

Cours d'Optimisation

Par Maggie LEKPA

1

Rappel

Dans le cas d'une base de données, la mémoire vive et les disques sont les niveaux à considérer.

Matériellement, une base de données est un ensemble de fichiers stockés sur un support non volatile (disque magnétique, SSD(solide state drive)) pour des raisons de taille (données volumineuses) et de persistance.

Les données peuvent être chargées dans la mémoire vive afin d'être traitées rapidement; cela implique le transfert de données entre le disque et la mémoire vive.

3

Rappel

Il existe plusieurs systèmes de stockage.

Mémoire volatile : elle permet de stocker les données et les processus constituant l'espace de travail du processeur; **la donnée doit y être stockée pour pouvoir être traitée rapidement**. C'est une mémoire à accès rapide.

Mémoire persistante (en ligne) : elle offre une grande capacité de stockage. Les accès en lecture et écriture sont relativement rapides.

Ex : disque magnétique, SSD(Solid State Drive)

Mémoire persistante (hors ligne): il s'agit des supports de sauvegarde. Ces dernières ont un accès en lecture/écriture très lent et un coût réduit.

2

Rappel

Ordre de grandeur :

mémoire vive d'un serveur de base de données : plusieurs dizaines de Go (des centaines pour de gros serveurs de bd)

Disque de stockage : ordre de téraoctet. En pratique toute la base de données ne peut être copiée dans la mémoire vive.

La mémoire vive ne contient alors qu'une fraction du disque.

4

OPTIMISATION

Pourquoi optimiser?

- Réduire le temps de réponse : objectif numéro 1; Les utilisateurs sont très sensibles aux problèmes de performance de leurs traitements (rapports, mise à jour, ...), dégradation des temps de réponse
- Diminuer les consommations de ressources (cpu, mémoire vive, disques durs)
- Répartir les charges

OPTIMISATION

Rappel :

SQL est déclaratif → Dans une requête SQL, l'utilisateur spécifie ce qu'il veut obtenir et ne dit pas comment l'obtenir → besoin d'un programme (**forme opératoire**).

OPTIMISATION

Considérons la table client ci-dessous :

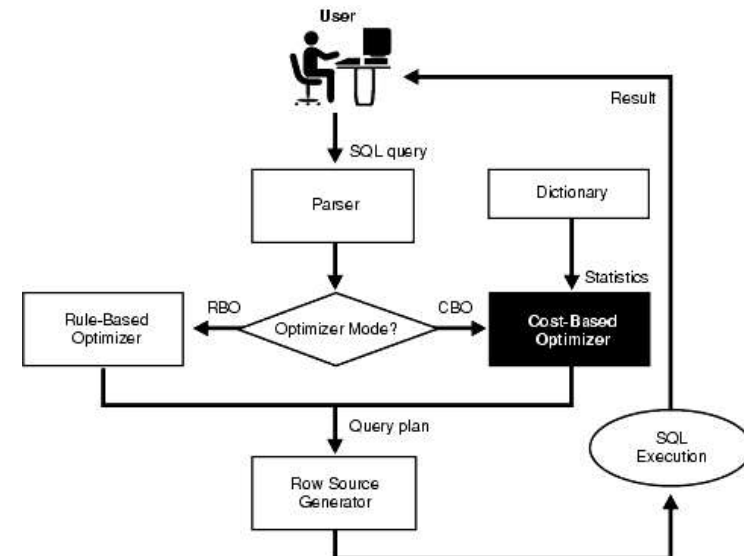
	PRENOM	NOM	EMAIL	VILLE
1	Flavie	Da costa	f.da.costa@example.com	Pomoy
2	Valentin	Vespasien	valentin@example.com	Buvilly
3	Gustave	Collin	gust@example.com	Marseille
4	Emilien	Camus	emilien@example.com	Toulouse
5	Firmin	Marais	firmin.marais@example.com	Lyon
6	Olivier	Riou	olive.de.lugagnac@example.com	Lugagnac
7	Lucas	Jung	lucas.jung@example.com	Coulgens
8	Maurice	Huet	maurice.villemareuil@example.com	Villemareuil
9	Manon	Durand	m.durand.s.e@example.com	Saint-Etienne
10	Joachim	Leon	joachim@example.com	Longwy-sur-le-Doubs
11	Muriel	Dupuis	muriel@example.com	Paris
12	Christiane	Riou	chritianelesabrets@example.com	Les Abrets
13	Jacinthe	Langlois	jacinthe.langlois@example.com	Lagny
14	Amaury	Payet	amaury@example.com	Avermes

Requête : *SELECT * FROM CLIENT WHERE nom = 'Riou'*

Que se passe t'il?

