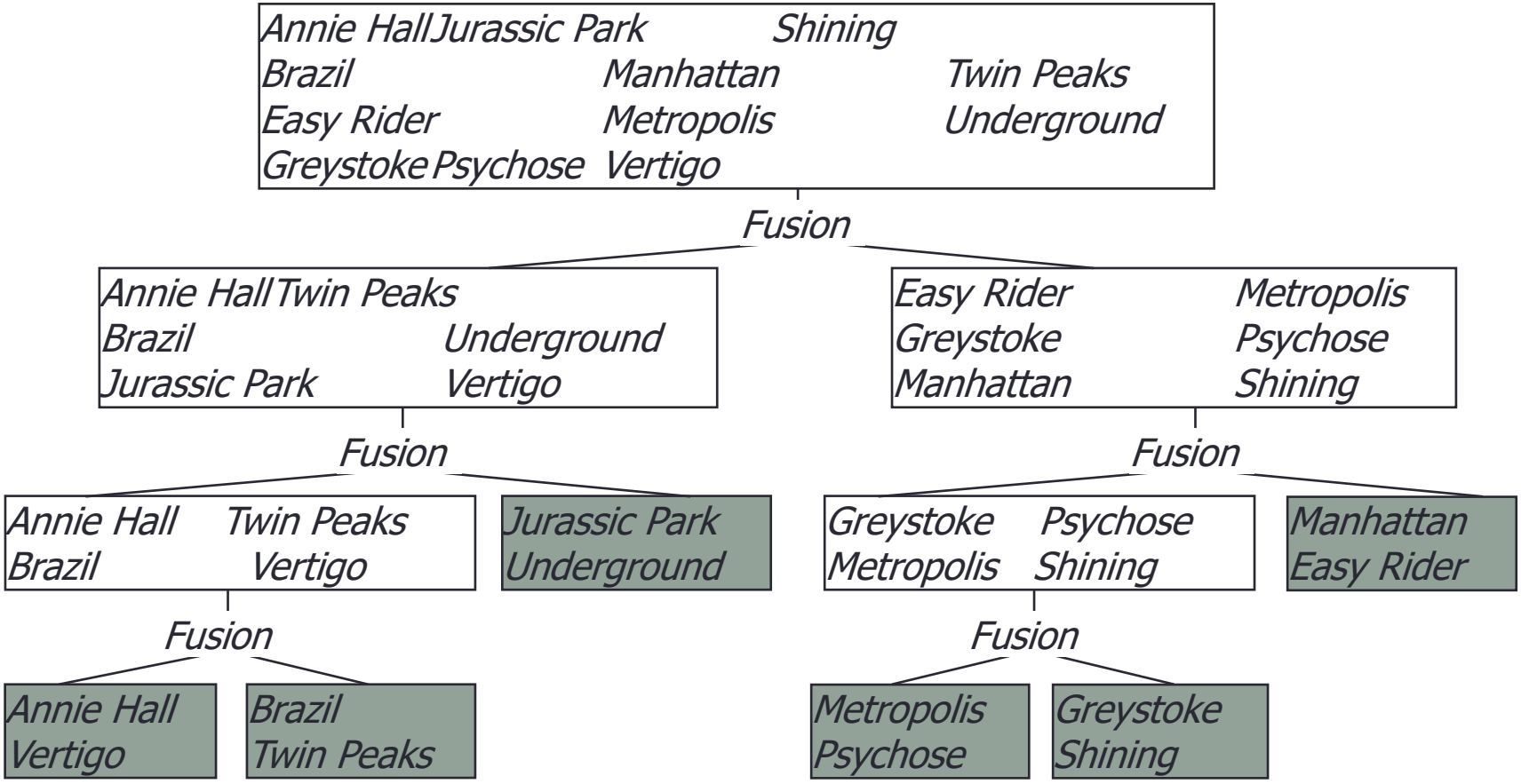
Tri

- Le tri est une opération fréquente. On l'utilise :
  - Pour afficher des données ordonnées (clause `ORDER BY`).
  - Pour éliminer des doublons ou faire des agrégats.
  - Pour certains algorithmes de jointure (tri-fusion).
- Sans index ou hachage, l'algorithme de jointure utilisé dans les SGBD est le **tri - fusion** *(`SORT & MERGE`).*

