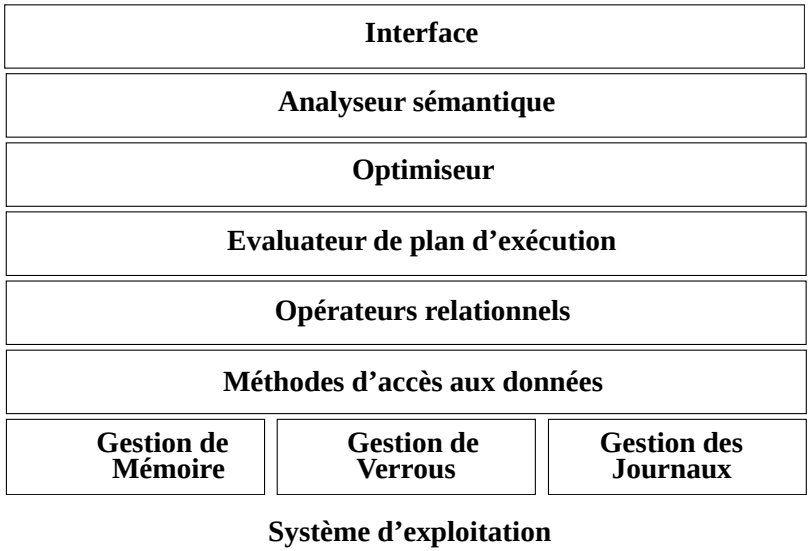
Opérateurs relationnels

1. Gestion de la mémoire
2. Algorithmes de sélection
3. Algorithmes de jointure
4. Autres opérateurs

Architecture en couche d'un SGBD



Quel algorithme pour quel contexte ?

- Le choix de la bonne implémentation d'un opérateur relationnel (select, project, join, union, etc.) dépend
  - de la requête et de sa sélectivité
  - des caractéristiques des inputs (tables en entrée)
  - de ses paramètres (prédicat de jointure, de sélection, etc.)
  - des structures d'indexation existantes
  - et de la quantité de mémoire disponible
- Quels sont les différents choix ? ➔ Objet de ce cours
- Comment choisir ? ➔ Problématique d'optimisation

Les opérateurs

- Sélection (σ), sélection d'un sous ensemble des tuples d'une table
  - Par Scan, par index
- Projection (π), sélection d'un sous ensemble d'attributs d'une table
  - Par Scan, par index, par tri ou hachage (si élimination des doublons)
- Jointure (∞), combinaison de deux tables sur critère
  - Par boucles imbriquées
  - Par index
  - Par tri
  - Par hachage
- Autres opérateurs
  - Groupement, agrégats
  - Différence (-), Union ($\cup$), Intersection ($\cap$)
- Chaque opérateur prend en entrée une ou deux tables et renvoie une table
  ➔ possibilité de composer les opérateurs pour former des plans (arbres) d'exécution de requêtes

Sélection σ: par Scan

- Algorithme Select (R, Q) :

```
For each page p de R do {
    Read (p);
    For each tuple t de p
        { if  Check (t, Q) then result = result ∪ t ;} ;
}
```

- La solution la plus simple et la meilleure quand la sélectivité est très faible (ex: chaque page contient au moins un tuple qualifié)
  - $||\text{Result}|| = S^* ||R||$, avec $0 \leq S \leq 1$
  - ATTENTION: Sélectivité faible = S élevé et vice-versa

Sélection σ : utilisation d'un index

- Quand utiliser l'index ?
  - Si la sélectivité est forte (S très petit)
  - car coût(I/O aléatoire) >> coût(I/O séquentielle)
- Ex.

```
SQL> SELECT *
   > FROM Medicament
   > WHERE Nom = 'N%';
```

Comparer les coûts d'accès:
1) Sans index;
2) Avec index non plaçant sur Nom;
3) Avec index plaçant sur Nom

  - Médicaments = 100.000 tuples stockés dans 1000 pages contiguës;
  - Noms uniformément distribués;
  - Il y a 5 % de médicaments commençant par un N (='*N%*')
  - I/O aléatoire = 10ms ; I/O séquentielle = 0,2ms
- NB : optimisation des accès aux données par index non plaçant
  - Trouver dans l'index les *Rid* qualifiés
  - Trier les *Rid qualifiés*
  - Accéder les tuples sur disque dans l'ordre des Rid (pour ne pas accéder 2 fois la même page…)