OPTIMISATION

Comment ça marche ?



OPTIMISATION

1. **Parseur** : analyse syntaxique et sémantique
2. **L'optimiseur détermine le chemin le plus efficace pour produire le résultat de la requête** : deux méthodes d'optimisation
 - **Cost Based Optimizer (CBO)** : elle se base sur des statistiques relatives aux tables pour déterminer le meilleur chemin d'accès. Ces statistiques concernent le volume des tables (nombre de lignes), le nombre de blocs alloués à la table, le nombre de valeurs distinctes de chaque colonne, la distribution de la clé, le nombre de valeurs différentes d'un index.
 - **Rule Based Optimizer (RBO)** : elle est basée sur un ensemble de règles internes (très peu utilisé).

L'optimiseur retourne un plan qui décrit la méthode la plus optimale pour l'exécution.

OPTIMISATION - OPTIMISEUR

Que fait l'optimiseur ?

1. Evaluer les expressions et les conditions
2. Transformer les déclarations SQL
3. Le choix de l'optimiseur : CBO ou RBO
4. Le choix des chemins d'accès
5. Le choix de la méthode de jointure

OPTIMISATION

3. Le **générateur de lignes** reçoit le plan optimal et génère **le plan d'exécution** de l'ordre SQL.
3. Le moteur SQL exécute le plan d'exécution et produit le résultat de la requête SQL

OPTIMISATION - OPTIMISEUR

1. Evaluer les expressions et les conditions

Certaines expressions sont changées si l'expression résultante peut être évaluée plus rapidement.

Exemple :

```
SELECT * FROM CLIENT WHERE nom IN ('Riou', 'Collin')  
→ WHERE nom = 'Riou' or nom = 'Collin'
```

where salaire > 24000/12 => salaire > 200

2. Transformer les déclarations SQL

A - *SELECT * FROM employees*

WHERE job_id = 'ST_CLERK' OR department_id = 50;

B - *SELECT * FROM employees*

WHERE job_id = 'ST_CLERK'

UNION – supprime les doublons contrairement à UNION ALL

*SELECT * FROM employees*

WHERE department_id = 50 ;

4. Le choix des chemins d'accès

Les chemins d'accès sont les chemins par lesquels les données sont lues dans la base de données. On peut distinguer

- **Full table Scan (FTS)** est un balayage complet de la table
- **Rowid Scan** permet d'accéder au fichier et au block qui contient la ligne. C'est le mode d'accès le plus rapide (accès direct).
- **Index Scan** : parcours de l'index pour récupérer les adresses des lignes.
- **Cluster Scan** permet de récupérer les données stockées dans une table créée sous forme de cluster d'index.
- **Hash Cluster Scan** permet de récupérer les données stockées dans un hash cluster

3. Le choix de la méthode d'optimisation

RBO (Rule Based Optimiser) méthode basée sur plusieurs règles. Elle ne tient pas en compte des données contenues dans les tables. Est obsolète mais reste disponible pour certaines applications anciennes.

CBO (Cost Based Optimizer) méthode basée sur les statistiques (nombres de lignes, nombre de nœuds et feuilles sur les index...) et les chemins d'accès disponibles sur les tables et les index. Le plan d'exécution choisi est celui qui a le coût le plus faible.

5. Le choix de la méthode de jointure

En fonction du coût, l'optimisateur va choisir la jointure la mieux adaptée.

On peut distinguer les jointures ci-dessous:

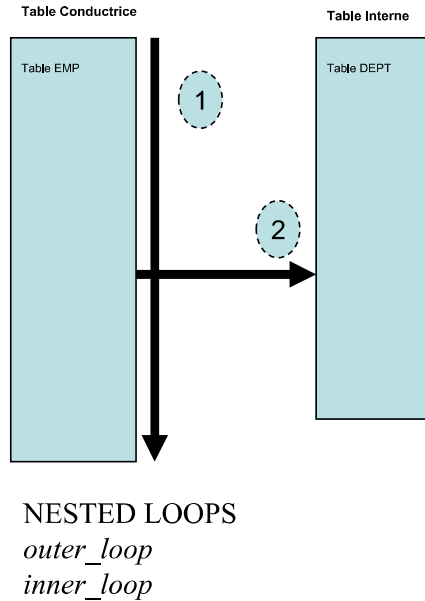
- **Nested Loop** : boucle imbriquée
- **Sort Merge Join** : Tri fusion
- **Hash Join** : Jointure par hachage

OPTIMISATION – Opérateur

Nested Loop :

La table extérieure est la table conductrice.