Algorithmes de jointure : tri-fusion

- On applique la stratégie dite "diviser pour régner". Elle consiste à :
  - **Découper** la table en partitions telles que chaque partition tienne en mémoire centrale + Tri de chaque partition en mémoire.
  - **Fusionner** les partitions triées
- PostgreSQL : MERGE JOIN
- **ATTENTION : c'est l'algorithme de jointure le moins performant !**

Algorithmes de jointure : exemple de tri-fusion



⇒ ensemble de films trié sur le nom du film.

⇒ 3 phases de fusion, à partir de 6 fragments initiaux que l'on regroupe 2 à 2.

Algorithmes de jointure : exemple de coût du tri-fusion

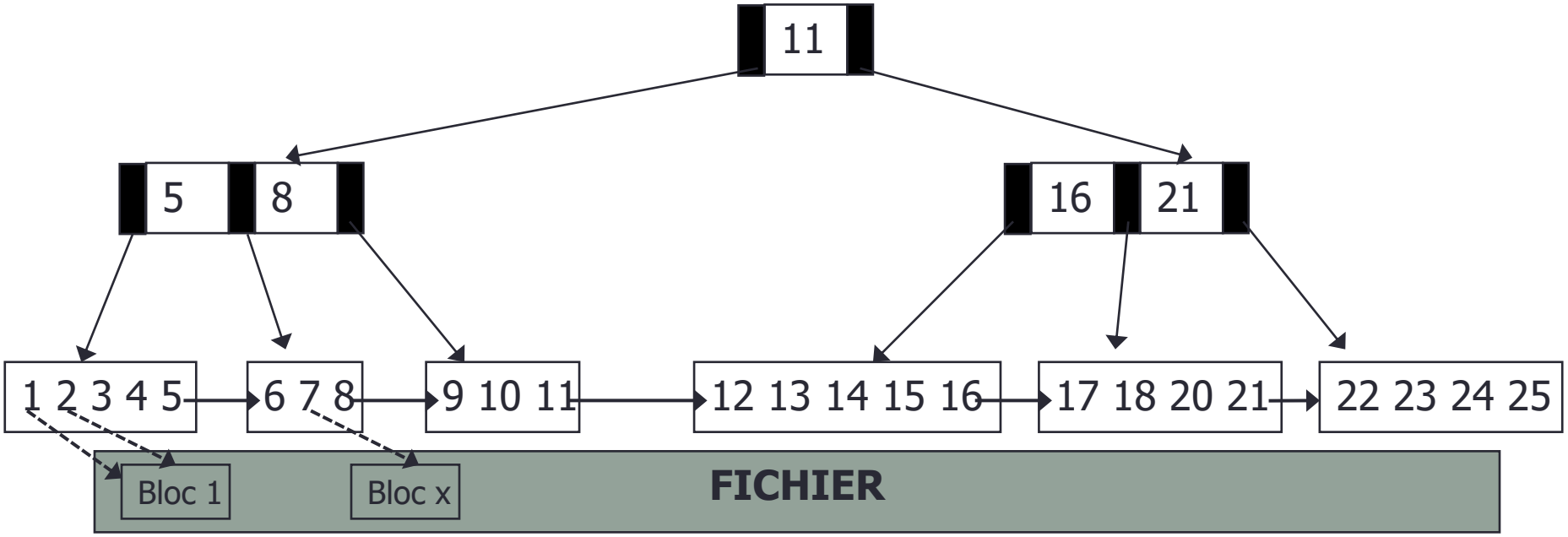
- Tri d'un fichier de 75 000 pages de 4 Ko, soit 307 Mo.
  - Avec une mémoire M>307 Mo, tout le fichier peut être chargé et trié en mémoire. Une seule lecture suffit.
  - Avec une mémoire **M=2Mo**, soit 500 pages.
    - On divise le fichier en $\lceil 307/2 \rceil$=154 fragments. Chaque fragment est trié en mémoire et stocké sur le disque. On a lu et écrit une fois le fichier en entier, soit 714 Mo.
    - On associe chaque fragment à une page en mémoire, et on effectue la fusion (noter qu'il reste 500-154 pages disponibles). On a lu encore une fois le fichier, pour un coût total de 714 + 307=**1021 Mo**
  - Avec une mémoire **M=1Mo**, soit 250 pages.
    - On divise le fichier en $\lceil 307/1 \rceil$=307 fragments. Chaque fragment est trié en mémoire et stocké sur le disque. On a lu et écrit une fois le fichier en entier, soit 714 Mo.
    - On associe les 249 premiers fragments à une page en mémoire et on effectue la fusion (on garde la dernière page pour la sortie). On obtient un nouveau fragment de taille 249 Mo. On prend les 307-249=58 fragments restant et on les fusionne : on obtient F2 de taille 58 Mo. On a lu et écrit encore une fois le fichier, pour un coût total de 714*2=1428 Mo
    - Finalement, on prend les 2 derniers fragments F1 et F2 et on les fusionne. Cela représente une lecture de plus soit 1428 + 307 = **1735 Mo**.

