4I803

Cours 3 et 4

Opérateurs relationnels

Implémentation et coût

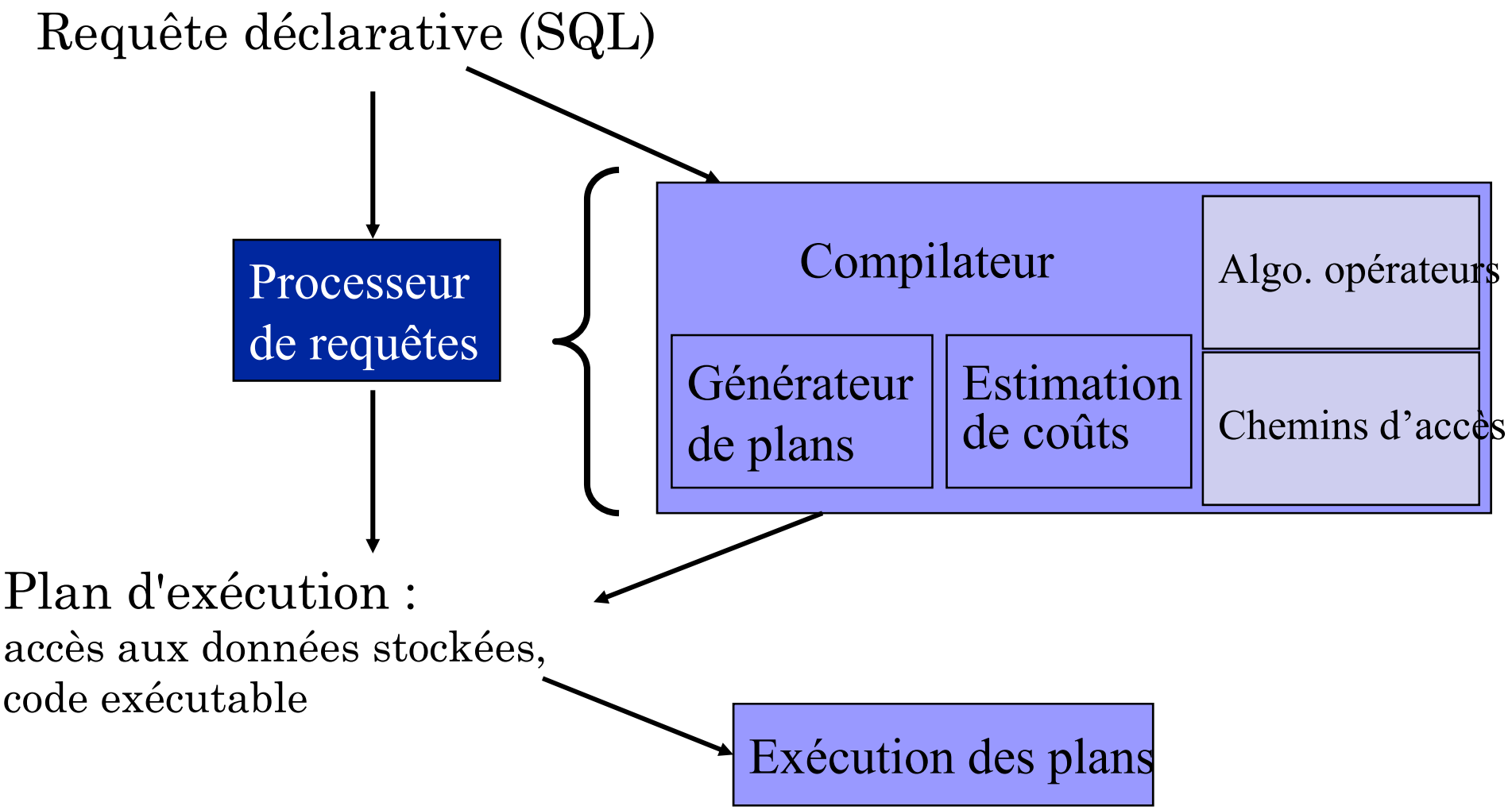
7 février 2023

Connaitre les diapos marquées d'une étoile 

Plan

- Rappel et Objectif
- Notion de pipeline
- Tri
- Sélection
- Projection
- Jointure
- Autres

Traitement des requêtes (rappel)



Objectif

- Comprendre les **algorithmes** qui évaluent les opérateurs relationnels
- Quantifier les accès aux données nécessaire pour évaluer une opération
  - Unité de mesure : la page
  - Les opérations principales sont :
    - sélection, projection, jointure, tri, ...
- Disposer d'un modèle (i.e., des formules) pour déterminer le **coût** d'une opération en termes d'accès aux données
- Hypothèses : coût E/S >> coût CPU
  - Lire un nuplet à partir d'une page de données stockée sur disque dure beaucoup plus longtemps que de calculer un nuplet à partir de données déjà en mémoire.