Sélection σ : utilisation de plusieurs index

- Identification des index les plus sélectifs
- Intersection et union de listes de Rid de ces index
  - Accès aux tuples dont les identifiants sont retenus
  - Vérification du reste du critère sur les tuples résultat
  - Exemple :
    - (SPEC='Dentiste' OR SPEC='Stomatologie') AND (ADRESSE = 'Versailles') AND (AGE>30)
    - L = (S1 ∪ S2) ∩ A1
    - Accéder aux tuples de L et vérifier le critère AGE
- Les index bitmap permettent de combiner efficacement des critères peu sélectifs

Projection

- Habituellement intégrée à la production du résultat du dernier opérateur du plan (➔ coût nul)
- L'opération coûteuse est l'élimination des doublons
  - SQL n'élimine pas les doublons (sauf si la clause DISTINCT est spécifiée dans la requête)

```
SQL> SELECT DISTINCT V.id, V.doc
   > FROM Visite V ;
```

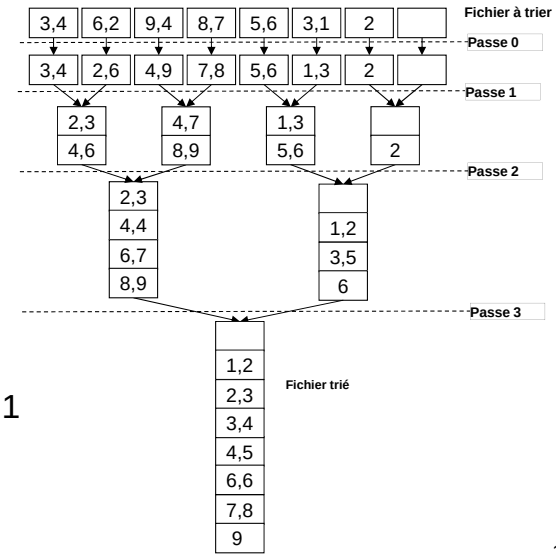
- Algorithme
  - Basé sur du tri ou du hachage
- Remarques sur **l'approche tri**
  - Le résultat final est de fait trié
  - C'est l'implantation standard choisie par les éditeurs de SGBD

Coût du Tri

- Comment trier une table qui ne tient pas en mémoire ?
  - Problème récurrent en base de données
  - Solution = algorithme de tri-fusion
    - La table est triée par morceaux (monotonies)
    - Les monotonies sont fusionnées entre elles
- Nombre de passes = $1+\lceil \log_{M-1} \lceil N / M \rceil \rceil$ (**N = Nb de pages à trier, M = Nb de pages mémoire)**
- Coût I/O = 2N x (nombre de passes)
  - Lecture et écriture (complète) des pages de la table à chaque passe
- Ex. 5 pages en RAM, pour trier 108 pages
  - Passe 0: $\lceil 108 / 5 \rceil$ = 22 monotonies triées de 5 pages chacune (sauf la dernière qui contient 3 pages)
  - Passe 1: $\lceil 22 / 4 \rceil$ = 6 monotonies triées de 20 pages chacune (resp. 8 pages)
  - Passe 2: 2 monotonies triées de 80 pages (et 28 pages)
  - Passe 3: fichier trié de 108 pages
  - Coût total = 2*108*4 = 864 I/O

Coût du Tri : exemple en 2 phases

- A chaque passe : lecture et écriture du fichier complet
  - Soit 2 *N* I/Os



- Fichier de N pages
  - Nb de passes = $\lceil \log_2 N \rceil + 1$
  - I/Os = 2 *N* ($\lceil \log_2 N \rceil + 1$)

Jointure

- L'opérateur le plus étudié en BD car le plus coûteux
- Variations de l'algorithme de jointure
  - Par boucle
    - Jointure par boucles imbriquées ((*Block-) Nested Loop Join*)
  - Par index
    - Jointure par index (*Index Join*)
  - Par tri
    - Jointure par tri fusion (*Sort Merge Join*)
  - Par hachage
    - Jointure par hachage (*Hash Join*)
    - Jointure par hachage de Grace (*Grace Hash Join*)
    - Jointure par hachage hybride (*Hybrid Hash Join*)
- NB : ces d'algorithmes peuvent êtres adaptés à d'autres opérateurs binaires (intersection, différence, etc.)

