INFORMATIQUE Développement d'applications
BLOC 1
UE05 Bases de données

Chapitre 4.1 : Optimisation

Vincent Reip

Février 2023

Objectif

- Au terme de ce chapitre, l'étudiant sera capable de :
  - comprendre et expliquer les bases de l'optimisation de l'exécution des requêtes au sein d'un SGBD

Optimisation des requêtes : introduction

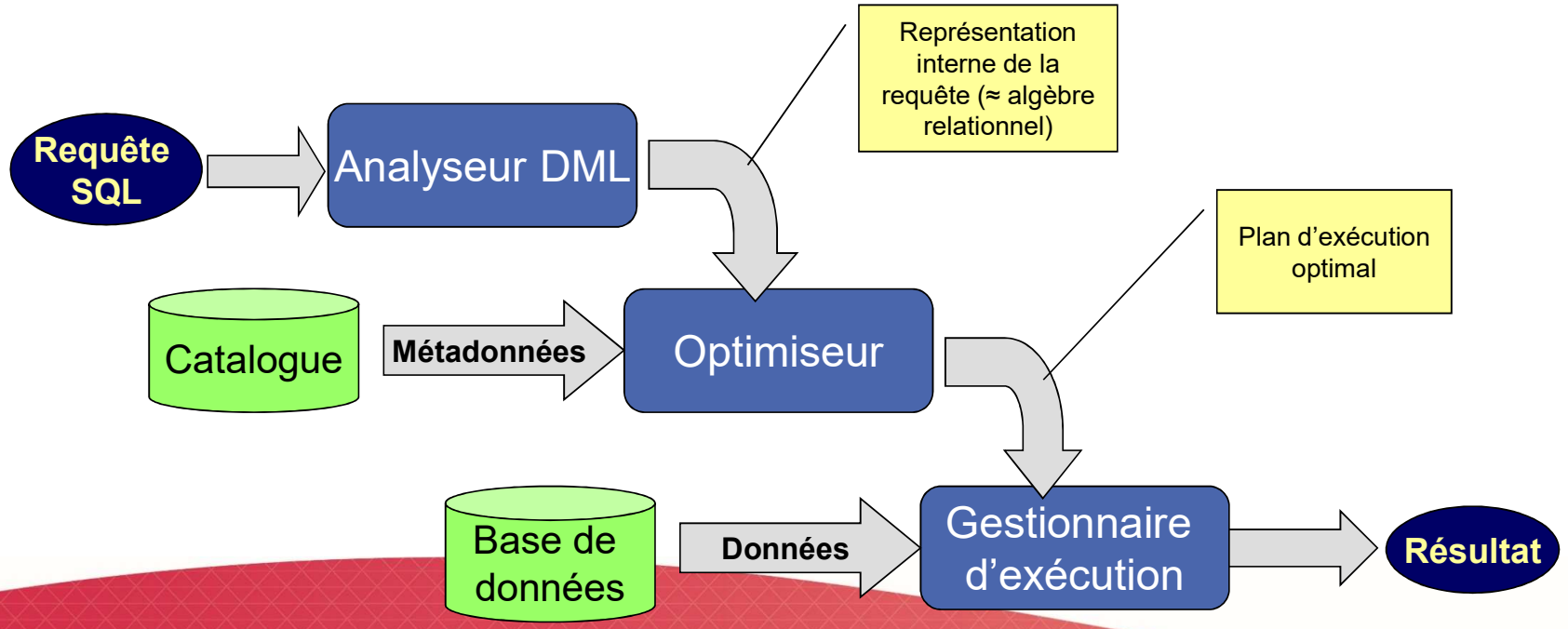
- L'optimisation des requêtes est souhaitée pour obtenir des performances acceptables même avec des volumes de données importants.
- La « correspondance » entre le langage SQL et l'algèbre relationnelle permet de se baser sur des règles (mathématiques) précises pour établir l'équivalence d'une requête optimisée avec la requête originale.
  - On cherche à minimiser le nombre nécessaire d'opérations à effectuer pour obtenir le résultat d'une requête
  - L'effort (temps) d'optimisation ne doit pas annihiler le gain (temps) résultant de l'optimisation

Optimisation des requêtes : introduction

- L'optimisation est déléguée au SGBD (plutôt qu'à l'utilisateur)
  - L'utilisateur ne doit donc généralement pas (trop) se préoccuper de l'optimisation lors de l'écriture de la requête
  - Le programme d'optimisation bénéficie d'informations statistiques sur les tables (nombre de tuples, nombre de valeurs distinctes pour un attribut, nombre de répétition d'une même valeur pour un attribut…) qui sont stockées dans le catalogue du SGBD
  - Le programme d'optimisation a connaissance du schéma interne, du schéma physique, des contraintes et de la présence d'index sur certains attributs (aussi stockés dans le catalogue)
  - Le programme d'optimisation pourra « courageusement » explorer une multitude de scénarii pour déterminer la meilleure marche à suivre

Optimisation des requêtes : introduction

- Le SGBD traite les requêtes SQL en trois étapes :
  - L'analyse et la traduction
  - L'optimisation
  - L'exécution



Optimisation des requêtes : introduction

- Analyse
  - Vérification de la syntaxe
  - Écriture de la requête en une représentation interne (souvent un arbre de requête algébrique)
- Optimisation
  - Réécriture en requêtes sémantiquement équivalentes (en exploitant les propriétés de l'algèbre relationnelle)
  - Obtention de plans d'exécutions de différents coûts
  - Sélection du plan de coût minimum (critères de coût : E/S sur disque et utilisation CPU)
- Exécution de la requête
  - Le meilleur plan est soumis au gestionnaire d'exécution (machine algébrique) qui l'exécute et produit le résultat escompté

Plan d'exécution

- Un plan d'exécution est une séquence d'opérations élémentaires dont l'exécution débouche sur le résultat souhaité d'une requête.
- Les SGBD fournissent des outils permettant de visualiser les plans d'exécution et le coût associé
  - EXPLAIN PLAN FOR …
  - Le plan d'exécution sera stocké dans une table spécifique qui pourra à son tour être interrogée.

