BDW - Optimisation des bases de données

Fabien Duchateau

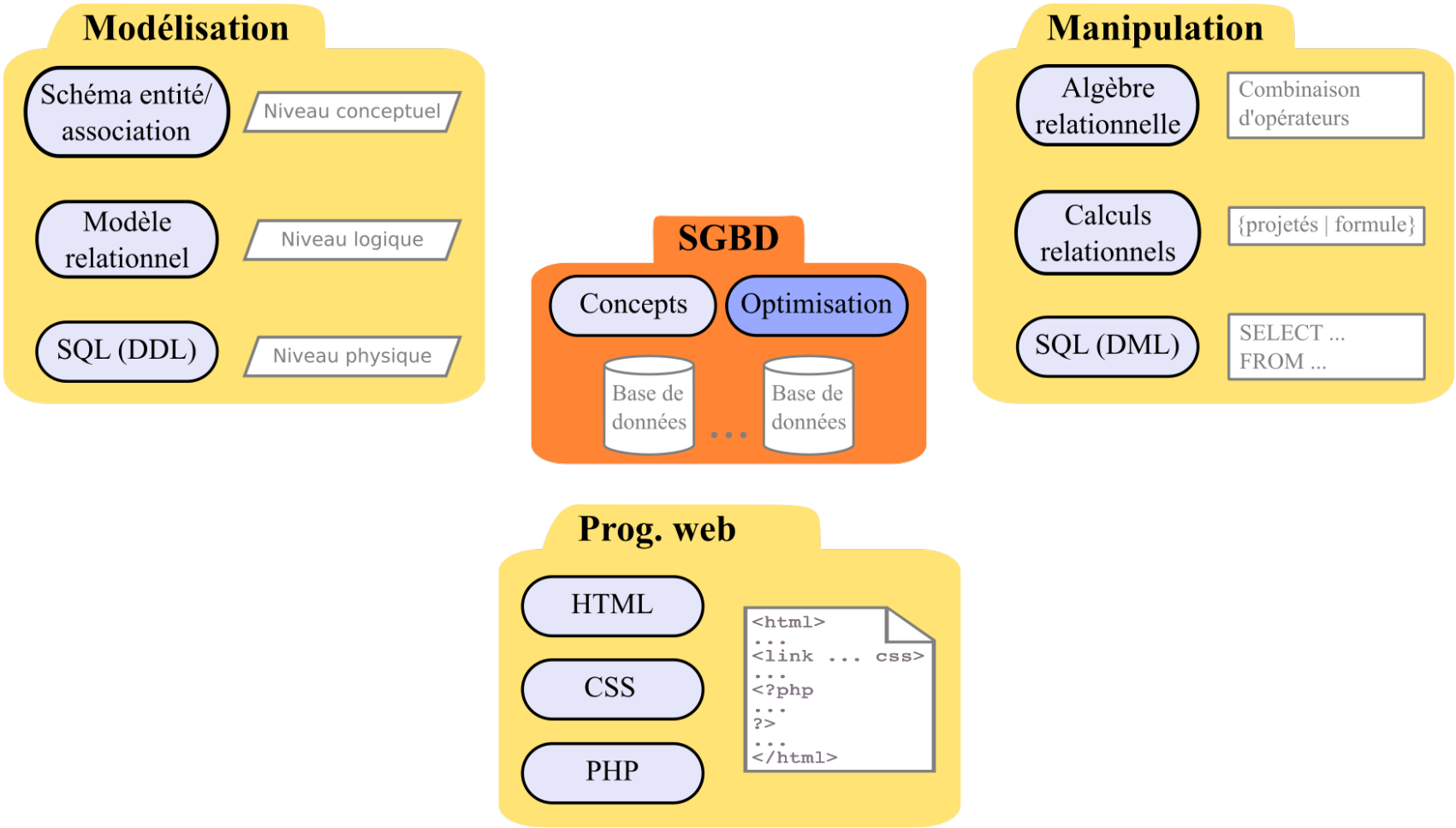
fabien.duchateau [at] univ-lyon1.fr

Université Claude Bernard Lyon 1

2023 - 2024

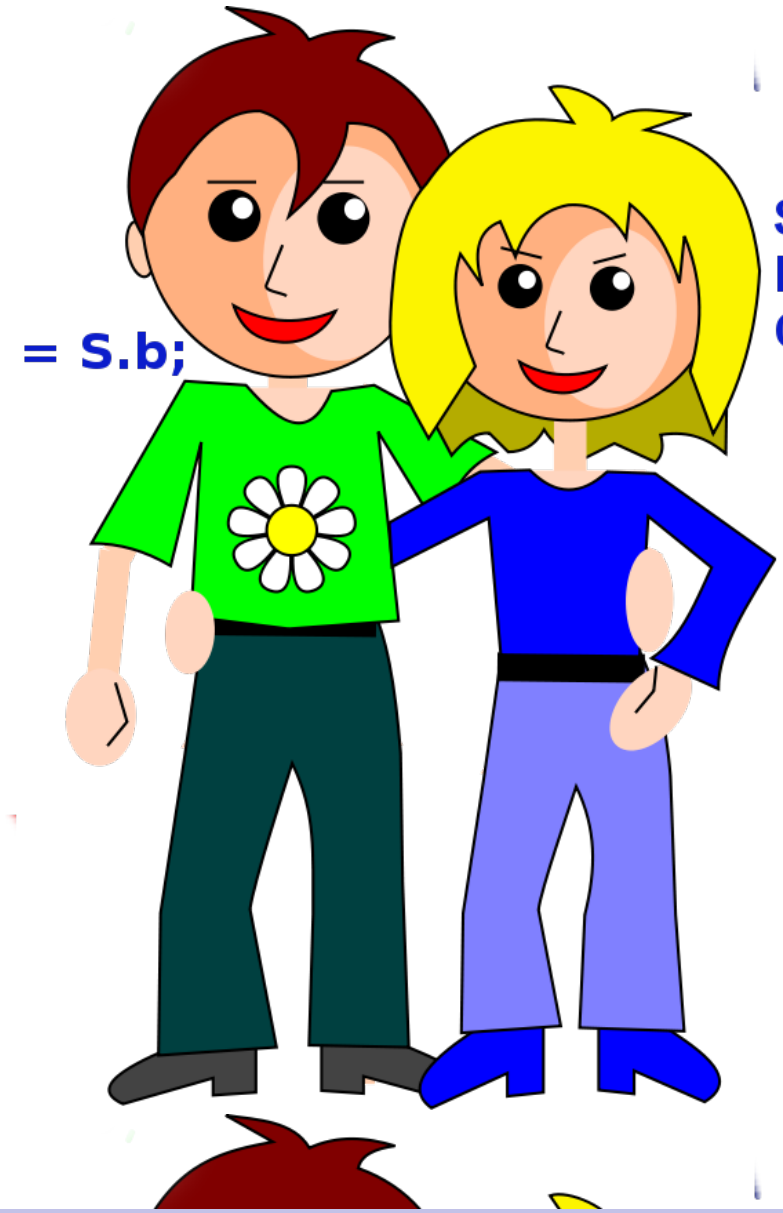
https://perso.liris.cnrs.fr/fabien.duchateau/BDW/

Positionnement dans BDW



Ces diapositives utilisent le genre féminin (e.g., chercheuse, développeuses) plutôt que l'écriture inclusive (moins accessible, moins concise, et pas totalement inclusive)

Motivation



Motivation (2)

Une requête SQL est exécutée par le SGBD, mais :

- Quel est l'impact des différentes manières d'écrire une requête pour un même résultat (e.g., NOT IN / NOT EXISTS) ?
- Comment passe t-on de la requête (définie dans un langage déclaratif) à un programme (impératif) manipulant les données ?
- Comment optimise t-on une requête afin de l'exécuter efficacement pour trouver un résultat correct ?
- Pourquoi les requêtes préparées permettent de gagner en performance (entre autre) ?