- 1- La boucle externe parcourt selon la méthode d'accès la plus adaptée les lignes de la table externe (outer loop).
- 2 - Pour chaque ligne de la table externe, on recherche toutes les lignes de la table interne (inner loop) à l'aide de la méthode d'accès optimal sur la base du critère de jointure.



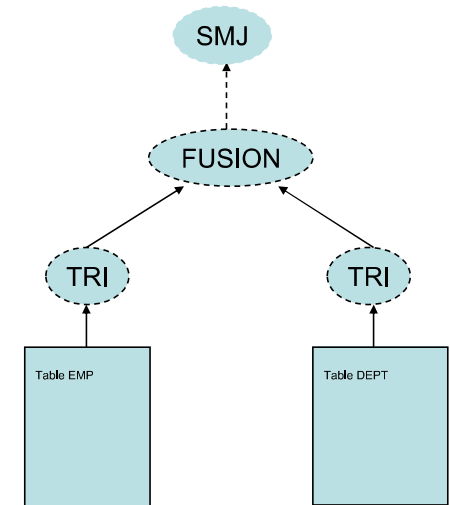
17

OPTIMISATION – Opérateur

Sort Merge Join :

Les deux tables sont triées selon le critère de jointure (ici numéro de département).

Les deux listes triées sont fusionnées.



18

OPTIMISATION – Opérateur

Hash Join :

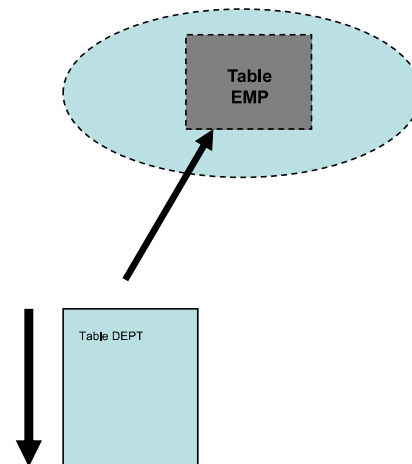
Table EMP est chargée en mémoire cache

Construire la table de hachage de EMP

Parcourir la table DEPT puis appliquer la fonction de hachage pour retrouver les lignes de la table EMP susceptible de joindre avec elle.

Pas de tri !

Jointure avec égalité uniquement



19

OPTIMISATION – Opérateur

Le choix de la méthode de jointure :

Coût de la Nested Loop :

$$\text{cost} = \text{outer access cost} + (\text{inner access cost} * \text{outer cardinality})$$

Coût du Sort Merge Join :

$$\text{cost} = \text{access cost of A} + \text{access cost of B} + (\text{sort cost of A} + \text{sort cost of B})$$

Coût du Hash Join :

$$\text{cost} = (\text{outer access cost} * \text{number of hash partitions of B}) + \text{inner access cost}$$

20

OPTIMISATION

Une fois les étapes de l'optimiseur effectuées (réécriture sql, choix du chemin d'accès...) un plan d'exécution est généré.

Le plan d'exécution regroupe :

- le chemin d'accès utilisé
- les opérations physiques
- l'ordre dans lequel les opérations seront effectuées.

LE PLAN D'EXECUTION

Pour générer un plan d'exécution on exécute les commandes ci-dessous

```
SQL> set timing on
```

```
SQL> set autotrace on
```

```
SQL> « requête sql »;
```

Ou

```
SQL> explain plan for « requête sql »;
```

```
SQL> select * from table(dbms_xplan.display());
```