Plan d'exécution

Home > SQL > SQL Commands

Autocommit Display 10

```
SELECT v.nom, v.prenom, a.statut, m.texte
FROM Vendeur v
JOIN Alerter a ON v.id_vendeur = a.id_vendeur
JOIN Message m ON a.id_message = m.id_message
WHERE a.statut = 'N'
```

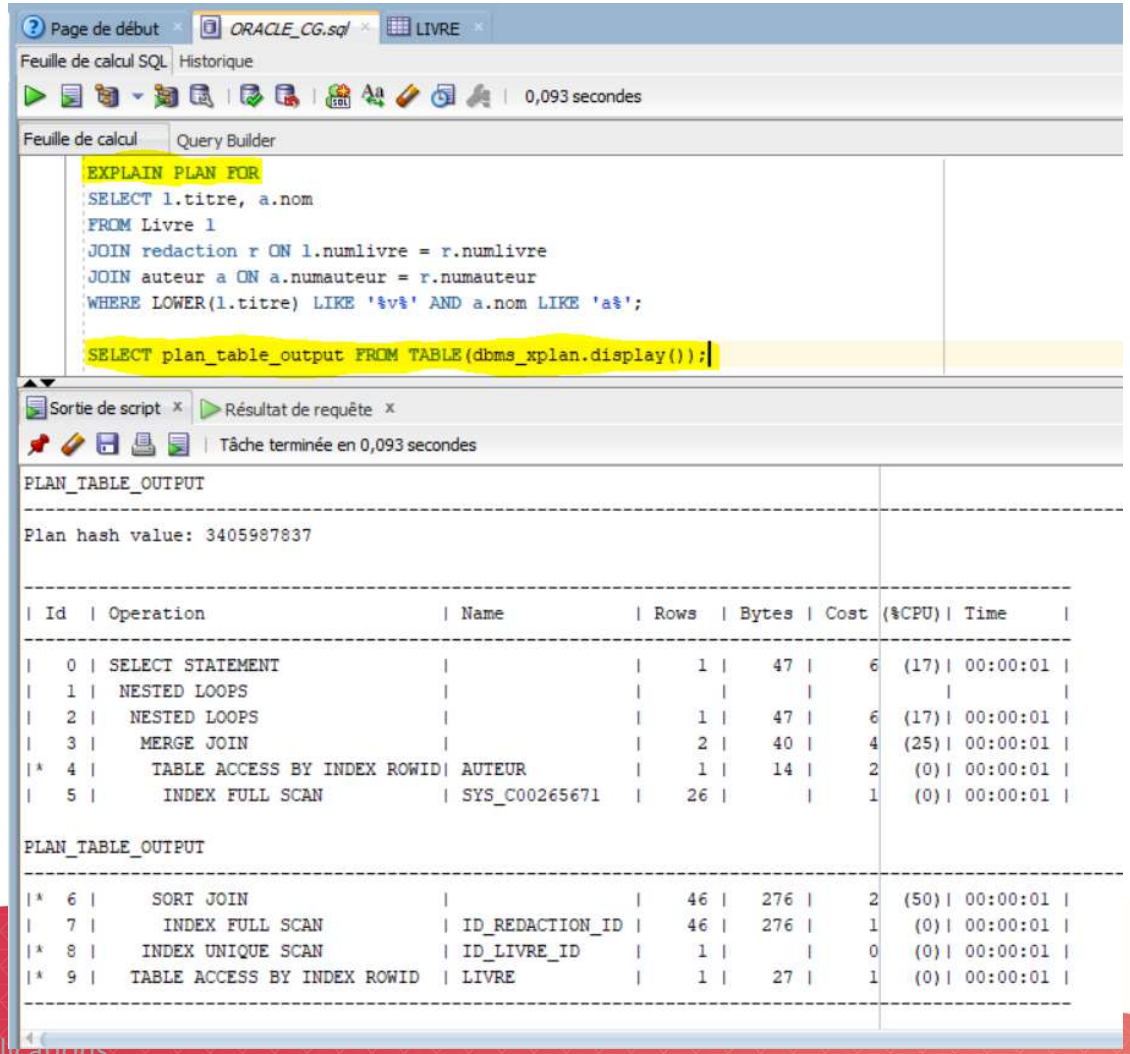
Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			36	1	5	2 232		
NESTED LOOPS			36	1	5	2 232		
NESTED LOOPS			36	1	4	1 692		
TABLE ACCESS	FULL	ALERTER	36	1	3	180	"A""STATUT" = 'N'	
TABLE ACCESS	BY INDEX ROWID	MESSAGE	1	1	1	42		
INDEX	UNIQUE SCAN	PK_MESSAGE	1	1	0			"A"."ID_MESSAGE" = "M"."ID_MESSAGE"
TABLE ACCESS	BY INDEX ROWID	VENDEUR	1	1	1	15		
INDEX	UNIQUE SCAN	PK_VENDEUR	1	1	0			"V"."ID_VENDEUR" = "A"."ID_VENDEUR"