---

http://sqlpro.developpez.com/cours/optimiser/

Optimiseurs

L'optimiseur de requêtes d'un SGBD est un composant crucial :

- Il réécrit la requête sous différentes formes
- Il choisit la réécriture la plus performante pour exécuter la requête
- Des benchmarks évaluent et comparent les SGBD, notamment les performances de leur optimiseur

---

Jarke, Matthias, and Jurgen Koch. *Query optimization in database systems*. ACM Computing surveys (1984)

Graefe Goetz. *Query evaluation techniques for large databases*. ACM Computing Surveys (1993)

Ioannidis, Yannis. *Query Optimization*. The Computer Science and Engineering Handbook (1996)

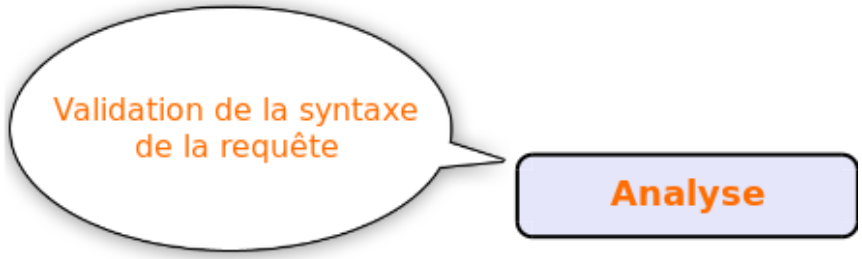
Plan

Traitement d'une requête SQL

Optimisation à base de règles

Estimation du coût d'un plan

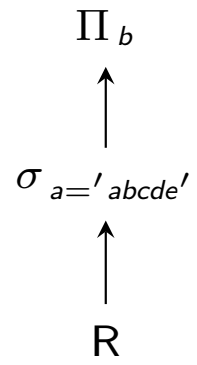
Traitement d'une requête



Où l'on retrouve l'algèbre relationnelle...

Une requête en algèbre relationnelle peut se représenter sous forme d'arbre algébrique

$\Pi_b(\sigma_{a='abcde'}(R))$



- Racine de l'arbre = résultat de la requête
- Feuilles de l'arbre = relations
- Noeuds intermédiaires de l'arbre = un opérateur algébrique (e.g., sélection, union, jointure)

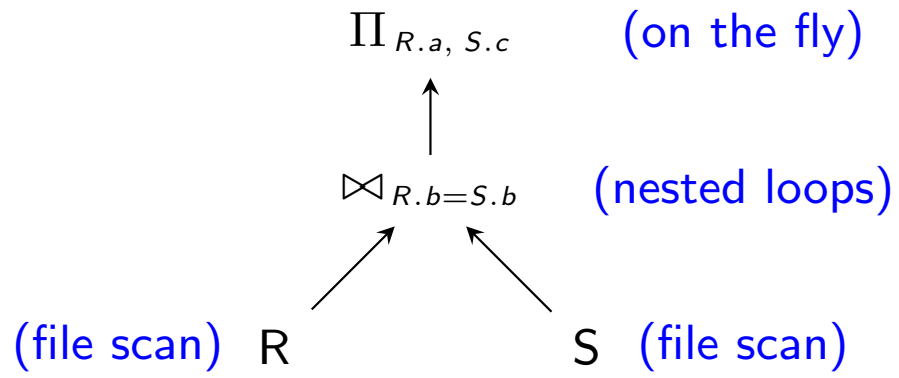
Arbre algébrique ≈ plan d'exécution d'une requête

Où l'on retrouve l'algèbre relationnelle... (2)

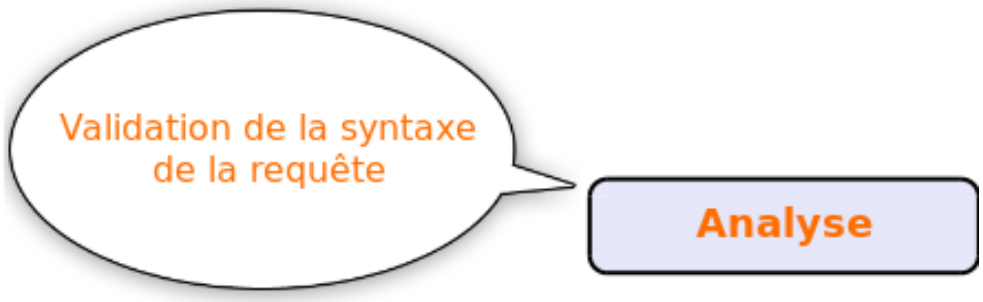
- Le plan d'exécution d'une requête est généralement optimisé selon trois opérateurs (projection, sélection et jointure)
- Les autres opérateurs (e.g., regroupement, tri) sont réalisés ensuite (ajoutés au plan optimisé des trois opérateurs)
- Un plan d'exécution indique quel algorithme est appliqué pour chaque opérateur

SELECT R.a, S.c
FROM R, S
WHERE R.b = S.b ;

$\Longrightarrow$



Étape d'analyse

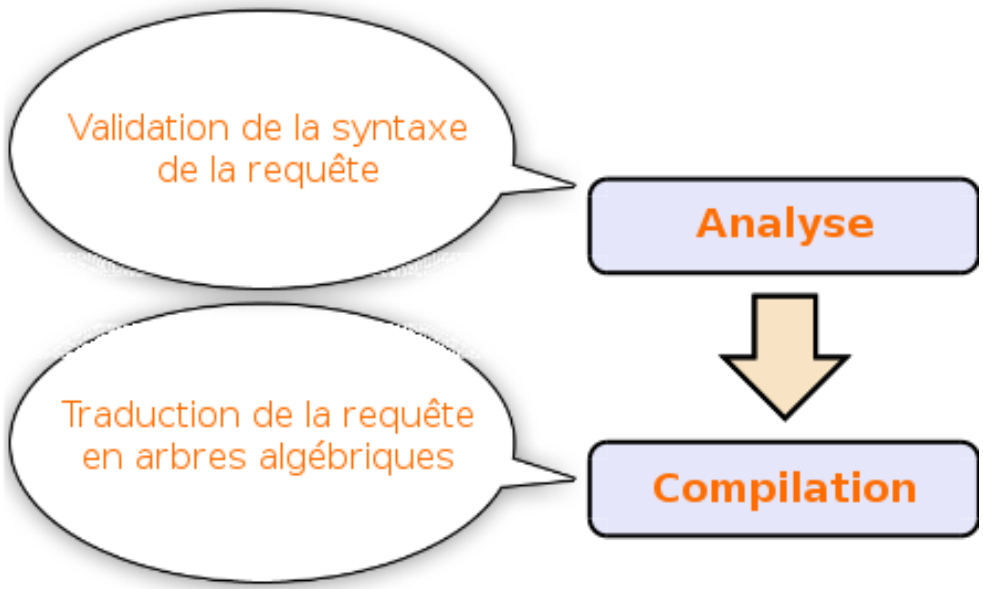


Étape d'analyse - détails

Analyse lexicale et syntaxique :

- Validation par rapport à la syntaxe SQL
- Vérification des types
  - présence des attributs et relations dans le schéma
  - compatibilité des types dans les prédicats
- Décomposition en requêtes/sous-requêtes

Étape de compilation



Étape de compilation - détails

Règles de passage d'une requête SQL en AR :

- SELECT pour définir une ??**projection**
- FROM pour définir les ?? **relations** (feuilles de l'arbre) et ??**jointures / produits cartésiens**
- WHERE pour définir :
  - avec les comparaisons *attribut/constante* des ??**sélections**
  - avec les comparaisons *attribut/attribut* des ??**jointures**