La dernière option ne donne pas les ressources consommées

LE PLAN D'EXECUTION

Exemple : *set autotrace on;*

```
select * from HR.EMPLOYEES where employee_id = 104;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	68	1 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	68	1 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	EMP_EMP_ID_PK	1		0 (0)	00:00:01

Rows : estimation du nombre de lignes parcourues

Cost: sans unite; estimation du coût de l'operation.

Etape du plan d'exécution :

- Balayage de l'index unique → adresse de la ligne
- accès direct de la ligne dans la table
- retourne le resultat

LE PLAN D'EXECUTION -- optimisation

Ressources consommées : 1^{ière} execution

Statistiques

335 recursive calls -- opérations supplémentaires nécessaires pour l'exécution

0 db block gets -- nombre de blocs lus pour être modifiés

110 consistent gets -- lecture cohérente d'un block en mémoire

0 physical reads -- block lus sur disque : 0 → les données étaient déjà en mémoire

0 redo size -- données générées par la journalisation

1076 bytes sent via SQL*Net to client

380 bytes received via SQL*Net from client

2 SQL*Net roundtrips to/from client

5 sorts (memory) – tri effectué en mémoire

0 sorts (disk) – tri effectué sur disque en cas de besoin

1 rows processed

LE PLAN D'EXECUTION -- optimisation

Ressources consommées : 2nd execution

Statistiques

0 recursive calls
0 db block gets
2 consistent gets
0 physical reads
0 redo size
1076 bytes sent via SQL*Net to client
380 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

STATISTIQUES

OPTIMISATION

Nous avons vu ce que l'optimiseur fait pour nous.

Que devons nous faire pour améliorer les requêtes ?

Mise à jour des statistiques

Création des index

STATISTIQUES

! La méthode d'optimisation CBO (Cost Based Optimizer) qui est la méthode d'optimisation utilisée dans la majeure partie des cas utilise les statistiques → ces statistiques doivent être mises à jour en continue afin que l'optimiseur utilise les statistiques à jour.

Le package DBMS_STATS permet de générer les statistiques indispensables au bon fonctionnement de l'optimiseur.

Statistiques à jour → plan d'exécution optimal

STATISTIQUES

Plusieurs fonctions et vues existantes dans le package DBMS_STATS

➤ Afficher les statistiques

Plusieurs fonctions disponibles (GET_TABLE_STATS, GET_COLUMN_STATS, GET_INDEX_STATS, GET_SYSTEM_STATS); lourdes à utiliser car plusieurs paramètres.

Il existe des vues déjà définies (DBA_TAB_STATISTICS, DBA_IN_STATISTICS, DBA_TAB_COL_STATISTICS)

STATISTIQUES

Exemple : `select table_name, num_rows, blocks, stale_stats from DBA_TAB_STATISTICS where owner = 'HR'`

TABLE_NAME	NUM_ROWS	BLOCKS	STALE_STATS
1 REGIONS	4	5 NO	
2 COUNTRIES	25	(null) NO	
3 LOCATIONS	23	5 NO	
4 DEPARTMENTS	27	5 NO	
5 JOBS	19	5 NO	
6 EMPLOYEES	107	5 NO	
7 JOB_HISTORY	10	5 NO	

`select index_name, table_name, leaf_blocks, distinct_keys, num_rows from DBA_IND_STATISTICS where OWNER = 'HR'`