* Unindexed columns are shown in red

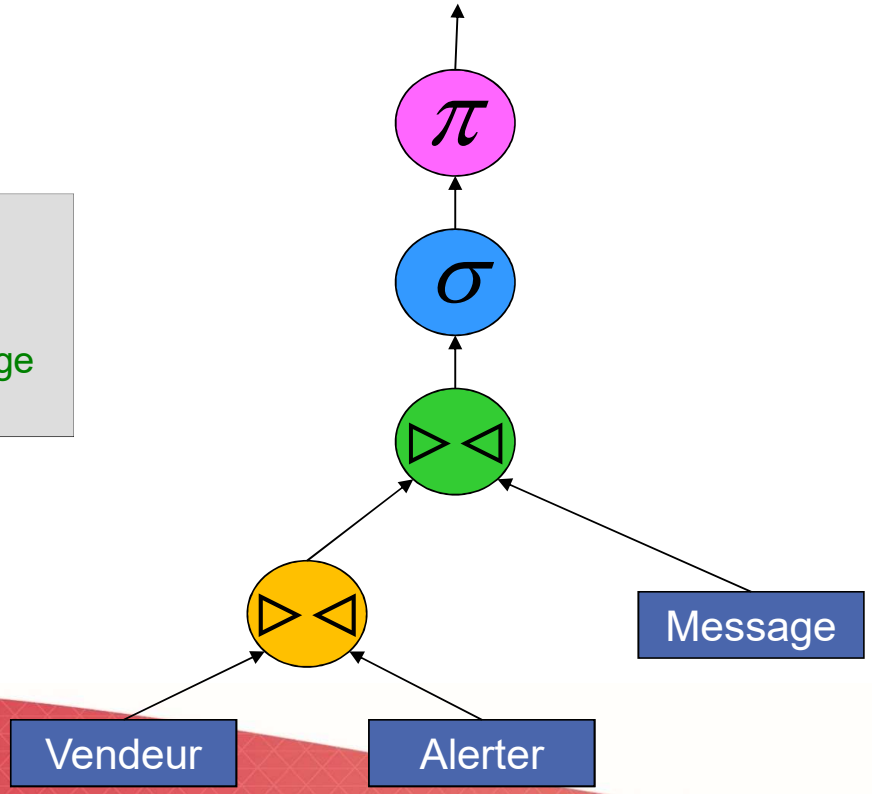
Plan d'exécution



Plan d'exécution

- Un plan d'exécution est souvent représenté (en interne) sous forme d'un arbre d'opérations

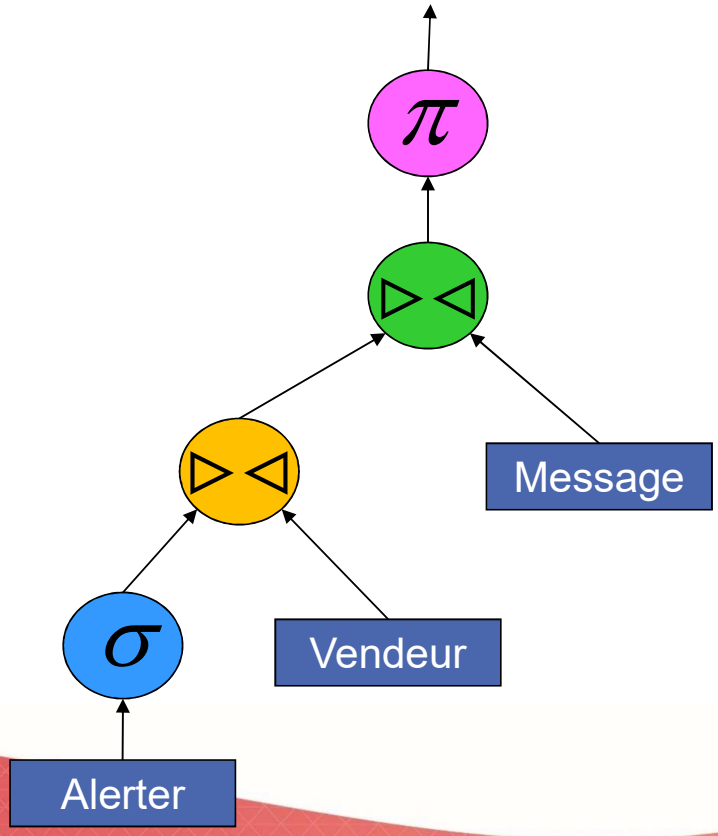
```
SELECT v.nom, v.prenom
FROM Vendeur v
JOIN Alerter a ON a.id_vendeur = v.id_vendeur
JOIN Message m ON a.id_message = m.id_message
WHERE a.statut = 'N'
```



Plan d'exécution (bis)

- Certaines opérations peuvent être permutées au sein de l'arbre tout en gardant une équivalence

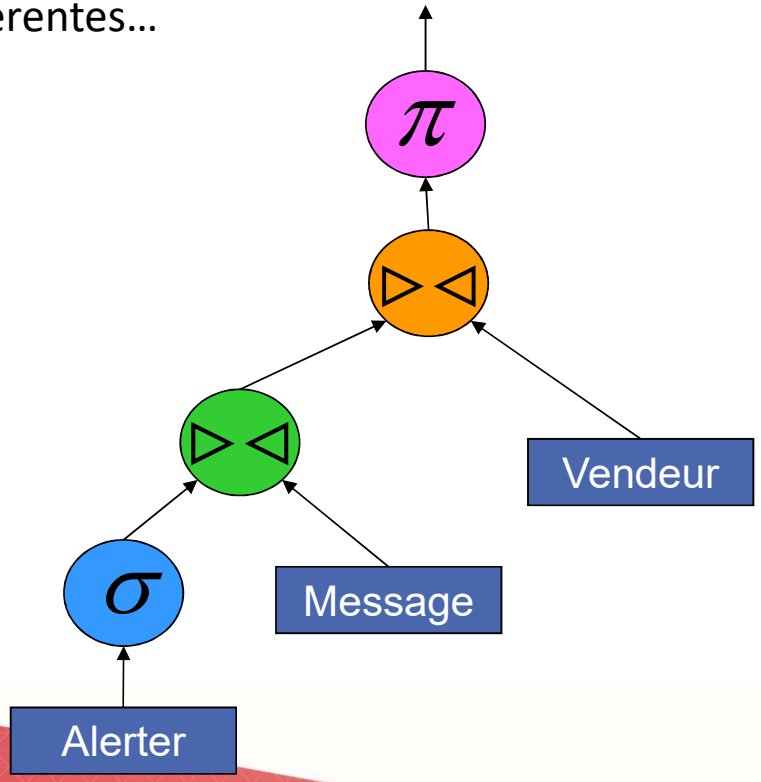
```
SELECT v.nom, v.prenom
FROM Vendeur v
JOIN Alerter a ON a.id_vendeur = v.id_vendeur
JOIN Message m ON a.id_message = m.id_message
WHERE a.statut = 'N'
```