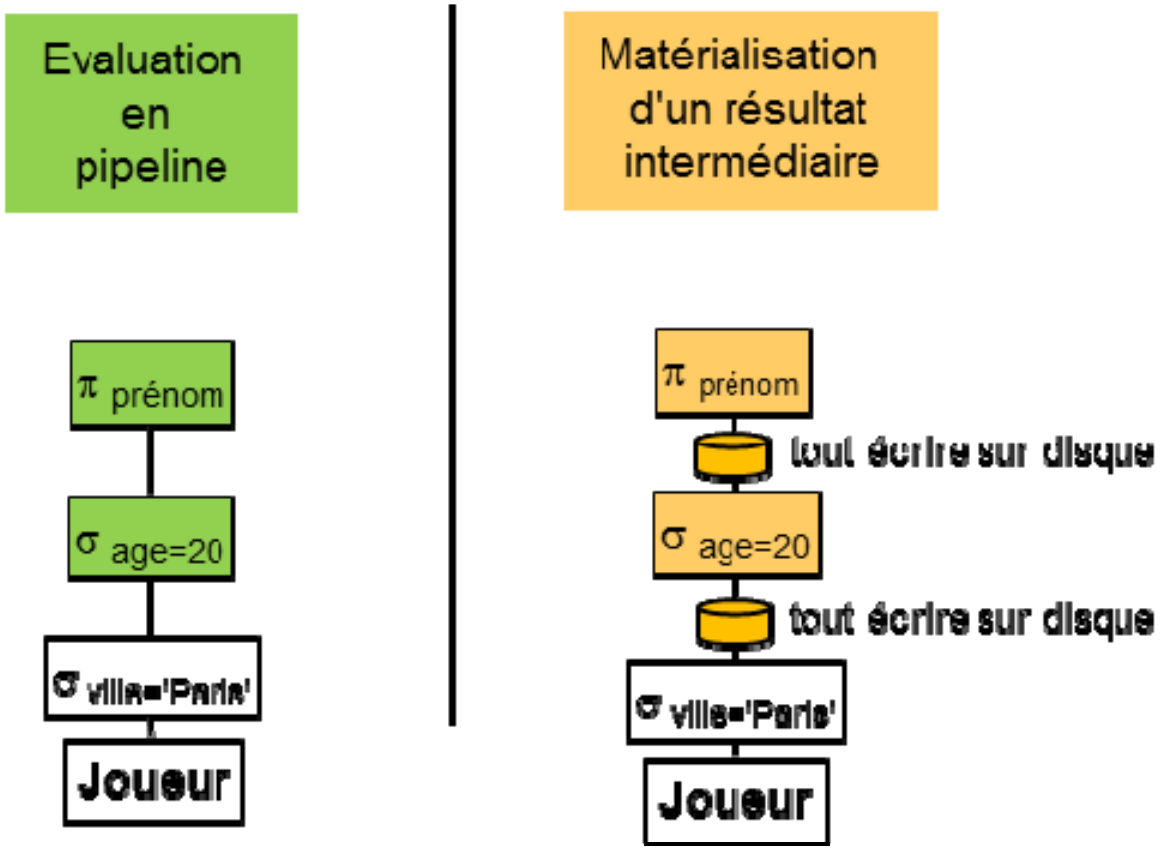
Implémentation des opérateurs

- Il existe plusieurs algorithmes *physiques* possibles pour un opérateur *logique*
  - comprendre les différentes variantes
- Détailler les **étapes** de l'algorithme physique
- Faire la distinction entre
  - Etape impliquant un accès aux données
  - Etape sans accès aux données

Evaluation en pipeline d'une opération

- Une opération est évaluée en **pipeline**
  - Si elle est évaluée **sans** lire aucune donnée stockée dans la base
    - Chaque opérande (données en entrée) doit être le résultat d'une autre opération
    - Opérande ≠ table
    - Opérande **non** matérialisée : jamais écrite temporairement sur disque avant d'évaluer l'opération
  - On « consomme » les opérandes pour « produire » la sortie progressivement
- Pipeline = traitement à la volée
- Avantage : moins couteux car pas de matérialisation

Pipeline vs. matérialisation



- Rmq: pas de pipeline pour la première sélection car accès aux données stockées.

Opération unaire évaluée en pipeline

- Une opération unaire a une opérande *e*
  - Exples : $\sigma_{prédicat}(e)$ $\pi_{attributs}(e)$
- Si l'opérande *e* est une expression composée d'au moins une opération, c-à-d, si $e \neq$ table alors
  - coût($\sigma_{prédicat}(e)$) = coût(*e*)
  - coût($\pi_{attributs}(e)$) = coût(*e*)

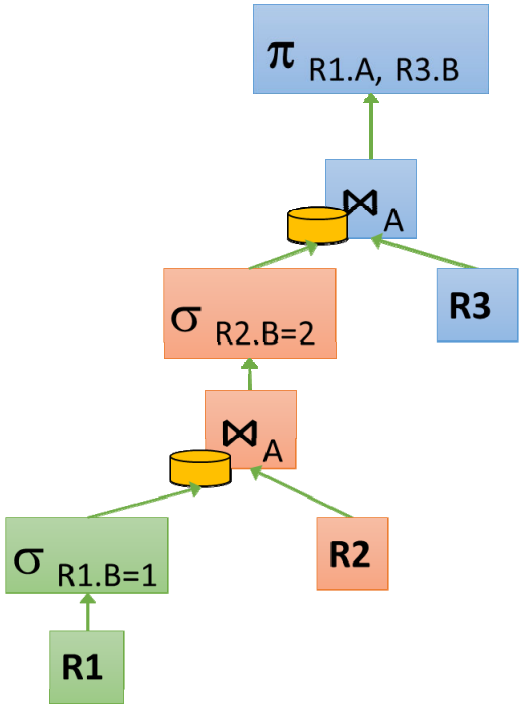
Opération binaire évaluée en pipeline

- Deux branches en pipeline
  - Union avec doublons
    - opération non relationnelle: UNION ALL en SQL
  - Fusion de listes déjà triées
- Une branche en pipeline
  - Jointure entre une "petite" et une "grande" relation
- Coût d'une opération *n-aire* en pipeline