Algorithmes de jointure : boucle imbriquée (NESTED LOOP)

- Utilise 2 requêtes imbriquées : la requête externe pour récupérer les résultats d'une table et la deuxième requête *pour chaque ligne* provenant de la première requête, pour récupérer les données correspondantes dans l'autre table.
- Nécessite la présence d'au moins 1 index
- Simple boucle imbriquée :
  - On utilise l'index d'un des champs figurant dans la jointure => 1 seule requête passe par l'index
  - Souvent la PK (index unique créé par le SGBD)
- Double boucle imbriquée :
  - On utilise les index des 2 champs de la jointure => chaque requête est exécutée en passant par l'index
  - Si jointure PK = FK, nécessite d'indexer la FK (si elle ne l'est pas déjà)
- **Algorithme le plus performant si peu de lignes retournées par la requête (d'autant plus vrai si double boucle imbriquée).**

Algorithmes de jointure : boucle imbriquée (NESTED LOOP)

- Une boucle imbriquée fonctionne généralement sur des index de type B-tree (arbre B) :



- Autres index possibles : Index bitmap (Cf. TD), etc.

Algorithmes de jointure : jointure par hachage (HASH JOIN)

- Vise le point faible de la jointure par boucle imbriquée si de nombreuses lignes sont retournées par la requête : les nombreux parcours d'index lors de l'exécution de la requête interne
- À la place, il charge les enregistrements candidats d'un côté de la jointure dans une table de hachage (marqué avec le mot Hash dans le plan) qui peut être sondée très rapidement pour chaque ligne provenant de l'autre côté de la jointure.
- **Algorithme le plus performant si beaucoup de lignes sont retournées par la requête.**

HASH JOIN vs. NESTED LOOP

- HASH JOIN est utilisé quand :
  - il n'y a pas de **restriction** (pas de clause WHERE)
  - OU si la restriction est très limitée (beaucoup de lignes renvoyées)
  - OU **si le (ou les) champ(s) présent(s) dans la restriction (WHERE) n'est (ne sont) pas indexé(s).**
- NESTED LOOP est privilégié quand il y a une restriction (retournant moins de 20 à 30% des lignes renvoyées) et que cette restriction utilise un ou plusieurs index.
  - Si **restriction**, penser à indexer les champs car NESTED LOOP est plus performant que HASH JOIN.

