L'opérateur qui nous manque

La jointure : opération très courante, potentiellement coûteuse.

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

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

Plusieurs algorithmes possibles.

Principaux algorithmes

On ve se limiter à quelques exemples représentatifs :

Jointure avec index

- Algorithme de jointure par boucles imbriquées indexées.

Jointure sans index

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

Jointure avec index

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

- Les films et leur metteur en scène

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

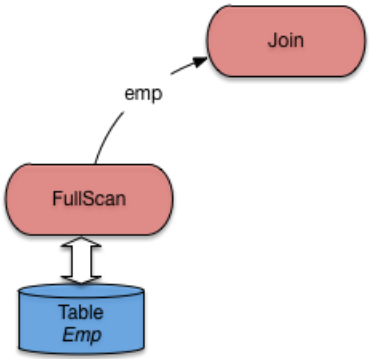
- Les employés et leur département

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

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

Jointure avec index : l'algorithme

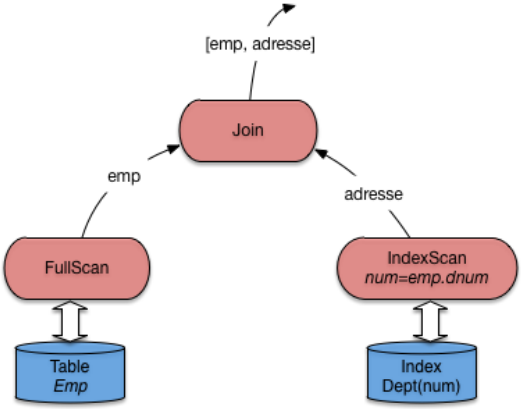
On parcourt séquentiellement la table contenant la clé étrangère.



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

Jointure avec index : l'algorithme

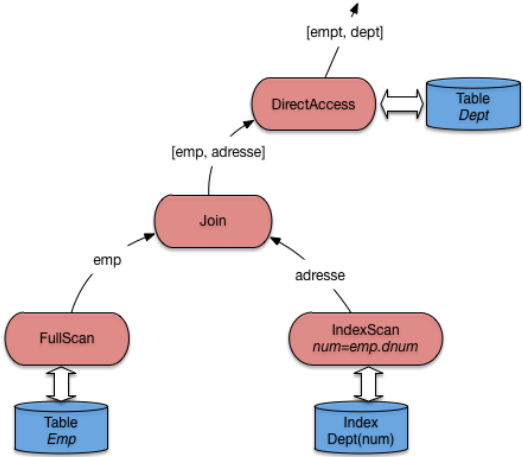
On utilise le no de département pour un accès à l'index `Dept`.



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

Jointure avec index : l'algorithme

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



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

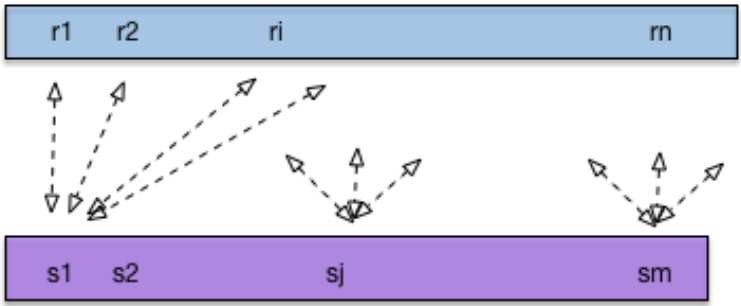
Jointure avec index : l'algorithme

Avantages :

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

Jointures par boucles imbriquées

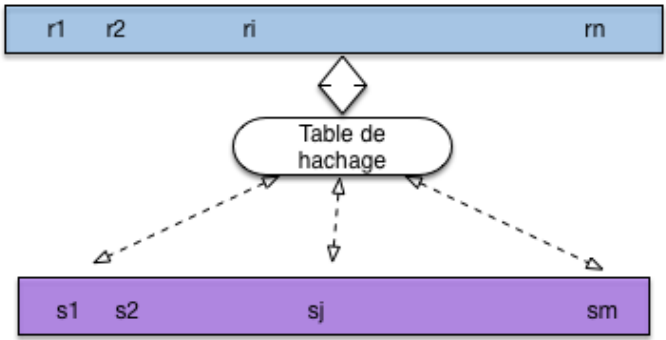
Pas d'index ? La méthode de base est d'énumérer **toutes** les solutions possibles.