  = somme du coût de ses opérandes

Opération binaire évaluée en pipeline

Arbre linéaire à gauche :



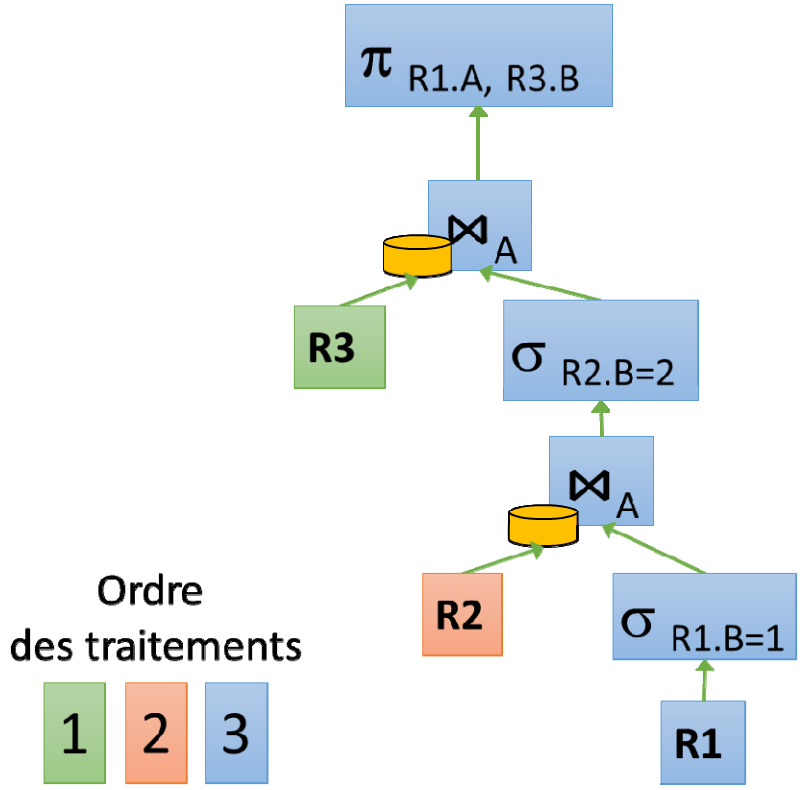
Ordre
des traitements

1 2 3

- "Flux" de nuplets "remontant" sur une branche
  - 3 branches : verte, rouge, bleue

Opération binaire évaluée en pipeline

Arbre linéaire à droite :



Evaluation itérative en pipeline

- Une opération en pipeline est **itérative**
  - Parcours itératif des nuplets de l'opérande pour calculer, un par un, les nuplets du résulat.
    - Le 1er nuplet du résultat dépend seulement des *m* premiers éléments de l'opérande
    - Le 2ème nuplet du résultat dépend seulement des *n* éléments suivants de l'opérande
    - … ainsi de suite jusqu'au dernier nuplet du résultat
- Avantage :
  - Chaque opérateur produit son résultat à la demande de son père
  - **Contrôle** du flux des nuplets intermédiaires

Implémention des opérateurs

- Modèle d'Itérateur
  - Interface commune à tous les opérateurs :
    - méthodes open(), nextTuple(), close()
    - nextTuple() invoque récursivement nextTuple() des opérandes
  - Permet une implémentation en pipeline ou avec matérialisation
  - Avantages :
    - Facilite l'exécution d'un plan : le plan est lui-même un itérateur
    - Permet de calculer progressivement le résultat de manière interactive
      - latence réduite pour produire les n premiers tuples du résultat
- Génération dynamique de code
  - Avantages :
    - optimisation tardive avec information plus récente sur les ressources disponibles (mémoire)
    - pas d'appels imbriqués de méthodes nextTuple()
  - Inconvénient: temps pour compiler la requête

Implémention des opérateurs

- Modèle Itérateur

```
[ ]  class Operateur():
         def __init__(self):
             pass

         def open(self):
             pass

         def next(self):
             pass

         def close(self):
             pass
```

```
[ ]  # TABLE ACCESS FULL
     class ParcoursSequentiel(Operateur):
```

```
[ ]  class Selection(Operateur):
```

```
[ ]  class HashJoin(Operateur):
```

Requête SQL transformée en un plan

Plan = arbre composé d'objets de classe Operateur

Résultat : itérer sur l'objet racine du plan

Algorithmes des opérateurs relationnels

- Diapos suivantes
  - Parcours séquentiel
  - Sélection
  - Projection
  - Jointure
  - Tri
  - ...

Parcours séquentiel d'une table

- Requête:
  - Select * from R

- *R* stockée dans *page(R)* pages du disque
  - Taille d'une page en octets : T_{page}
  - Nombre de nuplets de R dans une page :
    - T_{page} / largeur(R)
  - page(R) = card(R) / (T_{page} / largeur(R))

- Parcours séquentiel : *Table Access FULL*
  - Coût(R) = page(R) . *c*
    - avec $c < 1$ si les pages à lire sont contigües (exple c=0,27 en TME)
    - sinon $c = 1$

Sélection : σ

- Sélection : $\sigma_{p(A)}$ (...)
  - avec p(A) est un prédicat dépendant de l'attribut A
- Si E est une expression composée
  - Coût($\sigma_{p(A)}$ (E)) = coût(E)
- Si T est une table et l'attribut A n'est pas indexé
  - Coût($\sigma_{p(A)}$ (T)) = *page(T)*