INDEX_NAME	TABLE_NAME	LEAF_BLOCKS	DISTINCT_KEYS	NUM_ROWS
LOC_CITY_IX	LOCATIONS	1	23	23
LOC_ID_FK	LOCATIONS	1	23	23
DEPT_LOCATION_IX	DEPARTMENTS	1	7	27
DEPT_ID_FK	DEPARTMENTS	1	27	27
JOB_ID_FK	JOBS	1	19	19
EMP_NAME_IX	EMPLOYEES	1	107	107
EMP_MANAGER_IX	EMPLOYEES	1	18	106
EMP_JOB_IX	EMPLOYEES	1	19	107
EMP_DEPARTMENT_IX	EMPLOYEES	1	11	106
EMP_EMP_ID_FK	EMPLOYEES	1	107	107

STATISTIQUES

➤ Collecter les statistiques : première utilité du package.

Plusieurs fonctions disponibles; GATHER_SCHEMA_STATS, GATHER_TABLE_STATS, GATHER_INDEX_STATS pour collecter respectivement les statistiques d'un schéma, une table ou encore d'un index.

Exemple : `EXEC DBMS_STATS.GATHER_SCHEMA_STATS('HR');`

TABLE_NAME	NUM_ROWS	BLOCKS	STALE_STATS	LAST_ANALYZED
1 REGIONS	4	5 NO		08/01/23
2 COUNTRIES	25	(null) NO		08/01/23
3 LOCATIONS	23	5 NO		08/01/23
4 DEPARTMENTS	27	5 NO		08/01/23
5 JOBS	19	5 NO		08/01/23
6 EMPLOYEES	107	5 NO		08/01/23
7 JOB_HISTORY	10	5 NO		08/01/23

STATISTIQUES

➤ Supprimer les statistiques

Quelques fonctions disponibles : DELETE_TABLE_STATS, DELETE_INDEX_STATS, DELETE_COLUMN_STATS pour la suppression respective des statistiques d'une table, d'un index et d'une colonne.

Exemple : `EXEC DBMS_STATS.DELETE_TABLE_STATS('HR', 'EMPLOYEES');`

TABLE_NAME	NUM_ROWS	BLOCKS	STALE_STATS	LAST_ANALYZED
1 REGIONS	4	5 NO		08/01/23
2 COUNTRIES	25	(null) NO		08/01/23
3 LOCATIONS	23	5 NO		08/01/23
4 DEPARTMENTS	27	5 NO		08/01/23
5 JOBS	19	5 NO		08/01/23
6 EMPLOYEES	(null)	(null) (null)	(null)	
7 JOB_HISTORY	10	5 NO		08/01/23

`EXEC DBMS_STATS.GATHER_TABLE_STATS('HR', 'EMPLOYEES');` pour recollecter les stats

INDEXATION

INDEXATION

Parcours séquentiel de la table

Si la table est volumineuse → ?

Possibilité de trouver directement l'information → ajout d'un index sur la table

OPTIMISATION

Considérons la table client ci-dessous :

	PRENOM	NOM	EMAIL	VILLE
1	Flavie	Da costa	f.da.costa@example.com	Pomoy
2	Valentin	Vespasien	valentin@example.com	Buvilly
3	Gustave	Collin	gust@example.com	Marseille
4	Emilien	Camus	emilien@example.com	Toulouse
5	Firmin	Marais	firmin.marais@example.com	Lyon
6	Olivier	Riou	olive.de.lugagnac@example.com	Lugagnac
7	Lucas	Jung	lucas.jung@example.com	Coulgens
8	Maurice	Huet	maurice.villemareuil@example.com	Villemareuil
9	Manon	Durand	m.durand.s.e@example.com	Saint-Etienne
10	Joachim	Leon	joachim@example.com	Longwy-sur-le-Doubs
11	Muriel	Dupuis	muriel@example.com	Paris
12	Christiane	Riou	chritianelesabrets@example.com	Les Abrets
13	Jacinthe	Langlois	jacinthe.langlois@example.com	Lagny
14	Amaury	Payet	amaury@example.com	Avermes

Requête : *SELECT * FROM CLIENT WHERE nom = 'Riou'*

Que se passe t'il?

INDEXATION

L'indexation est une technique de structure de données permettant d'extraire efficacement les enregistrements des fichiers de bases de données en fonction de certains attributs sur lesquels l'indexation est faite.

L'index est une notion connue

Exemple : Le sommaire d'un livre est un index.

En SGBD, c'est le même principe