OPTIMISATION DE REQUÊTES DANS **POSTGRESQL**

Opérateurs utilisés pour les chemins d'accès et jointures

- **Parcours :**
  - Séquentiel : `SEQ SCAN`.
  - Adresse/index : `INDEX SCAN`
  - Adresse/hachage : `HASH SCAN`

- **Jointure :**
  - Boucle imbriquée : `NESTED LOOP`
  - Par hachage : `HASH JOIN`
  - Tri fusion : `SORT/MERGE`

Autres opérations physiques

- Autres (principaux) opérateurs physiques :
  - INTERSECTION : intersection de 2 ensembles de n-uplets. `INTERSECT` dans PostgreSQL.
  - CONCATENATION : union de deux ensembles. Dans PostgreSQL : `OR`.
  - FILTRAGE : élimination de n-uplets (restriction). Dans PostgreSQL : `FILTER` ou `INDEX COND` (si index) selon les cas.
  - PROJECTION : affichage des colonnes (contrairement à Oracle, PostgreSQL n'affiche pas l'opération `SELECT`)
  - AGREGATION : si fonction d'agrégation (`SUM`, `COUNT`, etc.). Opérateurs `AGGREGATE, GROUP` dans PostgreSQL.
  - TRI : `SORT`
  - Etc.

Types d'index PostgreSQL

- Quand vous exécutez la commande `CREATE INDEX`, PostgreSQL choisit le type d'index à créer en fonction du type de colonne :
  - `B-tree` (arbre équilibré) : type d'index créé par défaut
    - Pour l'organisation de l'index Cf. slide 37
    - Excellent en écriture (réorganisation), très bon en lecture.
  - `BITMAP` : utilisé normalement dans les datawarehouse
    - Faible coût de stockage
    - Excellent en lecture, par contre mauvais en écriture (réorganisation)
  - Index FULLTEXT :
    - `GIN` : utilisé par le type `JSONB` (type utilisé pour stocker un Json format Json binaire)
    - `GiST` : pour les types de données UDT
    - https://docs.postgresql.fr/14/textsearch-indexes.html
  - Autres types : `BRIN, SP-GiST`
  - https://docs.postgresql.fr/14/indexes.html

  - PostgreSQL ne gère que des index denses

L’outil EXPLAIN

- Donne le plan d’exécution d’une requête.
- La description comprend :
  - Le chemin d'accès utilisé.
  - Les opérations physiques (jointure, intersection, filtrage,...).
  - L'ordre des opérations. Il est représentable par un arbre.

L'outil EXPLAIN : exemple

- Schéma de la base :

```
CINEMA (ID_cinema*, Nom, Rue, CP, Ville);
SALLE (ID_salle*, Nom, Capacite+, ID_cinema+);
FILM (ID_film*, Titre, Annee, ID_realisateur+);
-- ID_realisteur pointe vers Id_artiste de ARTISTE
SEANCE (ID_seance*, Heure_debut, Heure_fin, ID_salle+,
ID_film);
ARTISTE(ID_artiste*, Nom, Prenom, Annee_naissance);
```

Attributs avec une * : index unique.

Attributs avec une +: index non unique.

L'outil EXPLAIN : exemple

- Schéma de la base :

```
CREATE UNIQUE INDEX IX_CINEMA_IDCINEMA ON cinema (id_cinema); -- PK
CREATE UNIQUE INDEX IX_SALLE_IDSALLE ON salle (id_salle); -- PK
CREATE INDEX IX_SALLE_CAPACITE ON salle (capacite);
CREATE INDEX IX_SALLE_iDCINEMA ON salle (id_cinema); -- FK
CREATE UNIQUE INDEX IX_FILM_IDFILM ON film (id_film); -- PK
CREATE INDEX IX_FILM_IDREALISATEUR ON film (id_realisateur); -- FK
CREATE UNIQUE INDEX IX_SEANCE_IDSEANCE ON seance (id_seance); -- PK
CREATE INDEX IX_SEANCE_IDSALLE ON seance (id_salle); -- FK
CREATE UNIQUE INDEX IX_ARTISTE_IDARTISTE ON artiste (id_artiste); -- PK
```

Remarque : ici, nous n'avons que des index mono-colonnes et non multi-colonnes. Par exemple, si on fait souvent une recherche sur le nom et/ou le prénom, il peut être intéressant de créer un index portant sur ces 2 colonnes, ***à condition que le SGBD gère correctement ce type d'index…***