Sélection par index **non** plaçant: : σ_{NP}

- Soit *IdxA* l'index non plaçant sur R.A, et *p(A)* le prédicat de sélection

```
for rowid in IdxA.getRowIds( p(A) ):
    r = R.getTuple(rowid)
    add r in result
```

1. Traverser l'index : **`getRowIds( p(A) )`**
   - Atteindre une feuille de l'index
     - $C_{index} = 0$ si l'index tient en mémoire
     - Sinon C_{index} = (hauteur de l'arbre -1).
   - Lire le(s) **`rowid`**
     - Index Unique Scan : $C_{rowid} = 0$ (les feuilles de l'index unique contiennent des rowid)
     - Index Range Scan : $C_{rowid} = \lceil (card(\sigma_{p(A)}(R)) / card(R)) * $ nbre de pages contenant les rowids $\rceil$
2. **Lire les tuples associés aux rowId : `getTuple( rowID )`**
   - Voir: Table Access By Rowid

- Coût($\sigma_{NP\ p(A)}(R)$) = $C_{index} + C_{rowid} + card(\sigma_{p(A)}(R))$ **. CF / card(R)**

CF: Clustering facteur

- On a : page(R) < CF < card(R)
- Indique dans quelle mesure le « rangement » des tuples dans une page dépend de l'attribut indexé
- CF est déterminé à partir d'entrées consécutives de l'index
  - Pour une plage [v1, v2] de valeurs consécutives
    - Les entrées { (v1, {rowid}), (v2, {rowid}) } dans cette plage contiennent N rowid
      - N rowid font référence à P pages à lire
      - Nombre moyen de rowid retrouvés dans une page
        1 < N/P < nbre de tuples par page
    - CF = card(R) / (N/P)

- CF faible =

  les tuples ont été insérés avec des valeurs croissantes de l'attribut indexé
- CF élevé =

  tuples ont été insérés indépendant de la valeur de l'attribut indexé

Sélection par index **plaçant :** σ_P

- Soit *IdxA* l'index plaçant sur R.A, et *p(A)* le prédicat de sélection

```
for page P_R in IdxA.getPages( p(A)):
  for tuple r in P_R:
       add r in result
```
on suppose que tous les r de Pr satisfont p(A)

1. Traverser l'index pour obtenir l'adresse de la première page indexée
   - C_{index}= 0 l'index tient en mémoire ou pour le hachage linéaire
   - Sinon
     - C_{index}= 1 pour le hachage extensible
     - C_{index}= hauteur de l'arbre – 1 pour un arbre B+
2. Lire les pages "pleines" de tuples du résultat = **lire une fraction de la table**

- Coût ($\sigma_{P\ p(A)}$ (R)) = $\lceil$ page(R) * **SF(p(A))** $\rceil$
  - Rappel : SF(p(A)) est le facteur de sélectivité du prédicat p(A)

Sélections complexes

- Sélections complexes sur une table contenant plusieurs prédicats
- Lorsque plusieurs attributs sont indexés séparément
  - Conjonction (AND)
    - Intersection d'adresses de nuplets (rowid)
    - Vecteur binaire puis ET logique
  - Disjonction (OR)
    - Union d'adresses de nuplets
    - Vecteur binaire puis OU logique

Projection : π

- Projection sans doublons : $\pi_{Attributs}(R)$
  - R est une relation
  - Correspond au "Select distinct Attributs"
  - si $\pi_{Attr}(R)$ tient en mémoire ou si R est sans doublons
    - Coût($\pi_{Attr}(R)$) = coût(R)
  - si $\pi_{Attr}(R)$ ne tient **pas** en mémoire. Deux possibilités :
    - En triant
      - Lire R pour matérialiser R trié selon *Attributs,* puis lire R trié
    - OU en hachant
      - Lire R et la hacher sur disque, insérer uniquement si nouvelle valeur pour « Attributs » puis lire chaque paquet
- Projection SQL avec doublons : Select Attributs
  - Opération **non** relationnelle
  - Coût($proj_{Attr}(R)$) = coût(R)

Jointures: ⋈

- Diapos suivantes :
- Boucles imbriquées
  - simple
  - par bloc
  - avec index
- Par hachage
- Par Tri fusion
- ...

Boucles imbriquées : $\bowtie$

- On suppose que S est une table
- Jointure notée R $\bowtie_{R.a=S.a}$ S
  - On note r un tuple r de R, s un tuple de S

```
for r in R:
  for s in S:
      if r.a = s.a:
         add (r,s) in result
```

- Avantage :
  - Algo général, permet d' évaluer tout prédicat de jointure
  - Exple (théta jointure) : R $\bowtie_{R.a > S.a}$ S
- Inconvénient :
  - Relire S pour chaque tuple de R
- Amélioration :
  - Relire S pour chaque **partie** de R : voir diapo suivante

Boucles imbriquées par blocs : ⋈

- On suppose que M+2 pages de R tiennent en mémoire
  - On peut "charger" M pages de R en mémoire
- Possibilité de **réduire** le nombre d'accès à S
  - Itération principale sur R par blocs de M pages
  - Puis joindre chaque nuplet de S avec le bloc courant
- Algo :

```
foreach Br of R do
  foreach tuple s ∈ S do
    foreach tuple r ∈ Br,
      if r.a=s.a then add <r,s> in result
```

- Coût(R $\bowtie_{A.a=B.a}$ S) = coût(R) + page(R)/M . page(S)
  - Rmq, en TD: M = 1 page