Étape de compilation - exemple

SELECT R.a
FROM R, S
WHERE R.b = S.b
    AND S.c = 99 **;**

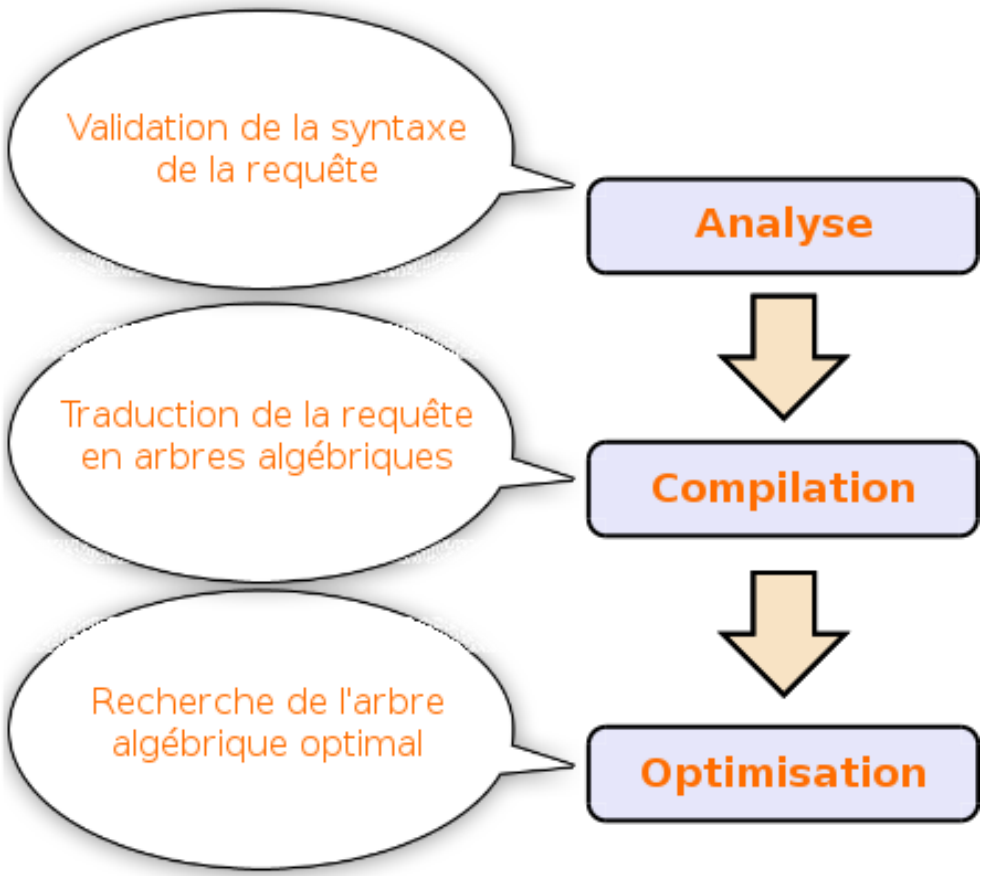
Compilation "naïve" :

- $\pi_a(\sigma_{R.b=S.b \wedge c=99}(R \times S))$

Autres possibilités :

- $\pi_a(\sigma_{R.b=S.b}(R \times \sigma_{c=99}(S)))$
- $\pi_a(R \bowtie_{R.b=S.b} (\sigma_{c=99}(S)))$

Étape d'optimisation



Étape d'optimisation - définition

Optimiser, c'est trouver le plan d'exécution d'une requête dont le coût soit minimal (i.e., le temps de réponse à la requête soit le plus rapide possible)

Qu'est-ce qui peut être optimisé dynamiquement ?

- Les accès disques

Il est indispensable que le temps lié à l'optimisation soit négligeable par rapport au temps imparti à l'exécution

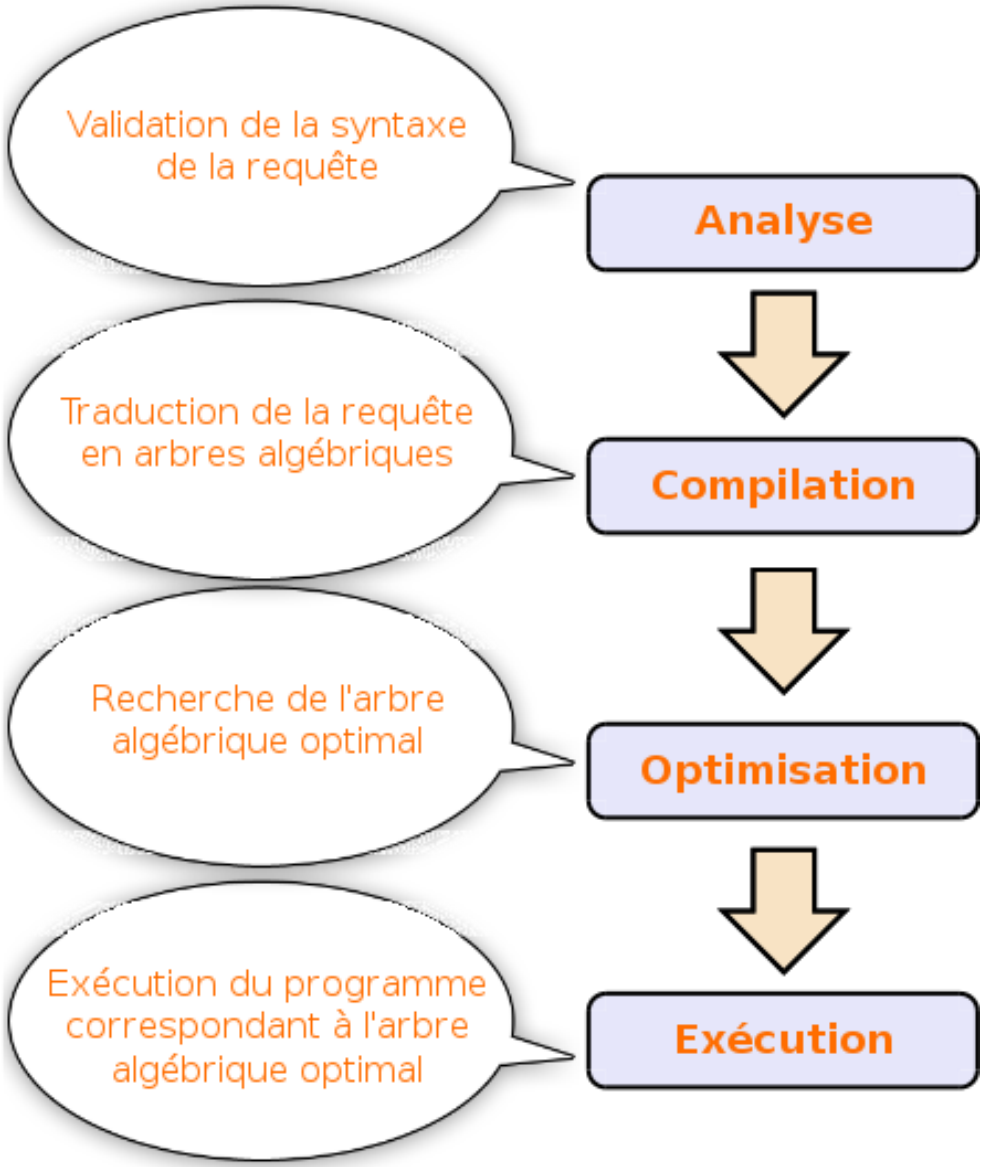
Étape d'optimisation - détails

Intuitions pour optimiser et sélectionner le plan optimal :

- Des compositions de projections et/ou sélections peuvent se réécrire en une opération de filtrage
- Réduire au plus tôt la taille et le nombre de tuples manipulés :
  - tuples moins nombreux : grâce à la ??**sélection**
  - tuples plus petits : grâce à la ??**projection**
- Réordonner les jointures, en fonction de la taille des relations manipulées
- Choisir des algorithmes efficaces pour chaque opérateur (e.g., *merge join*, *hash join* pour la jointure)

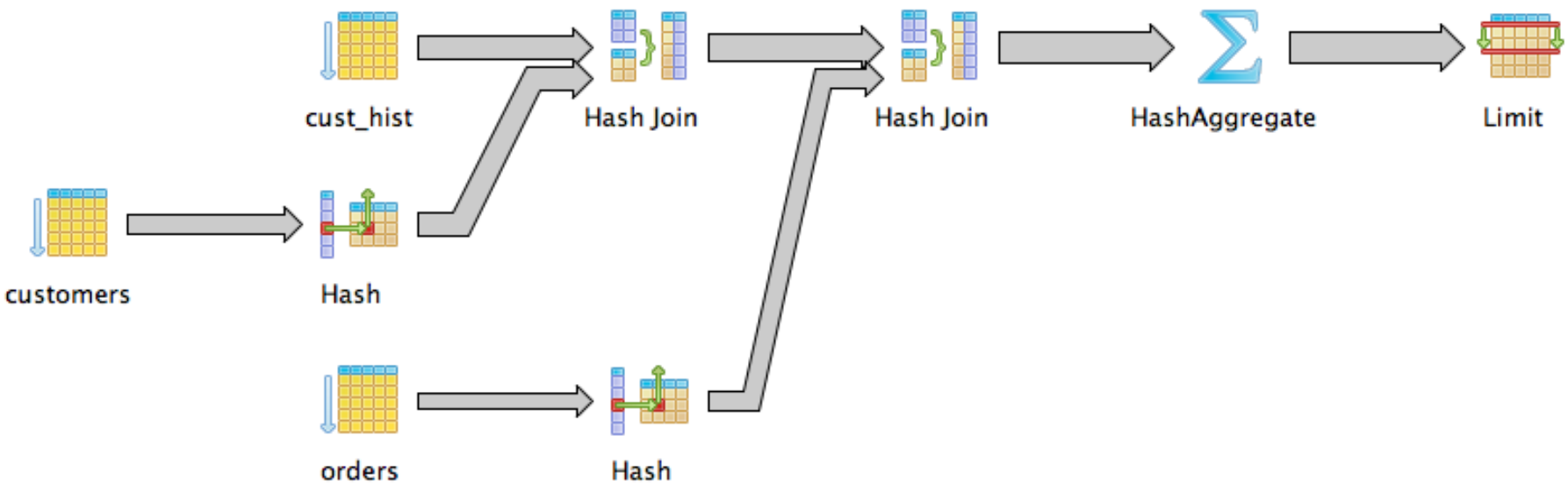
Explications détaillées dans les parties suivantes