L'outil EXPLAIN : exemple

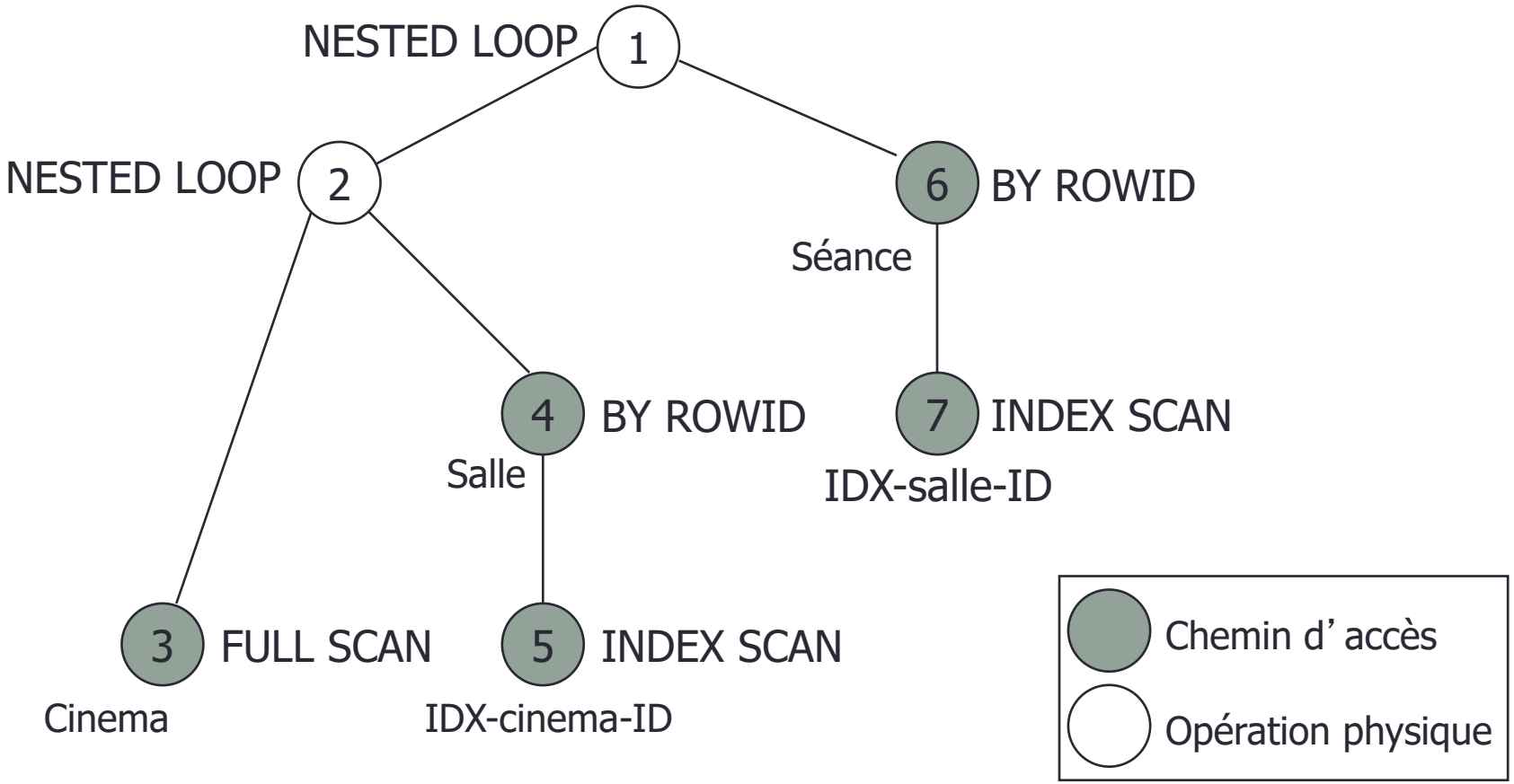
- Quels films passent aux Rex à 20 heures ?

```
EXPLAIN SELECT Se.ID_film
        FROM Cinema C
               JOIN Salle Sa ON C.ID_cinema = Sa.ID_cinema
               JOIN Seance Se ON Sa.ID_salle = Se.ID_salle
        WHERE  C.nom = 'Rex'
               AND Se.heure_debut = 20.00
```