Boucles imbriquées avec matérialisation : $\bowtie_{Mat}$

- Sert lorsque S est une sous-expression : S ≠ table
  - Matérialiser S **avant** de calculer la jointure
- Jointure notée R $\bowtie_{Mat\ R.a=S.a}$ S
  - Etape préliminaire
    - Evaluer S : coût(S)
    - Stocker son résultat dans S_{Mat} Coût pour écrire S_{Mat} : page(S)
  - Jointure par boucles imbriquées entre R et S_{Mat}

Coût(R $\bowtie_{Mat\ R.a=S.a}$ S) = coût(S) + page(S) +coût(R $\bowtie_{R.a=S.a}$ S)

  - Exple pour M=1
    - Coût(R $\bowtie_{Mat\ R.a=S.a}$ S) = coût(S) + page(S) + coût(R) + page(R) . page(*S*)

Jointure par boucles avec index (1)

- Jointure
  - par boucles imbriquées et
  - index sur l'attribut *a* de S
    - Évite de parcourir S entièrement pour chaque bloc de R

$$\text{Coût}(R \bowtie_{\text{Ind R.a=S.a}} S) = \text{coût}(R) + \text{card}(R) \,.\, \text{coût}(\sigma_{a=v}(S))$$

- Cas particulier de la jointure sur clé
  - si l'attribut *a* est une clé de S, alors $\text{coût}(\sigma_{a=v}(S)) = 1$

- Cas général :
  - le terme $\textit{coût}(\sigma_{a=v}(S))$ dépend du type d'index
    - voir détails sur les 2 diapos suivantes

Jointure par boucles avec index (2)

Jointure avec index **non plaçant** *IdxSa* sur S.a

- On suppose que l'index tient en mémoire : $C_{index} = 0$

```
for r in R:
    for i in IdxSa.getRowIds(r.a):
        s = S.getTuple(i)
        add (r,s) in result
```

- coût($\sigma_{a=v}(S)$) = C_{rowid} + card($\sigma_{a=v}(S)$) * **CF** / card(S) *(cf diapo sélection)*

- donc coût(R $\bowtie_{Ind\ R.a=S.a}$ S) =

  coût(R) + card(R) . [C_{rowid} + card($\sigma_{a=v}(S)$) * **CF** / card(S)]

Jointure par boucles avec index (3)

- Jointure avec index **plaçant** sur S.a
  - En supposant que tous les *s* de *Ps* satisfont s.a = p.a

```
foreach tuple r ∈ R do
    foreach page Ps ∈ IdxSa.getPages(r.a) do
      foreach tuple s ∈ Ps do
          add <r,s> in result
```

  - coût($\sigma_{a=v}$ (*S*)) = ⌈ page(S) * SF(a=v) ⌉
  - donc coût(R $\bowtie_{\text{Ind R.a=S.a}}$ S) =

    coût(R) + card(R) . ⌈ page(S) * SF(a=v) ⌉

Jointure par tri puis fusion (1)

- Trier *R* et *S* sur l'attribut de jointure : Sort(join)
  - voir tri externe
- Fusionner les relations triées : Merge join
- Amélioration pour réduire le nombre de lectures et écritures
  - Laisser R triée en plusieurs morceaux sans les fusionner. Idem pour S.
  - On peut fusionner directement les 2 relations dès que le nombre de paquets restant dans les 2 relations est inférieur à k
    - Trier R en P_R paquets, et trier S en P_S paquets, tq : $P_R + P_S < k$
  - Fusion des Pr paquets de R avec les Ps paquets de S en **une** seule étape
- Exemple : jointure entre 2 tables R et S
  - Page(R)=6000, page(S)=3000, k=100 pages en mémoire
    - Coût du tri = 2. (page(R) + page(S)
    - On obtient 60 blocs de R et 30 blocs de S soit un total de 90 blocs
    - Il suffit de lire les blocs pour les fusionner : coût = page(R) + page(S)
    - Bilan: coût($R \bowtie_{TF\ R.a=S.a} S$) = 3 (page(R) + page(S))

Jointure par fusion (2) algorithme détaillé

- Principe : itérer progressivement sur R et S
- S'il y a plusieurs tuples de r (ou s de S) pour une valeur de a, produire toutes les paires (r,s)
  - R={(1,b) (7,z) (7,b) (7,c)} S = {(6,e) (7,e) (7,a) (9,i)}
  - Le résultat contient 6 tuples: 7ze 7be 7ce 7za 7ba 7ca
- Algo :
  - Initialiser r et s
    - r ← premier tuple de R, s← premier tuple de S
  - tant que r et s existent
    - si r.A = s.A alors
      - tmpR← r
      - continuer d'itérer sur R pour ajouter dans tmpR les tuples tq R.A = S.A
      - tant que s existe et s.A = r.A
        - » Pour chaque t dans tmpR, ajouter (s,t) dans le résultat
        - » S← suivant(S)
    - si r.A < s.A alors r ← suivant(R) sinon si s.A < r.A suivant(S)

Jointure par hachage (1)

- Hypothèse : R est plus grande que S page(R) ≥ page(S)
- Principe : traitement en 2 étapes
  1) Lire S pour la hacher selon la clé de jointure
  2) Itérer sur les tuples de R et jointure sur clé
- S tient en mémoire créer une hashmap en mémoire

```
1) init: Map : hashmap <A, List<Tuple>>
     foreach tuple s ∈ S do
         -- créer la liste L et l'associer à S.a si elle n'existe pas
         List L ← Map.getOrCreate(S.a)
          add s in L
2) foreach tuple r ∈ R do
      L ← Map.get(r.a)
      foreach tuple s ∈ L add (r,s) in result
```