Étape d'exécution



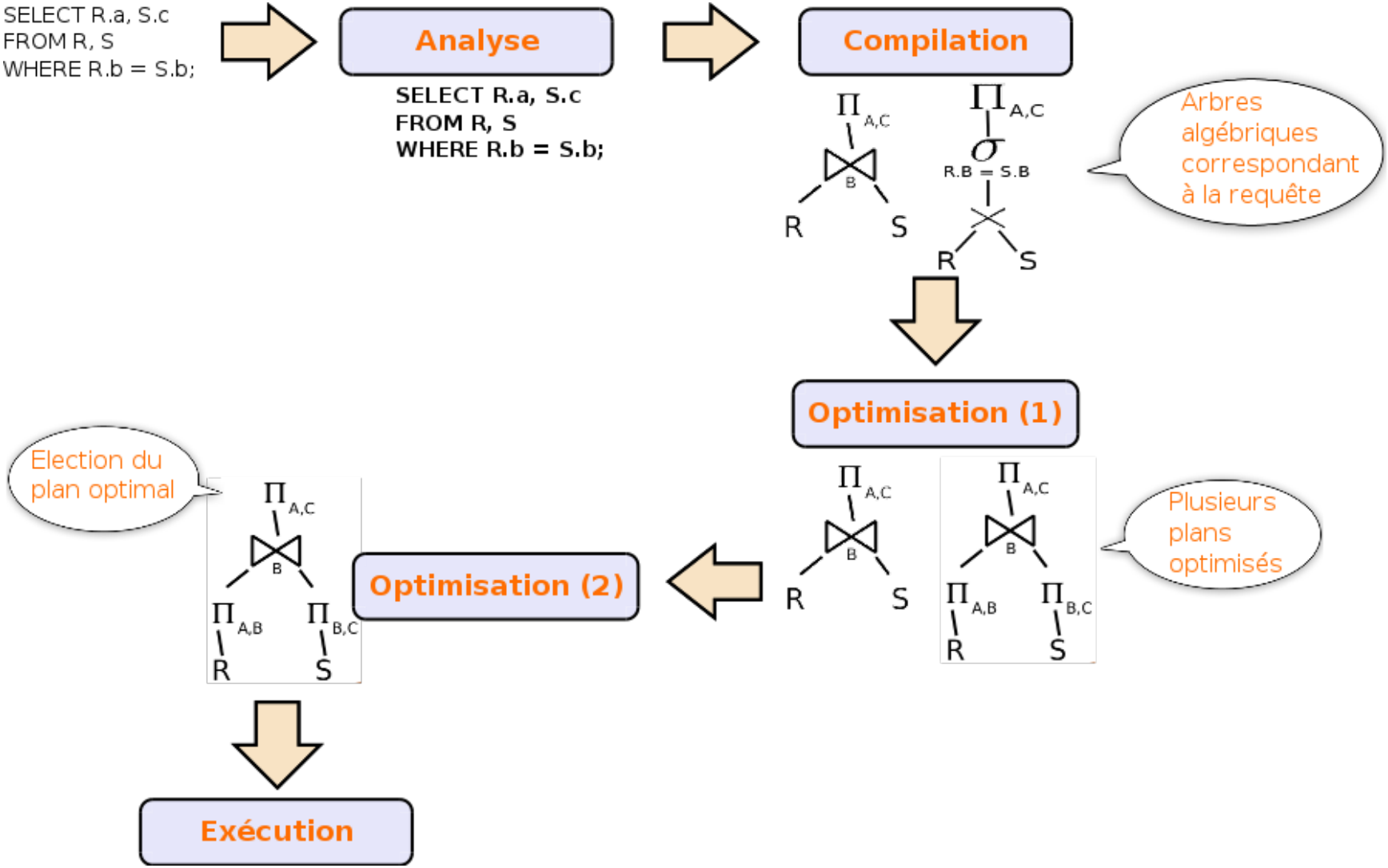
Étape d'exécution - détails

Application des algorithmes pour chaque opération du plan d'exécution sélectionné



http://mariadb.com/kb/en/mariadb/analyze-and-explain-statements/

Exemple complet de traitement d'une requête



En résumé

Traitement d'une requête :

- Validation
- Compilation en arbre algébrique
- Optimisation (génération de plans optimisés et sélection du plan optimal)
- Exécution du plan optimal

http://vidberg.blog.lemonde.fr/

Plan

Traitement d'une requête SQL

Optimisation à base de règles

Estimation du coût d'un plan

Optimisation

L'objectif est de trouver une stratégie d'évaluation permettant d'accélérer l'évaluation :

- Par manipulations algébriques, indépendante de la manière dont sont stockées les informations
- En utilisant une stratégie dépendante du stockage (clés, index)

Dans ce cours, manipulations algébriques (à base de règles)

Plan d'exécution

Un plan d'éxécution $\approx$ arbre algébrique

Plus formellement, un **plan d'exécution** est un programme qui vise à évaluer une requête en algèbre relationnelle et qui consiste en une suite d'étapes parmi les suivantes :

- Application d'un opérateur unaire (sélection ou projection)
- Application d'une sélection puis d'une projection
- Application d'un opérateur binaire ($\times, \cup, \cap, \setminus, \bowtie$)
  - éventuellement précédé et/ou suivi d'opérateurs unaires

Un algorithme d'optimisation a pour but de trouver un bon (le meilleur si possible) plan d'exécution

Espace de recherche

Espace de recherche = l'ensemble des plans "équivalents" pour une même requête :

- Ceux qui donnent le même résultat
- Générés en appliquant des règles de transformation

Caractéristiques des plans d'un espace de recherche :

- Le coût de chaque plan est en général différent
- L'ordre des jointures est important

Si l'espace de recherche est très grand, l'optimiseur peut chercher un plan raisonnable, mais pas forcément optimal (par exemple sous PostgreSQL, dès qu'il y a plus de 12 jointures)

Exemple d'espace de recherche

Élève	(idE, *nomE*, *moyenneLycée*, *effectifLycée*)
Candidature	(#*idE*, #*nomU*, département, *décision*)
Université	(nomU, *ville*, *effectif*)

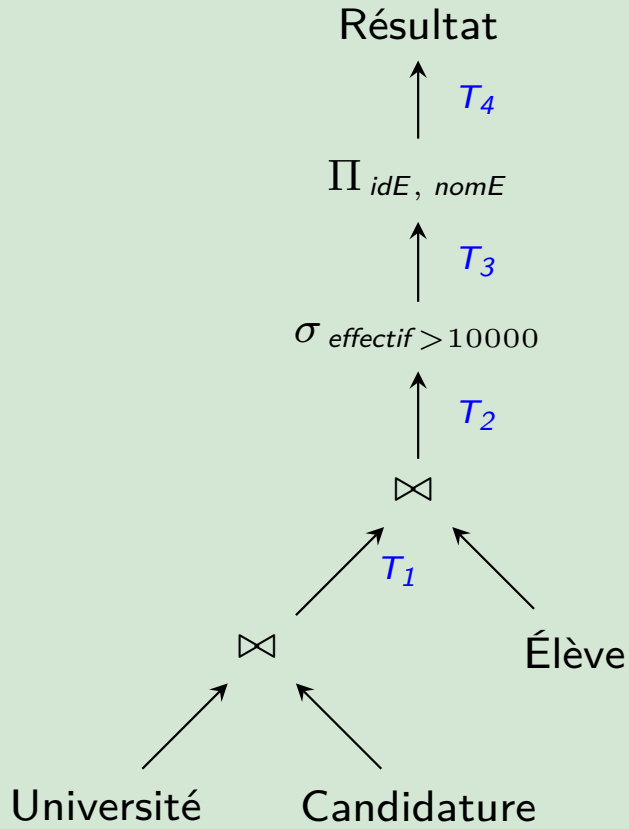
L'identifiant et le nom des élèves qui ont candidaté dans des universités avec un effectif supérieur à 10000

```
SELECT DISTINCT idE, nomE
FROM Élève e INNER JOIN Candidature c
     ON e.idE = c.idE INNER JOIN
     Université u ON c.nomU = u.nomU
WHERE effectif > 10000 ;
```