- Plan d'exécution donné par EXPLAIN (Oracle) :

```
0 SELECT STATEMENT
  1 NESTED LOOP
    2 NESTED LOOP
      3 TABLE ACCESS FULL CINEMA
      4 TABLE ACCESS BY ROWID SALLE
        5 INDEX RANGE SCAN IDX-CINEMA-ID
    6 TABLE ACCESS BY ROWID SEANCE
      7 INDEX RANGE SCAN IDX-SALLE-ID
```

L'outil EXPLAIN : exemple



L'outil EXPLAIN : exemple

- Plan d'exécution obtenu dans PostgreSQL :

```
1  Nested Loop  (cost=1.51..6.06 rows=1 width=10)
2    ->  Hash Join  (cost=1.51..2.70 rows=1 width=19)
3          Hash Cond: (sa.id_salle = se.id_salle)
4          ->  Seq Scan on salle sa  (cost=0.00..1.13 rows=13 width=18)
5          ->  Hash  (cost=1.50..1.50 rows=1 width=19)
6                ->  Seq Scan on seance se  (cost=0.00..1.50 rows=1 width=19)
7                      Filter: (heure_debut = 20.00)
8    ->  Index Scan using cinema_pkey on cinema c  (cost=0.00..3.35 rows=1 width=9)
9          Index Cond: (c.id_cinema = sa.id_cinema)
10         Filter: ((c.nom)::text = 'Rex'::text)
```

Coût mini : 1.51 (coût de traitement de la requête si les tables sont vides ⇔ coût CPU). En général l'utilisation d'un HASH ou d'un INDEX augmente ce coût CPU.
Coût max : 6.06 (coût réel de la requête, fonction du nombre de lignes de la table : plus le nombre de ligne va augmenter plus le coût max va croître)

L'outil EXPLAIN : exemple

- Restriction sur un champ non indexé :

```
SELECT *
FROM cinema
WHERE nom = 'Rex'
```

  - Plan d'exécution :
    - Oracle :

```
0 SELECT STATEMENT
 1 TABLE ACCESS FULL CINEMA
```

    - PostgreSQL :

1	Seq Scan on cinema (cost=0.00..16.50 rows=3 width=125)
2	Filter: ((nom)::text = 'Rex'::text)

L'outil EXPLAIN : exemple

- Restriction sur un champ indexé :

```
SELECT *
FROM cinema
WHERE id_cinema = 2
```

  - Plan d'exécution :
    - Oracle :

```
0 SELECT STATEMENT
  1 TABLE ACCESS BY ROWID CINEMA
        2 INDEX UNIQUE SCAN CINEMA_PKEY
```

    - PostgreSQL :

1	Index Scan using cinema_pkey on cinema (cost=0.00..8.27 rows=1 width=125)
2	Index Cond: (id_cinema = 2::numeric)

L'outil EXPLAIN : exemple

- Restriction conjonctive avec 1 (seul) index :

```
SELECT adresse FROM Cinema
WHERE Id_cinema =1 AND nom = 'Rex'
```

  - Plan d'exécution :
    - Oracle :

```
0 SELECT STATEMENT
      1 TABLE ACCESS BY ROWID CINEMA
        2 INDEX UNIQUE SCAN CINEMA_PKEY
```

    - PostgreSQL :

1	Index Scan using cinema_pkey on cinema (cost=0.00..8.27 rows=1 width=78)
2	Index Cond: (id_cinema = 1::numeric)
3	Filter: ((nom)::text = 'Rex'::text)

L'outil EXPLAIN : exemple

- Restriction disjonctive avec 1 (seul) index :

```
SELECT adresse FROM Cinema
WHERE Id_cinema =1 OR nom = 'Rex'
```

  - Plan d'exécution :
    - Oracle :

```
0 SELECT STATEMENT
    1 TABLE ACCESS FULL CINEMA
```

    - PostgreSQL :

1	Seq Scan on cinema (cost=0.00..17.80 rows=4 width=78)
2	Filter: ((id_cinema = 1::numeric) OR ((nom)::text = 'Rex'::text))