Comparaison des algorithmes de jointure

- On veut joindre les tables DOCteur et VISite

DOC

id	*Nom*	*spécialité*	……
1	5	Pédiatre	……
2	2	Radiologue	……
3	1	Pneumologue	……
…	…	…	……

VIS

id	*docid*	*date*	*prix*	……
1	3	……	……	……
2	2	……	……	……
3	2	……	……	……
4	3	……	……	……
5	1	……	……	……
6	1	……	……	……
7	3	……	……	……
…	…	……	……	……

- Coût = Nombre d'I/O
  - *on ne distingue pas I/O séquentielles et aléatoires*
  - *on ne compte pas le coût d'écriture du résultat final (identique pour tous)*
- Paramètres
  - |DOC| = Nombre de pages occupées par DOC
  - ||DOC|| = Nombre de tuples dans DOC
  - |DOC| << |VIS|
  - M = Nombre de pages mémoire disponibles

  (NB : quelques approximations pour simplifier les formules)

Jointure Brute-Force : Nested Loop

- Egalement appelée jointure par Produit Cartésien
- Algorithme Join(DOC, VIS, Q)

```
For each page p de DOC
  For each page q de VIS
    For each tuple tp de p
      For each tuple tq de q
          if  Check (tp, tq, Q) then result = result ∪ (tp||tq) ;
```

- Coût I/O = |DOC| + |DOC| * |VIS|
- Fonctionne quel que soit Q et quel que soit M ≥ 3
- Très sous-optimal dès que M > 3

Jointure

- L'opérateur le plus étudié en BD car le plus coûteux
- Variations de l'algorithme de jointure
  - Par boucle
    - Jointure par boucles imbriquées ((*Block-) Nested Loop Join*)
  - Par index
    - Jointure par index (*Index Join*)
  - Par tri
    - Jointure par tri fusion (*Sort Merge Join*)
  - Par hachage
    - Jointure par hachage (*Hash Join*)
    - Jointure par hachage de Grace (*Grace Hash Join*)
    - Jointure par hachage hybride (*Hybrid Hash Join*)
- NB : ces d'algorithmes peuvent êtres adaptés à d'autres opérateurs binaires (intersection, différence, etc.)

Comparaison des algorithmes de jointure

- On veut joindre les tables DOCteur et VISite

DOC

id	*Nom*	*spécialité*	……
1	5	Pédiatre	……
2	2	Radiologue	……
3	1	Pneumologue	……
…	…	…	……

VIS

id	*docid*	*date*	*prix*	……
1	3	……	……	……
2	2	……	……	……
3	2	……	……	……
4	3	……	……	……
5	1	……	……	……
6	1	……	……	……
7	3	……	……	……
…	…	……	……	……

- Coût = Nombre d'I/O
  - *- on ne distingue pas I/O séquentielles et aléatoires*
  - *- on ne compte pas le coût d'écriture du résultat final (identique pour tous)*
- Paramètres
  - |DOC| = Nombre de pages occupées par DOC
  - ||DOC|| = Nombre de tuples dans DOC
  - |DOC| << |VIS|
  - M = Nombre de pages mémoire disponibles

  (NB : quelques approximations pour simplifier les formules)

Jointure Brute-Force : Nested Loop

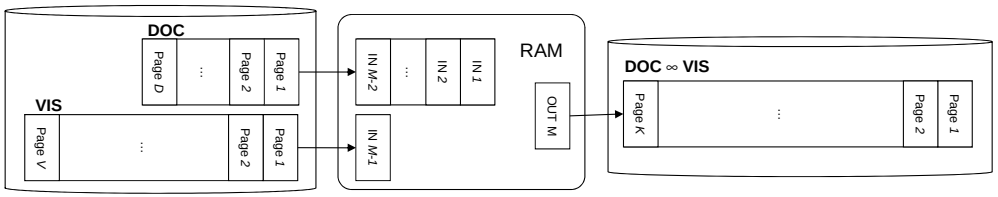
- Egalement appelée jointure par Produit Cartésien
- Algorithme Join(DOC, VIS, Q)

```
For each page p de DOC
  For each page q de VIS
    For each tuple tp de p
      For each tuple tq de q
          if  Check (tp, tq, Q) then result = result ∪ (tp||tq) ;
```

- Coût I/O = |DOC| + |DOC| * |VIS|
- Fonctionne quel que soit Q et quel que soit M ≥ 3
- Très sous-optimal dès que M > 3

Jointure : Block Nested Loop

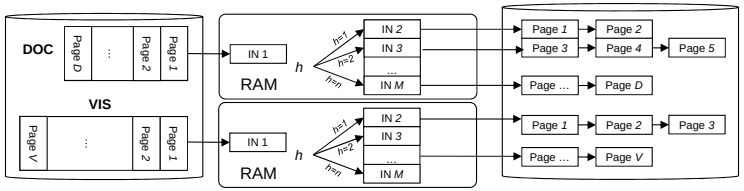
- Algorithme
  - DOC = table externe (*outer relation*) de la jointure
    - Choisir la table la plus petite comme table externe
    - Chargement par bloc de M-2 pages
  - VIS = table interne (*inner relation*) de la jointure
    - Chargement par bloc d'une page en RAM