Plan d'exécution (ter)

- En fonction de différents facteurs, certains plans d'exécution seront plus performants que d'autres
  - Nombre de tuples, index, nombre de valeurs différentes…

```
SELECT v.nom, v.prenom
FROM Vendeur v
JOIN Alerter a ON a.id_vendeur = v.id_vendeur
JOIN Message m ON a.id_message = m.id_message
WHERE a.statut = 'N'
```



Equivalence algébrique

Commutativité et associativité des jointures

$$T_1 \triangleright\triangleleft T_2 = T_2 \triangleright\triangleleft T_1$$

$$(T_1 \triangleright\triangleleft T_2) \triangleright\triangleleft T_3 = T_1 \triangleright\triangleleft (T_2 \triangleright\triangleleft T_3)$$

Optimiser le calcul des jointures multiples

Cascade de projections

$$\pi_{c1,c2...cm}(\pi_{c1,c2,..,cn}(T)) = \pi_{c1,c2...cm}(T) \wedge m < n$$

Minimise le nombre de projections

Cascade de sélections

$$\sigma_{C1}(\sigma_{C2}(T)) = \sigma_{C1 \wedge C2}(T)$$

Remplacer plusieurs sélections par une seule ou au contraire la décomposer pour exploiter un index

Equivalence algébrique

Commutativité sélection et projection
dans le cas où C porte uniquement sur $c_1, \ldots, c_n$

$$\pi_{c1,c2\ldots cn}(\sigma_C(T)) = \sigma_C(\pi_{c1,c2\ldots cn}(T))$$

Permet d'appliquer la sélection sur une relation plus petite.

Commutativité sélection et projection
dans le cas où C porte sur $c_1, \ldots, c_n$ et d'autres colonnes $b_1, \ldots, b_m$

$$\pi_{c1,c2\ldots cn}(\sigma_C(T)) = \pi_{c1,c2\ldots cn}(\sigma_C(\pi_{c1,c2\ldots cn,b1\ldots,bm}(T)))$$

Permet d'appliquer la sélection sur une relation plus petite (mais qui contient les attributs nécessaires à l'évaluation de la condition de sélection).

Equivalence algébrique

Commutativité sélection et jointure, produit cartésien, union ou différence

$$\sigma_C(E_1 \triangleright\triangleleft E_2) = \sigma_C(E_1) \triangleright\triangleleft \sigma_C(E_2)$$

$$\sigma_C(E_1 \mathrm{X} E_2) = \sigma_C(E_1) \mathrm{X} \sigma_C(E_2)$$

$$\sigma_C(E_1 \cup E_2) = \sigma_C(E_1) \cup \sigma_C(E_2)$$

$$\sigma_C(E_1 - E_2) = \sigma_C(E_1) - \sigma_C(E_2)$$

Permet de réduire la taille des relations avant d'appliquer les opérations de jointure, union, différence.

Commutativité projection et produit cartésien ou union

$$\pi_{c1,c2\ldots cn}(E_1 \mathrm{X} E_2) = \pi_{c1,c2\ldots cn}(E_1) \mathrm{X} \pi_{c1,c2\ldots cn}(E_2)$$

$$\pi_{c1,c2\ldots cn}(E_1 \cup E_2) = \pi_{c1,c2\ldots cn}(E_1) \cup \pi_{c1,c2\ldots cn}(E_2)$$

Equivalence logique

- Certaines conditions peuvent être simplifiées en faisant appel aux propriétés de la logique
  - Lois de Morgan :
    - NOT(A OR B) = NOT A AND NOT B
    - NOT(A AND B) = NOT A OR NOT B
  - Loi d'absorption logique :
    - A OR (A AND B) = A
    - A AND (A OU B) = A
  - Loi d'adjacence logique :
    - (A AND B) OR (A AND NOT B) = A
    - (A OR B) AND (A OR NOT B) = A
- Exercice : simplifiez la requête suivante
  - SELECT *
    FROM Vins
    WHERE (degre=12 OR cru='Morgon' OR cru='Chenas')
    AND NOT(cru='Morgon' OR cru='Chenas');

Equivalence logique

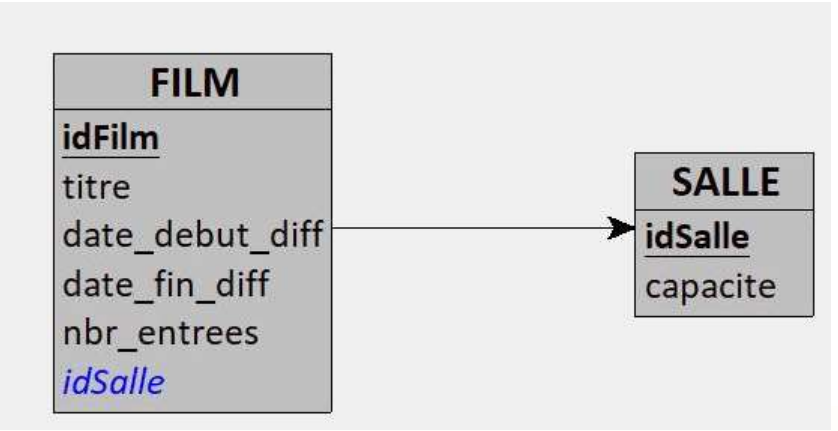
		Degré=12	cru='Morgon'	cru='Chenas'	A OR B OR C	NOT (cru='Morgon' OR cru='Chenas')	
Degré	**Cru**	A	B	C	D	E	D AND E
12	Morgon	1	1	0	1	0	0
12	Chenas	1	0	1	1	0	0
12	**Beaujolais**	1	0	0	1	1	1
14	Morgon	0	1	0	1	0	0
14	Chenas	0	0	1	1	0	0
14	Beaujolais	0	0	0	0	1	0