Exemple d'espace de recherche - plan 1

SELECT DISTINCT idE, nomE
FROM Élève e **INNER JOIN** Candidature c
ON e.idE = c.idE **INNER JOIN**
Université u **ON** c.nomU = u.nomU
WHERE effectif > 10000 ;

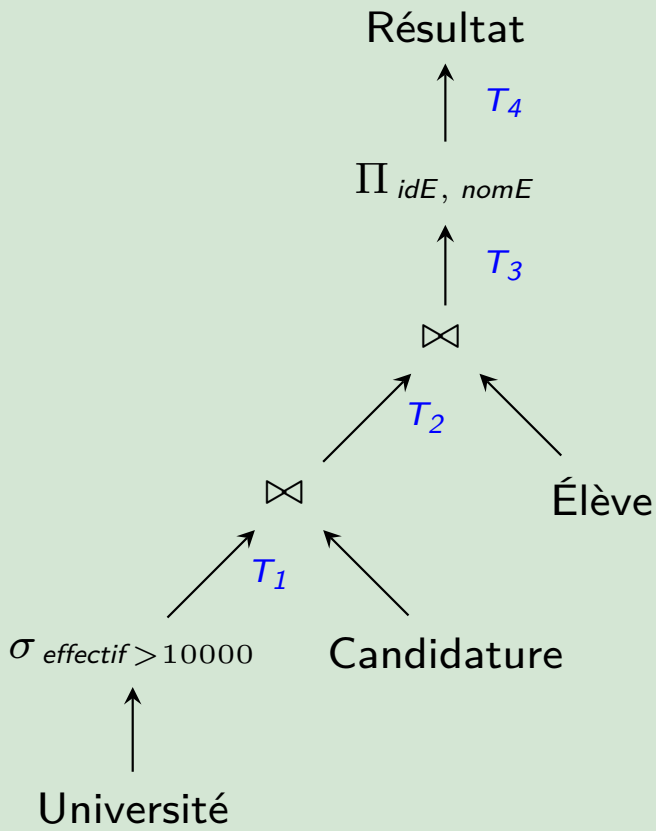
- $T_1 \leftarrow$ Joindre la table UNIVERSITÉ avec la table CANDIDATURE
- $T_2 \leftarrow$ Joindre T_1 avec la table ÉLÈVE
- $T_3 \leftarrow$ Lire T_2 et sélectionner les tuples d'*effectif* > 10000
- $T_4 \leftarrow$ Projeter T_3 sur *idE*, *nomE*



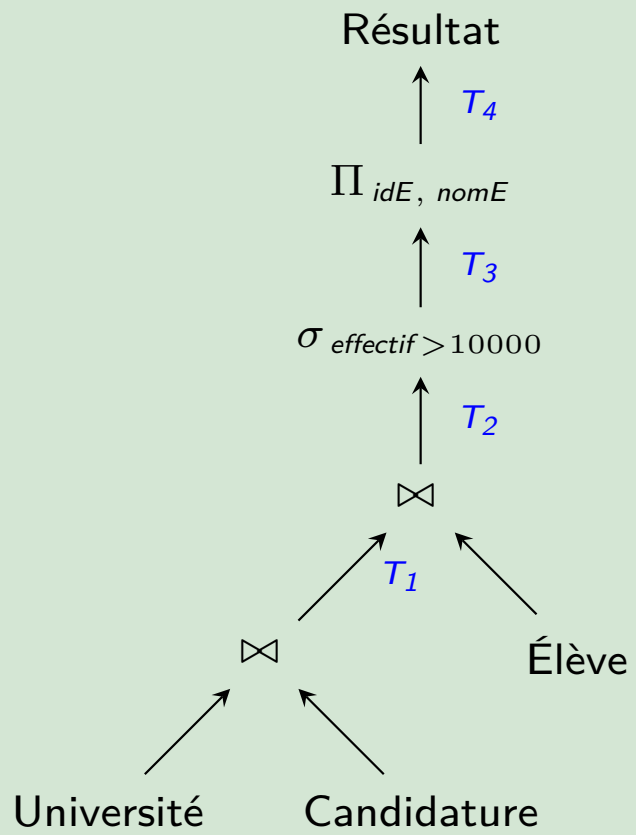
Exemple d'espace de recherche - plan 2

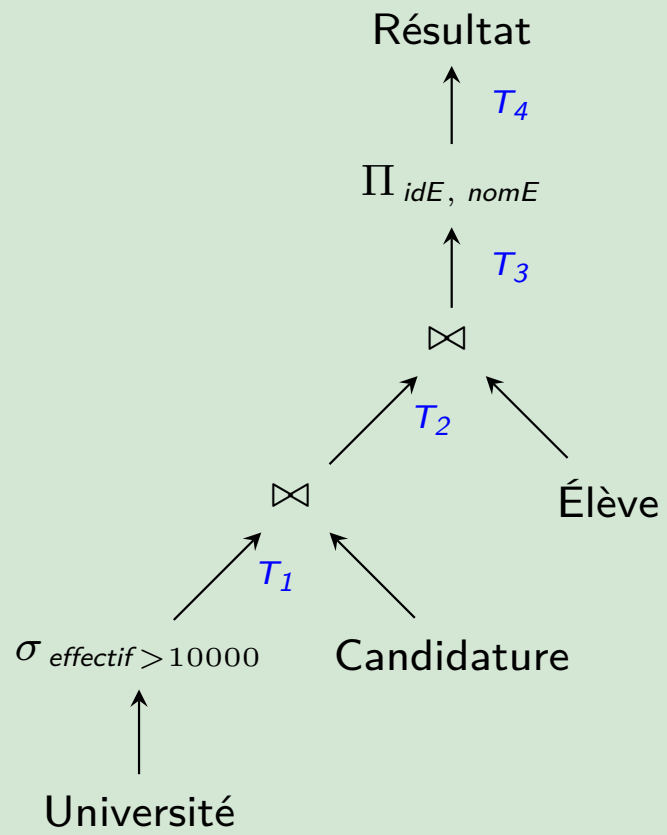
SELECT DISTINCT idE, nomE
FROM Élève e **INNER JOIN** Candidature c
ON e.idE = c.idE **INNER JOIN**
Université u **ON** c.nomU = u.nomU
WHERE effectif > 10000 ;

- $T_1 \leftarrow$ Lire la table ÉLÈVE et sélectionner les tuples d'*effectif* > 10000
- $T_2 \leftarrow$ Joindre T_1 avec la table CANDIDATURE
- $T_3 \leftarrow$ Joindre T_2 avec la table ÉLÈVE
- $T_4 \leftarrow$ Projeter T_3 sur *idE*, *nomE*



Exemple d'espace de recherche - choix de plan





Espace de recherche contenant deux plans d'exécution. Quel est le meilleur plan (sans considérer d'autres informations) ?