Coût quadratique. Acceptable pour deux petites tables.

Jointures par hachage

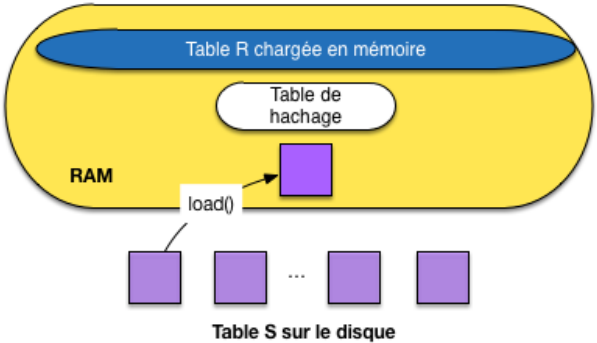
Meilleure solution : construire une table de hachage sur une des tables.



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

Mémoire insuffisante ?

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

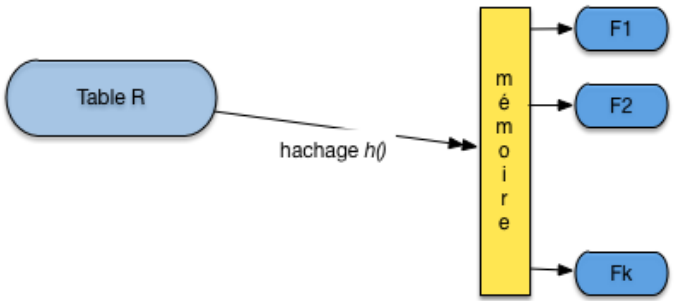


On charge l'autre bloc par bloc ; on applique **JoinList**.
Une seule lecture de chaque table suffit.

.

Et quand *aucune* table ne tient en mémoire ?

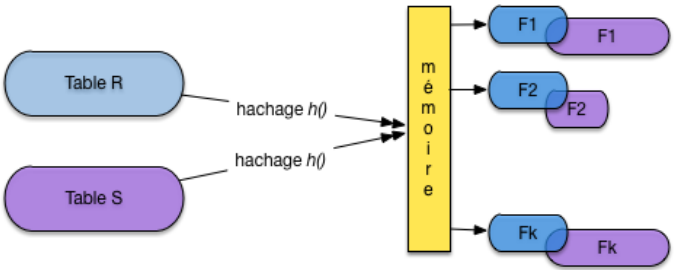
On hache la plus petite des deux tables en k fragments.



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

Et quand *aucune* table ne tient en mémoire ?

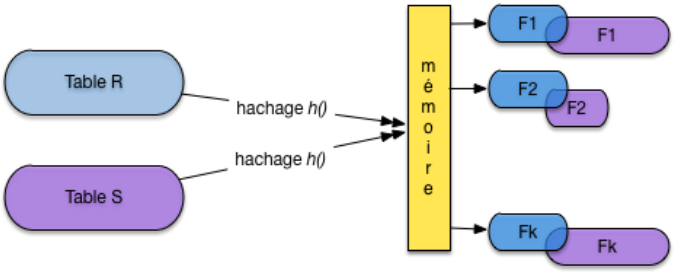
On hache la seconde table, avec la même fonction $h()$, en k autres fragments.



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

Et quand *aucune* table ne tient en mémoire ?

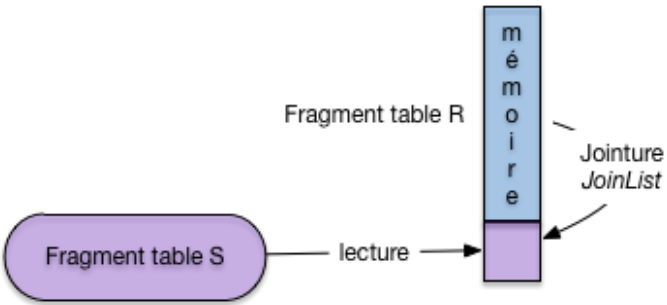
On effectue la jointure sur les paires de fragments $(F_1^R, F_1^S), (F_2^R, F_2^S), (F_k^R, F_k^S),$



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

Illustration : phase de jointure

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



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

Résumé : la jointure

Un opérateur potentiellement coûteux. Quelques principes généraux :

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

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

Résumé : la jointure

Un opérateur potentiellement coûteux. Quelques principes généraux :

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

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

Merci !