INDEXATION

Exemple :

Clé (attribut) de recherche:

- * nom du client
- * Ville

Opérations :

- * Recherche des informations du client ayant le nom Riou
- * Liste des clients résidant dans la ville de Fontainebleau

INDEXATION

Considérons la collection de films suivant

Titre	Année	(autres colonnes)
Vertigo	1958	...
Brazil	1984	...
Twin Peaks	1990	...
Underground	1995	...
Easy Rider	1969	...
Psychose	1960	...
Greystoke	1984	...
Shining	1980	...
Annie Hall	1977	...
Jurassic Park	1992	...
Metropolis	1926	...
Manhattan	1979	...
Reservoir Dogs	1992	...
Impitoyable	1992	...
Casablanca	1942	...
Smoke	1995	...

INDEXATION – B-Tree

B-Tree pour Balanced Tree

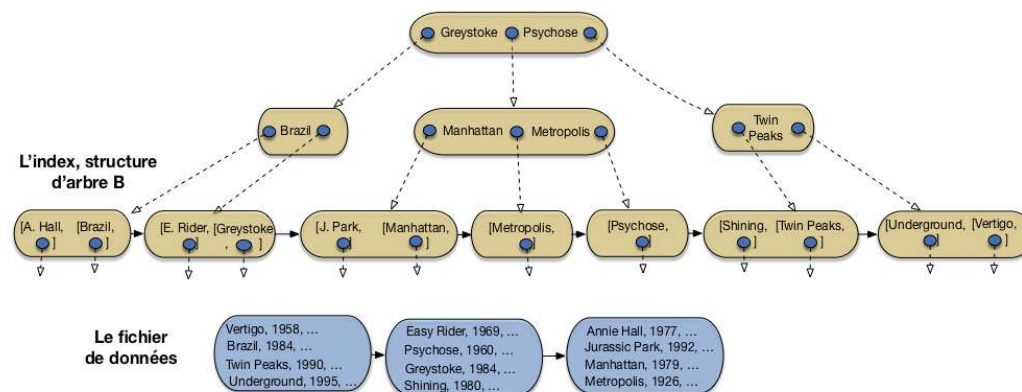
Structure arborescente constituée de plusieurs niveaux

l'indexation se fait au niveau bloc (et non au niveau d'un fichier)

Les feuilles contiennent les pointeurs vers les données.

INDEXATION – B-Tree

Exemple : index B-Tree de la collection de film précédent avec 4 entrées d'index max par bloc



Indexation – B-Tree

Comment est utilisé l'index ?

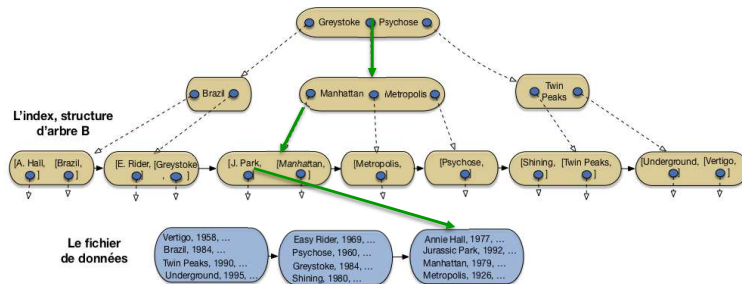
Select * from Film where titre ='Jurassic Park'

Sans index : parcours du fichier entier → lecture de 9 films inutile

Avec l'index

3 lectures (index)

1 lecture fichier



42

INDEXATION B-Tree

Syntaxe : **CREATE INDEX** *nom_index* **ON** *table_name*(*column1*, *column2*,...)

La clé d'un index peut avoir jusqu'à 32 colonnes max

CREATE INDEX filmTitre **ON** Film(Titre) : créé un index sur la table Film avec pour clé l'attribut « Titre ».

Le SGBD se charge de la maintenance des index.

Lors de l'ajout, la mise à jour ou la suppression d'un film dans la table, l'index est mis à jour automatiquement.

42

INDEXATION B-Tree

- Toutes les feuilles sont à la même profondeur => temps de recherche est toujours le même
- La recherche s'effectue par une traversée en profondeur (de la racine vers les feuilles).
- L'arbre est toujours équilibré
- Systématique utilisé par les SGBD car a de très bonne performance.

43

INDEXATION Bitmap

Index qui transforme les données en bit (1 ou 0) avant de les stocker.

Performant pour indexer les colonnes avec une faible cardinalité.

Exemple : colonne genre (femme, homme, autre)

Syntaxe :

CREATE BITMAP INDEX *index_name* **ON** *table_name*(*key*);

“Par défaut, Oracle créé un index B-TREE.

44

INDEXATION Bitmap

Considérons la table produit ci-dessous

N-PROD	COULEUR	TAILLE	PRIX
1	BLANC	PETIT	100
2	BLANC	MOYEN	200
3	BLANC	GRAND	300
4	BLEU	PETIT	100
5	BLEU	MOYEN	200
6	BLEU	GRAND	300
7	NOIR	PETIT	100
8	NOIR	MOYEN	200
9	NOIR	GRAND	300
10	JAUNE	PETIT	100
11	JAUNE	MOYEN	200
12	JAUNE	GRAND	300
13	BLANC	PETIT	100
14	BLANC	MOYEN	200
15	BLANC	GRAND	300

INDEXATION Bitmap

CREATE BITMAP INDEX idx_bp_couleur ON produit(couleur)

CREATE BITMAP INDEX idx_bp_taille ON produit(Taille)

--- idx_bp_taille

Rowid 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Petit 1 0 0 1

Moyen 0 1 0

Grand 0 0 1

--- idx_bp_couleur