		Degré=12	NOT (cru='Morgon' OR cru='Chenas')	
Degré	**Cru**	A	E	A AND E
12	Morgon	1	0	0
12	Chenas	1	0	0
12	**Beaujolais**	1	1	1
14	Morgon	0	0	0
14	Chenas	0	0	0
14	Beaujolais	0	1	0

```
SELECT *
FROM Vins
WHERE degre=12 AND NOT(cru='Morgon' OR cru='Chenas');
```

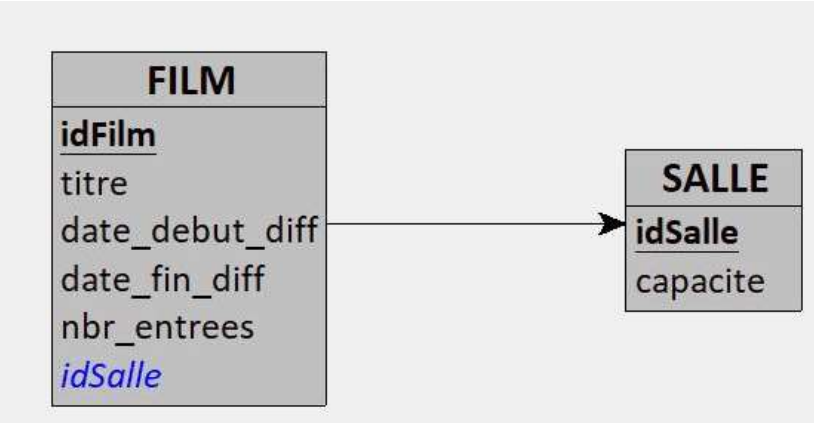
Exemple

- Soient les relations
  - FILMS(id_film, titre, date_debut_diff, date_fin_diff, *salle_de_diffusion*, nbr_entrees)
  - SALLES(id_salle, capacite)
  - *salle_de_diffusion* dans FILMS référence id_salle dans SALLES
- On suppose en outre que
  - FILMS contient 200 tuples
  - SALLES contient 7 tuples



Exemple

- On désire connaître les identifiants des films diffusés dans des salles dont la capacité est d'au moins 150 places
- Quelle sera la requête SQL correspondante ?




```
SELECT f.id_film
FROM FILMS f
JOIN SALLES s ON  f.salle_de_diffusion = s.id_salle
WHERE s.capacite >= 150
```

Exemple

- En algèbre relationnelle, on a plusieurs possibilités pour traduire cette requête SQL
- Chaque possibilité correspond à un plan d'actions différent

$$\pi_{id\_film}(films \triangleright\!\triangleleft_{salle\_diffusion=id\_salle} \sigma_{capacite>150}(salles))$$

$$\pi_{id\_film}(\sigma_{capacite\geq 150}(films \triangleright\!\triangleleft_{salle\_diffusion=id\_salle} salles))$$

Exemple

- On suppose que 2 salles ont une capacité >= 150 places

  1. Sélection → 7 E/S (2 salles) ;
     Jointure → 2*200 = 400 E/S
     →7+400 E/S

  2. Jointure → 7*200 = 1400 E/S ; (résultat : 200 films liés au 7 salles)
     Sélection → 200 E/S
     → 1600 E/S

- Conclusion : la première requête est plus efficace
- Comment le SGBD peut-il sélectionner la meilleure requête ?

NB: on considère dans cet exemple que l'accès à un tuple nécessite une E/S, qu'il n'y a pas d'index et que les algorithmes de groupement (jointure) utilise la « force brute » (c-à-d. que toutes les combinaisons possibles sont examinées). Il existe bien sûr des algorithmes plus performants (recherche indexée, tri et fusion, hachage…) dont l'étude sort du cadre de ce cours.

Limites de l'optimisation

- Problème :
  - Le nombre de plans d'exécution possibles peut être très important
  - Idéalement, l'optimisation doit prendre un temps négligeable par rapport à l'exécution effective de la requête
  - Utilisation de méthodes heuristiques(*) afin de limiter le nombre de plan d'exécution :
    - On effectuera toujours les sélections et les projections les plus tôt possible afin de réduire la tailles des relations à traiter

(*) En informatique, l'heuristique est une méthode de résolution de problèmes, non fondée sur un modèle formel et qui n'aboutit pas nécessairement à une solution. Elle procède par évaluations successives et hypothèses provisoires.