Coût(R $\bowtie_{H\ R.a=S.a}$ S) = coût(S) + coût(R)

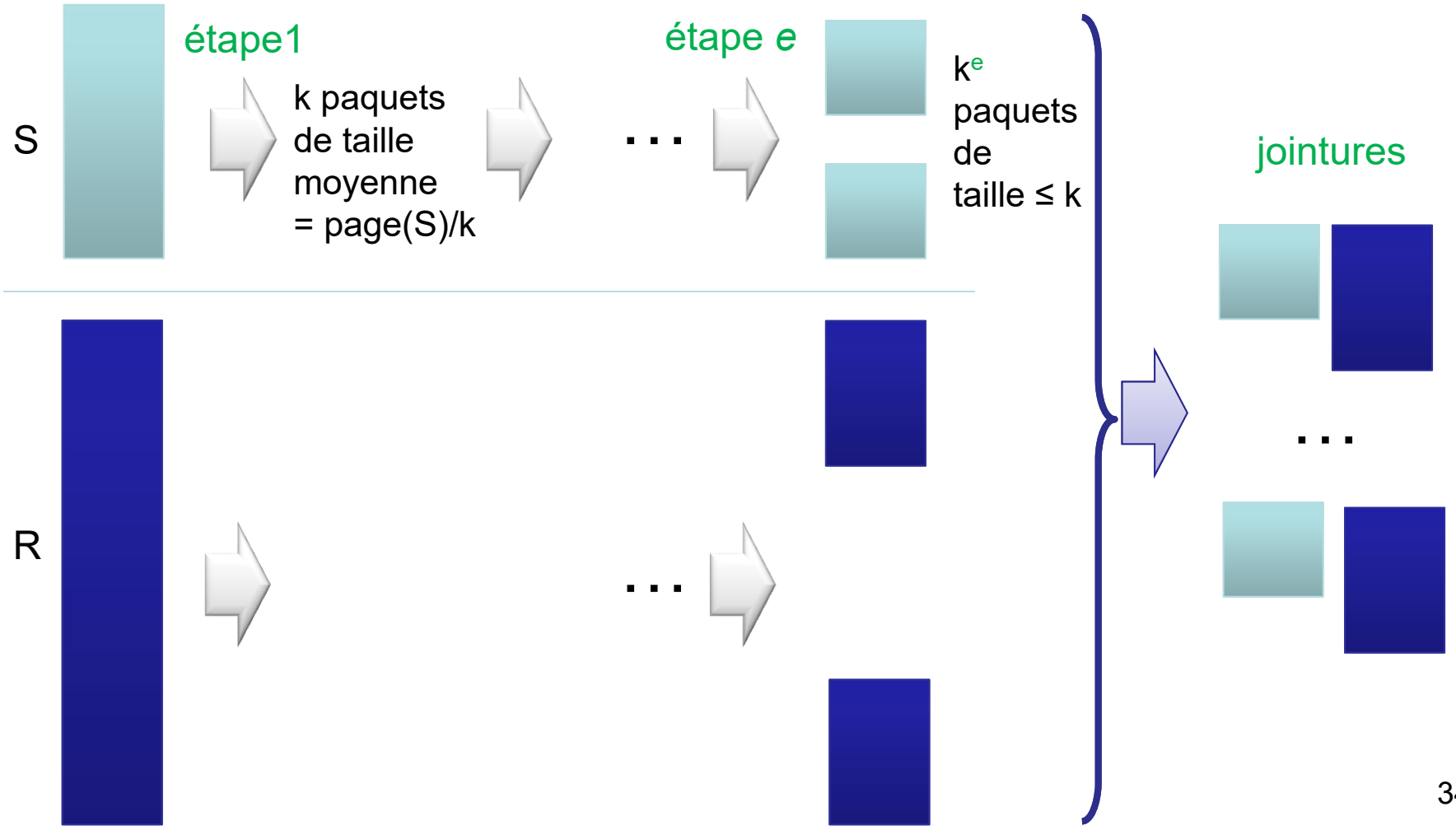
Jointure par hachage : cas du hachage externe : $\bowtie_{HExt}$

- Hachage externe si S ne tient **pas** en mémoire :
  - Algorithme « Grace Hash join »
  - Taille de la mémoire : k+1 pages
  - Hacher S (puis R) sur disque
    - Répartir les données de S dans k 'paquets'
      - Le paquet Si contient les tuples de S tels que h(S.a) = i
      - Continuer à répartir récursivement jusqu'à ce que la taille des paquets de S soit <= k pages
    - Répartir les données de R en utilisant les mêmes fonctions de hachage que celles ayant servit à hacher S.
      - Un paquet de R peut avoir une taille > k
    - Coût : $2e.\ (\ page(R) + page(S)\ )$ avec $e = \lfloor \log_k(page(S)) \rfloor$
  - Charger le 1er paquet de S en mémoire puis lire le 1er paquet de P pour faire la jointure. Idem pour les paquets suivants
    - Coût : page(S) + page(R)

$$Coût(R \bowtie_{HExt\ R.a=S.a} S) = 2e\,(\ page(R) + page(S)\ ) + page(R) + page(S)$$

Rmq: dans Oracle le surcoût est moindre (variante du Grace join) .