Rowid 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Blanc 1 1 1 0 0 0 0 0 0 0 0 1 1

Bleu

Considérons la table produit ci-dessous

N-PROD	COULEUR	TAILLE	PRIX
1	BLANC	PETIT	100
2	BLANC	MOYEN	200
3	BLANC	GRAND	300
4	BLEU	PETIT	100
5	BLEU	MOYEN	200
6	BLEU	GRAND	300
7	NOIR	PETIT	100
8	NOIR	MOYEN	200
9	NOIR	GRAND	300
10	JAUNE	PETIT	100
11	JAUNE	MOYEN	200
12	JAUNE	GRAND	300
13	BLANC	PETIT	100
14	BLANC	MOYEN	200
15	BLANC	GRAND	300



INDEXATION Bitmap

select * from produit
where couleur = 'BLANC' and taille = 'PETIT';

(‘BLANC’ = 1,1,1,0,0,0,0,0,0,0,0,0,1,1,1)

AND

(‘PETIT’ = 1,0,0,1,0,0,1,0,0,1,0,0,1,0,0)

1,0,0,0,0,0,0,0,0,0,0,0,1,0,0

INDEXATION Bitmap

⇒ Performant pour les colonnes à faible cardinalité (très peu de valeurs distinctes)

⇒ À chaque ligne est associée une valeur booléenne : true ou false

⇒ Performant sur la combinaison de critères (and, or)

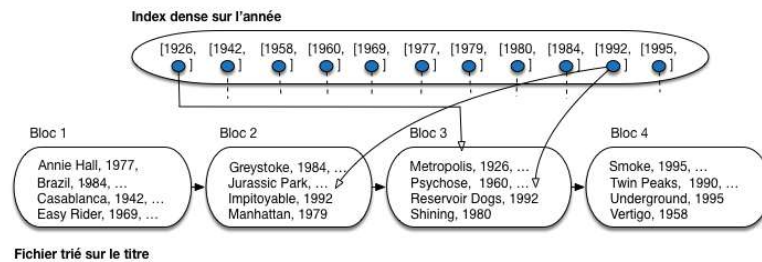
⇒ Peu volumineux

INDEXATION – Index dense

Un index dense est un index qui comporte un enregistrement pour chaque valeur de la clé de tri du fichier indexé.

Temps de recherche très court.

Volumineux



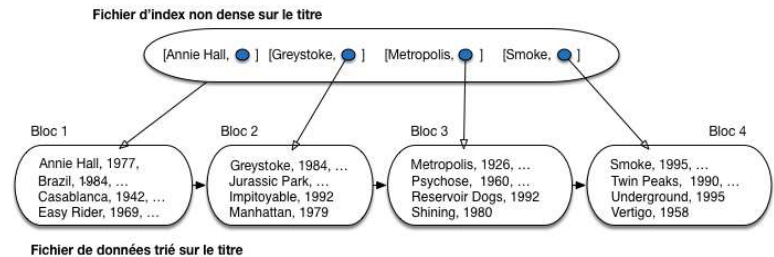
49

INDEXATION - index non dense

Un index non dense est un index qui contient certaines valeurs de la clé du tri du fichier indexé.

La recherche se fait par plage de valeur

Moins volumineux



50

INDEXATION – Index unique

Un index peut être unique ou non.

Le mot clé UNIQUE le rend unique.

Syntaxe : **CREATE UNIQUE INDEX** *index_name* **ON** *table_name*(*column1*, *column2*...)

La création d'un index unique empêche d'avoir des doublons dans la (les) colonne(s) clé(s)

Exemple : **CREATE UNIQUE INDEX** *ix_Film_Titre* **ON** *Film*(Titre) → on ne peut pas avoir deux films ayant le même titre dans la table *Film*

51

INDEXATION – Index unique

NB : lorsque qu'une clé primaire est créée sur une table, Oracle crée automatique un index UNIQUE sur la clé primaire.

De même lorsqu'une contrainte d'unicité est ajoutée sur un attribut de la table, un index unique est créé sur cet attribut.

52

INDEXATION – Index basé sur une fonction

La clé de l'index est le résultat d'une fonction appliquée sur un ou plusieurs attributs.

Syntaxe : `CREATE INDEX index_name ON table_name(expression);`

Expression : `function(column)`

Exemple : `CREATE INDEX ix_Film ON film(LOWER(Titre));`
`CREATE INDEX ix_Emp ON Employee(salaire+commission)`

INDEXATION

INDEX CONCATENE ; index sur plusieurs colonnes

`create index I_emp_ename_job on emp (ename, job);`

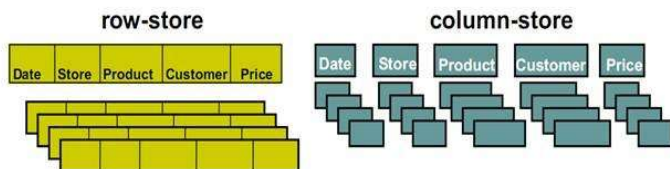
32 colonnes max.

La clause where doit porter sur (ename) ou (ename, job) pour que l'index soit utilisé

L'index ne sera pas utilisé si la where clause porte sur Job ou sur (job, ename)

INDEXATION - COLUMNSTORE

Structure qui stocke les données en colonne.



➤ Compression optimale

➤ Reduction I/O

INDEXATION

Index créé de façon implicite

➤ Primary key

➤ Contrainte unicité

OPTIMISATION

Exemple d'optimisation