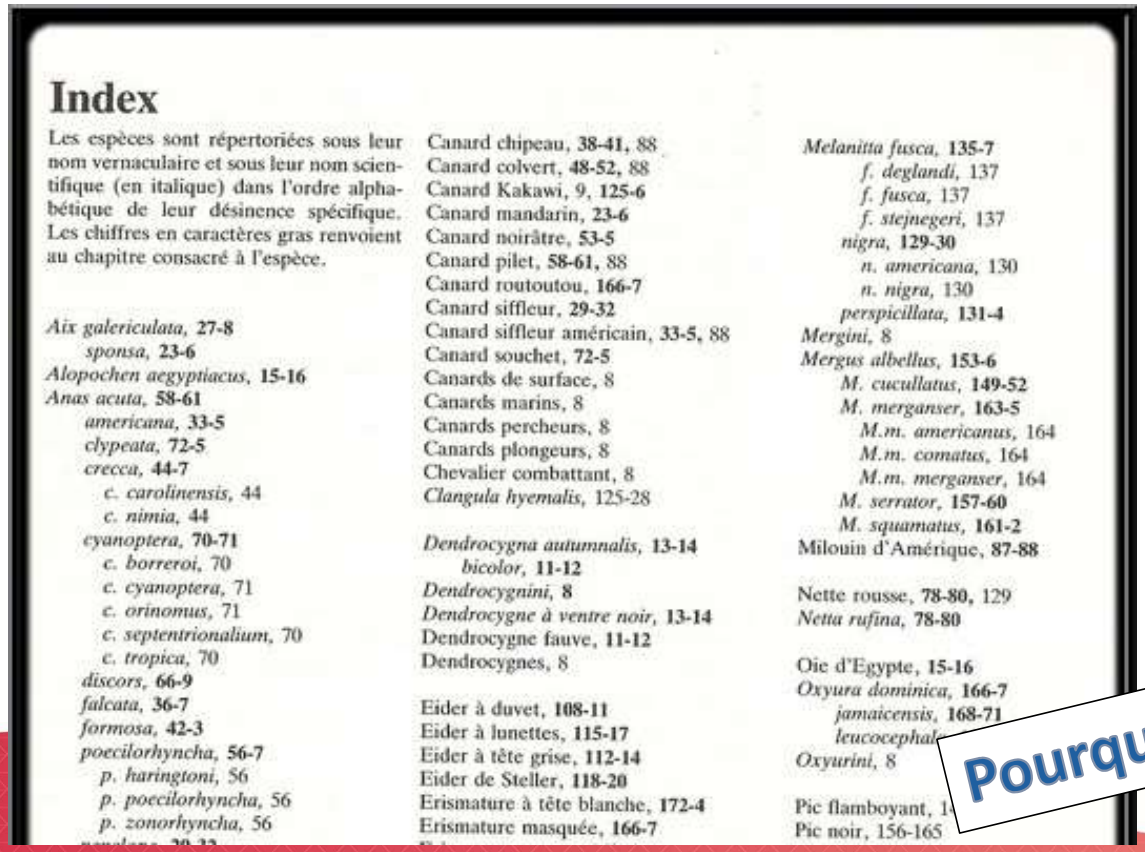
Les index

```
CREATE INDEX [nom index]
ON [nom relation(nom-attr *)]
```

- Un index est une structure de données ordonnée qui permet d'optimiser la recherche d'éléments.
- Un index est généralement créé automatiquement pour les clés primaires et les clés candidates
- Lors de la phase d'optimisation, le SGBD décidera d'employer (ou non) les index disponibles sur les tables impliquées dans une requête SQL
- En analysant un plan d'exécution, un utilisateur peut décider d'ajouter un/des index qui vont permettre une exécution plus rapide.

Les index

- Analogie : index d'un livre



Index

Les espèces sont répertoriées sous leur nom vernaculaire et sous leur nom scientifique (en italique) dans l'ordre alphabétique de leur désinence spécifique. Les chiffres en caractères gras renvoient au chapitre consacré à l'espèce.

Pourquoi est-ce efficace ?

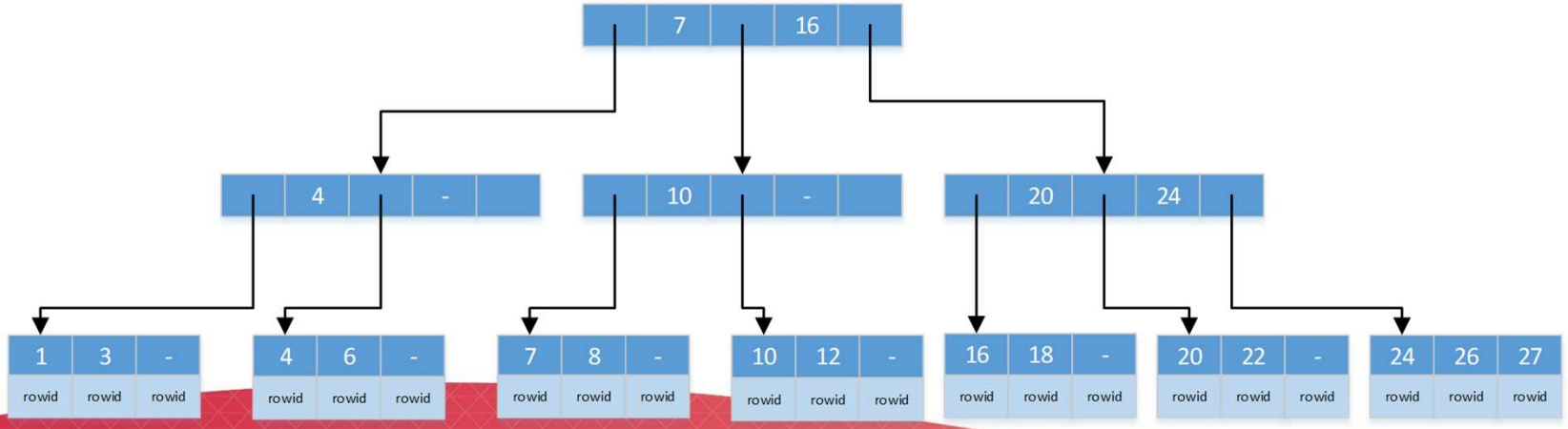
Les index

```
SELECT proprietaire
FROM parking
WHERE numPlace = 26
```

parking		
id	numPlace	Proprietaire
1	1	Tom Smith
2	3	Robert Marr
3	16	Johnny Bolan
4	10	Marc Barat
5	4	Karl Doherty
6	24	Pete Harvey
7	7	Mick Lennon
8	6	John Deal
9	27	Kim Cave
10	20	Nick Black
11	18	Franck Harry
12	8	Debbie Jones
13	22	Mick Page
14	26	Jimmy Wright
15	12	Shanon Morello