Optimisation à base de règles

Application de règles pour transformer une expression de l'algèbre relationnelle en plan d'exécution optimisé :

- Entrée : arbre représentant l'expression à évaluer
- Sortie : plan d'exécution pour la requête

L'arbre passé en entrée est construit comme suit :

- Les noeuds internes de l'arbre sont les opérateurs de l'expression
- Chaque noeud a pour fils le ou les sous-arbres construits à partir de son ou ses arguments
- Les feuilles sont des relations de la bases de données

Règles de transformation

Idées :

- Effectuer les sélections le plus tôt possible
- Combiner les sélections et les produits cartésiens pour faire des jointures
- Combiner les séquences d'opérations unaires, comme les sélections et les projections
- Chercher les sous-expressions communes dans une expression

Pour cela, on exploite des identités remarquables de l'algèbre relationnelle

Lois sur les jointures et les produits

Commutativité :

$$E_1 \bowtie_C E_2 \equiv E_2 \bowtie_C E_1 \quad (1)$$
$$E_1 \bowtie E_2 \equiv E2 \bowtie E_1 \quad (2)$$
$$E_1 \times E_2 \equiv E_2 \times E_1 \quad (3)$$

Associativité :

$$E_1 \bowtie_C (E_2 \bowtie_D E_3) \equiv (E_1 \bowtie_C E_2) \bowtie_D E_3 \quad (4)$$
$$E_1 \bowtie (E_2 \bowtie E_3) \equiv (E_1 \bowtie E_2) \bowtie E_3 \quad (5)$$
$$E_1 \times (E_2 \times E_3) \equiv (E_1 \times E_2) \times E_3 \quad (6)$$

Lois sur les projections et les sélections

Cascade de projections :

$$\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_k}(E)) \equiv \pi_{A_1,\ldots,A_n}(E) \quad (7)$$

Cascade de sélections :

$$\sigma_C(\sigma_D(E)) \equiv \sigma_{C\wedge D}(E) \quad (8)$$
$$\sigma_C(\sigma_D(E)) \equiv \sigma_D(\sigma_C(E)) \quad (9)$$

Permutations de projections et sélections :

- Si C ne porte que sur des attributs parmi $A_1, \ldots, A_n$:

$$\pi_{A_1,\ldots,A_n}(\sigma_C(E)) \equiv \sigma_C(\pi_{A_1,\ldots,A_n}(E)) \quad (10)$$

Lois sur les sélections et les autres opérations

Sélections et produits cartésiens/jointures :

- Si C ne porte que sur les attributs de E_1 :

$$\sigma_C(E_1 \times E_2) \equiv \sigma_C(E_1) \times E_2 \quad (11)$$

$$\sigma_{C \wedge D}(E_1 \times E_2) \equiv \sigma_D(\sigma_C(E_1) \times E_2) \quad (12)$$

- Si C ne porte que sur les attributs de E_1 et D sur les attributs de E_2 :

$$\sigma_{C \wedge D}(E_1 \times E_2) \equiv \sigma_C(E_1) \times \sigma_D(E_2) \quad (13)$$

Sélections et union / différence :

$$\sigma_C(E_1 \cup E_2) \equiv \sigma_C(E_1) \cup \sigma_C(E_2) \quad (14)$$

$$\sigma_C(E_1 - E_2) \equiv \sigma_C(E_1) - \sigma_C(E_2) \quad (15)$$

Projection et autres opérations

Projection et autres opérations :

- Produit cartésien :

$$\pi_{A_1,\ldots,A_i,\ldots,A_n}(E_1 \times E_2) \equiv \pi_{A_1,\ldots,A_i}(E_1) \times \pi_{A_{i+1},\ldots,A_n}(E_2) \qquad (16)$$

- Union :

$$\pi_{A_1,\ldots,A_n}(E_1 \cup E_2) \equiv \pi_{A_1,\ldots,A_n}(E_1) \cup \pi_{A_1,\ldots,A_n}(E_2) \qquad (17)$$

Algorithme d'application des règles

1. Utiliser (8) pour transformer les sélections $\sigma_{C_1 \wedge \ldots \wedge C_n}(E)$ en cascades $\sigma_{C_1}(\ldots \sigma_{C_n}(E))$

2. Pour chaque sélection, utiliser (9), (10), (11), (12), (13), (14) et (15) pour pousser les sélections en bas de l'arbre

3. Pour chaque projection, utiliser les règles (7), (16), (17) et (10) pour pousser la projection au plus bas dans l'arbre. Éliminer une projection qui conserve tous les attributs

4. Utiliser les règles (7), (8) et (10) pour combiner les cascades de sélections et projections en une sélection unique, une projection unique, ou une sélection suivie d'une projection

Algorithme d'application des règles (2)

5. Partitionner l'arbre résultant en groupes comme suit :
   - Créer un groupe par noeud binaire ($\times, \cup, \cap, \setminus, \bowtie$)
   - Le groupe inclut tous les ancêtres unaires intermédiaires
   - Le groupe inclut toute branche étiquetée par des opérateurs unaires et se terminant à une feuille (table)

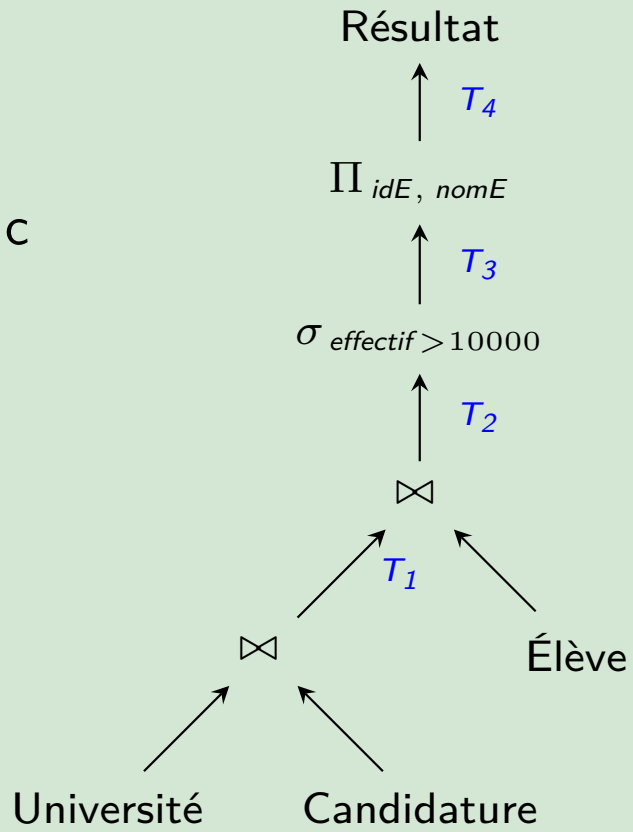
6. Chaque groupe correspond à une étape. Les étapes sont à évaluer dans n'importe quel ordre tant qu'un groupe est évalué après ses descendants

Exemple d'application des règles

L'identifiant et le nom des élèves qui ont candidaté dans des universités avec un effectif supérieur à 10000

SELECT DISTINCT idE, nomE
FROM Élève e **INNER JOIN** Candidature c
ON e.idE = c.idE **INNER JOIN**
Université u **ON** c.nomU = u.nomU
WHERE effectif > 10000 ;

⇒ Plan non optimisé !

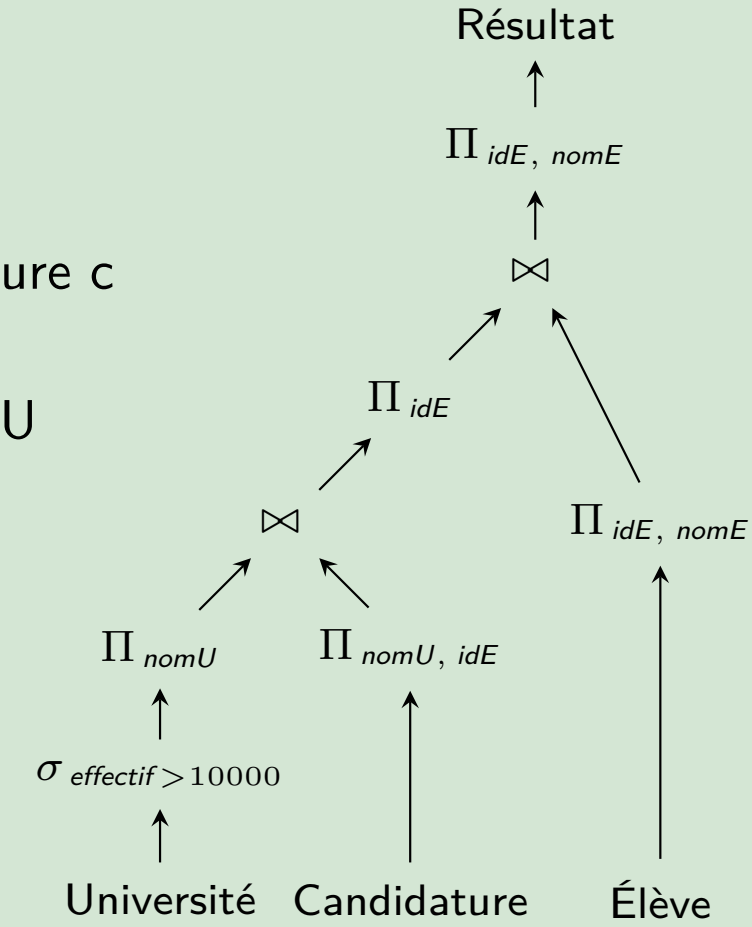


Exemple d'application des règles (2)