```
SQL> select * from TestData where name='MAG500';
```

Statistiques

```
-----  
0 recursive calls  
0 db block gets  
40 consistent gets  
0 physical reads  
0 redo size  
540 bytes sent via SQL*Net to client  
380 bytes received via SQL*Net from client  
2 SQL*Net roundtrips to/from client  
0 sorts (memory)  
0 sorts (disk)  
1 rows processed
```

Exemple d'optimisation

```
SQL> select * from TestData where name='MAG500';
```

```
-----  
| Id | Operation          | Name      | Rows | Bytes | Cost (%CPU)| Time      |  
-----  
|  0 | SELECT STATEMENT    |           |     1 |    19 |      13   (8)| 00:00:01 |  
|*  1 |  TABLE ACCESS FULL | TESTDATA  |     1 |    19 |      13   (8)| 00:00:01 |  
-----
```

Predicate Information (identified by operation id):

```
-----  
1 - filter("NAME"='MAG500')
```

Exemple d'optimisation

On crée un index non unique sur la colonne name

```
SQL > create index IX_TestData on TestData(name)
```

On collecte les statistiques.

```
EXEC DBMS_STATS.GATHER_TABLE_STATS('SYSTEM','TESTDATA');
```

Exemple d'optimisation

```
SQL> select * from TestData where name='MAG500';
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	

0	SELECT STATEMENT		1	19	2 (0)	00:00:01	
1	TABLE ACCESS BY INDEX ROWID	TESTDATA	1	19	2 (0)	00:00:01	
* 2	INDEX RANGE SCAN	IX_TESTDATA	1		1 (0)	00:00:01	

Predicate Information (identified by operation id):

2 - access("NAME"='MAG500')

Exemple d'optimisation

Statistiques

```
-----
0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
540 bytes sent via SQL*Net to client
380 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

GATHER_PLAN_STATISTICS

Soit la requête suivante : *SELECT * FROM EMPLOYEES WHERE UPPER(FIRST_NAME) = 'Maggie';*

Resultat : 0 ligne

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	

0	SELECT STATEMENT		100	4900	23 (5)	00:00:01	
* 1	TABLE ACCESS FULL	EMPLOYEES	100	4900	23 (5)	00:00:01	

Statistiques

```
-----
0 recursive calls
0 db block gets
79 consistent gets
0 physical reads
0 redo size
638 bytes sent via SQL*Net to client
369 bytes received via SQL*Net from client
1 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
0 rows processed
```

HINT GATHER_PLAN_STATISTICS

L'option GATHER_PLAN_STATISTICS (Hint de statistique) nous permet d'avoir le nombre de lignes lues VS le nombre de lignes retournées.

SELECT /*+ GATHER_PLAN_STATISTICS */ * FROM EMPLOYEES WHERE UPPER(FIRST_NAME) = 'Maggie';	
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, NULL, 'ALLSTATS LAST'));	
Sorte de script x Résultat de requête x Résultat de requête 1 x Résultat de requête 2 x	
Toutes les lignes extraites : 18 en 0,01 secondes	
PLAN_TABLE_OUTPUT	
7	
8	
9 Id Operation Name Starts E-Rows A-Rows A-Time Buffers	
10	
11 * 1 TABLE ACCESS FULL EMPLOYEES 1 100 0 00:00:00.01 79	
12	

HINT for Join

Il est possible de proposer à l'optimiseur CBO des recommandations pour le choix d'un index ou d'un algorithme de jointure. Notamment lorsque vous maîtrisez bien la donnée et la structure de la base est bien connue.

HINT vous permet de faire ces recommandations.

HINTS for Join Operations

- USE_NL
- USE_MERGE
- USE_HASH

HINT

```
SQL> select /*+ USE_MERGE (emp dept) */ ename, dname from  
emp natural join dept;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		14	308	5 (20)	00:00:01
1	MERGE JOIN		14	308	5 (20)	00:00:01
2	TABLE ACCESS BY INDEX ROWID	DEPT	4	52	2 (0)	00:00:01
3	INDEX FULL SCAN	SYS_C004637	4		1 (0)	00:00:01
* 4	SORT JOIN		14	126	3 (34)	00:00:01
5	TABLE ACCESS FULL	EMP	14	126	2 (0)	00:00:01

HINT

```
SQL> select ename, dname from emp natural join dept ;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		14	308	3 (0)	00:00:01
1	NESTED LOOPS		14	308	3 (0)	00:00:01
2	TABLE ACCESS FULL	EMP	14	126	2 (0)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	DEPT	1	13	1 (0)	00:00:01
* 4	INDEX UNIQUE SCAN	SYS_C004637	1		0 (0)	00:00:01

Fin

Performance d'une mémoire

Deux métriques à prendre en compte :

- **Temps d'accès** : durée nécessaire pour lire une donnée dont l'adresse est connue (accès direct). De l'ordre de 10ms pour les disque magnétique et 0,1 ms pour les SSD.
- **Débit** : volume de données lues ou écrites par unité de temps dans le meilleur des cas.