Hachage externe : illustration du Grace Hash join



Jointure n-aire

- Evaluer n jointure binaires
- Parcourir l'arbre de jointure en profondeur d'abord
- Evaluer les opérations en remontant
  - depuis l'opération la plus à gauche
- Exple: $(R \bowtie_a S) \bowtie_b T$ et jointure par boucles imbriquées

```
– foreach tuple r ∈ R  do
    • foreach tuple s ∈ S do
        – if r.a=s.a then
            » foreach tuple t ∈ T do
                » if r.b=t.b then
                    » add <r,s,t> in result
```

- Exple: $(R \bowtie_a S) \bowtie_b T$ et jointure par hachage

```
– foreach tuple r ∈ R  do
    • add (a,r) in hashMap M1
– foreach tuple s ∈ S do
    • Lr = M1.get(s.a)
    • foreach r ∈ Lr
        – add (b ,(r,s)) in hashMap M2
– foreach tuple t ∈ T do
    • Lrs = M2.get(t.b)
    • foreach (r,s) ∈ Lrs
        – add <r,s,t> in result
```

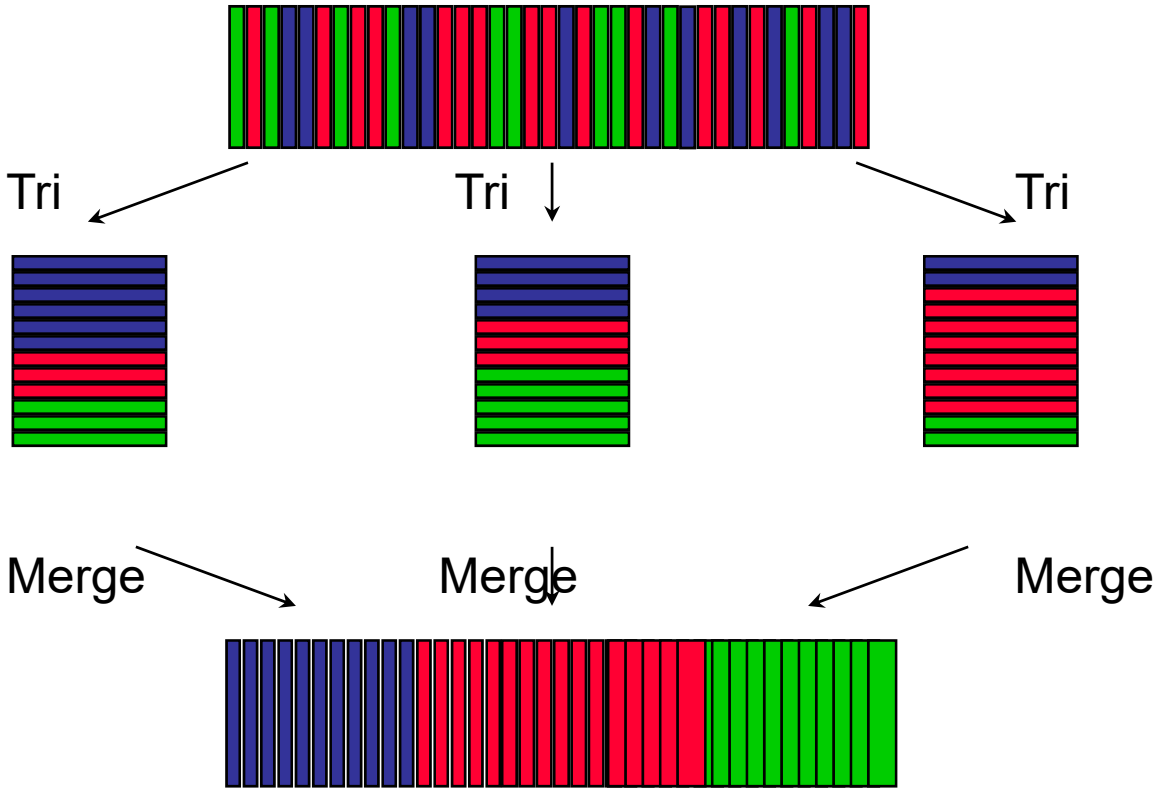
Order by

- voir tri externe

Tri externe

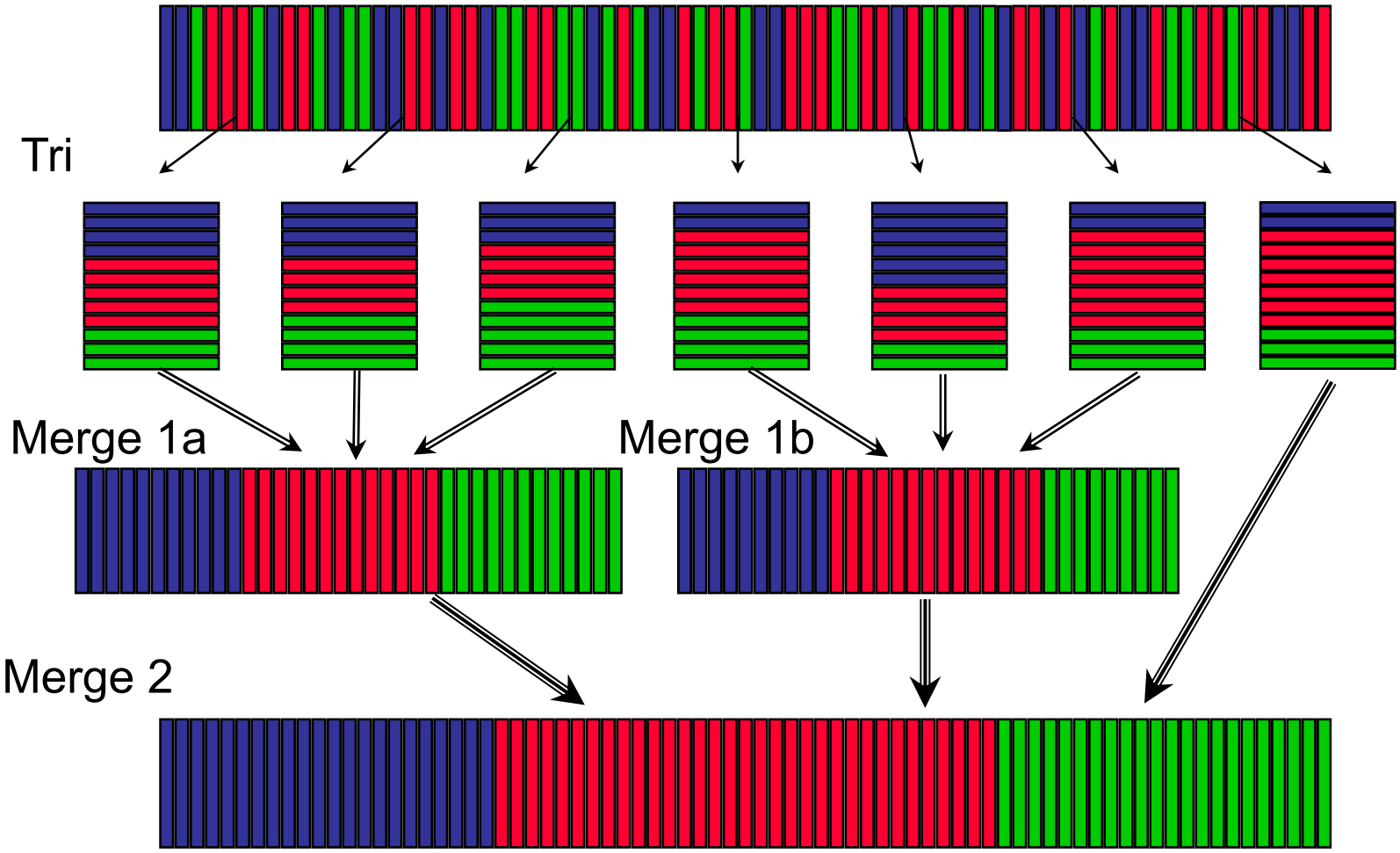
- Hypothèse : k pages tiennent en mémoire
- Algorithme en ***s*** étapes : tri de blocs puis fusions de blocs
- Etape n°1 : tri
  - Lire R pour créer des paquets triés de k pages chacun
  - Nombre de paquets obtenus : page(R)/k
  - Coût de l'étape de tri (lecture + matérialisation) : 2.page(R)
- Puis étapes n° 2, 3, ..., *s* : fusion
  - Fusion
    - Charger la première page de k paquets et les fusionner
    - Dès qu'une page est vide, charger la suivante du même paquet
    - Dès que les k premiers paquets ont été fusionné, fusionner les k paquets suivants
    - On obtient des paquets triés de taille k^2 pages
    - Coût d'une étape : lire et matérialiser toutes les données : 2.page(R)
    - Nombre de paquets : page(R)/k^2
  - Continuer jusqu'à obtenir **un seul** paquet, on a donc :
    - **page(R) / k^s ≤ 1**

Tri externe : Fusion en 1 seule étape



Ref: CS460 Vassilis.C