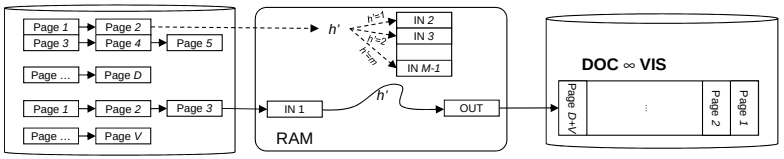
- Coût en I/O =
- Les idées simples sont parfois les meilleures
  - NB1: algo optimal si la table externe tient en mémoire !
  - *NB2: en plus, ne génère que des I/O séquentielles*

Jointure par hachage (*Grace Hash Join*)

- Principe de l'algorithme : (équi-jointure DOC.id = VIS.docid)
  - Phase de construction (*Build*) : Hacher DOC et VIS sur disque avec la même fonction h
    - Hachage en *n* partitions, avec $n \leq$ M-1



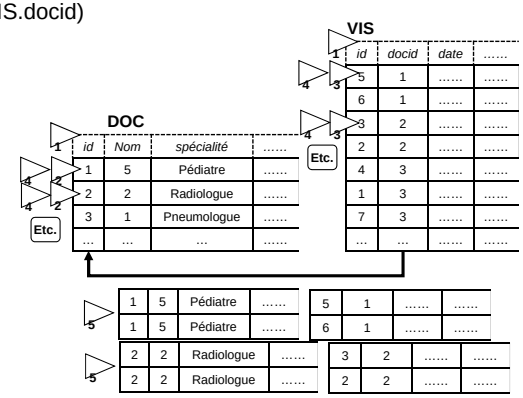
    - Coût I/O du Build =
  - Phase de test (*Probe*) : Joindre DOC et VIS par partition
    - Les tuples de la partition i de DOC joignent uniquement avec la partition i de VIS
    - Joindre les partitions de même numéro par Hash join



    - Coût I/O du Probe =
  - Diviser pour régner: une grosse jointure remplacée par n petites

Jointure par Tri Fusion : Sort-Merge Join

- Principe de l'algorithme : (équi-jointure DOC.id = VIS.docid)
  - Trier DOC (resp. VIS) sur DOC.id (resp. VIS.docid)
  - fusionner DOC et VIS
  - Produire les tuples résultat



- Coût du Sort-Merge Join
  - Tri des deux tables
    - $2\times|DOC|\times(1 + \lceil \log_{M-1} |DOC|/M \rceil)$ // ou 0 si DOC déjà triée
    - $2\times|VIS|\times(1 + \lceil \log_{M-1} |VIS|/M \rceil)$ // ou 0 si VIS déjà triée
  - Fusion
    - |DOC| + |VIS|

Condition générale de jointure

- NB : pour l'instant, equi-jointure mono-attribut…
- Equi-jointure multi-attributs
  - Jointure par boucle
    - Supporte sans modification toute condition de jointure
  - Jointure par index
    - Sans modification si index multi-attributs
    - Sinon, utilisation de l'index mono-attribut suivi d'un test systématique des autres prédicats de la condition
  - Jointure par tri-fusion ou hachage
    - Prendre en compte la combinaison des attributs comme clé de jointure
- Inequi-jointures
  - Jointure par boucle
    - Sans modification (et probablement optimal pour ce type de jointure)
  - Jointure par index et tri-fusion adaptables mais avec un coût élevé
  - Jointure par hachage inappliquable

Autres opérateurs

- Groupements et agrégats
  - Les tuples peuvent être groupés par tri ou hachage
  - Optimisation: calculer l'agrégat en même temps que le groupement
    - évite une seconde passe et réduit le nb d'I/O du groupement en maintenant en RAM le calcul intermédiaire plutôt que de la liste des tuples membres
- Union ($\cup$), intersection ($\cap$) et différence ($-$)
  - Adaptation des algorithmes de jointure (exemples)
    - $R \cap S$
      - Évaluer la condition de jointure sur tous les attributs clé et projeter sur les attributs d'une seule des deux tables
    - $R - S$
      - Évaluer la condition de jointure sur tous les attributs clé
      - Retirer de R tous les tuples qui joignent et produire le résultat
    - $R \cup S$
      - Calculer $(R - S) + S$, où + est une simple concaténation

CONCLUSION

- Pour chaque opérateur de l'algèbre, plusieurs implémentations possibles
- Pas d'algorithme systématiquement meilleur !
  - Le bon choix dépend de :
    - La taille des tables opérandes
    - La quantité de RAM affectée à l'opérateur
    - L'organisation des tables (indexées, hachées)
    - la sélectivité de la condition de jointure (PS: 90% des jointures sont des équi-jointures sur clé)
- Nécessité d'un modèle de coût précis et d'un optimiseur de requêtes
  - Mais encore faut-il que l'administrateur BD ait produit les conditions d'un bon choix !