Les index B-Tree

- Type d'index par défaut sur Oracle
- Index stocké sous forme d'arbre ordonné et équilibré (Balanced-tree)
- La distance entre la racine et les feuilles est identique partout
- Les nœuds intermédiaires indiquent la manière dont les valeurs sont distribuées dans les feuilles
- Les feuilles contiennent toutes les valeurs de l'index ainsi que les 'rowid' des tuples concernés
  - rowid : méthode interne à Oracle pour accéder à un tuple

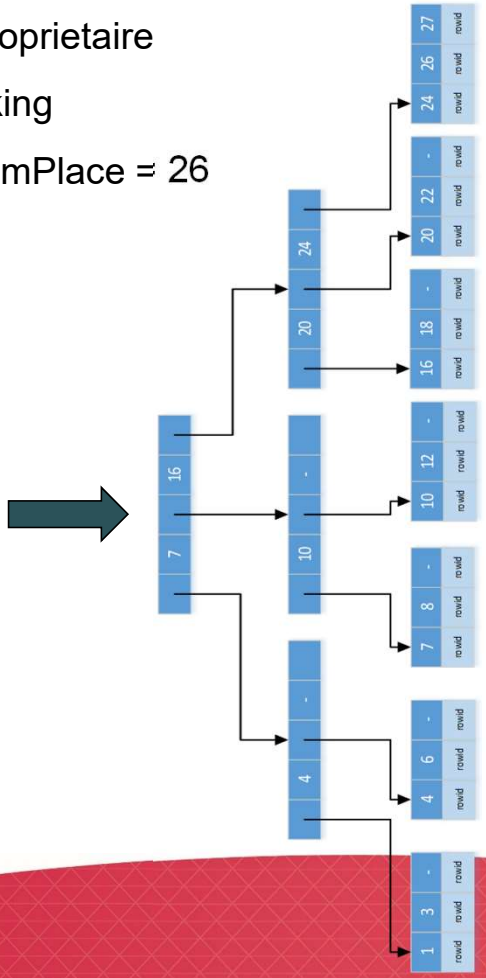


Les index B-Tree

SELECT proprietaire

FROM parking

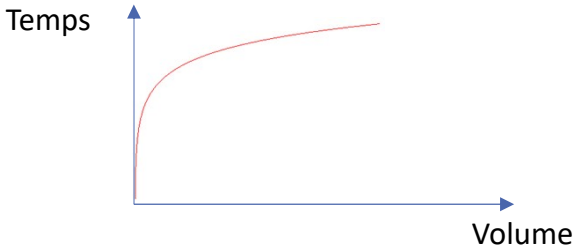
WHERE numPlace = 26



parking		
id	numPlace	Proprietaire
1	1	Tom Smith
2	3	Robert Marr
3	16	Johnny Bolan
4	10	Marc Barat
5	4	Karl Doherty
6	24	Pete Harvey
7	7	Mick Lennon
8	6	John Deal
9	27	Kim Cave
10	20	Nick Black
11	18	Franck Harry
12	8	Debbie Jones
13	22	Mick Page
14	26	Jimmy Wright
15	12	Shanon Morello

Les index B-Tree

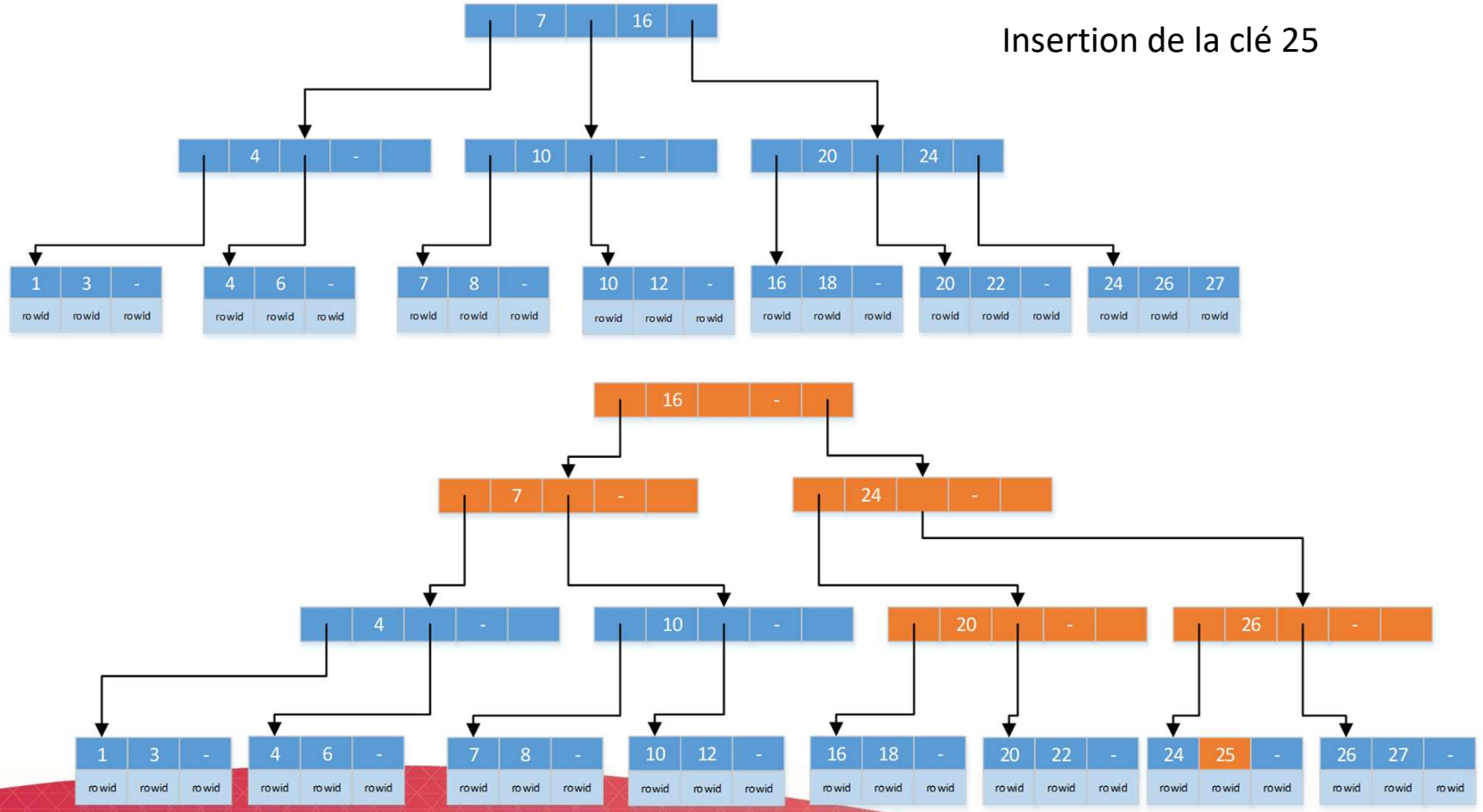
- Les index ne sont pas ‘gratuits’ en terme de performance
  - Chaque opération d’insertion ou de suppression nécessite un rééquilibrage de l’arbre
- Le B-TREE permet d’effectuer les opérations d’insertion et de suppression dans un temps d’ordre logarithmique
  - Même si la taille des données augmente considérablement, le temps nécessaire pour l’opération d’insertion ou de suppression dans l’index n’augmentera que lentement