L'identifiant et le nom des élèves qui ont candidaté dans des universités avec un effectif supérieur à 10000

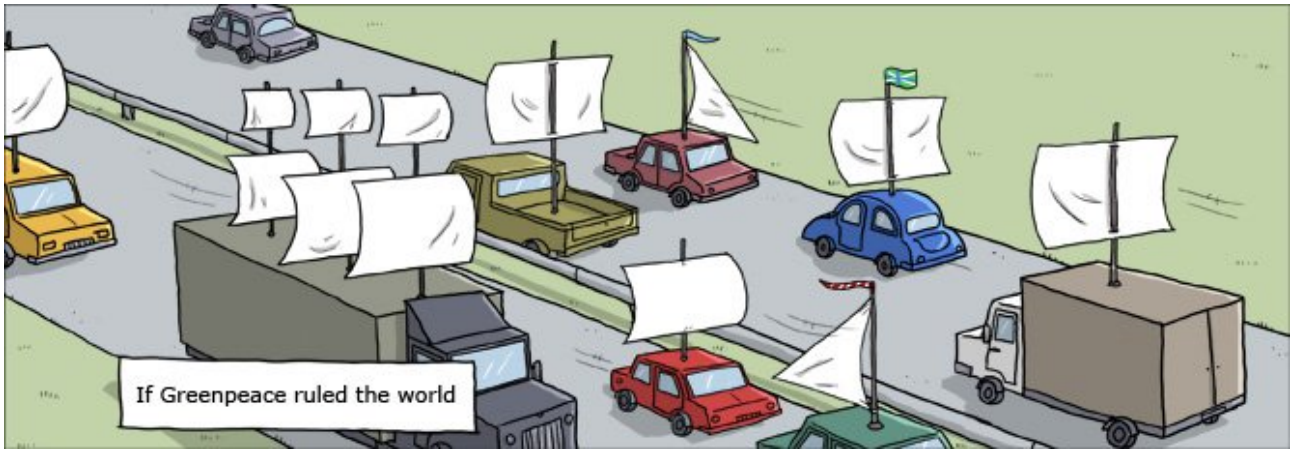
SELECT DISTINCT idE, nomE
FROM Élève e **INNER JOIN** Candidature c
ON e.idE = c.idE **INNER JOIN**
Université u **ON** c.nomU = u.nomU
WHERE effectif > 10000 ;

⇒ Plan optimisé par les règles :
(11), (10), (16)



En résumé

- Optimisation à base de règles = application de règles de transformation entre les opérateurs de l'AR
- Une quinzaine de règles et un algorithme à suivre
- Idée générale : placer les sélections et les projections en bas de l'arbre (le plus tôt possible)



http://wumo.com/

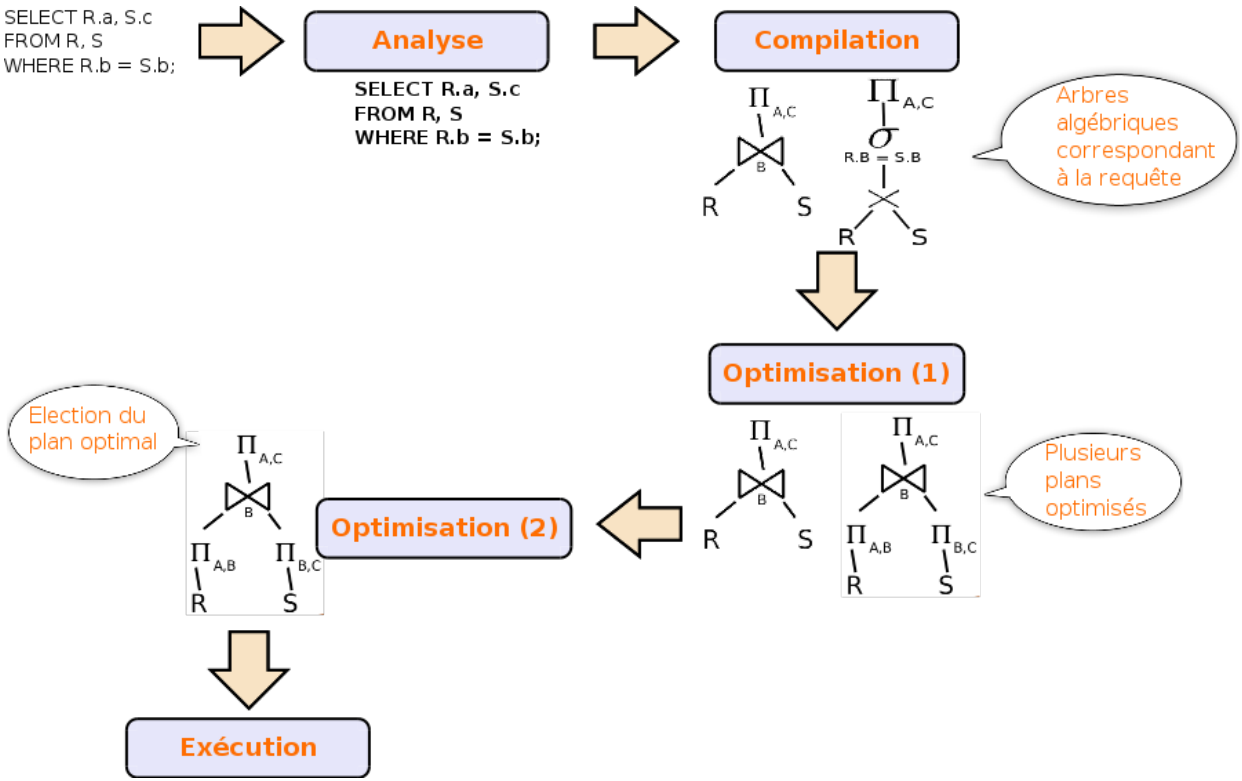
Plan

Traitement d'une requête SQL

Optimisation à base de règles

Estimation du coût d'un plan

Motivation



Pour sélectionner un plan, besoin d'estimer le coût de chaque plan optimisé algébriquement en fonction des algorithmes utilisés pour chaque opérateur

Temps d'accès

Les algorithmes manipulent les données sur disque et en mémoire

	Capacité	**Débit**	**Temps d'accès**
Disque dur	quelques To	100 Mo/s	$\sim$ 5 ms
Mémoire RAM	quelques Go	1 Go/s	$\sim$ 10 ns

Important d'évaluer le coût des lectures, notamment sur disque

Types d'algorithmes

Trois catégories d'algorithmes pour les différents opérateurs d'un plan d'exécution :

- Par itération (analyse de chaque n-uplet)
- Par partitionnement (e.g., par tri ou par fonction de hachage)
- Basé sur les index

Un SGBD sélectionne un algorithme pour chaque opérateur selon les métadonnées du catalogue (e.g., type d'index, cardinalité, statistiques sur les tables)

Les index seront vus en L3 (cours avancé de BD)
http://fr.wikipedia.org/wiki/Fonction_de_hachage
http://fr.wikipedia.org/wiki/Index_(base_de_donn%C3%A9es)

Algorithmes pour la projection

Pour rappel, la projection consiste à filtrer des attributs

Mais besoin de manipuler les n-uplets en cas de suppression des doublons (DISTINCT) :

- Algorithme de partitionnement basé sur un tri (*sorting*)
  - en cas de nombreux doublons
  - en cas de distribution peu uniforme des valeurs
  - résultat déjà trié
- Algorithme de partitionnement basé sur une fonction de hachage (*hashing*)
  - meilleures performances que le tri, mais jusqu'à une certaine limite

Algorithmes pour la projection (2)

```
1 Entrée : une relation R
2 Sortie : une relation S
3 pour chaque n-uplet r de R // parcours de R
4   stocker r dans S (uniquement attributs projetés)
5 trier S sur tous ses attributs
6 pour chaque n-uplet s de S // parcours de S
7   x = successeur(s) // x est le successeur de s
8   si x == s // x et s sont des doublons
9     supprimer s // suppression de s
```

Pseudo-code de l'algorithme de projection (avec suppression des doublons) basé sur un tri

Algorithmes pour la sélection

Pour rappel, une sélection consiste à filtrer des n-uplets et sa condition utilise des expressions de la forme "`attribut` **opérateur** *valeur*"