Fusion en plusieurs étapes



Ref: CS460 Vassilis.C

Tri externe (suite)

- Nombre d'étape *s* tq : $k^s \geq$ page(R) :
  $s = \lceil \log_k(\text{page}(R)) \rceil$

- Coût total des *s* étapes (inclut le tri et les étapes de fusion):
  - Lorsque le résultat final du tri est matérialisé
  - Coût(tri(R)) = 2 . page(R) . *s*

- Si on ne matérialise **pas** le résultat de la dernière étape
  - Dernière étape = seulement lire R
    - Exple : R trié sera affiché ou transmis à une autre opération.
  - Coût(tri(R)) = 2 . page(R) . (*s* - 1) + page(R)

- Si E est une expression composée (E ≠ table)
  - Tri fait en pipeline : la première lecture de E consiste à évaluer E et le stocker. Le dernier résultat n'est pas matérialisé.
    - Coût(tri(E)) = [coût(E) + page(E)] + [2.page(E).(*s*-2)] + [page (E)]
  - ⇔ Coût(tri(E)) = coût(E) + 2 . page(E) . (*s* - 1)

Autres opérations

- Group By avec agrégation
  - Hachage ou tri

- a IN (*sous-requête*)
  - Evaluer la sous-requête pour chaque valeur de a
    - Algorithme général de type "boucles imbriquées"
  - Lorsque la sous-requête ne dépend **pas** de la requête principale
    - Evaluer une (semi) jointure : Requête principale $\ltimes$ sous-requête
      - Charger la sous requête en mémoire : voir algo de type $\bowtie_{H}$

      Ou
      - Matérialiser la sous-requête : voir algo de type $\bowtie_{Mat}$

Perspectives

- Nombreuses autres variantes pour implémenter les opérateurs relationnels
- Evaluation en parallèle d'un opérateur
- Tri externe en parallèle
  - Sort benchmark : voir le site sortbenchmark.org
    - 2009: 500 GO/mn, 2013: 1,4 TO/mn,
    - 2014: 4,3 TO/mn http://sortbenchmark.org/ApacheSpark2014.pdf
    - 2015: 16 TO/mn
    - 2016: **44** TO/mn http://sortbenchmark.org/TencentSort2016.pdf
    - Coût 'vert' du tri:
      - » prix $ par TO/mn, énergie consommée par TO/mn