- La recherche est elle aussi très efficace car la profondeur de l’arbre est minimisée (notamment en permettant aux nœuds de contenir plus de valeur)
  - Le nombre de lectures et comparaisons est déterminé par la profondeur de l’arbre

Insertion dans un index B-Tree

Insertion de la clé 25



Les index

```
DROP INDEX [nom index]
```

- Supprime un index

Les index

```
SELECT b.*, 'Vente gérée par ' || v.prenom || ' ' || v.nom
FROM Bien b
JOIN Gerer g ON b.id_bien = g.id_bien
JOIN Vendeur v ON g.id_vendeur = v.id_vendeur
WHERE b.code_postal = 4000
```

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			2	1	5	572		
NESTED LOOPS			2	1	5	572		
NESTED LOOPS			2	1	4	542		
TABLE ACCESS	FULL	BIEN	1	1	3	267	"B""CODE_POSTAL" = 4000	
INDEX	FULL SCAN	PK_GERER	1	1	1	4	"G"."ID_BIEN" = "B"."ID_BIEN"	"G"."ID_BIEN" = "B"."ID_BIEN"
TABLE ACCESS	BY INDEX ROWID	VENDEUR	1	1	1	15		
INDEX	UNIQUE SCAN	PK_VENDEUR	1	1	0			"G"."ID_VENDEUR" = "V"."ID_VENDEUR"

* Unindexed columns are shown in red

Total ‘Cost’ : 19

Les index

```
CREATE INDEX idx_code_postal
ON Bien(code_postal)
```

```
SELECT b.*, 'Vente gérée par ' || v.prenom || ' ' || v.nom
FROM Bien b
JOIN Gerer g ON b.id_bien = g.id_bien
JOIN Vendeur v ON g.id_vendeur = v.id_vendeur
WHERE b.code_postal = 4000
```

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			2	1	4	572		
NESTED LOOPS			2	1	4	572		
NESTED LOOPS			2	1	3	542		
TABLE ACCESS	BY INDEX ROWID	BIEN	1	1	2	267		
INDEX	RANGE SCAN	IDX_CODE_POSTAL	1	1	1			"B"."CODE_POSTAL" = 4000
INDEX	FULL SCAN	PK_GERER	1	1	1	4	"G"."ID_BIEN" = "B"."ID_BIEN"	"G"."ID_BIEN" = "B"."ID_BIEN"
TABLE ACCESS	BY INDEX ROWID	VENDEUR	1	1	1	15		
INDEX	UNIQUE SCAN	PK_VENDEUR	1	1	0			"G"."ID_VENDEUR" = "V"."ID_VENDEUR"

* Unindexed columns are shown in red

Total ‘Cost’ : 15

Optimisation des requêtes : conclusion

- L'optimisation est une partie indispensable d'un SGBD :
  - Evalue plusieurs plans d'actions pour une requête (utilisation des règles de l'algèbre relationnelle)
  - Sélectionne le meilleur plan d'actions
  - Minimise le temps d'attente pour le traitement d'une requête
  - Exploite intelligemment les ressources dont le SGBD dispose
    - Statistiques, index, contraintes…
  - Utilise des heuristiques (notamment pour limiter le nombre de plan d'exécution à étudier)

Catalogue d'une base de données

- Toute base de données est décrite par une base de données particulière : le catalogue (aussi appelé dictionnaire)
- Le catalogue décrit les objets que comportent la BD
  - Définition des tables, des vues, des index, des contraintes, des triggers, procédures stockées, des utilisateurs et de leurs droits…
- Le catalogue contient aussi des données statistiques (utilisation pour l'optimisation)

Catalogue d'une base de données

- Le catalogue est donc un ensemble de tables qui peuvent être elles-mêmes interrogées
- Quelques exemples (Oracle) :
  - SELECT * FROM USER_INDEXES
  - SELECT * FROM USER_TABLES
  - SELECT * FROM ALL_TABLES
  - SELECT * FROM ALL_USERS
  - SELECT * FROM ALL_TRIGGERS
  - SELECT * FROM ALL_VIEWS
  - SELECT * FROM USER_CONSTRAINTS
  - ….