- Algorithme d'index avec fonction de hachage (*hash index*)
  - meilleures performances
  - uniquement si un index existe pour l'attribut, et si l'opérateur est une égalité
- Algorithme d'index *arbre B*
  - si un index existe pour l'attribut, et quand l'opérateur n'est pas une égalité
- Algorithme d'itération (*scan*)
  - si la relation est triée, optimisation par recherche binaire
  - les optimiseurs sont peu efficaces pour optimiser des conditions avec disjonctions (`OR`)

Algorithmes pour la sélection (2)

```
1 Entrée : une relation R, une condition "R.att op val"
2 Sortie : une relation S
3 pour chaque n-uplet r de R // parcours de R
4   si r[R.att] op val // condition vérifiée
5     stocker r dans S
```

Pseudo-code de l'algorithme de sélection basé sur un scan

Sélectivité d'un attribut

Sélectivité d'un attribut : ratio entre le nombre de valeurs distinctes d'un attribut et le nombre total de valeurs.
Utilisé pour évaluer le coût de sélection des n-uplets dont un attribut vaut une valeur spécifique

$$\text{Sélectivité}(att) = \frac{d_{att}}{n_R}$$

- Un attribut att appartenant à une table R
- La table R possède n_R n-uplets (et donc n_R valeurs pour l'attribut att)
- Le nombre de valeurs distinctes pour l'attribut att est d_{att}
- La sélectivité d'un attribut qui est clé primaire vaut 1

Algorithmes pour la jointure

Pour rappel, une jointure consiste à associer des n-uplets de deux tables R1 et R2 et sa condition utilise des expressions de la forme "`R1.attribut` **opérateur** `R2.attribut`"

- Algorithme de boucles imbriquées (*nested loops*)
  - algorithme basique
- Algorithme de boucles imbriquées avec bloc (*block nested loops*)
  - amélioration qui maximise l'utilisation des blocs mémoire
  - uniquement si un index existe pour l'attribut, et si l'opérateur est une égalité
- Algorithme de boucles imbriquées avec index (*index nested loops*)

Les algorithmes de jointure sont aussi applicables au produit cartésien

Algorithmes pour la jointure (2)

```
1 Entrée : une relation R1, une relation R2, une condition
      "R1.att op R2.att"
2 Sortie : une relation S
3 pour chaque n-uplet r1 de R1 // parcours de R1
4   pour chaque n-uplet r2 de R2 // parcours de R2
5     si r1[R1.att] op r2[R2.att] // condition vérifiée
6       stocker (r1, r2) dans S
```

Pseudo-code de l'algorithme de jointure basé sur des boucles imbriquées

Algorithmes pour la jointure (3)

Nombreux travaux pour améliorer les performances des algorithmes de jointure (opérateur le plus coûteux)

- Algorithme de tri fusion (*sort merge*)
  - les deux tables sont triées sur leur(s) attribut(s) de jointure
  - les tables sont ensuite parcourues par "intervalle linéaire" et les n-uplets partageant les mêmes valeurs pour leur(s) attribut(s) de jointure sont assemblés
- Algorithme basé sur une fonction de hachage (*hash join*)
  - la plus petite table est hachée sur ses attributs de jointure
  - pour chaque n-uplet de la grande table, vérification d'une correspondance dans la table de hachage

---

http://en.wikipedia.org/wiki/Sort-merge_join
http://en.wikipedia.org/wiki/Hash_join

Algorithmes pour la jointure (4)

```
1  Entrée : une relation R1, une relation R2, une condition
       "R1.att op R2.att"
2  Sortie : une relation S
3  pour chaque n-uplet r1 de R1 // parcours de R1
4    clé_r1 = hacher(r1[R1.att]) // attribut de jointure
5    stocker (clé_r1, r1) dans la table de hachage H
6  pour chaque n-uplet r2 de R2 // parcours de R2
7    clé_r2 = hacher(r2[R2.att]) // attribut de jointure
8    corresp = retrouver(H, clé_r2) // corresp vaut un n-
       uplet de r1 ou NULL
9    si corresp != NULL // un n-uplet correspond à r2
10     stocker (corresp, r2) dans S
```

Pseudo-code de l'algorithme de jointure basé sur une fonction de hachage

Algorithmes pour autres opérateurs

Produit cartésien :

- Équivalent à une jointure sans condition

Intersection (INTERSECT) :

- Équivalent à une jointure dont la condition est une égalité entre les valeurs de tous les $i^{èmes}$ attributs

Union et différence (UNION et MINUS) :

- Algorithme de tri
- Algorithme basé sur une fonction de hachage

Algorithmes pour autres opérateurs (2)

```
1  Entrée : une relation R1, une relation R2
2  Sortie : une relation S
3  trier R1 sur tous ses attributs
4  trier R2 sur tous ses attributs
5  indiceR2 = 0
6  pour chaque n-uplet r1 de R1 // parcours de R1
7    tant que (r1) > (R2[indiceR2]) // avancement dans R2
8      indiceR2 = indiceR2 + 1
9    si (r1) != (R2[indiceR2]) // n-uplets différents
10     stocker (r1) dans S
```

Pseudo-code de l'algorithme de différence basé sur le tri

Algorithmes pour autres opérateurs (3)

Fonctions d'agrégation (e.g., COUNT, AVG, MIN) :

- Par itération (scan de la table, en conservant l'information recherchée, e.g., la plus petite valeur courante)

Regroupements (GROUP BY) :

- Algorithme de tri
- Algorithme basé sur une fonction de hachage

```
1 Entrée : une relation R, un attribut R.att
2 Sortie : une relation S
3 nombreValeurs = 0
4 pour chaque n-uplet r de R // parcours de R
5   si r[R.att] != NULL // valeur de l'attribut non nulle
6     nombreValeurs = nombreValeurs + 1
```

Pseudo-code de l'algorithme d'agrégation basé sur un scan

Estimation du coût d'un plan

Le coût d'un plan est la somme des coûts de ses opérateurs

Comment estimer le coût de l'algorithme de boucles imbriquées avec bloc ?

- Soient deux relations $R1$ et $R2$ contenant n_{R1} et n_{R2} n-uplets
- Un bloc disque peut contenir b_{R1} n-uplets de $R1$ ou b_{R2} n-uplets de $R2$
- Nombre de blocs pour $R1$ et $R2$: $k_{R1} = \frac{n_{R1}}{b_{R1}}$ et $k_{R2} = \frac{n_{R2}}{b_{R2}}$
- On suppose que la relation la plus petite est $R1$ ($k_{R1} < k_{R2}$)
- m blocs peuvent tenir en mémoire centrale

formule de coût $= \frac{k_{R1}}{m-1} \times (m - 1 + k_{R2})$ **lectures de bloc**

Estimation du coût d'un plan (2)

Intuitions pour le coût :

- On garde $m-1$ blocs de $R1$ en mémoire, et le bloc libre sert à lire les n-uplets de $R2$
- Il faut faire $\frac{k_{R1}}{m-1}$ fois lire $m-1$ blocs de $R1$ et k_{R2} blocs de $R2$

Illustration : avec $k_{R1} = 6$, $k_{R2} = 8$, et $m = 4$ blocs en mémoire, il faut $\frac{6}{4-1} \times (4 - 1 + 8) = 22$ lectures de bloc

⇒ 3 lectures de blocs pour $R1$ (fixés) et 8 lectures pour chaque bloc de $R2$

⇒ 3 lectures de blocs pour $R1$ (fixés) et 8 lectures pour chaque bloc de $R2$

Exemple d'estimation du coût d'un plan

- Relation R, 20000 n-uplets, 100 n-uplets/bloc $\Rightarrow k_R = 200$
- Relation S, 100000 n-uplets, 100 n-uplets/bloc $\Rightarrow k_S = 1000$
- Nombre de blocs en mémoire : 100
- Nombre de blocs lus par seconde : $n_{bps} = 20$

Quel est le temps d'accès en blocs et en secondes pour la jointure "boucles imbriquées avec bloc" entre R et S ?

Nombre de blocs $= \frac{200}{99} \times (99 + 1000) \approx 2220$ blocs

Temps $= \frac{2220}{20} \approx 111$ secondes

Et si une sélection filtre 50% de R avant jointure ?

Réduction à 1310 blocs (200 pour la sélection et 1110 pour la jointure), soit 65 secondes

En résumé

- Pour chaque opérateur, choix d'un algorithme efficace
- Calcul du coût selon le nombre d'accès aux blocs contenant les n-uplets
- L'estimation du coût donne les temps I/O et CPU de chaque plan ($\Rightarrow$ élection du plan optimal)